M.Sc.(C.S.) - 10

VARDHAMAN MAHAVEER OPEN UNIVERSITY, KOTA

Operating System

Course Development Committee

Chairman

Prof. (Dr.) Naresh Dadhich
Vice Cahncellor
Vardhaman Mahaveer Open University Kota

Convener / Coordinator

Prof. (Dr.) D.S. Chauhan
Department of Mathematics
University of Rajasthan Jaipur

Member Secretary / Internal Coordinator

Sh. Rakesh Sharma
Assistant Professor (Computer Application)
V.M. Open University Kota

Members

1. Prof. (Dr.) Neeraj Bhargava
   M.D.S. University Ajmer
2. Dr. (Mrs.) Madhavi Sinha
   BITS Jaipur
3. Prof. (Ms.) Swati Chandey
   IIIM Jaipur
4. Prof. (Dr.) D.P. Sharma, Jaipur
5. Sh. Rajeev Shrivastava
   LBS College Jaipur

Editing and Course Writing

Editor

Prof. (Dr.) D.P. Sharma
Maharishi Arvind College of Mgt & IT Jaipur

Writers

1. Sh. Bright Keswani
   Gyan Vihar Univ Jaipur
2. Sh. Aman Jain
   Deepshikha College Jaipur
3. Ms. Shweta Sharma
   MIT Kota
4. Sh. Rajeev Shrivastava
   LBS College Jaipur
5. Sh. D.K. Gupta
   MIT Kota
6. Mr. Sanjay Verma
   MIT Kota

Academic and Administrative Arrangement

Prof. (Dr.) Naresh Dadhich	Prof.(Dr.) M.K. Ghadoliya	Yogendra Goyal
Vice Chancellor	Director	In Charge
Vardhaman Mahaveer Open University	Academic	Material Production & Distribution

Course Production

Yogendra Goyal
Assistant Production Officer
Vardhaman Mahaveer Open University

PRODUCTION : MARCH 2010

Printed and published on behalf of Registrar, VM Open University Kota.
Printed by The Diamond Printing Press, Jaipur, 500 Books

VARDHAMAN MAHAVEER OPEN UNIVERSITY, KOTA

Index

Operating System

UNIT 1 : SECURITY AND PROTECTION

Structure of the Unit

1.0 OBJECTIVES

Pallavi

After completing this unit you will learn :-

- About Security and protection method in Operating System.
- How security violation takes place due to parameter passing mechanisms
- Security in distributed environment, encryption system,
- Concepts of private,public keys, digital signatures and authentication as well as protection.

1.1 INTRODUCTION

The term 'Security' is defined variously in the computer literature, For instance, 'Trusted Computer System Evaluation Criteria ' defines it rather loosely and informally as follows:

"In general, secure systems will control, through use of specific security features" access to information that only properly authorized individuals or processes operating on their behalf will have access to read, write, create or delete".

Open Systems Interconnection (OSI) defines the elements of security in the following terms:

- **Confidentiality:** Ensuring that information is not accessed in an unauthorized manner (essentially controlling the Read operations).
- **Integrity:** Ensuring that information is not amended or deleted in an unauthorized manner (essentially controlling the Write operations).
- **Availability:** Ensuring that information is available to authorized users at the right time (essentially controlling the Read and Delete operations and ensuring fault recovery).

Generally security is concerned with the ability of the operating system to enforce control over the storage and transportation of data in and between the objects that the operating system supports.

In a multiuser environment, the concepts of security and protection are very important. User programs should not interfere with one another or with the operating system. As user programs execute in the computer memory, the first step is to identify and protect the memory allocated to each user program as well as the one that is allocated to the operating system.

1.2 SECURITY THREATS

The cost of hardware is falling at a rapid rate, millions of ordinary users and programmers have access to small or large computing equipment. With a trend towards networking, the user/programmer has access to data and programs at different remote locations. This has increased the threat to the security of computing environments.

Sharing and protection are requirements of any modern computing, but sharing gives rise to more possibility of security threats, or penetration, thus requiring higher protection.

When the Personal Computer (PC) was designed, it was intended strictly for individual use. This is the reason why MS-DOS was not very' strong in the security/protection areas.

In the PC environment, in the earlier days, the only way data could be protected was by locking the room physically where the PC and the floppy disks were kept. Today, as a number of PCs are being networked together for sharing of data and programs, a need has arisen to have better and stricter control over the protection aspects. This was the main motivation behind the development of a Network Operating System (NOS) such as Novell NetWare.

The major threats to security can be categorized as follows:

(i) Unauthorized use of service

(ii) Unauthorized disclosure of information

(iii) Unauthorized alteration or deletion of information

(iv) Unauthorized fabrication of information

(v) Denial of service to authorized

Security threats can arise from unintentional or deliberate reasons. Again, there may be casual or malicious attempts for penetration.

1.3 ATTACKS ON SECURITY

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

Authentication

Authentication means verification of access to the system resources.

Following could be some of the ways of penetration of the system in this regard.

(i) A user may guess or steal somebody else's password and then use it.

(ii) A user may use the vendor-supplied password which is expected to be used for the purpose of system

generation and maintenance by only the system administrators.

(iii) A user may find out the password by trial and error method. A program could also be written to assist this trial and error method.

Browsing

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

1.4 SECURITY VOILATION THROUGH PARAMETERS

When one program calls another program or function , it may require some parameter to be passed from the caller function to the call function

1.4.1 Trap Doors

Sometimes, software designers may want to modify their programs after their installation or even after they have gone in production. To assist them in this task, the programmers leave some secret entry points which do not require authorization to access certain objects. Essentially, they bypass certain validation checks. Only the software designers know how to make use of these shortcuts. These are called trap doors. At times, such shortcuts may be necessary for coping with emergency situations, but then these trap doors can also be abused by some others to penetrate into the system.

1.4.2 Invalid Parameters

Serious security violations can take place due to passing of invalid parameters or due to the failure to validate them properly .

1.4.3 Line Tapping

A special terminal can be used in this case to tap into the communications line and access, or even modify, confidential data. The security threat could be in the form of tapping, amendment or fabrication, once the intruder gets hold of the line. The intruder needs different hardware and techniques depending upon whether the attack is passive or active.

1.4.4 Electronic Data Capture

A penetrator can use active or passive wire-taps, or a mechanism to pick up the screen radiation and recognize what is displayed on the screen.

1.4.5 Lost Line

In networking environments, a line can get lost. In such a case, a sturdy operating system can log out a user and allow an access only after reestablishing the identity of the user. Some operating systems cannot do this. In such a case, the process created before losing the line just floats about, and hence, an intruder can gain control of this floating process and access the resources which were accessible by that process. Though this is difficult, it is certainly a possible way of an attack on security.

1.4.6 Improper Access Controls

In some operating systems, the system itself does not allow the planning of a meticulous, rigorous access control mechanism. On top of this, the system administrator may not plan his access controls properly.

This may lead to some users having far too many privileges and some others, very few. This situation obviously can potentially lead to unauthorized disclosure of information or denial of service. In either case, it leads to security violation.

1.4.7 Rough Software

A variety of software programs exist under this title. Computer virus is the most notorious of them all. This is a deliberately written program or a part of it, intended to create mischief. Such programs vary in terms of their complexity or damage they cause. Obviously, all of them require a deep knowledge of the operating system and the underlying hardware. They are, therefore, normally produced by very clever systems programmers. Various types of rough software are exists.

1.5 COMPUTER WORMS

A computer worm is a full program by itself. It is written in such a way that it spreads to their computers over a network, but while doing this, it consumes the network resources to a very large extent.

1.5.1 Origins

Worms are small programs which could by themselves spread to other computers. worm would execute on a machine if idle capacity was available on that machine otherwise it would continue to hunt for other machines 'in search of idleness'. This was the original purpose.

1.5.2 Mode of Operation

Usually, though not always, a computer worm does not harm any other program or data. It just spreads, thereby consuming large resources such as transmission capacity or disk storage, thereby denying services to legitimate users. A computer worm usually operates on a network. Each node on a network maintains a list of all other nodes on the network. It also maintains a 'mailing list' which contains the names and addresses of the reachable machines on the network. The worm gets an access into this list and using this, sends a copy of itself to all those addresses.

If a worm is more intelligent and less harmful, after reaching there, it checks at a new node whether its copy already exists, and if it does, it does not create another one. If it is dumb as well as malefactor, it copies itself to all the nodes on the mailing list regardless. Hence, if a node is on a mailing list of dozens or may be hundreds of other nodes in a large network, as many copies of the worm could exist on that node, sent by all those nodes.

As a result of this large continuous transfer over the network, a major portion of the network resources such as line capacity, disk capacity, network buffers, process tables are used up so that the network speed can reduce substantially.

1.5.3 The Internet Worm

Worm is normally not harmful to the existing programs and data, it can be extremely harmful to the organizations and governments operating over a network.

1.5.4 Safeguards against Worms

There are two major safeguards against computer worms.

Prevent its creation: This can be achieved by having very strong security and protection policies and mechanisms.

generation and maintenance by only the system administrators.

(iii) A user may find out the password by trial and error method. A program could also be written to assist this trial and error method.

Browsing

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

1.4 SECURITY VOILATION THROUGH PARAMETERS

When one program calls another program or function , it may require some parameter to be passed from the caller function to the call function

1.4.1 Trap Doors

Sometimes, software designers may want to modify their programs after their installation or even after they have gone in production. To assist them in this task, the programmers leave some secret entry points which do not require authorization to access certain objects. Essentially, they bypass certain validation checks. Only the software designers know how to make use of these shortcuts. These are called trap doors. At times, such shortcuts may be necessary for coping with emergency situations, but then these trap doors can also be abused by some others to penetrate into the system.

1.4.2 Invalid Parameters

Serious security violations can take place due to passing of invalid parameters or due to the failure to validate them properly .

1.4.3 Line Tapping

A special terminal can be used in this case to tap into the communications line and access, or even modify, confidential data. The security threat could be in the form of tapping, amendment or fabrication, once the intruder gets hold of the line. The intruder needs different hardware and techniques depending upon whether the attack is passive or active.

1.4.4 Electronic Data Capture

A penetrator can use active or passive wire-taps, or a mechanism to pick up the screen radiation and recognize what is displayed on the screen.

1.4.5 Lost Line

In networking environments, a line can get lost. In such a case, a sturdy operating system can log out a user and allow an access only after reestablishing the identity of the user. Some operating systems cannot do this. In such a case, the process created before losing the line just floats about, and hence, an intruder can gain control of this floating process and access the resources which were accessible by that process. Though this is difficult, it is certainly a possible way of an attack on security.

1.4.6 Improper Access Controls

In some operating systems, the system itself does not allow the planning of a meticulous, rigorous access control mechanism. On top of this, the system administrator may not plan his access controls properly.

This may lead to some users having far too many privileges and some others, very few. This situation obviously can potentially lead to unauthorized disclosure of information or denial of service. In either case, it leads to security violation.

1.4.7 Rough Software

A variety of software programs exist under this title. Computer virus is the most notorious of them all. This is a deliberately written program or a part of it, intended to create mischief. Such programs vary in terms of their complexity or damage they cause. Obviously, all of them require a deep knowledge of the operating system and the underlying hardware. They are, therefore, normally produced by very clever systems programmers. Various types of rough software are exists.

1.5 COMPUTER WORMS

A computer worm is a full program by itself. It is written in such a way that it spreads to their computers over a network, but while doing this, it consumes the network resources to a very large extent.

1.5.1 Origins

Worms are small programs which could by themselves spread to other computers. worm would execute on a machine if idle capacity was available on that machine otherwise it would continue to hunt for other machines 'in search of idleness'. This was the original purpose.

1.5.2 Mode of Operation

Usually, though not always, a computer worm does not harm any other program or data. It just spreads, thereby consuming large resources such as transmission capacity or disk storage, thereby denying services to legitimate users. A computer worm usually operates on a network. Each node on a network maintains a list of all other nodes on the network. It also maintains a 'mailing list' which contains the names and addresses of the reachable machines on the network. The worm gets an access into this list and using this, sends a copy of itself to all those addresses.

If a worm is more intelligent and less harmful, after reaching there, it checks at a new node whether its copy already exists, and if it does, it does not create another one. If it is dumb as well as malefactor, it copies itself to all the nodes on the mailing list regardless. Hence, if a node is on a mailing list of dozens or may be hundreds of other nodes in a large network, as many copies of the worm could exist on that node, sent by all those nodes.

As a result of this large continuous transfer over the network, a major portion of the network resources such as line capacity, disk capacity, network buffers, process tables are used up so that the network speed can reduce substantially.

1.5.3 The Internet Worm

Worm is normally not harmful to the existing programs and data, it can be extremely harmful to the organizations and governments operating over a network.

1.5.4 Safeguards against Worms

There are two major safeguards against computer worms.

Prevent its creation: This can be achieved by having very strong security and protection policies and mechanisms.

Prevent its spreading: This can be done by introducing, various checkpoints in the communications system. You can disallow the transfer of executable files over a network. But then, this may prove to be an operational hindrance. An option could be to force the user or an operator to "sanction" such a transfer. Many corporate gateways to public networks employ this technique.

1.6 COMPUTER VIRUS

A computer virus is written with a clear intention of infecting other programs. It is a part of a program which normally backs onto an otherwise valid useful program. In this regard, it differs from a worm, which is a complete program by itself. On the other hand, the computer virus normally cannot, operate independently. Another way a virus differs from a worm is that the virus normally causes direct harm to the system. It can corrupt the code as well as data whereas a worm only consumes system resources.

1.6.1 Types of Viruses

There are several types of computer viruses. We give below a list of types that have been encountered so far. As time goes on, new varieties are likely to be added to this list.

- Boot Sector Infectors
- Memory Resident Infectors
- File Specific Infectors
- Command Processor Infectors
- General Purpose Infectors.

1.7 SECURITY DESIGN PRINCIPELS

Public Design

The design of the security system should not be a secret. The designer should, in fact, assume that the penetrator will know about it. For instance, the penetrator may know the algorithms of cryptographic systems. However, security can still be maintained, because he may not know the keys.

Least Privilege

Every process should be given the least possible privileges that are necessary for its execution. For instance, a word processor program should be given access to only the file being manipulated and which is specified at the beginning. This principle helps in countering attacks such as Trojan horse. This means that each protection domain is normally very small but then switching between the domains may be needed more frequently.

Explicit Demand

No access rights should be granted to a process as a default. Each subject should have to demand the access rights explicitly. The only thing that the designer has to keep in mind is that in this case, a legitimate user can be denied access at times. But, this is less dangerous than granting of unauthorized access. Also, the denial of access is reported or detected, and therefore, can be corrected.

Continuous Verification

The access rights should be verified at every request from the subject. Checking for the access rights only at the beginning and not checking subsequently is a wrong policy. For instance, it is not sufficient to verify

access rights only at the time of opening a file. The verification must be made at every read/write request too. This will take care of a possibility of somebody changing the access rights after a file is opened. However, this continuous verification can degrade the system performance. Hence, it has to be done quite efficiently.

Simple Design

The design of the security system should be simple and uniform, so that it is not difficult to verify its correct implementation. Security has to be built in all the layers including the lowest ones. It has to be built in the heart of the system as an integral part of it. It cannot be an additional new feature.

User Acceptance

The design should be simple and easy to use to facilitate acceptance by the users. Users should not have to spend a lot of efforts merely to learn how to protect their files.

Multiple Conditions

Whenever possible, the system should be designed in such a fashion that the access depends on fulfilling more than one condition, e.g. the system could demand two passwords, or, in a cryptographic system, it should ask for two keys.

1.8 AUTHENTICATION

Authentication is a process of verifying whether a person is a genuine user or not. There are two types of authentication that are possible, and in fact, necessary. The first is the verification of users logging into a centralized system. The second is the authentication of computers that are required to cooperate in a network or distributed environment. We will consider both of these scenarios one by one.

Authentication in Centralized Environment

Authentication in this environment can be achieved in the following three ways.

- A secret, known only to that user
- Something possessed only by that user
- Some human

These three ways are considered one by one.

(a) Password

The password is the most commonly used scheme which is also easy to implement. The operating system associates a password along with the username of each user, and stores it after encryption in a system file. When a user wants to log onto the system, the operating system demands that the user keys in both his username and password. The operating system then encrypts this keyed in password using the same encryption technique and then matches it with the one stored in the

System file. It allows to login only after they tally. This technique is quite popular as it does not require any extra hardware. But then, it provides only limited protection.

The password scheme is very easy to implement, but it is just as easy to break. All you need to do is to know somebody else's password. In order to counter this threat, the designers of the password systems make use of a number of techniques

1.9 PROTECTION MECHANISMS

A number of mechanisms that are available to protect the system resources-hardware or software-will be studied.

1.9.1 Protection Framework

One of the main problems that any file system has to tackle is to protect the files from unauthorized users. Confidential information from a very sensitive file should not be accessible to any ordinary user for reading, let alone changing and deleting.

In some cases, it may be necessary to prevent a user, or users belonging to a certain group, from accessing a complete directory, i.e. all the files and directories underneath it. This function of protection is not treated as important in single-user systems such as CP/M or MS/DOS. However, in multiuser systems, this assumes enormous importance. In fact, like files, it may be necessary that certain devices are accessible only to certain users. The same thing may also be true about processes, databases or semaphores. The operating system has to have a generalized strategy to deal witl1 all of them. All such items are called objects, which need to be protected by giving certain access -rights to known 'subjects' who want to access them. A subject, in reality, could be and normally; a process created by either a user or the operating system.

1) Access Rights

For various objects, the operating system allows different 'Access Rights' for different subjects. For example, for a file, these access rights can be Own, Write, Append, Read, Execute. UNIX has only Read, Write and execute (rwx) access rights, For example, for a printer as a device, the access rights can be 'Write' or 'None' only.

2) Domains and Domain Switching

The operating system defines another concept called domain which is a combination of the objects and a set of different access rights for each of the objects. You could then associate these domains with a subject such as a user or a process created by him.

A user process executing in domain 0 has an access right to Read from or Write to file 0, Read from file 1 and Write onto the printer O. Domains 1, 2 and 3 are similarly defined. It will be noticed that domains 1 and 2 intersect. This means that an object can belong to two domains simultaneously. It also means that a user process executing in either domain 1 or domain 2.can write onto printer 1.

1.10 SUMMARY

Generally security is concerned with the ability of the operating system to enforce control over the storage and transportation of data in and between the objects that the operating system supports.

The major threats to security can be categorized as follows:

(i) Unauthorized use of service

(ii) Unauthorized disclosure of information

(iii) Unauthorized alteration or deletion of information

(iv) Unauthorized fabrication of information

(v) Denial of service to authorized

- A computer worm is a full program by itself. It is written in such a way that it spreads to their computers over a network, but while doing this, it consumes the network resources to a very large extent.
- A computer virus is written with a clear intention of infecting other programs. It is a part of a program which normally backs onto an otherwise valid useful program. In this regard, it differs from a worm, which is a complete program by itself.
- Authentication is a process of verifying whether a person is a genuine user or not.

1.11 QUESTIONS

1. What is the difference between worm and virus ? How they spread and prevented.
2. What do you mean by security and protection ?
3. Explain following.
   - Authentication
   - Trap House
   - Trojan Hourse
4. What is Virus ? how they infect the computer ?
5. What are private and public key ? How they are used in encryption ?
6. What is digital signature ? How they works ?
7. Discuss the different ways in which can be maintained indicating their merits and demerits

□□□□

UNIT – 2: PROCESS SYNCHRONIZATION

Structure of the Unit

2.0 INTRODUCTION

A cooperating process is one that can affect or be affected by other processes executing in the system. Cooperating processes may either directly share a logical address space (code ,data), or be allowed to share data only through files. The former case is achieved through the use of lightweight processes or threads, Concurrent access to shared data may result in data inconsistency.

2.1 OBJECTIVE

In this chapter, we discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

2.2. CONCURRENT PROCESSES - THREAD

A thread is a single sequence stream within in a process. Because threads have some of the properties of processes, they are sometimes called lightweight processes. In a process, threads allow multiple executions of streams. In many respect, threads are popular way to improve application through parallelism. The CPU switches rapidly back and forth among the threads giving illusion that the threads are running in parallel. Like a traditional process i.e., process with one thread, a thread can be in any of several states (Running, Blocked, Ready or Terminated). Each thread has its own stack. Since thread will generally call different procedures and thus a different execution history. This is why thread needs its own stack. An operating system that has thread facility, the basic unit of CPU utilization is a thread. A thread has or

consists of a program counter (PC), a register set, and a stack space. Threads are not independent of one other like processes as a result threads shares with other threads their code section, data section, OS resources also known as task, such as open files and signals.

Race conditions

In operating systems, processes that are working together share some common storage (main memory, file etc.) that each process can read and write. When two or more processes are reading or writing some shared data and the final result depends on who runs precisely when, are called race conditions. Concurrently executing threads that share data need to synchronize their operations and processing in order to avoid race condition on shared data. Only one 'customer' thread at a time should be allowed to examine and update the shared variable Race conditions are also possible in Operating Systems. If the ready queue is implemented as a linked list and if the ready queue is being manipulated during the handling of an interrupt, then interrupts must be disabled to prevent another interrupt before the first one completes. If interrupts are not disabled than the linked list could become corrupt

2.3 THE CRITICAL-SECTION PROBLEM

A Critical Section is a section of code that must never allow more than one process inside it at all times, for fear of Race Conditions. To protect a critical section from this, a protection mechanism must abide by three rules:

1. Mutual Exclusion - No two processes can co-exist in a critical section at once. In order to prove that a mechanism abides by this rule, you must assume that two processes are currently in the critical section and by backtracking their steps, find that this is an impossible state.

2. Progress - A process that is not currently in a critical section must never be allowed to block another from entering. In order to prove that a mechanism abides by this rule, you must show that the condition x_i ($x_i = true$ means the process i can enter the critical section) cannot be changed indefinitely to *false* by a process j.

3. Bounded Wait - A process's waiting time to enter the critical section is bounded by a finite time, that is that a process will never be locked out of a critical section indefinitely. In order to prove that a mechanism abides by this rule, you must assume a process is barred from entering the critical section and prove that it will eventually enter it.

2.4 MUTUAL EXCLUSION

A way of making sure that if one process is using a shared modifiable data, the other processes will be excluded from doing the same thing.

Formally, while one process executes the shared variable, all other processes desiring to do so at the same time moment should be kept waiting; when that process has finished executing the shared variable, one of the processes waiting; while that process has finished executing the shared variable, one of the processes waiting to do so should be allowed to proceed. In this fashion, each process executing the shared data (variables) excludes all others from doing so simultaneously. This is called Mutual Exclusion.

Note that mutual exclusion needs to be enforced only when processes access shared modifiable data - when processes are performing operations that do not conflict with one another they should be allowed to proceed concurrently.

Mutual Exclusion Conditions

If we could arrange matters such that no two processes were ever in their critical sections simultaneously, we could avoid race conditions. We need four conditions to hold to have a good solution for the critical section problem (mutual exclusion).

- No two processes may at the same moment inside their critical sections.
- No assumptions are made about relative speeds of processes or number of CPUs.
- No process should outside its critical section should block other processes.
- No process should wait arbitrary long to enter its critical section.

2.5 SEMAPHORES

A semaphore is a protected variable whose value can be accessed and altered only by the operations P and V and initialization operation.

Binary Semaphores can assume only the value 0 or the value 1 counting semaphores also called general semaphores can assume only nonnegative values.

The P (or wait or sleep or down) operation on semaphores S, written as P(S) or wait (S), operates as follows:

```
P(S):  IF  S > 0
          THEN  S := S - 1
          ELSE  (wait on S)
```

The V (or signal or wakeup or up) operation on semaphore S, written as V(S) or signal (S), operates as follows:

```
V(S):  IF (one or more process are waiting on S)
          THEN (let one of these processes proceed)
          ELSE  S := S +1
```

Operations P and V are done as single, indivisible, atomic action. It is guaranteed that once a semaphore operations has stared, no other process can access the semaphore until operation has completed. Mutual exclusion on the semaphore, S, is enforced within P(S) and V(S).

If several processes attempt a P(S) simultaneously, only process will be allowed to proceed. The other processes will be kept waiting, but the implementation of P and V guarantees that processes will not suffer indefinite postponement.

Semaphores solve the lost-wakeup problem.

Producer-Consumer Problem Using Semaphores

The Solution to producer-consumer problem uses three semaphores, namely, full, empty and mutex.

The semaphore 'full' is used for counting the number of slots in the buffer that are full. The 'empty' for counting the number of slots that are empty and semaphore 'mutex' to make sure that the producer and consumer do not access modifiable shared section of the buffer simultaneously.

Initialization

Set full buffer slots to 0.

i.e., semaphore Full = 0.

Set empty buffer slots to N.

i.e., semaphore empty = N.

For control access to critical section set mutex to 1.

i.e., semaphore mutex = 1.

```
Producer ( )
WHILE (true)
        produce-Item ( );
    P (empty);
    P (mutex);
    enter-Item ( )
    V (mutex)
    V (full);

Consumer ( )
WHILE (true)
    P (full)
    P (mutex);
    remove-Item ( );
    V (mutex);
    V (empty);
    consume-Item (Item)
```

2.6 CLASSIC PROBLEMS OF SYNCHRONIZATION

In this section, we present a number of different synchronization problems as examples for a large class of concurrency-control problems. These problems are used for testing nearly every newly proposed synchronization scheme. Semaphores are used for synchronization in our solutions.

The Bounded-Buffer Problem

The bounded-buffer problem is commonly used to illustrate the power of synchronization primitives. We present here a general structure of this scheme, without committing ourselves to any particular implementation. We assume that the pool consists of n buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers, respectively. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.

The structure of the producer process

```
do {
    // produce an item
  wait (empty);
  wait (mutex);
    // add the item to the buffer
  signal (mutex);
  signal (full);
} while (true);
```

The Readers- Writers Problem

A data object (such as a file or record) is to be shared among several concurrent processes. Some of these processes may want only to read the content of the shared object, whereas others may want to update (that is, to read and write) the shared object. We distinguish between these two types of processes by referring to those processes that are interested in only reading as readers, and to the rest as writers. Obviously, if two readers access the shared data object simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the shared object simultaneously, chaos may ensue.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared object. This synchronization problem is referred to as the readers-writers problem. Since it was originally stated, it has

The structure of the consumer process

```
do {
  wait (full);
```

```
    wait (mutex);
      // remove an item from buffer
    signal (mutex);
    signal (empty);

      // consume the removed item
  } while (true);
```

The structure of a writer process

```
    do {
    wait (wrt) ;

    //   writing is performed
    signal (wrt) ;
  } while (true)
```

been used to test nearly every new synchronization primitive. The readerswriters problem has several variations, all involving priorities. The simplest one, referred to as the first readers-writers problem, requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting. The second readers-writers problem requires that, once a writer is ready, that writer performs its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve. For this reason, other variants of the problem have been proposed. In this section, we present a solution to the first readers-writers problem. Refer to the Bibliographical Notes for relevant references on starvation-free solutions to the readers-writers problem.

In the solution to the first readers-writers problem, the reader processes share the following data structures:

```
semaphore mutex, wrt;
int readcount;
```

The semaphores mutex and w r t are initialized to 1; readcount is initialized to 0. The semaphore w r t is common to both the reader and writer processes.

The mutex semaphore is used to ensure mutual exclusion when the variable readcount is updated. The readcount variable keeps track of how many processes are currently reading the object. The semaphore wrt functions as a mutual-exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical sections.

if a writer is in the critical section and n readers are waiting, then one reader is queued on wrt , and n - 1 readers are queued on mutex. Also observe that, when a writer executes signal (wrt), we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

The structure of a reader process

```
do {
wait (mutex) ;
readcount ++ ;
if (readercount == 1) wait (wrt) ;
signal (mutex)
// reading is performed
wait (mutex) ;
readcount - - ;
if redacount == 0) signal (wrt) ;
signal (mutex) ;
} while (true)
```

The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a common circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks as shown in figure. When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.

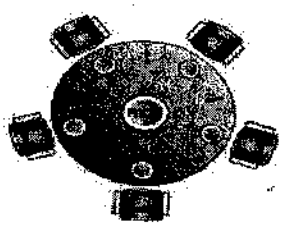


Figure : The situation of the dining philosophers

The structure of Philosopher *i*:

```
Do {
   wait ( chopstick[i] );
        wait ( chopStick[ (i + 1) % 5] );      // eat
        signal ( chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );     // think
} while (true) ;
```

The dining-philosophers problem is considered a classic synchronization problem, neither because of its practical importance nor because computer scientists dislike philosophers, but because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock- and starvation free manner.

One simple solution is to represent each chopstick by a semaphore. A philosopher tries to grab the chopstick by executing a wait operation on that semaphore; she releases her chopsticks by executing the signal operation on the appropriate semaphores. Thus, the shared data are

semaphore chopstick [5];

where all the elements of chopstick are initialized to 1.

Although this solution guarantees that no two neighbors are eating simultaneously,

it nevertheless must be rejected because it has the possibility of creating a deadlock. Suppose that all five philosophers become hungry simultaneously, and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Several possible remedies to the deadlock problem are listed next.

Here we present a solution to the dining-philosophers problem that ensures freedom from deadlocks.

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this she must pick them up in a critical section).
- Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick and then her right chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

Finally, any satisfactory solution to the dining-philosophers problem must guard against the possibility that one of the philosophers will starve to death. A deadlock free solution does not necessarily eliminate the possibility of starvation.

2.9 OS SYNCHRONIZATION

The operating system must provide the means to guard against timing errors. Several language constructs have been proposed to deal with these problems. Critical regions can be used to implement mutual-exclusion and arbitrary- synchronization problems safely and efficiently. Monitors provide the synchronization mechanism for sharing abstract data types. A condition variable provides a method for a monitor procedure to block its execution until it is signaled to continue.

Solaris 2 is an example of a modern operating system that implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing. It uses adaptive mutexes for

efficiency when protecting data from short code segments. Condition variables and readers writers locks are used when longer sections of code need access to data. Solaris uses turnstiles to order the list of threads waiting to acquire an adaptive mutex or readers-writers lock.

2.10 ATOMIC TRANSACTIONS

A transaction is a program unit that must be executed atomically, that is either all the operations associated with it are executed to completion, or none are performed. To ensure atomicity despite system failure, we can use a write ahead log. All updates are recorded on the log, which is kept in stable storage.

If a system crash occurs, the information in the log is used in restoring the state of the updated data items, which is accomplished with the use of the undo and redo operations. To reduce the overhead in searching the log after a system failure has occurred, we can use a checkpoint scheme.

When several transactions overlap their execution, the resulting execution may no longer be equivalent to an execution where these transactions executed atomically. To ensure correct execution, we must use a concurrency-control scheme to guarantee serialization. There are various different concurrency control schemes that ensure serialization by either delaying an operation or aborting the transaction that issued the operation. The most common ones are locking protocols and timestamp-ordering schemes

2.11 SUMMARY

- A thread is a single sequence stream within in a process. Because threads have some of the properties of processes, they are sometimes called lightweight processes.
- A Critical Section is a section of code that must never allow more than one process inside it at all times, for fear of Race Conditions.
- Mutual Exclusion is a way of making sure that if one process is using a shared modifiable data, the other processes will be excluded from doing the same thing.
- A transaction is a program unit that must be executed atomically, that is either all the operations associated with it are executed to completion, or none are performed. To ensure atomicity despite system failure, we can use a write ahead log.
- When several transactions overlap their execution, the resulting execution may no longer be equivalent to an execution where these transactions executed atomically. To ensure correct execution, we must use a concurrency-control scheme to guarantee serialization.

2.12 QUESTIONS

1. What is a critical Section ?
2. What is Process Synchronization ?
3. What is semaphore ?
4. What is Mutual Exclusion ?
5. Explain the concept of transaction atomicity.
6. What is the critical-section problem?

What is the meaning of the term **busy waiting?** What other kinds of waiting are there? Can busy waiting be avoided altogether? Explain your answer.

□□□□

UNIT 3 : DEADLOCK

Structure of the Unit

3.0 OBJECTIVES

After completing this unit we will be able to know

1. What is deadlock and characteristics of deadlock?
2. Methods of handling deadlock
3. How to prevention deadlock
4. How to avoid deadlock
5. Deadlock detection
6. Recovery from deadlock

3.1 INTRODUCTION

In a multithreaded environment, so many processes may need to use a finite number of resources. When a process need to use resources, if the resources are not available at that moment , the process enters a waiting state. Waiting processes may never again change state, because the resources they have requested are held by other waiting processes. This situation is called a deadlock.

3.2 SYSTEM MODEL

Deadlock occurs when a number of processes are waiting for an event, which can only be caused by another of the waiting processes. These are the essential requirements for a deadlock:

1. **Circular waiting** - There must be a set of processes where is waiting for a resource or signal from, is waiting for... and is waiting for.
2. **Non-sharable resources** - It is not possible to share the resources or signals, which are being waited for. If the resource can be shared, there is no reason to wait.

3. **No preemption** - The processes cannot be forced to give up the resources they are holding.

There are likewise three methods for handling deadlock situations:

1. **Prevention.** We can try to design a protocol, which ensures that deadlock never occurs.

2. **Recovery.** We can allow the system to enter a deadlock state and then recover.

3. **Ostrich method.** We can pretend that deadlocks will never occur and live happily in our ignorance. This is the method used by most operating systems. User programs are expected to behave properly. The system does not interfere. This is understandable: it is very hard to make general rules for every situation, which might arise.

3.3 DEADLOCK CHARACTERIZATION

In a deadlock, processes never finish executing and system resources are tried up, preventing other jobs from ever starting. Before we discuss the various methods for dealing with the deadlock problem, we shall describe features that characterize deadlocks.

Necessary Condition

A deadlock situation, Condition hold simultaneously in a system:

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

Hold and wait :- There must be exist a process holding at least one resource do other processes hold waiting to acquire additional resources.

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

Circular wait :- there exists a set {P0, P1... P0} 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 P0 is waiting for a resource that is held by P0.

Resource Allocation Graph

Deadlocks can be described by a resource allocation graph. The resource allocation graph is a directed graph consisting of vertices and directed edges. The vertex set is partitioned into two types, a subset representing processes and another subset representing resources. Pictorially, the resources are represented by rectangles with dots within, each dot representing an instance of the resource and circles represent processes.

A directed edge from a process to a resource (Pi -> Rj) signifies a request from a process Pi for an instance of the resource Rj and Pi is waiting for Rj. A directed edge from a resource to a process (Rj -> Pi) indicates that an instance of the resource Rj has been allotted to process Pi. Thus a resource allocation graph consists of vertices which include resources and processes and directed edges which consist of request edges and assignment edges. A request edge is introduced into the graph when a process requests for a resource. This edge is converted into an assignment edge when the resource is granted. When the process releases the resource, the assignment edge is deleted. Consider the following system:

There are 3 processes P1, P2 and P3.

Resources R1, R2, R3 and R4 have instances 1, 2, 1, and 3 respectively.

P1 is holding R2 and waiting for R1.

P2 is holding R1, R2 and is waiting for R3.

P3 is holding R3.

The resource allocation graph for a system in the above situation is as shown below

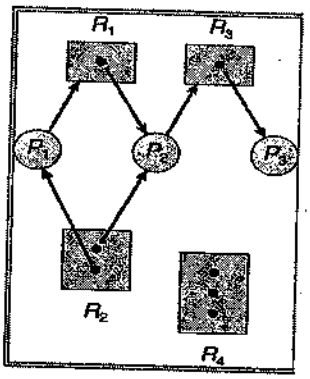


Figure : Resource allocation graph

If a resource allocation graph has no cycles (a closed loop in the direction of the edges), then the system is not in a state of deadlock. If on the other hand, there are cycles, then a deadlock may exist. If there are only single instances of each resource type, then a cycle in a resource allocation graph is a necessary and sufficient condition for the existence of a deadlock Here two cycles exist:

P1 -> R1 -> P2 -> R3 -> P3 -> R2-> P1

P2->R3 -> P3 -> R2 -> P2

Processes P0, P1 and P3 are deadlocked and are in a circular wait. P2 is waiting for R3 held by P3. P3 is waiting for P1 or P2 to release R2. So also P1 is waiting for P2 to release R1.

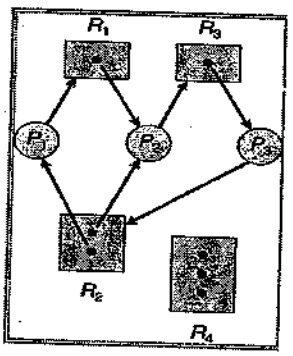


Figure : Resource allocation graph with a deadlock

If there are multiple instances of resources types, then a cycle does not necessarily imply a deadlock. Here a cycle is a necessary condition but not a sufficient condition for the existence of a deadlock . Here also there is a cycle:

P1 -> R1 -> P3 -> R2 -> P1

The cycle above does not imply a deadlock because an instance of R1 released by P2 could be assigned to P1 or an instance of R2 released by P4 could be assigned to P3 there-by breaking the cycle.

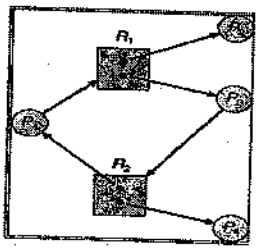


Figure : Resource allocation graph with a cycle but no deadlock

A set of processes is deadlocked if each process in the set is waiting for a resource/event that only another process in the set can provide.

There is no way for the operating system to detect every conceivable deadlock condition without expending large amounts of CPU time. Thus, the only way to recover from a deadlock is to kill the processes in question.

Responsibility for preventing deadlock situations is placed on the programmer. Fortunately, situations where deadlock can occur are infrequent; however, you should keep an eye out for them and try to work around them when they do occur.

3.4 METHODS FOR HANDLING DEADLOCKS

- Ensure that the system will never enter a deadlock state.
- Allow the system to enter a deadlock state and then recover.
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

Principally, we can deal with the deadlock problem in one of three ways:

1. We can use a protocol to prevent or avoid deadlock, ensuring that the system will never enter a deadlock state.
2. We can allow the system to enter a deadlock state, detect it, and recover.
3. We can ignore the problem altogether, and pretend that deadlock never occur in the system.

3.5 DEADLOCK PREVENTION

Prevention establishes system policies that make it impossible for deadlock to ever take place. The way to do this is to ensure that one of the four necessary conditions cannot be satisfied.

Prevention by Forbidding Mutual Exclusion Since mutual exclusion is necessary for correct operation in many situations, this cannot be used as a general-purpose prevention technique.

Prevention by Forbidding the Hold-and-Wait Condition There are at least two ways to do this:

- Force a process to request all its resources in advance; it is in the blocked state until it has all resources. This is wasteful because a process must tie up all its resources from the beginning, even if it won't need them for a long time. It also can lead to starvation for processes that need several popular resources; they may never all be available at the same time. Some batch systems used this approach, however.
- Force a process to release all its resources each time it requests a new one and then re-request all resources. For example, a process that has a tape drive and a disk may also need a printer. To get the printer, it must first release the tape drive and disk. Then it requests all 3. This can also result in long waits, poor resource utilization, and the possibility of starvation.

Prevention by Forbidding the No-Preemption Condition

There are several ways to do this; two are described below.

- If a process requests a resource and it is not available, the system will preempt the resources the process currently holds. Then it must wait for all resources; not just the newly requested one.
- Let high priority processes preempt resources from low priority processes. In this case the low priority process may just be terminated.

Preemption may work if the resources being preempted have states that can be easily saved (e.g. CPU) but as a general solution, this is not effective. For example, suppose the resource being held is a data

structure that requires mutual exclusion. If the data structure is preempted and given to another process, then a race condition will ensue.

Prevention by Forbidding the Circular Wait Condition

A simplistic approach is to permit a process to own only one resource at a time. This makes programming very difficult. A more common solution is based on resource ordering.

* Number all resource types and require processes to request resources in ascending order.
* Now, although processes may block on a resource request, it is impossible for a circular wait to develop. Consequently, there will never be deadlock.
* Proof:
* Suppose process Pi requests resource Rn. This means that any other resource held by Pi has a number less than n.
* Now suppose that Rn is assigned to some other process Pj.
* Pi will block on Rn, but it cannot be deadlocked, because Pj will never request a process held by Pi. (Pj can only request resources with numbers greater than n, since it already holds Rn.)

3.6 DEADLOCK AVOIDANCE

Whereas deadlock prevention is a global approach that prevents deadlocks by establishing system-wide policies, deadlock avoidance avoids deadlocks by using local strategies that consider each resource request on a case-by-case basis. Notice that both prevention and avoidance maintain a deadlock-free system (although at a cost).

* Deadlock avoidance is based on the concept of a safe state, which is defined as follows:

  A state is safe if
* It is not already deadlocked.
* There is some ordering of resource allocation that will allow all active processes to complete execution without deadlocking. This is called a safe sequence.
* A state that is not safe is unsafe. This is not the same as deadlock.
* To prove that P1... Pn is a safe sequence show that there are enough free resources for P1 to f inish. Now show that after P1 releases its resources, P2 will be able to acquire enough resources to finish, and that when it finishes P3 will be able to get enough resources to finish, and so on until you show that Pn is able to finish.

• The Banker's Algorithm can be used to determine if a safe state exists. It is used as follows:

* Process requests resource.
* Assume request is granted; use Banker's Algorithm to determine if resulting state is safe; i.e., if a safe sequence exists.
* If the state is safe, then grant the resource request.
* Otherwise, block the process, which requested the resource until a time when the request can be granted safely.
* Note that when a new process is created, any initial resource requests it makes must be treated the same way.

• Problems with Banker's Algorithm:

• It must be run every time a resource request is made.

- It requires processes to state maximum resource requirements in advance.
  It assumes no resources will disappear from the system. For example, if a printer suddenly goes offline a currently safe state may suddenly become unsafe

3.7 DEADLOCK DETECTION AND RECOVERY

Detecting Deadlocks with single unit resources

Consider the case in which there is only one instance of each resource.

- So a request can be satisfied by only one specific resource.
- In this case the 4 necessary conditions for deadlock are also sufficient.
- Remember we are making an assumption (single unit resources) that is often invalid. For example, many systems have several printers and a request is given for "a printer" not a specific printer. Similarly, one can have many tape drives.
- So the problem comes down to finding a directed cycle in the resource allocation graph. Why?

  Answer: Because the other three conditions are either satisfied by the system we are studying or are not in which case deadlock is not a question. That is, conditions 1, 2, 3 are conditions on the system in general not on what is happening right now.

To find a directed cycle in a directed graph is not hard. The algorithm is in the book. The idea is simple.

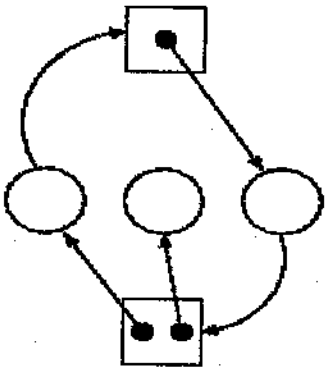
1. For each node in the graph do a depth first traversal (hoping the graph is a DAG (directed acyclic graph), building a list as you go down the DAG
2. If you ever find the same node twice on your list, you have found a directed cycle and the graph is not a DAG and deadlock exists among the processes in your current list.
3. If you never find the same node twice, the graph is a DAG and no deadlock occurs.
4. The searches are finite since the list size is bounded by the number of nodes.

Detecting Deadlocks with multiple unit resources

- The figure on the right shows a resource allocation graph with multiple unit resources.
- Each unit is represented by a dot in the box.
- Request edges are drawn to the box since they represent a request for any dot in the box.
- Allocation edges are drawn from the dot to represent that this unit of the resource has been assigned (but all units of a resource are equivalent and the choice of which one to assign is arbitrary)
- Note that there is a directed cycle in black, but there is no deadlock. Indeed the middle process might finish, erasing the magenta arc and permitting the blue dot to satisfy the rightmost process.
- The book gives an algorithm for detecting deadlocks in this more general setting. The idea is as follows.

1. look fora process that might be able to terminate (i.e., all its request arcs can be satisfied).
2. If one is found pretend that it does terminate (erase all its arcs), and repeat step 1.
3. If any processes remain, they are deadlocked.

- We will soon do in detail an algorithm (the Banker's algorithm) that has some of this flavor.



Detection-Algorithm Usage

Detection algorithms need to be executed to detect a deadlock.

The frequency and time when we run such algorithm is dependent on how often we assume deadlocks occur and how many processes they may effect.

If deadlocks may happen often, we run the detection often. If it affects many processes, we may decide to run it often so that less processes are affected by the deadlock.

We could run the algorithm on every resource request, but considering that deadlocks are rare, it is not very efficient use of resources. We could run the algorithm from time to time, say every hour, or when CPU utilization crosses some threshold, or at random times during the system execution lifetime.

Deadlock Recovery

We can recover from a deadlock via two approaches: we either kill the processes (that releases all resources for killed process) or take away resources.

Process Termination

When recovering from a deadlock via process termination, we have two approaches. We can terminate all processes involved in a deadlock, or terminate them one by one until the deadlock disappears.

Killing all processes is costly (since some processes may have been doing something important for a long time) and will need to be re-executed again.

Killing a process at a time until deadlock is resolved is also costly, since we must rerun deadlock detection algorithm every time we terminate a process to make sure we got rid of the deadlock.

Also, some priority must be considered when terminating processes, since we don't want to kill an important process when less important processes are available. Priority might also include things like how many resources are being held by that

process, or how long has it executed, or how long it has to go before it completes, or how many resources it needs to complete its job, etc.

Resource Preemption

This approach takes resources from waiting processes and gives them to other processes. Obviously, the victim process cannot continue regularly, and we have a choice of how to handle it. We can either terminate that process, or roll it back to some previous state so that it can request the resources again.

Again, there are many factors that determine which process we choose as the victim.

3.10 SUMMARY

* A deadlock state occurs when two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes: Principally, there are three methods for dealing with deadlocks:
* Use some protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlock state.
* Allow the system to enter deadlock state, detect it, and then recover.
* Ignore the problem all together, and pretend that deadlocks never occur in the system. This solution is the one used by most operating systems, including UNIX.
* A deadlock situation may occur if and only if four necessary conditions hold simultaneously in the system: mutual exclusion, hold and wait, no preemption, and circular wait. To prevent deadlocks, we ensure that at least one of the necessary conditions never holds.

3.11 QUESTIONS

* What is Deadlock?
* What are various method of Deadlock prevention?
* Explain Deadlock Detection process?
* How to recover from Deadlock?

UNIT 4 : SHELL PROGRAMMING

Structure of the Unit

4.0 OBJECTIVES

After completing this unit you will be able to explain

- What is shell ,standard streams in UNIX Operating?
- Aliases , Predefine and user defined variables
- Redirection ,pipes and filters , sorting
- Regular expiration , wild card and operators, grep command

Shell script writing using sed etc.

4.1 INTRODUCTION - SHELL AS HUMAN INTERFACE OF THE SYSTEM

The shell is the utility that processes users requests. When user type in a command at any terminal, the shell interprets the command and runs the requested program. UNIX supports multiple users and multiple tasks, and different users can invoke the same utility programs, including the shell. The shell uses standard syntax for all commands. There are many different shell programs available, such as:

- Bourne shell aka **sh**, the original shell used in UNIX systems and in UNIX related environment.
- Bourne Again shell aka **bash**, the standard GNU shell, intuitive and flexible. Probably most advis able for beginning users while being at the same time powerful tool for the advanced and profes sional user.
- C shell aka **csh**, the syntax of this shell resembles that of the C programming language.

- Turbo C shell aka **tcsh**, a superset of the common C shell; enhancing user-friendliness and speed.
- Korn shell aka **ksh**.

4.2 LOGIN SHELL VERIFICATION AND RELATIONSHIP BETWEEN VARIOUS SHELLS IN LINUX.

4.2.1 Usernames

Every person who uses a UNIX computer should have an account. An account is identified by a username. Traditionally, each account also has a secret password associated with it to prevent unauthorized use. Usernames are sometimes called account names. You need to know both your username and your password to log into a UNIX system. For example, Rama has an account on his college computer system. His username is rahul , His password is "sharma" When he wants to log into the college computer system, he types:

login: rahul

password: sharma

The username is an identifier: it tells the computer who you are. The password is an authenticator: you use it to prove to the operating system that you are who you claim to be Standard UNIX usernames may be between one and eight characters long. Within a single UNIX computer, usernames must be unique: no two users can have the same one.

A single person can have more than one UNIX account on the same computer. In this case, each account would have its own username. A username can be any sequence of characters you want (with some exceptions), and does not necessarily correspond to a real person's name. Your username identifies you to UNIX the same way your first name identifies you to your friends. When you log into the UNIX system, you tell it your username the same way you might say, "Hello, this is Sabrina," when you pick up the telephone.

4.2.2 password

Passwords are one form of user authentication. This means the system relies on something you know to validate who is logging onto the server. This works based on the idea of each user having a unique login, and a secret password that only they know.

Telling the computer your password is the way that you prove to the computer that you are you. In classical security parlance, your password is what the computer uses to authenticate your identity (two words that have a lot of significance to security gurus, but generally mean the same thing that they do to ordinary people).

When you log in, you tell the computer who you are by typing your username at the login prompt. You then type your password (in response to the password prompt) to prove that you are who you claim to be. For example:

login: kamal

password: kant

As we mentioned above, UNIX does not display your password when you type it. If the password that you supply with your username corresponds to the one on file, UNIX logs you in and gives you full access to all of your files, commands, and devices. If either the password or the username does not match, UNIX does not log you in.

On some versions of UNIX, if somebody tries to log into your account and supplies an invalid

password several times in succession, your account will be locked. Only the system administrator can unlock a locked account. Locking has two functions:

1. It protects the system from someone who persists in trying to guess a password; before they can guess the correct password, the account is shut down.

2. It notifies you that someone has been trying to break into your account. If you find yourself locked out of your account, you should contact your system administrator and get your password changed to something new. Don't change your password back to what it was before you were locked out.

Changing Your Password

You can change your password with the UNIX passwd command. passwd first asks you to type your old password, then asks for a new one. By asking you to type your old password first, passwd prevents somebody from walking up to a terminal that you left yourself logged into and then changing your password without your knowledge.

UNIX makes you type the password twice when you change it:

% passwd

Changing password for kamal

Old password:kant

New password: hero

Retype new password: hero

%

If the two passwords you type don't match, your password remains unchanged. This is a safety precaution: if you made a mistake typing the new password and UNIX only asked you once, then your password could be changed to some new value and you would have no way of knowing that value.

Verifying Your New Password

After you have changed your password, try logging into your account with the new password to make sure that you've entered the new password properly. Ideally, you should do this without logging out, so you will have some recourse if you did not change your password properly. This is especially crucial if you are logged in as root and you have just changed the root password.

4.3 RELATIONSHIP BETWEEN SHELLS AND KERNEL IN LINUX

Kernel

- Kernel is the heart of the Linux/Unix operating system.
- Kernel is the collection of system programs mostly written in C.
- Kernel is part of the operating system that directly communicate with the hardware.
- Kernel takes care about all the lower-level (hardware level) tasks.

A modern computer system consists of one or more processors, some main memory, hard drives, printers, a keyboard, a display and many more input/output devices. Kernel keeps track of all these components and use them correctly.

Kernel allocates processors, memories, and I/O devices among the various programs competing for them.

Shell

- Shell is the interface between the user and the kernel.
- The user doesn't have any information about the kernel functions because user communicate with the "shell" and communicate with the kernel.
- The shell is represented by /bin/sh.
- Shell is a program that reads our commands and gets our work done.

Shell watches everything users do.

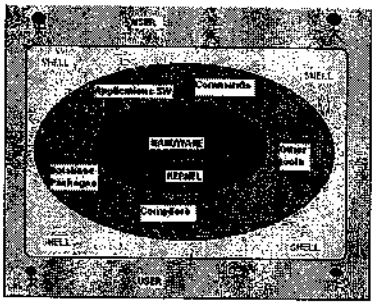


FIG 1: Kernel-shell Relationship.

- Users can communicate with kernel through shell.
- Shell starts runnig when user logs in and wait for the input from the user. When user enters the command,
- the shell analize the command and its associated arguments before send them to the kernel. The kernel,then, executes the command and gives the control back to shell.
- The shell plays a role of interpreter. It interprets the command entered by the user so that kernel can understand.
- There is only one kernel for Linux operating system.
- The kernel is the part of the operating system that is loaded into the computer memory when the system is booted.

4.4 STANDARD STREAMS

In most operating systems predating Unix, programs had to explicitly connect to the appropriate input and output data. On many of those systems, this could be an intimidating programming challenge created by OS-specific intricacies such as obtaining control environment settings, accessing a local file table, determining the intended data set, and handling the correct case of a card reader, magnetic tape drive, disk drive, or interactive terminal.

Unix provided several groundbreaking advances, one of which was to provide abstract devices: it removed the need for a program to know or care what kind of device it was communicating with. Older operating systems forced upon the programmer a record structure and, frequently non-orthogonal data semantics and device control. Unix eliminated this complexity with the concept of a data stream: an ordered sequence of data bytes which can be read until the end of file. A program may also write bytes as desired and need not (and can't easily) declare how many there will be, or how they will be grouped. In mathematics, orthogonal is synonymous with perpendicular when used as a simple adjective that is not part of any longer phrase with a standard definition. ...

Another Unix breakthrough was to automatically associate input and output by default—the program (and programmer) did absolutely nothing to establish input and output for a typical input-process-output program (unless the program chose a different paradigm). In contrast, previous operating systems usually required some—often complex—job control language to establish connections, or the equivalent burden had to be orchestrated by the program. Job Control Language (JCL) is a scripting language used on IBM mainframe operating systems to instruct the Job Entry Subsystem on how to run a batch program or start a subsystem.

Since Unix provided standard streams, the Unix C runtime environment was obligated to support it as well. As a result, most C runtime environments (and C's descendants), regardless of the operating system, provide equivalent functionality.

Standard input (stdin)

Standard input is data (often text) going into a program. The program requests data transfers by use of the read operation. Not all programs require input. For example, the dir or ls program (which displays file names contained in a directory) performs its operation without any stream data input.

Unless redirected, input is expected from the text terminal which started the program. A typical text terminal produces input and displays output and errors A text terminal or often just terminal (sometimes text console) is a serial computer interface for text entry and display.

The file descriptor for standard input is 0 (zero); the corresponding <stdio.h> variable is FILE* stdin; similarly, the <iostream> variable is std::cin. The term file descriptor is generally used in POSIX operating systems.

Standard output (stdout)

Standard output is the stream where a program writes its output data. The program requests data transfer with the write operation. Not all programs generate output. For example the file rename command (variously called mv, move, ren) is silent on success.

Unless redirected, standard output is the text terminal which initiated the program. A typical text terminal produces input and displays output and errors A text terminal or often just terminal (sometimes text console) is a serial computer interface for text entry and display.

The file descriptor for standard output is 1 (one); the corresponding <stdio.h> variable is FILE* stdout; similarly, the <iostream> variable is std::cout. The term file descriptor is generally used in POSIX operating systems.

Standard error (stderr)

Standard error is another output stream typically used by programs to output error messages or diagnostics. It is a stream independent of standard output and can be redirected separately. The usual destination is the text terminal which started the program to provide the best chance of being seen even if standard output is redirected (so not readily observed). For example, output of a program in a pipeline is redirected to input of the next program, but errors from each program still go directly to the text terminal. This page is a candidate for speedy deletion. A typical text terminal produces input and displays output and errors A text terminal or often just terminal (sometimes text console) is a serial computer interface for text entry and display.

It is acceptable—and normal—for standard output and standard error to be directed to the same destination, such as the text terminal. Messages appear in the same order as the program writes them, unless buffering is involved. (For example, a common situation is when the standard error stream is unbuffered but the standard output stream is line-buffered; in this case, text written to standard error later may

appear on the terminal earlier, if the standard output stream's buffer is not yet full.)

4.5 REDIRECTION

Mostly all command gives output on screen or take input from keyboard, but in Linux (and in other OSs also) it's possible to send output to file or to read input from file.

For e.g.

$ ls command gives output to screen; to send output to file of ls command give command

$ ls > filename

It means put output of ls command to filename.

There are three main redirection symbols >,>>,<

(1) > Redirector Symbol

Syntax:

Linux-command > filename

To output Linux-commands result (output of command or shell script) to file. Note that if file already exist, it will be overwritten else new file is created. For e.g. To send output of ls command give

$ ls > myfiles

Now if '**myfiles**' file exist in your current directory it will be overwritten without any type of warning.

(2) >> Redirector Symbol

Syntax:

Linux-command >> filename

To output Linux-commands result (output of command or shell script) to END of file. Note that if file exist , it will be opened and new information/data will be written to END of file, without losing previous information/data, And if file is not exist, then new file is created. For e.g. To send output of date command to already exist file give command

$ date >> myfiles

(3) < Redirector Symbol

Syntax:

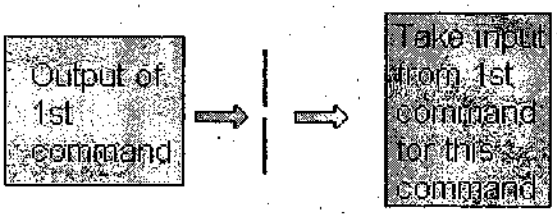
Linux-command < filename

To take input to Linux-command from file instead of key-board. For e.g. To take input for cat command give

$ cat < myfiles

4.6 PIPES

A pipe is a way to connect the output of one program to the input of another program without any temporary file.



Pipe Defined as:

"A pipe is nothing but a temporary storage place where the output of one command is stored and then passed as the input for second command. Pipes are used to run more than two commands (Multiple commands) from same command line."

Syntax:

Examles:

command1 | command2

Command using Pipes	Meaning or Use of Pipes
$ ls | more	Output of ls command is given as input to more command So that output is printed one screen full page at a time.
$ who | sort	Output of who command is given as input to sort command So that it will print sorted list of users
$ who | sort > user_list	Same as above except output of sort is send to (redirected) user_list file
$ who | wc -l	Output of who command is given as input to wc command So that it will number of user who logon to system
$ ls -l | wc -l	Output of ls command is given as input to wc command So that it will print number of files in current directory.
$ who | grep raju	Output of who command is given as input to grep command So that it will print if particular user name if he is logon or nothing is printed (To see particular user is logon or not)

4.7 FILTERS

A filter is a small and specialized program in Unix-like operating systems that transforms plain text data in some meaningful way and that can be used together with other filters and pipes to form a series of operations that produces highly specific results.

Simple Filters	Advance Filters
More	*grep*
less	*sed*
pg	*awk*
head	
Tail	
Wc	
tee	
tr	
sort	
cut	
uniq	

1. **More :-** More is a filter to display the paging through text page by page

Syntax: $ more <filename>

f - To move next page

b - To move previous page

Spacebar -- To scroll the page

Enter Key -- To scroll the page line by line

q - Quit

Ex: $ cat Imorz | more

2. **less :-** Less is a program similar to more (1), but which allows backward movement in the file as well as forward movement. Also, less does not have to read the entire input file before starting, so with large input files it starts up faster than text editors like vi

Syntax: $ less <filename>

f - To move next page

b - To move previous page

Spacebar -- To scroll the page

Enter Key -- To scroll the page line by line

q -- Quit

Ex: $ cat file1 | less

Note: This command works on Linux only

3. **Pg :-** It displays page by page

Syntax: $ pg <filename>

f - To move next page

b - To move previous page

Spacebar - To scroll the page

Enter Key – To scroll the page line by line

q – Quit

Ex: cat venkat | pg

Note: This command don't works in Linux

4. **head :-** It displays the first 10 lines of the file (page) – default

Syntax: $ head <filename>

Ex: $ head -15 venkat

Ex: $ cat <filename> | head 20 - It display first 20 lines

5. **tail :-** It displays last 10 records of the file

Syntax: tail <filename>

: tail -20 <filename>

Ex: cat <filename> | tail -20

Ex: tail +25 <filename> - It displays last 25 records

In-between records

Records from 15 to 20

Ex: $ head -10 <filename> | tail -5 (Records 15 to 20)

Ex: $ tail +30 <filename> | head -5 (Records 30 to 35)

If u want to redirect to permanent

Ex: $ head -20 <filename> |tail -5 > file1 (file1 is newfile)

Ex: $ tail +30 <filename> |head -5 > file1 (file1 is newfile)

6. **wc :-** This filter is used to word count

Syntax: $ wc <filename>

Ex: $ wc kiran

no,of records to display

Ex: $ wc -l <filename>

no. of words to display

Ex: $ wc -w <filename>

no. of characters to display

Ex: $ wc -c <filename>

max – length – characters to disply in a file

Ex: $ wc -L <filename>

7. **tee :-** It displays and redirects the same time

read from standard input and write to standard output and files

Ex: $ ls -l <filename>

$ ls -l | tee <filename>

$ ls ---full-time | tee -a <filename> - appending existing file

Note: without | (pipe symbol) tee filter don't works

8. **tr :-** Translate characters by characters

Translate or delete characters

Syntax: $ tr [option] <filename>

Ex: $ tr "*", "#" <filename

Ex: $ tr " abcd", "xyz" <filename

Duplicate characters avoid with sequential order

Ex: $ tr -s "r" < filename

Delete the characters

Ex: $ tr -d "r" < filename

9. **Sort :-** sort lines of text files according to the first character

Syntax: sort [option] <filename>

Ex : sort student

student filename

01 vijay 60 90

12 anil 70 75

32 sujan 65 40

04 hari 70 45

22 raju 85 50

Sorting priority

1. Blank space
2. Special character
3. Numeric
4. Uppercase
5. Lowercase

Numeric :

Ex: $ sort -n <filename>

$ Sort -nr <filename> - Numeric and reverse

If u want permanent

Ex: $ sort -nr <filename> > new file

$ sort -t ":" - field separator

According to the second fields

$ Sort -dt ":" -k2 - directory sorting

$ ls -l | sort -b – k5 - it takes the blank about Field separator

10. Cut :- Remove sections from each line of files.

1. Character cutting

Ex: $ cut -c <filename>

First character of the all lines

Ex: $ cut -c1 <filename>

To cut the first Character Of the all records

Ex: $ cut -c8 <filename>

To cut the 8th character in each line

Ex: $ cut -c4, 8 <filename>

To cut 4th and 8th Character In each line

Ex: $ cut -c4 -8 <filename>

To cut 4th to 8th Characters range

2. Field cutting

-d field separator

Ex: $ cut -d ":" -f2 <filename>

Cut 2nd field in all records

Ex: $ cut -d ":" -f2, 5 <filename>

Cut 2nd and 5th records

Ex: $ cut -d ":" -f2, -5 <filename>

Cut 2nd to 5th records

Ex: $ ls -l | tr -s " "

Sequence space

Ex: $ ls -l | tr -s " " | cut -d " " -f5

Sequence and cut

11. Uniq :- Remove duplicate lines from a sorted file

Syntax: $ uniq [option] <filename>

Ex: $ uniq -d <filename >

Displays only the Duplicate records

Ex: $ uniq -D <filename>

Print all duplicate Records and demeliting is done with blank lines

Ex: $ uniq –c <filename>

Prefix lines by the Number of occurrences

Advance Filters

1. Grep

The grep utilities are a family of Unix tools, including grep, egrep, and fgrep, that perform repetitive searching tasks. The tools in the grep family are very similar, and all are used for searching the contents of files for information that matches particular criteria. For most purposes, you'll want to use fgrep, since it's generally the fastest

The general syntax of the grep commands is:

Syntax: grep [-options] pattern [filename]

You can use fgrep to find all the lines of a file that contain a particular word. For example, to list all the lines of a file named my file in the current directory that contain the word "dog", enter at the Unix prompt:

Ex: fgrep dog myfile

This will also return lines where "dog" is embedded in larger words, such as "dogma" or "dogged". You can use the -w option with the grep command to return only lines where "dog" is included as a separate word:

Ex: grep -w dog myfile

To search for several words separated by spaces, enclose the whole search string in quotes, for example:

Ex: fgrep "dog named Checkers" myfile

The fgrep command is case sensitive; specifying "dog" will not match "Dog" or "DOG". You can use the -i option with the grep command to match both upper- and lowercase letters:

Ex: grep -i dog myfile

To list the lines of myfile that do not contain "dog", use the -v option:

Ex: fgrep -v dog myfile

If you want to search for lines that contain any of several different words, you can create a second file (named second file in the following example) that contains those words, and then use the -f option:

Ex: fgrep -f second file my file

You can also use wildcards to instruct fgrep to search any files that match a particular pattern. For example, if you wanted to find lines containing "dog" in any of the files in your directory with names beginning with "my", you could enter:

Ex: fgrep dog my*

This command would search files with names such as my file, my.hw1, and my stuff in the current directory. Each line returned would be prefaced with the name of the file where the match was found.

By using pipes and/or redirection, you can use the output from any of these commands with other Unix tools, such as more, sort, and cut. For example, to print the fifth word of every line of my file containing "dog", sort the words alphabetically, and then filter the output through the more command for easy reading, you would enter at the Unix prompt:

Ex: fgrep dog myfile | cut -f5 -d" " | sort | more

If you want to save the output in a file in the current directory named new file, enter:

Ex: fgrep dog my file | cut -f5 -d" " | sort > new file

$ grep –n <file> - To display record number

$ grep -i<file> - To ignore record

$ grep -c <file> - To count in how many records expression

$ grep –E <file> - To search for multiple expression

$ grep –L <file> - It give the file name and which the regular expression

$ grep –r <file> - To search regressively and present working directory

$ grep –w <file> - To search for the exact match for the word

$ grep –s <file> - To suppress errors

$ grep "jai" <file> - It prints the expressive record which have got the "jai"

$ grep –n "jai" <file> - It display the record number and expression what we give "jai"

$ grep –in "jay" <file> - It ignore and prints the record "Jay"

$ grep –E "jay prem" <file> - It search for multiple expressions and prints the record

$ grep –l "jay"* - It displays the filename which the "Jay" expression is

2. SED

Sed is a stream editor. A stream editor is used to perform basic text transformations on an input stream (a file or input from a pipeline). While in some ways similar to an editor which permits scripted edits (such as ed), sed works by making only one pass over the input(s), and is consequently more efficient. But it is sed's ability to filter text in a pipeline, which particularly distinguishes it from other types of editors.

Sed works as follows: it reads from the standard input, one line at a time. for each line, it executes a series of editing commands, then the line is written to STDOUT. An example that shows how it works: we use the s command. s means "substitute" or search and replace. The format is

s/regular-expression/replacement text/{flags}

We won't discuss all the flags yet. The one we use below is g, which means, "replace all matches"

```
>cat file
I have three dogs and two cats
>sed -e 's/dog/cat/g' -e 's/cat/elephant/g' file
I have three elephants and two elephants
>
```

OK. So what happened? Firstly, *sed* read in the line of the file and executed

s/dog/cat/g

Which produced the following text:

I have three cats and two cats

and then the second command was performed on the *edited line* and the result was

I have three elephants and two elephants

We actually have a name for the "current text": it is called the *pattern space*. So a precise definition of what *sed* does is as follows:

sed reads the standard input into the pattern space, performs a sequence of editing commands on the pattern space, then writes the pattern space to STDOUT.

Syntax: sed [OPTION]... {Script-only-if-no-other-script} [Input-file]...

OR

Syntax: sed [option] 'address/action' <file>

Options:	-n	To suppress input/output
	-e	Multiple Expressions
	-d	Delete
	-p	Print
	-s	Substitute

Firstly, the way you usually use *sed* is as follows:

```
>sed -e 'command1' -e 'command2' -e 'command3' file
>{shell command}|sed -e 'command1' -e 'command2'
>sed -f sedscript.sed file
>{shell command}|sed -f sedscript.sed
```

so *sed* can read from a file or STDIN, and the commands can be specified in a file or on the command line. Note the following:

That if the commands are read from a file, trailing white space can be fatal, in particular, it will cause scripts to fail for no apparent reason. I recommend editing sed scripts with an editor such as vim, which can show end of line characters so that you can "see" trailing white space at the end of line.

Substitute

The format for the substitute command is as follows:

[address1[,address2]]s/pattern/replacement/[flags]

The flags can be any of the following

n replace nth instance of pattern with replacement

p write pattern space to STDOUT if a successful substitution takes place

w file Write the pattern space to file if a successful substitution takes place

Delete

The delete command is very simple in its syntax: it goes like this

[address1[,]]d

And it deletes the content of the pattern space. All following commands are skipped (after all, there's very little you can do with an empty pattern space), and a new line is read into the pattern space.

Example - 1

>cat file

http://www.foo.com/mypage.html

>sed -e 's@http://www.foo.com@http://www.bar.net@' file

http://www.bar.net/mypage.html

Note that we used a different delimiter, @ for the substitution command. Sed permits several delimiters for the s command including @%,;: these alternative delimiters are good for substitutions which include strings such as filenames, as it makes your sed code much more readable.

Example - 2

>cat file

the black cat was chased by the brown dog

>sed -e 's/black/white/g' file

the white cat was chased by the brown dog

That was pretty straightforward. Now we move on to something more interesting.

Example - 3

>cat file

the black cat was chased by the brown dog.
the black cat was not chased by the brown dog

>sed -e '/not/s/black/white/g' file

the black cat was chased by the brown dog.
the white cat was not chased by the brown dog.

In this instance, the substitution is only applied to lines matching the regular expression not. Hence it is not applied to the first line.

Example - 4

>cat file

line 1 (one)
line 2 (two)
line 3 (three)

Example 4a

>sed -e '1,2d' file
line 3 (three)

Example 4b

>sed -e '3d' file
line 1 (one)
line 2 (two)

Example 4c

>sed -e '1,2s/line/LINE/' file
LINE 1 (one)
LINE 2 (two)
line 3 (three)

Example 4d

>sed -e '/^line.*one/s/line/LINE/' -e '/line/d' file
LINE 1 (one)

3a This was pretty simple: we just deleted lines 1 to 2.

3b This was also pretty simple. We deleted line 3.

3c In this example, we performed a substitution on lines 1-2.

3d now this is more interesting, and deserves some explanation. Firstly, it is clear that line 2 and 3 get deleted. But let's look closely at what happens to line 1.
First, line 1 is read into the pattern space. It matches the regular expression ^line.*one So the substitution is carried out, and the resulting pattern space looks like this:

LINE 1 (one)

So now the second command is executed. but since the pattern space does not match the regular expres-

sion line, the delete command is not executed.

Example 5

```
>cat file
hello
this text is wiped out
Wiped out
hello (also wiped out)
WiPed out TOO!
goodbye
1.  This text is not deleted
2.  neither is this ... ( goodbye )
3.  neither is this
    hello
    but this is
    and so is this
    and unless we find another g**dbye
    every line to the end of the file gets deleted

>sed -e '/hello/,/goodbye/d' file
1.  This text is not deleted
2.  neither is this ... (goodbye)
3.  neither is this
```

This illustrates how the addressing works when two pattern addresses are specified. sed finds the first match of the expression "hello", deleting every line read into the pattern space until it gets to the first line after the expression "goodbye". It doesn't apply the delete command to any more addresses until it comes across the expression "hello" again. Since the expression "goodbye" is not on any subsequent line, the delete command is applied to all remaining lines.

AWK

AWK is a simple and elegant pattern scanning and processing language

AWK is also the most portable scripting language

It was created in late 70th of the last century. The name was composed from the initial letters of three original authors Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. It is commonly used as a command-line filter in pipes to reformat the output of other commands. It's the precursor and the main inspiration of Perl. Although originated in Unix it is available and widely used in Windows environment too.

AWK takes two inputs: data file and command file. The command file can be absent and necessary commands can be passed as augments.

The basic function of awk is to search files for lines (or other units of text) that contain certain patterns. When a line matches one of the patterns, awk performs specified actions on that line. awk keeps processing input lines in this way until it reaches the end of the input files

Syntax : awk [option] 'selection criteria {action}' <file>

Options : -F - To specify the field separator

-f - To invoke the source code

{action} - Only the print action

predefined variables in awk

all predefined variables are in upper cases

FS - Input field separator

OFS - Ouput field separator

NF - Number of fields

NR - Record numbers or No. of records

$ - Fields in awk

Comparation Operator in awk

> - Grater than

>= - Grater than equal

< - Less than

<= - Less than equal

== - Equal to

!= - Not equal

~ - Matching

!~ - not matching

Logical Operator

&& - AND

|| - OR

awk has got 3 sections

1. BIGIN
2. MIDDLE
3. END

Begin is keyword for the begin section the variable can be assign in begin section

All the operator in the middle sections, Middle is not keyword for the middle section

What ever u print every thing in the section, End is the keyword for the end section

$ awk '/ajay/{print}'<file>

It prints the all the records

$ awk '/ajay|ramu/{print}'<file>

To search for multiple expressions and print

$ awk 'NR==4{print}' <file>

To print specific record

$ awk 'NR==3,NR==7{print}' <file>

To print range of records

$ awk 'NR>4{print}' <file>

To print all the records which are >4

$ awk 'NR>=4{print}' <file>

$ awk 'NR<4{print}' <file>

To print all the records which are <4

$ awk 'NR<=4{print}' <file>

if u want print only specific fields

$ awk -F ":" 'NR==4 {print $1, $3, $4}' <file>

simple awk program emulates the cat utility; it copies whatever you type on the keyboard to its standard output (why this works is explained shortly).

```
$ awk '{ print }'
Now is the time for all good men
-| Now is the time for all good men
to come to the aid of their country.
-| to come to the aid of their country.
Four score and seven years ago, ...
-| Four score and seven years ago, ...
What, me worry?
-| What, me worry?
```

Ctrl-d

Print the length of the longest input line:

```
awk '{if(length($0)> max)max = length($0) }
 END { print max }' data
```

Print every line that is longer than 80 characters:

```
awk 'length($0) > 80' data
```

The sole rule has a relational expression as its pattern and it has no action—so the default action, printing the record, is used.

Print the length of the longest line in data:

```
expand data | awk '{if x < length()) x =length()}
 END {print "maximum line length is " x}'
```

The input is processed by the expand utility to change tabs into spaces, so the widths compared are actually the right-margin columns.

Print every line that has at least one field:

```
awk 'NF > 0' data
```

This is an easy way to delete blank lines from a file (or rather, to create a new file similar to the old file but from which the blank lines have been removed).

Print seven random numbers from 0 to 100, inclusive:

```
awk 'BEGIN {for(i=1;i<=7;i++)print int(101*rand())}'
```

Print the total number of bytes used by *files*:

```
ls -l files | awk '{ x +=  $5 }
   END { print "total bytes: " x }'
```

Print the total number of kilobytes used by *files*:

```
ls -l files | awk '{ x +=  $5 }
 END { print "total K-bytes: " x + 1023)/1024 }'
```

Print a sorted list of the login names of all users:

```
awk -F: '{ print $1 }' /etc/passwd | sort
```

Count the lines in a file:

```
awk 'END { print NR }' data
```

Print the even-numbered lines in the data file:

```
awk 'NR % 2 == 0' data
```

If you use the expression 'NR % 2 == 1' instead, the program would print the odd-numbered lines

EXAMPLES

// is the comment character for awk. 'field' means 'column'

```
# Print first two fields in opposite order:
 awk '{ print $2, $1 }' file

// Print lines longer than 72 characters:
 awk 'length > 72' file

# Print length of string in 2nd column
 awk '{print length($2)}' file

# Add up first column, print sum and average:
    { s += $1 }
END { print "sum is", s, "average is", s/NR }

# Print fields in reverse order:

awk '{for i = NF; i > 0; --i)print $i }' file

// Print the last line
   {line = $0}
END {print line}

// Print the total number of lines that
 contain the word Pat
/Pat/ {nlines = nlines + 1}
END {print nlines}

// Print all lines between start/stop pairs:
awk '/start/, /stop/' file
```

```
# Print all lines whose first field is
 different from previous one:

 awk '$1 != prev { print; prev = $1 }' file

# Print column 3 if column 1 > column 2:
 awk '$1 > $2 {print $3}' file

# Print line if column 3 > column 2:
 awk '$3 > $2' file

# Count number of lines where col3 >col 1
 awk '$3 > $1 {print i + "1"; i++}' file
# Print sequence number and then column 1 of file:
 awk '{print NR, $1}' file

# Print every line after erasing the 2nd field
 awk '{$2 = ""; print}' file

# Print hi 28 times
 yes | head -28 | awk '{ print "hi" }'

# Print hi.0010 to hi.0099 (NOTE IRAF USERS!)
 yes | head -90 | awk '{printf("hi00%2.0f\n",
                          NR+9)}'

# Replace every field by its absolute value
 { for (i = 1; i <= NF; i = i + 1) if ($i < 0)
              $i = -$i print}

#       If you have another character that delimits fields, use the -F option
#       For example, to print out the phone number for Jones in the following file,
```

```
#       000902|Beavis|Theodore|333-242-2222|149092
#       000901|Jones|Bill|532-382-0342|234023
#       ...
#       type
 awk -F"|" '$2=="Jones"{print $4}' filename

# Some looping for printouts
 BEGIN{
        for (i=875;i>833;i--){
                printf "lprm -Plw %d\n", i
        } exit
    }
```

Formatted printouts are of the form printf("format\n", value1, value2, ... valueN)

e.g. printf("howdy %-8s What it is bro. %.2f\n", $1, $2*$3)

%s = string

%-8s = 8 character string left justified

%.2f = number with 2 places after .

%6.2f = field 6 chars with 2 chars after .

\n is newline

\t is a tab

```
# Print frequency histogram of column of numbers
$2 <= 0.1 {na=na+1}
($2 > 0.1) && ($2 <= 0.2) {nb = nb+1}
($2 > 0.2) && ($2 <= 0.3) {nc = nc+1}
($2 > 0.3) && ($2 <= 0.4) {nd = nd+1}
($2 > 0.4) && ($2 <= 0.5) {ne = ne+1}
($2 > 0.5) && ($2 <= 0.6) {nf = nf+1}
($2 > 0.6) && ($2 <= 0.7) {ng = ng+1}
($2 > 0.7) && ($2 <= 0.8) {nh = nh+1}
($2 > 0.8) && ($2 <= 0.9) {ni = ni+1}
($2 > 0.9) {nj = nj+1}
```

```
END {print na, nb, nc, nd, ne, nf, ng, nh, ni, nj, NR}

# Find maximum and minimum values present in
 column 1
NR == 1 {m=$1 ; p=$1}
$1 >= m {m = $1}
$1 <   p {p   $1}
END { print "Max = " m, "  Min = " p }

# Example of defining variables, multiple
 commands on one line

NR == 1 {prev=$4; preva = $1; prevb = $2; n=0; sum=0}
$4 != prev {print preva, prevb, prev, sum/n; n=0; sum=0; prev = $4; preva = $1; prevb = $2}
$4 == prev {n++; sum=sum+$5/$6}
END {print preva, prevb, prev, sum/n}

# Example of using substrings
# substr($2,9,7) picks out characters 9 thru 15 of column 2
{print "imarith", substr($2,1,7) " - " $3, "out."substr($2,5,3)}
{print "imarith", substr($2,9,7) " - " $3, "out."substr($2,13,3)}
{print "imarith", substr($2,17,7) " - " $3, "out."substr($2,21,3)}
print "imarith", substr($2,25,7) " - " $3, "out."substr($2,29,3)}
```

1. **Renaming within the name :-**

```
ls -1 *old* | awk '{print "mv "$1" "$1}' | sed s/old/new/2 | sh
```

(although in some cases it will fail, as in file_old_and_old)

2. **Remove only files:**

```
ls -l * | grep -v drwx | awk '{print "rm "$9}' | sh
```

or with awk alone:

```
ls -l|awk '$1!~/^drwx/{print $9}'|xargs rm
```

Be careful when trying this out in your home directory. We remove files!

3. **Remove only directories**

```
ls -l | grep '^d' | awk '{print "rm -r "$9}' | sh
```

or

```
ls -p | grep /$ | wk '{print "rm -r "$1}'
```

or with awk alone:

```
ls -l|awk '$1~/^d.*x/{print $9}'|xargs rm -r
```

Be careful when trying this out in your home directory. We remove things!

4. **Killing processes by name** (in this example we kill the process called netscape):
kill `ps auxww | grep netscape | egrep -v grep | awk '{print $2}'`

4.8 UNIX VARIABLE

Variables are a way of passing information from the shell to programs when you run them. Programs look "in the environment" for particular variables and if they are found will use the values stored. Some are set by the system, others by you, yet others by the shell, or any program that loads another program.

Standard UNIX variables are divide into two categories, environment variables and shell variables. In broad terms, shell variables apply only to the current instance of the shell and are used to set short-term working conditions; environment variables have a farther reaching significance, and those set at login are valid for the duration of the session. By convention, environment variables have UPPER CASE and shell variables have lower case names.

Environment Variables

An example of an environment variable is the OSTYPE variable. The value of this is the current operating system you are using. Type

% echo $OSTYPE

More examples of environment variables are

- USER (your login name)
- HOME (the path name of your home directory)
- HOST (the name of the computer you are using)
- ARCH (the architecture of the computers processor)
- DISPLAY (the name of the computer screen to display X windows)
- PRINTER (the default printer to send print jobs)
- PATH (the directories the shell should search to find a command)

Finding out the current values of these variables.

ENVIRONMENT variables are set using the setenv command, displayed using the printenv or env commands, and unset using the unsetenv command.

To show all values of these variables, type

% printenv | less

Shell Variables

An example of a shell variable is the history variable. The value of this is how many shell commands to save, allow the user to scroll back through all the commands they have previously entered. Type

% echo $history

More examples of shell variables are

- cwd (your current working directory)

- home (the path name of your home directory)
- path (the directories the shell should search to find a command)
- prompt (the text string used to prompt for interactive commands shell your login shell)

Finding out the current values of these variables.

SHELL variables are both set and displayed using the set command. They can be unset by using the unset command.

To show all values of these variables, type

% set | less

So what is the difference between PATH and path ?

In general, environment and shell variables that have the same name (apart from the case) are distinct and independent, except for possibly having the same initial values. There are, however, exceptions.

Each time the shell variables home, user and term are changed, the corresponding environment variables HOME, USER and TERM receive the same values. However, altering the environment variables has no effect on the corresponding shell variables.

PATH and path specify directories to search for commands and programs. Both variables always represent the same directory list, and altering either automatically causes the other to be changed.

Using and setting variables

Each time you login to a UNIX host, the system looks in your home directory for initialisation files. Information in these files is used to set up your working environment. The C and TC shells uses two files called .login and .cshrc (note that both file names begin with a dot).

At login the C shell first reads .cshrc followed by .login

* login is to set conditions which will apply to the whole session and to perform actions that are relevant only at login.
* cshrc is used to set conditions and perform actions specific to the shell and to each invocation of it.

The guidelines are to set ENVIRONMENT variables in the .login file and SHELL variables in the .cshrc file.

Setting shell variables in the .cshrc file

For example, to change the number of shell commands saved in the history list, you need to set the shell variable history. It is set to 100 by default, but you can increase this if you wish.

% set history = 200

Check this has worked by typing

% echo $history

However, this has only set the variable for the lifetime of the current shell. If you open a new xterm window, it will only have the default history value set. To PERMANENTLY set the value of history, you will need to add the set command to the .cshrc file.

First open the .cshrc file in a text editor. An easy, user-friendly editor to use is nedit.

% nedit ~/.cshrc

Add the following line AFTER the list of other commands.

set history = 200

Save the file and force the shell to reread its .cshrc file buy using the shell source command.

% source .cshrc

Check this has worked by typing

% echo $history

Setting the path

When you type a command, your path (or PATH) variable defines in which directories the shell will look to find the command you typed. If the system returns a message saying "command: Command not found", this indicates that either the command doesn't exist at all on the system or it is simply not in your path.

For example, to run units, you either need to directly specify the units path (~/units174/bin/units), or you need to have the directory ~/units174/bin in your path.

You can add it to the end of your existing path (the $path represents this) by issuing the command:

% set path = ($path ~/units174/bin)

Test that this worked by trying to run units in any directory other that where units is actually located.

% cd

% units

To add this path PERMANENTLY, add the following line to your .cshrc AFTER the list of other commands.

set path = ($path ~/units174/bin)

4.9 ALIASES

Definition :- alias command: An alternate name, symbolic link, network configuration, or icon that points to a corresponding file or directory location. Aliases are useful for concatenating long strings or commands into shorter, less complex names. It can save you a lot of typing by assigning a name to long commands. alias is a built-in shell command in Linux / Unix operating systems.

Aliases can be created by supplying name/value pairs as arguments for the alias command. An example of the Bash shell syntax is:

alias copy="cp"

The corresponding syntax in the C shell or tcsh shell is:

alias copy "cp"

This alias means that when the command copy is read in the shell, it will be replaced with cp and that command will be executed instead.

4.10 SUMMARY

- The shell is the utility that processes users requests. When user type in a command at any terminal, the shell interprets the command and runs the requested program. UNIX supports multiple users and multiple tasks, and different users can invoke the same utility programs, including the shell\
- A pipe is nothing but a temporary storage place where the output of one command is stored and then passed as the input for second command. Pipes are used to run more than two commands (Multiple commands) from same command line
- A filter is a small and specialized program in Unix-like operating systems that transforms plain text data in some meaningful way and that can be used together with other filters and pipes to form a series of operations that produces highly specific results.
- The grep utilities are a family of Unix tools, including grep, egrep, and fgrep, that perform repetitive searching tasks. The tools in the grep family are very similar, and all are used for searching the contents of files for information that matches particular criteria.
- Sed is a stream editor. A stream editor is used to perform basic text transformations on an input stream (a file or input from a pipeline).
- alias command is An alternate name, symbolic link, network configuration, or icon that points to a corresponding file or directory location. Aliases are useful for concatenating long strings or commands into shorter, less complex names.

4.11 QUESTIONS

1 What is kernel?

2 What is Shell? List all types of shell.

3 What is the reletion between shell and kernel?

4 How we can login in the UNIX? list the steps.

5 What is standard streams in UNIX? List all types of streams.

6 Write a short on UNIX Redirection.

7 What is piper and filters? explain with example.

8 Write a short note on :

(i) regular expression

(ii) variable in UNIX

(iii) aliases

UNIT 5 : AWK PROGRAMMING

Structure of the Unit

5.0 OBJECTIVES

After completing this unit you will be able to explain

- What is awk and gawk? when to use awk.
- Input and output in awk ,accessing multipleline record and fields in record
- Numeric, string and regular expression
- Patterns and action, array in awk etc.

5.1 INTRODUCTION

Several kinds of tasks occur repeatedly when working with text files. You might want to extract certain lines and discard the rest. Or you may need to make changes wherever certain patterns appear, but leave the rest of the file alone. Writing single-use programs for these tasks in languages such as C, C++, or Pascal is time-consuming and inconvenient. Such jobs are often easier with awk. The awk utility interprets a special-purpose programming language that makes it easy to handle simple data-reformatting jobs.

The GNU implementation of awk is called gawk it is fully compatible with the System every version of awk. gawk is also compatible with the POSIX specification of the awk language. This means that all properly written awk programs should work with gawk. Thus, we usually don't distinguish between gawk and other awk implementations.

Using awk allows you to:

- Manage small, personal databases
- Generate reports
- Validate data
- Produce indexes and perform other document preparation tasks
- Experiment with algorithms that you can adapt later to other computer languages In addition, gawk provides facilities that make it easy to:
- Extract bits and pieces of data for processing

- Sort data
- Perform simple network communications

This book teaches you about the awk language and how you can use it effectively. You should already be familiar with basic system commands, such as cat and ls,1 as well as basic shell facilities, such as input/output (I/O) redirection and pipes.

Implementations of the awk language are available for many different computing environments. gawk runs on a broad range of Unix systems, ranging from 80386 PC-based computers up through large-scale systems, gawk has also been ported MS-DOS, Microsoft Windows (all versions) and OS/2 PCs.

5.2 HISTORY OF AWK AND GAWK

The name awk comes from the initials of its designers: Alfred V. Aho, Peter J. Weinberger and Brian W. Kernighan. The original version of awk was written at AT&T Bell Laboratories. a new version made the programming language more powerful, introducing user-defined functions, multiple input streams, and computed regular expressions. The specification for awk in the POSIX Command Language and Utilities standard further clarified the language. Both the gawk designers and the original Bell Laboratories awk designers provided feedback for the POSIX specification.

Paul Rubin wrote the GNU implementation, gawk, and Jay Fenlason completed it, with advice from Richard Stallman. John Woods contributed parts of the code as well.

5.3 WHEN TO USE AWK

The basic function of awk is to search files for lines (or other units of text) that contain certain patterns. When a line matches one of the patterns, awk performs specified actions on that line. awk keeps processing input lines in this way until it reaches the end of the input files.

Programs in awk are different from programs in most other languages, because awk programs are data-driven; that is, you describe the data you want to work with and then what to do when you find it. Most other languages are procedural; you have to describe, in great detail, every step the program is to take. When working with procedural languages, it is usually much harder to clearly describe the data your program will process. For this reason, awk programs are often refreshingly easy to read and write.

When you run awk, you specify an awk program that tells awk what to do. The program consists of a series of rules. Each rule specifies one pattern to search for and one action to perform upon finding the pattern.

Syntactically, a rule consists of a pattern followed by an action. The action is enclosed in curly braces to separate it from the pattern. Newlines usually separate rules. Therefore, an awk program looks like this:

```
pattern { action }

pattern { action }
```

5.4 INPUT AND OUTPUT IN AWK

5.4.1 How to Run awk Programs

There are several ways to run an awk program. If the program is short, it is easiest to include it in

the command that runs awk, like this:

awk 'program' input-file1 input-file2 ...

When the program is long, it is usually more convenient to put it in a file and run it with a command like this:

awk -f program-file input-file1 input-file2 ...

This section discusses both mechanisms, along with several variations of each.

5.4.2 Running awk Without Input Files

You can also run awk without any input files. If you type the following command line:

awk 'program'

awk applies the program to the standard input, which usually means whatever you type on the terminal. This continues until you indicate end-of-file by typing Ctrl-d. (On other operating systems, the end-of-file character may be different. For example, on MS-DOS, it is Ctrl-z.)

example

(BEGIN is a feature we haven't discussed yet):

```
$ awk "BEGIN { print \"Don't Panic!\" }"
Don't Panic!
```

This program does not read any input. The '\' before each of the inner double quotes is necessary because of the shell's quoting rules—in particular because it mixes both single quotes and double quotes.1 This next simple awk program emulates the cat utility; it copies whatever you type on the keyboard to its standard output .

```
$ awk '{ print }'
HELLO HOW ARE YOU
HELLO HOW ARE YOU
Ctrl-d
```

5.4.3 Running Long Programs

Sometimes your awk programs can be very long. In this case, it is more convenient to put the program into a separate file. In order to tell awk to use that file for its program, you type:

awk -f source-file input-file1 input-file2 ...

The '-f' instructs the awk utility to get the awk program from the file source-file. Any file name can be used for source-file.

5.4.4 Comments in awk Programs

A comment is some text that is included in a program for the sake of human readers; it is not really an executable part of the program. Comments can explain what the program does and how it works. Nearly all programming languages have provisions for comments, as programs are typically hard to

understand without them.

In the awk language, a comment starts with the sharp sign character ('#') and continues to the end of the line. The '#' does not have to be the first character on the line. The awk language ignores the rest of a line following a sharp sign. For example, we could have put the following into 'advice':

 # This program prints a nice friendly message. It helps

 # keep novice users from being afraid of the computer.

BEGIN { print "Don't Panic!" }

You can put comment lines into keyboard-composed throwaway awk programs, but this usually isn't very useful; the purpose of a comment is to help you or another person understand the program when reading it at a later time.

5.4.5 Reading Input Files

In the typical awk program, all input is read either from the standard input (by default, this is the keyboard, but often it is a pipe from another command) or from files whose names you specify on the awk command line. If you specify input files, awk reads them in order, processing all the data from one before going on to the next. The name of the current input file can be found in the built-in variable FILENAME

The input is read in units called records, and is processed by the rules of your program one record at a time. By default, each record is one line. Each record is automatically split into chunks called fields. This makes it more convenient for programs to work on the parts of a record.

On rare occasions, you may need to use the getline command. The getline command is valuable, both because it can do explicit input from any number of files, and because the files used with it do not have to be named on the awk command line

5.4.6 Specifying How Fields Are Separated

The field separator, which is either a single character or a regular expression, controls the way awk splits an input record into fields. awk scans the input record for character sequences that match the separator; the fields themselves are the text between the matches. In the examples that follow, we use the bullet symbol (•) to represent spaces in the output. If the field separator is 'oo', then the following line:

5.4.7 Using Regular Expressions to Separate Fields

The previous subsection discussed the use of single characters or simple strings as the value of FS. More generally, the value of FS may be a string containing any regular expression. In this case, each match in the record for the regular expression separates fields. For example, the assignment:

FS = ", \t"

makes every area of an input line that consists of a comma followed by a space and a TAB into a field separator. For a less trivial example of a regular expression, try using single spaces to separate fields the way single commas are used. FS can be set to "[ ]" (left bracket, space, right bracket). This regular expression matches a single space and nothing else .

There is an important difference between the two cases of 'FS = " "' (a single space) and 'FS = "[\t\n]+"' (a regular expression matching one or more spaces, tabs, or newlines). For both values of FS, fields are separated by runs (multiple adjacent occurrences) of spaces, tabs, and/or newlines. However, when the value of FS is " ", awk first strips leading and trailing whitespace from the record and then decides where the fields are. For example, the following pipeline prints 'b':

```
$ echo ' a b c d ' | awk '{ print $2 }'
a b
```

However, this pipeline prints 'a' (note the extra spaces around each letter):

```
$ echo ' a b c d ' | awk 'BEGIN { FS = "[ \t\n]+" }
```

```
> { print $2 }'
a a
```

In this case, the first field is null or empty. The stripping of leading and trailing whitespace also comes into play whenever $0 is recomputed. For instance, study this pipeline:

```
$ echo ' a b c d' | awk '{ print; $2 = $2; print }'
 a b c d
a b c d
```

The first print statement prints the record as it was read, with leading whitespace intact. The assignment to $2 rebuilds $0 by concatenating $1 through $NF together, separated by the value of OFS. Because the leading whitespace was ignored when finding $1, it is not part of the new $0. Finally, the last print statement prints the new $0.

5.4.8 Making Each Character a Separate Field

There are times when you may want to examine each character of a record separately. This can be done in gawk by simply assigning the null string ("") to FS. In this case, each individual character in the record becomes a separate field. For example:

```
$ echo a b | gawk 'BEGIN { FS = "" }
> {
> for (i = 1; i <= NF; i = i + 1)
> print "Field", i, "is", $i
> }'
Field 1 is a
Field 2 is
Field 3 is b
```

5.4.9 Multiple-Line Records

In some databases, a single line cannot conveniently hold all the information in one entry. In such cases, you can use multiline records. The first step in doing this is to choose your data format.

One technique is to use an unusual character or string to separate records. For example, you could use the formfeed character (written \f in **awk**, as in C) to separate them, making each record a page of the file. To do this, just set the variable **RS** to "\f" (a string containing the formfeed character). Any other character could equally well be used, as long as it won't be part of the data in a record.

Another technique is to have blank lines separate records. By a special dispensation, an empty string as the value of **RS** indicates that records are separated by one or more blank lines. When **RS** is set to the empty string, each record always ends at the first blank line encountered. The next record doesn't start until the first nonblank line that follows. No matter how many blank lines appear in a row, they all act as one record separator. (Blank lines must be completely empty; lines that contain only whitespace do not count.)

You can achieve the same effect as **RS = ""** by assigning the string **"\n\n+"** to **RS**. This regexp matches the newline at the end of the record and one or more blank lines after the record. In addition, a regular expression always matches the longest possible sequence when there is a choice So the next record doesn't start until the first nonblank line that follows—no matter how many blank lines appear in a row, they are considered one record separator.

There is an important difference between **RS = ""** and **RS = "\n\n+"**. In the first case, leading newlines in the input data file are ignored, and if a file ends without extra blank lines after the last record, the final newline is removed from the record. In the second case, this special processing is not done. (d.c.)

Now that the input is separated into records, the second step is to separate the fields in the record. One way to do this is to divide each of the lines into fields in the normal manner. This happens by default as the result of a special feature. When **RS** is set to the empty string, the newline character *always* acts as a field separator. This is in addition to whatever field separations result from **FS**.

The original motivation for this special exception was probably to provide useful behavior in the default case (i.e., **FS** is equal to " "). This feature can be a problem if you really don't want the newline character to separate fields, because there is no way to prevent it. However, you can work around this by using the **split** function to break up the record manually

Another way to separate fields is to put each field on a separate line: to do this, just set the variable **FS** to the string **"\n"**. (This simple regular expression matches a single newline.) A practical example of a data file organized this way might be a mailing list, where each entry is separated by blank lines. Consider a mailing list in a file named **addresses**, which looks like this:

```
Jane Doe

123 Main Street

Anywhere, SE 12345-6789

John Smith

456 Tree-lined Avenue

Smallville, MW 98765-4321

...
```

A simple program to process this file is as follows:

```
# addrs.awk — simple mailing list program
# Records are separated by blank lines.
# Each line is one field.
BEGIN { RS = "" ; FS = "\n" }
{
   print "Name is:", $1
   print "Address is:", $2
   print "City and State are:", $3
   print ""
}
```

Running the program produces the following output:

```
$ awk -f addrs.awk addresses
```

```
-| Name is: Jane Doe
-| Address is: 123 Main Street
-| City and State are: Anywhere, SE 12345-6789
-|
-| Name is: John Smith
-| Address is: 456 Tree-lined Avenue
-| City and State are: Smallville, MW 98765-4321
-|
...
```

for a more realistic program that deals with address lists. The following table summarizes how records are split, based on the value of **RS**:

RS == "\n"

Records are separated by the newline character (**\n**). In effect, every line in the data file is a separate record, including blank lines. This is the default.

RS == *any single character*

Records are separated by each occurrence of the character. Multiple successive occurrences delimit empty records.

RS == ""

Records are separated by runs of blank lines. The newline character always serves as a field separator, in addition to whatever value **FS** may have. Leading and trailing newlines in a file are ignored.

RS == *regexp*

Records are separated by occurrences of characters that match *regexp*. Leading and trailing matches of *regexp* delimit empty records. (This is a **gawk** extension; it is not specified by the POSIX standard.)

In all cases, **gawk** sets **RT** to the input text that matched the value specified by **RS**.

5.5 EXPRESSIONS

Expressions are the basic building blocks of awk patterns and actions. An expression evaluates to a value, which you can print, test, store in a variable or pass to a function. Additionally, an expression can assign a new value to a variable or a field, with an assignment operator.

An expression can serve as a pattern or action statement on its own. Most other kinds of statements contain one or more expressions which specify data on which to operate. As in other languages, expressions in awk include variables, array references, constants, and function calls, as well as combinations of these with various operators.

5.5.1 Numeric and String Constants

A **numeric constant** stands for a number. This number can be an integer, a decimal fraction, or a number in scientific (exponential) notation. Here are some examples of numeric constants, which all have the same value:

105

1.05e+2

1050e-1

A string constant consists of a sequence of characters enclosed in double-quote marks. For example:

"hello"

represents the string whose contents are 'hello'. Strings in **gawk** can be of any length and they can contain any of the possible eight-bit ASCII characters including ASCII NUL (character code zero). Other **awk** implementations may have difficulty with some character codes.

5.5.2 Regular Expressions

A **regular expression**, or **regexp**, is a way of describing a set of strings. Because regular expressions are such a fundamental part of **awk** programming, their format and use deserve a separate chapter.

A regular expression enclosed in slashes ('/') is an **awk** pattern that matches every input record whose text belongs to that set.

The simplest regular expression is a sequence of letters, numbers, or both. Such a regexp matches any string that contains that sequence. Thus, the regexp 'foo' matches any string containing 'foo'. Therefore, the pattern **/foo/** matches any input record containing the three characters 'foo', *anywhere* in the record. Other kinds of regexps let you specify more complicated classes of strings.

5.5.3 Variables

Variables are ways of storing values at one point in your program for use later in another part of your program. You can manipulate them entirely within your program text, and you can also assign values to them on the **awk** command line.

Using Variables: Using variables in your programs.
Assignment Options: Setting variables on the command line and a summary of command line syntax. This is an advanced method of input.

5.5.4 Using Variables in a Program

Variables let you give names to values and refer to them later. You have already seen variables in many of the examples. The name of a variable must be a sequence of letters, digits and underscores, but it may not begin with a digit. Case is significant in variable names; **a** and **A** are distinct variables.

A variable name is a valid expression by itself; it represents the variable's current value. Variables are given new values with **assignment operators**, **increment operators** and **decrement operators**.

A few variables have special built-in meanings, such as **FS**, the field separator, and **NF**, the number of fields in the current input record, for a list of them. These built-in variables can be used and assigned just like all other variables, but their values are also used or changed automatically by **awk**. All built-in variables names are entirely upper-case.

Variables in **awk** can be assigned either numeric or string values. By default, variables are initialized to the empty string, which is zero if converted to a number. There is no need to "initialize" each variable explicitly in **awk**, the way you would in C and in most other traditional languages.

5.5.5 Assigning Variables on the Command Line

You can set any **awk** variable by including a **variable assignment** among the arguments on the

command line when you invoke **awk** .Such an assignment has this form:

variable=text

With it, you can set a variable either at the beginning of the **awk** run or in between input files.

If you precede the assignment with the '-v' option, like this:

-v *variable=text*

then the variable is set at the very beginning, before even the **BEGIN** rules are run. The '-v' option and its assignment must precede all the file name arguments, as well as the program text.

Otherwise, the variable assignment is performed at a time determined by its position among the input file arguments: after the processing of the preceding input file argument. For example:

awk '{ print $n }' n=4 inventory-shipped n=2 BBS-list

prints the value of field number **n** for all input records. Before the first file is read, the command line sets the variable **n** equal to four. This causes the fourth field to be printed in lines from the file 'inventory-shipped'. After the first file has finished, but before the second file is started, **n** is set to two, so that the second field is printed in lines from 'BBS-list'.

```
$ awk '{ print $n }' n=4 inventory-shipped n=2 BBS-list
-| 15
-| 24
...
-| 555-5553
-| 555-3412
...
```

Command line arguments are made available for explicit examination by the **awk** program in an array named **ARGV** .

awk processes the values of command line assignments for escape sequences (d.c.)

5.5.6 Arithmetic Operators

The **awk** language uses the common arithmetic operators when evaluating expressions. All of these arithmetic operators follow normal precedence rules, and work as you would expect them to.

Here is a file 'grades' containing a list of student names and three test scores per student (it's a small class):

```
Pat  100 97 58

Sandy  84 72 93
```

```
Chris  72 92 89
```

This programs takes the file 'grades', and prints the average of the scores.

```
$ awk '{ sum = $2 + $3 + $4 ; avg = sum / 3
>       print $1, avg }' grades
-| Pat 85
-| Sandy 83
-| Chris 84.3333
```

This table lists the arithmetic operators in **awk**, in order from highest precedence to lowest:

-x Negation.

+x Unary plus. The expression is converted to a number.

x ^ y

*x ** y*

Exponentiation: *x* raised to the *y* power. '2 ^ 3' has the value eight. The character sequence '**' is equivalent to '^'. (The POSIX standard only specifies the use of '^' for exponentiation.)

*x * y* Multiplication.

x / y Division. Since all numbers in **awk** are real numbers, the result is not rounded to an integer: '3 / 4' has the value 0.75.

x % y

Remainder. The quotient is rounded toward zero to an integer, multiplied by *y* and this result is subtracted from *x*. This operation is sometimes known as "trunc-mod." The following relation always holds:

```
b * int(a / b) + (a % b) == a
```

One possibly undesirable effect of this definition of remainder is that *x* **%** *y* is negative if *x* is negative. Thus,

```
-17 % 8 = -1
```

In other **awk** implementations, the signedness of the remainder may be machine dependent.

x + y Addition.

x - y Subtraction.

For maximum portability, do not use the '**' operator.

Unary plus and minus have the same precedence, the multiplication operators all have the same precedence, and addition and subtraction have the same precedence.

5.5.7 String Concatenation

There is only one string operation: concatenation. It does not have a specific operator to represent it. Instead, concatenation is performed by writing expressions next to one another, with no operator. For example:

```
$ awk '{ print "Field number one: " $1 }' BBS-list

-| Field number one: aardvark

-| Field number one: alpo-net

...
```

Without the space in the string constant after the ':', the line would run together. For example:

```
$ awk '{ print "Field number one:" $1 }' BBS-list

-| Field number one:aardvark

-| Field number one:alpo-net

...
```

Since string concatenation does not have an explicit operator, it is often necessary to insure that it happens where you want it to by using parentheses to enclose the items to be concatenated. For example, the following code fragment does not concatenate **file** and **name** as you might expect:

```
file = "file"

name = "name"

print "something meaningful" > file name
```

It is necessary to use the following:

```
print "something meaningful" > (file name)
```

We recommend that you use parentheses around concatenation in all but the most common contexts (such as on the right-hand side of '=').

5.5.8 Boolean Expressions

A **boolean expression** is a combination of comparison expressions or matching expressions, using the boolean operators "or" (`||'), "and" (`&&'), and "not" (`!'), along with parentheses to control nesting. The truth value of the boolean expression is computed by combining the truth values of the component expressions. Boolean expressions are also referred to as **logical expressions**. The terms are equivalent.

Boolean expressions can be used wherever comparison and matching expressions can be used. They can be used in **if**, **while**, **do** and **for** statements. They have numeric values (one if true, zero if false), which come into play if the result of the boolean expression is stored in a variable, or used in arithmetic.

In addition, every boolean expression is also a valid pattern, so you can use one as a pattern to control the execution of rules.

Here are descriptions of the three boolean operators, with examples.

boolean1 **&&** boolean2

True if both *boolean1* and *boolean2* are true. For example, the following statement prints the current input record if it contains both '2400' and 'foo'.

```
if ($0 ~ /2400/ && $0 ~ /foo/) print
```

The subexpression *boolean2* is evaluated only if *boolean1* is true. This can make a difference when *boolean2* contains expressions that have side effects: in the case of '$0 ~ /foo/ && ($2 == bar++)', the variable **bar** is not incremented if there is no 'foo' in the record.

boolean1 || boolean2

True if at least one of *boolean1* or *boolean2* is true. For example, the following statement prints all records in the input that contain *either* '2400' or 'foo', or both.

```
if ($0 ~ /2400/ || $0 ~ /foo/) print
```

The subexpression *boolean2* is evaluated only if *boolean1* is false. This can make a difference when **boolean2** contains expressions that have side effects.

! boolean

True if **boolean** is false. For example, the following program prints all records in the input file 'BBS-list' that do *not* contain the string 'foo'.

```
awk '{ if (! ($0 ~ /foo/)) print }' BBS-list
```

The '&&' and '||' operators are called **short-circuit** operators because of the way they work. Evaluation of the full expression is "short-circuited" if the result can be determined part way through its evaluation.

You can continue a statement that uses '&&' or '||' simply by putting a newline after them. But you cannot put a newline in front of either of these operators without using backslash continuation.

The actual value of an expression using the '!' operator will be either one or zero, depending upon the truth value of the expression it is applied to.

The '!' operator is often useful for changing the sense of a flag variable from false to true and back again. For example, the following program is one way to print lines in between special bracketing lines:

```
$1 == "START"  { interested = ! interested }

interested == 1 { print }

$1 == "END"    { interested = ! interested }
```

The variable **interested**, like all **awk** variables, starts out initialized to zero, which is also false. When a line is seen whose first field is 'START', the value of **interested** is toggled to true, using '!'. The next rule prints lines as long as **interested** is true. When a line is seen whose first field is 'END', **interested** is toggled back to false.

5.6 PATTERNS AND ACTIONS

As you have already seen, each awk statement consists of a pattern with an associated action. Patterns in awk control the execution of rules: a rule is executed when its pattern matches the current input record.

5.6.1 Kinds of Patterns

Here is a summary of the types of patterns supported in **awk**.

/regular expression/

A regular expression as a pattern. It matches when the text of the input record fits the regular expression.

expression

A single expression. It matches when its value is non-zero (if a number) or non-null (if a string).

pat1, pat2

A pair of patterns separated by a comma, specifying a range of records. The range includes both the initial record that matches *pat1*, and the final record that matches *pat2*.

BEGIN

END

Special patterns for you to supply start-up or clean-up actions for your **awk** program.

empty

The empty pattern matches every input record.

5.6.2 Regular Expressions as Patterns

We have been using regular expressions as patterns since our early examples. This kind of pattern is simply a regexp constant in the pattern part of a rule. Its meaning is '$0 ~ */pattern/*'. The pattern matches when the input record matches the regexp. For example:

```
/foo|bar|baz/  { buzzwords++ }

END        { print buzzwords, "buzzwords seen" }
```

5.6.3 Expressions as Patterns

Any **awk** expression is valid as an **awk** pattern. Then the pattern matches if the expression's value is non-zero (if a number) or non-null (if a string).

The expression is reevaluated each time the rule is tested against a new input record. If the expression uses fields such as **$1**, the value depends directly on the new input record's text; otherwise, it depends only on what has happened so far in the execution of the **awk** program, but that may still be useful.

A very common kind of expression used as a pattern is the comparison expression, using the comparison operators described in section Variable Typing and Comparison Expressions.

Regexp matching and non-matching are also very common expressions. The left operand of the '~' and '!~' operators is a string. The right operand is either a constant regular expression enclosed in slashes (*/regexp/*), or any expression, whose string value is used as a dynamic regular expression .

The following example prints the second field of each input record whose first field is precisely 'foo'.

```
$ awk '$1 == "foo" { print $2 }' BBS-list
```

(There is no output, since there is no BBS site named "foo".) Contrast this with the following regular expression match, which would accept any record with a first field that contains 'foo':

```
$ awk '$1 ~ /foo/ { print $2 }' BBS-list

-| 555-1234

-| 555-6699

-| 555-6480

-| 555-2127
```

Boolean expressions are also commonly used as patterns. Whether the pattern matches an input record depends on whether its subexpressions match.

For example, the following command prints all records in 'BBS-list' that contain both '2400' and 'foo'.

```
$ awk '/2400/ && /foo/' BBS-list
-| fooey      555-1234    2400/1200/300    B
```

The following command prints all records in 'BBS-list' that contain *either* '2400' or 'foo', or both.

```
$ awk '/2400/ || /foo/' BBS-list
-| alpo-net     555-3412    2400/1200/300          A
-| bites         555-1675    2400/1200/300          A
```

```
-| fooey      555-1234   2400/1200/300     B
-| foot       555-6699   1200/300          B
-| macfoo     555-6480   1200/300          A
-| sdace      555-3430   2400/1200/300     A
-| sabafoo    555-2127   1200/300          C
```

The following command prints all records in 'BBS-list' that do *not* contain the string 'foo'.

```
$ awk '! /foo/' BBS-list
-| aardvark   555-5553   1200/300          B
-| alpo-net   555-3412   2400/1200/300     A
-| barfly     555-7685   1200/300          A
-| bites      555-1675   2400/1200/300     A
-| camelot    555-0542   300               C
-| core       555-2912   1200/300          C
-| sdace      555-3430   2400/1200/300     A
```

The subexpressions of a boolean operator in a pattern can be constant regular expressions, comparisons, or any other **awk** expressions. Range patterns are not expressions, so they cannot appear inside boolean patterns. Likewise, the special patterns **BEGIN** and **END**, which never match any input record, are not expressions and cannot appear inside boolean patterns.

A regexp constant as a pattern is also a special case of an expression pattern. **/foo/** as an expression has the value one if 'foo' appears in the current input record; thus, as a pattern, **/foo/** matches any record containing 'foo'.

5.6.4 Specifying Record Ranges with Patterns

A **range pattern** is made of two patterns separated by a comma, of the form '*begpat*, *endpat*'. It matches ranges of consecutive input records. The first pattern, *begpat*, controls where the range begins, and the second one, *endpat*, controls where it ends. For example,

```
awk '$1 == "on", $1 == "off"'
```

prints every record between 'on'/'off' pairs, inclusive.

A range pattern starts out by matching *begpat* against every input record; when a record matches **begpat**, the range pattern becomes **turned on**. The range pattern matches this record. As long as it stays turned on, it automatically matches every input record read. It also matches *endpat* against every input record; when that succeeds, the range pattern is turned off again for the following record. Then it goes back to checking **begpat** against each record.

The record that turns on the range pattern and the one that turns it off both match the range pattern. If you don't want to operate on these records, you can write **if** statements in the rule's action to distinguish them from the records you are interested in.

It is possible for a pattern to be turned both on and off by the same record, if the record satisfies

both conditions. Then the action is executed for just that record.

For example, suppose you have text between two identical markers (say the '%' symbol) that you wish to ignore. You might try to combine a range pattern that describes the delimited text with the **next** statement, which causes **awk** to skip any further processing of the current record and start over again with the next input record. Such a program would like this:

```
/^%$/,/^%$/    { next }

               { print }
```

This program fails because the range pattern is both turned on and turned off by the first line with just a '%' on it. To accomplish this task, you must write the program this way, using a flag:

```
/^%$/     { skip = ! skip; next }

skip == 1 { next } # skip lines with 'skip' set
```

Note that in a range pattern, the ',' has the lowest precedence (is evaluated last) of all the operators. Thus, for example, the following program attempts to combine a range pattern with another, simpler test.

```
echo Yes | awk '/1/,/2/ || /Yes/'
```

The author of this program intended it to mean '(/1/,/2/) || /Yes/'. However, **awk** interprets this as '/1/, (/2/ || /Yes/)'. This cannot be changed or worked around; range patterns do not combine with other patterns.

5.6.5 The BEGIN and END Special Patterns

BEGIN and **END** are special patterns. They are not used to match input records. Rather, they supply start-up or clean-up actions for your **awk** script.

- Using BEGIN/END: How and why to use BEGIN/END rules.
- I/O And BEGIN/END: I/O issues in BEGIN/END rules.

Startup and Cleanup Actions

A **BEGIN** rule is executed, once, before the first input record has been read. An **END** rule is executed, once, after all the input has been read. For example:

```
$ awk '

> BEGIN { print "Analysis of \"foo\"" }

> /foo/ { ++n }

> END   { print "\"foo\" appears " n " times." }' BBS-list
```

-|Analysis of "foo"

-| "foo" appears 4 times.

This program finds the number of records in the input file 'BBS-list' that contain the string 'foo'. The **BEGIN** rule prints a title for the report. There is no need to use the **BEGIN** rule to initialize the counter **n** to zero, as **awk** does this automatically.

The second rule increments the variable **n** every time a record containing the pattern 'foo' is read. The **END** rule prints the value of **n** at the end of the run.

The special patterns **BEGIN** and **END** cannot be used in ranges or with boolean operators (indeed, they cannot be used with any operators).

An **awk** program may have multiple **BEGIN** and/or **END** rules. They are executed in the order they appear, all the **BEGIN** rules at start-up and all the **END** rules at termination. **BEGIN** and **END** rules may be intermixed with other rules.

Multiple **BEGIN** and **END** rules are useful for writing library functions, since each library file can have its own **BEGIN** and/or **END** rule to do its own initialization and/or cleanup.

If an **awk** program only has a **BEGIN** rule, and no other rules, then the program exits after the **BEGIN** rule has been run.

BEGIN and **END** rules must have actions; there is no default action for these rules since there is no current record when they run.

Input/Output from BEGIN and END Rules

The Empty Pattern

An empty (i.e. non-existent) pattern is considered to match *every* input record. For example, the program:

```
awk '{ print $1 }' BBS-list
```

prints the first field of every record.

Overview of Actions

An **awk** program or script consists of a series of rules and function definitions, interspersed.

A rule contains a pattern and an action, either of which (but not both) may be omitted. The purpose of the **action** is to tell **awk** what to do once a match for the pattern is found. Thus, in outline, an **awk** program generally looks like this:

```
[pattern] [{ action }]
[pattern] [{ action }]
...
function name(args) { ... }
```

An action consists of one or more **awk statements**, enclosed in curly braces ('{' and '}'). Each statement specifies one thing to be done. The statements are separated by newlines or semicolons.

The curly braces around an action must be used even if the action contains only one statement, or even if it contains no statements at all. However, if you omit the action entirely, omit the curly braces as well. An omitted action is equivalent to '{ print $0 }'.

```
/foo/ { } # match foo, do nothing - empty action

/foo/   # match foo, print the record - omitted action
```

Here are the kinds of statements supported in **awk**:

- Expressions, which can call functions or assign values to variables. Executing this kind of statement simply computes the value of the expression. This is useful when the expression has side effects.
- Control statements, which specify the control flow of **awk** programs. The **awk** language gives you C-like constructs (**if**, **for**, **while**, and **do**) as well as a few special ones.
- Compound statements, which consist of one or more statements enclosed in curly braces. A compound statement is used in order to put several statements together in the body of an **if**, **while**, **do** or **for** statement.
- Input statements, using the **getline** command, the **next** statement, and the **nextfile** statement.
- Output statements, **print** and **printf**.
- Deletion statements, for deleting array elements.

5.7 ARRAYS IN AWK

An **array** is a table of values, called **elements**. The elements of an array are distinguished by their indices. **Indices** may be either numbers or strings. **awk** maintains a single set of names that may be used for naming variables, arrays and functions. Thus, you cannot have a variable and an array with the same name in the same **awk** program.

- Array Intro: Introduction to Arrays
- Reference to Elements: How to examine one element of an array.
- Assigning Elements: How to change an element of an array.
- Array Example: Basic Example of an Array
- Scanning an Array: A variation of the **for** statement. It loops through the indices of an array's existing elements.
- Delete: The **delete** statement removes an element from an array.

5.7.1 Introduction to Arrays

The **awk** language provides one-dimensional **arrays** for storing groups of related strings or numbers.

Every **awk** array must have a name. Array names have the same syntax as variable names; any valid variable name would also be a valid array name. But you cannot use one name in both ways (as an array and as a variable) in one **awk** program.

Arrays in **awk** superficially resemble arrays in other programming languages; but there are fundamental differences. In **awk**, you don't need to specify the size of an array before you start to use it. Additionally, any number or string in **awk** may be used as an array index, not just consecutive integers.

In most other languages, you have to **declare** an array and specify how many elements or components it contains. In such languages, the declaration causes a contiguous block of memory to be allocated for that many elements. An index in the array usually must be a positive integer; for example, the index zero specifies the first element in the array, which is actually stored at the beginning of the block of memory. Index one specifies the second element, which is stored in memory right after the first element, and so on. It is impossible to add more elements to the array, because it has room for only as many elements as you declared. (Some languages allow arbitrary starting and ending indices, e.g., '15 .. 27', but the size of the array is still fixed when the array is declared.)

A contiguous array of four elements might look like this, conceptually, if the element values are eight, **"foo"**, **""** and 30:

Only the values are stored; the indices are implicit from the order of the values. Eight is the value at index zero, because eight appears in the position with zero elements before it.

Arrays in **awk** are different: they are **associative**. This means that each array is a collection of pairs: an index, and its corresponding array element value:

Element 4 Value 30

Element 2 Value "foo"

Element 1 Value 8

Element 3 Value ""

We have shown the pairs in jumbled order because their order is irrelevant.

One advantage of associative arrays is that new pairs can be added at any time. For example, suppose we add to the above array a tenth element whose value is **"number ten"**. The result is this:

Element 10 Value "number ten"

Element 4 Value 30

Element 2 Value "foo"

Element 1 Value 8

Element 3 Value ""

Now the array is **sparse**, which just means some indices are missing: it has elements 1—4 and 10, but doesn't have elements 5, 6, 7, 8, or 9.

Another consequence of associative arrays is that the indices don't have to be positive integers. Any number, or even a string, can be an index. For example, here is an array which translates words from English into French:

Element "dog" Value "chien"

Element "cat" Value "chat"

Element "one" Value "un"

Element 1 Value "un"

Here we decided to translate the number one in both spelled-out and numeric form—thus illustrating that a single array can have both numbers and strings as indices. (In fact, array subscripts are always

strings; this is discussed in more detail in section Using Numbers to Subscript Arrays.)

When **awk** creates an array for you, e.g., with the **split** built-in function, that array's indices are consecutive integers starting at one.

5.7.2 Referring to an Array Element

The principal way of using an array is to refer to one of its elements. An array reference is an expression which looks like this:

array[*index*]

Here, *array* is the name of an array. The expression *index* is the index of the element of the array that you want.

The value of the array reference is the current value of that array element. For example, **foo[4.3]** is an expression for the element of array **foo** at index '4.3'.

If you refer to an array element that has no recorded value, the value of the reference is **""**, the null string. This includes elements to which you have not assigned any value, and elements that have been deleted. Such a reference automatically creates that array element, with the null string as its value. (In some cases, this is unfortunate, because it might waste memory inside **awk**.)

You can find out if an element exists in an array at a certain index with the expression:

index in *array*

This expression tests whether or not the particular index exists, without the side effect of creating that element if it is not present. The expression has the value one (true) if *array*[*index*] exists, and zero (false) if it does not exist.

For example, to test whether the array **frequencies** contains the index '2', you could write this statement:

```
if (2 in frequencies)
    print "Subscript 2 is present."
```

Note that this is *not* a test of whether or not the array **frequencies** contains an element whose *value* is two. (There is no way to do that except to scan all the elements.) Also, this *does not* create **frequencies[2]**, while the following (incorrect) alternative would do so:

```
if (frequencies[2] != "")
    print "Subscript 2 is present."
```

5.7.3 Assigning Array Elements

Array elements are lvalues: they can be assigned values just like **awk** variables:

array[*subscript*] = *value*

Here *array* is the name of your array. The expression *subscript* is the index of the element of the

array that you want to assign a value. The expression *value* is the value you are assigning to that element of the array.

Basic Array Example

The following program takes a list of lines, each beginning with a line number, and prints them out in order of line number. The line numbers are not in order, however, when they are first read: they are scrambled. This program sorts the lines by making an array using the line numbers as subscripts. It then prints out the lines in sorted order of their numbers. It is a very simple program, and gets confused if it encounters repeated numbers, gaps, or lines that don't begin with a number.

```
{
 if ($1 > max)
  max = $1
 arr[$1] = $0
}

END {
 for (x = 1; x <= max; x++)
  print arr[x]
}
```

The first rule keeps track of the largest line number seen so far; it also stores each line into the array **arr**, at an index that is the line's number.

The second rule runs after all the input has been read, to print out all the lines.

When this program is run with the following input:

```
5 I am the Five man
2 Who are you? The new number two!
4 . . . And four on the floor
1 Who is number one?
3 I three you.
```

its output is this:

```
1 Who is number one?
2 Who are you? The new number two!
3 I three you.
4 . . . And four on the floor
```

```
5 I am the Five man
```

If a line number is repeated, the last line with a given number overrides the others.

Gaps in the line numbers can be handled with an easy improvement to the program's **END** rule:

```
END {
 for (x = 1; x <= max; x++)
  if (x in arr)
   print arr[x]
}
```

5.7.4 Scanning All Elements of an Array

In programs that use arrays, you often need a loop that executes once for each element of an array. In other languages, where arrays are contiguous and indices are limited to positive integers, this is easy: you can find all the valid indices by counting from the lowest index up to the highest. This technique won't do the job in **awk**, since any number or string can be an array index. So **awk** has a special kind of **for** statement for scanning an array:

```
for (var in array)
 body
```

This loop executes *body* once for each index in *array* that your program has previously used, with the variable *var* set to that index.

Here is a program that uses this form of the **for** statement. The first rule scans the input records and notes which words appear (at least once) in the input, by storing a one into the array **used** with the word as index. The second rule scans the elements of **used** to find all the distinct words that appear in the input. It prints each word that is more than 10 characters long, and also prints the number of such words. See section Built-in Functions for String Manipulation, for more information on the built-in function **length**.

```
# Record a 1 for each word that is used at least once.
{
  for (i = 1; i <= NF; i++)
    used[$i] = 1
}
# Find number of distinct words more than 10 characters long.
END {
  for (x in used)
```

```
if (length(x) > 10) {
    ++num_long_words
    print x
  }
print num_long_words, "words longer than 10 characters"
  }
```

See section Generating Word Usage Counts, for a more detailed example of this type.

The order in which elements of the array are accessed by this statement is determined by the internal arrangement of the array elements within **awk** and cannot be controlled or changed. This can lead to problems if new elements are added to *array* by statements in the loop body; you cannot predict whether or not the **for** loop will reach them. Similarly, changing *var* inside the loop may produce strange results. It is best to avoid such things.

5.7.5 The delete Statement

You can remove an individual element of an array using the **delete** statement:

```
delete array[index]
```

Once you have deleted an array element, you can no longer obtain any value the element once had. It is as if you had never referred to it and had never given it any value.

Here is an example of deleting elements in an array:

```
for (i in frequencies)
    delete frequencies[i]
```

This example removes all the elements from the array **frequencies**.

If you delete an element, a subsequent **for** statement to scan the array will not report that element, and the **in** operator to check for the presence of that element will return zero (i.e. false):

```
delete foo[4]
if (4 in foo)
  print "This will never be printed"
```

It is important to note that deleting an element is *not* the same as assigning it a null value (the empty string, ""):

```
foo[4] = ""
if (4 in foo)
```

print "This is printed, even though foo[4] is empty"

It is not an error to delete an element that does not exist.

You can delete all the elements of an array with a single statement, by leaving off the subscript in the **delete** statement.

delete array

This ability is a **gawk** extension; it is not available in compatibility mode.

Using this version of the **delete** statement is about three times more efficient than the equivalent loop that deletes each element one at a time.

5.8 QUESTION

1. What is the use of Awk?
2. Discuss input and output in Awk?
3. Discuss different types of Patterns?
4. What is a Regular expression?

□□□□

UNIT 6 : INTRODUCTION TO ADVANCED OPERATING SYSTEMS

Structure of the Unit

6.0 OBJECTIVES

After completing this unit you will learn

- Define Advanced Operating System.
- Types of Advanced Operating system.
- Database Operating System.
- Multiprocessor Operating System.
- Real time Operating system

6.1 INTRODUCTION

The operating system design was mainly focused on the conventional operating systems, which run on stand-alone computers with single processors. The improvement integrated circuit and computer communication technologies over the last two decades have made unprecedented interest in multicomputer systems and have resulted in the development of a variety of computer architectures, like, shared memory multiprocessors to distributed memory distributed systems. These multicomputer systems were prompted by the need for high-speed computing that conventional single processor systems were not capable to provide.

6.2 NEED FOR ADVANCED OPERATING SYSTEMS

Multiprocessor systems and distributed systems have many new features which are not present in traditional single-processor systems. These features render the design of operating systems for these multicomputer systems extremely complicated and require that nontrivial design issues be addressed.

Due to their relative newness and massive design complexity, operating systems for this multi computer are referred to as advanced or modern operating systems.

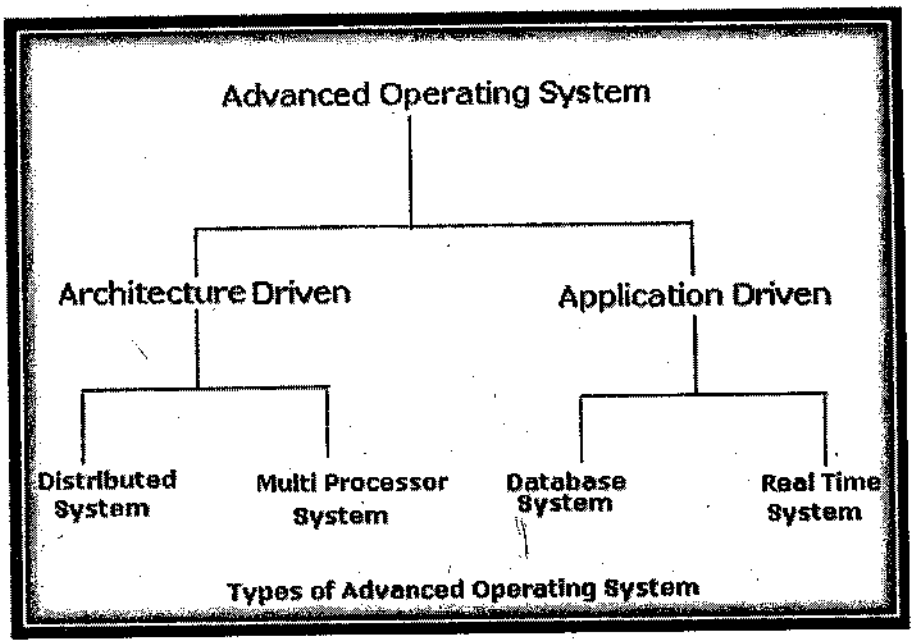
An advanced operating system not only harnesses the power of a multicomputer system; it also provides a high-level coherent view of the system; a user views a multicomputer system as a single monolithic powerful machine.

The high demand and popularity of multicomputer systems, advanced operating systems have gained substantial importance and a substantial amount of research has been done on them over the last few years.

6.3 TYPES OF ADVANCED OPERATING SYSTEMS

Below figure gives a classification of advanced operating systems. The impetus for advanced

operating systems has come from two directions. First, it has come from advanced in the architecture of multicomputer systems and is now driven by a wide variety of high-speed architectures. Hardware design of extremely fast parallel and distributed systems is fairly well understood. These architectures offer great potential for speed up but they also present a substantial challenge to operating system designers. Operating system designs for two types of multicomputer systems, namely, multiprocessor systems and distributed computing systems, have been well-studied.



A second class of advanced operating systems is driven by applications. There are a number of important applications that require special operating system support, as a requirement as well as for efficiency. General purpose operating systems are too broad in nature and inefficient and fail to provide adequate support for such applications. Two specific applications, namely, database systems and real-time systems, have received considerable attention in the past and the operating system issues for these systems have been extensively examined. Other applications include graphics systems, surveillance, and process control.

6.3.1 Distributed Operating System

Distributed operating systems are operating systems for a network of independent computers connected by a communication network. A distributed operating system controls and manages the hardware and software resources of a distributed system such that its users view the entire system as a powerful massive computer system. When a program is executed in a distributed system, the user is not aware of where the program is executed or of the location of the resources accessed.

The basic issues in the design of a distributed operating system are the same as in a conventional operating system, that is, process synchronization, deadlocks, scheduling, file systems, inter process communication, memory and buffer management, failure recovery, etc. However, several idiosyncrasies of a distributed system, namely, the lack of both shared memory and a physical global clock, and unpredictable communication delays, make the design of distributed operating systems much more difficult.

An operating system is a program that manages the resources of a computer system and provides users with a friendly interface to the system. A distributed operating system extends the concepts of re-

source management and user friendly interface for shared *memory* computers a step further, encompassing a distributed computing system consisting of several autonomous computers connected by a communication network.

A distributed operating system appears to its users as a centralized operating system for a single machine, but it runs on multiple-independent computers. An identical copy of the operating system may run at every computer. On the other hand, some computers in the system that serve a special purpose might run an extended version of the operating system.

Issues in distributed operating systems

Some important issues that arise in the design of a distributed operating system include the unavailability of up-to-date global knowledge, naming, scalability, compatibility, process synchronization, resource management, security, and structuring of the operating system.

Global knowledge

In the case of shared memory computer systems, the up-to-date state of all the processes and resources, in other words, the global (entire) state of the system, is completely and accurately known. Hence, the potentially problematic issues that arise in the design of these systems are well understood and efficient solutions to them exist. in distributed computing systems, these same issues take on new dimensions and their solutions become much more complex for the following reasons. Due to the unavailability of a global *memory* and a global clock, and due to unpredictable message delays, it is practically impossible for a computer to collect up-to-date information about the global state of the distributed computing system . Therefore, a fundamental problem in the design of a distributed operating system is to determine efficient techniques to implement decentralized system wide control, where a computer does not know the current and complete status of the global state. Another significant problem, given the absence of a global clock, is the question of how to order all the events that occur at different times at different computers present in the system.

Naming

Names are used to refer to objects. Objects that can be named in computer systems include computers, printers, services, files, and users. An example of a service is a name service. A name service maps a logical name into a physical address by making use of a table lookup, an algorithm, or through a combination of the two. in the implementation of a table lookup, tables or directories that store names and their physical addresses are used for mapping names to their addresses. in distributed systems, the directories may be replicated and stored at many different locations to overcome a single point of failure as well as to increase the availability of the name service.

Scalability

Systems generally grow with time. The techniques used in designing a system should not result in system unavailability or degraded performance when growth occurs. For example, broadcast based protocols work well for small systems (systems having a small number of computers) but not for large systems. Consider a distributed file system that locates files by broadcasting queries. Under this file system, every computer in the distributed system is subjected to message handling overhead, irrespective of whether it has the requested file or not. As the number of users increase and the system gets larger, the number of file location queries will increase and the overhead will grow larger as well, hurting the performance of every computer. In general, any design approach in which the requirement of a scarce resource (such as storage, communication bandwidth, and manpower) increases linearly with the number of computers in the system, is likely to be too costly to implement.

Compatibility

Compatibility refers to the notion of interoperability among the resources in a system. The three different levels of compatibility that exist in distributed systems are the binary level, the execution level, and the protocol level

In a system that is compatible at the binary level, all processors execute the same binary instruction repertoire, even though the processors may differ in performance and in input-output. A significant advantage of binary level compatibility is that it is easier for system development, as the code for many functions provided by the system programs directly depend on the underlying machine level instructions. On the other hand, the distributed system cannot include computers with different architectures from the same or different vendors. Because of this major restriction, binary compatibility is rarely supported in large distributed systems.

Execution level compatibility is said to exist in a distributed system if the same source code can be compiled and executed properly on any computer in the system.

Protocol level compatibility is the least restrictive form of compatibility. It achieves interoperability by requiring all system components to support a common set of protocols. A significant advantage of protocol level compatibility is that individual computers can run different operating systems while not sacrificing their interoperability

Process Synchronization

The synchronization of processes in distributed systems is difficult because of the unavailability of shared memory. A distributed operating system has to synchronize processes running at different computers when they try to concurrently access a shared resource, such as a file directory. For correctness, it is necessary that the shared resource be accessed by a single process at a time. This problem is known as the mutual exclusion problem, wherein concurrent access to a shared resource by several uncoordinated user requests must be serialized to secure the integrity of the shared resource

In distributed systems, processes can request resources and release resources in any order that may not be known. If the sequence of the allocation of resources to processes is not controlled in such environments, deadlocks may occur. It is important that deadlocks are detected and resolved as soon as possible, otherwise, system performance can degrade severely.

Resource Management

Resource management in distributed operating systems is concerned with making both local and remote resources available to users in an effective manner. Users of a distributed operating system should be able to access remote resources as easily as they can access local resources. In other words, the specific location of resources should be hidden from the users. The resources of a distributed system are made available to users in the various ways like data migration, computation migration, and distributed scheduling.

Security

The security of a system is the responsibility of its operating system. Two issues that must be considered in the design of security for computer systems are *authentication* and *authorization*. Authentication is the process of guaranteeing that an entity is what it claims to be. Authorization is the process of deciding what privileges an entity has and making only these privileges available.

Structuring

The structure of an operating system defines how various parts of the operating system are organized.

6.3.2 Multiprocessor Operating System

A typical multiprocessor system consists of a set of processors that share a set of physical memory blocks over an interconnection network. Thus, a multiprocessor system is a tightly coupled system where processors share an address space. A multiprocessor operating system controls and manages the hardware and software resources such that users view the entire system as a powerful uniprocessor system; a user is not aware of the presence of multiple processors and the interconnection-network.

The basic issues in the design of a multiprocessor operating system are the same as in a conventional operating system. However, the issues of process synchronization, task scheduling, memory management, and protection and security, become more complex because the main memory is shared by many physical processors. A multiprocessor operating system must be able to support the concurrent execution of multiple tasks and must practically utilize the power of multiple processors to increase performance.

Structure Of Multiprocessor System

Based upon the nature of the control structure and organization of Multiprocessor Operating System, there are three basic classes of multiprocessor operating systems: separate supervisor, master-slave, and symmetric.

Separate Supervisor Configuration

In the separate supervisor configuration all processors have their own copy of the kernel, supervisor, and data structure. There are some common data structures for the interaction among processors. The access to which is protected by using some synchronization mechanism. Each processor has its own I/O devices and file system. There is very little coupling among processors and each processor acts as an autonomous, independent system. Therefore, it is difficult to perform parallel execution of a single task Also, this configuration is inefficient because the supervisor/kernel data structure code is replicated for each processor. This configuration, however, degrades gracefully in the face of processor failures because there is very little coupling among processors.

Master Slave Configuration

In the master-slave configuration, one processor, called the master, monitors the status and assigns work to all other processors, the slaves. Slaves are treated as a schedulable pool of resources by the master. Such an operating system is simple because it runs only on the master processor. Since the operating system is executed by a single processor, it is efficient and its implementation is easy. The master-slave configuration permits the parallel execution of a single task, where a task can be broken into several subtasks and the subtasks can be scheduled on multiple processors concurrently.

Symmetric Configuration

In the symmetric configuration, all processors are autonomous and are treated equally. There is one copy of the supervisor or kernel that can be executed by all processors concurrently. However, concurrent access to the shared data structures of the supervisor needs to be controlled in order to maintain their integrity. The simplest way to achieve this is to treat the entire operating system as a critical section and allow only one processor to execute the operating system at one time.

The symmetric configuration is the most flexible and versatile of all the configurations. It permits

the parallel execution of a single task. It degrades gracefully under failures and makes very efficient use of resources. However, it is the most difficult configuration to design and implement

6.3.3 Database Operating System

Database systems place special requirements on operating systems. These requirements have their roots in the specific environment that database systems support. A database system must support the concept of a transaction, operations to store, retrieve, and manipulate a large volume of data efficiently; primitives for concurrency control, and system failure recovery. To store temporary data and data retrieved from secondary storage, it must have a buffer management scheme.

General purpose operating systems are designed to provide the users with facilities for general purpose software development, testing and execution, and file manipulation. They support the concepts of a process (by providing mechanisms for process creation, deletion, synchronization, scheduling, and inter process communication), a virtual memory system (by providing mechanisms for address translation, buffer management, page replacement, etc.), a file system (by providing mechanisms for file storage, manipulation, and protection), and general purpose library routines. CPU scheduling, buffer management, memory management, I/O services, protection, file system management, etc. are all designed to provide a general purpose computing environment.

Operating systems support an abstraction of files, which are variable size arrays of characters. This abstraction is suitable for language processors, text proces,sors, editors, etc., but it is not suitable for database systems. If a database system requires objects with rich, complex structures, it has to build them on top of the file abstraction. Instead, for database systems, it is desirable that the operating system support complex objects on which the database system can create structured files. The operating system should support primitives for creation, manipulation, and navigation through complex objects.

most operating systems provide support for locking files. However, database systems require support for locking at a finer granularity, such as a page, a record or a byte. Finer, variable size locks are essential in database systems for efficiency reasons .

Requirements of a Database Operating System

The requirements that a database system places on an operating system to meet its goals. These requirements primarily arise due to the following features of database systems.

1. A database system must support the concept of a transaction, which is the unit of consistency and reliability.
2. Database systems are characterized by the existence of huge, persistent, complex data that are shared among its users.
3. Nontrivial integrity constraints must be satisfied by the shared data of the database system. A database system is consistent if its data satisfy a set of integrity constraints. We next discuss the requirements of a database system.

Transaction Management

A user accesses a database system by executing a program, called a transaction. Informally, a transaction consists of a sequence of read and writes operations on the database and is the unit of user interaction with the database system.

Transaction is a unit of consistency in the sense that when a transaction is executed alone in a database system, it maintains database consistency. A database system must ensure that database consis-

tency is maintained even when several transactions are running concurrently.

The database system should also ensure that a transaction is either executed completely or is not executed at all . A partially executed transaction may leave the database in an inconsistent state. In addition, in the face of a system failure, the database system must guarantee that either the actions of all partially executed transactions are undone or all partially executed transactions are run to completion.

In database systems, a user runs a transaction by indicating its beginning and end to the system, thereby ignoring the problems associated with concurrent transaction execution and system failures. It is the responsibility of the database system to maintain database system consistency and transaction atomicity in the presence of concurrent transaction execution and system failures. The operating system should support mechanisms to facilitate the implementation of the following properties in transactions: concurrency control, atomic commit, and failure recovery.

Support For Complex Persistent Data

Database systems manage a large volume of complex, persistent data. Traditional operating systems support persistent data in the form of files. As discussed before, files are not suitable for the direct creation and manipulation of complex data.

Database operating systems must support definition, efficient manipulation, and efficient storage on secondary devices of files with complex structures. In database systems, a table is a collection of row and column. An environment to build database systems must provide facilities to define and manipulate files of records of any arbitrary structures. Database systems are dominated by heavy I/O accesses and I/O traffic is usually a bottleneck. I/O efficiency can be improved by judiciously structuring blocks of a file on a disk so that disk-head movement is reduced while accessing blocks of a file. Thus, an operating system should organize a file on secondary storage such that neighboring pages of a file are stored next to each other on the disk.

Buffer Management

Data of a database system are stored on a secondary storage and database systems maintain buffers in the main memory to cache the needed data. Data on secondary storage and the buffer in main memory are divided into equal size pages and data pages are brought into the buffer as and when needed for computation.

When a transaction accesses a data page, the database system looks into the buffer to check if the page is present in it. If not, a page fault occurs and the page is brought from the secondary storage into the buffer. If the buffer is full, a page in the buffer must be swapped out to secondary storage.

Therefore, a database system requires mechanisms to perform the following operations efficiently.

1. Search the buffer to see if a page is present.
2. Select a page for replacement that optimizes the cache hit ratio and locate and retrieve the needed data page from secondary storage. In addition, for higher reliability, a database system must be able to flush a selected set of pages in the buffer to secondary storage.

Real-Time Operating Systems

Real-time systems also place special requirements on operating systems, which have their roots in the specific application that the real-time system is supporting. A distinct feature of realtime systems is that jobs have completion deadlines. A job should be completed before its deadline to be of use (in soft real-time systems) or to avert a disaster (in hard real-time systems). The major issue in the design of real-time

operating systems is the scheduling of jobs in such a way that a maximum number of jobs satisfy their deadlines. Other issues include designing languages and primitives to effectively prepare and execute a job schedule.

OPERATING SYSTEM DESIGN ISSUES

A multiprocessor operating system encompasses all the functional capabilities of the operating system of a multi programmed uniprocessor system. However, the design of a multiprocessor operating system is complicated because it must fulfill the following requirements. A multiprocessor operating system must be able to support concurrent task execution, it should be able to exploit the power of multiple processors, it should fail gracefully, and it should work correctly despite physical concurrency in the execution of processes. The design of multiprocessor operating systems involves the following major issues:

Threads

The effectiveness of parallel computing depends greatly on the performance of the primitives that are used to express and control parallelism within an application. It has been recognized that traditional processes impose too much overhead for context switching. In light of this, threads have been widely utilized in, recent systems to run applications concurrently on many processors.

Process Synchronization

In a multiprocessor operating system, disabling interrupts is not sufficient to synchronize concurrent access to shared data. A more elaborate mechanism that is based on shared variables is needed. Moreover, a synchronization mechanism must be carefully designed so that it is efficient; otherwise, it could result in significant performance penalty.

Processor Scheduling

To ensure the efficient use of its hardware, a multiprocessor operating system must be able to utilize the processors effectively in executing the tasks. A multiprocessor operating system, in cooperation with the compiler, should be able to detect and exploit the parallelism in the tasks being executed.

Memory Management

The design of virtual memory is complicated because the main memory is shared by many processors. The operating system must maintain a separate map table for each processor for address translation. When several processors share a page or segment, the operating system must enforce the consistency of their entries in respective map tables. Moreover, efficient page replacement becomes a complex issue.

Reliability and Fault Tolerance

The performance of a multiprocessor system must be able to degrade gracefully in the event of failures. Thus, a multiprocessor operating system must 'provide reconfiguration schemes to restructure the system in the face of failures to ensure graceful degradation.

Other issues include protection and inter process communication. Protection deals with the design of mechanisms that prevent unauthorized access to resources. Inter process communication in an operating system calls for a support of a variety of models for communication between processes.

6.4 SUMMARY

* Distributed operating systems are operating systems for a network of independent computers connected by a communication network. A distributed operating system controls and manages the hard-

ware and software resources of a distributed system such that its users view the entire system as a powerful massive computer system.

* Some important issues that arise in the design of a distributed operating system include the unavail ability of up-to-date global knowledge, naming, scalability, compatibility, process synchronization, resource management, security, and structuring of the operating system.

* A multiprocessor operating system controls and manages the hardware and software resources such that users view the entire system as a powerful uniprocessor system; a user is not aware of the presence of multiple processors and the interconnection-network.

* Database systems place special requirements on operating systems. These requirements have their roots in the specific environment that database systems support. A database system must support the concept of a transaction, operations to store, retrieve, and manipulate a large volume of data effi ciently.

6.5 QUESTIONS

1. Define Distributed OS.
2. Explain various features of Distributed OS.
3. List the various design issues for distributed OS.
4. Define real time Operating System.
5. Define Database Operating System, list various types of database OS.

□□□□

UNIT 7 : DISTRIBUTED OPERATING SYSTEMS

Structure of the Unit

7.0 OBJECTIVES

After completing this unit you will learn

- Define Distributed Operating System.
- Designing issues in Distributed Operating System
- Client server computing model
- Communication primitives (message passing model and RPC)

7.1 INTRODUCTION

The term distributed system consists of several computers that do not share a memory or a clock. The computers communicate with each other by exchanging messages over a communication network and each computer has its own memory and runs its own operating system. The resources owned and controlled by a computer are called local to it, while the resources owned and controlled by other computers and those that can only be accessed through the network are said to be remote.

Typically, accessing remote resources is more expensive than accessing local resources because of the communication delays that occur in the network and the CPU overhead incurred to process communication protocols Based on computer.

The development of distributed systems was the availability of powerful microprocessors at low

cost as well as significant advances in communication technology.

The availability of powerful yet cheap microprocessors led to the development of powerful workstations that satisfy a single user's needs. These powerful stand-alone workstations satisfy user need by providing such things as bit-mapped displays and visual interfaces, which traditional time-sharing mainframe systems do not support.

When a group of people work together, there is generally a need to communicate with each other, to share data, and to share costly resources (like high quality printers, disk drives, etc.). This requires interconnecting computers and resources. Designing such systems became feasible with the availability of cheap and powerful microprocessors, and advances in communication technology.

When a few powerful workstations are interconnected and can communicate with each other, the total computing power available in such a system can be enormous. Such a system generally only costs tens of thousands of dollars. On the other hand, if one tries to obtain a single machine with the computing power equal to that of a network of workstations, the cost can be as high as a few million dollars. Hence, the main advantage of distributed systems is that they have a decisive price/performance advantage over more traditional time-sharing systems.

Other significant advantages of distributed systems over traditional time-sharing systems are as follows:

Resource sharing

Since a computer can request a service from another computer by sending an appropriate request to it over the communication network, hardware and software resources can be shared among computers. For example, a printer, a compiler, a text processor, or a database at a computer can be shared with remote computers.

Enhanced performance

A distributed computing system is capable of providing rapid response time and higher system throughput. This ability is mainly due to the fact that many tasks can be concurrently executed at different computers. Moreover, distributed systems can employ a load distributing technique to improve response time. In load distributing, tasks at heavily loaded computers are transferred to lightly loaded computers, thereby reducing the time tasks wait before receiving service.

Improved reliability and availability

A distributed computing system provides improved reliability and availability because a few components of the system can fail without affecting the availability of the rest of the system. the replication of data (e.g., files and directories) and services, distributed systems can be made fault tolerant. Services are processes that provide functionality such as a file service provides file system management and a mail service provides an electronic mail facility.

Modular expandability

In Distributed computing systems new hardware and software resources can be easily added without replacing the existing resources.

Distributed Operating System

An operating system is a program that manages the resources of a computer system and provides users with a friendly interface to the system. A distributed operating system extends the concepts of resource management and user friendly interface for shared memory computers a step further, encompassing a distributed computing system consisting of several autonomous computers connected by a communication network.

A distributed operating system appears to its users as a centralized operating system for a single machine, but it runs on multiple-independent computers. An identical copy of the operating system may run at every computer. On the other hand, some computers in the system that serve a special purpose might run an extended version of the operating system.

The key concept is transparency is the use of multiple processors and the accessing of remote data should be invisible to the user. A user simply submits a job to the distributed operating system through a computer. The distributed operating system performs distributed execution of the job. The user does not know on what computers the job was executed, on what computers the files needed for execution were stored, or how the communication and synchronization among different computers were carried out.

7.2 ISSUES IN DISTRIBUTED OPERATING SYSTEMS

Some important issues that arise in the design of a distributed operating system include the unavailability of up-to-date global knowledge, naming, scalability, compatibility, process synchronization, resource management, security, and structuring of the operating system.

7.2.1 Global knowledge

In the case of shared memory computer systems, the up-to-date state of all the processes and resources, in other words, the global (entire) state of the system, is completely and accurately known. Hence, the potentially problematic issues that arise in the design of these systems are well understood and efficient solutions to them exist. in distributed computing systems, these same issues take on new dimensions and their solutions become much more complex for the following reasons. Due to the unavailability of a global memory and a global clock, and due to unpredictable message delays, it is practically impossible for a computer to collect up-to-date information about the global state of the distributed computing system.

Therefore, a fundamental problem in the design of a distributed operating system is to determine efficient techniques to implement decentralized system wide control, where a computer does not know the current and complete status of the global state. Another significant problem, given the absence of a global clock, is the question of how to order all the events that occur at different times at different computers present in the system.

7.2.2 Naming

Names are used to refer to objects. Objects that can be named in computer systems include computers, printers, services, files, and users. An example of a service is a name service. A name service maps a logical name into a physical address by making use of a table lookup, an algorithm, or through a combination of the two. in the implementation of a table lookup, tables or directories that store names and their physical addresses are used for mapping names to their addresses. in distributed systems, the directories may be replicated and stored at many different locations to overcome a single point of failure as well as to increase the availability of the name service.

7.2.3 Scalability

Systems generally grow with time. The techniques used in designing a system should not result in system unavailability or degraded performance when growth occurs. For example, broadcast based protocols work well for small systems (systems having a small number of computers) but not for large systems. Consider a distributed file system that locates files by broadcasting queries. Under this file system, every computer in the distributed system is subjected to message handling overhead, irrespective of whether it has the requested file or not. As the number of users increase and the system gets larger, the number of file location queries will increase and the overhead will grow larger as well, hurting the performance of every computer. In general, any design approach in which the requirement of a scarce resource (such as storage, communication bandwidth, and manpower) increases linearly with the number of computers in the system, is likely to be too costly to implement.

7.2.4 Compatibility

Compatibility refers to the notion of interoperability among the resources in a system. The three different levels of compatibility that exist in distributed systems are the binary level, the execution level, and the protocol level

In a system that is compatible at the binary level, all processors execute the same binary instruction repertoire, even though the processors may differ in performance and in input-output. A significant advantage of binary level compatibility is that it is easier for system development, as the code for many functions provided by the system programs directly depend on the underlying machine level instructions. On the other hand, the distributed system cannot include computers with different architectures from the same or different vendors. Because of this major restriction, binary compatibility is rarely supported in large distributed systems.

Execution level compatibility is said to exist in a distributed system if the same source code can be compiled and executed properly on any computer in the system.

Protocol level compatibility is the least restrictive form of compatibility. It achieves interoperability by requiring all system components to support a common set of protocols. A significant advantage of protocol level compatibility is that individual computers can run different operating systems while not sacrificing their interoperability

7.2.5 Process synchronization

The synchronization of processes in distributed systems is difficult because of the unavailability of shared memory. A distributed operating system has to synchronize processes running at different computers when they try to concurrently access a shared resource, such as a file directory. For correctness, it is necessary that the shared resource be accessed by a single process at a time. This problem is known as the mutual exclusion problem, wherein concurrent access to a shared resource by several uncoordinated user requests must be serialized to secure the integrity of the shared resource

In distributed systems, processes can request resources and release resources in any order that may not be known. If the sequence of the allocation of resources to processes is not controlled in such environments, deadlocks may occur. It is important that deadlocks are detected and resolved as soon as possible, otherwise, system performance can degrade severely.

7.2.6 Resource management

Resource management in distributed operating systems is concerned with making both local and remote resources available to users in an effective manner. Users of a distributed operating system should be able to access remote resources as easily as they can access local resources. In other words, the specific location of resources should be hidden from the users. The resources of a distributed system are made available to users in the various ways like data migration, computation migration, and distributed scheduling.

7.2.7 Security

The security of a system is the responsibility of its operating system. Two issues that must be considered in the design of security for computer systems are authentication and authorization. Authentication is the process of guaranteeing that an entity is what it claims to be. Authorization is the process of deciding what privileges an entity has and making only these privileges available.

7.2.8 Structuring

The structure of an operating system defines how various parts of the operating system are organized.

7.3 CLIENT-SERVER COMPUTING MODEL

In the client-server model, processes are categorized as servers and clients. Servers are the processes that provide services. Processes that need services are referred to as clients. In the client-server model, a client process needing a service (like reading data from a file) sends a message to the server and waits for a reply message. The server process, after performing the requested task, sends the result in the form of a reply message to the client process.

7.4 COMMUNICATION PRIMITIVES

The communication network provides a means to send streams of data in distributed systems. The communication primitives are the high-level constructs with which programs use the underlying communication network. They play a significant role in the effective usage of distributed systems. The communication primitives influence a programmer's choice of algorithms as well as the ultimate performance of the programs. They influence both the ease of use of a system and the efficiency of applications that are developed for the system.

There are two communication models, (1) message passing and (2) remote procedure call, that provide communication primitives. These two models have been widely used to develop distributed operating systems and applications for distributed systems.

7.4.1 Message passing model

The message passing model provides two basic communication primitives, send and receive. The send primitive has two parameters, a message and its destination. The receive primitive has two parameters the source and destination.

7.4.2 Remote procedure calls.

While the message passing communication model provides a highly flexible communication ability, programmers using such a model must handle the following details.

- Pairing of responses with request messages.
- Data representation (when computers of different architectures or programs written in different programming languages are communicating).
- Knowing the address of the remote machine or the server.
- Taking care of communication and system failure.

7.5 SUMMARY

- The term distributed system consists of several computers that do not share a memory or a clock. The computers communicate with each other by exchanging messages over a communication network and each computer has its own memory and runs its own operating system.
- significant advantages of distributed systems over traditional time-sharing systems are : Resource sharing, Enhanced performance, Improved reliability and availability, Modular expandability
- important issues that arise in the design of a distributed operating system include the unavailability of up-to-date global knowledge, naming, scalability, compatibility, process synchronization, resource management, security, and structuring of the operating system.
- in distributed systems, processes can request resources and release resources in any order that may not be known. If the sequence of the allocation of resources to processes is not controlled in

such environments, deadlocks may occur. It is important that deadlocks are detected and resolved as soon as possible, otherwise, system performance can degrade severely.

7.6 QUESTION

1. Define Distributed OS.
2. Explain various features of Distributed OS.
3 Explain the client ser model of distributed OS.
4. Explain various design issue in distributes OS.
5. Write a short note on

   (i) Message passing (ii) Remote Procedure Call

□□□□

UNIT 8 : DISTRIBUTED MUTUAL EXCLUSION

Structure of the Unit

8.0 OBJECTIVES

After going through this unit student will be able to :

- understand the meaning of mutual exclusion.
- Understand the requirement and performance of mutual exclusion
- Learn about the various token, non-token and quorum based distributed mutual exclusion algorithms

8.1 INTRODUCTION

There are several resources in a system that must not be used simultaneously by multiple processes if program operation is to be correct. For example, a file must not be simultaneously updated by multiple processes. Similarly, use of unit record peripherals such as tape drives or printers must be restricted to a single process at a time. Therefore, exclusive access to such a shared resource by a process must be ensured. This exclusiveness of access is called **mutual exclusion** between processes. The sections of a program that need exclusive access to shared resources are referred to as **critical sections.**

Mutual exclusion is a fundamental problem in distributed computing systems. Mutual exclusion ensures that concurrent access of processes to a shared resource or data is serialized, that is, executed

in a mutually exclusive manner. Mutual exclusion in a distributed system states that only one process is allowed to execute the critical section (CS) at any given time. In a distributed system, shared variables (semaphores) or a local kernel cannot be used to implement mutual exclusion. Message passing is the sole means for implementing distributed mutual exclusion. The decision as to which process is allowed access to the CS next is arrived at by message passing, in which each process learns about the state of all other processes in some consistent way. The design of distributed mutual exclusion algorithms is complex because these algorithms have to deal with unpredictable message delays and incomplete knowledge of the system state. There are three basic approaches for implementing distributed mutual exclusion:

1. Token-based approach.
2. Non-token-based approach.
3. Quorum-based approach.

In the token-based approach, a unique token (also known as the PRIVILEGE message) is shared among the sites. A site is allowed to enter its CS if it possesses the token and it continues to hold the token until the execution of the CS is over. Mutual exclusion is ensured because the token is unique. The algorithms based on this approach essentially differ in the way a site carries out the search for the token. In the non-token-based approach, two or more successive rounds of messages are exchanged among the sites to determine which site will enter the CS next. A site enters the critical section (CS) when an assertion, defined on its local variables, becomes true. Mutual exclusion is enforced because the assertion becomes true only at one site at any given time. In the quorum-based approach, each site requests permission to execute the CS from a subset of sites (called a quorum). The quorums are formed in such a way that when two sites concurrently request access to the CS, at least one site receives both the requests and this site is responsible to make sure that only one request executes the CS at any time.

8.2 SYSTEM MODEL

The system consists of N sites, S1, S2, ..., SN . Without loss of generality, we assume that a single process is running on each site. The process at site Si is denoted by pi. All these processes communicate asynchronously over an underlying communication network. A process wishing to enter the CS requests all other or a subset of processes by sending REQUEST messages, and waits for appropriate replies before entering the CS. While waiting the process is not allowed to make further requests to enter the CS. A site can be in one of the following three states: requesting the CS, executing the CS, or neither requesting nor executing the CS (i.e., idle). In the "requesting the CS" state, the site is blocked and cannot make further requests for the CS. In the "idle" state, the site is executing outside the CS. In the token-based algorithms, a site can also be in a state where a site holding the token is executing outside the CS. Such state is refereed to as the idle token state. At any instant, a site may have several pending requests for CS. A site queues up these requests and serves them one at a time.

We do not make any assumption regarding communication channels if they are FIFO or not. This is algorithm specific. We assume that channels reliably deliver all messages, sites do not crash, and the network does not get partitioned. Some mutual exclusion algorithms are designed to handle such situations. Many algorithms use Lamport-style logical clocks to assign a timestamp to critical section requests. Timestamps are used to decide the priority of requests in case of a conflict. The general rule followed is that the smaller the timestamp of a request, the higher its priority to execute the CS.

We use the following notation: N denotes the number of processes or sites involved in invoking

the critical section, T denotes the average message delay, and E denotes the average critical section execution time.

8.3 REQUIREMENTS OF MUTUAL EXCLUSION ALGORITHMS

A mutual exclusion algorithm should satisfy the following properties:

1. Safety property : The safety property states that at any instant, only one process can execute the critical section. This is an essential property of a mutual exclusion algorithm.

2. Liveness property : This property states the absence of deadlock and starvation. Two or more sites should not endlessly wait for messages that will never arrive. In addition, a site must not wait indefinitely to execute the CS while other sites are repeatedly executing the CS. That is, every requesting site should get an opportunity to execute the CS in finite time.

3. Fairness : Fairness in the context of mutual exclusion means that each process gets a fair chance to execute the CS. In mutual exclusion algorithms, the fairness property generally means that the CS execution requests are executed in order of their arrival in the system (the time is determined by a logical clock).

The first property is absolutely necessary and the other two properties are considered important in mutual exclusion algorithms.

8.4 PERFORMANCE METRICS OF MUTUAL EXCLUSION ALGORITHMS

The performance of mutual exclusion algorithms is generally measured by the following four metrics:

1. Message complexity : This is the number of messages that are required per CS execution by a site.

2. Synchronization : delay After a site leaves the CS, it is the time required and before the next site enters the CS. Note that normally one or more sequential message exchanges may be required after a site exits the CS and before the next site can enter the CS.

3. Response time : This is the time interval a request waits for its CS execution to be over after its request messages have been sent out. Thus, response time does not include the time a request waits at a site before its request messages have been sent out.

4. System throughput : This is the rate at which the system executes requests for the CS. If SD is the synchronization delay and E is the average critical section execution time, then the throughput is given by the following equation:

$$\text{System throughput} = 1/(SD + E)$$

5. Low and high load performance : The load is determined by the arrival rate of CS execution requests. Performance of a mutual exclusion algorithm depends upon the load and we often study the performance of mutual exclusion algorithms under two special loading conditions, viz., "low load" and "high load." Under low load conditions, there is seldom more than one request for the critical section present in the system simultaneously. Under heavy load conditions, there is always a pending request for critical section at a site. Thus, in heavy load conditions, after having executed a request, a site immediately initiates activities to execute its next CS request. A site is seldom in the idle state in heavy

load conditions. For many mutual exclusion algorithms, the performance metrics can be computed easily under low and heavy loads through a simple mathematical reasoning.

8.4.1 Best and worst case performance

Generally, mutual exclusion algorithms have best and worst cases for the performance metrics. In the best case, prevailing conditions are such that a performance metric attains the best possible value. For example mutual exclusion algorithms the best value of the response time is a roundtrip message delay plus the CS execution time, 2T +E. Often for mutual exclusion algorithms, the best and worst cases coincide with low and high loads, respectively. For examples, the best and worst values of the response time are achieved when load is, respectively, low and high; in some mutual exclusion algorithms the best and the worse message traffic is generated at low and heavy load conditions, respectively.

8.5 LAMPORT'S ALGORITHM

Lamport developed a distributedmutual exclusion algorithm as an illustration of his clock synchronization scheme. The algorithm is fair in the sense that a request for CS is executed in the order of their timestamps and time is determined by logical clocks. When a site processes a request for the CS, it updates its local clock and assigns the request a timestamp. The algorithm executes CS requests in the increasing order of timestamps. Every site Si keeps a queue, request_queuei, which contains mutual exclusion requests ordered by their timestamps. (Note that this queue is different from the queue that contains local requests for CS execution awaiting their turn.) This algorithm requires communication channels to deliver messages the FIFO order.

The Algorithm

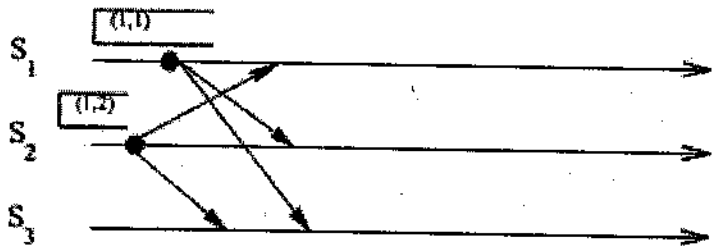


FIG 8.1 : SITES S1 AND S2 ARE MAKING REQUESTS FOR THE CS.

Requesting the critical section :

- When a site Si wants to enter the CS, it broadcasts a REQUEST(tsi, i) message to all other sites and places the request on request_queuei. ((tsi, i) denotes the timestamp of the request.)
- When a site Sj receives the REQUEST(tsi, i) message from site Si,places site Si's request on request_queuej and it returns a timestamped REPLY message to Si.

Executing the critical section : Site Si enters the CS when the following two conditions hold:

L1 : Si has received a message with timestamp larger than (tsi, i) from all other sites.

L2 : Si's request is at the top of request_queuei.

Releasing the critical section:

- Site Si, upon exiting the CS, removes its request from the top of its request queue and broadcasts a timestamped RELEASE message to all other sites.
- When a site Sj receives a RELEASE message from site Si, it removes Si's request from its request queue.

When a site removes a request from its request queue, its own request may come at the top of the queue, enabling it to enter the CS. Clearly, when a site receives a REQUEST, REPLY or RELEASE message, it updates its clock using the timestamp in the message.

8.5.1 Correctness

Theorem 1: Lamport's algorithm achieves mutual exclusion.

Proof: Proof is by contradiction. Suppose two sites Si and Sj are executing the CS concurrently. For this to happen conditions L1 and L2 must hold at both the sites concurrently. This implies that at some instant in time, say t, both Si and Sj have their own requests at the top of their request_queues and condition L1 holds at them. Without loss of generality, assume that Si's request has smaller timestamp than the request of Sj . From condition L1 and FIFO property of the communication channels, it is clear that at instant t the request of Si must be present in request_queuej when Sj was executing its CS. This implies that Sj 's own request is at the top of its own request_queue when a smaller timestamp request, Si's request, is present in the request_queuej – a contradiction!! Hence, Lamport's algorithm achieves mutual exclusion.

Theorem 2 : Lamport's algorithm is fair.

Proof: A distributed mutual exclusion algorithm is fair if the requests for CS are executed in the order of their timestamps. The proof is by contradiction. Suppose a site Si's request has a smaller timestamp than the request of another site Sj and Sj is able to execute the CS before Si. For Sj to execute the CS, it has to satisfy the conditions L1 and L2. This implies that at some instant in time say t, Sj has its own request at the top of its queue and it has also received a message with timestamp larger than the timestamp of its request from all other sites. But request_queue at a site is ordered by timestamp, and according to our assumption Si has lower timestamp. So Si's request must be placed ahead of the Sj 's request in the request_queuej . This is a contradiction. Hence Lamport's algorithm is a fair mutual exclusion algorithm.

8.5.2 Performance

For each CS execution, Lamport's algorithm requires (N - 1) REQUEST messages, (N - 1) REPLY messages, and (N-1) RELEASE messages. Thus, Lamport's algorithm requires 3(N-1) messages per CS invocation. Synchronization delay in the algorithm is T.

8.5.3 An Optimization

In Lamport's algorithm, REPLY messages can be omitted in certain situations. For example, if site Sj receives a REQUEST message from site Si after it has sent its own REQUEST message with timestamp higher than the timestamp of site Si's request, then site Sj need not send a REPLY message to site Si. This is because when site Si receives site Sj 's request with timestamp higher than its own, it can conclude that site Sj does not have any smaller timestamp request which is still pending (because communication channels preserves FIFO ordering).

With this optimization, Lamport's algorithm requires between 3(N-1) and 2(N-1) messages per CS execution.

8.6 RICART-AGRAWALA ALGORITHM

The Ricart-Agrawala algorithm assumes the communication channels are FIFO. The algorithm uses two types of messages: REQUEST and REPLY. A process sends a REQUEST message to all other processes to request their permission to enter the critical section. A process sends a REPLY message to a process to give its permission to that process. Processes use Lamport-style logical clocks to assign a timestamp to critical section requests. Timestamps are used to decide the priority of requests in case of con?ict if a process pi that is waiting to execute the critical section, receives a REQUEST message from process pj , then if the priority of pj 's request is lower, pi defers the REPLY to pj and sends a REPLY message to pj only after executing the CS for its pending request. Otherwise, pi sends a REPLY message to pj immediately, provided it is currently not executing the CS. Thus, if several processes are requesting execution of the CS, the highest priority request succeeds in collecting all the needed REPLY messages and gets to execute the CS.

Each process pi maintains the Request-Deferred array, RDi, the size of which is the same as the number of processes in the system. Initially, □i □j: RDi[j]=0. Whenever pi defer the request sent by pj , it sets RDi[j]=1 and after it has sent a REPLY message to pj , it sets RDi[j]=0.

8.6.1 Description of the Algorithm

Requesting the critical section:

(a) When a site Si wants to enter the CS, it broadcasts a timestamped REQUEST message to all other sites.

(b) When site Sj receives a REQUEST message from site Si, it sends a REPLY message to site Si if site Sj is neither requesting nor executing the CS, or if the site Sj is requesting and Si's request's timestamp is smaller than site Sj's own request's timestamp. Otherwise, the reply is deferred and Sj sets RDj[i]=1

Executing the critical section :

(c) Site Si enters the CS after it has received a REPLY message from every site it sent a REQUEST message to.

Releasing the critical section:

(d) When site Si exits the CS, it sends all the deferred REPLY messages: □j if RDi[j]=1, then send a REPLY message to Sj and set RDi[j]=0.

When a site receives a message, it updates its clock using the timestamp in the message. Also, when a site takes up a request for the CS for processing, it updates its local clock and assigns a timestamp to the request. In this algorithm, a site's REPLY messages are blocked only by sites which are requesting the CS with higher priority (i.e., smaller timestamp). Thus, when a site sends out deferred REPLY messages, site with the next highest priority request receives the last needed REPLY message and enters the CS. Execution of the CS requests in this algorithm is always in the order of their timestamps.

8.6.2 Correctness

Theorem 3 : Ricart-Agrawala algorithm achieves mutual exclusion.

Proof : Proof is by contradiction. Suppose two sites Si and Sj are executing the CS concurrently and Si's request has higher priority (i.e., smaller timestamp) than the request of Sj . Clearly, Si received Sj 's request after it has made its own request. (Otherwise, Si's request will have lower priority.) Thus, Sj can concurrently execute the CS with Si only if Si returns a REPLY to Sj (in response to Sj's request) before Si exits the CS. However, this is impossible because Sj 's request has lower priority. Therefore, Ricart-Agrawala algorithm achieves mutual exclusion.

In Ricart-Agrawala algorithm, for every requesting pair of sites, the site with higher priority request will always defer the request of the lower priority site. At any time only the highest priority request succeeds in getting all the needed REPLY messages.

8.6.3 Performance

For each CS execution, Ricart-Agrawala algorithm requires (N - 1) REQUEST messages and (N-1) REPLY messages. Thus, it requires 2(N-1) messages per CS execution. Synchronization delay in the algorithm is T.

8.7 SINGHAL'S DYNAMIC INFORMATION-STRUCTURE ALGORITHM

Most mutual exclusion algorithms use a static approach to invoke mutual exclusion; i.e., they always take the same course of actions to invoke mutual exclusion no matter what is the state of the system. A problem with these algorithms is the lack of efficiency because these algorithms fail to exploit the changing conditions in the system. Note that an algorithm can exploit dynamic conditions of the system to optimize the performance.

For example, if few sites are invoking mutual exclusion very frequently and other sites invoke mutual exclusion much less frequently, then a frequently invoking site need not ask for the permission of less frequently invoking site every time it requests an access to the CS. It only needs to take permission from all other frequently invoking sites. Singhal developed an adaptive mutual exclusion algorithm based on this observation. The information-structure of the algorithm evolves with time as sites learn about the state of the system through messages. Dynamic information-structure mutual exclusion algorithms are attractive because they can adapt to ?uctuating system conditions to optimize the performance.

The design of such adaptive mutual exclusion algorithms is challenging and we list some of the design challenges next :

- How does a site efficiently know what sites are currently actively invoking mutual exclusion?
- When a less frequently invoking site needs to invoke mutual exclusion, how does it do it?
- How a less frequently invoking site does makes a transition to more frequently invoking site and vice-versa?
- How to insure that mutual exclusion is guaranteed when a site does not take the permission of every other site?
- How to insure that a dynamic mutual exclusion algorithm does not waste resources and time in collecting systems state, offsetting any gain?

8.7.1 System Model

We consider a distributed system consisting of n autonomous sites, say S1, S2, ..., Sn, which

are connected by a communication network. We assume that the sites communicate completely by message passing. Message propagation delay is ?nite but unpredictable and between any pair of sites, messages are delivered in the order they are sent. For the ease of presentation, we assume that the underlying communication network is reliable and sites do not crash. However, methods have been proposed for recovery from message losses and site failures.

8.7.2 Data Structures

Information-structure at a site Si consists of two sets. The ?rst set Ri, called request set, contains the sites from which Si must acquire permission before executing CS. The second set Ii, called inform set, contains the sites to which Si must send its permission to execute CS after executing its CS.

Every site Si maintains a logical clock Ci, which is updated according to Lamport's rules. Every request for CS execution is assigned a timestamp which is used to determine its priority. The smaller the timestamp of a request, the higher its priority. Every site maintains three Boolean variables to denote the state of the site: Requesting, Executing, and My_priority. Requesting and executing are true if and only if the site is requesting or executing CS, respectively. My_priority is true if pending request of Si has priority over the current incoming request.

8.7.3 Initialization

The system starts in the following initial state:

For a site Si (i = 1 to n),

Ri := {S1, S2,..., Si - 1, Si}

Ii := Si

Ci := 0

Requesting = Executing := False

Thus, initially site Si, 1=i=n, sends request messages only to sites Si, Si - 1, ..., S1. If we stagger sites Sn to S1 from left to right, then the initial system state has the following two properties :

1. Each site requests permission from all the sites to its right and from no site to its left. Conversely, for a site, all the sites to its left asks for its permission and no site to its right asks for its permission. Or putting together, for a site, only all the sites to its left will ask for its permission and it will ask for the permission of only all the sites to its right. Therefore, every site Si divides all the sites into two disjoint groups; all the sites in the ?rst group request permission from Si and Si requests permission from all the sites in the second group. This property is important for enforcing mutual exclusion.

2. The cardinality of Ri decreases in stepwise manner from left to right. Due to this reason, this con?guration has been called "staircase pattern" in topological sense.

The Algorithm

Site Si executes the following three steps to invoke mutual exclusion:

Step 1: (Request Critical Section)

Requesting = true;

Ci = Ci + 1;

```
Send REQUEST(Ci, i) message to all sites in Ri;
Wait until Ri = Ø;      /* Wait until all sites in Ri have sent a reply to Si */
Requesting = false;
Step 2: (Execute Critical Section)
Executing = true;
Execute CS;
Executing = false;
Step 3: (Release Critical Section)
For every site Sk in Ii (except Si) do
Begin
Ii= Ii -- {Sk};
Send REPLY(Ci, i) message to Sk;
Ri= Ri + {Sk}
End
```

8.7.4 REQUEST message handler

REQUEST message handler at a site processes incoming REQUEST messages. It takes actions such as updating information-structure and sending REQUEST/REPLY messages to other sites.

REQUEST message handler at site Si is given below:

```
/* Site Si is handling message REQUEST(c, j) */
Ci := max{Ci, c};
Case
Requesting = true:
Begin
if My_priority then Ii:= Ii + {j}
/*My_Priority is true if the pending request of Si has priority over the incoming request */
Else
Begin
Send REPLY(Ci, i) message to Sj ;
If not (Sj ∈ Ri) then
Begin
Ri= Ri + {Sj};
Send REQUEST(Ci, i) message to site Sj ;
```

End;

End;

End;

Executing = true: Ii = Ii + {Sj};

Executing = false ∧ Requesting = false:

Begin

Ri= Ri + {Sj};

Send REPLY(Ci, i) message to Sj ;

End;

8.7.5 REPLY message handler

The REPLY message handler at a site processes incoming REPLY messages. It updates the information-structure. REPLY message handler at site Si is given below:

/* Site Si is handling a message REPLY(c, j) */

Begin

Ci := max{Ci, c};

Ri = Ri – {Sj};

End;

Note that REQUEST and REPLY message handlers and the steps of the algorithm access shared data structures, viz., Ci, Ri, and Ii. To guarantee the correctness, it's important that execution of REQUEST and REPLY message handlers and all three steps of the algorithm (except "wait for Ri = Ø to hold" in Step 1) mutually exclude each other.

8.7.6 Performance Analysis

The synchronization delay in the algorithm is T. Below, we compute the message complexity in low and heavy loads.

Low load condition : In case of low traffic of CS requests, most of the time only one or no request for the CS will be present in the system. Consequently, the staircase pattern will reestablish between two successive requests for CS and there will seldom be an interference among the CS requests from different sites. In the staircase con?guration, cardinality of the request sets of the sites is 1, 2, ..., (n-1), n, respectively, from right to left. Therefore, when traffic of requests for CS is low, sites will send 0, 1, 2, ..., (n-1) number of REQUEST messages with equal likelihood (assuming uniform traffic of CS requests at sites). Therefore, the mean number of REQUEST messages sent per CS execution for this case is =(0+1+2+ .....+(n-1))/n = (n-1)/2. Since a REPLY message is returned for every REQUEST message, the average number of messages exchanged per CS execution is 2*(n-1)/2 = (n-1).

Heavy load condition : When the rate of CS requests is high, all the sites always have a pending request for CS execution. In this case, a site on the average receives (n-1)/2 REQUEST messages from other sites while waiting for its REPLY messages. Since a site sends REQUEST messages only in response

to REQUEST messages of higher priority, on the average it will send (n-1)/4 REQUEST messages while waiting for REPLY messages. Therefore, the average number of messages exchanged per CS execution in high demand is 2*[(n-1)/2+(n-1)/4] = 3*(n-1)/2.

8.8 LODHA AND KSHEMKALYANI'S FAIR MUTUAL EXCLUSION ALGORITHM

Lodha and Kshemakalyani's algorithm decreases the message complexity of Ricart-Agrawala algorithm by using the following interesting observation: When a site is waiting to execute the CS, it need not receive REPLY messages from every other site. To enter the CS, a site only needs to receive a REPLY message from the site whose request just precedes its request in priority. For example, if sites Si1,Si2 , ..Sin have a pending request for CS and the request of Si1 has the highest priority and that of Sin has the lowest priority and the priority of requests decreases from Si1 to Sin, then a site Sik only needs a REPLY message from site Sik-1 , 1 < k = n to enter the CS.

8.8.1 System Model

Each request is assigned a priority ReqID and requests for the CS access are granted in the order of decreasing priority. We will defer the details of what ReqID is composed of to later sections.

The underlying communication network is assumed to be error free.

De?nition 1: Ri and Rj are concurrent iff Pi's REQUEST message is received by Pj after Pj has made its request and Pj 's REQUEST message is received by Pi after Pi has made its request.

De?nition 2: Given Ri, we de?ne the concurrency set of Ri as follows:

CSeti = {Rj | Ri is concurrent with Rj}∪ {Ri}.

The Algorithm

The algorithm uses three types of messages: REQUEST, REPLY and FLUSH and obtains savings on the number of messages exchanged per CS access by assigning multiple purposes to each. For the purpose of blocking a mutual exclusion request, every site Si has a data structure called local_request_queue (denoted as LRQi) which contains all concurrent requests made with respect to Si's request and these requests are ordered with respect to the priority.

All requests are totally ordered by their priorities and the priority is determined by the timestamp of the request. Hence, when a process receives a REQUEST message from some other process, it can immediately determine if it is allowed the CS access before the requesting process or after it.

In this algorithm, messages play multiple roles.

Multiple uses of a REPLY message

1. A REPLY message acts as a reply from a process that is not requesting.
2. A REPLY message acts as a collective reply from processes that have higher priority requests.

A REPLY(Rj) from a process Pj indicates that Rj is the request made by Pj for which it has executed the CS. It also indicates that all the requests with priority = priority of Rj have ?nished executing CS and are no longer in contention.

Thus, in such situations, a REPLY message is a logical reply and denotes a collective reply from all processes that had made higher priority requests.

Uses of a FLUSH message

Similar to a REPLY message, a FLUSH message is a logical reply and denotes a collective reply from all processes that had made higher priority requests. After a process has exited the CS, it sends a FLUSH message to a process requesting with the next highest priority, which is determined by looking up the process's local request queue. When a process Pi ?nishes executing the CS, it may ?nd a process Pj in one of the following states:

1. Rj is in the local queue of Pi and located in some position after Ri, which implies that Rj is concurrent with Ri.
2. Pj had replied to Ri and Pj is now requesting with a lower priority.
3. Pj 's request had higher priority than Pi's (implying that it had finished the execution of the CS) and is now requesting with a lower priority.

A process Pi after executing the CS, sends a FLUSH message to a process identi?ed in the Case 1 above, which has the next highest priority, whereas it sends REPLY messages to the processes identi?ed in Cases 2 and 3 as their requests are not concurrent with Ri (the resuests of processes in Cases 2 and 3 were deferred by Pi till it exits the CS). Now it is up to the process receiving the FLUSH message and the processes recieving REPLY messages in Cases 2 and 3 to determine who is allowed to enter the CS next.

Consider a scenario where we have a set of requests R3 R0 R2 R4 R1 ordered in decreasing priority where R0 R2 R4 are concurrent with one another, then P0 maintains a local queue of [R0, R2, R4] and when it exits the CS, it sends a FLUSH (only) to P2.

- Multiple uses of a REQUEST message

Considering two processes Pi and Pj , there can be two cases:

Case 1: Pi and Pj are not concurrently requesting. In this case, the process which requests

?rst will get a REPLY message from the other process.

Case 2: Pi and Pj are concurrently requesting. In this case, there can be two subcases:

1. Pi is requesting with a higher priority than Pj . In this case, Pj 's REQUEST message serves as an implicit REPLY message to Pi's request. Also, Pj should wait for REPLY/FLUSH message from some process to enter the CS.
2. Pi is requesting with a lower priority than Pj . In this case, Pi's REQUEST message serves as an implicit REPLY message to Pj 's request. Also, Pi should wait for REPLY/FLUSH message from some process to enter the CS.

The Algorithm

- Initial local state for process Pi

- int My_Sequence_Numberi=0

- array of boolean RVi[j]=0, □j ∈ {1...N}

- queue of ReqID LRQi is NULL

- int Highest_Sequence_Number_Seeni=0

- InvMutEx: Process Pi executes the following to invoke mutual exclusion:

1. My_Sequence_Numberi = Highest_Sequence_Number_Seeni + 1
2. LRQi = NULL
3. Make REQUEST(Ri) message, where Ri = (My_Sequence_Numberi, i).
4. Insert this REQUEST in the LRQi in sorted order.
5. Send this REQUEST message to all other processes.
6. RVi[k]=0 □k ∈ {1...N}-{i}. RVi[i]=1.

- RcvReq: Process Pi receives REQUEST(Rj), where Rj=(SN, j), from process Pj :

1. Highest_Sequence_Number_Seeni = max(Highest_Sequence_Number_Seeni, SN).
2. if Pi is requesting:

(a) if RVi[j] = 0, then insert this request in the LRQi (in sorted order) and mark RVi[j] = 1. If (CheckExecuteCS), then execute CS.

(b) if RVi[j] = 1, then defer the processing of this request, which will be processed after Pi executes CS.

3. If Pi is not requesting, then send a REPLY(Ri) message to Pj . Ri denotes the ReqID of the last request made by Pi that was satis?ed.

- RcvReply: Process Pi receives REPLY(Rj) message from process Pj : Rj denotes the ReqID of the last request made by Pj that was satis?ed.

1. RVi[j]=1
2. Remove all requests from LRQi that have a priority = the priority of Rj
3. If(CheckExecuteCS), then execute CS.

- FinCS: Process Pi ?nishes executing CS.

1. Send FLUSH(Ri) message to the next candidate in the LRQi. Ri denotes the ReqID that was satis?ed.
2. Send REPLY(Ri) to the deferred requests. Ri is the ReqID corresponding to which Pi just executed the CS.

- RcvFlush: Process Pi receives a FLUSH(Rj) message from a process Pj :

1. RVi[j] = 1
2. Remove all requests in LRQi that have the priority = the priority of Rj .
3. If (CheckExecuteCS) then execute CS.

- CheckExecuteCS: if (RVi[k]=1, □k ∈ {1..N}) and Pi's request is at the head of LRQi, then return true, else return false.

8.9 QUORUM-BASED MUTUAL EXCLUSION ALGORITHMS

Quorum-based mutual exclusion algorithms respresented a departure from the trend in the

following two ways:

1. A site does not request permission from all other sites, but only from a subset of the sites. This is a radically different approach as compared to Lamport and Ricart-Agrawala algorithms where all sites participate in con?ict resolution of all other sites. In quorum-based mutual exclusion algorithm, the request set of sites are chosen such that □i □j : 1 = i, j = N :: Ri n Rj ≠ Φ. Consequently, every pair of sites has a site which mediates conflicts between that pair.

2. In quorum-based mutual exclusion algorithm, a site can send out only one REPLY message at any time. A site can send a REPLY message only after it has received a RELEASE message for the previous REPLY message. Therefore, a site Si locks all the sites in Ri in exclusive mode before executing its CS.

Quorum-based mutual exclusion algorithms signi?cantly reduce the message complexity of invoking mutual exclusion by having sites ask permission from only a subset of sites.

Since these algorithms are based on the notion of 'Coteries' and 'Quorums', we ?rst describe the idea of coteries and quorums. A coterie C is de?ned as a set of sets, where each set g ∈ C is called a quorum. The following properties hold for quorums in a coterie:

Intersection property : For every quorum g, h ∈ C, g n h ≠ Ø.

For example, sets {1, 2, 3}, {2, 5, 7} and {5, 7, 9} cannot be quorums in a coterie because the first and third sets do not have a common element.

Minimality property : There should be no quorums g, h in coterie C such that g ⊇ h. For example, sets {1,2,3} and {1,3} cannot be quorums in a coterie because the ?rst set is a superset of the second.

Coteries and quorums can be used to develop algorithms to ensure mutual exclusion in a distributed environment. A simple protocol works as follows: Let 'a' is a site in quorum 'A'. If 'a' wants to invoke mutual exclusion, it requests permission from all sites in its quorum 'A'. Every site does the same to invoke mutual exclusion. Due to the Intersection Property, quorum 'A' contains at least one site that is common to the quorum of every other site. These common sites send permission to only one site at any time. Thus, mutual exclusion is guaranteed.

Note that the Minimality property ensures ef?ciency rather than correctness. In the simplest form, quorums are formed as sets that contain a majority of sites. There exists a variety of quorums and a variety of ways to construct quorums. For example, Maekawa used the theory of projective planes to develop quorums of size vN.

8.10 AGARWAL-EL ABBADI QUORUM-BASED ALGORITHM

Agarwal and El Abbadi developed a simple and efficient mutual exclusion algorithm by introducing tree quorums. They gave a novel algorithm for constructing tree-structured quorums in the sense that it uses hierarchical structure of a network. The mutual exclusion algorithm is independent of the underlying topology of the network and there is no need for a multicast facility in the network. However, such facility will improve the performance of the algorithm. The mutual exclusion algorithm assumes that sites in the distributed system can be organized into a structure such as tree, grid, binary tree, etc. and there exists a routing mechanism to exchange messages between different sites in the system.

Agarwal-El Abbadi quorum-based algorithm, however, constructs quorums from trees. Such

quorums are called 'tree-structured quorums'. The following sections describe an algorithm for constructing tree-structured quorums and present an analysis of the algorithm and a protocol for mutual exclusion in distributed systems using tree-structured quorums.

8.10.1 Constructing a tree-structured quorum

All the sites in the system are logically organized into a complete binary tree. To build such a tree, any site could be chosen as the root, any other two sites may be chosen as its children and so on. For a complete binary tree with level 'k', we have 2^{k+1}– 1 sites with its root at level k and leaves at level 0. The number of sites in a path from the root to a leaf is equal to the level of the tree k+1 which is equal to O(log n). There will be 2k leaves in the tree. A path in a binary tree is the sequence a1, a2. . . ai , ai+1. . . . ak such that ai is the parent of ai+1.

The algorithm for constructing structured quorums from the tree is given below. For the purpose of presentation, we assume that the tree is complete, however, the algorithm works for any arbitrary binary tree.

The algorithm for constructing tree-structured quorums uses two functions called GetQuorum(tree) and GrantsPermission(site) and assumes that there is a well-de?ned root for the tree. GetQuorum is a recursive function that takes a tree node 'x' as the parameter and calls GetQuorum for its child node provided that the GrantsPermission(x) is true. The GrantsPermission(x) is true only when the node 'x' agrees to be in the quorum. If the node 'x' is down due to a failure, then it may not agree to be in the quorum and the value of GrantsPermission(x) will be false. The algorithm tries to construct quorums in a way that each quorum represents any path from the root to a leaf, i.e., in this case (no failures) quorum is any set $a_1, a_2, \ldots a_i, a_{i+1}, \ldots .. ak$ where a_1 is the root and a_k is a leaf, and for all i<k, a_i is the parent of a_{i+1}. If it fails to ?nd such a path (say, because node 'x' has failed), the control goes to the ELSE block which speci?es that the failed node 'x' is substituted by two paths both of which start with the left and right children of 'x' and end at leaf nodes. Note that each path must terminate in a leaf site. If the leaf site is down or inaccessible due to any reason, then the quorum cannot be formed and the algorithm terminates with an error condition. The sets that are constructed using this algorithm are termed as tree quorums.

```
FUNCTION GetQuorum (Tree: NetworkHierarchy): QuorumSet;
VAR left, right : QuorumSet;
BEGIN
IF Empty (Tree) THEN
RETURN ({});
ELSE IF GrantsPermission(Tree?.Node) THEN
RETURN ((Tree?.Node) ∪ GetQuorum (Tree?.LeftChild));
OR
RETURN ((Tree?.Node) ∪ GetQuorum (Tree?.RightChild));
ELSE
left?GetQuorum(Tree?.left);
```

```
right?GetQuorum(Tree?.right);
IF (left = Ø ∨ right = Ø) THEN
(* Unsuccessful in establishing a quorum *)
EXIT(-1);
ELSE
RETURN (left ∪ right);
END; (* IF *)
END; (* IF *)
END; (* IF *)
END GetQuorum
```

8.11 THE ALGORITHM FOR DISTRIBUTED MUTUAL EXCLUSION

We now describe the algorithm for achieving distributed mutual exclusion using tree-structured quorums. Suppose a site s wants to enter the critical section (CS). The following events should occur in the given order:

1. Site s sends a 'Request' message to all other sites in the structured quorum it belongs to.
2. Each site in the quorum stores incoming requests in a request queue, ordere by their timestamps.
3. A site sends a 'Reply' message, indicating its consent to enter CS, only to the request at the head of its request queue, having the lowest timestamp.
4. If the site s gets a 'Reply' message from all sites in the structured quorum it belongs to, it enters the CS.
5. After exiting the CS, s sends a 'Relinquish' message to all sites in the structured quorum. On the receipt of the 'Relinquish' message, each site removes s's request from the head of its request queue.
6. If a new request arrives with a timestamp smaller than the request at the head of the queue, an 'Inquire' message is sent to the process whose request is at the head of the queue and waits for a 'Yield' or 'Relinquish' message.
7. When a site s receives an 'Inquire' message, it acts as follows:

If s has acquired all of its necessary replies to access the CS, then it simply ignores the 'Inquire' message and proceeds normally and sends a 'Relinquish' message after exiting the CS.

If s has not yet collected enough replies from its quorum, then it sends a 'Yield' message to the inquiring site.

8. When a site gets the 'Yield' message, it puts the pending request (on behalf of which the 'Inquire' message was sent) at the head of the queue and sends a 'Reply' message to the requestor.

8.12 TOKEN-BASED ALGORITHMS

In token-based algorithms, a unique token is shared among the sites. A site is allowed to enter

its CS if it possesses the token. A site holding the token can enter its CS repeatedly until it sends the token to some other site. Depending upon the way a site carries out the search for the token, there are numerous token-based algorithms. Next, we discuss two token-based mutual exclusion algorithms.

Before we start with the discussion of token-based algorithms, two comments are in order. First, token-based algorithms use sequence numbers instead of timestamps. Every request for the token contains a sequence number and the sequence numbers of sites advance independently. A site increments its sequence number counter every time it makes a request for the token. (A primary function of the sequence numbers is to distinguish between old and current requests.) Second, the correctness proof of token-based algorithms, that they enforce mutual exclusion, is trivial because an algorithm guarantees mutual exclusion so long as a site holds the token during the execution of the CS. Instead, the issues of freedom from starvation, freedom from deadlock, and detection of the token loss and its regeneration become more prominent.

8.12.1 Suzuki–Kasami's token based algorithm

In Suzuki–Kasami's algorithm, if a site that wants to enter the CS does not have the token, it broadcasts a REQUEST message for the token to all other sites. A site that possesses the token sends it to the requesting site upon the receipt of its REQUEST message. If a site receives a REQUEST message when it is executing the CS, it sends the token only after it has completed the execution of the CS.

Although the basic idea underlying this algorithm may sound rather simple, there are two design issues that must be efficiently addressed:

1. How to distinguishing an outdated REQUEST message from a current REQUEST message Due to variable message delays, a site may receive a token request message after the corresponding request has been satisfied. If a site cannot determined if the request corresponding to a token request has been satisfied, it may dispatch the token to a site that does not need it. This will not violate the correctness, however, but it may seriously degrade the performance by wasting messages and increasing the delay at sites that are genuinely requesting the token. Therefore, appropriate mechanisms should implemented to determine if a token request message is outdated .

2. How to determine which site has an outstanding request for the CS After a site has finished the execution of the CS, it must determine what sites have an outstanding request for the CS so that the token can be dispatched to one of them. The problem is complicated because when a site Si receives a token request message from a site Sj , site Sj may have an outstanding request for the CS. However, after the corresponding request for the CS has been satisfied at Sj , an issue is how to inform site Si (and all other sites) efficiently about it..

Outdated REQUEST messages are distinguished from current REQUEST messages in the following manner: a REQUEST message of site Sj has the form REQUEST(j, n) where n (n = 1 2.....) is a sequence number that indicates that site Sj is requesting its nth CS execution. A site Si keeps an array of integers RNi [1,...,N] where RNi [j] denotes the largest sequence number received in a REQUEST message so far from site Sj . When site Si receives a REQUEST(j, n) message, it sets RNi [j] = max(RNi [j], n). Thus, when a site Si receives a REQUEST(j, n) message the request is outdated if RNi [j]>n. Sites with outstanding requests for the CS are determined in the following manner: the token consists of a queue of requesting sites, Q, and an array of integers LN[1,...,N], where LN[j] is the sequence number of the request which site Sj executed most recently. After executing its CS, a site Si updates LN[i]: = RNi [i] to indicate that its request corresponding to sequence number RNi [i] has been executed. Token array LN[1,...,N] permits a site to determine if a site has an outstanding request for the CS. Note that at site Si if RNi[j]=LN[j]+1, then site Sj is currently requesting a token. After executing the CS, a site checks this condition for all the j's to determine all the sites that are requesting the token

and places their i.d.'s in queue Q if these i.d.'s are not already present in Q. Finally, the site sends the token to the site whose i.d. is at the head of Q.

The beauty of the Suzuki–Kasami algorithm lies in its simplicity and efficiency. No message is needed and the synchronization delay is zero if a site holds the idle token at the time of its request. If a site does not hold the token when it makes a request, the algorithm requires N messages to obtain the token. The synchronization delay in this algorithm is 0 or T.

8.12.2 Raymond's Tree-Based Algorithm

Raymond's Tree-Based mutual exclusion algorithm uses a spanning tree of the computer network to reduce the number of messages exchanged per critical section execution. The algorithm exchanges only O(log N) messages under light load, and approximately four messages under heavy load to execute the CS, where N is the number of nodes in the network.

The algorithm assumes that the underlying network guarantees message delivery. The time or order of message arrival cannot be predicted. All nodes of the network are 'completely reliable. If the network is viewed as a graph, where the nodes in the network are the vertices of the graph, and the links between nodes are the edges of the graph, a spanning tree of a network of N nodes will be a tree that contains all the N nodes. A minimal spanning tree is one such tree with minimum cost. Typically, this cost function is based on the network link characteristics. The algorithm operates on a minimal spanning tree of the network topology or a logical structure imposed on the network.

The algorithm considers the network nodes to be arranged in an unrooted tree structure as shown in the following figure. Messages between nodes traverse along the undirected edges of the tree in the figure. The tree is also a spanning tree of the seven nodes A, B, C, D, E, F, and G. It also turns out to be a minimal spanning tree because it is the only spanning tree of these seven nodes. A node needs to hold information about and communicate only to its immediate-neighboring nodes.

In the following figure, for example, node C holds information about and communicates only to nodes B, D, and G; it does not need to know about the other nodes A, E, and F for the operation of the algorithm.

Similar to the concept of tokens used in token-based algorithms, this algorithm uses a concept of privilege to signify which node has the privilege to enter the critical section. Only one node can be in possession of the privilege (called the privileged node) at any time, except when the privilege is in transit from one node to another in the form of a PRIVILEGE message. When there are no nodes requesting for the privilege, it remains in possession of the node that last used it.

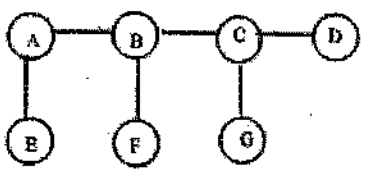


FIGURE : NODES WITH AN UNROOTED TREE STRUCTURE.

8.13 SUMMARY

Mutual exclusion is a fundamental problem in distributed computing systems, where concurrent access to a shared resource or data is serialized. Mutual exclusion in a distributed system requires that only one process be allowed to execute the critical section at any given time. Mutual exclusion algorithms

for distributed computing systems have been designed based on three approaches: token-based approach, non-token-based approach, and quorum-based approach. In token-based algorithms, a unique token is shared among the sites and a site is allowed to enter its critical section only if it possesses the token. Depending upon the way the token is managed in the system, there are several token-based algorithms.

In the non-token-based approach, sites exchange two or more rounds of messages to determine which site will enter the critical section next. In the quorum-based approach, each site requests permission from a subset of sites (called a quorum). The quorums are formed in such a way that when two sites concurrently request access to the CS, at least one site receives both the requests and which is responsible to make sure that only one request executes the critical section at any time.

A large number of mutual exclusion algorithms based on these approaches have been developed. In this chapter, we described a set of representative mutual exclusion algorithms. Early mutual exclusion algorithms were static in the sense they always take the same course of actions to invoke mutual exclusion regardless of the state of the system. These algorithms lack efficiency because these algorithms fail to exploit the changing conditions in the system. Lately, dynamic mutual exclusion algorithms have been developed. Such algorithms exploit dynamic conditions of the system to optimize the performance.

8.14 UNIT-END QUESTIONS

1. What do you mean by mutual exclusion?
2. Explain the requirements of mutual exclusion algorithm.
3. Explain with example Lamport's Non-token based algorithm for mutual exclusion.
4. Explain with example Ricart-Agrawala Non-token based algorithm for mutual exclusion.
5. Explain the difference between Token-based and Non-token based mutual exclusion algorithm.
6. Explain the Suzuki-Kasami's broadcast algorithm with proof of correctness.
7. Explain Singhal's Heuristics algorithm.
8. Discuss the comparative performance analysis for distributed mutual exclusion algorithms.
9. Show that in Lamport's algorithm the critical section is accessed according to the increasing order of timestamps.

8.15 FURTHER READINGS

1. Distributed Systems: Concepts and Design, G. Coulouris, J. Dollimore, T. Kindberg, (c) 1996 Addison Wesley Longman, Ltd., pp. 300-309.
2. Distributed Operating Systems, Andrew Tanenbaum, 1995.

□□□□

UNIT 9 : DISTRIBUTED DEADLOCK

Structure of the Unit

9.0 OBJECTIVES

After going through this unit student will be able to:

- Understand the meaning of deadlock and deadlock system models
- Understand deadlock handling strategies in distributed systems
- Understand Issues and controls in deadlock detection
- Understand various algorithms for distributed deadlock detection

9.1 INTRODUCTION

Deadlocks are a fundamental problem in distributed systems and deadlock detection in distributed systems has received considerable attention in the past. In distributed systems, a process may request resources in any order, which may not be known a priori, and a process can request a resource while holding others. If the allocation sequence of process resources is not controlled in such environments, deadlocks can occur. A deadlock can be defined as a condition where a set of processes request resources that are held by other processes in the set.

Deadlocks can be dealt with using any one of the following three strategies: Deadlock prevention, Deadlock avoidance, and Deadlock detection. **Deadlock prevention** is commonly achieved by either having a pro- ll the needed resources simultaneously before it begins execution or by pre- ls the needed resource. In the **Deadlock avoidance** approach to distributed ted to a process if the resulting global system is safe. **Deadlock detection** he status of the process–resources interaction for the presence of a deadlock eadlock, we have to abort a deadlocked process.

, **deadlock** refers to a specific condition when two or more processes are b release a resource, or more than two processes are waiting for resources

in a circular chain. Deadlock is a common problem in multiprocessing where many processes share a specific type of mutually exclusive resource known as a **software lock or soft lock**. Computers intended for the **time-sharing** and/or **real-time** markets are often equipped with a **hardware lock (or hard lock)** which guarantees **exclusive access** to processes, forcing serialized access. Deadlocks are particularly troubling because there is no **general** solution to avoid (soft) deadlocks.

9.2 SYSTEM MODEL

A distributed system consists of a set of processors that are connected by a communication network. The communication delay is finite but unpredictable. A distributed program is composed of a set of n asynchronous processes $P_1, P_2, \ldots, P_i, \ldots, P_n$ that communicate by message passing over the communication network. Without loss of generality we assume that each process is running on a different processor. The processors do not share a common global memory and communicate solely by passing messages over the communication network. There is no physical global clock in the system to which processes have instantaneous access. The communication medium may deliver messages out of order, messages may be lost, garbled, or duplicated due to timeout and retransmission, processors may fail, and communication links may go down. The system can be modeled as a directed graph in which vertices represent the processes and edges represent unidirectional communication channels.

We make the following assumptions :

- The systems have only reusable resources.
- Processes are allowed to make only exclusive access to resources.
- There is only one copy of each resource.

A process can be in two states, running or blocked. In the running state (also called active state), a process has all the needed resources and is either executing or is ready for execution. In the blocked state, a process is waiting to acquire some resource.

9.2.1 Wait-for graph (WFG)

In distributed systems, the state of the system can be modeled by directed graph, called a wait-for graph (WFG). In a WFG, nodes are processes and there is a directed edge from node P1to mode P2 if P1 is blocked and is waiting for P2to release some resource. A system is deadlocked if and only if there exists a directed cycle or knot in the WFG.

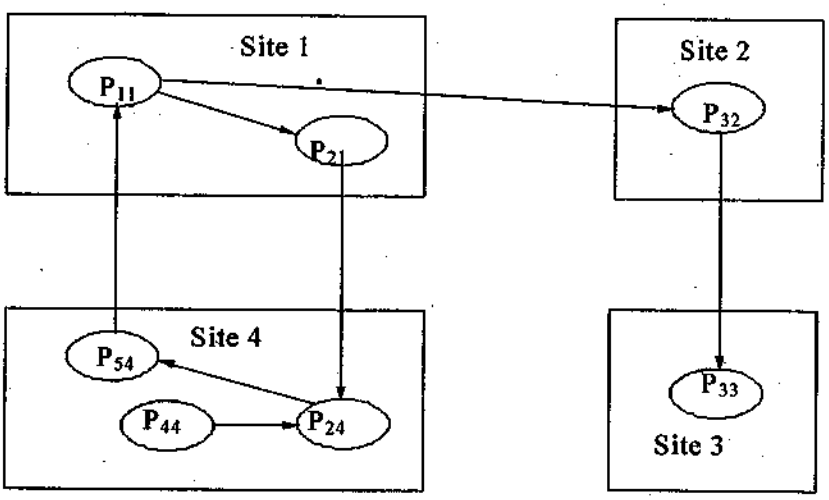


FIGURE 9.1

FIGURE 9.1 SHOWS A WFG, WHERE PROCESS P_{11} of site 1 has an edge to process P_{21} of site 1 and an edge to process P_{32} of site 2. Process P_{32} of site 2 is waiting for a resource that is currently held by process P_{33} of site 3. At the same time process P_{21} at site 1 is waiting on process P_{24} at site 4 to release a resource, and so on. If P_{33} starts waiting on process P_{24}, then processes in the WFG are involved in a deadlock depending upon the request model.

9.3 A COMPARISON OF RESOURCE AND COMMUNICATION DEADLOCKS

There are several differences between the resource model and the communication model. One critical difference is that in the communication model, a process can know the identity of those processes from which it must receive a message before it can continue. If process A needs to receive a message from process B, then A can know that it is waiting for B. Thus, the processes have the necessary information to perform deadlock detection if they act collectively. In the resource model the dependence of one transaction on actions of other transactions is not directly known. All that is known is whether a transaction is waiting for a given resource or whether a transaction holds a given resource. A controller at each site keeps track of its resources and only the controllers can deduce that one transaction is waiting for another. Thus the agent of deadlock detection in the two environments is not the same.

The second major difference is that in a resource allocation model a process cannot proceed with execution until it receives all the resources for which it is waiting. In CSP and similar communication models, a process cannot proceed with its execution until it can communicate with at least one of the processes for which it is waiting. For instance, a process in CSP executing a guarded command may wait to receive from several processes; a guard succeeds and execution continues when a message is received from any one of these processes. The difference between the resource model and the communication model is between waiting for all resources and waiting for any one message; this difference results in very different algorithms for the two models.

In graph-theoretic terms, deadlock arises in the resource model when there is a cycle of (idle) dependent processes, whereas in the communication model there must be a knot 1 of (idle) waiting processes.

The communication model is more general than the resource model. In particular, the resource model can be simulated as a communication model. Furthermore, the communication model can handle the case where a process waits for a logical combination of resources, as in resource a and resource b or resource c.

9.4 DEADLOCK HANDLING

The problem of deadlocks can be handled in several ways: Prevention, Avoidance, and Detection. In prevention, some requirement of the system makes deadlocks impossible so that no runtime support is required. Avoidance schemes require decisions by the system while it is running to insure that deadlocks will not occur. Detection requires the most sophisticated runtime support: the system must find deadlocks and break them by choosing a suitable **victim** that is terminated or **aborted** and restarted if appropriate.

9.4.1 Deadlock Prevention

Prevention is the name given to schemes that guarantee that deadlocks can never happen because of the way the system is structured. One of the four conditions for deadlock is prevented, thus preventing deadlocks. One way to do this is to make processes declare all of the resources they might eventually

need, when the process is first started. Only if all the resources are available is the process allowed to continue. All of the resources are acquired together, and the process proceeds, releasing all the resources when it is finished. Thus, **hold and wait** cannot occur.

The major disadvantage of this scheme is that resources must be acquired because they might be used, not because they will be used. Also, the pre-allocation requirement reduces potential concurrency.

Another prevention scheme is to impose an order on the resources and require processes to request resources in increasing order. This prevents *cyclic wait* and thus makes deadlocks impossible.

One advantage of prevention is that process aborts are never required due to deadlocks. While most systems can deal with rollbacks, some systems may not be designed to handle them and thus must use deadlock prevention.

9.4.2 Deadlock Avoidance

In deadlock avoidance the system considers resource requests while the processes are running and takes action to insure that those requests do not lead to deadlock. Avoidance based on the **banker's algorithm**, sometimes used in centralized systems, is considered not practical for a distributed system. Two popular avoidance algorithms based on timestamps or priorities are **wound-wait** and **wait-die**. They depend on the assignment of unique global timestamps or priority to each process when it starts. Some authors refer to these as prevention.

In wound-wait, if process A requests a resource currently held by process B, their timestamps are compared. B is **wounded** and must restart if it has a larger timestamp (is younger) than A. Process A is allowed to wait if B has the smaller timestamp. Deadlock cycles cannot occur since processes only wait for older processes. In wait-die, if a request from process A conflicts with process B, A will wait if B has the larger timestamp (is younger). If B is the older process, A is not allowed to wait, so it **dies** and restarts.

In timeout based avoidance, a process is blocked when it requests a resource that is not currently available. If it has been blocked longer than a timeout period, it is aborted and restarted. Given the uncertainty of message delays in distributed systems, it is difficult to determine good timeout values.

These avoidance strategies have the disadvantage that the aborted process may not have been actually involved in a deadlock.

9.4.3 Deadlock Detection

Deadlock detection attempts to find and resolve actual deadlocks. These strategies rely on a Wait-For-Graph (WFG) that in some schemes is explicitly built and analyzed for cycles. In the WFG, the nodes represent processes and the edges represent the blockages or dependencies. Thus, if process A is waiting for a resource held by process B, there is an edge in the WFG from the node for process A to the node for process B.

In the AND model (resource model), a cycle in the graph indicates a deadlock. In the OR model, a cycle may not mean a deadlock since any of a set of requested resources may unblock the process. A *knot* in the WFG is needed to declare a deadlock. A knot exists when all nodes that can be reached from some node in a directed graph can also reach that node.

In a centralized system, a WFG can be constructed fairly easily. The WFG can be checked for cycles periodically or every time a process is blocked, thus potentially adding a new edge to the

WFG. When a cycle is found, a victim is selected and aborted.

9.4.4 Distributed Deadlock

A distributed system consists of a number of sites connected by a network. Each site maintains some of the resources of the system. Processes with a globally unique identifier run on the distributed system. They make resource requests to a controller. There is one controller per site. If the resource is local, the process makes a request of the local controller. If the desired resource is at a remote site, the process sends a message. After a process makes a request, but before it is granted, it is blocked and said to be **dependent** on the process that holds the desired resource.

The controller at each site could maintain a WFG on the process requests that it knows about. This is the local WFG. However, each site's WFG could be cycle free and yet the distributed system could be deadlocked. This is called **global deadlock**. This would occur in the following situation:

1. Process A at site 1 holds a lock on resource X.
2. Process A has requested, but has not been granted, resource Y at site 2.
3. Process B at site 2 holds a lock on resource Y.
4. Process B has requested, but has not been granted, resource X at site 1.

Both processes are blocked by the other one. There is a global deadlock. However, the deadlock will not be discovered at either site unless they exchange information via some detection scheme.

9.4.5 Distributed Detection Schemes

A detection scheme is evaluated by two criteria:

1. If there exist an actual deadlock, it must be detected in a finite amount of time, and;
2. The scheme must not find a deadlock that is not actually there.

One way to detect deadlocks in distributed systems is for each site to construct a local WFG on the part it knows about. A site that suspects deadlock initiates a global snapshot protocol that constructs a consistent global state of the distributed system. The global state corresponds to a deadlock if the union of the local WFGs has a cycle. This algorithm is correct because deadlock is a stable property.

9.4.5.1 The Probe Scheme

The scheme proposed by Chandy, Misra and Haas uses local WFGs to detect local deadlocks and **probes** to determine the existence of global deadlocks. If the controller at a site suspects that process A, for which it is responsible, is involved in a deadlock it will send a probe message to each process that A is dependent on. When A requested the remote resource, a process or *agent* was created at the remote site to act on behalf of process A to obtain the resource. This agent receives the probe message and determines the dependencies at the current site.

The probe message identifies process A and the path it has been sent along. Thus, the message probe (i,j,k) says that it was initiated on behalf of process i and was sent from the controller for agent j (which i is dependent on) to the controller for agent k.

When an agent whose process is not blocked receives a probe, it discards the probe. It is not blocked so there is no deadlock. An agent that is blocked sends a probe to each agent that it is blocked by. If process i ever receives probe (i,j,i), it knows it is deadlocked.

This popular approach, often called "edge-chasing", has been shown correct in that it detects all deadlocks and does not find deadlocks when none exist.

In the OR model, where only one out of several requested resources must be acquired, all of the blocking processes are sent a query message and all of the messages must return to the originator in order for a deadlock to be declared. The query message is similar to a probe, but has an additional identifier so the originator can determine whether or not all the paths were covered.

9.4.6 Distributed Deadlock Detection

- Deadlock prevention is achieved by denying the existence of one or more of the necessary conditions for deadlock. Commonly adopted strategies include :

1. Having a process acquire all needed resources simultaneously before it begins execution, or
2. Using a liner ordering scheme for resource acquisition to prevent circular waits. Unless information about resource availability and process requests are maintained centrally on one site, such an approach is impractical for a distributed system.

- In deadlock avoidance, a resource is granted to a process if the resulting state is safe. For a distributed system, the safe state checking must examine the global state that involves all processes and resources in all sites. This approach is impractical for distributed systems because :

(1) The need to maintain information on the global state for every site leads to huge storage requirements and extensive communication costs,

(2) The safe state checking will be computationally expensive due to the potentially large number of processes and resources involved, and

(3) The safe state checking for the global state must be mutually exclusive. If we allow multiple sites to perform safe state checking concurrently, each site for a different request, they may find the state safe but the net global state may not be.

- In light of the difficulty in deadlock detection for the general resource system, the study of deadlock problems in distributed systems usually adopts a simpler system model assuming:
- Reusable resources only
- Only exclusive accesses to resources
- Each resource has only one instance
- With the simpler system model as defined above, deadlock detection can be achieved by finding a cycle in the Wait For Graph (WFG) of the resource system.
- One major difficulty in deadlock detection in distributed systems is the potential for detecting phantom deadlocks.

9.5 ISSUES IN DEADLOCK DETECTION

Deadlock handling using the approach of deadlock detection entails addressing two basic issues: first, **detection of existing deadlocks** and, second, **resolution of detected deadlocks.**

9.5.1 Detection of deadlocks

Detection of deadlocks involves addressing two issues: maintenance of the WFG and searching

of the WFG for the presence of cycles or to be called knots. Since, in distributed systems, a cycle or knot may involve several sites, the search for cycles greatly depends upon how the WFG of the system is represented across the system. Depending upon the way WFG information is maintained and the search for cycles is carried out, there are centralized, distributed, and hierarchical algorithms for deadlock detection in distributed systems.

9.5.2 Correctness criteria

A deadlock detection algorithm must satisfy the following two conditions :

- **Progress (no undetected deadlocks) :** The algorithm must detect all existing deadlocks in a finite time. Once a deadlock has occurred, the deadlock detection activity should continuously progress until the deadlock is detected. In other words, after all wait-for dependencies for a deadlock have formed, the algorithm should not wait for any more events to occur to detect the deadlock.
- **Safety (no false deadlocks) :** The algorithm should not report deadlocks that do not exist (called phantom or false deadlocks). In distributed systems where there is no global memory and there is no global clock, it is difficult to design a correct deadlock detection algorithm because sites may obtain an out-of-date and inconsistent WFG of the system. As a result, sites may detect a cycle that never existed but whose different segments existed in the system at different times. This is the main reason why many deadlock detection algorithms reported in the literature are incorrect.

9.5.3 Resolution of a detected deadlock

Deadlock resolution involves breaking existing wait-for dependencies between the processes to resolve the deadlock. It involves rolling back one or more deadlocked processes and assigning their resources to blocked processes so that they can resume execution. Note that several deadlock detection algorithms propagate information regarding wait-for dependencies along the edges of the wait-for graph. Therefore, when a wait-for dependency is broken, the corresponding information should be immediately cleaned from the system. If this information is not cleaned in a timely manner, it may result in detection of phantom deadlocks. Untimely and inappropriate cleaning of broken wait-for dependencies is the main reason why many deadlock detection algorithms reported in the literature are incorrect.

9.5.4 Models of deadlocks

Distributed systems allow many kinds of resource requests. A process might require a single resource or a combination of resources for its execution. This section introduces a hierarchy of request models starting with very restricted forms to the ones with no restrictions whatsoever. This hierarchy shall be used to classify deadlock detection algorithms based on the complexity of the resource requests they permit.

9.5.4.1 The AND model

In the AND model, a process can request more than one resource simultaneously and the request is satisfied only after all the requested resources are granted to the process. The requested resources may exist at different locations. The out degree of a node in the WFG for AND model can be more than 1. The presence of a cycle in the WFG indicates a deadlock in the AND model. Each node of the WFG in such a model is called an AND node. Consider the example WFG described in the Figure 9.1. Process P11has two outstanding resource requests. In case of the AND model, P11shall become active from idle state only after both the resources are granted. There is a cycle P11? P21? P24? P54? P11, which corresponds to a deadlock situation.

In the AND model, if a cycle is detected in the WFG, it implies a deadlock but not vice versa. That is, a process may not be a part of a cycle, it can still be deadlocked. Consider process P44 in

Figure 9.1. It is not a part of any cycle but is still deadlocked as it is dependent on P24, which is deadlocked. Since in the single-resource model, a process can have at most one outstanding request, the AND model is more general than the single-resource model.

9.5.4.2 The OR model

In the OR model, a process can make a request for numerous resources simultaneously and the request is satisfied if any one of the requested resources is granted. The requested resources may exist at different locations. If all requests in the WFG are OR requests, then the nodes are called OR nodes. Presence of a cycle in the WFG of an OR model does not imply a deadlock in the OR model. To make it clearer, consider Figure 9.1. If all nodes are OR nodes, then process P11 is not deadlocked because once process P33 releases its resources, P32 shall become active as one of its requests is satisfied. After P32 finishes execution and releases its resources, process P11 can continue with its processing.

In the OR model, the presence of a knot indicates a deadlock. In a WFG, a vertex v is in a knot if for all u :: u is reachable from v : v is reachable from u. No paths originating from a knot shall have dead ends.

A deadlock in the OR model can be intuitively defined as follows:

A process Pii blocked if it has a pending OR requests to be satisfied. With every blocked process, there is an associated set of processes called dependent set. A process shall move from an idle to an active state on receiving a grant message from any of the processes in its dependent set. A process is permanently blocked if it never receives a grant message from any of the processes in its dependent set. Intuitively, a set of processes S is deadlocked if all the processes in S are permanently blocked. To formally state that a set of processes is deadlocked, the following conditions hold true:

1. Each of the process is the set S is blocked.
2. The dependent set for each process in S is a subset of S.
3. No grant message is in transit between any two processes in set S.

We now show that a set of processes S shall remain permanently blocked in the OR model if the above conditions are met. A blocked process P is the set S becomes active only after receiving a grant message from a process in its dependent set, which is a subset of S. Note that no grant message can be expected from any process in S because they are all blocked. Also, the third condition states that no grant messages in transit between any two processes in set S. So, all the processes in set S are permanently blocked.

Hence, deadlock detection in the OR model is equivalent to finding knots in the graph. Note that, there can be a deadlocked process that is not a part of a knot. Consider Figure 9.1, where P44 can be deadlocked even though it is not in a knot. So, in an OR model, a blocked process P is deadlocked if it is either in a knot or it can only reach processes on a knot.

9.5.4.3 The AND-OR model

A generalization of the previous two models (OR model and AND model) is the AND-OR model. In the AND-OR model, a request may specify any combination of AND and OR in the resource request. For example, in the AND-OR model, a request for multiple resources can be of the form x and (y or z). The requested resources may exist at different locations. To detect the presence of deadlocks in such a model, there is no familiar construct of graph theory using WFG. Since a deadlock is a stable property (i.e., once it exists, it does not go away by itself), this property can be exploited and a deadlock in the AND-OR model can be detected by repeated application of the test for OR-model deadlock. However, this is a very inefficient strategy.

9.5.4.4 Unrestricted model

In the unrestricted model, no assumption$ are made regarding the underlying structure of resource requests. In this model, only one assumption that the deadlock is stable is made and hence it is the most general model. This way of looking at the deadlock problem helps in separation of concerns: concerns

about properties of the problem (stability and deadlock) are separated from underlying distributed systems computations (e.g., message passing versus synchronous communication). Hence, these algorithms can be used to detect other stable properties as they deal with this general model. But, these algorithms are of more theoretical value for distributed systems since no further assumptions are made about the underlying distributed systems computations which leads to a great deal of overhead (which can be avoided in simpler models like AND or OR models).

9.6 CONTROLS IN DEADLOCK DETECTION

The following three control methods are common for deadlock detection in distributed systems:

9.6.1 Centralized Control

In centralized deadlock detection algorithms, a designated site (often called a *control site*) has the responsibility of constructing the global WFG and searching it for cycles. The control site may maintain the global WFG constantly or it may build it whenever a deadlock detection is to be carried out by soliciting the local WFG from every site.

Advantages of centralized algorithm include detection as well as resolutions are conceptually simple and easy to implement since the control site has complete information of the WFG and the deadlock cycle. Disadvantages include single point of failure, congestion likely at the control site and at communication links near that site, design not scalable to growth of system.

9.6.2 Distributed Control

In distributed deadlock detection algorithms, the responsibility of detecting a global deadlock is shared equally among all sites. Deadlock detection is initiated only when a waiting process is suspected to be part of a deadlock. When the detection of a global deadlock is to be carried out, several sites participate in the detection and the WFG is spread over many sites.

Distributed control is not vulnerable to a single point of failure, no site will become a bottleneck due to deadlock activities, and the design is much more scalable than the centralized organization. However, distributed deadlock detection algorithms are difficult to implement due to the lack of globally shared memory. Additional problems include: several sites may initiate detection of the same deadlock, deadlock resolution is often cumbersome since several sites may detect the same deadlock without being aware of it, and the proof of correctness of these algorithms is difficult.

9.6.3 Hierarchical Control

In hierarchical deadlock detection algorithms, sites are arranged in a hierarchical fashion, and a site detects deadlocks involving only its descendent sites. If the hierarchy structure coincides with resource access patterns local to clusters of sites, this approach can provide efficient detection of deadlocks and rip the benefits of both the centralized and the distributed control organizations. On the other hand, if deadlocks are not mostly localized to a few clusters but often span several clusters, this approach will be inefficient.

9.7 CENTRALIZED DEADLOCK-DETECTION ALGORITHMS

9.7.1 The Completely Centralized Algorithm

The completely centralized algorithm is the simplest centralized deadlock detection algorithm, wherein a designated site called the control site, maintains the WFG of the entire system and checks it for the existence of deadlock cycles. All sites request and release resources (even local resources) by sending request resource and release resource messages to the control site, respectively. When the

control site receives a request resource or a release resource message, it correspondingly updates its WFG. The control site checks the WFG for deadlocks whenever a request edge is added to the WFG.

9.7.2 The Ho-Ramamoorthy Algorithms

The Two-Phase Algorithm. In the two-phase algorithm, every site maintains a status table that contains the status of all processes initiated at that site. The status of a process includes all resources locked and all resources being waited upon. Periodically, a designated site requests the status table from all sites, constructs a WFG from the information received, and searches it for cycles. If there is no cycle, then the system is free from deadlocks, otherwise, the designated site again requests status tables from all the sites and again constructs a WFG using **only** those transactions which are common to both reports. If the same cycle is detected again, the system is declared deadlocked.

The One-Phase Algorithm. The one-phase algorithm requires only one status report from each site; however, each site maintains two status tables: a **resource status** table and a **process status** table. The resource status table at a site keeps track of the transactions that have locked or are waiting for resources stored at that site. The process status table at a site keeps track of the resources locked by or waited for by all the transactions at that site. Periodically, a designated site requests both tables from every site, constructs a WFG using only those transactions for which the entry in the resource table matches the corresponding entry in the process table, and searches the WFG for cycles. If no cycle is found, then the system is not deadlocked, otherwise a deadlock is detected.

The one-phase algorithm is faster and requires fewer messages as compared to the two-phase algorithm. However, it requires more storage because every site maintains two status tables and exchanged bigger messages because a message contains two tables instead of one.

9.7.3 Path-Pushing Algorithm

In path-pushing deadlock detection algorithms, information about the wait-for dependencies is propagated in the form of paths. Obermarck's algorithm is chosen to illustrate a path-pushing deadlock detection algorithm.

Obermarck's algorithm has two interesting features:

- The nonlocal portion of the global Transaction Wait-For (TWF) graph at a site is abstracted by a distinguished node (called External or Ex) which helps in determining potential multisided deadlocks without requiring a huge global TEF graph to be stored at each site.
- Transactions are totally ordered, which reduces the number of messages and consequently decreases deadlock detection overhead. It also ensures that exactly one transaction in each cycle detects the deadlock.

The Algorithm

Deadlock detection at a site follows the following iterative process:

1. The site waits for deadlock-related information (produced in Step 3 of the previous deadlock detection iteration) from other sites. (Note tht deadlock-related information is passed by sites in the form of paths.)
2. The site combines the received information with its local TWF graph to build an updated TWF graph. It then detects all cycles and breaks only those cycles which do not contain the node 'Ex'. Note that these cycles are local to this site. All other cycles have the potential to be a

part of global cycles.

3. For all cycles 'Ex T_1 T_2 Ex' which contain the node 'Ex' (these cycles are potential candidates for global deadlocks), the site transmits them in string form 'Ex, T_1, T_2, Ex' to all other sites where an agent of T_2 is waiting for a resource being held by another transaction. The algorithm reduces message traffic by lexically ordering transactions and sending the string 'Ex, T_1, T_2, T_3, Ex' to other sites only if T_1 is higher than T_3 in the lexical ordering. Also, for a deadlock, the highest priority transaction detects the deadlock.

9.7.4 An Edge-Chasing Algorithm

Chandy et al.'s algorithm uses a special message called a probe. A probe is a triplet (*i, j, k*) denoting that it belongs to a deadlock detection initiated by process P_i and it is being sent by the home site of process P_j to the home site of process P_k. A probe message travels along the edges of the global TWF graph, and a deadlock is detected when a probe message returns to its initiating process. The system maintains a Boolean array, dependent, for each process P_i, where $dependent_i(j)$ is true only if P_i knows that P_j is dependent on it. Initially, $dependent_i(j)$ is false for all *i* and *j*.

The Algorithm

To determine if a blocked process is deadlocked, the system executes the following algorithm:

if P_i is locally dependent on itself

then declare a deadlock

else for all P_j and P_k such that

a) P_i is locally dependent upon P_j, and

b) P_j is waiting on P_k, and

c) P_j and P_k are on different sites,

send probe (*i, j, k*) to the home site of P_k

On the receipt of probe (i, j, k), the site takes the following actions:

if

a) P_k is blocked, and

b) $dependent_k(i)$ is false, and

c) P_k has not replied to all requests of P_j

then

begin

$dependent_k(i)$:= true;

if $k = i$

then declare that P_i is deadlocked

else for all P_m and P_n such that

a) P_k is locally dependent upon P_m, and

b) P_{im} is waiting on P_{jn}, and

c) P_{im} and P_{jn} are on different sites,

send probe (i, m, n) to the home site of P_{jn}

end.

Thus, a probe message is successively propagated along the edges of the global TWF graph and a deadlock is detected when a probe message returns to its initiating process. Chandy et al.'s algorithm sends one probe message (per deadlock detection initiation) on each edge of the WFG, which spans two sites. Thus, the algorithm at most exchanges $m(n-1)/2$ messages to detect a deadlock that involves m processes and spans over n sites. The size of messages exchanged is fixed and very small (only 3 integer words). The delay in detecting the deadlock is $O(n)$.

9.8 DEADLOCK RESOLUTION

A deadlock is resolved by aborting one or more processes involved in the deadlock and granting the released resources to other processes involved in the deadlock. The efficient resolution of a deadlock requires knowledge of all the processes involved in the deadlock and all the resources held by these processes. Resolution of deadlocks detected using distributed control is complicated due to the following issues:

- A process that detects a global deadlock does not know all the processes involved in a deadlock and the resources held by them.
- Two or more processes may independently detect the same deadlock and then independently resolves it. Consequently several processes will be aborted (likely more than necessary) in an attempt to resolve a deadlock, resulting in inefficiency.

Obermarck's suggestion of assigning unique priorities to transactions/processes and using these priorities in detection as well as resolution as follows has helped resolved these problems.

- Each deadlock is detected only by the highest priority process in the deadlock (i.e., deadlock detections initiated by all other processes involved in the deadlock are suppressed.)
- When the highest priority process detects a deadlock, it knows the lowest priority processes in the cycle, which can be aborted to resolve the deadlock.

It should be noted that deadlock resolution is a nontrivial procedure that involves the following steps:

- The victim must be aborted, all the resources held by it must be released, the state of all the released resources must be restored to their states prior to the allocation, and the released resources must be granted to other processes involved in the deadlock.
- All the deadlock detection information concerning the victim must be cleaned at all the sites.

9.9 SUMMARY

In computer science, **deadlock** refers to a specific condition when two or more processes are each waiting for each other to release a resource, or more than two processes are waiting for resources in a circular chain.

There are several differences between the resource model and the communication model. One critical difference is that in the communication model, a process can know the identity of those processes from which it must receive a message before it can continue.

The second major difference is that in a resource allocation model a process cannot proceed with execution until it receives all the resources for which it is waiting.

The problem of deadlocks can be handled in several ways: Prevention, Avoidance, and Detection. In prevention, some requirement of the system makes deadlocks impossible so that no runtime support is required. Avoidance schemes require decisions by the system while it is running to insure that deadlocks will not occur. Detection requires the most sophisticated runtime support: the system must find deadlocks and break them by choosing a suitable *victim* that is terminated or *aborted* and restarted if appropriate.

A distributed deadlock detection scheme is evaluated by two criteria:

1) If there exist an actual deadlock, it must be detected in a finite amount of time, and;

2) The scheme must not find a deadlock that is not actually there.

Deadlock handling using the approach of deadlock detection entails addressing two basic issues: first, **detection of existing deadlocks** and, second, **resolution of detected deadlocks.**

There are four types of deadlock models (1) AND model (2) OR model (3) AND – OR model and (4) Unrestricted model.

The following three control methods are common for deadlock detection in distributed systems: (1) Centralized Control (2) Distributed Control (3) Hierarchical Control.

9.10 UNIT-END QUESTIONS

1. What is system model for deadlocks? Explain
2. Compare resource and communication deadlock.
3. Discuss the issues in deadlock detection and resolution.
4. Explain distributed deadlock detection algorithm.
5. Explain Ho-Ramamoorthy Algorithm.
6. Explain path-pushing deadlock detection algorithm.

9.11 FURTHER READINGS

1. "A Distributed Algorithm for Detecting Resource Deadlocks in Distributed Systems" by S.N. Maheshwari
2. "Foundations of software technology and theoretical computer science", fourth edition By M. Joseph, Rudrapatna Shyamasundar.
3. "Distributed systems : concepts and design" by George F. Coulouris, Jean Dollimore, Tim Kindberg.

□□□□

UNIT 10 : DISTRIBUTED FILE SYSTEMS

Structure of the Unit

10.0 OBJECTIVE

The purpose of a distributed file system (DFS) is to allow users of physically distributed computers to share data and storage resources by using a common file system. A typical configuration for a DFS is a collection of workstations and mainframes connected by a local area network (LAN). A DFS is implemented as part of the operating system of each of the connected computers. This chapter establishes a viewpoint that emphasizes the dispersed structure and decentralization of both data and control in the design of such systems. It defines the concepts of cache consistency, availability, scalability, and semantics and discusses them in the context of DFSs. The Chapter claims that the principle of distributed operation is fundamental for a fault tolerant and scalable DFS design. It also presents alternatives for the semantics of sharing and methods for providing access to remote files.

In this chapter we define simple architecture for file systems and describe one the two basic distributed file service implementations with contrasting designs that have been in widespread use for last few decades –

1. Sun Network File System, NFS;

2. Andrew File System, AFS

In this Chapter, our focus is only on the 'Sun Network File System'.

10.1 INTRODUCTION

Distributed file systems are generally employed for storage of large quantities of data and to reduce input/output (I/O) bottlenecks where there are many requests made for file access. In a distributed file system, the file data is spread across multiple data processing systems. File system control and management of file system meta-data is distributed in varying degrees in different systems. Sharing of information is the most important aspect of distributed resource sharing. There are several mechanisms which are commonly used for data and information sharing. File system administration for a large, growing computer installation built with today's technology is a laborious task. To hold more files and serve more users, one must add more disks, attached to more machines. Each of these components requires human administration. Groups of files are often manually assigned to particular disks, then manually moved or replicated when components fill up, fail, or become performance hot spots. Joining multiple disk drives into one unit using RAID technology is only a partial solution; administration problems still arise once the system grows large enough to require RAIDs and multiple server machines.

File Management is the most important service which is provided by the Operating system. A distributed file system allowing users to access files from any computer on a network. In other words, a 'Distributed File System' enables programs to store and access remote files exactly as they do local ones. The performance and reliability experienced for access to files stored at a server should be comparable to files stored on local disks. A distributed file system manages a collection of disks on multiple machines as a single shared pool of storage. The machines are assumed to be under a common administration and to be able to communicate securely. In other words, the ideal distributed file system would provide all its users with coherent, shared access to the same set of files, yet would be arbitrarily scalable to provide more storage space and higher performance to a growing user community. It would require minimal human administration, and administration would not become more complex as more components were added. There are some key features of a distributed file system-

1. All users are given a consistent view of the same set of files.
2. More servers can easily be added to an existing system to increase its storage capacity and throughput, without changing the configuration of existing servers, or interrupting their operation. The servers can be viewed as 'bricks' that can be stacked incrementally to build as large a file system as needed.
3. A system administrator can add new users without concern for which machines will manage their data or which disks will store it.
4. A system administrator can make a full and consistent backup of the entire file system without bringing it down. Backups can optionally be kept online, allowing users quick access to accidentally deleted files.
5. The file system tolerates and recovers from machine, network and disk failures without operator intervention.

DFS allows a single network connection to branch off into any number of other shares on the network, which may even be on a different file server. In its simplest form, DFS allows an administrator

to hide the structure of the network from the users.

10.2 DFS ARCHITECTURE

The need to share resources in a computer system arises due to economics or the nature of some applications. In such cases, it is necessary to facilitate sharing long-term storage devices and their data. This chapter discusses 'Distributed File Systems' (DFSs) as the means of sharing storage space and data.

A file system is a subsystem of an operating system whose purpose is to provide to facilitate sharing long-term storage devices and their data. It does so by implementing files-named objects that exist from their explicit creation until their explicit destruction and are immune to temporary failures in the system. A DFS is a distributed implementation of the classical time sharing model of a file system, where multiple users share files and storage resources. The UNIX time-sharing file system is usually regarded as the model. The purpose of a DFS is to support the same kind of sharing when users are physically dispersed in a distributed system.

A 'Distributed System' is a collection of loosely coupled machines- either a mainframe or a workstation – interconnected by a communication network. Unless specified otherwise, the *network* is a Local Area Network (LAN). From the point of view of a specific machine in a distributed system, the rest of the machines and their respective resources are *remote* and the machine's own resources are *local.*

To explain the structure of a DFS, we need to define service, server, and client. A 'service' is a software entity running on one or more machines and providing a particular type of function to a priori unknown clients. A 'server' is the service is software running on a single machine. A 'client' is a process that can invoke a service using a set of operations that form its *client interface*. Sometimes, a lower level interface is defined for the actual cross-machine interaction. When the need arises, we refer to this interface as the inter-machine interface. Clients implement interfaces suitable for higher level applications or direct access by humans.

Using the above terminology, we say a file system provides file services to clients. A client interface for a file service is formed by a set of file operations. The most primitive operations are Create a file, Delete a file, Read from a file, and Write to a file. The primary hardware component a file server controls is a set of secondary storage devices (i.e., magnetic disks) on which files are stored and from which they are retrieved according to the client's requests. We often say that a server, or a machine, stores a file, meaning the file resides on one of its attached devices.

A DFS is a file system, whose clients, servers, and storage devices are dispersed among the machines of a distributed system. Accordingly, service activity has to be carried out across the network, and instead of a single centralized data repository there are multiple and independent storage devices. As will become evident, the concrete configuration and implementation of a DFS may vary. There are configurations where servers run on dedicated machines, as well as configurations where a machine can be both a server and a client. A DFS can be implemented as part of a distributed operating system or, alternatively, by a software layer whose task is to manage the communication between conventional operating systems and file systems. The distinctive features of a DFS are the multiplicity and autonomy of clients and servers in the system.

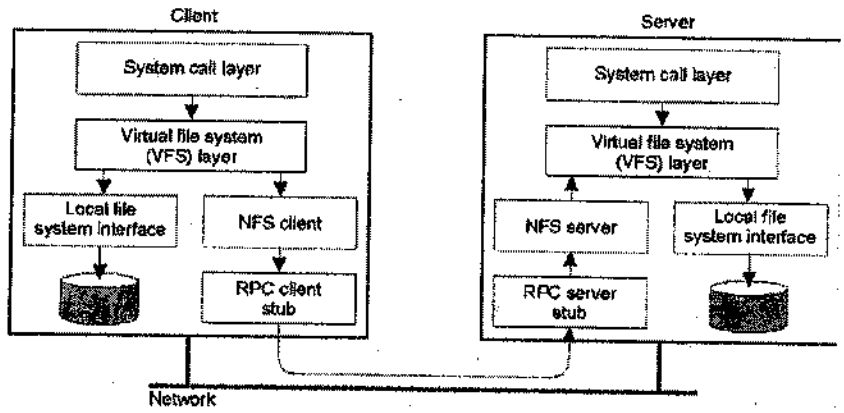


Figure 10.1: A DFS Architecture

The chapter is mainly divided into two parts. In the first part, the basic concepts underlying the design of a DFS will be discussed. In particular, alternatives and trade-offs regarding the designs of a DFS are pointed out. The second part surveys one DFS i.e. Sun's Network File System (NFS). This system exemplifies the concepts and observations mentioned in the first part and demonstrate various implementations.

Check your progress 10.1

1. A distributed system is a collection of –
(a) Tightly coupled machines (b) loosely coupled machines
(c) Both a and b (d) None of above

2. _____ is a example of DFS.
(a) SUN (b) Windows
(c) DOS (d) None of above

3. A _____ is a file system, whose clients, servers, and storage devices are dispersed among the machines of a distributed system.
(a) Operating System (b) Distributed File System
(c) Distributed Shared Memory (d) None of above

4. A _____ is a process that can invoke a service using a set of operations that form its *client interface*.
(a) Client (b) Server
(c) Both-Client as well as Server (d) None of above

5. A _____ allowing users to access files from any computer on a network.
(a) Distributed File System (b) Distributed Shared Memory
(c) Distributed Process System (d) None of above

10.3 MECHANISM FOR BUILDING DFS

Mounting, Caching and Bulk data transfer are a few mechanisms for building a distributed file system. In a client-Server system, each with main memory and a disk, there are four places to store files –

1. The Server's Disk;
2. The Server's Main Memory;
3. The Client's Disk;
4. The Client's Main Memory.

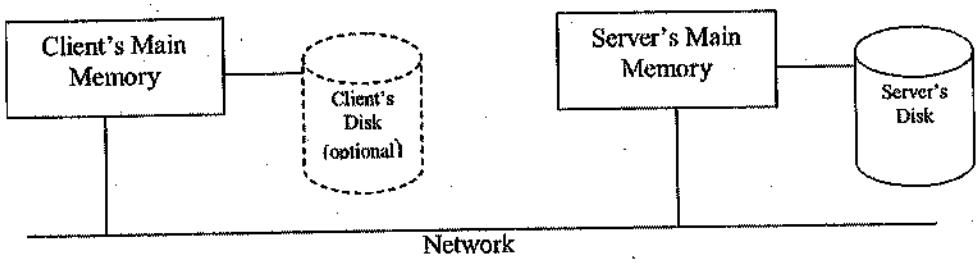


Figure 10.2: File storage places in a Distributed Environment

The most straightforward palce to store all files is on the server's disk. The problem with using the server's disk is performance. Before a client can read a file, the file must be tranferred from the server's disk to the server's main memory, and them again over the network to the client's main memory. Both transfers take time.

A considerable performance gain can be achieved by caching the most recently used files in the server's main memory. A client reading a file that happens to be in the server's cache eliminates the disk transfer, although the network transfer still has to be done. Since main memory is invariably smaller than the disk, some technique is helpful in this situation. One or more cache management technique(s) are implemented in a distributed file system of a computer network to enforce equitable use of the cache among the file data in the cache.

- One technique is a timestamp handicapping routine that functions to keep small-to-medium files in the cache.

- Another technique implements a "cache quota", which limits the percentage of the cache a single file may consume when there is other data in the cache. When caching of a single file approaches the cache quota, the file data is made to look older than it really is so that upon a subsequent file I/O operation, portions of such data are recycled from the cache earlier than they would have been otherwise. When caching of a single file reaches the cache quota, the file must begin reusing cache from itself or from handicapped chunks from other files. The cache quota technique has the effect of causing cached data towards the end of large files to get flushed or recycled from the cache first.

- A third technique helps to detect file I/O that is not conducive to caching, such as sequential I/O on a file that is larger that the entire cache. A cache policy prevents the large file from stealing cache space by establishing a small, but dedicated area of cache in which portions of such large file may be stored and recycled without requiring a least recently used (LRU) evaluation process.

Having a cache in the server's main memory is easy to do and totally transparent to the clients.

10.4 DESIGN ISSUES

10.4.1 Naming and name resolution

Naming : An operating system maintains a namespace for a single system. Examples of the objects it supports are processes, files and I/O streams. Names are defined in a context. It may be that a distributed system design is based on a homogeneous operating system which supports a distributed object model. Alternatively, we may have to work in a heterogeneous world and devise a naming scheme for the distributed application we wish to build. For example, names may be used only within the context of a distributed file service, mail service, news service or bank account management

service. It is useful to model this as a type manager naming objects of that type. In all cases each object named must have a unique name within the context in which it is used.

Uniqueness may be achieved through so-called unique identifiers or by using a hierarchical naming scheme. The namespace is a specified number of bits, so a UID is a number 0 to $2^N - 1$ for an N-bit UID. Thirty-two, 64 and 128 bits are typical choices. A UID is never reused and a given bit pattern either refers to the same object at all times or to no object at all. Uniqueness is achievable by using a hierarchy as well as a long bit pattern, for example, `abccl.cam.ac.uk` is a unique name for a computer. A manager of the domain `cl` must ensure that abccl is unique in that context; the manager of the domain `cam` must ensure that the name `cl` is unique in that context, and so on.

A **pure name** yields no information such as the location of the named object, or the context in which the name is to be resolved, i.e. looked up. An example is a UID which is interpreted as a flat bit pattern with no internal structure. All we can do with a pure name is compare it with other names of that type, for example in table lookup.

The major problem with pure names is therefore to know where to look them up. It might be that a pure name refers to nothing, but how do we avoid a global search to be sure that no object with that name exists?

An impure name yields information and commits the system to maintaining the context in which the name is to be resolved, for example:

puccini.cl.cam.ac.uk (a computer)

aman@india.lcs.mit.edu (a registered user)

bright@cl.cam.ac.uk (a registered user)

We have examples here of location-dependent names. Note that there is no way of telling from the names that they both refer to the same person: bright is unique within the domain cl; aman is unique within the domain india.

The major problems with impure names are object mobility and the difficulty of restructuring a hierarchical namespace. If an object changes its resolution context by moving to a new location then its name also changes. A hierarchical namespace may be designed to reflect the structure of an organization. It may be desirable to restructure within an organization, by merging departments or creating new ones, or one company may merge with or purchase another. Many names will change if the new structure is to be reflected in the name hierarchy. This problem was tackled in the design of the Global Names Service (GNS).

The Internet Domain Name Service (DNS): An outline of this familiar name service will help to introduce the issues of name service provision. In practice, the objects named are computers, mail hosts and domains.

Examples of domain names are:

uk

ac.uk

cam.ac.uk

An example of a computer name is: puccini.cl.cam.ac.uk

An example of a mail host name is: swan.cl.cam.ac.uk

The type of the object named is therefore not apparent from the name itself.

Below a notional root are top-level domain names such as:

com (US companies)

edu (US academic institutions)

gov (US government)

net (network management)

org (organizations)

int (international)

uk (the United Kingdom's root)

fr (the root for France)

... and so on ...

Although the namespace is badly designed it is impossible for it to be restructured without invalidating huge numbers of existing names, for example for all names that end edu to be renamed to end edu.us. A domain has a manager but the management role may be delegated to subdomains. It is the responsibility of the manager of a domain to ensure that unique names are assigned within it. A directory is a sequence of names, each with an associated list of values (attributes). Users may query DNS to obtain the attributes associated with a given name. The DNS namespace is huge. Before 1989 a naming database was held centrally and was downloaded into selected hosts periodically. When the scale of the internet made this approach impossible, DNS was introduced. The domain database is partitioned (as are all large-scale naming databases) into directories and the directories are replicated at suitable locations on the internet.

To resolve a multi-component name such as puccini.cl.cam.ac.uk:

ac is looked up in the directory for the uk domain

cam is looked up in ac

cl is looked up in cam

puccini is looked up in cl.

If these directories were stored on different computers, four computers would have to be contacted to resolve the name. These directories need not necessarily be held at different locations, i.e. heavily used directories may be stored and replicated strategically to avoid many computers being contacted to resolve a multi-component name.

As we have seen, name resolution could be a lengthy process and to increase efficiency a resolved name is likely to be cached by your local software. Also, because many users are using DNS simultaneously there is scope for batching queries and responses. Updates are made locally by the domain manager. This makes the local copy of that domain directory up to date but all other copies have become out of date. Changes are propagated to all copies which will be up to date in due course.

Namespace : A namespace is a collection of valid names recognized by a name service, for

example:

- a 128-bit UID;
- a pathname in a filing system such as `/a/b/c/d` in UNIX;
- a DNS multicomponent name such as `abbc1.cl.cam.ac.uk`.

The structure (syntax) of names must be specified.

A naming domain is a namespace for which there exists a single administrative authority for assigning names within it. This authority may delegate name assignment for nested subdomains. For example, we have seen within DNS that the manager of the domain `ac.uk` may delegate management of the subdomain `cam.ac.uk`.

The values (attributes) associated with the names for a domain are held in a directory for that domain. The name service is offered by a number of name servers which hold the naming directories and respond to queries.

Name Resolution :

This is the process of obtaining a value which allows an object to be used; for example, to determine the network address of a named computer or the network addresses of the computers which receive mail (hold the mailbox) for a named individual.

For large-scale systems the naming database is invariably partitioned, replicated and distributed among the name servers, so resolving a multipart name is an iterative process which involves navigation among name servers.

It is good programming practice to bind late and to embed unresolved names, not addresses, in programs.

A name service maintains an information database. The fundamental requirement is to hold the locations of various types of objects such as users and resources so that they can be contacted or used. A number of attributes may be associated with a given name and a query must specify both the name and which attribute is required.

An alternative form of query, which may or may not be supported by a name service, is the attribute-based query. Here an attribute is supplied and a list of names with that attribute is returned. This is sometimes referred to as a yellow pages service, although a naming service may have the functionality of both the white pages (directories) and yellow pages services of the telephone companies. We shall focus on the white pages or directory function.

Name services have typically held the following types of object and associated attributes:

user:	login name, a list of mail hosts for that user, ordered by preference, authenticator, e.g. password (usually held elsewhere)
computer:	architecture, OS, network address, owner
service:	list of <network address, version #, access protocol>
group:	list of names of members of the group
alias:	name
(directory:	list of computers which hold that directory)

It may be that, as an optimization to avoid several queries, the network addresses of the mail hosts, etc., are stored and returned on the first query.

The type of service listed here is likely to be stable, long lived and widely used, such as a mail service. We shall see later that distributed programs that are short lived and 'private' will tend to use a special-purpose name service for name-to-location mapping.

A design choice is whether a directory should be treated as just another type of named object or whether directories should form a separate structure. In either case, a directory name will resolve to a list of computer names (plus network addresses) which hold that directory.

It may be argued that making a great deal of information freely available is bad for system security. It could be useful for a potential intruder to be able to look up which OS and which version of a given service are running.

DFS provides DFS Namespaces and DFS Replication, which together enable fault-tolerant access to distributed files with low-bandwidth replication:

1. **DFS Namespaces :** Formerly known as the Distributed File System, the DFS Namespaces service allows administrators to group shared folders that are located on different servers. Folders are presented as a virtual tree called a namespace, so users no longer have to remember physical file locations.

2. **DFS Replication :** A follow-up to the File Replication Service that was introduced in the Microsoft Windows 2000 Operating System, the DFS Replication service is designed to address low network bandwidth using the remote differential compression (RDC) algorithm. RDC enables DFS Replication to transfer only the changes that have been made since the last file update. Another feature, cross-file RDC, identifies files that are similar to the one being replicated. DFS Replication can then use portions of those similar files to replicate the file, helping reduce the amount of data transferred over the network.

The namespace service of a distributed file system provides client applications with location information for the various files in the file system. The location information includes, for example, a server identifier, a storage element identifier, and a storage address. Since a distributed file system generally supports multiple client applications, the namespace service includes logic to maintain coherency and consistency of the namespace data. The coherency and consistency logic may present barriers to the scalability of a distributed file system.

10.4.2 Cache on disk / main memory

Any centralized entity, be it a central controller or a central data repository, introduces both a severe point of failure and a performance bottleneck. Therefore, a scalable and fault-tolerant DFS should have multiple and independent servers controlling multiple and independent storage devices.

The fact that a DFS manages a set of dispersed storage devices is its key distinguishing feature. The overall storage space managed by a DFS consists of different and remotely located smaller storage spaces. Usually there is correspondence between these constituent storage spaces and sets of files.We use the term componentunit to denote the smallest set of files that can be stored on a single machine, independently from other units. All files belonging to the same component unit must reside in the same location.

10.4.3 Cache Consistency

Client caching introduces inconsistency into the system. If two clients simultaneously read the same file and then both modify it, several problems occur. For one, when a process reads the file from the server, it will get the original version. Effects of modifying a file are not supposed to be visible globally until the file is closed. This 'incorrect' behavior is simply declared to be the 'correct' behavior. To solve the consistency problem there are a number of algorithms used. One of them is **'Write-through Algorithm'**. According to this algorithm - When a cache entry is modified, the new value is kept in the cache, but is also sent immediately to the server. As a consequence, when another process reads the file, it gets the most recent value. Suppose that a client process on machine *A* reads a file *f*. The client terminates but the machine keeps *f* in its cache. Later, a client on machine *B* reads the same file, modifies it, and writes it through to the server. Finally, a new client process is started up on machine *A*. the first thing it does is open and read *f*, which is taken from the cache. Unfortunately, the value there is now obsolete.

Solution – A possible way out is to require the cache manager to check with the server before providing any client with a file from the cache. This check could be done by comparing the time of last modification of the cached version with the server's version. If they are the same, the cache is up-to-date. If not, the current version must be fetched from the server. Instead of using dates, version numbers or checksums can also be used. Although going to the server to verify dates, version numbers, or checksums takes an RPC, the amount of data exchanged is small. Still, it takes some time.

10.4.4 Availability

The term **availability** refers to the accessibility of a file. If a client requests access to a file that the client has the right to access and is unable to access the file, then the file is said to be unavailable. The fraction of time that legal operations can be performed on a file by a client defines the availability of the file. Failures and/or restrictions imposed by a file system, can make a file available to some clients and not available to others; in this manner a file that n clients have access to has a lower availability than a file that m clients are able to access when $m > n$.

Failures reduce the availability of a file by making it impossible for clients' file requests to be granted. In a system free of failures, a client should always be allowed access to a file unless the file is locked or the client does not have the required privileges. This paper does not consider the lack of availability due to locking performed by an application or due to a user's lack of privileges. Availability is then a measure of a file system's ability to serve requests when failures are present.

10.4.5 Scalability

A desirable characteristic of many distributed file systems is scalability. Scalability is a characteristic that refers to the ease with which a distributed file system can be expanded to accommodate increased data access needs or increased storage needs. For example, as additional users are granted access to the distributed file system, new storage servers may be introduced, and the requests of the additional users may be further spread across the old servers and new servers. The scalability of any distributed file system is limited or enhanced by the system design. Scaling a distributed file system is complicated by the fact that the architecture of the distributed file system may possess inherent bottlenecks that limit the extent to which the system can benefit from additional computation and storage capacity.

10.4.6 Semantics

A principal issue when designing a file system is its behavior in the presence of:

1. **Concurrent Conflicting Requests and**
2. **Failures.**

This issue is identified as the **file semantics** issue. Four types of commonly used file semantics are: **UNIX semantics, session semantics, immutable files** and **transactional semantics.**

- **UNIX Semantics : When** one uses basic file operations as the units to be serialized, and these operations are constrained by real-time consistency, **UNIX semantics** result. Under UNIX semantics update operations are immediately visible to all read operations that follow and are always applied to a copy of the file that reflects all previous updates. The strictness of UNIX semantics can hinder the efficiency of distributed file systems due to the overhead in guaranteeing all updates on a file are reflected in all copies of the file (i.e., cached copies, and server replicas), and in guaranteeing that the updates are applied in the same order at all servers.

- **Session Semantics :** Under session semantics, each file session gets a logical copy of the current version of the file. The current version of a file reflects the updates from closed file sessions, and none of the updates from ongoing file sessions. All read and write accesses for the file session are performed on this copy. When the file session ends, the logical copy becomes the current version of the file (if the session updated the file). An update file session, S1, will have its updates completely obliterated when another updating file session S2 closes, if S1 opens after S2 starts and closes before S2 closes, even if the two sessions update non-overlapping sections of the file.

Session semantics are not always defined quite so strictly. File sessions on a single machine may or may not share a copy of the file; it depends on the interpretation by the implementor. Sharing the local copy introduces two questions, one on open and one on close. When the second session starts, would the version brought from the server destroy any updates to the cached version that were performed by the first session? When the first close is received, is the file written back to the server, or are the updates kept locally until all sessions have closed?

- **Immutable Files :** Similar to session semantics, each file session is given a logical copy of the file being accessed and updates to the file are only noticeable to other sessions after the close. The difference is that an update session creates a new version of the file so that both the old version and the new version are present and accessible. An open system call can request an old version so that any version of the file is accessible (if all versions are kept).This method partially overcomes the problem of session semantics where the effects of some sessions can be obliterated. A session's effect will not be obliterated, but it may not be reflected in the most current version of the file. Immutable files also require extra disk space to store the multiple versions of each file.

- **Transactional Semantics :** Transactional semantics require that a set of file sessions be serializable, and that any given file session appear to be an *atomic action*. A set of file sessions executed concurrently is serializable if the outcome of the execution is the same as the outcome of some serial execution of the same set of file sessions. Atomicity of file sessions guarantees that either all of the actions of a file session will be performed, or none of them will, and that the actions appear to be instantaneous, no transient state of the file is noticeable. Each file session can be equated to a transaction in a database environment.

Check your progress 10.2

1. _____ is a characteristic that refers to the ease with which a distributed file system can be expanded to accommodate increased data access needs or increased storage needs.
(a) Scalability (b) Availability
(c) Consistency (d) None of above

2. Which service of a distributed file system provides client applications with location information for the various files in the file system.
(a) Cache consistency (b) Scalability
(c) Namespace service (d) None of above

3. The term _____ refers to the accessibility of a file.
(a) Consistency (b) Availability
(c) Cache (d) All of above

4. _____ process of obtaining a value which allows an object to be used.
(a) Name Resolution (b) Server
(c) Both (d) None of above

5. A _____ is a namespace for which there exists a single administrative authority for assigning names within it.
(a) Naming Domain (b) Cache
(c) Both (d) All of above

10.5 SUN NETWORK FILE SYTEM: A CASE STUDY

The Network File System (NFS) is a name for both an implementation and a specification of a software system for accessing remote files across LANs. The implemenation is part of the SunOS operating system, which is a flavor of UNIX running on Sun workstations using an unreliable datagram protocol (UDP/IP protocol) and Ethernet. The specification and implementation are interwined in the following description; whenever a level of details is needed we refer to the Sun OS implemention, and whenever the description is general enough it also applies to the specification.

The system is presented in three levels of detail. First (Section 10.5.1), an overview is given. Then, two service protocols that are the building vlocks for the implemention are examined (Section 10.5.2). Finally (in Section 10.5.3), a description of the SunOS implementation is given.

10.5.1 Overview

NFS views a set of interconnected workstations as a set of independent machines with independent file systems. The goal is to allow some degree of sharing among these file systems in a transparent manner. Sharing is based on server-client relationship. A machine may be, and often is, both a client and a server. Sharing is allowed between any pair of machines, not only with dedicated server machines. Consistent with the independence of a machine is the critical observation that NFS sharing of a remote file system affects only the client machine and no other machine. Therefore, there is no notion of a globally shared file system as in Locus, Sprite, UNIX United, and Andrew.

To make a remote directory accessible in a transparent manner from a client machine, a user of that machine first has to carry out a mount operation. Actually, only a super-user can invoke the

mount operation. Specifying the remote directory as an argument for the mount operation is done in a nontransparent manner; the location (i.e., hostname) of the remote directory has to be provided. From then on, users on the client machine can access files in the remote directory in a totally transparent manner, as if the directory were local. Since each machine is free to configure its own name space, it is not guaranteed that all machines have a common view of the shared space.

The convention is to configure the system to have a uniform name space. By mounting a shared file system over user home directories on all the machines, a user can log in to any workstation and get his or her home environment. Thus, user mobility can be provided, although again by convention.

Subject to access rights accreditation, potentially any file system or a directory within a file system can be remotely mounted on top of any local directory.

In the latest NFS version, diskless workstations can even mount their own roots from servers. In previous NFS versions, a diskless workstation depends on the Network Disk (ND) protocol that provides raw block I/O service from remote disks; the server disk was partitioned and no sharing of root file systems was allowed. One of the design goals of NFS is to provide file services in a heterogeneous environment of different machines, operating systems, and network architecture. The NFS specification is independent of these media and thus encourages other implementations. This independence is achieved through the use of RPC primitives built on top of an External Date Representation (XDR) protocol. Hence, if the system consists of heterogeneous machines and file systems that are properly interfaced to NFS, file systems of different types can be mounted both locally and remotely.

10.5.2 NFS Services

The NFS specification distinguishes between the services provided by a mount mechanism and the actual remote file access services. Accordingly, two separate protocols are specified for these services a mount protocol and a protocol for remote file accesses called the NFS protocol. The protocols are specified as sets of RPCs that define the protocols' functionality. These RPCs are the building blocks used to implement transparent remote file access.

10.5.2.1 Mount Protocol

We first illustrate the semantics of mounting by a series of examples. In Figure 10.3 (a), the independent file systems belonging to the machines named client, server1, and server2 are shown. At this stage, at each machine only the local files can be accessed. The triangles in the figure represent subtrees of directories of interest in this example. In Figure 10.3 (b), the effects of the mounting server 1:/usr/shared over client:/usr/local are shown. This figure depicts the view users on client have of their file system. Observe that any file within the dir1 directory, for instance, can be accessed using the prefix /usr/local/ dir1 in client after the mount is complete.

The original directory /usr/local on that machine is not visible any more. Cascading mounts are also permitted. That is, a file system can be mounted over another file system that is not a local one, but rather a remotely mounted one. A machine's name space, however, is affected only by those mounts the machine's own superuser has invoked. By mounting a remote file system, access is not gained for other file systems that were, by chance, mounted over the former file system. Thus, the mount mechanism does not exhibit a transitivity property.

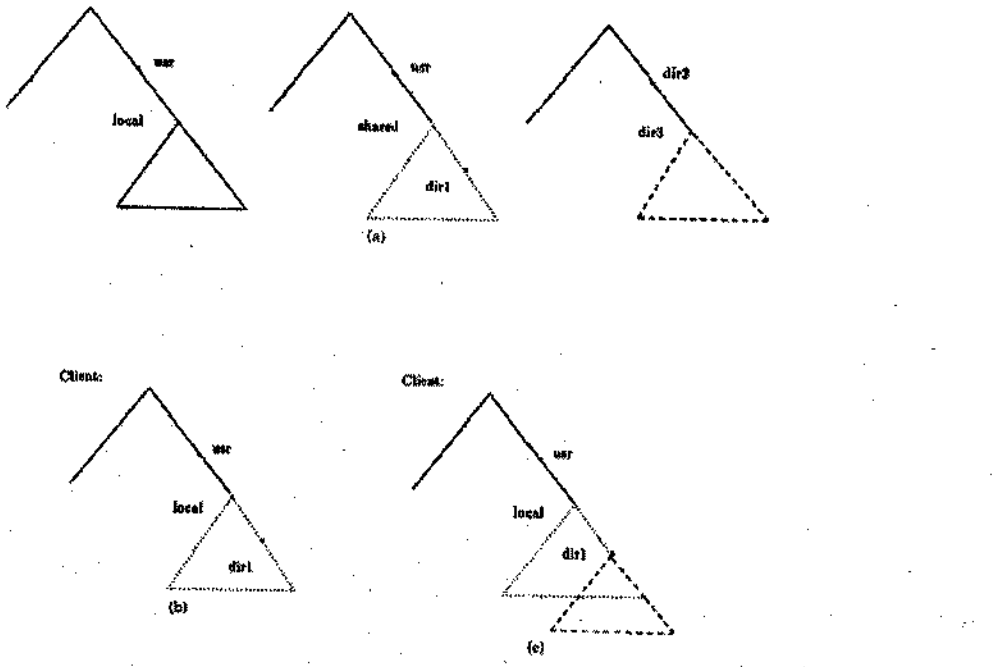


Figure 10.3: (a) NFS joins indiependent file systems;

(b) by mounts; (c) Cascading mounts

In Figure 10.3 (c) we illustrate cascading mounts by continuing our example. The figure shows the result of mounting server2:/dir2/dir over client:/usr/local/dir 1, which is already remotely mounted from serverl. Files within dir3 can be accessed in client using the prefix /usr/local/dir 1. The mount protocol is used to establish the initial connection between a server and a client. The server maintains an export list (the /etc/exports in UNIX) that specifies the local file systems it exports for mounting, along with names of machines permitted to mount them. Any directory within an exported file system can be remotely mounted by an accredited machine. Hence, a component unit is such a directory. When the server receives a mount request that conforms to its export list, it returns to the client a file handle that is the key for further accesses to files within the mounted file system.

The file handle contains all the information the server needs to distinguish individual files it stores. In UNIX terms, the file handle consists of a file system identifier and an i-node number to identify the exact mounted directory within the exported file system. The server also maintains a list of the client machines and the corresponding currently mounted directories. This list is mainly for administrative purposes, such as for notifying all clients that the server is going down. Adding and deleting an entry in this list is the only way the server state is affected by the mount protocol. Usually a system has some static mounting pre-configuration that is established at boot time; however, this layout can be modified (/etc/fstab in UNIX).

10.5.2.2 NFS Protocol

The NFS protocol provides a set of remote procedure calls for remote file operations.

The procedures support the following operations:

- Searching for a file within a directory (i.e., lookup).
- Reading a set of directory entries. Manipulating links and directories.
- Accessing file attributes.

- Reading and writing files.

These procedures can be invoked only after having a file handle for the remotely mounted directory. Recall that the mount operation supplies this file handle the omission of Open and Close operations is intentional. A prominent feature of NFS servers is that they are stateless. There are no parallels to UNIX's open files table or file structures on the server side. A further implication of the stateless server philosophy and a result of the synchrony of an RPC is that modified data (including indirection and status blocks) must be committed to the server's disk before the call returns results to the client.

The NFS protocol does not provide concurrency control mechanisms. The claim is that since locks management is inherently stateful, a service outside the NFS should provide locking. It is advised that users would coordinate access to shared files using mechanisms outside the scope of NFS (e.g., by means provided in a database management system).

10.5.3 Implementation

In general, Sun's implementation of NFS is integrated with the SunOS kernel for reasons of efficiency (although such integration is not strictly necessary). In this section we outline this implementation.

10.5.3.1 Architecture

The NFS architecture is schematically depicted in Figure 10.4. The user interface is the UNIX system calls interface based on the Open, Read, Write, Close calls, and file descriptors. This interface is on top of a middle layer called the Virtual File System (VFS) layer. The bottom layer is the one that implements the NFS protocol and is called the NFS layer. These layers comprise the NFS software architecture.

The figure also shows the RPC/XDR software layer, local file systems, and the network and thus can serve to illustrate the integration of a DFS with all these components. The VFS serves two important functions:

- It separates file system generic operations from their implementation by defining a clean interface. Several implementations for the VFS interface may coexist on the same machine, allowing transparent access to a variety of types of file systems mounted locally.

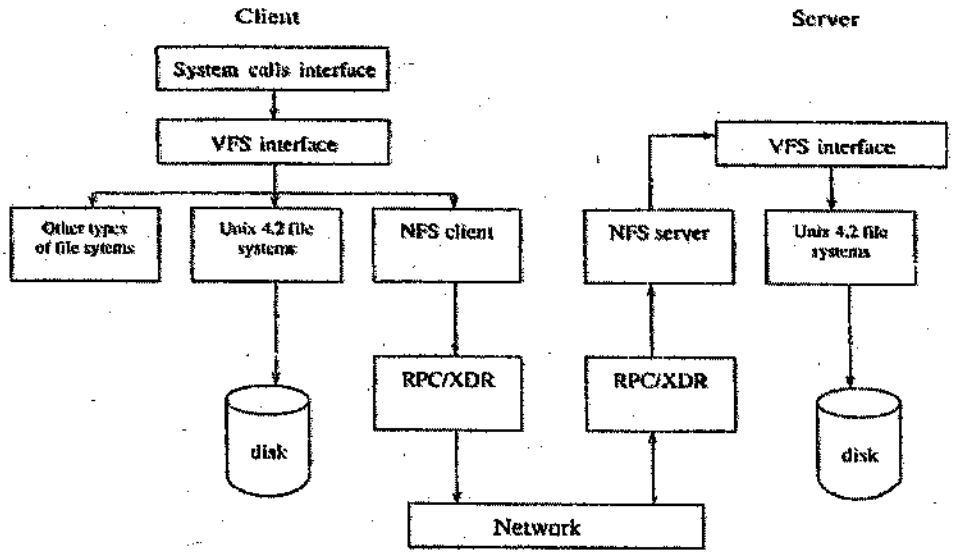


Figure 10.4: Schematic view of the NFS architecture.

- The VFS is based on a file representation structure called a vnode, which contains a numerical designator for a file that is networkwide unique. (Recall that UNIX i-nodes are unique only within a single file system.) The kernel maintains one vnode structure for each active node (file or directory). Essentially, for every file the vnode structures complemented by the mount table provide a pointer to its parent file system, as well as to the file system over which it is mounted.

Thus, the VFS distinguishes local files from remote ones, and local files are further distinguished according to their file system types. The VFS activates file system specific operations to handle local requests according to their file system types and calls the NFS protocol procedures for remote requests. File handles are constructed from the relevant vnodes and passed as arguments to these procedures. As an illustration of the architecture, let us trace how an operation on an already open remote file is handled. The client initiates the operation by a regular system call. The operating system layer maps this call to a VFS operation on the appropriate v-node The VFS layer identifies the file as a remote one and invokes the appropriate NFS procedure. An RPC call is made to the NFS service layer at the remote server. This call is re-injected into the VFS layer, which finds that it is local and invokes the appropriate file system operation. This path is retraced to return the result. An advantage of this architecture is that the client and the server are identical; thus, it is possible for a machine to be a client, or a server, or both. The actual service on each server is performed by several kernel processes, which provide a temporary substitute to a LWP facility.

10.5.3.2 Pathname Translation

Pathname translation is done by breaking the path into component names and doing a separate NFS lookup call for every pair of component name and directory v-node. Thus, lookups are performed remotely by the server. Once a mount point is crossed, every component lookup causes a separate RPC to the server. This expensive path-name traversal scheme is needed, since each client has a unique layout of its logical name space, dictated by the mounts if performed. It would have been much more efficient to pass a pathname to a server and receive a target v-node once a mount point was encountered. But at any point there can be another mount point for the particular client of which the stateless server is unaware. To make lookup faster, a directory name lookup cache at the client holds the v-nodes for remote directory names. This cache speeds up references to files with the same initial pathname. The directory cache is discarded when attributes returned from the server do not match the attributes of the cached v-node. Recall that mounting a remote file system on top of another already mounted remote file system (cascading mount) is allowed in NFS. A server cannot, however, act as an intermediary between a client and another server. Instead, a client must establish a direct server-client connection with the second server by mounting the desired server directory. Therefore, when a client does a lookup on a directory on which the server has mounted a file system, the client sees the underlying directory instead of the mounted directory. When a client has a cascading mount, more than one server can be involved in a pathname traversal. Each component lookup is, however, performed between the original client and some server.

10.5.3.3 Caching and Consistency

With the exception of opening and closing files, there is almost a one-to-one correspondence between the regular UNIX system calls for file operations and the NFS protocol RPCs. Thus, a remote file operation can be translated directly to the corresponding RPC. Conceptually, NFS adheres to the remote service paradigm, but in practice buffering and caching techniques are used for the sake of performance. There is no direct correspondence between a remote operation and an RPC. Instead, file

blocks and file attributes are fetched by the RPCs and cached locally. Future remote operations use the cached data subject to some consistency constraints. There are two caches: file blocks cache and file attribute (i-node information) cache. On a file open, the kernel checks with the remote server about whether to fetch or revalidate the cached attributes by comparing time stamps of the last modification. The cached file blocks are used only if the corresponding cached attributes are up to date. The attribute cache is updated whenever new attributes arrive from the server after a cache miss. Cached attributes are discarded typically after 3 s for files or 30 s for directories. Both read-ahead and delayed-write techniques are used between the server and the client. The caching unit is fairly large (8Kb) for performance reasons. Clients do not free delayed-write blocks until the server confirms the data are written to disk. In contrast to Sprite, delayed-write is retained even when a file is open concurrently in conflicting modes. Hence, UNIX semantics are not preserved. Tuning the system for performance makes it difficult to characterize the sharing semantics of NFS. New files created on a machine may not be visible elsewhere for 30 s. It is indeterminate whether writes to a file at one site are visible to other sites that have the file open for reading. New opens of that file observe only the changes that have already been flushed to the server. Thus, NFS fails to provide either strict emulation of UNIX semantics or any other clear semantics.

Finally, it should be realized that NFS is commercially available, has very reasonable performance, and is perceived as a de facto standard in the user community.

10.6 SUMMARY

Distributed File System (DFS) allows administrators to group shared folders located on different servers and present them to users as a virtual tree of folders known as a namespace. A namespace provides numerous benefits, including increased availability of data, load sharing, and simplified data migration. An operating system maintains a namespace for a single system. A pure name yields no information such as the location of the named object, or the context in which the name is to be resolved, i.e. looked up. A namespace is a collection of valid names recognized by a name service. The name resolution is the process of obtaining a value which allows an object to be used; for example, to determine the network address of a named computer or the network addresses of the computers which receive mail (hold the mailbox) for a named individual. The term availability refers to the accessibility of a file. If a client requests access to a file that the client has the right to access and is unable to access the file, then the file is said to be unavailable. Scalability is a characteristic that refers to the ease with which a distributed file system can be expanded to accommodate increased data access needs or increased storage needs.

10.7 GLOSSARY

Logical Name Structure: A fundamental observation is that every machine establishes its own view of the logical name structure. There is no notion of global name hierarchy. Each machine has its own root serving as a private and absolute point of reference for its own view of the name structure. Selective mounting of parts of file systems upon explicit request allows each machine to obtain its unique view of the global file system. As a result, users enjoy some degree of independence, flexibility, and privacy. It seems that the penalty paid for this flexibility is administrative complexity.

Network service versus distributed operating system : NFS is a network service for sharing files rather than an integral component of a distributed operating system. This characterization does not contradict the SunOS kernel implementation of NFS, since the kernel integration is only for

performance reasons. Being a network service has two main implications. First, remote-file sharing is not the default; the service initiating remote sharing (i.e., mounting) has to be explicitly invoked. Moreover, the first step in accessing a remote file, the mount call, is a location dependent one. Second, perceiving NFS as a service and not as part of the operating system allows its design specification to be implementation independent.

Remote service : Once a file can be accessed transparently I/O operations are performed according to the remote service method: The data in the file are not fetched en masse; instead, the remote site potentially participates in each Read and Write operation. NFS uses caching to improve performance, but the remote site is conceptually involved in every I/O operation.

Fault tolerance : A novel feature of NFS is the stateless approach taken in the design of the servers. The result is resiliency to client, server, or network failures. Should a client fail, it is not necessary for the server to take any action. Once caching was introduced, various patches had to be invented to keep the cached data consistent without making the server stateful.

Sharing semantics : NFS does not provide UNIX semantics for concurrently open files. In fact, the current semantics cannot be characterized clearly, since they are timing dependent.

10.8 FURTHER READINGS

1. George Coulouris, 'Distributed Systems: Concepts and Design' (Fourth Edition), Pearson Education Ltd.
2. Andrew S. Tanenbaum, 'Distributed Operating Systems' (Fourth Edition), Pearson Education Ltd.
3. M.L.Liu, 'Distributed Computing: Principles and Applications' (Third Edition), Pearson Education Ltd
4. Andrew S. Tanenbaum, 'Distributed Systems: Principles and Paradigms' (Third Edition), Pearson Education Ltd.

10.9 ANSWER TO THE SELF LEARNING EXERCISE

Answer to check your progress 10.2

1. (b) 2. (a) 3. (b) 4. (a) 5. (a)

Answer to check your progress 10.3

1. (a) 2. (c) 3. (b) 4. (a) 5. (a)

10.10 UNIT-END QUESTIONS

1. What is Distributed Operating System? Define the Distributed File System (DFS).
2. Explain about the key features of a Distributed File System (DFS).
3. Illustrate and explain the architecture of a Distributed File System.
4. Explain various DFS design issues in brief.
5. Explain the 'Write-through Algorithm' in brief.

□□□□

UNIT 11 : DISTRIBUTED SHARED MEMORY

Structure of the Unit

11.0 OBJECTIVE

Distributed shared memory (DSM) systems have attracted considera recently, since they combine the advantages of two different computer clas

and speed ra

multiprocessors and distributed systems. The most important one is the use of shared memory programming paradigm on physically distributed memories.

A Distributed Shared Memory (DSM) is a memory space that is logically shared by processes running on computers connected by a communication network. While such an organization exists in shared memory multiprocessors, in the domain of distributed systems it is unusual. Most existing distributed systems are structured as a number of processes with independent address spaces. These processes communicate with each other through some form of interprocess communication (IPC), typically message passing or remote procedure call. In a DSM system, data sharing (and thus IPC) is supported directly. Processes communicate with each other by reading and modifying shared directly-addressable data. A DSM can be a flat and paged virtual address space, a segmented single level store, or even a physical address space.

This chapter provides a clear understanding about the 'Distributed Shared Memory' (DSM) and issues related to it, like need, architecture, algorithms for easy implementation of DSM, and some design issues.

11.1 INTRODUCTION

A large progress was recently made in the research and development of systems with multiple processors, capable of delivering high computing power in order to satisfy the constantly increasing demands of typical applications.

A distributed system can be viewed as group of computers cooperating with each other to achieve some goal. These computers are autonomous, in that each computer has an independent flow of control, and there is no sharing of physical memory between them, unlike multiprocessors. Processes running on different computers have distinct address spaces. They communicate by sending and receiving messages. An important characteristic of cooperation is state sharing. Unfortunately, message passing primitives do not support data sharing directly. Data sharing is still possible with these primitives. This can be done by implementing the shared data in a dedicated process and operating on the data by sending predefined operations to this process. Other methods may involve moving data around explicitly using message passing primitives. Special care must be taken to maintain the consistency if a piece of data is replicated.

As more experience is gained with message passing programming, it is found that having to move data back and forth explicitly within programs puts a significant burden on application programmers. Remote procedure call (RPC), was introduced to provide a procedure call like interface. Since the "procedure call" is performed in a separate address space, it is difficult for the caller to pass context related data or complicated data structures, i.e., parameters must be passed by value. Birrell indicated the desire for distributed shared memory so that data could be passed by reference. RPC can be viewed as a "poor man's" version of shared memory, since the semantics are basically those of shared memory, with limitations imposed by implementation constraints (e.g., limited copying of data). A shared memory space provides direct support for data sharing. The mapping of shared data to a shared memory space is natural; Young have observed the relationship between memory and communication. Thus, the question of extension to a distributed setting arose. Ideally, processes on each node should be able to access the same address space with fetch and store operations. However, since the latency involved in communication through the network is high, simple implementation of the fetch and store as remote operations to a shared memory server is not attractive. "Latency" represents a ratio between remote access and local access, and if the value of this ratio is large, the mismatch

must be remedied for adequate performance. Such a mismatch, albeit a generally smaller one, exists in shared memory multiprocessors. Thus, we look to shared memory multiprocessor architectures for inspiration.

A relatively new and promising concept-distributed shared memory (DSM) tries to combine the advantages of two classes of systems: shared memo y systems, having a single global physical memory, equally accessible to all processors, and distributed memory systems, that consist of multiple processing nodes communicating by means of message passing. A DSM system logically implements the shared memory model on a physically distributed memory system. The DSM system, implemented in hardware and/or software, hides the remote communication mechanism from the application writer, so the ease of programming and the portability typical of shared memory systems, as well as the scalability and cost-effectiveness of distributed memory systems are both inherited.

The DSM research area is strongly affected by issues and results generated in a number of closely related disciplines of computer engineering (Figure 11.1).

A distributed system is a collection of heterogeneous computers and processes connected via a network that works closely together to accomplish a common goal. A distributed system is a collection of independent computers that appears to its users as a single coherent system.

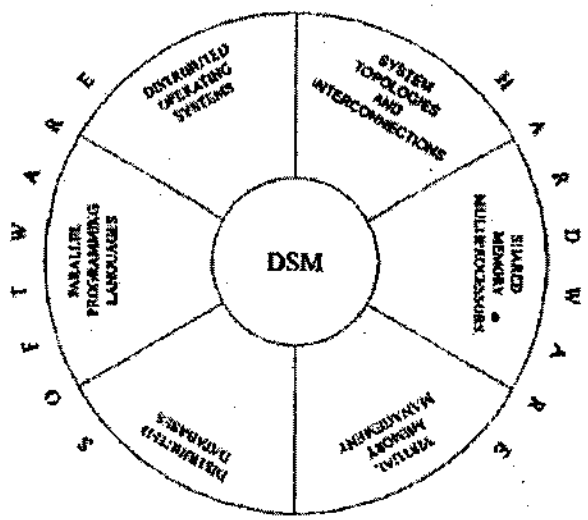


Figure 11.1: DSM and related disciplines

High performance computing will increasingly utilize distributed platforms consisting of standard workstations connected by high-speed networks, called a network of workstations. In order to make such platforms practical for utilization by a wide community of users, an extensive research effort by computer scientists has been undertaken. The approaches taken include:

1. Message passing
2. Software distributed shared memory
3. Shared tuple space

4. Remote procedure calls.

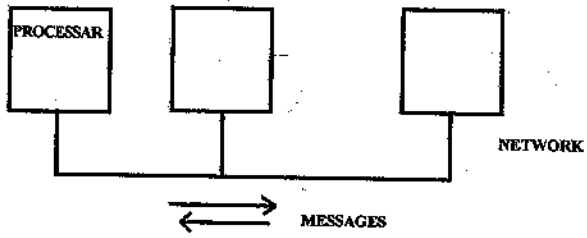


FIGURE 11.2: MESSAGE PASSING NETWORK

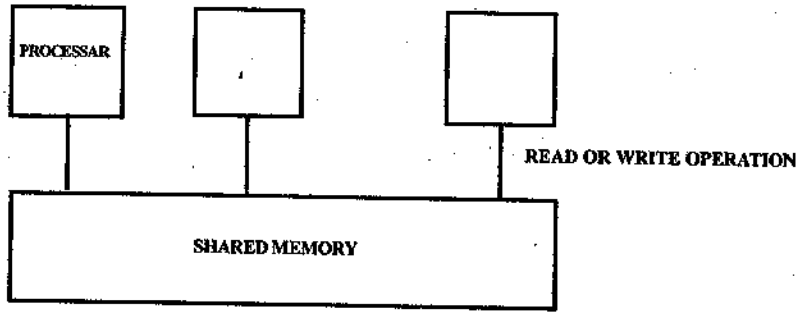


Figure 11.3: Distributed Shared Memory

Two of these approaches are currently most viable for providing the software environment to enable the utilization of networked workstations for parallel computation. One is based on the message passing model, Fig. 11.2, embodied in, e.g. PVM and MPI. Although this approach is the most popular now, it is rather low level and makes programming very difficult.

Advantages of DSM are:

1. In the message passing model, programs make shared data available through explicit message passing. In other words, programmers need to be conscious of the data movement between processes (have to explicitly use communication primitives, such as SEND and RECEIVE). In contrast, DSM systems hide this explicit data movement and provide a simpler abstraction for sharing data that programmers are already well versed with. Hence, it is easier to design and write parallel algorithms using DSM rather than through explicit message passing.

2. In the message passing model, data moves between two different address spaces. This makes it difficult to pass complex data structure between two processes. Moreover, passing data by reference and passing data structures containing pointers is generally difficult and expensive. In contrast, DSM systems allow complex structures to be passed by reference.

3. The physical memory available at all nodes of a DSM system combined together is enormous. This large memory can be used to efficiently run programs that require large memory without incurring disk latency due to swapping in the traditional distributed systems.

4. In tightly coupled multiprocessor systems with a single shared memory, main memory is

accessed via a common bus – a serialization point – that limits the size of the multiprocessor system to a few tens of processors. DSM systems do not suffer from this drawback and can easily be scaled upwards.

5. Programs written for shared memory multiprocessors can in principle be run on DSM systems without any changes.

11.2 ARCHITECTURE OF DISTRIBUTED SHARED MEMORY (DSM)

Distributed shared memory (DSM) system is a resourse management component of a distributed operating system that implements the shared memory model in the distributed systems, which have no physically shared memory.

The shared memory model provides a virtual address space that is shared among all nodes (computers) in a distributed system (see Fig. 11.4).

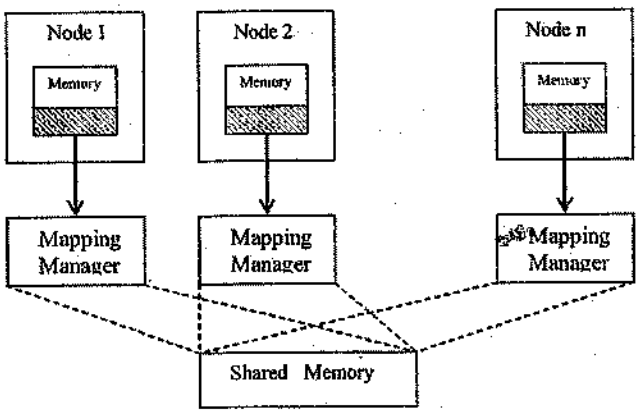


Fig. 11.4 Distributed shared memory

With DSM, programs access data in the shared address space just as they access data in traditional virtual memory. Each node can own data stored in the shared address space, and the ownership can change when data moves from one node to another.

When a process accesses data in the shared address space, a mapping manager maps the shared memory address to the physical memory (which can be local or remote). The mapping manager is a layer of software implemented either in the operating system kernel or as a runtime library routine.

Approach that of providing the programmer with the illusion of global shared memory (Fig. 11.5), by means of physically distributed memory, is currently a very promising approach for the next generation of software environments. When programs execute on distributed shared memory (DSM) system, the low level details of data movement are handled dynamically by the system itself without the intervention of the application programmer. Of course, the underlying system still implements the shared memory by means of message passing.

However, the programmer is not aware of this and does not need to control it. The key issue of efficiency depends heavily on the synchronization required by the computation and the amount of messages required. The interest in the potential of software DSM-based systems is evidenced by the vigorous program of research and by several prototypes that have been developed in recent years. In

fact, researchers have identified bottlenecks in the performance of software DSM systems and have proposed techniques for removing them.

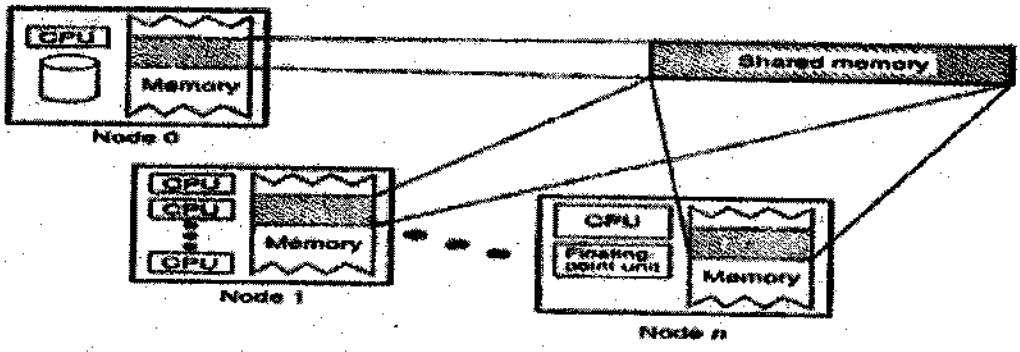


Figure 11.5: Shared Memory Illusion

When the network of workstations is viewed as a shared memory system (as shown in Fig.11.5), developing parallel programs is much easier, from the programmer point of view. In order to provide such abstraction, software based distributed shared memory (DSM) system is required. While sharing the physically distributed memory, local copies are maintained for performance reasons. Memory consistency must be ensured for all the shared copies.

In short we can say that Distributed Shared Memory enables programs to access data in traditional virtual memory. It is primarily a tool for parallel application or a group of applications in which individual shared data items can be accessed directly. In systems that support Distributed Shared Memory, data moves between secondary memory and main memory as well as between main memories of different nodes. Each node can own data stored in the shared address space, and the ownership can change when data moves from one node to another. When a process accesses data in the shared address space, a mapping manager maps the shared memory address to the physical memory. The mapping manager is a layer of software implemented either in the operating system kernel or as a runtime library routine.

The Distributed Shared Memory spares the programmer the concerns of message passing when writing applications that might otherwise have to use it. Shared memory provides the fastest possible communication, hence the greatest opportunity for concurrent execution.

The DSM architecture comprises as the following,

1. DSM Subsystem:

Following points should be remember,

1. Routines to handle page faults relating to virtual addresses corresponding to a DSM region.
2. Code to service system calls which allow a user process to get, attach and detach a DSM region.
3. Code to handle system calls from the DSM server.

2. DSM Server

Following points should be remember,

1. **In-serve :** Receives messages from remote DSM servers and takes appropriate action. (E.g. Invalidate its copy of a page)

2. **Out-serve :** Receives requests from the DSM subsystem and communicates with its peer DSM servers at remote nodes. Note that the DSM subsystem itself does not directly communicate over the network with other hosts.

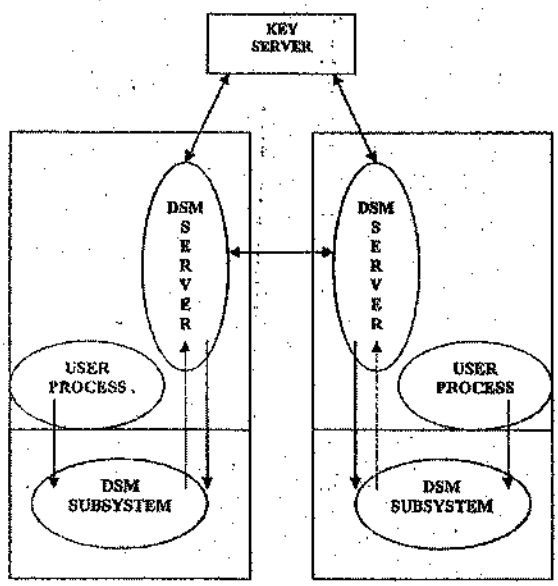
3. Communication with key Server.

3. **KEY Server**

Following points should be remember,

1. Each region must be uniquely identifiable across the entire LAN. When a process executes 'shmget' system call with a key and is the first process at that host to do so, the key server is consulted.

2. Key server's internal table is looked-up for the key, if not found then it stores the specified key in the table as a new entry.



11.3 ALGORITHMS FOR IMPLEMENTING DSM

The advantages of distributed shared memory have made it the focus of recent study and have prompted the development of various algorithms for implementing the shared data model. Several implementations have demonstrated that, in terms of performance, distributed shared memory can compete with direct use of data passing in loosely coupled distributed systems.

The central issues in the implementation of DSM are:

(a) how to keep track of the location of remote data,

(b) how to overcome the communication delays and high overhead associated with the

execution of communication protocols in distributed systems when accessing remote data, and

(c) how to make shared data concurrently accessible at several nodes in order to improve system performance.

The algorithms described can be categorized by whether they migrate and /or replicate data, as depicted in figure below. Two of the algorithms migrate data to the site where it is accessed in an attempt to exploit locality in data accesses and decrease the number of remote accesses, thus avoiding communication overhead. The two other algorithms replicate data so that multiple read accesses can take place at the same time using local accesses.

Basically there are four Distributed Memory Algorithms given as follows,

	Non-Replicated	Replicated
Non-Migrating	Central	Full-replication
Migrating	Migration	Read-replication

11.3.1 Central Server Algorithm

In the Central-server Algorithm, a central-server maintains all the shared data. It services the **read requests** from other nodes or clients by returning the data items to them. It updates the data on **write requests** by clients and returns acknowledgment messages. A timeout can be employed to resend the requests in case of failed acknowledgments. Duplicate write requests can be detected by associating sequence numbers with write requests. A failure condition is returned to the application trying to access shared data after several retransmissions without a response.

Although, the central-server algorithm is simple to implement, the central-server can become a bottleneck. To overcome this problem, shared data can be distributed among several servers. In such a case, clients must be able to locate the appropriate server for every data access. Multicasting data access requests is undesirable as it does not reduce the load at the servers compared to the central-server scheme. A better way to distribute data is to partition the shared data by address and use a mapping function to locate the appropriate server.

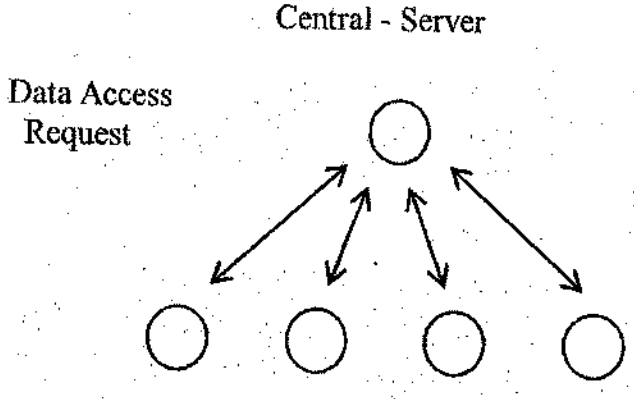


Fig. 11.6 The central-server algorithm

11.3.2 The Migration Algorithm

In the Migration Algorithm, the data is shipped to the location of the data access request allowing subsequent accesses to the data to be performed locally. The migration algorithm allows only one node to access a shared data at a time. This is a single reader/single writer protocol, since only the threads executing on one host can read or write a given data item at any time.

Typically, the whole page or block containing the data item migrates instead of an individual item requested. This algorithm takes advantage of the locality of reference exhibited by programs by amortizing the cost of migration over multiple accesses to the migrated data. However, this approach is susceptible to thrashing, where pages frequently migrate between nodes while servicing only a few requests.

To reduce thrashing, we could use a tunable parameter that determines the duration for which a node can possess a shared data item. This allows a node to make a number of accesses to the page before it is migrated to another node. The migration algorithm provides an opportunity to integrate DSM with the virtual memory provided by the operating system running at individual nodes. When the page size used by DSM is a multiple of the virtual memory page size, a locally held shared memory page can be mapped to an application's virtual address space and accessed using normal machine instructions. On a memory access fault, if the memory address maps to a remote page, a fault-handler will migrate the page before mapping it to the process's address space. Upon migrating, the page is removed from all the address spaces it was mapped to at the previous node. Note that several processes can share a page at a node.

To locate a data block, the migration algorithm can make use of a server that keeps track of the location of pages, or through hints maintained at nodes. These hints direct the search for a page toward the node currently holding the page. Alternatively, a query can be broadcasted to locate a page.

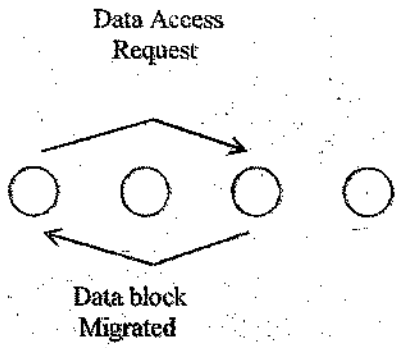


Fig. 11.7 The migration algorithm

11.3.3 The Read-Replication Algorithm

One disadvantage of the migration algorithm is that only the threads on one host can access data contained in the same block at any given time. Replication can reduce the average cost of read operations, since it allows read operations to be simultaneously executed locally (with no communication overhead) at multiple hosts.

However, some of the write operations may become more expensive, since the replicas may

have to be invalidated or updated to maintain consistency. Nevertheless, **if the ratio of reads over writes is large**, the extra expense for the write operations may be more than offset by the lower average cost of the read operations.

Replication can be naturally added to the migration algorithm by allowing either one site a read/write copy of a particular block or multiple sites read-only copies of that block. This type of replication is referred to as **multiple readers/single writer** replication.

For a read operation on a data item in a block that is currently not local, it is necessary to communicate with remote sites to first acquire a read-only copy of that block and to change to read only the access rights to any writable copy if necessary before the read operation can complete. For a write operation to data in a block that is either not local or for which the local host has no write permission, all copies of the same block held at other sites must be invalidated before the write can proceed. The read-replication algorithm is consistent because a read access always returns the value of the most recent write to the same location.

In the read replication algorithm, DSM must keep track of the location of all the copies of data blocks. One way to do this is to have the owner node of a data block keep track of all the nodes that have a copy of the data block. Alternatively, a distributed linked list may be used to keep track of all the nodes that have a copy of the data block.

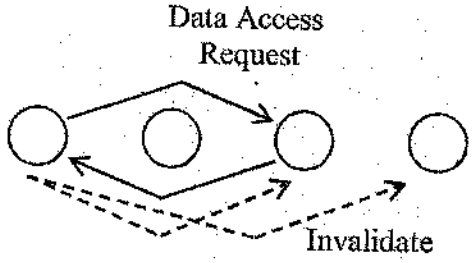


Fig. 11.8 Write operation in the read-replication algorithm.

11.3.4 Full replication Algorithm

The full replication algorithm is an extension of the read replication algorithm. It allows multiple nodes to have both read and write access to shared data blocks (the multiple readers-multiple writers protocol). Because many nodes can write shared data concurrently, the access to shared data must be controlled to maintain its consistency.

One possible way to keep the replicated data consistent is to globally sequence the write operations. A simple strategy based on sequencing uses a single global gap-free sequencer which is a process executing on a host participating in DSM. When a process attempts a write to shared memory, the intended modification is sent to the sequencer. This sequencer assigns the next sequence number to the modification with this sequence number to all sites. Each site processes broadcast write operations in sequence number order. When a modification arrives at a site, the sequence number is verified as the next expected one. If a gap in the sequence numbers is detected, either a modification was missed or a modification was received out of order, in which case a retransmission of the modification message is requested. In effect, this strategy implements a negative acknowledgment

protocol.

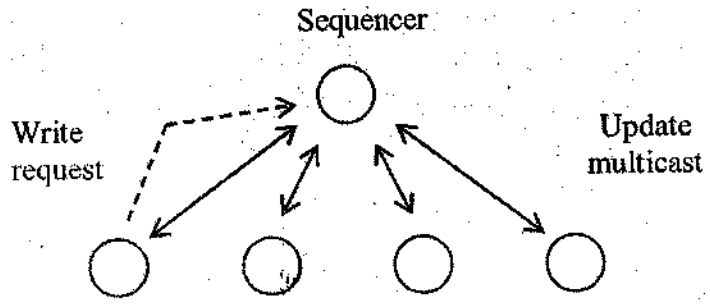


Fig. 11.9 Write operation in the full-replication algorithm

In this scheme, all nodes wishing to modify shared data will send the modifications to a sequencer. The sequencer will assign a sequence number and multicast the modifications with the sequence number to all the nodes that have a copy of the shared data item. Each node processes the modification requests in the sequence number order. A gap between the sequence number of a modification request and the expected sequence number at a node indicates that one or more modifications have been missed. Under such circumstances, the node will ask for the retransmission of the modifications it has missed.

Check your progress 11.1

1. _____ is a memory space that is logically shared by processes running on computers connected by a communication network.
(a) DSM (b) DFS
(c) Both a and b (d) None of above

2. When a process accesses data in the shared address space, a _____ maps the shared memory address to the physical memory (which can be local or remote).
(a) Mapping Manager (b) Windows
(c) Distributed File System (d) None of above

3. In the _____, a central-server maintains all the shared data.
(a) Operating System (b) Central-server Algorithm
(c) Distributed Shared Memory (d) None of above

4. The migration algorithm allows only one node to access a shared data at a time. This is a _____ protocol, since only the threads executing on one host can read or write a given data item at any time.
(a) Single Reader (b) Single Writer
(c) Single Reader/Single Writer (d) None of above

5. One disadvantage of the _____ is that only the threads on one host can access data contained in the same block at any given time.
(a) Distributed File System (b) Distributed Shared Memory
(c) Distributed Process System (d) Migration Algorithm

11.4 MEMORY COHERENCE AND CONSISTENCY MODELS

To improve performance, DSM systems rely on replicating shared data items and allowing concurrent access at many nodes. However, if the concurrent accesses are not carefully controlled, memory accesses may be executed in an order different from that which the programmer expected. Informally, a memory is **coherent** if the value returned by a read operation is always the value that the programmer expected. For example, it is quite natural for a programmer to expect a read operation to return a value stored by the most recent write operation. Thus, to maintain the coherence of shared data items, a mechanism that controls or synchronizes the accesses is necessary.

The word consistency is used to refer to a specific kind of coherence and can be defined as follows:

> **Consistency:** *a read returns the most recently written value*

Strict consistency requires the ability to determine the latest write, which in turn implies a total ordering of requests. The total ordering of requests leads to inefficiency due to more data movement and synchronization requirements than what a program may really call for. To counter this problem, some DSM systems attempt to improve the performance by providing relaxed coherence semantics.

Following are several forms of memory consistency:

11.4.1 Strict Consistency

The most stringent consistency model is called **strict consistency.** It is defined by the following condition:

Any read to a memory location X returns the value stored by the most recent write operation to X.

This definition implicitly assumes the existence of absolute global time so that the determination of "most recent" is unambiguous. Uniprocessors have traditionally observed strict consistency.

In summary, when memory is strictly consistent, all writes are instantaneously visible to all processes and an absolute global time order is maintained. If a memory location is changed, all subsequent reads from that location see the new value, no matter how soon after the change the reads are done and no matter which processes are doing the reading and where they are located. Similarly, if a read is done, it gets the then current value, no matter how quickly the next write is done.

11.4.2 Causal Consistency

The causal consistency model represents a weakening of sequential consistency in that it makes a distinction between events that are potentially causally related and those that are not.

Consider a memory example. Suppose that process **P1** writes a variable **X**. Then **P2** reads **X** and writes **Y**. Here the reading of **X** and the writing of **Y** are potentially causally related because the computation of **Y** may have depended on the value of **X** read by **P2** (i.e., the value written by **P1**).

On the other hand, if two processes spontaneously and simultaneously write two variables, these are not causally related. When there is a read followed later by a write, the two events are

potentially causally related. Similarly, a read is causally related to the write that provided the data the read got. Operations that are not causally related are said to be **concurrent.**

For a memory to be considered causally consistent, it is necessary that the memory obey the following condition:

Writes that are potentially causally related must be seen by all processes in the same order. Concurrent writes may be seen in a different order on different machines.

11.4.3 PRAM Consistency and Processor Consistency

In causal consistency, it is permitted that concurrent writes be seen in a different order on different machines, although causally related ones must be seen in the same order by all machines. The next step in relaxing memory is to drop the latter requirement. Doing so gives PRAM consistency, which is subject to the condition:

Writes done by a single process are received by all other processes in the order in which they were issued, but writes from different processes may be seen in a different order by different processes.

PRAM consistency is due to Lipton and Sandberg. **PRAM** stands for **Pipelined RAM**, because writes by a single process can be pipelined, that is, the process does not have to stall waiting for each one to complete before starting the next one.

11.4.4 Weak Consistency

Although PRAM consistency and processor consistency can give better performance than the stronger models, they are still unnecessarily restrictive for many applications because they require that writes originating in a single process be seen everywhere in order. Not all applications require even seeing all writes, let alone seeing them in order. Consider the case of a process inside a critical section reading and writing some variables in a tight loop. Even though other processes are not supposed to touch the variables until the first process has left its critical section, the memory has no way of knowing when a process is in a critical section and when it is not, so it has to propagate all writes to all memories in the usual way.

Dubois, et al define this model, called **weak consistency**, by saying that it has three properties:

Accesses to synchronization variables are sequentially consistent.

No access to a synchronization variable is allowed to be performed until all previous writes have completed everywhere.

No data access (read or write) is allowed to be performed until all previous accesses to synchronization variables have been performed.

11.4.5 Release Consistency

Weak consistency has the problem that when a synchronization variable is accessed, the memory does not know whether this is being done because the process is finished writing the shared variables or about to start reading them. Consequently, it must take the actions required in both cases, namely making sure that all locally initiated writes have been completed , as well as gathering in all writes from other machines. If the memory could tell the difference between entering a critical region and leaving one, a more efficient implementation might be possible. To provide this information, two

kinds of synchronization variables or operations are needed instead of one.

Release consistency provides these two kinds. **Acquire** accesses are used to tell the memory system that a critical region is about to be entered. **Release** accesses say that a critical region has just been exited. These accesses can be implemented either as ordinary operations on special variables or as special operations.

11.4.6 Entry Consistency

Another consistency model that has been designed to be used with critical sections is **entry consistency**. Like both variants of release consistency, it requires the programmer to use acquire and release at the start and end of each critical section, respectively. However, unlike release consistency, **entry consistency requires each ordinary shared variable to be associated with some synchronization variable such as a lock or barrier**. If it is desired that elements of an array be accessed independently in parallel, then different array elements must be associated with different locks. When an acquire is done on a synchronization variable, only those ordinary shared variables guarded by that synchronization variable are made consistent. Entry consistency differs from lazy release consistency in that the latter does not associate shared variables with locks or barriers and at acquire time has to determine empirically which variables it needs.

Formally, a memory exhibits entry consistency if it meets all the following conditions:

1. An acquire access of a synchronization variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process.
2. Before an exclusive mode access to a synchronization variable by a process is allowed to perform with respect to that process, no other process may hold the synchronization variable, not even in nonexclusive mode.
3. After an exclusive mode access to a synchronization variable has been performed, any other process next nonexclusive mode access to that synchronization variable may not be performed until it has performed with respect to that variable's owner.

11.5 COHERENCE PROTOCOLS

In DSM, we have two types of coherence protocols discussed below-

11.5.1 Write Invalidate

This protocol is commonly implemented in the form of **multiple-reader-single-writer sharing.** At any time, a data item may either be:

- accessed in read-only mode by one or more processes
- read and written by a single process

An item that is currently accessed in read-only mode can be copied indefinitely to other processes. When a process attempts to write to it, a multicast message is sent to all other copies to invalidate them, and this is acknowledged before the write can take place; the other processes are thereby prevented from reading stale data. Any processes attempting to access the data item are blocked if a writer exists. Eventually, control is transferred from the writing process and other accesses may take place once the update has been sent. The effect is to process all accesses to the item on a first-come-first-served basis. This scheme achieves **sequential consistency**.

Under the invalidation scheme, updates are only propagated when data are read, and several updates can take place before communication is necessary. Against this must be placed the cost of invalidating read-only copies before a write can occur. In the multiple-reader-single-write scheme described, this is potentially expensive. But, if the read/write ratio is sufficiently high, then the parallelism obtained by allowing multiple simultaneous readers offsets this cost. Where the read/write ratio is relatively small, a single-reader-single-writer scheme can be more appropriate: i.e, one in which at most one process may be granted read-only access at a time.

11.5.2 Write Update

In the write update protocol, the updates made by a process are made locally and multicast to all other replica managers possessing a copy of the data item, which immediately modify the data read by local processes. Processes read the local copies of data items, without the need for communication. In addition to allowing multiple readers, several processes may write the same data item at the same time; this is known as **multiple-reader-multiple-writer sharing**.

The memory consistency model that is implemented with write-update depends on several factors, mainly the multicast ordering property. **Sequential consistency** can be achieved by using multicasts that are totally ordered which do not return until the update message has been delivered locally. All processes then agree on the order of updates. The set of reads that take place between any two consecutive updates is well defined, and there ordering is immaterial to sequential consistency.

Reads are cheap in the write-update option. However, ordered multicast protocols are relatively expensive to implement in software.

11.6 DESIGN ISSUES

Granularity and page replacement are the two important issues to be considered in the design of a DSM system. They are important because the efficiency of DSM depends on the effectiveness of the size chosen for granularity and the protocol used for page replacement.

11.6.1 Granularity

Granularity refers to the **size of the shared memory unit**. A page size of that is a multiple of the size provided by the underlying hardware or the memory management system allows for the integration of DSM and the memory management systems. By integrating DSM with the underlying memory management system, a DSM system can take advantage of the built in protection mechanism to detect incoherent memory references, and use built in fault handlers to prevent and recover from inappropriate references.

A large page size for the shared memory unit will take advantage of the locality of reference exhibited by processes. By transferring large pages, less overhead is incurred due to page size, but there is greater chance for contention to access a page by many processes. Smaller page sizes are less apt to cause contention as they reduce the likelihood of false sharing. **False sharing** of a page occurs when two different data items, not shared but accessed by two different processes, are allocated to a single page. So the protocols that adapt to a granularity size that is appropriate to the sharing pattern will perform better than those protocols that make use of a static granularity size.

11.6.2 Page Replacement

A memory management system has to address the issue of page replacement because the size of physical memory is limited. In DSM systems that support data movement, traditional methods such

as least recently used (LRU) cannot be used directly. Data may be accessed in different modes such as **shared, private, read-only, writable, etc.**, in DSM systems. To avoid degradation in the system performance, a page replacement policy would have to take the **page access modes** into consideration. For instance, private pages may be replaced before shared pages, as shared pages would have to be moved over the network, possibly to their owner. Read-only pages can simply be deleted as their owners will have a copy. Thus the LRU policy with classes is one possible strategy to handle page replacement. Once a page is selected for replacement, the DSM system must ensure that the page is not lost forever. One option is to swap the page onto disk memory. However, if the page is a replica and is not owned by the node, it can be sent to the owner node.

Check your progress 11.2

1. When memory is _____, all writes are instantaneously visible to all processes and an absolute global time order is maintained.
   (a) Strictly Consistent (b) Causal Consistent
   (c) Both a and b (d) None of above

2. Operations that are not causally related are said to be _____.
   (a) Concurrent (b) Strict
   (c) DSM (d) None of above

3. In the Distributed Shared Memory Systems, PRAM stands for _____.
   (a) Programmable RAM (b) Pipelined RAM
   (c) Both a and b (d) None of above

4. _____ are used to tell the memory system that a critical region is about to be entered.
   (a) Ready Accesses (b) Acquire accesses
   (c) Real Accesses (d) None of above

5. _____ requires each ordinary shared variable to be associated with some synchronization variable such as a lock or barrier.
   (a) Real Consistency (b) Release Consistency
   (c) Entry Consistency (d) Migration Algorithm

11.7 SUMMARY

Distributed Shared Memory (DSM) is a memory area shared by processes running on computers connected by a network. DSM provides direct system support of the shared memory programming model. When assisted by hardware, it can also provide a low-overhead interprocess communication (IPC) mechanism to software. Shared pages are migrated on demand between the hosts. Since computer network latency is typically much larger than that of a shared bus, caching in DSM is necessary for performance. We use caching and issues such as address space structure and page replacement schemes to define taxonomy.

11.8 GLOSSARY

Shared memory: A term applied to both hardware and software indicating the presence of a memory region that is shared between system components. For programming environments, the term means that memory is shared between processes or threads. Applied to hardware, it means that the architectural feature tying processors together is shared memory.

Distributed shared memory: A computer system that constructs an address space shared among multiple UEs from physical memory subsystems that are distinct and distributed about the

system. There may be operating-system and hardware support for the distributed shared memory system, or the shared memory may be implemented entirely in software as a separate middleware layer.

Node: Common term for the computational elements that make up a distributed-memory parallel machine. Each node has its own memory and at least one processor; that is, a node can be a uniprocessor or some type of multiprocessor.

NUMA (Non-Uniform Memory Access): This term is used to describe a shared-memory computer containing a hierarchy of memories, with different access times for each level in the hierarchy. The distinguishing feature is that the time required to access memory locations is not uniform; i.e., access times to different locations can be different.

Virtual shared memory. A system that provides the abstraction of shared memory, allowing programmers to write to a shared memory even when the underlying hardware is based on distributed-memory architecture. Virtual shared memory systems can be implemented within the operating system or as part of the programming environment.

11.9 FURTHER READINGS

1. George Coulouris, 'Distributed Systems: Concepts and Design' (Fourth Edition), Pearson Education Ltd.
2. Andrew S. Tanenbaum, 'Distributed Operating Systems' (Fourth Edition), Pearson Education Ltd.
3. M.L.Liu, 'Distributed Computing: Principles and Applications' (Third Edition), Pearson Education Ltd
4. Andrew S. Tanenbaum, 'Distributed Systems: Principles and Paradigms' (Third Edition), Pearson Education Ltd.

11.10 ANSWER TO THE SELF LEARNING EXERCISES

Answer to check your progress 11.1

1. (a) 2. (a) 3. (b) 4. (c) 5. (d)

Answer to check your progress 11.2

1. (a) 2. (a) 3. (b) 4. (b) 5. (c)

11.11 UNIT-END QUESTIONS

1. Explain the Distributed Shared Memory (DSM) and its significance.
2. Illustrate and explain the architecture of a Distributed Shared Memory.
3. Explain various DSM design issues in brief.
4. Explain different types of Coherence Protocols.
5. Give various algorithms used for implementing the DSM.

□□□□

UNIT 12 : DISTRIBUTED RESOURCE SECURITY

Structure of the Unit

12.0 OBJECTIVE

The topic of security is huge and includes all aspects of controlling access to computers, networks and the information stored, processed and transferred in computer systems. Because file system contains information that is highly valuable to their users, protecting this information against unauthorized access is a major concern now-a-days. In this chapter we will look at a variety of issues concerned with security and protection of various resources of a distributed system.

12.1 INTRODUCTION

Word 'security' and 'protection' are often used interchangeably. The boundary between them is not well defined so to avoid confusion, we will use the term 'security' to refer to the overall problem and the term 'protection methods' to refer to the specific operating system mechanisms used to safeguard information in the computer system. But both terms are closely related to the general problems involved in making sure that files are not read or modified by unauthorized persons, as well as to the specific operating system mechanisms used to provide security.

12.2 POTENTIAL SECURITY VIOLATIONS

Generic Security Attacks

Apart from data loss and intruders some of the common causes of data loss are as follows,

1. Hardware and Software errors, which includes malfunctioning of CPU, unreadable storage devices, program and telecommunication errors etc.
2. Act of God, like fire, flood, earthquakes etc.

3. Human errors, for example mounting of useful data on a wrong device, incorrect data entry, wrong program run, lost of storage device etc.

Regular and adequate backups are the only solution of the problems mentioned above.

On the other hand, intruders are also a problem. There are two types of intruders,

1. Passive Intruders are those who just want to read files they are not authorized to read.
2. Active Intruders are those who want to make unauthorized changes to data.

These are some common causes to make a person acts as an intruder,

1. Many people have terminals to timesharing systems on their desks and these non-technical per sons will read other people's emails and other files if no barriers are placed in the way.
2. System programmers, operators and students are often highly skilled and sometimes devote a substantial amount of time to break the security of the local computer system as a challenging job.
3. Determined attempt to make money like sometimes bank programmer have attempted to break into a banking system to steal from the bank.
4. Commercial or Military espionage is a well-funded and serious attempt to steal programs or tech nology, plans, secrets etc. of others.

The idea of modifying a normal program to do nasty things in addition to its usual function and arranging for the victim to use the modified version in known as the 'Trojan horse attack'. Trojan horse attack is also acts as a major security violation.

The greatest computer security violation of all time began in the 1998. In 1998, Robert Tappan Morris discovered two bugs in Berkeley UNIX and he wrote a self replicating program called a 'Worm' that would exploit errors and replicate itself in seconds on every machine it could gain access to.

Technically, the worm consisted of two programs,

1. The bootstrap and
2. The worm itself

The 'bootstrap' was 99 lines of C called 11.c. It was compiled and executed on the system under attack. Once running, it connected to the machine from which it came, uploaded the main worm and executed it. After going to some trouble to hide its existence, the worm then looked through its new host's routing tables to see what machine that host was connected to, and attempted to spread the bootstrap to those machine.

Computer Viruses are a special category of attack which has become a major problem for computer users. Basically virus is a program fragment that is attached to an original program with the intention of infecting other programs. Virus is differs from a worm only in that a virus piggybacks on an existing program, whereas a worm is a complete program in itself. Viruses and worms both attempt to spread themselves and both can do severe damage.

12.3 EXTERNAL Vs. INTERNAL SECURITY, POLICIES AND MECHANISMS

In outline, security comprises:

* **External controls -security classifications**

A general framework for classifying computer systems is given in the above references. A frame-

work may relate only to internal mechanisms or may cover the classification of information into 'top secret', 'secret', etc. The people using the computer system are then tagged according to which categories of information they may read, write and transfer. The system must be able to enforce such specified policies.

* **Encryption**

The idea of encryption is to transform information so that it cannot be understood by unauthorized parties but can be recovered by decryption. Encryption is often used when data is to be transferred across networks. Data may also be stored in encrypted form. Passwords are usually stored in encrypted form rather than as clear text.

* **Authentication**

The use of a login procedure and a password is part of authentication. The idea is to establish the identity of a principal involved in any computational procedure. A principal can be thought of as a process running a program on behalf of a logged-on user. Identity is typically based on knowledge of a secret, such as a password, or possession of an object, such as a swipe card. The latter may have an associated secret such as a PIN (personal identity number).

Authentication in distributed systems is more complex than in centralized systems because secret information, such as a password, needs to be transferred across a network. A principal may own encryption keys for use in transforming data to and from an encrypted representation. A principal may use a private key, known only to itself, to encrypt the data. The recipient of such data can be sure that the data originated from the expected principal because that principal's publicly available key transforms the data back to an intelligible form. That is, the private and public keys form a pair for use by the algorithms which encrypt and decrypt the data. The login procedure associates the human user with the registered user who owns items such as encryption keys.

* **Authorization, protection or access control**

An authorization policy specifies which principals may access an object and in what way. The system must have mechanisms which can enforce the policies.

The concept of principal is used because greater generality than 'user' is often required. For example, we may wish to express 'anyone running program X may access file Y' or 'if a principal with rights to a file which include read issues a command to printer software to print that file then the printer software is "delegated" the right to read that file (but no other files of the principal) until the print command has been carried out'.

* **Validation of imported software**

The access control policies and mechanisms outlined above may be subverted by the injudicious use of unchecked software. If you acquire and run a program, that program runs with all your access rights. It could read, overwrite or delete your files. For this reason it is desirable that the source of software can be authenticated.

12.3.1 Protection domain

A computer system contains many 'objects' that need to be protected from unauthorized access. These objects can be drives, terminals memory segments or can be processes, databases, files etc. Each object referenced by a unique name and a set of operations that can be carried out on it. There must be a way to ensure the authorized access of objects.

To provide a way to discuss different protection mechanisms, it is convenient to introduce the concept of a domain. A domain is a set of pairs of 'Objects' and 'Rights'. Each pair specifies an object and some subset of the operations that can be performed on it. A 'right' means permission to perform one of the operations.

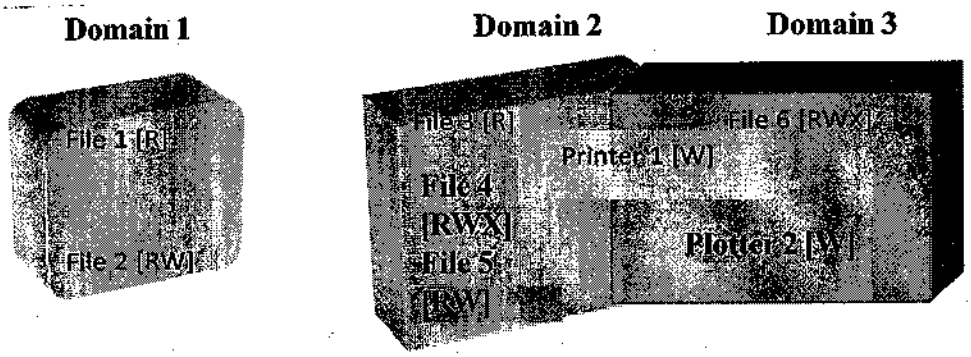


Figure 12.1: Three protection domains

Above figure depicts 03 domains that are showing the object in each domain and the rights [Read, Write, eXecute] available on each object. In the above figure the Printer1 is in two domains at the same time. So, it is possible for the same object to be in multiple domains, with different rights in each domain.

At every instant of time, each process runs in some protection domain. So, there is some collection of objects it can access, and for each object it has some set of rights. Processes can also switch from domain to domain during execution, and this rule is system dependent.

12.3.2 Design Principles for secure systems

Saltzer and Schroeder have identified several general principles that can be used as a guide to designing secure systems. Some principles are as follows,

1. The system design should be public. Assuming that the intruder will not know how the system works serves only to make fool the designers.
2. The default should be no access because errors in which legitimate access is refused will be reported much faster than errors in which unauthorized access is allowed.
3. Check for current authority because a user who opens a file, and keeps it open for weeks will continue to have access even if the owner has long since changed the file protection.
4. Give each process the least privilege possible. If an editor has only the authority to access the file to be edited, editors with Trojan horses will not be able to do much damage.
5. The protection mechanism should be simple, uniform, and built in to the lowest layers of the system.
6. The chosen scheme must be psychologically acceptable because if users feel that protecting their files is too much work they just will not do it.

12.4 ACCESS MATRIX MODEL

In computer science, an Access Control Matrix or Access Matrix is an abstract, formal security model of protection state in computer systems, which characterize the rights of each subject with respect to every object in the system. It was first introduced by Butler W. Lampson in 1971.

A computer security model is a scheme for specifying and enforcing security policies. A security model may be founded upon a formal model of access rights, a model of computation, a model of distributed computing, no particular theoretical grounding at all.

As described in the 'Protection Domain' segment, a protection matrix is large and with the rows being the domains and the columns being the objects. Each box lists the rights that the domain contains for the object. Given the matrix and the current domain number, the system can tell if an access to a given object in a particular way from a specified domain is allowed.

	Object							
Domain	File1	File2	File3	File4	File5	File6	Printer1	Plotter2
1	Read	Read Write						
2			Read	Read Write Execute	Read Write		Write	
3						Read Write Execute	Write	Write

Figure 12.2: A protection matrix

12.4.1 Access Control List Method and Capabilities

In an ACL-based security model, when a subject requests to perform an operation on an object, the system first checks the list for an applicable entry in order to decide whether to proceed with the operation. A key issue in the definition of any ACL-based security model is the question of how access control lists are edited. For each object; who can modify the object's ACL and what changes are allowed.

ACL models are assigned to individual objects, or to a collection of objects, and correspond to what may or may not be permitted to "access" the object to which they have been assigned.

ACL-based security models are given as follows,

1. File system ACLs

The list is a data structure, usually a table, containing entries that specify individual user or group rights to specific system objects, such as a program, a process, or a file. These entries are known as access control entries (ACEs) in the Microsoft Windows NT, OpenVMS, Unix-like and Mac OS X operating systems. Each accessible object contains an identifier to its ACL. The privileges or permissions determine specific access rights, such as whether a user can read from, write to, or execute an object. In some implementations an ACE can control whether or not a user, or group of users, may alter the ACL on an object.

Most of the Unix-like operating systems - Linux, FreeBSD or Solaris, support so called POSIX.1e ACLs, based on early POSIX draft that was abandoned.

2. Networking ACLs

In certain proprietary computer hardware an Access Control List refers to rules that are applied to port numbers or network daemon names that are available on a host or other layer 3 device, each with a list of hosts and/or networks permitted to use the service. Both individual servers as well as routers can have network ACLs. Access control lists can generally be configured to control both inbound and out-bound traffic, and in this context they are similar to firewalls.

An object model for access control is given as under,

A general protection model is achieved by forming a matrix with rows representing principals, columns representing objects, and entries representing the access rights of the principal to the object. In addition we assume that objects are typed, that the rights are associated with the object type and can be

enforced by a type manager module, see Figure 12.3.

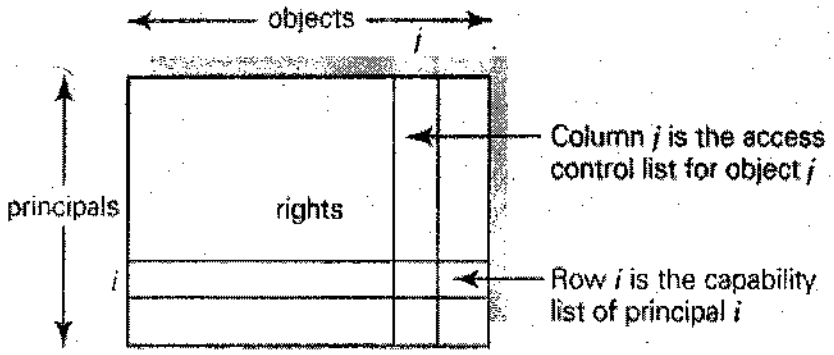


Figure 12.3: An access matrix.

In practice, such an access matrix would be sparse and would be held in a partitioned representation. Most domains have no access at all to most objects, so storing a big, empty matrix is a waste of disk space. For example, with each object could be associated a list of the principals with access rights to the object together with their specific rights. This is a list of the non-null entries in the column associated with the object. For objects of type file such a list is known as an access control list (ACL).

Alternatively, with each principal could be associated a list of the non-null entries of the corresponding row, that is the objects the principal can access together with access rights. Each component of such a list is known as a capability. A typical capability list is shown as below,

	Type	Rights	Object
0	File	R - -	Pointer to File3
1	File	RWX	Pointer to File4
2	File	RW -	Pointer to File5
3	Printer	-W-	Pointer to Printer1

Figure 12.4: The example of Capability list

As figure 12.4 depicted, each capability has a 'Type' field, which tells what kind of an object it is, a 'Rights' field, which is a bit map indicating which of the legal operations on this type of object are permitted, and an 'Object' field, which is a pointer to the object itself and i.e. its i-node number.

Capability lists are themselves objects, and may be pointed to from other capability lists, thus facilitating sharing of sub-domains. Capabilities are often referred to by their position in the capability list.

Following are three ways by which capability lists protect users,

1. The first way requires a tagged architecture, a hardware design in which each memory word has an extra bit that tells whether the word contains a capability or not.
2. The second way is to keep the capability list inside the operating system, and just have processes refer to capabilities by their slot number.
3. The third way is to keep the capability list in user space, but encrypt each capability with a secret key unknown to the user.

Both ACLs and capabilities have been used in system designs and will be defined in detail when we study specific object management functions.

12.4.2 Lock Key Method

The Lock-key Mechanism is a compromise between access lists and capability lists. Each object has a list of unique bit patterns and that is called as 'locks'. On the other hand, each domain has a list of unique bit patterns that is called 'keys'. A process executing in a domain can access an object only if that domain has a key that matches one of the locks of the object.

For this purpose, the list of keys for a domain must be managed by the operating system on behalf of the domain. Users are not directly allowed to examine or modify the list of keys or locks.

12.4.3 Safety in Access Matrix Model

Protection matrices are not static, they change frequently as new object is created or old objects are destroyed, and owners decide to increase or restrict the set of users for their objects. Considerable amounts of attention have to pay to modeling protection systems in which the protection matrix is constantly changing. Some primitives can be combined into protection commands by which the user programs can execute to change the matrix. They may not execute the primitives directly.

For example, the system might have a command to create a new file, which would test to see if the file already existed, and if not, create a new object and give the owner all rights to it. There might also be a command to allow the owner to grant permission to read the file to everyone in the system by inserting the 'read' right in the new file's entry in every domain. At any moment, the matrix determines what a process in any domain can do, not what it is authorized to do. The matrix is what is enforced by the system, authorization has to do with management policy.

For this purpose, the set of all possible matrices can be partitioned into two disjoint sets:

1. The set of all authorized states and;
2. The set of all unauthorized states.

The protection commands mechanism is adequate to enforce some protection policy. Each object is unclassified, confidential and secret. So, the security policy mainly has two rules-

1. No process may read any object whose level is higher than its own.
2. No process may write information into any object whose level is lower than its own.

Check your progress 12.1

1. Protection matrices are _____.
   (a) Static (b) Not Static
   (c) Both a and b (d) None of above
2. The _____ is a compromise between access lists and capability lists.
   (a) Lock-key Mechanism (b) ACL
   (c) Both a and b (d) None of above
3. A general protection model is achieved by forming a matrix with _____ representing principals.
   (a) Rows (b) columns
   (c) Both a and b (d) None of above
4. ACL stands for _____.
   (a) Across control list (b) Access control list
   (c) Real Accesses (d) None of above
5. A computer system contains many _____ that need to be protected from unauthorized access.
   (a) Objects (b) Rights
   (c) Rows (d) Columns

12.5 SUMMARY

Regular and adequate backups are the only solution of the problems related to the loss of data and equipments. Authentication in distributed systems is more complex than in centralized systems because secret information, such as a password, needs to be transferred across a network. The use of a login procedure and a password is part of authentication. The idea is to establish the identity of a principal involved in any computational procedure. Access Control Lists are very helpful to prevent the unauthorized access to data.

12.6 GLOSSARY

Confidentiality :

the act of keeping something private and secret from all but those who are authorized to see it.

Data integrity :

a method of ensuring information has not been altered by unauthorized or unknown means.

Integrity :

assurance that data is not modified (by unauthorized persons) during storage or transmittal.

Identity certificate :

a signed statement that binds a key to the name of an individual and has the intended meaning of delegating authority from that named individual to the public key.

12.7 FURTHER READINGS

1. George Coulouris, 'Distributed Systems: Concepts and Design' (Fourth Edition), Pearson Education Ltd.
2. Andrew S. Tanenbaum, 'Distributed Operating Systems' (Fourth Edition), Pearson Education Ltd.
3. M.L.Liu, 'Distributed Computing: Principles and Applications' (Third Edition), Pearson Education Ltd
4. Andrew S. Tanenbaum, 'Distributed Systems: Principles and Paradigms' (Third Edition), Pearson Education Ltd.

12.8 ANSWER TO THE SELF LEARNING EXCERCISES

Answer to check your progress 12.1

1. (a) 2. (a) 3. (a) 4. (b) 5. (a)

12.9 UNIT END QUESTIONS

1. What is the difference between a virus and a worm? How do they each reproduce?
2. Can the Trojan Horse Attack work in a system protected by capabilities?
3. Explain the Access Control List.
4. Describe the Protection Matrix Models.
5. What is the need of security and protection of data?

□□□□

UNIT 13 : Distributed Data Security

Structure of the Unit

13.0 OBJECTIVE

During this time when the Internet provides essential communication between tens of millions of people and is being increasingly used as a tool for commerce, security becomes a tremendously important issue to deal with.

There are many aspects to security and many applications, ranging from secure commerce and payments to private communications and protecting passwords. One essential aspect for secure communications is that of cryptography, which is the focus of this chapter. But it is important to note that while cryptography is necessary for secure communications, it is not by itself sufficient. The reader is advised, then, that the topics covered in this chapter only describe the first of many steps necessary for better security in any number of situations.

This chapter has two major purposes. The first is to define some of the terms and concepts behind basic cryptographic methods, and to offer a way to compare the countless cryptographic schemes in use today. The second is to provide some real examples of cryptography in use today.

13.1 INTRODUCTION

Data security in modern distributed computing systems is a difficult problem. Network connections and remote file system services, while convenient, often make it possible for an intruder to gain access to sensitive data by compromising only a single component of a large system. Because of the difficulty of reliably protecting information, sensitive files are often not stored on networked computers, making access to them by authorized users inconvenient and putting them out of the reach of useful system services such as backup.

Cryptographic techniques offer a promising approach for protecting files against unauthorized access. When properly implemented and appropriately applied, modern cipher algorithms (such as the Data Encryption Standard (DES) and the more recent IDEA cipher) are widely believed sufficiently strong to render encrypted data unavailable to virtually any adversary who cannot supply the correct key. However, routine use of these algorithms to protect file data is uncommon in current systems. This is partly because file encryption tools, to the extent they are available at all, are often poorly integrated, difficult to use, and vulnerable to non-cryptanalytic system level attacks. File encryption is better handled by the file system itself.

In data and telecommunications, cryptography is necessary when communicating over any untrusted medium, which includes just about any network, particularly the Internet. Within the context of any application-to-application communication, there are some specific security requirements, including:

* Authentication: The process of proving one's identity.
* Privacy/confidentiality: Ensuring that no one can read the message except the intended receiver.
* Integrity: Assuring the receiver that the received message has not been altered in any way from the original.
* Non-repudiation: A mechanism to prove that the sender really sent this message.

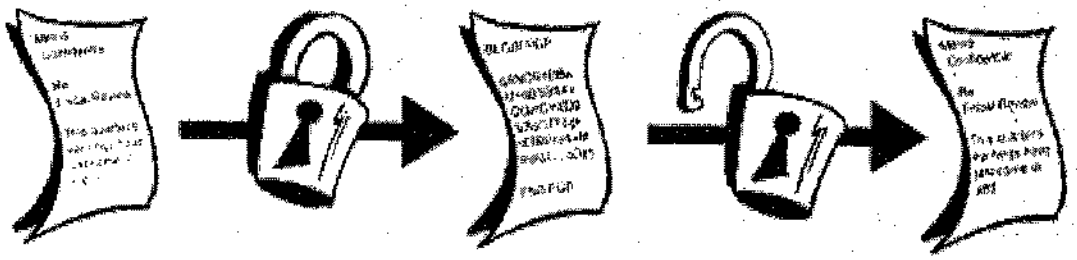
13.2 CRYPTOGRAPHY

When Julius Caesar sent messages to his generals, he didn't trust his messengers. So he replaced every A in his messages with a D, every B with an E, and so on through the alphabet. Only someone who knew the "shift by 3" rule could decipher his messages.

And so we begin.

13.2.1 Encryption and Decryption

Data that can be read and understood without any special measures is called plaintext or cleartext. The method of disguising plaintext in such a way as to hide its substance is called encryption. Encrypting plaintext results in unreadable gibberish called ciphertext. You use encryption to ensure that information is hidden from anyone for whom it is not intended, even those who can see the encrypted data. The process of reverting ciphertext to its original plaintext is called decryption.



plaintext **Encryption** **ciphertext** **Decryption** **plaintext**

Figure 13.1: Encryption and decryption

Cryptography is the science of using mathematics to encrypt and decrypt data. Cryptography enables you to store sensitive information or transmit it across insecure networks (like the Internet) so that it cannot be read by anyone except the intended recipient.

So, Cryptography not only protects data from theft or alteration, but can also be used for user authentication. There are, in general, three types of cryptographic schemes typically used to accomplish these goals: secret key (or symmetric) cryptography, public-key (or asymmetric) cryptography, and hash functions, each of which is described below. In all cases, the initial unencrypted data is referred to as plaintext. It is encrypted into ciphertext, which will in turn (usually) be decrypted into usable plaintext.

While cryptography is the science of securing data, cryptanalysis is the science of analyzing and breaking secure communication. Classical cryptanalysis involves an interesting combination of analytical reasoning, application of mathematical tools, pattern finding, patience, determination, and luck. Cryptanalysts are also called attackers. Cryptology embraces both cryptography and cryptanalysis.

13.2.2 Strong Cryptography

Cryptography can be strong or weak. Cryptographic strength is measured in the time and resources it would require to recover the plaintext. The result of strong cryptography is ciphertext that is very difficult to decipher without possession of the appropriate decoding tool. How difficult? Given all of today's computing power and available time-even a billion computers doing a billion checks a second-it is not possible to decipher the result of strong cryptography before the end of the universe.

13.2.3 How does cryptography work?

A cryptographic algorithm, or cipher, is a mathematical function used in the encryption and decryption process. A cryptographic algorithm works in combination with a key-a word, number, or phrase-to encrypt the plaintext.

The same plaintext encrypts to different ciphertext with different keys. The security of encrypted data is entirely dependent on two things: the strength of the cryptographic algorithm and the secrecy of the key.

A cryptographic algorithm, plus all possible keys and all the protocols that make it work comprise a cryptosystem.

13.3 CONVENTIONAL CRYPTOGRAPHY

In conventional cryptography, also called secret-key or symmetric-key encryption, one key is used both for encryption and decryption. The Data Encryption Standard (DES) is an example of a conventional cryptosystem that is widely employed by the Federal Government. Figure 13.2 is an illustration of the conventional encryption process.

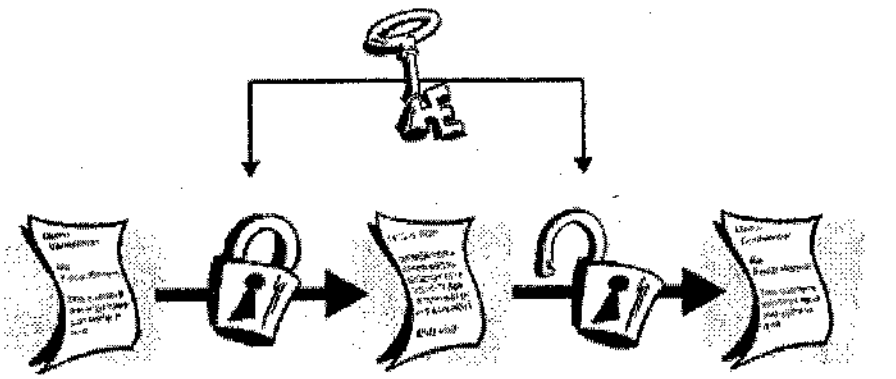


Figure 13.2 Conventional encryption

13.3.1 Caesar's Cipher

An extremely simple example of conventional cryptography is a substitution cipher. A substitution cipher substitutes one piece of information for another. This is most frequently done by offsetting letters of the alphabet. Two examples are Captain Midnight's Secret Decoder Ring, which you may have owned when you were a kid, and Julius Caesar's cipher. In both cases, the algorithm is to offset the alphabet and the key is the number of characters to offset it. For example, if we encode the word "SECRET" using Caesar's key value of 3, we offset the alphabet so that the 3rd letter down (D) begins the alphabet.

So starting with

ABCDEFGHIJKLMNOPQRSTUVWXYZ

and sliding everything up by 3, you get

DEFGHIJKLMNOPQRSTUVWXYZABC

where D=A, E=B, F=C, and so on.

Using this scheme, the plaintext, "SECRET" encrypts as "VHFUHW." To allow someone else to read the ciphertext, you tell them that the key is 3. Obviously, this is exceedingly weak cryptography by today's standards, but hey, it worked for Caesar, and it also illustrates how conventional cryptography works.

13.3.2 Key Management and Conventional encryption

Conventional encryption has benefits. It is very fast. It is especially useful for encrypting data that is not going anywhere. However, conventional encryption alone as a means for transmitting secure data can be quite expensive simply due to the difficulty of secure key distribution.

For a sender and recipient to communicate securely using conventional encryption, they must agree upon a key and keep it secret between themselves. If they are in different physical locations, they must trust a courier. Anyone who overhears or intercepts the key in transit can later read, modify, and forge all information encrypted or authenticated with that key.

13.4 MODELS OF CRYPTOGRAPHY

Most security mechanisms in computer systems are aimed at authenticating the users and clients of services and resources. Servers typically mistrust those who request services from them, and the protocols for obtaining access typically reflect the security needs of the server. In the case of a file system, the converse relationship is true as well; the user must be sure that the file system will not reveal private data without authorization. File encryption can be viewed as a mechanism for enforcing mistrust of servers by their clients.

In cryptography the standard model is the model of computation in which the adversary is only limited by the amount of time and computational power available. Other names used are bare model and plain model.

Cryptographic schemes are usually based on complexity assumptions, which state that some problem, e.g. factorization, cannot be solved in polynomial time. Schemes which can be proven secure using only complexity assumptions are said to be secure in the standard model. Security proofs are notoriously difficult to achieve in the standard model, so in many proofs, cryptographic primitives are replaced by idealized versions. The most usual example of this technique, known as the random oracle model, involves replacing a cryptographic hash function with a genuinely random function. Another example is the generic group model, where the adversary is given access to a randomly chosen encoding of a group, instead of the finite field or elliptic curve groups used in practice.

Other models used invoke trusted third parties to perform some task without cheating, for example, the public key infrastructure (PKI) model requires a certificate authority, which if it were dishonest, could produce fake certificates and use them to forge signatures, or mount a man in the middle attack to read encrypted messages. Other examples of this type are the common random string model and the common reference string model, where it is assumed that all parties have access to some string chosen uniformly at random or a string chosen according to some other probability distribution respectively. These models are often used for Non-interactive zero-knowledge proofs (NIZK). Collectively, these models are referred to as models with special setup assumptions.

Check your progress 13.1

1. Data that can be read and understood without any special measures is called ____.

   (a) Plaintext (b) Ciphertext

   (c) Both a and b (d) None of above (a)

2. The method of disguising plaintext in such a way as to hide its substance is called _____.

   (a) Encryption (b) Decryption

   (c) Key (d) None of above

3. Encrypting plaintext results in unreadable gibberish called _____.

   (a) Plaintext (b) Ciphertext

   (c) Both a and b (d) None of above

4. The process of reverting ciphertext to its original plaintext is called _____.

   (a) Encryption (b) Decryption

   (c) DES (d) None of above

5. _____ is the science of using mathematics to encrypt and decrypt data.

   (a) Real Consistency (b) Ciphertext

   (c) Cryptography (d) RAS

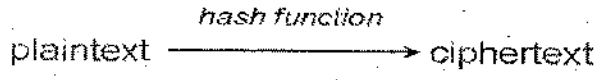
13.5 CRYPTOGRAPHIC SYSTEM AND ITS CLASSIFICATION

There are several ways of classifying cryptographic algorithms. For purposes of this paper, they will be categorized based on the number of keys that are employed for encryption and decryption, and further defined by their application and use. The three types of algorithms that will be discussed in given figure 3 are as follows,

* Secret Key Cryptography (SKC): Uses a single key for both encryption and decryption
* Public Key Cryptography (PKC): Uses one key for encryption and another for decryption
* Hash Functions: Uses a mathematical transformation to irreversibly "encrypt" information

A) Secret key (symmetric) cryptography. SKC uses a single key for both encryption and decryption.

B) Public key (asymmetric) cryptography. PKC uses two keys, one for encryption and the other for decryption.



C) Hash function (one-way cryptography). Hash functions have no key since the plaintext is not recoverable from the ciphertext.

Figure 13.3: Three types of cryptography: secret-key, public key, and hash function.

13.5.1 Private or Secret Key Cryptography - DES and Cipher Block Chaining

The problems of key distribution are solved by public key cryptography, the concept of which was introduced by Whitfield Diffie and Martin Hellman in 1975. Public key cryptography is an asymmetric scheme that uses a pair of keys for encryption: a public key, which encrypts data, and a corresponding private, or secret key for decryption. You publish your public key to the world while keeping your private key secret. Anyone with a copy of your public key can then encrypt information that only you can read. Even people you have never met.

It is computationally infeasible to deduce the private key from the public key. Anyone who has a public key can encrypt information but cannot decrypt it. Only the person who has the corresponding private key can decrypt the information.

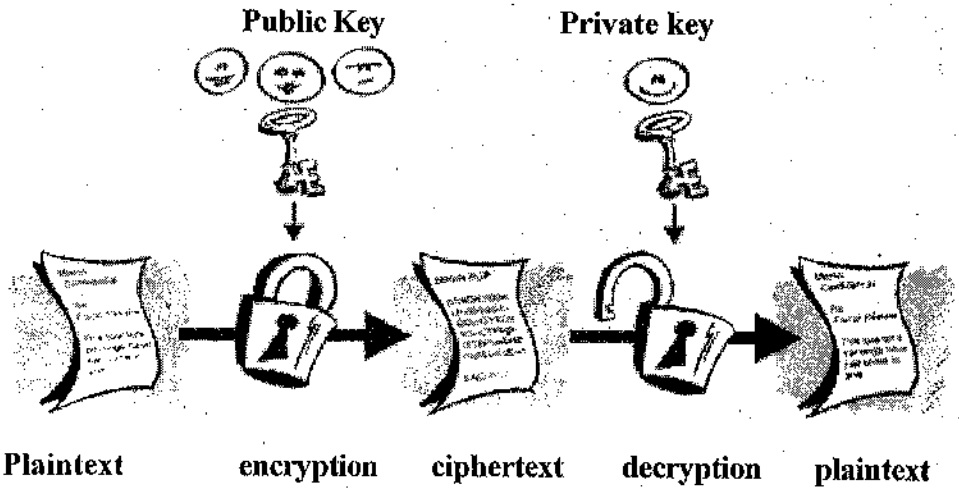


Figure 13.4. Public Key Encryption

The primary benefit of public key cryptography is that it allows people who have no preexisting security arrangement to exchange messages securely. The need for sender and receiver to share secret keys via some secure channel is eliminated; all communications involve only public keys, and no private key is ever transmitted or shared. Some examples of public-key cryptosystems are Elgamal (named for its inventor, Taher Elgamal), RSA (named for its inventors, Ron Rivest, Adi Shamir, and Leonard Adleman), Diffie-Hellman, and DSA, the Digital Signature Algorithm (invented by David Kravitz).Because conventional cryptography was once the only available means for relaying secret information, the expense of secure channels and key distribution relegated its use only to those who could afford it, such as governments and large banks (or small children with secret decoder rings). Public key encryption is the technological revolution that provides strong cryptography to the adult masses.

13.5.1.1 Key

A key is a value that works with a cryptographic algorithm to produce a specific ciphertext. Keys are basically really, really, really big numbers. Key size is measured in bits; the number representing a 1024-bit key is darn huge.

In public key cryptography, the bigger the key, the more secure the ciphertext. However, public key size and conventional cryptography's secret key size are totally unrelated. A conventional 80-bit key has the equivalent strength of a 1024-bit public key. A conventional 128-bit key is equivalent to a 3000-bit public key. Again, the bigger the key, the more secure, but the algorithms used for each type of cryptography are very different and thus comparison is like that of apples to oranges.

While the public and private keys are related, it's very difficult to derive the private key given only the public key; however, deriving the private key is always possible given enough time and computing power. This makes it very important to pick keys of the right size; large enough to be secure, but small enough to be applied fairly quickly.

Larger keys will be cryptographically secure for a longer period of time. If what you want to encrypt needs to be hidden for many years, you might want to use a very large key. Keys are stored in encrypted form.

13.5.1.2 Digital Signature

A major benefit of public key cryptography is that it provides a method for employing digital signatures. Digital signatures enable the recipient of information to verify the authenticity of the information's origin, and also verify that the information is intact. Thus, public key digital signatures provide authentication and data integrity. A digital signature also provides non-repudiation, which means that it prevents the sender from claiming that he or she did not actually send the information. These features are every bit as fundamental to cryptography as privacy, if not more.

A digital signature serves the same purpose as a handwritten signature. However, a handwritten signature is easy to counterfeit. A digital signature is superior to a handwritten signature in that it is nearly impossible to counterfeit, plus it attests to the contents of the information as well as to the identity of the signer.

The basic manner in which digital signatures are created is illustrated in Figure 13.5. Instead of encrypting information using someone else's public key, you encrypt it with your private key. If the information can be decrypted with your public key, then it must have originated with you.

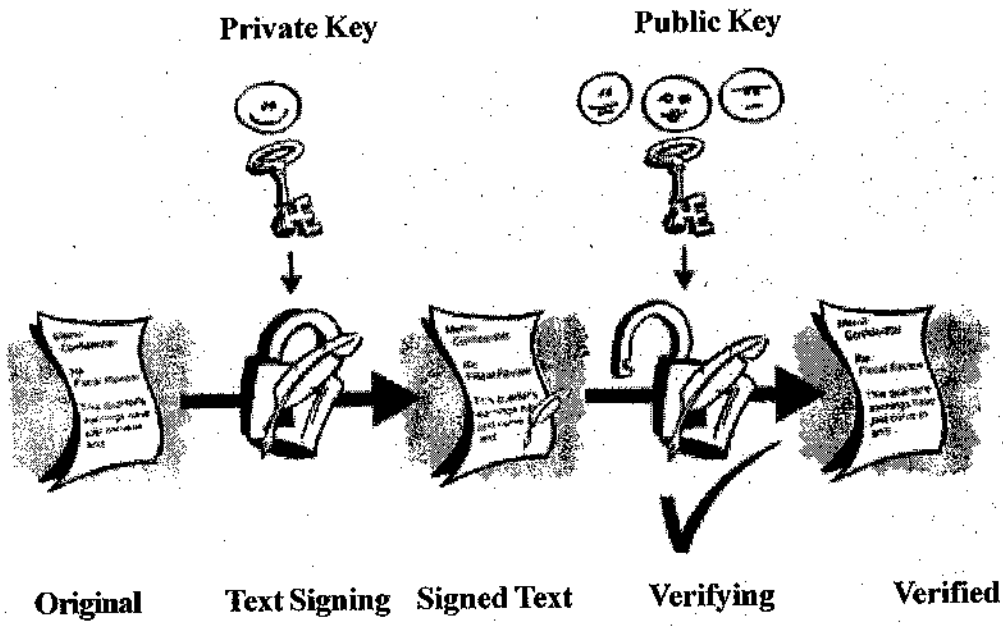


Figure 13.5: Simple Digital Signatures

Secret key cryptography schemes are generally categorized as being either stream ciphers or block ciphers. Stream ciphers operate on a single bit (byte or computer word) at a time and implement some form of feedback mechanism so that the key is constantly changing. A block cipher is so-called because the scheme encrypts one block of data at a time using the same key on each block. In general, the same plaintext block will always encrypt to the same ciphertext when using the same key in a block cipher whereas the same plaintext will encrypt to different ciphertext in a stream cipher.

Self-synchronizing stream ciphers calculate each bit in the keystream as a function of the previous n bits in the keystream. It is termed "self-synchronizing" because the decryption process can stay synchronized with the encryption process merely by knowing how far into the n-bit keystream it is. One problem is error propagation; a garbled bit in transmission will result in n garbled bits at the receiving side. Synchronous stream ciphers generate the keystream in a fashion independent of the message stream but by using the same keystream generation function at sender and receiver. While stream ciphers do not propagate transmission errors, they are, by their nature, periodic so that the keystream will eventually repeat.

Block ciphers can operate in one of several modes; the following four are the most important:

* Electronic Codebook (ECB) mode is the simplest, and in most of the applications the secret key is used to encrypt the plaintext block to form a ciphertext block. Two identical plaintext blocks, then, will always generate the same ciphertext block. Although this is the most common mode of block ciphers, it is susceptible to a variety of brute-force attacks.

* Cipher Block Chaining (CBC) mode adds a feedback mechanism to the encryption scheme. In CBC, the plaintext is exclusively-ORed (XORed) with the previous ciphertext block prior to encryption. In this mode, two identical blocks of plaintext never encrypt to the same ciphertext.

* Cipher Feedback (CFB) mode is a block cipher implementation as a self-synchronizing stream cipher. CFB mode allows data to be encrypted in units smaller than the block size, which might be useful in some applications such as encrypting interactive terminal input.

* Output Feedback (OFB) mode is a block cipher implementation conceptually similar to a synchronous stream cipher. OFB prevents the same plaintext block from generating the same ciphertext block by using an internal feedback mechanism that is independent of both the plaintext and ciphertext bitstreams.

Secret key cryptography algorithms that are in use today include:

* Data Encryption Standard (DES): It is the most common Secret Key Cryptography scheme used today. DES was designed by IBM in the 1970s and adopted by the National Bureau of Standards (NBS) in 1977 for commercial and unclassified government applications. DES is a block-cipher employing a 56-bit key that operates on 64-bit blocks. DES has a complex set of rules and transformations that were designed specifically to yield fast hardware implementations and slow software implementations. DES is defined in American National Standard X3.92 and three Federal Information Processing Standards (FIPS):

o FIPS 46-3: DES

o FIPS 74: Guidelines for Implementing and Using the NBS Data Encryption Standard

o FIPS 81: DES Modes of Operation

Two important variants that strengthen DES are:

o Triple-DES (3DES): A variant of DES that employs up to three 56-bit keys and makes three encryption/decryption passes over the block.

o DESX: A variant devised by Ron Rivest. By combining 64 additional key bits to the plaintext prior to encryption, effectively increases the key-length to 120 bits.

* Advanced Encryption Standard (AES): In 1997, NIST initiated a very public, 4-1/2 year process to develop a new secure cryptosystem for U.S. government applications. AES uses an SKC scheme called Rijndael, a block cipher designed by Belgian cryptographers Joan Daemen and Vincent Rijmen. The algorithm can use a variable block length and key length; the latest specification allowed any combination of keys lengths of 128, 192, or 256 bits and blocks of length 128, 192, or 256 bits. FIPS PUB 197 describes a 128-bit block cipher employing a 128-, 192-, or 256-bit key.

* CAST-128/256: CAST-128, described in Request for Comments (RFC) 2144, is a DES-like substitution-permutation crypto algorithm, employing a 128-bit key operating on a 64-bit block. CAST-256 (RFC 2612) is an extension of CAST-128, using a 128-bit block size and a variable length (128, 160, 192, 224, or 256 bit) key.

* International Data Encryption Algorithm (IDEA): Secret-key cryptosystem written by Xuejia Lai and James Massey, in 1992 and patented by Ascom; a 64-bit SKC block cipher using a 128-bit key. Also available internationally.

* Rivest Ciphers (aka Ron's Code): Named for Ron Rivest, a series of SKC algorithms.

o **RC2 :** A 64-bit block cipher using variable-sized keys designed to replace DES. It's code has not been made public although many companies have licensed RC2 for use in their products. Described in RFC 2268.

o **RC3 :** Found to be breakable during development.

o **RC4 :** A stream cipher using variable-sized keys; it is widely used in commercial cryptography products, although it can only be exported using keys that are 40 bits or less in length.

o **RC5 :** A block-cipher supporting a variety of block sizes, key sizes, and number of encryption passes over the data. Described in RFC 2040.

o **RC6 :** An improvement over RC5, RC6 was one of the AES Round 2 algorithms.

* **Blowfish :** A symmetric 64-bit block cipher optimized for 32-bit processors with large data caches, it is significantly faster than DES on a Pentium/PowerPC-class machine. Key lengths can vary from 32 to 448 bits in length.

* **Twofish :** A 128-bit block cipher using 128-, 192-, or 256-bit keys. Designed to be highly secure

and highly flexible, well-suited for large microprocessors, 8-bit smart card microprocessors, and dedicated hardware.

* **Camellia :** A secret-key, block-cipher crypto algorithm developed jointly by Nippon Telegraph and Telephone (NTT) Corp. and Mitsubishi Electric Corporation (MEC) in 2000. Camellia has some characteristics in common with AES: a 128-bit block size, support for 128-, 192-, and 256-bit key lengths, and suitability for both software and hardware implementations on common 32-bit processors as well as 8-bit processors (e.g., smart cards, cryptographic hardware, and embedded systems).

* **MISTY1 :** Developed at Mitsubishi Electric Corp., a block cipher using a128-bit key and 64-bit blocks, and a variable number of rounds. Designed for hardware and software implementations, and is resistant to differential and linear cryptanalysis.

* Secure and Fast Encryption Routine (SAFER): Secret-key crypto scheme designed for implementation in software. Versions have been defined for 40-, 64-, and 128-bit keys.

* **KASUMI :** A block cipher using a 128-bit key that is part of the Third-Generation Partnership Project (3gpp), formerly known as the Universal Mobile Telecommunications System (UMTS). KASUMI is the intended confidentiality and integrity algorithm for both message content and signaling data for emerging mobile communications systems.

* **SEED :** A block cipher using 128-bit blocks and 128-bit keys. Developed by the Korea Information Security Agency (KISA) and adopted as a national standard encryption algorithm in South Korea.

* **Skipjack :** SKC scheme proposed for Capstone. Although the details of the algorithm were never made public, Skipjack was a block cipher using an 80-bit key and 32 iteration cycles per 64-bit block.

13.5.2 Public Key Cryptography - RSA Method

Public-key cryptography has been said to be the most significant new development in cryptography in the last 300-400 years. Modern PKC was first described publicly by Stanford University professor Martin Hellman and graduate student Whitfield Diffie in 1976. They described a two-key cryptosystem in which two parties could engage in a secure communication over a non-secure communications channel without having to share a secret key.

PKC depends upon the existence of so-called one-way functions, or mathematical functions that are easy to computer whereas their inverse function is relatively difficult to compute. Let we give you two simple examples:

1. **Multiplication vs. factorization :** Suppose we tell you that we have two numbers, 9 and 16, and that we want to calculate the product; it should take almost no time to calculate the product, 144. Suppose instead that we tell you that we have a number, 144, and we need you tell us which pair of integers we multiplied together to obtain that number. You will eventually come up with the solution but whereas calculating the product took milliseconds, factoring will take longer because you first need to find the 8 pair of integer factors and then determine which one is the correct pair.

2. **Exponentiation vs. logarithms :** Suppose we tell you that we want to take the number 3 to the 6th power; again, it is easy to calculate $36=729$. But if we tell you that we have the number 729 and want you to tell us the two integers that we used, x and y so that $\log x\ 729 = y$, it will take you longer to find all possible solutions and select the pair that we used.

While the examples above are trivial, they do represent two of the functional pairs that are used with PKC; namely, the ease of multiplication and exponentiation versus the relative difficulty of factoring and calculating logarithms, respectively. The mathematical "trick" in PKC is to find a trap door in the one-way function so that the inverse calculation becomes easy given knowledge of some item of information.

Generic PKC employs two keys that are mathematically related although knowledge of one key does not allow someone to easily determine the other key. One key is used to encrypt the plaintext and the other key is used to decrypt the ciphertext. The important point here is that it does not matter which key is applied first, but that both keys are required for the process to work. Because a pair of keys are required, this approach is also called asymmetric cryptography.

In PKC, one of the keys is designated the public key and may be advertised as widely as the owner wants. The other key is designated the private key and is never revealed to another party. It is straight forward to send messages under this scheme. Suppose A wants to send B a message. A encrypts some information using B's public key; B decrypts the ciphertext using his private key. This method could be also used to prove who sent a message; A, for example, could encrypt some plaintext with her/him private key; when B decrypts using A's public key, he/she knows that A sent the message and A cannot deny having sent the message (non-repudiation).

Public-key cryptography algorithms that are in use today for key exchange or digital signatures include:

* **RSA :** The first, and still most common, PKC implementation, named for the three MIT mathematicians who developed it and they are Ronald Rivest, Adi Shamir, and Leonard Adleman. RSA today is used in hundreds of software products and can be used for key exchange, digital signatures, or encryption of small blocks of data. RSA uses a variable size encryption block and a variable size key. The key-pair is derived from a very large number, n, that is the product of two prime numbers chosen according to special rules; these primes may be 100 or more digits in length each, yielding an n with roughly twice as many digits as the prime factors. The public key information includes n and a derivative of one of the factors of n; an attacker cannot determine the prime factors of n (and, therefore, the private key) from this information alone and that is what makes the RSA algorithm so secure. (Some descriptions of PKC erroneously state that RSA's safety is due to the difficulty in factoring large prime numbers. In fact, large prime numbers, like small prime numbers, only have two factors!) The ability for computers to factor large numbers, and therefore attack schemes such as RSA, is rapidly improving and systems today can find the prime factors of numbers with more than 200 digits. Nevertheless, if a large number is created from two prime factors that are roughly the same size, there is no known factorization algorithm that will solve the problem in a reasonable amount of time; a 2005 test to factor a 200-digit number took 1.5 years and over 50 years of compute time. Regardless, one presumed protection of RSA is that users can easily increase the key size to always stay ahead of the computer processing curve.

* **Diffie-Hellman :** After the RSA algorithm was published, Diffie and Hellman came up with their own algorithm. D-H is used for secret-key key exchange only, and not for authentication or digital signatures.

* **Digital Signature Algorithm (DSA) :** The algorithm specified in NIST's Digital Signature Standard (DSS), provides digital signature capability for the authentication of messages.

* **ElGamal :** Designed by Taher Elgamal, a PKC system similar to Diffie-Hellman and used for key exchange.

* **Elliptic Curve Cryptography (ECC) :** A PKC algorithm based upon elliptic curves. ECC can offer levels of security with small keys comparable to RSA and other PKC methods. It was designed for devices with limited compute power and/or memory, such as smartcards and PDAs.

* **Public-Key Cryptography Standards (PKCS) :** A set of interoperable standards and guidelines for public-key cryptography, designed by RSA Data Security Inc.

13.5.3 Hash Functions

The system described above has some problems. It is slow, and it produces an enormous volume

of data-at least double the size of the original information. An improvement on the above scheme is the addition of a one-way hash function in the process. A one-way hash function takes variable-length input-in this case, a message of any length, even thousands or millions of bits-and produces a fixed-length output; say, 160-bits. The hash function ensures that, if the information is changed in any way-even by just one bit-an entirely different output value is produced. Cryptosystem uses a cryptographically strong hash function on the plaintext the user is signing. This generates a fixed-length data item known as a message digest.

Then Cryptosystem uses the digest and the private key to create the "signature." Cryptosystem transmits the signature and the plaintext together. Upon receipt of the message, the recipient uses Cryptosystem to re-compute the digest, thus verifying the signature. Cryptosystem can encrypt the plaintext or not; signing plaintext is useful if some of the recipients are not interested in or capable of verifying the signature.

As long as a secure hash function is used, there is no way to take someone's signature from one document and attach it to another, or to alter a signed message in any way. The slightest change in a signed document will cause the digital signature verification process to fail.

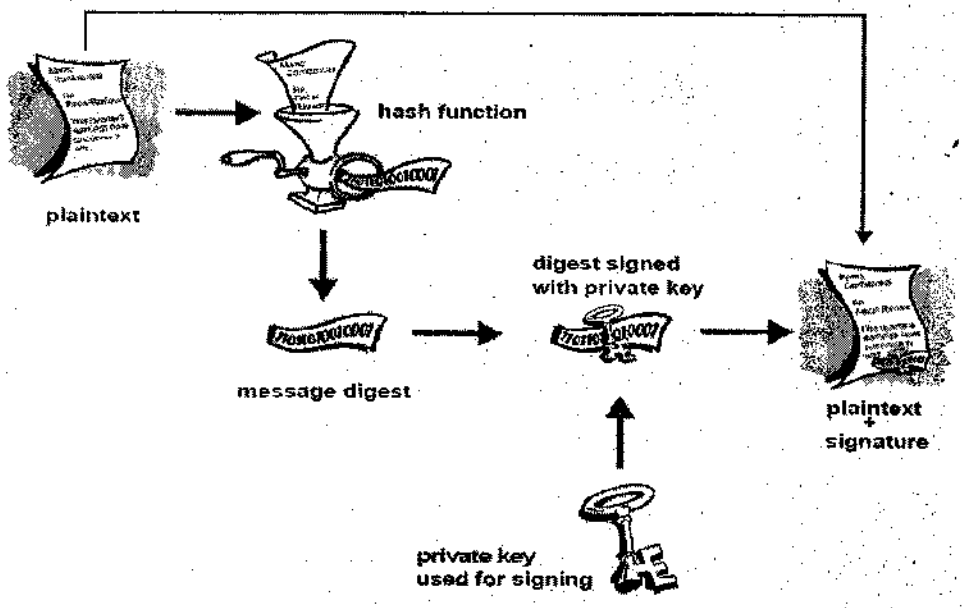


Figure 13.6: Secure Digital Signature

Digital signatures play a major role in authenticating and validating other cryptosystem users' keys.

13.6 AUTHENTICATION IN DISTRIBUTED SYSTEM

13.6.1 Performing one-way communication and digital signature

To better understand how cryptography is used to secure electronic communications, let's look at a process we are all familiar with: writing and sending a check.

Securing the electronic version

The simplest electronic version of the check can be a text file, created with a word processor, asking your bank to pay someone a specific sum. However, sending this check over an electronic network poses several security problems:

* since anyone could intercept and read the file, you need confidentiality.

* since someone else could create a similar counterfeit file, the bank needs to authenticate that it was actually you who created the file.

* since you could deny creating the file, the bank needs non-repudiation.

* since someone could alter the file, both you and the bank need data integrity.

One issue with public key cryptosystems is that users must be constantly vigilant to ensure that they are encrypting to the correct person's key. In an environment where it is safe to freely exchange keys via public servers, man-in-the-middle attacks are a potential threat. In this type of attack, someone posts a phony key with the name and user ID of the user's intended recipient. Data encrypted to and intercepted by the true owner of this bogus key is now in the wrong hands.

In a public key environment, it is vital that you are assured that the public key to which you are encrypting data is in fact the public key of the intended recipient and not a forgery. You could simply encrypt only to those keys which have been physically handed to you. But suppose you need to exchange information with people you have never met; how can you tell that you have the correct key?

Digital certificates, or certs, simplify the task of establishing whether a key truly belongs to the purported owner.

Webster's dictionary defines certificate as "a document containing a certified statement, especially as to the truth of something." A certificate is a form of credential. Examples might be your passport, your social security card, or your birth certificate. Each of these has some information on it identifying you and some authorization stating that someone else has confirmed your identity. A digital certificate is data that functions much like a physical certificate. A digital certificate is information included with a person's public key that helps others verify that a key is genuine or valid.

A digital certificate consists of three things:

* A public key.

* Certificate information. ("Identity" information about the user, such as name, user ID, and so on.)

" One or more digital signatures.

The purpose of the digital signature on a certificate is to state that the certificate information has been attested to by some other person or entity. The digital signature does not attest to the authenticity of the certificate as a whole; it vouches only that the signed identity information goes along with, or is bound to, the public key.

13.6.2 Validity and Trust

Every user in a public key system is vulnerable to mistaking a phony key (certificate) for a real one. Validity is confidence that a public key certificate belongs to its purported owner. Validity is essential in a public key environment where you must constantly establish whether or not a particular certificate is authentic.

When you've assured yourself that a certificate belonging to someone else is valid, you can sign the copy on your keyring to attest to the fact that you've checked the certificate and that it's a good one. If you want others to know that you gave the certificate your stamp of approval, you can export the signature to a certificate server so that others can see it.

Some companies designate one or more Certification Authorities (CA), whose job it is to go around and check the validity of all the certificates in the organization and then sign the good ones.

With secret key cryptography, a single key is used for both encryption and decryption. As shown in Figure 1A, the sender uses the key (or some set of rules) to encrypt the plaintext and sends the ciphertext to the receiver. The receiver applies the same key (or ruleset) to decrypt the message and recover the plaintext. Because a single key is used for both functions, secret key cryptography is also called symmetric encryption.

With this form of cryptography, it is obvious that the key must be known to both the sender and the receiver; that, in fact, is the secret. The biggest difficulty with this approach is the distribution of the key.

13.6.3 Digital signature

The process of digitally signing starts by taking a mathematical summary (called a hash code) of the check. This hash code is a uniquely-identifying digital fingerprint of the check. If even a single bit of the check changes, the hash code will dramatically change. The next step in creating a digital signature is to sign the hash code with your private key. This signed hash code is then appended to the check. The recipient of your check can verify the hash code sent by you, using your public key. At the same time, a new hash code can be created from the received check and compared with the original signed hash code. If the hash codes match, then the recipient has verified that the check has not been altered. The recipient also knows that only you could have sent the check because only you have the private key that signed the original hash code.

13.6.4 Confidentiality and encryption

Once the electronic check is digitally signed, it can be encrypted using a high-speed mathematical transformation with a key that will be used later to decrypt the document. This is often referred to as a symmetric key system because the same key is used at both ends of the process.

As the check is sent over the network, it is unreadable without the key. The next challenge is to securely deliver the symmetric key to the bank.

13.6.5 Public-key cryptography for delivering symmetric keys

Public-key encryption is used to solve the problem of delivering the symmetric encryption key to the bank in a secure manner. To do so, you would encrypt the symmetric key using the bank's public key. Since only the bank has the corresponding private key, only the bank will be able to recover the symmetric key and decrypt the check.

Why use this combination of public-key and symmetric cryptography? The reason is simple. Public-key cryptography is relatively slow and is only suitable for encrypting small amounts of information - such as symmetric keys. Symmetric cryptography is much faster and is suitable for encrypting large amounts of information.

13.7 KERBEROS: A CASE STUDY

Many users are under the misconception that by using their password to log on to a service, they are securing information that is sent over to the server. Unfortunately, this is typically not the case. Even if you do not keep confidential or important information online, you should still be concerned with network security. You may not care if someone reads your email, but you probably would be alarmed if your account was used to send email for the purpose of organizing a crime. Additionally, someone with your password may be able to make your account unusable to you. For these and other reasons, it's a good idea to protect your password and to practice secure networking.

13.7.1 An Introduction to Kerberos

Kerberos is a commonly used authentication scheme on the Internet. Developed by MIT's Project Athena, according to Greek mythology Kerberos is named for the three-headed dog who guards the entrance of Hades. Kerberos employs a client/server architecture and provides user-to-server authentication rather than host-to-host authentication. In this model, security and authentication will be based on secret key technology where every host on the network has its own secret key. It would clearly be unmanageable if every host had to know the keys of all other hosts so a secure, trusted host somewhere on the network, known as a Key Distribution Center (KDC), knows the keys for all of the hosts (or at least some of the hosts within a portion of the network, called a realm). In this way, when a new node is brought

online, only the KDC and the new node need to be configured with the node's key; keys can be distributed physically or by some other secure means.

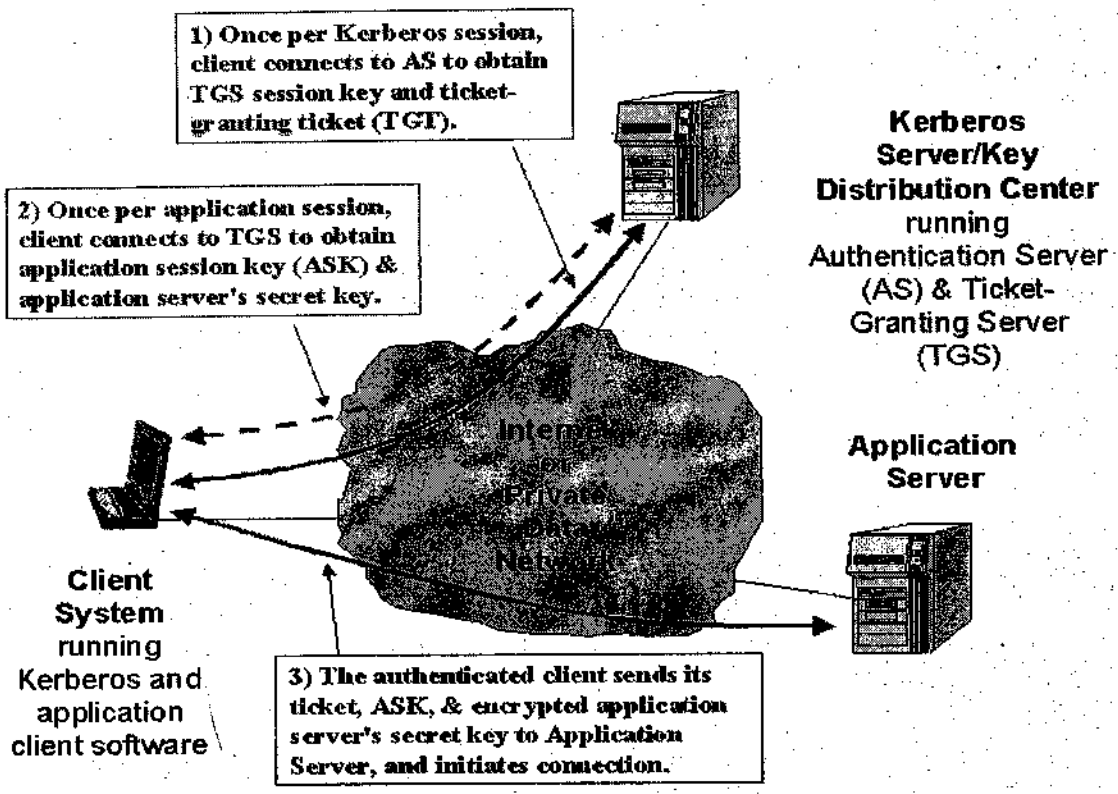


Figure 13.7: Kerberos architecture.

The Kerberos Server/KDC has two main functions (Figure 13.7), known as the Authentication Server (AS) and Ticket-Granting Server (TGS). The steps in establishing an authenticated session between an application client and the application server are:

1. The Kerberos client software establishes a connection with the Kerberos server's AS function. The AS first authenticates that the client is who it purports to be. The AS then provides the client with a secret key for this login session (the TGS session key) and a ticket-granting ticket (TGT), which gives the client permission to talk to the TGS. The ticket has a finite lifetime so that the authentication process is repeated periodically.
2. The client now communicates with the TGS to obtain the Application Server's key so that it (the client) can establish a connection to the service it wants. The client supplies the TGS with the TGS session key and TGT; the TGS responds with an application session key (ASK) and an encrypted form of the Application Server's secret key; this secret key is never sent on the network in any other form.
3. The client has now authenticated itself and can prove its identity to the Application Server by supplying the Kerberos ticket, application session key, and encrypted Application Server secret key. The Application Server responds with similarly encrypted information to authenticate itself to the client. At this point, the client can initiate the intended service requests (e.g., Telnet, FTP, HTTP, or e-commerce transaction session establishment).

The current shipping version of this protocol is Kerberos V5, although Kerberos V4 still exists and is seeing some use. While the details of their operation, functional capabilities, and message formats are different, the conceptual overview above pretty much holds for both. One primary difference is that Kerberos V4 uses only DES to generate keys and encrypt messages, while V5 allows other schemes to be employed (although DES is still the most widely algorithm used).

13.7.2 How Kerberos Works

Kerberos is an authentication protocol which uses a shared secret and a trusted third party arbi-

trator in order to validate the identity of clients. In Kerberos, clients may be users, servers, or pieces of software. The trusted third party arbitrator is a server known as a Key Distribution Center (KDC) which runs the Kerberos daemons. The shared secret is the users password transformed into a cryptographic key. In the case of servers or software systems, a random key is generated.

In Kerberos, users are known as principals. The KDC has a database of principals and their secret keys which is uses to perform authentication. In Kerberos knowledge of the secret key is considered sufficient for proof of identity. Since knowledge of a secret key translates into proof of identity in Kerberos, the Kerberos server can be trusted to authenticate any client to any other client. Authentication is Kerberos is done with out sending any clear text passwords across the wire.

13.7.3 The Benefits of Kerberos

For individuals unfamiliar with the Kerberos protocol, the benefits of deploying it in their network may not be clear. However, all administrators are familiar with the problems Kerberos was designed to mitigate. Those problems include, password sniffing, password filename/database stealing, and the high level of effort necessary to maintain a large number of account databases.

A properly deployed Kerberos Infrastructure will help you address these problems. It will make your enterprise more secure. Use of Kerberos will prevent plaintext passwords from being transmitted over the network. The Kerberos system will also centralize your username and password information which will make it easier to maintain and manage this data. Finally, Kerberos will also prevent you from having to store password information locally on a machine, whether it is a workstation or server, thereby reducing the likelihood that a single machine compromise will result in additional compromises.

To summarize, in a large enterprise, the benefits of Kerberos will translate into reduced administration costs through easier account and password management and through improved network security. In a smaller environment, scalable authentication infrastructure and improved network security are the clear benefits.

Check your progress 13.2

1. DES stands for _____.

   (a) Data Encryption Standard (b) Data Encryption System

   (c) Data Editing System (d) None of above

2. Schemes which can be proven secure using only complexity assumptions are said to be secure in the _____.

   (a) Random Oracle Model (b) Standard Model

   (c) Generic Group Model (d) None of above

3. SKC stands for _____.

   (a) System Key Cipher (b) Secure Key Code

   (c) Secret Key Cryptography (d) None of above

4. PKC stands for _____.

   (a) Public Key Cryptography (b) Public Key Cipher

   (c) Pure Key Cipher (d) None of above

5. _____uses a mathematical transformation to irreversibly "encrypt" information.

   (a) PKC (b) SKC

   (c) DES (d) Hash Functions

13.8 SUMMARY

Cryptography is the art and science of creating messages that have some combination of being private, signed, unmodified with non-repudiation. Cryptographic system and mechanisms are very useful and they generally provide a strong security to data in a distributed environment. Encryption / Decryption algorithms, digital signature and digital certificates helps to protect data while it is in transmission.

13.9 GLOSSARY

Algorithm (encryption) :

a set of mathematical rules (logic) used in the processes of encryption and decryption.

Algorithm (hash) :

a set of mathematical rules (logic) used in the processes of message digest creation and key/ signature generation.

Asymmetric keys :

a separate but integrated user key-pair, comprised of one public key and one private key. Each key is one way, meaning that a key used to encrypt information can not be used to decrypt the same data.

Authentication :

to prove genuine by corroboration of the identity of an entity.

Authorization certificate :

an electronic document to prove one's access or privilege rights, also to prove one is who they say they are.

Authorization :

to convey official sanction, access or legal power to an entity.

Blind signature :

ability to sign documents without knowledge of content, similar to a notary public.

Block cipher :

a symmetric cipher operating on blocks of plain text and cipher text, usually 64 bits.

CBC (Cipher Block Chaining) :

the process of having plain text XORed with the previous cipher text block before it is encrypted, thus adding a feedback mechanism to a block cipher.

CDK (Crypto Developer Kit) :

a documented environment, including an API for third parties to write secure applications using a specific vendor's cryptographic library.

Certificate (digital certificate) :

an electronic document attached to a public key by a trusted third party, which provides proof that the public key belongs to a legitimate owner and has not been compromised.

CFM (Cipher Feedback Mode) :

a block cipher that has been implemented as a self-synchronizing stream cipher.

CDSA (Common Data Security Architecture) :

Intel Architecture Labs (IAL) developed this framework to address the data security problems inherent to Internet and Intranet for use in Intel and others' Internet products.

Certification :

endorsement of information by a trusted entity.

CHAP (Challenge Authentication Protocol) :

a session-based, two-way password authentication scheme.

Cipher text :

the result of manipulating either characters or bits via substitution, transposition, or both.

Clear text :

characters in a human readable form or bits in a machine-readable form (also called plain text).

Cryptography :

the art and science of creating messages that have some combination of being private, signed, unmodified with non-repudiation.

Cryptosystem :

a system comprised of cryptographic algorithms, all possible plain text, cipher text, and keys.

Decryption :

the process of turning cipher text back into plain text.

DES (Data Encryption Standard) :

a 64-bit block cipher, symmetric algorithm also known as Data Encryption Algorithm (DEA) by ANSI and DEA-1 by ISO. Widely used for over 20 years, adopted in 1976 as FIPS 46.

DSA (Digital Signature Algorithm) :

a public key digital signature algorithm proposed by NIST for use in DSS.

Digital signature :

an electronic identification of a person or thing created by using a public key algorithm. Intended to verify to a recipient the integrity of data and identity of the sender of the data.

DSS (Digital Signature Standard) :

a NIST proposed standard (FIPS) for digital signatures using DSA.

Encryption :

the process of disguising a message in such a way as to hide its substance.

Filter :

a function, set of functions, or combination of functions that applies some number of transforms to its input set, yielding an output set containing only those members of the input set that satisfy the transform criteria. The selected members may or may not be further transformed in the resultant output set. An example would be a search function that accepts multiple strings having a boolean relationship (like a or like b) but not containing c), and optionally forces the case of the found strings in the resultant output.

Hash function :

a one-way hash function - a function that produces a message digest that cannot be reversed to produced the original.

IDEA (International Data Encryption Standard) :

a 64-bit block symmetric cipher using 128-bit keys based on mixing operations from different algebraic groups.

Initialization vector (IV) :

a block of arbitrary data that serves as the starting point for a block cipher using a chaining feedback mode (see cipher block chaining).

Kerberos :

a trusted third-party authentication protocol developed at MIT.

Key :

a means of gaining or preventing access, possession, or control represented by any one of a large number of values.

Key escrow/recovery :

a mechanism that allows a third party to retrieve the cryptographic keys used for data confidentiality, with the ultimate goal of recovery of encrypted data.

Key exchange :

a scheme for two or more nodes to transfer a secret session key across an unsecured channel.

Key management :

the process and procedure for safely storing and distributing accurate cryptographic keys; the overall process of generating and distributing cryptographic key to authorized recipients in a secure manner.

One-way hash :

a function of a variable string to create a fixed length value representing the original pre-image, alsc called message digest, fingerprint, message integrity check (MIC).

Private key :

the privately held "secret" component of an integrated asymmetric key pair, often referred to as the decryption key.

Public key :

the publicly available component of an integrated asymmetric key pair often referred to as tne encryption key.

RSA: short for RSA Data Security, Inc. :

or referring to the principals - Ron Rivest, Adi Shamir, and Len Adleman; or referring to the algorithm they invented. The RSA algorithm is used in public key cryptography and is based on the fact that it is easy to multiply two large prime numbers together, but hard to factor them out of the product.

Secret key :

either the "private key" in public key (asymmetric) algorithms or the "session key" in symmetric algorithms.

SET (Secure Electronic Transaction) :

provides for secure exchange of credit card numbers over the Internet.

Stream cipher :

a class of symmetric key encryption where transformation can be changed for each symbol of plain text being encrypted, useful for equipment with little memory to buffer data.

Substitution cipher :

the characters of the plain text are substituted with other characters to form the cipher text.

Symmetric algorithm :

a.k.a., conventional, secret key, and single key algorithms; the encryption and decryption key are either the same or can be calculated from one another.

Two sub-categories exist - Block and Stream.

Triple DES :

an encryption configuration in which the DES algorithm is used three times with three different keys.

TTP (Trust Third-Party) :

a responsible party in which all participants involved agree upon in advance, to provide a service or function, such as certification, by binding a public key to an entity, time-stamping, or key-escrow.

Validation :

a means to provide timeliness of authorization to use or manipulate information or resources.

Verification: to authenticate, confirm, or establish accuracy.

VPN (Virtual Private Network) :

allows private networks to span from the end-user, across a public network (Internet) directly to the Home Gateway of choice, such as your company's Intranet.

WAKE (Word Auto Key Encryption) :

produces a stream of 32-bit words, which can be XORed with plain text stream to produce cipher text, invented by David Wheeler.

XOR :

exclusive-or operation; a mathematical way to represent differences.

13.10 FURTHER READINGS

1. George Coulouris, 'Distributed Systems: Concepts and Design' (Fourth Edition), Pearson Education Ltd.
2. Andrew S. Tanenbaum, 'Distributed Operating Systems' (Fourth Edition), Pearson Education Ltd.
3. M.L.Liu, 'Distributed Computing: Principles and Applications' (Third Edition), Pearson Education Ltd
4. Andrew S. Tanenbaum, 'Distributed Systems: Principles and Paradigms' (Third Edition), Pearson Education Ltd.

5. Howard, J.H., Kazar, M.L., Menees, S.G., Nichols, D.A., Satyanaryanan, M. & Sidebotham, R.N. "Scale and Performance in Distributed File Systems." ACM Trans. Computing Systems, Vol. 6, No. 1, (February), 1988.

6. National Bureau of Standards, "Data Encryption Standard." FIPS Publication #46, NTIS, Apr. 1977.

13.11 ANSWER TO THE SELF LEARNING EXERCISES

Answer to check your progress 13.1

1. (a) 2. (a) 3. (b) 4. (b) 5. (c)

Answer to check your progress 13.2

1. (a) 2. (b) 3. (c) 4. (a) 5. (d)

13.12 UNIT-END QUESTIONS

1. What is 'Cryptography'? Explain Encryption and Decryption process.
2. Explain Digital Signatures.
3. What is Digital Certificate ? explain in brief.
4. Explain 'Private Key Encryption' and 'Public Key Encryption' in brief.
5. What is 'Kerberos'? Explain Kerberos architecture in brief.

□□□□

UNIT 14 : MULTIPROCESSOR OPERATING SYSTEM

Structure of the Unit

14.0 OBJECTIVES

After going through this unit student will be able to:

- To understand about multiprocessors
- Different types of multiprocessors
- Learn about the Interconnection networks.
- Able to understand the threads, process of synchronization
- Able to understand about scheduling and memory management

14.1 INTRODUCTION

To increase the speed of computer hardware, we can enhance the new technologies and fabrica-

tion process in the computer systems, but it will increase the cost and other issues. The alternate solution to increase the efficiency of computer is multiprocessing. Such systems have more than one processor they have close communication to each other, sharing the computer bus, memory, clock and other devices.

A **multiprocessor** is a computer system in which two or more CPUs share full access to a common RAM. A program running on any of the CPUs sees a normal (usually paged) virtual address space. **The only unusual property this system has is that the CPU can write some value into a memory word and then read the word back and get a different value (because another CPU has changed it)**. When organized correctly, this property forms the basis of inter-processor communication: one CPU writes some data into memory and another one reads the data out.

There are several reasons to use multiprocessing systems; the first one is to increase system throughput by exciting different processes on different processor in parallel. We can get more work done in short time. Second is application speed up ratio by executing different portions of any application to different processors.

14.2 MULTIPROCESSORS

Although all multiprocessors have the property that every CPU can address all of memory, some multiprocessors have the additional property that every memory word can be read as fast as every other memory word. These machines are called **UMA (Uniform Memory Access)** multiprocessors. In contrast, **NUMA (Non-uniform Memory Access)** multiprocessors do not have this property.

Parallel processing demands the use of efficient system interconnects for fast communication among multiple processors and shared memory Input and Output and other peripheral devices. It uses hierarchical buses, crossbar switches and multi stage networks.

14.2.1 Advantages of Multiprocessors

Some advantages to use multiprocessors system are as follows:-

- **Reliability and Performance** - Multiprocessor systems increases reliability. If any problem distributed properly to several processors, then the failure of any processor will not halt the whole system, but it only slows the system. Performance of multiprocessor is much higher because problems can be solved. More quickly than in a distributed system because the bandwidth of inter-process communication is higher than distributed systems.

- **Fault Tolerance** - If we are having *n* processors and if one processor fails, than *n-1* processors share the work of that processor. Systems that are designed for such degradation are called fault tolerant systems.

- **Cost** - Because of the several processors are in a single system the money is saved as no need to other peripherals like memory buses or other devices.

14.2.2 Classification of Multiprocessors

Multiprocessors can be classified on the basis of processors and memory is as under-

- **Tightly Coupled - Tightly-coupled** multiprocessor systems contain multiple CPUs that are connected at the bus level. These CPUs may have access to a central shared memory, or may participate in a memory hierarchy with both local and shared memory. The shared global memory can access by all or many processors. This shared memory is fundamental for inter-process communication. The bandwidth is higher and delays are lower in tightly coupled multiprocessors.

Tightly-coupled systems perform better and are physically smaller than loosely-coupled systems, but have historically required greater initial investments and may depreciate rapidly.

Power consumption is also a consideration. Tightly-coupled systems tend to be much more energy efficient than clusters. This is because considerable economies can be realized by designing components to work together from the beginning in tightly-coupled systems.

- **Loosely Coupled - Loosely-coupled** multiprocessor systems (often referred to as clusters) are based on multiple standalone single or dual processor commodity computers interconnected via a high speed communication system (Gigabit Ethernet is common). These systems have their individual processors and private memories. In loosely coupled systems, message passing is the fundamental for inter-process communication. In such systems, bandwidth is lower and delays are higher their interconnection paths.

Nodes in a loosely-coupled system are usually inexpensive commodity computers and can be recycled as independent machines upon retirement from the cluster.

Loosely-coupled systems use components that were not necessarily intended specifically for use in such systems. Therefore, they consume lot of power.

Some loosely system and hybrid systems allow processors to access the memory of other processors. We can classify shared memory multiprocessor on the memory architecture and access delay base as: -

- Uniform Memory Access (UMA)
- Non-Uniform Memory Access (NUMA)
- No Remote Memory Access (NORMA)

14.2.3 Uniform Memory Access (UMA)

UMA (Uniform Memory Access) refers to a tightly coupled multiprocessor, where every CPU has direct access to the same central memory. Each CPU can access memory data in the same (uniform) amount of time.

In such type of systems, the physical memory is uniformly shared by all the processors. All the processors have equal access time to all memory words that is why it is called uniform memory access. Processor can access all the available memory with the same speed. Every processor can use private cache. Peripherals are also shared in same way.

In UMA bus-based multiprocessors -

a) CPUs communicate via bus to RAM

b) CPUs have a local cache to reduce bus access

c) CPUs have private memory, shared memory access via bus

In UMA multiprocessor using a crossbar switch -- It alleviates bus access problems, but is expensive (grows as n2).

UMA multiprocessors using multistage switching networks can be built from 2 × 2 switches. Input to switches is in the form of a message

a) 2 × 2 switch b) Message format

UMA omega switching network is less costly than crossbar switch.

14.2.4 Non-Uniform Memory Access (NUMA)

NUMA (Non-Uniform Memory Access) implies a loosely coupled multiprocessor. Each CPU has direct access to its own local memory but not directly to those of the other CPUs. Access to their own

memory data is very fast compared to the time it takes to get data from another CPU's memory.

Remote memory access is done over an external data network (like a backplane or a local area network). Access times are non-uniform across CPUs.

In such systems, a time difference is occurred in accessing different memories, base on proximity of given processor and the complexity of fabrication of switch between the processors.

- Single address space visible to all CPUs
- Access to remote memory via commands

  - LOAD

  - STORE
- Access to remote memory slower than to local

In a NUMA system, each computer board has its own CPU and memory. It is slid along rails into a rack mount, where it connects to a combination power supply and high speed communications bus, called a backplane. When a needed memory address is on the same board, its data appears in the local memory data register. When a memory address refers to a location on some other board, the backplane hardware electronically transfers the data into the local memory data register from the remote board, over the backplane. Remote memory access appears to be identical to local memory access, but backplane transfer time, fast as it is, is much longer than the time it takes to access a local memory.

NUMA can be further separated into NUMA and NORMA architectures. (NORMA stands for NO Remote Memory Access through hardware.

14.2.5 No Remote Memory Access (NORMA)

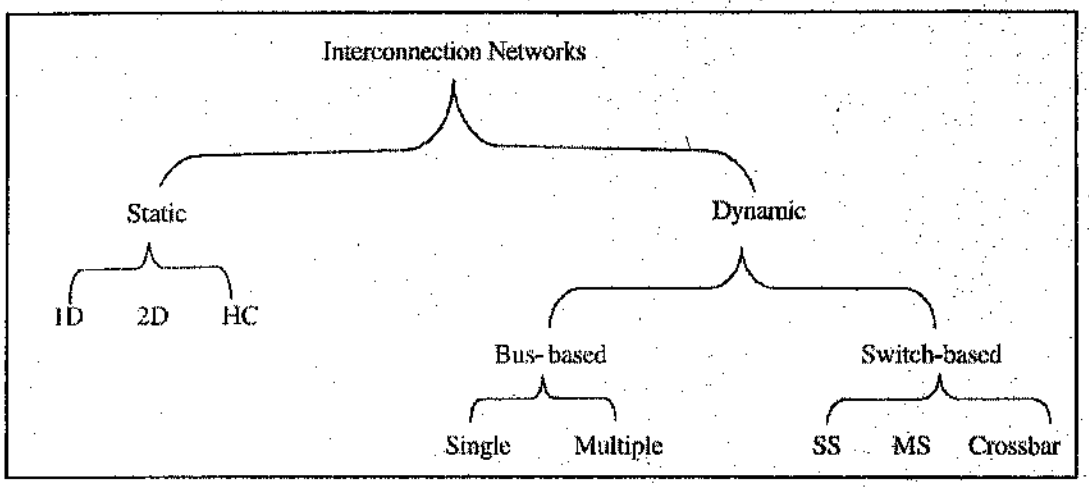
In **NORMA** systems, each node is a standalone computer. When one computer needs memory data from another, software catches the remote memory access and must send an inter-process communication to request that data from the remote machine. All application software stops until the remote machine sends that data back to the requesting system. When the data is moved into its expected location, management software resumes all the application processes again. These kinds of computers are also called a distributed shared memory computer, or DSM.

All this seem unnecessary to the outside observer, but these systems display very different properties, and computer people often need to talk about these differences.

These are the systems with no shared memory. NORMA architecture dictates message passing or the primary inter-processor communication and synchronization. This architecture is commonly occurred in loosely coupled systems.

14.3 Multiprocessor Interconnection Structure

An interconnection network could be either static or dynamic. Connections in a static network are fixed links, while connections in a dynamic network are established on the fly as needed. Static networks can be further classified according to their interconnection pattern as one-dimension (1D), two-dimension (2D), or hypercube (HC). Dynamic networks, on the other hand, can be classified based on interconnection scheme as bus-based versus switch-based. Bus-based networks can further be classified as single bus or multiple buses. Switch-based dynamic networks can be classified according to the structure of the interconnection network as single-stage (SS), multistage (MS), or crossbar networks.



Parallel processing requires efficient system interconnects for fast communication among multiple processors, shared memory, I/O and peripheral devices. Shared bus system, crossbar switches and multistage networks are used for this purpose. These are based on the topology, timing protocol, switching method and control strategy. Dynamic networks are used in multiprocessors in which the interconnections are under program control. The timing, switching and control are the main operational characteristics of an interconnection network.

The timing control can be either synchronous or asynchronous.

The synchronous networks are managed by a global clock that synchronizes all the network activities and Asynchronous networks uses handshaking or interlocking mechanisms to coordinate fast and slow devices requesting the use of the same network.

Network can transfer data using either circuit switching or packet switching. In circuit switching, when a device is granted a path in the network, it occupies the complete path for the entire data transfer session. In packet switching, the data is broken into small packets individually competing for a path/way in the network.

Control strategy can be centralized or distributed. In centralized control, a global controller receives requests from all the devices attached in the network and allows the network access to one or more requesters. In distributed system, requests are handled by local devices independently.

The components of multiprocessor operating systems are process, input/output devices and a memory unit, which are interconnected to each other. There are several forms for interconnecting the networks as follows: -

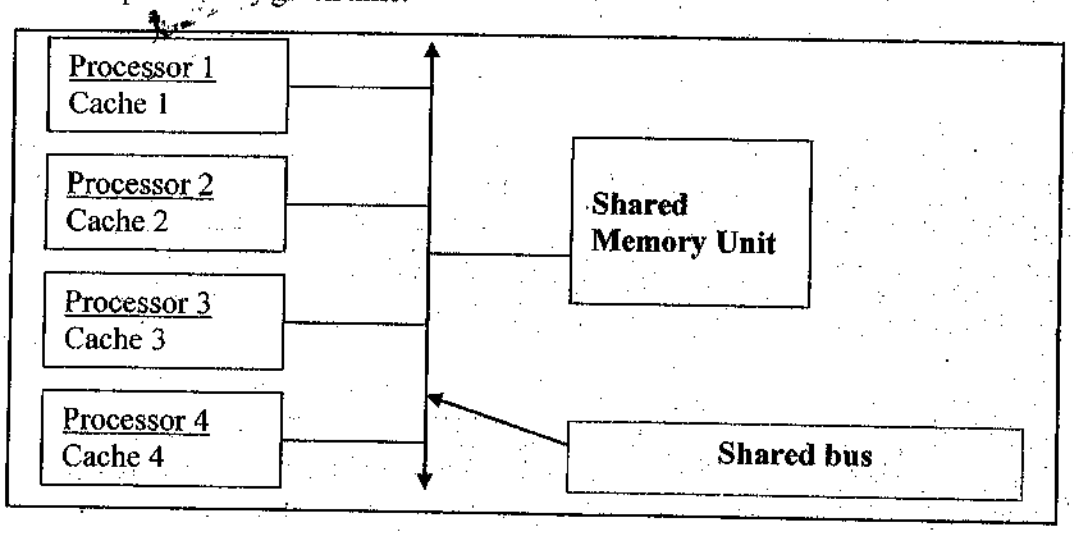
- Bus based System.
- single-stage switch based system
- Multistage Switching Based System.
- Crossbar – switching System

14.3.1 Bus based Systems

Single Bus Systems

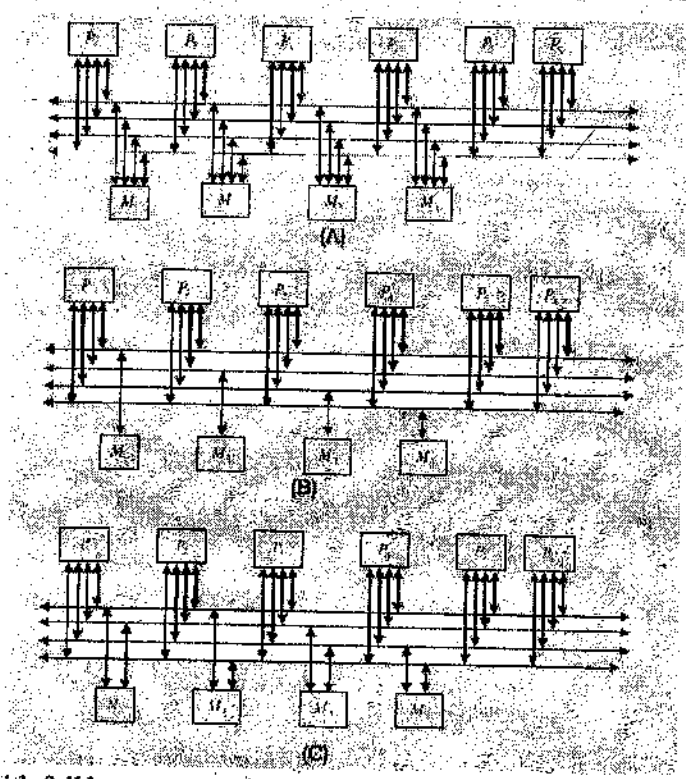
A single bus is considered the simplest way to connect multiprocessor systems. In its general form, such a system consists of N processors, each having its own cache, connected by a shared bus. The use of local caches reduces the processor–memory traffic. All processors communicate with a single shared memory. Although simple and easy to expand, single bus multiprocessors are inherently limited by the

bandwidth of the bus and the fact that only one processor can access the bus, and in turn only one memory access can take place at any given time.



Multiple Bus Systems

The use of multiple buses to connect multiple processors is a natural extension to the single shared bus system. A multiple bus multiprocessor system uses several parallel buses to interconnect multiple processors and multiple memory modules. A number of connection schemes are possible in this case. Among the possibilities are the multiple bus with full bus–memory connection (MBFBMC), multiple bus ith single bus memory connection (MBSBMC), multiple bus with partial bus– memory connection (MBPBMC), and multiple bus with class-based memory connection (MBCBMC). The multiple bus with full bus–memory connection has all memory modules connected to all buses. The multiple bus with single bus–memory connection has each memory module connected to a specific bus. The multiple bus with partial bus–memory connection has each memory module connected to a subset of buses.



(A) Multiple bus with full bus–memory connection (MBFBMC); (B) multiple bus with single bus-memory connection (MBSBMC); (C) multiple bus with partial bus– memory connection (MBPBMC);

A bus system consists of a hierarchy of buses connecting various system and subsystem compo-

nents in a computer. Each bus is formed with a number of signal, control and power lines. Different buses are used to perform different interconnection functions.

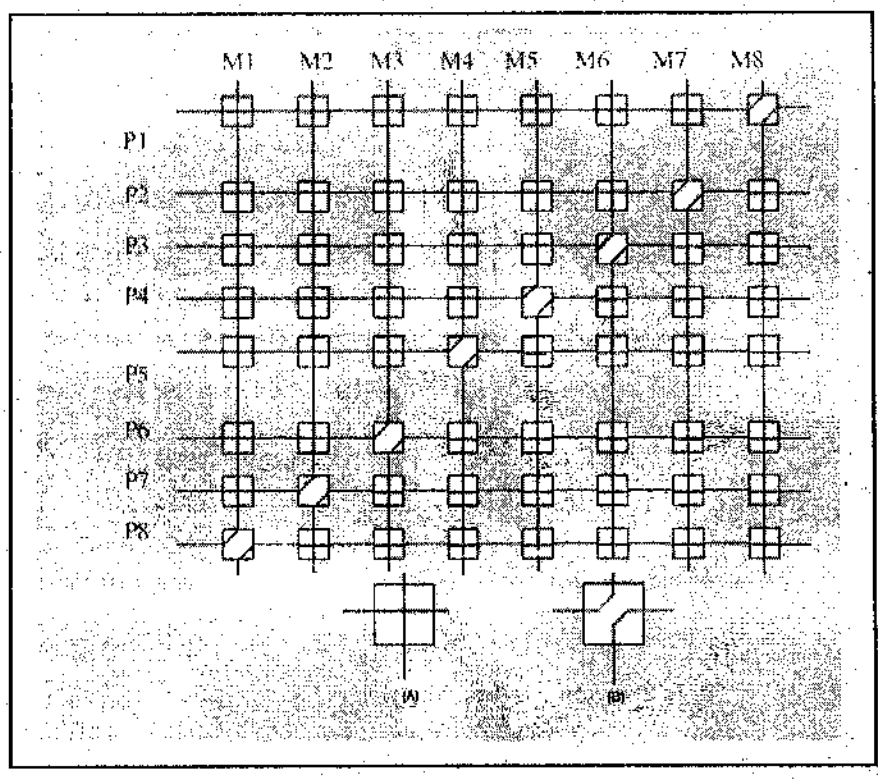
The multiple processors with their own cache memories are connected to a common shared memory unit with the use of a common shared bus. We can also use individual processor or to connect the shared –buses cache memory is often employed to reduce contention on the shared bus.

Basically, two arrangements of cache are possible in the shared-bus system. Cache is associated with the shared-memory and processors can access it over the bus. Cache is associated with each individual processor. The second approach is the cache can use many of the local memory reference and thus reduce contention for the common bus.

The primary advantage to use of cache is its ability to reduce the average access time. In shared-memory multiprocessor system, the same information may reside is number of copies in some local cache and the main memory.

14.3.2 Crossbar Networks

A crossbar network represents the other extreme to the limited single bus network. The single bus can provide only a single connection; the crossbar can provide simultaneous connections among all its inputs and all its outputs. The crossbar contains a switching element (SE) at the intersection of any two lines extended horizontally or vertically inside the switch.



An 8x8 crossbar network (A) straight switch setting (B) diagonal switch setting.

The 8 x 8 crossbar network shown in the above diagram, an SE (also called a cross-point) is provided at each of the 64 intersection points The two possible settings of an SE in the crossbar (straight and diagonal) are also shown in the diagram. The number of switching points required is 64 and the message delay to traverse from the input to the output is constant, regardless of which input/output are communicating.

In general for an N x N crossbar, the network complexity, measured in terms of the number of switching points, is $O(N^2)$ while the time complexity, measured in terms of the input to output delay, is

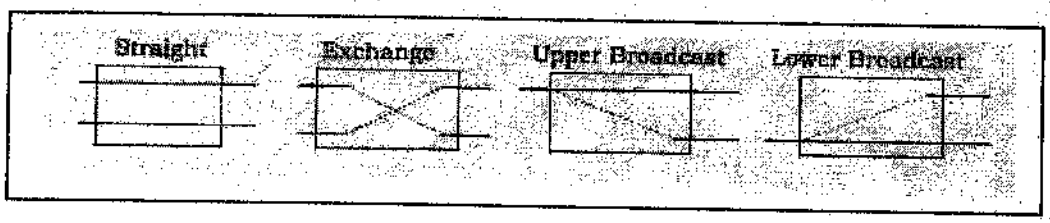
O(1). The complexity of the crossbar network pays off in the form of reduction in the time complexity. Also the crossbar is a non-blocking network that allows a multiple input–output connection pattern to be achieved simultaneously. For a large multiprocessor system the complexity of the crossbar can become a foremost financial factor.

14.3.3 Switch-based interconnection networks

In this type of network, connections among processors and memory modules are made using simple switches. Three basic interconnection topologies exist: crossbar, single-stage, and multistage.

Single-Stage Networks

In single stage networks, a single stage of switching elements (SEs) exists between the inputs and the outputs of the network. The switching element that can be used is the 2 x 2 switching element (SE). The four settings can be done by switching element. **These settings are called straight, exchange, upper-broadcast, and lower-broadcast.** In the straight setting, the upper input is transferred to the upper output and the lower input is transferred to the lower output. In the exchange setting the upper input is transferred to the lower output and the lower input is transferred to the upper output. In the upper-broadcast setting the upper input is broadcast to both the upper and the lower outputs. In the lower-broadcast the lower input is broadcast to both the upper and the lower outputs.

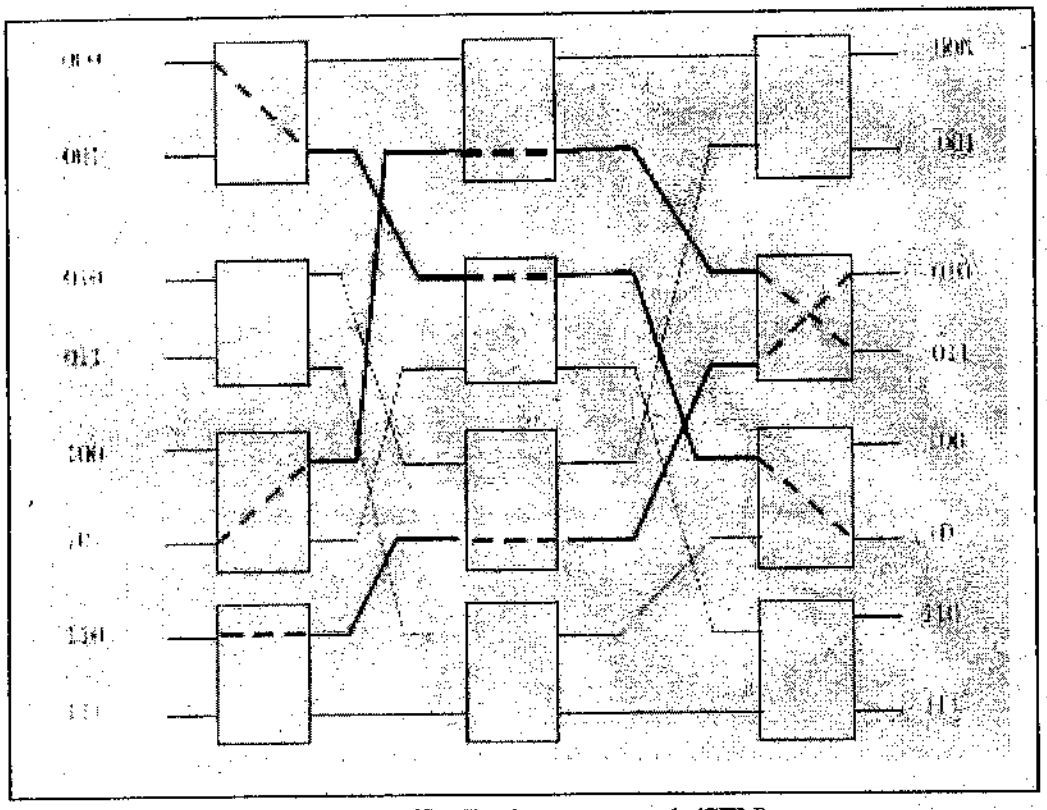


The different settings of the 2x2 SE.

Multistage Networks

Multistage interconnection networks (MINs) were introduced to improve some of the limitations of the single bus system while keeping the cost within an affordable limit. The MINs is set to improve is the availability of only one single path between the processors and the memory modules. Such multistage interconnection networks provide a number of simultaneous paths between the processors and the memory modules.

The routing of a message from a given source to a given destination is based on the destination address (self-routing). There exist $\log_2 N$ stages in an N x N Multistage interconnection networks. The number of bits in any destination address in the network is $\log_2 N$. Each bit in the destination address can be used to route the message thru one stage. The destination address bits are scanned from left to right and the stages are traversed from left to right. The first (most significant bit) is used to control the routing in the first stage; the next bit is used to control the routing in the next stage, and so on. For example, the routing of a message from source input 101 to destination output 011 in the 8 x 8 SEN shown in the following diagram-



An 8x8 Shuffle–Exchange network (SEN).

14.4 CACHE COHERENCE PROBLEM

Cache coherence problem exist commonly in multiprocessor system with local caches because of the need to share writable data. All inter-processor coordination and synchronization is also accomplished via the global memory.

Two main problems need to be addressed while designing a shared memory system: performance degradation due to contention, and coherence problems. Performance degradation happens when multiple processors are trying to access the shared memory simultaneously. A typical design might use caches to solve the contention problem.

Having multiple copies of data, spread throughout the caches, might lead to a coherence problem. The copies in the caches are coherent if they are all equal to the same value. if one of the processors writes over the value of one of the copies, then the copy becomes inconsistent because it no longer equals the value of the other copies.

The four combinations to maintain coherence among all caches and global memory are:

- Write-update and write-through;
- Write-update and write-back;
- Write-invalidate and write-through; and
- Write-invalidate and write-back.

If we permit a write-update and write-through directly on global memory location X, the bus would start to get busy and ultimately all processors would be idle while waiting for writes to complete. In write-update and write-back, only copies in all caches are updated. On the contrary, if the write is limited to the copy of X in cache Q, the caches become inconsistent on X. Setting the dirty bit prevents the spread of inconsistent values of X, but at some point, the inconsistent copies must be updated.

14.5 SOLUTION OF THE CACHE COHERENCE PROBLEM

Several solutions suggested to cache-coherence problem in multiprocessor shared memory. The simplest solution is to use a shared cache memory and not to use the local cache & for each process with shared memory.

This method violates the closeness of processor to caches and increases the memory access time. Other solution is to use only non-shared and read only data to be stored in caches. The complier must tag data as either cacheable or non cacheable, and the system hardware makes sure that only cacheable data are stored in cache. The non cacheable data are store in the main memory. This solution may degrade the performance.

The above two solution are software based procedures and the can slow-down the performance of the system. So, another solution is the combination of software and hardware or only by the hardware based.

Performances of such solutions are higher than previous two methods. In hardware based solution, a cache controller is designed to allow it to monitor all bus requests from processors. This controller is called snoopy cache controller.

Various solutions are performed by snoopy cache controller. The simplest is that the entire snoopy controllers watch the bus for memory store operation. When a data in a cache is updated, the corresponding location is also updated in main memory. The local snoop controllers of all other local caches check their cache memory to determine if they have a copy of data that is overwritten. If a copy exists in main cache, that location marked invalid. Because all caches snoop on all bus writes, when the word is written, the net effect is to update is in the original cache and the main memory and remove it from all other caches.

14.6 MULTIPROCESSOR OPERATING SYSTEM DESIGN ISSUES

There are some design and implementation issues which includes processor and memory management in multiprocessor systems. The operating system can support multiprocessing and parallelism by providing a mechanism for creation and management of a large number of processes and/or threads. The process model can be viewed to provide a virtual processor abstraction. Every process has a state a set of allocated resources like files and memory and I/O device connection.

A virtual multiprocessing is created by locate the physical processor to a different virtual processor during each time slice. Shred resource accessing is normally maintained by shared memory or by some inter-process communication mechanism.

In uni-processor operating systems, a process has a large number of words devoted to its state. This makes process creation and switching much expensive. To solve this problem threads are used.

14.6.1 Threads

An important aspect of modern operating systems is their support for threads within processes. A thread, sometimes called a lightweight process (LWP), is a basic unit of processor utilization. It runs sequentially on a processor and is interruptible so that the processor can switch to other threads. A process does nothing if it has no threads in it, and a thread must be in exactly one process. A thread is different from the traditional or heavy-weight process, which is equivalent to a task with one thread.

A thread is a lightweight process (LWP) represents a software approach to improve the performance of operating system. It is a basic unit of CPU utilization, and consists of a program counter, a register set and a stack space. Concurrency among processes can be exploited because threads in different processes may execute concurrently. Moreover, multiple threads within the same process can be assigned to different processors.

Threads can be in one of several states ready, blocked, running or terminated. Like processes threads share the processor and only thread at a time is active (running). Each thread belongs to exactly one process and no thread can exist outside of process.

In multiprocessing systems threads are used as implementing each application as a separate process, and its concurrent portions are coded as separate threads within the enclosing process. Threads belonging to a single process share the memory and all other resources acquired by the process.

14.6.2 Scheduling

Scheduling provides a resources control model in multiprocessor systems. Any available process can be used to run any processes in the queue. There are some problems on scheduling. Assume a system with an I/O device attached to a private bus of one processor. Processes wants to use that device must be schedule to run on that processor; otherwise the device would not be available. Individual processor in a multiprocessor system can be unit or multi-programmed, multiprogramming provides the potential for increasing throughput, but it can increase the problem of communication delays. The one solution is to schedule process groups that are known to communicate with each other. The other solution is asymmetric multiprocessing which simpler than symmetric multiprocessing, in which only single processor accesses the system data structure, alleviating the need for data sharing.

Another consideration of scheduling mechanism is the communication cost. The communication cost can be reduced by placing closely interacting processes into the same processor.

These considerations have to be parallelism, which is generally increased when processes are allocated to different processors.

14.6.3 Memory Management

A shared memory model is one in which processors communicate by reading and writing locations in a shared memory that is equally accessible by all processors. A number of basic issues in the design of shared memory systems have to be taken into consideration. These include access control, synchronization, protection, and security. Access control determines which process accesses are possible to which resources. Access control models make the required check for every access request issued by the processors to the shared memory, against the contents of the access control table. Requests from sharing processes may change the contents of the access control table during execution.

The flags of the access control with the synchronization rules determine the system's functionality. Synchronization constraints limit the time of accesses from sharing processes to shared resources. Appropriate synchronization ensures that the information flows properly and ensures system functionality.

Protection is a system feature that prevents processes from making arbitrary access to resources belonging to other processes. Sharing and protection are incompatible; sharing allows access, whereas protection restricts it.

In the multiprocessing environments, the operating system must facilitate and control access to shared memory. In the system that supports shared virtual memory, several translation look a side buffers (TLB) residing on every processor may contain mapping for pages belong to a shared segment.

In such cases, the operating system must cooperate with hardware to enforce TLB coherence. TLB coherence is normally supporting some techniques which are used for multiprocessor cache coherence.

Shared memory is used to improve the performance of message passing. This can be performed by avoiding the copying of messages whose inputs and outputs have access the same shared memory.

The write to back policy is also can used to maintain the high efficiency to avoid the copying of message in cache devices and the shared memory. In this approach, a unique copy of messages is maintained in shared memory. Message passing in performed by granting the receiver the right to access that copy the operating system then makes a updated copy of the affected page, which is then mapped to the updating process and is used to complete the write. The write to back technique is successfully applied to reduce the costs of process migration and of message passing in loosely coupled systems.

14.6.4 Process Synchronization

Synchronization is needed to protect shared variables by ensuring that they are accessed by only one process at a given time (mutual exclusion). They can also be used to coordinate the execution of parallel processes and synchronize at certain points in execution. There are two main synchronization constructs in shared memory systems: (1) locks and (2) barriers. Parallel processes using locks to ensure mutual exclusion.

14.7 SUMMARY

To increase the efficiency of computer is multiprocessing. Such systems have more than one processor they have close communication to each other, sharing the computer bus, memory, clock and other devices. A multiprocessor is a computer system in which two or more CPUs share full access to a common RAM. There are several reasons to use multiprocessing systems; the first one is to increase system throughput by exciting different processes on different processor in parallel. Second is application speed up ratio by executing different portions of any application to different processors.

Some multiprocessors have the additional property that every memory word can be read as fast as every other memory word. These machines are called UMA (Uniform Memory Access) multiprocessors. NUMA (Non-uniform Memory Access) multiprocessors do not have this property.

Tightly-coupled multiprocessor systems contain multiple CPUs that are connected at the bus level. These CPUs may have access to a central shared memory, or may participate in a memory hierarchy with both local and shared memory. Loosely-coupled multiprocessor systems (often referred to as clusters) are based on multiple standalone single or dual processor commodity computers interconnected via a high speed communication system. These systems have their individual processors and private memories. UMA (Uniform Memory Access) refers to a tightly coupled multiprocessor. NUMA (Non-Uniform Memory Access) implies a loosely coupled multiprocessor. In NORMA systems, each node is a standalone computer. When one computer needs memory data from another, software catches the remote memory access and must send an inter-process communication to request that data from the remote machine.

An interconnection network could be either static or dynamic. Connections in a static network are fixed links, while connections in a dynamic network are established on the fly as needed. There are several forms for interconnecting the networks, they are Bus based System, single-stage switch based system, Multistage Switching Based System and Crossbar -- switching System.

Cache coherence problem exist commonly in multiprocessor system with local caches because of the need to share writable data. All inter-processor coordination and synchronization is also accomplished via the global memory.

14.8 UNIT END QUESTIONS

1. Discuss the advantages and disadvantages of using the following interconnection networks in the design of a shared memory system.
   (a) Bus
   (b) Crossbar switch
   (c) Multistage network
2. Compare tightly coupled architectures with loosely coupled architectures.
3. What are UMA and NUMA? Explain.
4. What do you mean by cache coherence problem? Explain.
5. Explain the design issues related to multiprocessors operating systems.

□□□□

UNIT 15 : DATABASE OPERATING SYSTEM

Structure of the Unit

15.0 OBJECTIVES

After going through this unit student will be able to:

- To understand about transactions and its properties
- Learn about conflicts and concurrency control
- Learn about the inconsistent retrieval and update
- Learn about the serializability
- Learn about the recovery of database

15.1 INTRODUCTION

A database-management system (DBMS) is a collection of interrelated data and a set of programs to access those data. The collection of data, usually referred to as the database, contains information relevant to an organization. The primary goal of a DBMS is to provide a way to store and retrieve database information that is both convenient and efficient.

Basically a database system is designed to manage large number of information's. The database system should ensure the safety of information stored in the system as well safe enough from unauthorised access. The term transaction refers to a collection of operations that form a single logical unit of work. For

instance, transfer of money from one account to another is a transaction consisting of two updates, one to each account.

A database system is divided into subsystems that deal with each subsystem and has the responsibility of overall system. The basic function components of a database system are divided into storage manager and query processor manager.

The storage manager is so important because a database system requires very large space to store data. The main memory of an operating system cannot store the large information because of the data, so the data is stored on disks. Data is retrieved from disk to main memory or stored back from main memory to disk. The access time between main memory and to disk is relatively large, so it is important that the database system should structure the data in a way to minimize the need to move data between disk and main memory.

The query processor is important because it helps the database system simplify and facilitate access to data. Storage manager has a component called transaction manager, which ensures that the database remains in a consistent state and that concurrent transaction execution proceed without conflicting data.

15.2 TRANSACTION

A transaction is a unit of program execution that accesses and possibly updates various data items. Usually, a transaction is initiated by a user program written in a high-level data-manipulation language or programming language, where it is delimited by statements (or function calls) of the form begin transaction and end transaction. The transaction consists of all operations executed between the begin transaction and end transaction.

A transaction is a collection of operation that performs a single logical function. A transaction is a program unit whose execution may change the contents of a database. If the database was in a consistent state before a transaction, then on the completion of the execution of the program unit corresponding to the transaction, the database will be in a consistent state. The transaction consists of all operations executed between the begin transaction and end transaction. Once a transaction ends, the user may be notified of its success or failure and the changes made by the transaction are accessible. To ensure integrity of the data, we require that the database system must maintain the following properties of the transactions which are referred as ACID properties the acronym is derived from the first letter of each of the four properties.:-

i. **Atomicity** - In a database system, this property implies that it will run to completion as an indivisible unit. In the end of transaction, either no changes have occurred to the database or the database has been changed in a consistent manner.

ii. **Consistency** - Execution of a transaction with no other transaction executing concurrently preserves the consistency of the database i.e. if the database was in a consistent state before the start of a transaction, then on termination of a transaction the database will also be in a consistent state.

iii. **Isolation** - It indicates that actions performed by a transaction will be isolated or hidden from outside the transaction until the transaction terminates. It gives the relative independence to transaction i.e. even multiple transactions can execute concurrently, the database system guarantees that every transaction is unaware of other transactions executing concurrently is that system.

iv. **Durability** - It ensures that the commit action of a transaction, on its termination, will be reflected in the database i.e. in a database system the changes should be persisting, even if there are system failures.

Transaction access data by use of these two operations: -

- Read (A): - It transfers the data A from the database to the local buffer belonging to all transaction that executed the read operation.

- Write (A): - It transfers the data item A from the local buffer of the transaction that executed the write back to the database. It transfers the data A from main memory to the database. In database system, the write statement stores the value in temporary main memory and executes it on the disk.

In a real database system, the write operation does not necessarily result in the immediate update of the data on the disk; the write operation may be temporarily stored in memory and executed on the disk later.

15.2.1 Transaction State

In the absence of failures, all transactions complete successfully. However, a transaction may not always complete its execution successfully. Such a transaction is termed aborted. If we are to ensure the atomicity property, an aborted transaction must have no effect on the state of the database. Therefore, any changes that the aborted transaction made to the database must be undone. It is called Rollback. A transaction that completes its execution successfully is said to be committed. Once a transaction has committed, we cannot undo its effects by aborting it.

A transaction executes in various states these states will occur on every transaction and it is not necessary for the transaction that occur all these state. These states are as follows: -

i. **Active State** - It is the initial state for every transaction while it is executing.

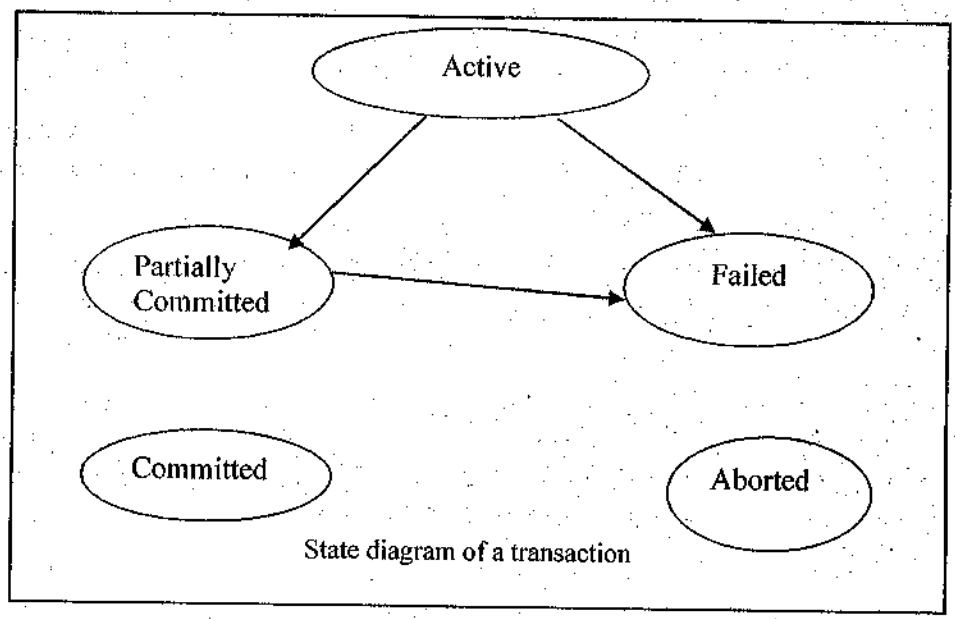
ii. **Partially Committed State** - When the final operation has been executed, the transaction is in partial commited, but the operation can be changed in this state.

iii. **Failed** - If the execution is no longer exists and can't perform the execution of this operation then the transaction will be in fail state.

iv. **Aborted** - This state is occurred it the transaction is rolled back and the database system has restored its active state again.

v. **Committed** - After the successful execution of the operation, a transaction will be in commited state.

The following figure shows the state of the transaction:-



State diagram of a transaction

A transaction starts in the active state. After completion of the final operations, it will be in the partially commited, but in this state the result still reside in the main memory, so it is possible to transaction to go into commited or in aborted state, if the failure of the transaction then the abort state of transaction

occurs. After the completion of result into the disk from main memory, transaction is in committed state.

A transaction is in the fail state after the system determines the transaction can no longer proceed. Such transaction must be rolled back. Then it enters in the aborted state. At this time, system has two options.

- Restart the Transaction
- Kill the transaction.

15.2.2 Transaction Processing

Transaction processing system normally allows multiple transactions to run concurrently allowing multiple transactions to update data concurrently which created several complications with consistency of the data.

The main purpose of using concurrent execution in the database system is to achieve the multi programming in an operating system. The database consistency can be reduced when the concurrent executions performed in a transaction processing. The database system must control the interaction among the concurrent transaction to prevent them from inconsistency of database.

Consider a multiple transaction-processing example. Let T1 and T2 are two transactions that transfer value x from one processing to another as: -

```
T1:  Read (X)
     X =X -10
     Write (X)
     Read (Y)
     Y=Y+10
     Write (Y)
```

Now transaction T2 performs another operation on x & y as:

```
T2:  Read (X)
     Z=X * 0.5
     X = X-Z
     Write (X)
     Read (Y)
     Y=Y+Z
     Write (Y)
```

15.3 CONCURRENCY CONTROL

Transaction-processing systems usually allow multiple transactions to run concurrently. Allowing multiple transactions to update data concurrently causes several complications with consistency of the data. When many transactions execute concurrently in a database system, the isolation property is no longer be preserved. Concurrency control refers to the co-ordination of execution of multiple transactions in a multi user database environment.

There are two reasons for allowing concurrency -

- **Improved throughput and resource utilization** - A transaction consists of many steps. Some involve I/O activity; others involve CPU activity. I/O activity can be done in parallel with processing at the CPU. The parallelism of the CPU and the I/O system can therefore be exploited to run multiple transactions in parallel. All of this increases the throughput of the system—that is, the number of transactions executed in a given amount of time. Correspondingly, the processor and disk utilization also increases, in other words, the processor and disk spend less time idle, or not performing any useful work.

- **Reduced waiting time** - There may be a mix of transactions running on a system, i.e. some are short and some are long. If transactions are running serially, a short transaction may have to wait for a preceding long transaction to complete, which can lead to delays in running a transaction. If the transactions are operating on different sections of the database, it is better to let them run concurrently, sharing the CPU cycles and disk accesses between them. Concurrent execution reduces the unpredictable delays in running transactions. Moreover, it also reduces the average response time, the average time for a transaction to be completed after it has been submitted.

The database system must control the interaction among the concurrent transactions to prevent them from destroying the consistency of the database. It does so through a variety of mechanisms called concurrency-control schemes.

The system must control the interaction among the concurrent transactions; this control is achieved through one of a variety of mechanisms called concurrency-control schemes. A concurrent execution has a problem called inconsistent retrieval.

Inconsistent Retrieval - This type of problem occurs when a transaction calculate an aggregate function or summary function (e.g. Sum) over a set of data, which the other transactions are updating, the inconsistency happens because the transaction may read some data before they are changed and read other data after they are changed. This problem can be solved out by two mechanisms –

- Locking - If one user is updating the data, all other user denied access to the same data
- Time Stamping - A unique global time stamp is assigned to each transaction.

15.3.1 Locking

In the concurrent execution, do one way is to ensure that data items be accessed in a mutually exclusive manner, that is while one transaction is accessing a data item, no other transaction can modify that data item. The most common method to this requirement is to allow a Lock-Bases protocol on that item. These are various modes in which a data may be locked two of these methods are: -

(a) Shared (S) Mode: - Data items can only be read, if a transaction T occurred, but cannot write, S-Lock is requested using Lock-S instruction.

(b) Exclusive (x) Mode: - In this data items in a transaction T can be both read and written x-lock is requested using Lock-x instruction.

	S	X
S	True	False
X	False	False

Lock Compatibility Matrix

By giving a set of lock mode a compatibility function will define on every data item.

For example - Let A and B is the two lock modes. Suppose that a transaction T1 request a lock of mode

A on the data item Q on which the transaction T2 currently holds a lock of mode B, then A is compatible with mode B. this function is represented by matrix in fig (1).

The shared modes can compatible with shared mode, but not with exclusive mode. Shared modes locks can be performed simultaneously any time on a data item but on exclusive mode lock request has to wait until the currently held shared mode locks are released.

To access a data item, transaction T1 must first lock on them. If the data is locked by another transaction already, in an incompatible mode, then concurrency control manager will not permit to lock the transaction T1, until transaction T2 is not unlocked.

T1	T2
Lock X(B)	
Read (B)	
B=B-10	Lock-S (A)
Write (B)	Read (A)
Unlock (B)	Unlock (A)
Lock –X(A)	Lock –S (B)
Read (A)	Read (B)
A=A+10	Unlock (B)
Write (A)	Display (A+B)
Unlock (A)	

15.3.2 Time-Stamping

With every transaction T1 in the system, we add a fixed time stamp TS (T1). The database system assigns this timestamp before the transaction T1 starts execution. If a time stamp TS (T1) is assigned and a new transaction T2 wants to execute, then TS (T1) should be less then TS(T2). To implement this scheme, there are two simple methods: -

i. Use the value of the clock of system i.e., a transaction's timestamp is equal to the value of the clock when the transaction is to be executed.

ii. Use a logical counter, which is incremented after a new timestamp has been assigned, i.e. a transaction's timestamp is equal to the value of the counter when the transaction is to be executed.

There are two types of time stamp values: -

(a) W-timestamp (Q): - It shows the largest timestamp of any transaction that executed write (Q).

(b) R-Timestamp: - It shows the largest timestamp of any transaction that executed read (Q).

Timestamp-Ordering Protocols: - These protocols tell that any conflicting read and write operation are executed in Timestamp order. The operation of this protocol is as: -

1. Let us assume that T1 issues Read (Q)

(a) If TS (T1) is less than W-Timestamp (Q) then T1 needs to read a value of Q, which is already overwritten. Hence, the read operation cannot be performed and T1 will rollback.

(b). If TRANSACTION (T1) $\geq$ W-Timestamp (Q), then the read operation is executed, and R-Timestamp (Q) is set to maximum of R-Timestamp (Q) and TS (T1).

2. Assure that T1 issues write (Q): -

(a) If TS (T1) < R-Timestamp (Q), then the value of Q that T1 is producing was needed previously, and the system assured that the value would never be produced, so the system will not execute write operation and rolled back.

(b) If TS (T1) < W-Timestamp (Q), then T1 is attempting to write an obsolete value of Q. so, the system rejects the write operation and T1 will roll back.

(c) Otherwise, the system executes the write operation and sets w-Timestamp (Q) to TS (T1).

15.4 Serializabilty

A database system must control concurrent execution of transaction, to ensure that the database state remains consistent, consider a system with two data items X & Y, that are both read an written by two transaction T1 and T2 as follows:

T1	T2
Read (X)	
Write (X)	
Read (Y)	
Write (Y)	
	Read (X)
	Write (X)
	Read (Y)
	Write (Y)

A serial schedule

This execution sequence is called a serial schedule, each serial schedule consists of a sequence of instructions from various transactions where the instruction belongs to one single transaction appear together in that schedule.

There are two types of serializability: -Conflict Serializability and View Serializability

15.4.1 Conflict Serializability

Let us assume that in a schedules S, there are two instruction I1 and I2, of transaction T1 and T2 have different data items, then the data value can be swapped into I1 and I2 with unchanged the results of any instruction in S. but if I1 and I2 have same data item Q, then the order of the two swaps may matter. These are four cases can be performed: -

(a) I1 = Read (Q), I2 = Read (Q). The order of I1 and I2 doesn't matter since the same value of Q is read by T1 and T2.

(b) I1 = Read (Q), I2 = write (Q). If I1 comes before I2, then T1 will not read the value of Q that is written by T2 in instruction I2. If I2 comes before I1, then T1 reads the value of Q that is written by T2. In this case, the order must matter.

(c) I1 = write (Q), I2 = Read (Q). In this case the order matters as in the case 2.

(d) I1 = Write (Q), I2 = Write (Q). In this case the order of both instruction doesn't matter. However, the value obtained by the next Read (A) instruction is affected, since the result of only the latter of two write instructions is stored in the database. If these are no other write (Q) instruction after I1 and I2 in schedule S. then the order of I1 and I2 directly affects the final value of Q in the database state that results from schedule S.

We can say now, if one of these instructions is a write operation, and they are operation by different transactions on the same data item, then I1 and I2 conflicts.

This is shown in the following figure as: -

T1	T2
Read (X) Write (X)	
	Read (X) Write (X)
Read (Y) Write (Y)	
	Read (Y) Write (Y)

Schedule S1 = Conflict Serializability

In this figure, the write (X) instruction of T1 is conflicts with the Read (X) of T2. But write (X) of T2 does not conflict withheld Read (Y) instruction of T1 because these two instructions have different data items.

Since the write (X) instruction of T2 doesn't conflict with the Read (Y) instruction of T1, we can swap theses instruction to generate an equivalent schedule S2 we continue to swamp no conflicting instruction as: -

- Swap the Read (X) instruction of T1 with Read (X) instruction of T2.
- Swap the write (Y) instruction T1 with write (XP instruction of T2.
- Swap the write (Y) instruction of T2 with the Read (X) instruction of T2

In this mechanism, the final state of both schedules S1 and S2 will be same

T1	T2
Read (X) Write (X) Read (Y) Write (Y)	
	Read (X) Write (X) Read (Y) Write (Y)

Schedule S2 = A serial Schedule equivalent to S1

This concept of conflict equivalent leads to the concept of conflict- serializability. We can say that a schedule S is conflict serializable if it is conflict equivalent to a serial schedule.

15.4.2 View Serializabilty

Let us assume that two schedules S1 and S2 are view equivalent if they fulfill these three conditions: -

- For each data item Q, if transaction T1 reads the initial value of Q in schedule S1, then in schedule S2, transaction T1 also read the initial value of Q.
- For each data items Q, if T1 execute Read (Q) instruction in S1 and if this value was generated by a Write (Q) operation executed by transaction T2, then the Read (Q) operation of T1 in S2, also read the value of Q that was produced by the same write (Q) operation of T2.
- For each data item Q, the transaction that performs the final Write (Q) operation in schedule S2.

By these three conditions, we can say that a schedule S1 is view serializable if it is view equivalent to the serial schedule.

Every conflict serializable schedule is also view serializable, but it not necessary that view serializable schedule is conflict serializable.

15.5 RECOVERY

An integral part of a database system is a recovery scheme that can restore the database to the consistent state that existed before the failure. The recovery scheme must also provide high availability; that is, it must minimize the time for which the database is not usable after a crash.

There are various types of failure that may occur in a system, each of which needs to be dealt with in a different manner. The simplest type of failure is one that does not result in the loss of information in the system. There are two types of errors that may cause a transaction to fail.

A. Transaction Failure

- Logical error. The transaction can no longer continue with its normal execution due to some internal condition, such as wrong input, data unavailability, overflow, or resource limit exceeded.
- System error. The system has entered an undesirable state (i.e. deadlock), as a result of which a transaction cannot continue with its normal execution. The transaction, however, can be re-executed at later stage.

B. System crash.

There is a hardware malfunction, or a bug in the database software or the operating system, that causes the loss of the contents of volatile storage, and brings transaction processing to a halt. The content of nonvolatile storage remains intact, and is not corrupted.

To determine how the system should recover from failures, we need to identify the failure modes of those devices used for storing data. There are various types of recovery schemes available for the database systems.

15.5.1 Log-Based Recovery

The widely used structure for recording database modifications is the log. The log is a sequence of log records, recording all the update activities in the database. There are several types of log records. An update log record describes a single database write. It has the following fields:

- **Transaction identifier** is the unique identifier of the transaction that performed the write operation.

- **Data-item identifier** is the unique identifier of the data item written. Typically, it is the location on disk of the data item.
- **Old value** is the value of the data item prior to the write.
- **New value** is the value that the data item will have after the write.

A log record can be one of the following entries:-

- < Start, T> Transaction T is started.
- < Write, T, A , Old-Value, new value > Indicates a write operation in T which changes the value of data items X from old value to new value.
- < Read, T, B > T read the data item X
- < Commit, T > T is commited.
- < Abort, T> T is aborted.

Whenever a transaction performs a write, it is necessary that the record for that write be created before the database is modified. Once a log record exists database if that is desirable. Also, we have the rights to undo a modification that has already been it by using the old value field in log records. For log records to be useful for recovery from system and disk failures, the log must reside in stable storage.

Log Equivalence:-

Let us assume for transaction T1 and T2

T1	< Start, T1>
	<Read, X>
	< Read, Y>
	<Write, X>
	<Commit, T1>
T2	< Start, T2>
	<Read, X>
	<Write, X>
	<Commit, T2>

Two logs over T are equivalent if they represent the same computation that is in both logs each transaction reads the value written by the same transaction and they have the same final write operations.

To consider the initial values of data items, logs are augmented by an initial transaction Ts, which initially writes all the data items and to consider the effect of find writes, logs are augmented by a final transaction Tf which finally reads all the data items.

Log1 = Ws [X] Ws [Y] W [X] R2 [X] W1 [Y] – [W2Y] Rf [X] Rf [Y]

And Log 2 = W0 [X] W1 [Y] R2 [X] W2 [Y]

Here W0 [X] and W2 [Y] are the final writes.

The log1 and Log2 are the equivalent logs as they have the same read from relation when they augmented by initial and final transactions.

A log is called serializable log if there exists and equivalent serial log.

15.5.2 Recovery using Undo logs

The first task of the recovery manager is to divide the transactions into committed and uncommitted transactions. If there is a log record <COMMIT T>, then by undo rule all changes made by transaction T were previously written to disk. Thus, T by itself could not have left the database in an inconsistent state when the system failure occurred.

However, suppose that we find a <START T> record on the log but no <COMMIT T> record. Then there could have been some changes to the database made by T that got written to disk before the crash, while other changes by T either were not made, even in the main-memory buffers, or were made in the buffers but not copied to disk. In this case, T is an incomplete transaction and must be undone.

15.5.3 Check Points

Log based recovery faces the following difficulties in recovery –

- The search process is time consuming.
- Most of the transactions that, according to our algorithm, need to be redone have already written their updates into the database. Although redoing them will cause no harm, it will nevertheless cause recovery to take longer.

To reduce these types of overhead, checkpoints are used. During execution, the system maintains the log. In addition, the system periodically performs checkpoints, which require the following sequence of actions to take place:

- Output onto stable storage all log records currently residing in main memory.
- Output to the disk all modified buffer blocks.
- Output onto stable storage a log record <checkpoint>.

Transactions are not allowed to perform any update actions, such as writing to a buffer block or writing a log record, while a checkpoint is in progress. The presence of a <checkpoint> record in the log allows the system to streamline its recovery procedure.

15.5.4 Shadow Paging

An alternative to log-based crash-recovery techniques is shadow paging. Under certain circumstances, shadow paging may require fewer disk accesses than do the log-based methods. There are, however, disadvantages to the shadow-paging approach, as well, that limit its use. It is hard to extend shadow paging to allow multiple transactions to execute concurrently.

The database is partitioned into some number of fixed-length blocks, which are referred to as pages. The term page is borrowed from operating systems, since we are using a paging scheme for memory management. The page table has n entries — one for each database page. Each entry contains a pointer to a page on disk. The first entry contains a pointer to the first page of the database, the second entry points to the second page, and so on.

The key idea behind the shadow-paging technique is to maintain two page tables during the life of a transaction: the current page table and the shadow page table. When the transaction starts, both page tables are identical. The shadow page table is never changed over the duration of the transaction. The current page table may be changed when a transaction performs a write operation. All input and output operations use the current page table to locate database pages on disk.

The shadow-page approach to recovery is to store the shadow page table in nonvolatile storage,so

that the state of the database prior to the execution of the transaction can be recovered in the event of a crash, or transaction abort.

15.6 SUMMARY

A database system is divided into subsystems that deal with each subsystem and has the responsibility of overall system. The basic function components of a database system are divided into storage manager and query processor manager. A transaction is a unit of program execution that accesses and possibly updates various data items. If the database was in a consistent state before a transaction, then on the completion of the execution of the program unit corresponding to the transaction, the database will be in a consistent state. The transaction must follow the ACID properties.

A transaction executes in various states these states will occur on every transaction and it is not necessary for the transaction that occur all these state. These states are Active State, Partially Committed State, Failed, Aborted and Committed.

Transaction processing system normally allows multiple transactions to run concurrently allowing multiple transactions to update data concurrently which created several complications with consistency of the data. The main purpose of using concurrent execution in the database system is to achieve the multi programming in an operating system. Concurrency control refers to the co-ordination of execution of multiple transactions in a multi user database environment. Concurrency problem of transactions are handled by locking and time-stamping.

An integral part of a database system is a recovery scheme that can restore the database to the consistent state that existed before the failure. The recovery scheme must also provide high availability; that is, it must minimize the time for which the database is not usable after a crash.

15.7 UNIT END QUESTIONS

1. What are the ACID properties? Explain the usefulness of each.
2. During its execution, a transaction passes through several states, until it finally commits or aborts. What are the all possible sequences of states through which a transaction may pass?
3. What do you mean by concurrent transactions? Explain.
4. Explain the purpose of the checkpoint mechanism. How often should checkpoints be performed?
5. Compare the shadow-paging recovery scheme with the log-based recovery schemes.
6. What do you mean by inconsistent update? Explain.
7. What is serializability? Explain.

□□□□