```
        int other;
        other = 1 - process;
        interested[process] = TRUE;
        turn = process;
        while ( turn == process && interested[other] == TRUE ) ;
    }
    void leave_region(int process) {
        interested[process] = FALSE;
    }
```

This code actually takes advantage of a race condition. When 2 processes try to enter simultaneously, setting turn to the entering pid releases the other interested process from the while loop. It's a clever trick. Dekker's algorithm was published in 1965, Peterson's in 1981.Both algorithms can be generalized to multiple processes, although that's beyond the scope of the class. Perhaps the most amazing thing about software-only mutual exclusion is that it can be done at all in the challenging environment of multitasking.

4.5 HARDWARE SUPPORT FOR MUTUAL EXCLUSION

One way to enforce mutual exclusion is to turn off interrupts while in the critical section. Some OS, for example, AmigaDOS actually allowed processes to do this. It's a bad idea for several reasons:

- It badly breaks process isolation - a process can prevent other processes from hearing from the disk or timers.
- Turning off interrupts for any period of time is extremely hazardous. The OS needs to field interrupts to keep the machine running, and frequently will be so confused after interrupts are off for a while that it will reboot anyway. (Most hardware will send an NMI (non maskable interrupt) after too long with interrupts disabled).

A more reasonable method of hardware assistance is the *test-and-set instruction,* which makes testing a flag and resetting its value an *atomic* action. The given memory location is set to the new value and its old value returned in a register without the opportunity for a context switch. This allows a critical section to be coded as:

```
    int flag;
    void enter_region(int process) {
        int my_flag= test_and_set(flag);
        while ( my_flag == 1 )
         my_flag = test_and_set(flag);
    }
    void leave_region(int process) {
        flag = 0;
    }
```

Because the test and the resetting of the flag is atomic, this system simplifies mutual exclusion by changing the rules - the Wrath of Khan approach.

Hardware supported mutual exclusion by

- Interrupt disabling
- Special instructions

4.5.1 Interrupt Disabling

In a single processor system, a process runs until it invokes an operating system service or until it is interrupted. So disabling interrupts can guarantee mutual exclusion

```
while( true ){
< disable interrupts >;
< critical section >;
< enable interrupts >;
< remainder >;
}
```

Problems with interrupt disabling

- Delays response to external events
- Critical sections must be very short
- Approach does not work with multiprocessing systems
- Disabling interrupts on one processor will not guarantee mutual exclusion

4.5.2 Special Instructions: TestAndSet

The TestAndSet instruction tests a memory location for 0. If the location contains 0 . . .

- The location is loaded with 1
- TestAndSet returns **true**

Otherwise, the location is unaltered and **false** is returned. The TestAndSet instruction is "atomic". VAX system has BBCS (Branch on Bit Clear and Set bit) instruction together with others. Each process follows the same protocol

```
< program mutualexclusion >
const int n = < number of processes >;
int bolt;
void P(int i) {
while ( true) {
while ( ! testset ( bolt ) )
< do nothing >;
< critical section >;
```

```
bolt = 0;
< remainder >;
     }
}
void main( ){
bolt = 0;
parbegin (P(1), P(2), . . . ,P(n));
}
```

- Any number of processes and processors are allowed
- One "bolt" for each critical resource
- Busy waiting is a disadvantage

4.5.3 Operating System Support

There are three major communication scenarios:

1. One-way Communication usually do not need synchronization.
2. Client/server communication is for multiple clients making service request to a shared server. If co-ordination is required among the clients, it is handled by the server and there is no explicit interaction among client process.
3. Interprocess communication:

- Not limited to making service requests.
- Processes need to exchange information to reach some conclusion about the system or some agreement among the cooperating processes.

These activities require peer communication; there is no shared object or centralized controller.

4.6 SEMAPHORES IMPLEMENTATION

Semaphore is a software concurrency control tool. It bears analogy to old Roman system of message transmission using flags. It enforces synchronization among communicating processes and does not require busy waiting.

One way to implement semaphores in computers is to use a flag (i.e., a bit of memory) that can be tested and set to either 0 or 1 in a single machine operation. Because both the test and set actions occur in the same machine instruction, this operation is indivisible and cannot be interrupted by another process which is running on the machine. By placing test-and-set operations around the critical section of a program, programmers can guarantee mutually exclusive access to the critical resource.

Another way to implement a semaphore is to use a count rather than just two values. Such semaphores are called counting semaphores in contrast to the binary semaphore presented above. Counting semaphores are used to solve synchronization problems such as the Bounded Buffer problem.

Technically a semaphore is an integral variable (say S) with two operations defined on it:

- Wait()
- Signal()

One implementation of these functions is presented below.

```
Wait (S)
{
while S <= 0;       /* no-operation ..wait*/
S–;
}
Signal (S)
{
S++;
}
```

There are two varieties of semaphores:

- Counting semaphore where integer value (S) can range over an unrestricted domain
- Binary semaphore where integer value (S) can range only between 0 and 1

Incidently they are also called mutexlocks. It provides mutual exclusion with busy waiting as shown below.

```
S=1;    /* initialized to 1 */
Wait (S);
Critical Section
Signal (S);
```

Note that it must guarantee that no two processes can execute wait ()and signal ()on the same semaphore at the same time. Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section. It could now have busy waiting in critical section implementation. Implementation code is short. When applications spend lots of time in critical sections this does not form a very good solution.

Semaphores can be implemented without busy waiting if each semaphore is associated with a waiting queue. Each entry in a waiting queue has two data items: value (of type integer) and a pointer to next record in the list. Besides, it also contains two operations:

Block – to place the process invoking the operation on the appropriate waiting queue

Wakeup – to remove one of processes in the waiting queue and place it in the ready queue.

The corresponding implementations of the same are presented below.

```
Wait (S)
{
value–;
if (value < 0)
```

```
{
/*add this process to waiting queue*/
block();
}
}
Signal (S)
{
value++;
if (value <= 0)
{
/*remove a this process (say P) from the waiting queue*/
wakeup(P);
}
}
```

Check Your Progress

Fill in the blanks:

1. The processes can execute in two different modes - sequential and ....
2. Deadlocks can be expressed more clearly using a directed graph, called a system resource-allocation graph or ....
3. Necessary conditions for deadlock to occur.

4.7 LET US SUM UP

In a multi-programming environment, more than one process exist in the system competing for the resources. Based on criteria and algorithms employed by the system, one of them is selected at a time, is granted requisite resources and is executed while other candidate processes wait for their turn. There are a number of different process co-ordination problems arising in practical situations that exemplify important associated issues. These problems also provide a base for solution testing for process co-ordination problems. In this section, we will see some of such classical process co-ordination problems.

Semaphore is a software concurrency control tool. It bears analogy to old Roman system of message transmission using flags. It enforces synchronization among communicating processes and does not require busy waiting.

4.8 KEYWORDS

Semaphore: It is a software concurrency control tool.

Progress: If no process is executing in its critical section and there exist some processes that wish to enter their critical sections, then only those processes that are not executing in their remainder section

can participate in the decision of which will enter its critical section next, and this selection cannot be postponed indefinitely.

Bounded Waiting: There exists a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

4.9 QUESTIONS FOR DISCUSSION

1. Explain the various principles of concurrency.
2. How software and hardware solutions support mutual exclusion?
3. Explain the concept of semaphores with its implementation.

Check Your Progress: Model Answers

1. Concurrent
2. Precedence Graph
3. Mutual Exclusion, hold and wait, no preemption, circular wait.

4.10 SUGGESTED READINGS

Andrew S. Tanenbaum, *Modern Operating System,*: Prentice Hall

Silberschatz Galvin, *Operating System Concepts,*: Addison Wesley

Andrew M. Lister, *Fundamentals of Operating Systems,*: Wiley

Colin Ritchie, *Operating Systems,*: BPB Publications

LESSON

5

DEADLOCK

CONTENTS

5.0 Aims and Objectives

5.1 Introduction

5.2 Basic Concepts of Deadlock

5.3 Deadlock Prevention

5.3.1 Linear Ordering of Resources

5.3.2 Hierarchical Ordering of Resources

5.4 Deadlock Detection and Avoidance

5.4.1 Recovery from Deadlock

5.5 Let us Sum up

5.6 Keywords

5.7 Questions for Discussion

5.8 Suggested Readings

5.0 AIMS AND OBJECTIVE

After studying this lesson, you should be able to:

- Understand the damage caused by a deadlock situation
- Understand the conditions that can potentially lead a system to deadlock state
- Explain the resource allocation graph to handle deadlocks
- Concept of the strategies of deadlock handling
- The ways and means of deadlock prevention, avoidance, detection and recovery

5.1 INTRODUCTION

A deadlock is a situation wherein two or more competing actions are waiting for the other to finish, and thus neither ever does. It is often seen in a paradox like 'the chicken or the egg'.

Deadlock occurs when we have a set of processes [not necessarily all the processes in the system], each holding some resources, each requesting some resources, and none of them is able to obtain what it needs, i.e. to make progress.

We will usually reason in terms of resources R1, R2, .., Rm and processes P1, P2, .., Pn. A process Pi that is waiting for some currently unavailable resource is said to be blocked.

This situation may be likened to two people who are drawing diagrams, with only one pencil and one ruler between them. If one person takes the pencil and the other takes the ruler, a deadlock occurs when the person with the pencil needs the ruler and the person with the ruler needs the pencil, before he can give up the ruler. Both requests can't be satisfied, so a deadlock occurs.

In this unit we will discuss about the deadlock.

5.2 BASIC CONCEPTS OF DEADLOCK

Recall that one definition of an operating system is a resource allocator. There are many resources that can be allocated to only one process at a time, and we have seen several operating system features that allow this, such as mutexes, semaphores or file locks.

Sometimes a process has to reserve more than one resource. For example, a process which copies files from one tape to another generally requires two tape drives. A process which deals with databases may need to lock multiple records in a database.

In general, resources allocated to a process are not preemptable; this means that once a resource has been allocated to a process, there is no simple mechanism by which the system can take the resource back from the process unless the process voluntarily gives it up or the system administrator kills the process. This can lead to a situation called deadlock. A situation when two more processes are unable to proceed because they are waiting for each other to do something.

A common example is a program communicating to a server, which may find itself waiting for output from the server before sending anything more to it, while the server is similarly waiting for more input from the controlling program before outputting anything. (It is reported that this particular flavor of deadlock is sometimes called a starvation deadlock, though the term starvation is more properly used for situations where a program can never run simply because it never gets high enough priority.

Another common flavor is constipation, in which each process is trying to send stuff to the other but all buffers are full because nobody is reading anything.

A set of processes or threads is deadlocked when each process or thread is waiting for a resource to be freed which is controlled by another process. Here is an example of a situation where deadlock can occur.

```
Mutex M1, M2;
/* Thread 1 */
while (1) {
  NonCriticalSection()
  Mutex_lock(&M1);
  Mutex_lock(&M2);
  CriticalSection();
  Mutex_unlock(&M2);
```

```
  Mutex_unlock(&M1);
}

/* Thread 2 */
while (1) {
NonCriticalSection()
  Mutex_lock(&M2);
  Mutex_lock(&M1);
  CriticalSection();
  Mutex_unlock(&M1);
  Mutex_unlock(&M2);
}
```

Suppose thread 1 is running and locks M1, but before it can lock M2, it is interrupted. Thread 2 starts running; it locks M2, when it tries to obtain and lock M1, it is blocked because M1 is already locked (by thread 1).

Eventually thread 1 starts running again, and it tries to obtain and lock M2, but it is blocked because M2 is already locked by thread 2. Both threads are blocked; each is waiting for an event which will never occur.

Traffic gridlock is an everyday example of a deadlock situation.

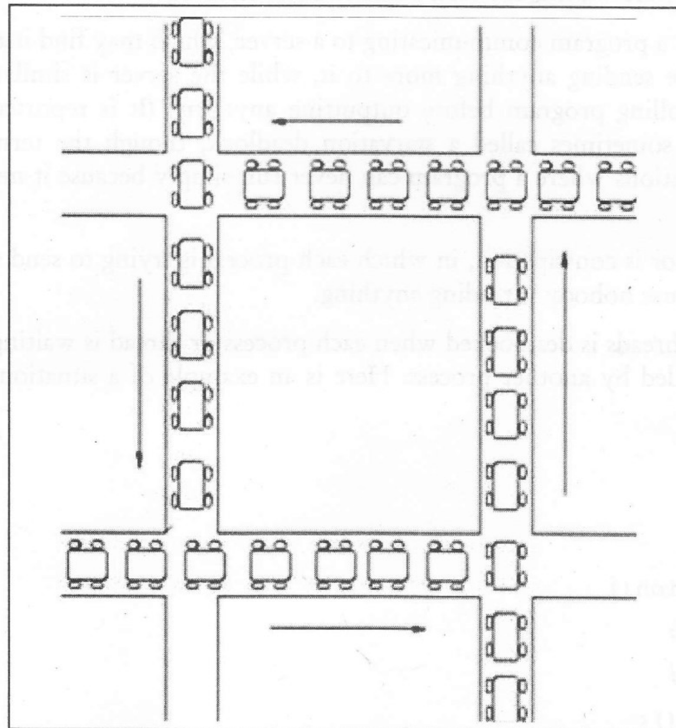

Figure 5.1: Every day Example of Deadlock Situation

In order for deadlock to occur, four conditions must be true.

- *Mutual exclusion:* Each resource is either currently allocated to exactly one process or it is available. (Two processes cannot simultaneously control the same resource or be in their critical section).
- *Hold and Wait:* processes currently holding resources can request new resources
- *No preemption:* Once a process holds a resource, it cannot be taken by another process or the kernel.
- *Circular wait:* Each process is waiting to obtain a resource which is held by another process.

The dining philosopher's problem discussed in an earlier section is a classic example of deadlock. Each philosopher picks up his or her left fork and waits for the right fork to become available, but it never does.

Deadlock can be modeled with a directed graph. In a deadlock graph, vertices represent either processes (circles) or resources (squares).

A process which has acquired a resource is show with an arrow (edge) from the resource to the process.

A process which has requested a resource which has not yet been assigned to it is modeled with an arrow from the process to the resource. If these create a cycle, there is deadlock. The deadlock situation in the above code can be modeled like this.

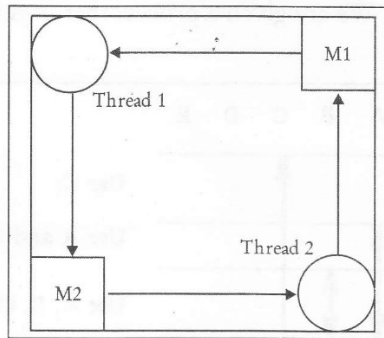

Figure 5.2 Deadlock Situation Model

This graph shows an extremely simple deadlock situation, but it is also possible for a more complex situation to create deadlock. Here is an example of deadlock with four processes and four resources.

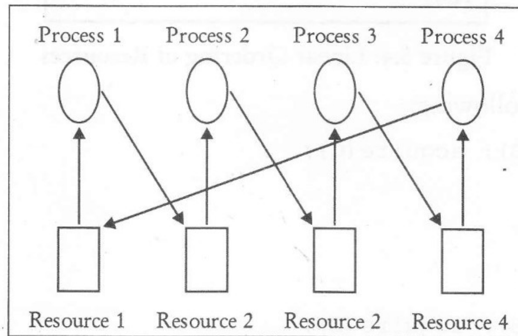

Figure 5.3: Example of Deadlock with four Processes and four Resources

There are a number of ways that deadlock can occur in an operating situation. We have seen some examples, here are two more.

- Two processes need to lock two files, the first process locks one file the second process locks the other, and each waits for the other to free up the locked file.
- Two processes want to write a file to a print spool area at the same time and both start writing. However, the print spool area is of fixed size, and it fills up before either process finishes writing its file, so both wait for more space to become available.

5.3 DEADLOCK PREVENTION

Deadlock Prevention is to use resources in such a way that we cannot get into deadlocks. In real life we may decide that left turns are too dangerous, so we only do right turns. It takes longer to get there but it works. In terms of deadlocks, we may constrain our use of resources so that we do not have to worry about deadlocks. Here we explore this idea with two examples.

5.3.1 Linear Ordering of Resources

Assume that all resources are totally ordered from 1 to r. We may impose the following constraint:

It is easy to see that with this rule we will not get into deadlocks. [Proof of contradiction]. Here is an example of how we apply this rule. We are given a process that uses resources ordered as A, B, C, D, E in the following manner:

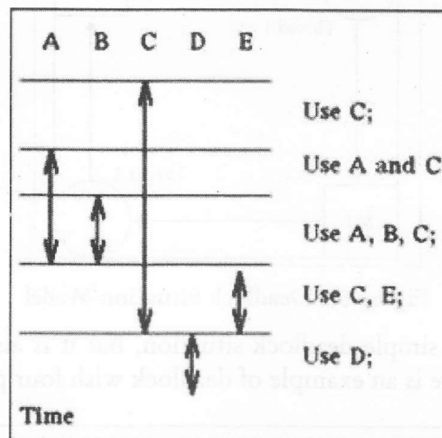

Figure 5.4: Linear Ordering of Resources

Then the process can do the following:

```
acquire(A); acquire(B); acquire(C);
use C
use A and C
use A, B, C
release(A); release(B); acquire(E);
use C and E
```

```
release(C); release(E); acquire(D);
use D
release(D);
```

A strategy such as this can be used when we have a few resources. It is easy to apply and does not reduce the degree of concurrency too much.

5.3.2 Hierarchical Ordering of Resources

Another strategy we may use in the case that resources are hierarchically structured is to lock them in hierarchical order. We assume that the resources are organized in a tree (or a forest) representing containment.

We can lock any node or group of nodes in the tree. The resources we are interested in are nodes in the tree, usually leaves. Then the following rule will guarantee avoidance of deadlocks.

Here is an example of use of this rule, locking a single resource at a time.

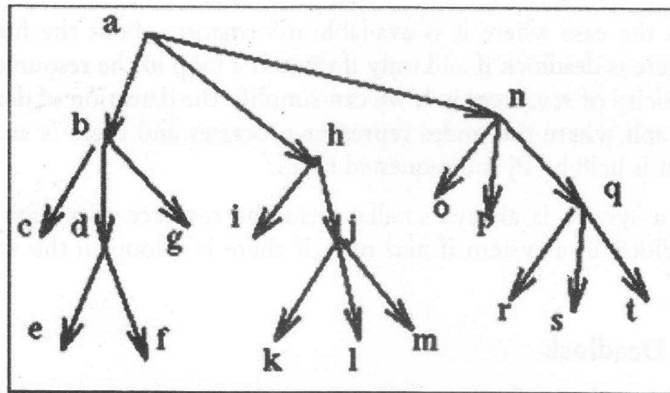

Figure 5.5: Hierarchical Ordering of Resources

Then if a process wants to use the resources e, f, i, k it uses in sequence the commands:

lock(a); lock(b); lock(h); unlock(a); lock(d); unlock(b); lock(i); lock(j); unlock(h); lock(k); unlock(j); lock(e); lock(f); unlock(d);

Of course by forcing all locking sequences to start at the root of the resource hierarchy we create a bottleneck, since now the root becomes a scarce resource.

As always there is trade-offs to be made. An obvious improvement on this prevention policy is to start not by locking the root, but by instead locking the lowest node subsuming all the nodes used by the current activity.

For example if we had an activity that accessed only the nodes p, r, and s then we could start by locking n, not a.

5.4 DEADLOCK AVOIDANCE AND DETECTION

Deadlock Avoidance, assuming that we are in a safe state (i.e. a state from which there is a sequence of allocations and releases of resources that allows all processes to terminate) and we are requested certain

resources, simulates the allocation of those resources and determines if the resultant state is safe. If it is safe the request is satisfied, otherwise it is delayed until it becomes safe.

The Banker's Algorithm is used to determine if a request can be satisfied. It uses requires knowledge of who are the competing transactions and what are their resource needs.

The Banker's algorithm is run by the operating system whenever a process requests resources.

The algorithm prevents deadlock by denying or postponing the request if it determines that accepting the request could put the system in an unsafe state (one where deadlock could occur).

For the Banker's algorithm to work, it needs to know three things:

- How much of each resource each process could possibly request
- How much of each resource each process is currently holding
- How much of each resource the system has available

Some of the resources that are tracked in real systems are memory, semaphores and interface access.

Deadlock Detection, in the case where it is available information about the full resource allocation graph, it is easy since there is deadlock if and only if there is a loop in the resource allocation graph. In the case that the multiplicity of resources is 1, we can simplify the detection of deadlocks by building a wait-for graph, i.e. a graph where the nodes represent processes and there is an arc from Pi to Pj if there is a resource R that is held by Pj and requested by Pi.

The wait-for graph of a system is always smaller than the resource allocation graph of that same system. There is a deadlock in a system if and only if there is a loop in the wait-for graph of that system.

5.4.1 Recovery from Deadlock

Once a deadlock has been detected, one of the 4 conditions must be invalidated to remove the deadlock. Generally, the system acts to remove the circular wait, because making a system suddenly preemptive with respect to resources, or making a resource suddenly sharable is usually impractical. Because resources are generally not dynamic, the easiest way to break such a cycle is to terminate a process.

Usually the process is chosen at random, but if more is known about the processes, that information can be used. For example the largest or smallest process can be disabled. Or the one waiting for longest. Such discrimination is based on assumptions about the system workload.

Some systems facilitate deadlock recovery by implementing check pointing and rollback. Check pointing is saving enough state of a process so that the process can be restarted at the point in the computation where the checkpoint was taken. Auto saving file edits is a form of check pointing. Check pointing costs depend on the underlying algorithm. Very simple algorithms (like linear primarily testing) can be check pointed with a few words of data. More complicated processes may have to save all the process state and memory.

Checkpoints are taken less frequently than deadlock is checked for. If a deadlock is detected, one or more processes are restarted from their last checkpoint. The process of restarting a process from a checkpoint is called rollback. The hope is that the resource requests will not interleave again to produce deadlock.

Deadlock recovery is generally used when deadlocks are rare, and the cost of recovery (process termination or rollback) is low.

Process check pointing can also be used to improve reliability (long running computations), assist in process migration (Sprite, Mach), or reduce startup costs (emacs).

If a system does eventually slip into a deadlock, measures must be taken to break the deadlock so that the execution may proceed.

There are two solutions for recovery from deadlock

- Process termination
- Resource preemption

Process Termination

When deadlock does occur, it may be necessary to bring the system down, or at least manually kill a number of processes, but even that is not an extreme solution in most situations.

Abort all deadlocked processes: this method clearly will break the deadlock cycle, but at a great expense, since these processes may have computed for a long time, and the results of these partial computations must be discarded, and probably must be recomputed later

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

Resource Preemption

Resource preemption is a technique used to break a deadlock in a computer system. In other words, it is a technique that an operating system uses to break up a situation in which every resource is used up by a variety of processes, each of which are waiting on more resources.

To eliminate deadlocks using resource preemption, we successively take away some resources from processes and give these resources to other processes until the deadlock is broken; this usually occurs when one process is given enough resources to be able to complete, then frees up all of the resources it was using.

Resources, in terms of resource preemption, refer to anything that might be of importance in a computer: memory, a particular piece of data, CPU time, use of devices such as your hard drive or your printer, and so forth. Managing these resources is a fundamental part of what your operating system does to keep your computer running.

If resource preemption is used to deal with deadlocks, three issues must be considered.

Selection of a Victim

Which processes can have resources taken away from them easily? The goal here is to minimize cost; if a resource can be taken away from one process much more easily than an identical resource can be taken away from another process, we select the victim that we will have the easiest time removing the resource from.

Rollback

If we preempt a resource from a process, what should be done with that process? If a resource is taken away, it obviously cannot continue in its current state; it would be much as if I chopped off one of your legs and expected you to continue to walk normally.

One thing we can do is roll the process back to a safe state; one where the resource hasn't been taken up by the process yet.

Unfortunately, most operating systems don't have the capability to store states of processes as they progress normally, and it is often impossible to determine a safe state given just the current state.

The solution then is a total rollback; aborting the process and starting from scratch. A total rollback is usually a last ditch resort, only usable if a deadlock can't be resolved without a total rollback.

Starvation

This happens when the same process is chosen again and again as a victim, making sure that that process can never finish. To prevent starvation, an operating system usually limits the number of times a specific process can be chosen as a victim.

Most modern operating systems attempt resource preemption as a way to solve deadlocks. The alternative method of killing off processes often means that a much greater amount of computational work is lost; it's not a preferable method of dealing with deadlocks simply because of the lost time.

Implementing resource preemption is often one of the trickiest parts of writing an operating system. Many operating systems use a very simple method of selecting processes for preemption: the one with the lowest priority receives a total rollback, then on up the chain until the highest priority process can run.

Other, more complex schemes exist, but most operating systems in wide use use some form of the method just described to break up deadlocks.

Resource preemption is an integral part of the operating system that runs your computer. Every time you start an application, resource preemption has to decide what program gets use the memory and CPU of your computer. It's a vital task that your operating system deals with for you.

Check Your Progress

1. What is rollback?
2. What is starvation?

5.5 LET US SUM UP

A deadlock is a situation where a group of processes are permanently blocked as a result of each process having acquired a subset of the resources needed for its completion and waiting for release of the remaining resources held by others in the same group-thus making it impossible for any of the processes to proceed. In a multiprogramming environment, more than one process exists in the system competing for the resources. Based on criteria and algorithms employed by the system, one of them is selected at a time, is granted requisite resources and is executed while other candidate processes wait for their turn. Deadlocks can occur in concurrent environments as a result of uncontrolled granting of system resources to requesting processes. A deadlock situation can arise if four conditions (called

Bernsterin's Conditions) hold simultaneously in a system - Mutual exclusion, Hold and wait, No preemption, and Circular wait. Resources of a computer system whose allocation is subject to deadlocks can be broadly categorized into two classes: reusable and consumable resources. Deadlocks can be expressed more clearly using a directed graph, called a system resource-allocation graph or precedence graph. This graph has vertices so arranged that depicts the before-after relationship between the processes. Deadlock prevention takes measures so that the system does not go into the deadlock condition in the first place.

Deadlock avoidance attempts to assure that the request will not lead to deadlock. Detection is a mechanism to identify a deadlock situation. If a system does eventually slip into a deadlock, recovery attempts to break the deadlock so that the execution may proceed. Banker's algorithm can be employed to determine whether a particular process state is safe from deadlock or not.

5.6 KEYWORDS

Deadlock: A situation wherein execution of processes halts because of cyclic waiting for the required resources.

Deadlock Prevention: It is a measure so that the system does not go into the deadlock condition in the first place.

Deadlock Avoidance: It attempts to assure that the request will not lead to deadlock.

Detection: It is a mechanism to identify a deadlock situation.

Reusable Resources: A resource that can be re-allocated and de-allocated between processes.

Consumable Resources: A resource that once allocated cannot be de-allocated and re-allocated again to any process.

Resource Allocation Graph: A graphical representation of resource allocation among various processes.

Banker's Algorithm: A simple algorithm that helps detect and therefore avoid a deadlock situation.

5.7 QUESTIONS FOR DISCUSSION

1. Explain the different Deadlock strategies.
2. Can a process be allowed to request multiple resources simultaneously in a system where deadlock are avoided? Discuss why or why not.
3. How deadlock situation are avoided and prevented so that no systems are locked by deadlock?
4. Determine whether there is a deadlock in the above situation.

Check Your Progress: Model Answers

1. One thing we can do is roll the process back to a safe state; one where the resource hasn't been taken up by the process yet.
2. This happens when the same process is chosen again and again as a victim, making sure that that process can never finish.

5.8 SUGGESTED READINGS

Andrew S. Tanenbaum, *Modern Operating System,*: Prentice Hall

Silberschatz Galvin, *Operating System Concepts,*: Addison Wesley

Andrew M. Lister, *Fundamentals of Operating Systems,*: Wiley

Colin Ritchie, *Operating Systems,*: BPB Publications

UNIT IV

LESSON

6

I/O MANAGEMENT

CONTENTS

6.0 Aims and Objectives

6.1 Introduction

6.2 I/O Management

6.3 I/O Devices

- 6.3.1 Input Device
- 6.3.2 Output Device

6.4 Device Controllers

6.5 Device Drivers

6.6 Memory-Mapped I/O

6.7 Disk Structure

- 6.7.1 Making Tracks
- 6.7.2 Sectors and Clusters

6.8 Disk Scheduling and Algorithms

- 6.8.1 First Come First Served (FCFS)
- 6.8.2 Circular Scan (C-Scan)
- 6.8.3 Look
- 6.8.4 Circular Look (C-Look)

6.9 Disk Management

6.10 Let us Sum up

6.11 Keywords

6.12 Questions for Discussion

6.13 Suggested Readings

6.0 AIMS AND OBJECTIVES

After studying this lesson, you will be able to:

- Explain the overview of I/O Systems
- Differentiate the types of I/O Devices
- Concept of Disk Scheduling and its algorithms

6.1 INTRODUCTION

The central processing unit is the unseen part of a computer system, and users are only dimly aware of it. But users are very much aware of the input and output associated with the computer. They submit input data to the computer to get processed information, the output. One of the important tasks of the operating system is to control all of the I/O devices, such as issuing commands concerning data transfer or status polling, catching and processing interrupts as well as handling different kind oferrors. In this lesson we will discuss how operating system handles the inputs and outputs. A variety of input-output devices that can be attached to a computer system, that varies greatly in many of their aspects. They may be character devices (i.e. transferring a single character at a time) or block devices (i.e. transferring a chunk of characters at a time). The speed of operation of these devices varies considerably. General types of I/O devices include secondary storage devices (disks, tapes, CD-ROMs etc.), human-interface devices (monitor, keyboard, mouse etc.), transmission devices (network cards, modems), multimedia devices (sound card, MIDI devices) etc. The control of these devices poses a major concern to the operating systems because each of these devices requires a different mechanism for their control by operating system.

Each device has device controller, the device controller are electronics hardware which is equipped in the port. The ports are the hardware interface, through which transmission medium (cable etc) are connected to the computer and sends signals through these ports.

6.2 I/O MANAGEMENT

Input/output, or I/O, refers to the communication between an information processing system (such as a computer), and the outside world - possibly a human, or another information processing system. Inputs are the signals or data received by the system, and outputs are the signals or data sent from it. The term can also be used as part of an action; to "perform I/O" is to perform an input or output operation. I/O devices are used by a person (or other system) to communicate with a computer. For instance, keyboards and mouses are considered input devices of a computer, while monitors and printers are considered output devices of a computer. Devices for communication between computers, such as modems and network cards, typically serve for both input and output.

Note that the designation of a device as either input or output depends on the perspective. Mouses and keyboards take as input physical movement that the human user outputs and convert it into signals that a computer can understand. The output from these devices is input for the computer. Similarly, printers and monitors take as input signals that a computer outputs. They then convert these signals into representations that human users can see or read. (For a human user the process of reading or seeing these representations is receiving input.)

In computer architecture, the combination of the CPU and main memory (i.e. memory that the CPU can read and write to directly, with individual instructions) is considered the heart of a computer, and from that point of view any transfer of information from or to that combination, for example to or from a disk drive, is considered I/O. The CPU and its supporting circuitry provide I/O methods that are used in low-level computer programming in the implementation of device drivers.

Higher-level operating system and programming facilities employ separate, more abstract I/O concepts and primitives. For example, most operating systems provide application programs with the concept of files. The C and C++ programming languages, and operating systems in the Unix family, traditionally abstract files and devices as streams, which can be read or written, or sometimes both. The C standard library provides functions for manipulating streams for input and output.

6.3 I/O DEVICES

I/O devices allow your managed system to gather, store, and transmit data. I/O devices are found in the server unit itself and in expansion units and towers that are attached to the server. I/O devices can be embedded into the unit, or they can be installed into physical slots.

Not all types of I/O devices are supported for all operating systems or on all server models. For example, Switch Network Interface (SNI) adapters are supported only on certain server models, and are not supported for i5/OS® logical partitions.

6.3.1 Input Device

A hardware device that sends information into the CPU is known as input device. Without any input devices a computer would simply be a display device and not allow users to interact with it, much like a TV. Below is a listing of different types of computer input devices.

Keyboard: One of the main input devices used on a computer, a PC's keyboard looks very similar to the keyboards of electric typewriters, with some additional keys.

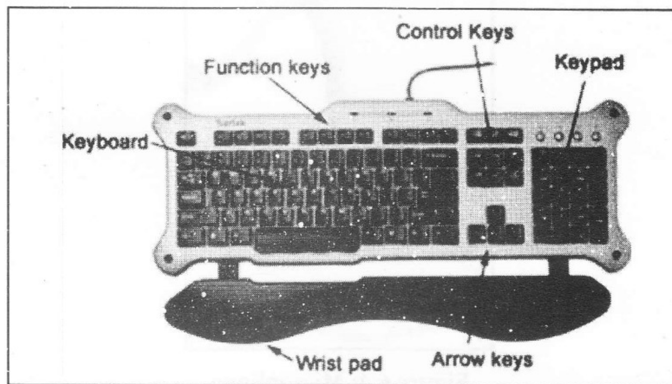

Figure 6.1: Keyboard

Mouse: An input device that allows an individual to control a mouse pointer in a graphical user interface (GUI). Utilizing a mouse a user has the ability to perform various functions such as opening a program or file and does not require the user to memorize commands, like those used in a text-based environment such as MS-DOS. To the right is a picture of a Microsoft IntelliMouse and is an example of what a mouse may look like.

Figure 6.2: Mouse

Scanner: Hardware input device that allows a user to take an image and/or text and convert it into a digital file, allowing the computer to read and/or display the scanned object. A scanner is commonly connected to a computer USB, Firewire, Parallel or SCSI port.

Figure 6.3: Scanner

Microphone: Sometimes abbreviated as mic, a microphone is a hardware peripheral that allows computer users to input audio into their computers.

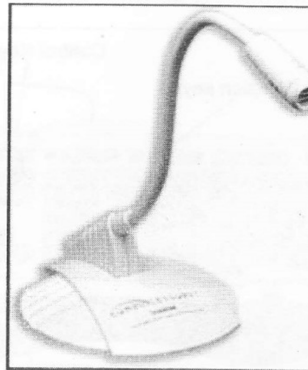

Figure 6.4: Microphone

Web Cam: A camera connected to a computer or server that allows anyone connected to the Internet to view still pictures or motion video of a user. The majority of webcam web sites are still pictures that are frequently refreshed every few seconds, minutes, hours, or days. However, there are some sites and personal pages that can supply streaming video for users with broadband.

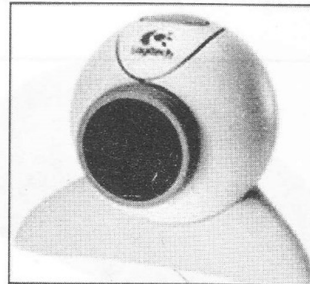

Figure 6.5: Web Cam

Digital camera: A type of camera that stores the pictures or video it takes in electronic format instead of to film. There are several features that make digital cameras a popular choice when compared to film cameras. First, the feature often enjoyed the most is the LCD display on the digital camera. This

display allows users to view photos or video after the picture or video has been taken, which means if you take a picture and don't like the results, you can delete it; or if you do like the picture, you can easily show it to other people. Another nice feature with digital cameras is the ability to take dozens, sometimes hundreds of different pictures.

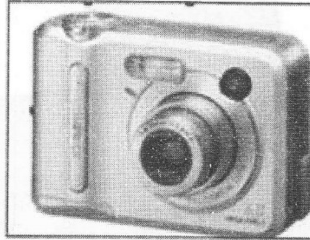

Figure 6.6: Digital Camera

Joystick: A computer joystick allows an individual to easily navigate an object in a game such as navigating a plane in a flight simulator.

Figure 6.7: Joystick

6.3.2 Output Device

Any peripheral that receives and/or displays output from a computer is known as output device. Below are some examples of different types of output devices commonly found on a computer.

Monitor: Also called a Video Display Terminal (VDT) a monitor is a video display screen and the hard shell that holds it. In its most common usage, monitor refers only to devices that contain no electronic equipment other than what is essentially needed to display and adjust the characteristics of an image.

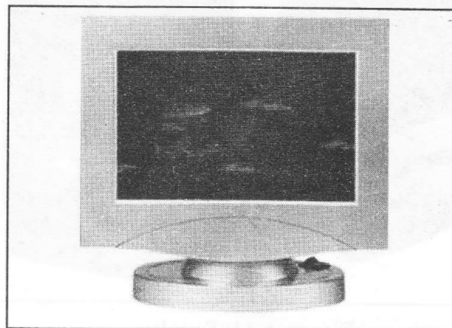

Figure 6.8: Monitor

Printer: An external hardware device responsible for taking computer data and generating a hard copy of that data. Printers are one of the most used peripherals on computers and are commonly used to print text, images, and/or photos.

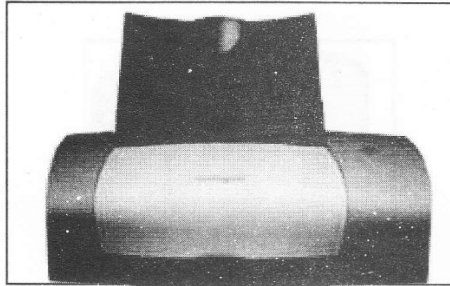

Figure 6.9: Printer

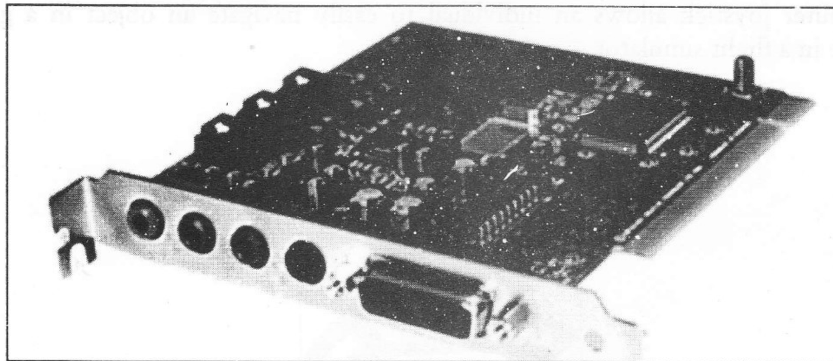

Figure 6.10: Sound Card

Speakers: A hardware device connected to a computer's sound card that outputs sounds generated by the card.

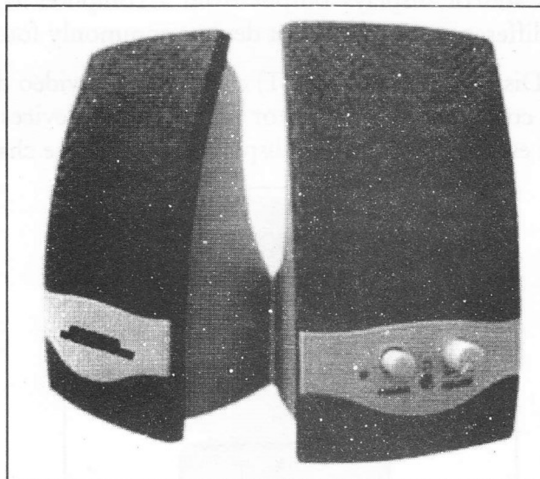

Figure 6.11: Speaker

Video card: Also known as a graphics card, video card, video board, or a video controller, a video adapter is an internal circuit board that allows a display device, such as a monitor, to display images from the computer.

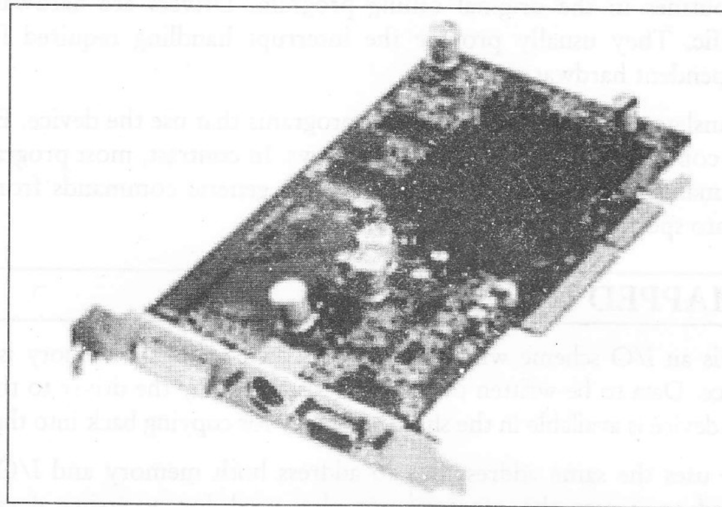

Figure 6.12 Video Card

6.4 DEVICE CONTROLLERS

The device controller is the hardware that controls the communication between the system and the peripheral drive unit. It takes care of low level operations such as error checking, moving disk heads, data transfer, and location of data on the device.

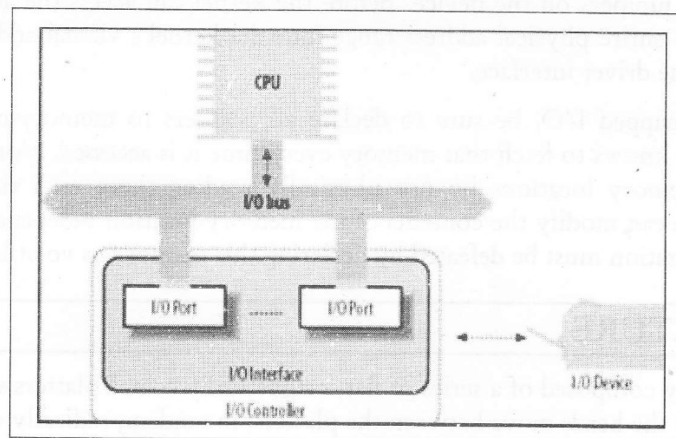

Figure 6.13: Diagram of Device Controller

6.5 DEVICE DRIVERS

The device driver is the software that operates the controller. A device driver, or software driver is a computer program allowing higher-level computer programs to interact with a computer hardware device.

A driver typically communicates with the device through the computer bus or communications subsystem to which the hardware is connected. When a calling program invokes a routine in the driver, the driver issues commands to the device. Once the device sends data back to the driver, the driver may invoke routines in the original calling program. Drivers are hardware-dependent and operating-system-specific. They usually provide the interrupt handling required for any necessary asynchronous time-dependent hardware interface.

A driver acts like a translator between the device and programs that use the device. Each device has its own set of specialized commands that only its driver knows. In contrast, most programs access devices by using generic commands. The driver, therefore, accepts generic commands from a program and then translates them into specialized commands for the device.

6.6 MEMORY-MAPPED I/O

Memory-mapped I/O is an I/O scheme where the device's own on-board memory is mapped into the processor's address space. Data to be written to the device is copied by the driver to the device memory, and data read in by the device is available in the shared memory for copying back into the system memory.

Memory-mapped I/O uses the same address bus to address both memory and I/O devices, and the CPU instructions used to access the memory are also used for accessing devices. In order to accommodate the I/O devices, areas of CPU's addressable space must be reserved for I/O rather than memory. The reservation might be temporary-the Commodore 64 could bank switch between its I/O devices and regular memory- or permanent. Each I/O device monitors the CPU's address bus and responds to any CPU's access of device-assigned address space, connecting the data bus to a desirable device's hardware register.

The adapter's memory is mapped into system address space through the PCI BIOS, a software setup program, or by setting jumpers on the device. Before the kernel can access the adapter's memory, it must map the adapter's entire physical address range into the kernel's virtual address space using the functions supplied by the driver interface.

When using memory-mapped I/O, be sure to declare all pointers to memory-mapped addresses as volatile so the compiler knows to fetch that memory every time it is accessed. Normally, the compiler optimizes access to memory locations by not physically reading them each time. With memory-mapped I/O, the device can modify the contents of the memory location independently of the kernel, so this compiler optimization must be defeated by declaring this memory as volatile.

6.7 DISK STRUCTURE

A hard disk is physically composed of a series of flat, magnetically coated platters stacked on a spindle. The spindle turns while the heads move between the platters, in tandem, radically reading/writing data onto the platters.

It is a sealed unit containing a number of platters in a stack. Hard disks may be mounted in a horizontal or a vertical position. In this description, the hard drive is mounted horizontally.

Electromagnetic read/write heads are positioned above and below each platter. As the platters spin, the drive heads move in toward the center surface and out toward the edge. In this way, the drive heads can reach the entire surface of each platter.

6.7.1 Making Tracks

On a hard disk, data is stored in thin, concentric bands. A drive head, while in one position can read or write a circular ring, or band called a track. There can be more than a thousand tracks on a 3.5-inch hard disk. Sections within each track are called sectors. A sector is the smallest physical storage unit on a disk, and is almost always 512 bytes (0.5 kB) in size.

The Figure below shows a hard disk with two platters.

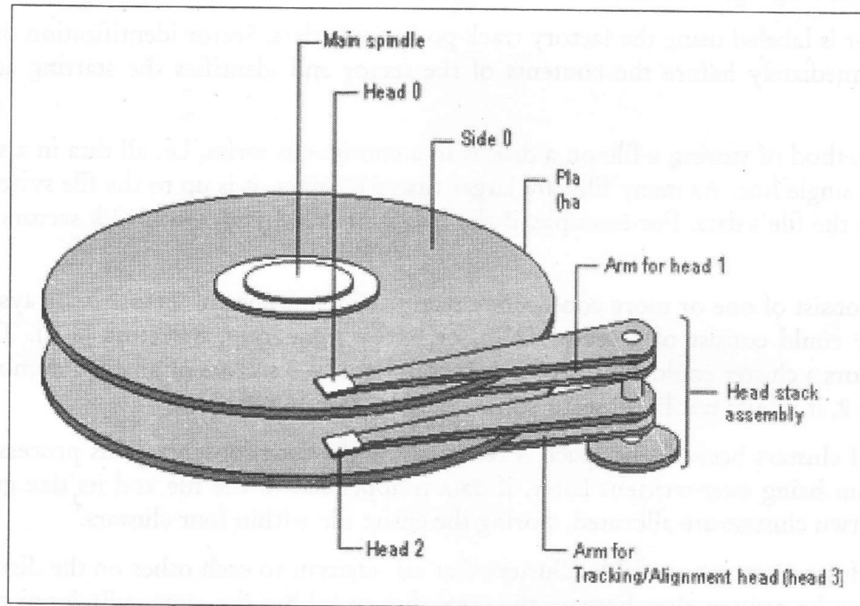

Figure 6.14: Physical Disk Structure

The structure of older hard drives (i.e. prior to Windows 95) will refer to a cylinder/ head/ sector notation. A cylinder is formed while all drive heads are in the same position on the disk. The tracks, stacked on top of each other form a cylinder. This scheme is slowly being eliminated with modern hard drives. All new disks use a translation factor to make their actual hardware layout appear continuous, as this is the way that operating systems from Windows 95 onward like to work.

To the operating system of a computer, tracks are logical rather than physical in structure, and are established when the disk is low-level formatted. Tracks are numbered, starting at 0 (the outermost edge of the disk), and going up to the highest numbered track, typically 1023, (close to the center). Similarly, there are 1,024 cylinders (numbered from 0 to 1023) on a hard disk.

The stack of platters rotate at a constant speed. The drive head, while positioned close to the center of the disk reads from a surface that is passing by more slowly than the surface at the outer edges of the disk. To compensate for this physical difference, tracks near the outside of the disk are less-densely populated with data than the tracks near the center of the disk. The result of the different data density is that the same amount of data can be read over the same period of time, from any drive head position.

The disk space is filled with data according to a standard plan. One side of one platter contains space reserved for hardware track-positioning information and is not available to the operating system. Thus, a disk assembly containing two platters has three sides available for data. Track-positioning data is

written to the disk during assembly at the factory. The system disk controller reads this data to place the drive heads in the correct sector position.

6.7.2 Sectors and Clusters

A sector, being the smallest physical storage unit on the disk, is almost always 512 bytes in size because 512 is a power of 2 (2 to the power of 9). The number 2 is used because there are two states in the most basic of computer languages - on and off.

Each disk sector is labeled using the factory track-positioning data. Sector identification data is written to the area immediately before the contents of the sector and identifies the starting address of the sector.

The optimal method of storing a file on a disk is in a contiguous series, i.e. all data in a stream stored end-to-end in a single line. As many files are larger than 512 bytes, it is up to the file system to allocate sectors to store the file's data. For example, if the file size is 800 bytes, two 512 k sectors are allocated for the file.

A cluster can consist of one or more consecutive sectors. The number of sectors is always an exponent of 2. A cluster could consist of 1 sector (2^0), or, more frequently, 8 sectors (2^3). The only odd number of sectors a cluster could consist of is 1. It could not be 5 sectors or an even number that is not an exponent of 2. It would not be 10 sectors, but could be 8 or 16 sectors.

They are called clusters because the space is reserved for the data contents. This process protects the stored data from being over-written. Later, if data is appended to the file and its size grows to 1600 bytes, another two clusters are allocated, storing the entire file within four clusters.

If contiguous clusters are not available (clusters that are adjacent to each other on the disk), the second two clusters may be written elsewhere on the same disk or within the same cylinder or on a different cylinder - wherever the file system finds two sectors available. A file stored in this non-contiguous manner is considered to be fragmented. Fragmentation can slow down system performance if the file system must direct the drive heads to several different addresses to find all the data in the file you want to read.

The extra time for the heads to travel to a number of addresses causes a delay before the entire file is retrieved.

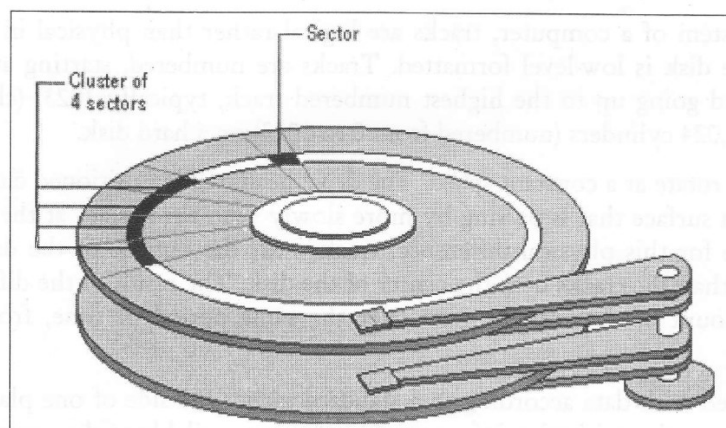

Figure 6.15: Sectors and Clusters

Cluster size can be changed to optimize file storage. A larger cluster size reduces the potential for fragmentation, but increases the likelihood that clusters will have unused space. Using clusters larger than one sector reduces fragmentation, and reduces the amount of disk space needed to store the information about the used and unused areas on the disk.

Most disks used in personal computers today rotate at a constant angular velocity. The tracks near the outside of the disk are less densely populated with data than the tracks near the center of the disk. Thus, a fixed amount of data can be read in a constant period of time, even though the speed of the disk surface is faster on the tracks located further away from the center of the disk.

Modern disks reserve one side of one platter for track positioning information, which is written to the disk at the factory during disk assembly. It is not available to the operating system. The disk controller uses this information to fine tune the head locations when the heads move to another location on the disk. When a side contains the track position information, that side cannot be used for data. Thus, a disk assembly containing two platters has three sides that are available for data.

6.8 DISK SCHEDULING & ALGORITHMS

In order to satisfy an I/O request the disk controller must first move the head to the correct track and sector. Moving the head between cylinders takes a relatively long time so in order to maximize the number of I/O requests which can be satisfied the scheduling policy should try to minimize the movement of the head. On the other hand, minimizing head movement by always satisfying the request of the closest location may mean that some requests have to wait a long time. Thus, there is a trade-off between throughput (the average number of requests satisfied in unit time) and response time (the average time between a request arriving and it being satisfied). Various different disk scheduling policies are used:

6.8.1 First Come First Served (FCFS)

The disk controller processes the I/O requests in the order in which they arrive, thus moving backwards and forwards across the surface of the disk to get to the next requested location each time. Since no reordering of request takes place the head may move almost randomly across the surface of the disk. This policy aims to minimise response time with little regard for throughput. Shortest Seek Time First (SSTF)

Each time an I/O request has been completed the disk controller selects the waiting request whose sector location is closest to the current position of the head. The movement across the surface of the disk is still apparently random but the time spent in movement is minimized. This policy will have better throughput than FCFS but a request may be delayed for a long period if many closely located requests arrive just after it. SCAN The drive head sweeps across the entire surface of the disk, visiting the outermost cylinders before changing direction and sweeping back to the innermost cylinders. It selects the next waiting requests whose location it will reach on its path backwards and forwards across the disk. Thus, the movement time should be less than FCFS but the policy is clearly fairer than SSTF.

6.8.1 Circular SCAN (C-SCAN)

C-SCAN is similar to SCAN but I/O requests are only satisfied when the drive head is traveling in one direction across the surface of the disk. The head sweeps from the innermost cylinder to the outermost