

Chapter 1

FUNDAMENTALS OF OPERATING SYSTEM

An operating system is a program that acts as an intermediary between a user of a computer and the computer hardware. The purpose of an operating system is to provide an environment in which a user can execute program.

An operating system is an important part of almost every computer system. A computer system can be divided roughly into four components: the hardware, the operating system, the applications programs, and the users)

The hardware – the central processing unit (CPU), the memory, and the input / output (I/O) devices - provides the basic computing resources. The applications programs – such as compilers, database systems, games, and business programs – define the ways in which these resources are used to solve the computing problems of the users.

An operating system is a control program. A control program controls the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.

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

Simple Batch Systems

To speed up processing, jobs with similar needs were batched together and were run through the computer as a group. Thus, the programmers would leave their programs with the operator. The operator would sort programs into batches with similar requirements and, as the computer became available, would run each batch. The output from each job would be sent back to the appropriate programmer.

The definitive feature of a batch system is the lack of interaction between the user and the job while that job is executing. The job is prepared and submitted, and at some later time, the output appears. The delay between job submission and job completion (called turnaround time) may result from the amount of computing needed, or from delays before the operating system starts to process the job.

In this execution environment, the CPU is often idle. This idleness occurs because the speeds of the mechanical I/O devices are intrinsically slower than those of electronic devices)

SPOOLING

When a job is executed, the operating system satisfies its requests for card-reader input by reading from the disk and when the job requests the printer to output a line that line is copied into a system buffer and is written to the disk. When the job is completed, the output is actually printed. This form of processing is called spooling i.e. simultaneous peripheral operation on-line. Spooling, uses the disk as a huge buffer, for reading as far ahead as possible on input devices and for storing output files until the output devices are able to accept them.

Spooling is also used for processing data at remote sites. The CPU sends the data via communications paths to a remote printer. The remote processing is done at its own speed, with no CPU intervention. The CPU just needs to be notified when the processing is completed, so that it can spool the next batch of data.

Spooling overlaps the I/O of one job with the computation of other jobs. Spooling has a direct beneficial effect on the performance of the system. Spooling can keep both the CPU and the I/O devices working at much higher rates

Multi-programmed Batched Systems

Spooling provides an important data structure: a job pool. Spooling will generally result in several jobs that have already been read waiting on disk, ready to run. A pool of jobs on disk allows the operating system to select which job to run next, to increase CPU utilization. When jobs come in directly on cards or even on magnetic tape, it is not possible to run jobs in a different order. Jobs must be run sequentially, on a first-come, first-served basis. However, when several jobs are on a direct-access device, such as a disk, job scheduling becomes possible.

The most important aspect of job scheduling is the ability to multiprogram. Off-line operation and spooling for overlapped I/O have their limitations. A single user cannot, in general, keep either the CPU or the I/O devices busy at all times. Multiprogramming increases CPU utilization by organizing jobs such that the CPU always has one to execute.

The idea is as follows. The operating system keeps several jobs in memory at a time. This set of jobs is a subset of the jobs kept in the job pool (since the number of jobs that can be kept simultaneously in memory is usually much smaller than the number of jobs that can be in the job pool.) The operating system picks and begins to execute one of the jobs in the memory. Eventually, the job may have to wait for some task, such as a tape to be mounted, or an I/O operation to complete. In a non-multiprogrammed system, the CPU would sit idle. In a multiprogramming system, the operating system simply switches to and executes another job. When that job needs to wait, the CPU is switched to

another job, and so on. Eventually, the first job finishes waiting and gets the CPU back. As long as there is always some job to execute, the CPU will never be idle.

This idea is common in other life situations. A lawyer does not have only one client at a time. Rather, several clients may be in the process of being served at the same time. While one case is waiting to go to trial or to have papers typed, the lawyer can work on another case. If she has enough clients, a lawyer never needs to be idle. (Idle lawyers tend to become politicians, so there is a certain social value in keeping lawyers busy.)

Multiprogramming is the first instance where the operating system must make decisions for the users. Multiprogrammed operating systems are therefore fairly sophisticated. All the jobs that enter the system are kept in the job pool. This pool consists of all processes residing on mass storage awaiting allocation of main memory. If several jobs are ready to be brought into memory, and there is not enough room for all of them, then the system must choose among them. When the operating system selects a job from the job pool, it loads that job into memory for execution. Having several programs in memory at the same time requires having some form of memory management. Finally, multiple jobs running concurrently require that their ability to affect one another be limited in all phases of the operating system, including process scheduling, disk storage, and memory management. These considerations are discussed throughout the text.

Time-Sharing Systems

Multiprogrammed batched systems provide an environment where the various system resources (for example, CPU, memory, peripheral devices) are utilized effectively.

Time sharing, or multitasking, is a logical extension of multiprogramming. Multiple jobs are executed by the CPU switching between them, but the switches occur so frequently that the users may interact with each program while it is running.

An interactive, or hands-on, computer system provides on-line communication between the user and the system. The user gives instructions to the operating system or to a program directly, and receives an immediate response. Usually, a keyboard is used to provide input, and a display screen (such as a cathode-ray tube (CRT) or monitor) is used to provide output.

If users are to be able to access both data and code conveniently, an on-line file system must be available. A file is a collection of related information defined by its creator. Batch systems are appropriate for executing large jobs that need little interaction.

Time-sharing systems were developed to provide interactive use of a computer system at a reasonable cost. A time-shared operating system uses CPU

scheduling and multiprogramming to provide each user with a small portion of a time-shared computer. Each user has at least one separate program in memory. A program that is loaded into memory and is executing is commonly referred to as a process. When a process executes, it typically executes for only a short time before it either finishes or needs to perform I/O. I/O may be interactive; that is, output is to a display for the user and input is from a user keyboard. Since interactive I/O typically runs at people speeds, it may take a long time to completed

A time-shared operating system allows the many users to share the computer simultaneously. Since each action or command in a time-shared system tends to be short, only a little CPU time is needed for each user. As the system switches rapidly from one user to the next, each user is given the impression that she has her own computer, whereas actually one computer is being shared among many users.

Time-sharing operating systems are even more complex than are multiprogrammed operating systems. As in multiprogramming, several jobs must be kept simultaneously in memory, which requires some form of memory management and protection.

CHAPTER 2 PROCESS & THREAD MANAGEMENT

The design of an operating system must be done in such a way that all requirement should be fulfilled.

- The operating system must interleave the execution of multiple processes, to maximize processor utilization while providing reasonable response time.
- The operating system must allocate resources to processes in conformance with a specific policy
- The operating system may be required to support interprocess communication and user creation of processes

Processes and Process Control Blocks

Process can be defined as :-

- A program in execution
- An instance of a program running on a computer

- The entity that can be assigned to and executed on a processor
- A unit of activity characterized by the execution of a sequence of instructions, a current state, and an associated set of system resources

A process as an entity that consists of a number of elements. Two essential elements of a process are **program code**, and a **set of data** associated with that code. At any given point in time, while the program is executing, this process can be uniquely characterized by a number of elements, including the following:

- **Identifier:** A unique identifier associated with this process, to distinguish it from all other processes.
- **State:** If the process is currently executing, it is in the running state.
- **Priority:** Priority level relative to other processes.
- **Program counter:** The address of the next instruction in the program to be executed.
- **Memory pointers:** Includes pointers to the program code and data associated with this process, plus any memory blocks shared with other processes.
- **Context data:** These are data that are present in registers in the processor while the process is executing.
- **I/O status information:** Includes outstanding I/O requests, I/O devices (e.g., tape drives) assigned to this process, a list of files in use by the process, and so on.
- **Accounting information:** May include the amount of processor time and clock time used, time limits, account numbers, and so on.

The information in the preceding list is stored in a data structure, called a **process control block**, which is created and managed by the operating system. Process control block contains sufficient information so that it is possible to interrupt a running process and later resume execution as if the interruption had not occurred.

PROCESS STATES

The operating system's responsibility is controlling the execution of processes; this includes determining the interleaving pattern for execution and allocating resources to processes.

A process may be in one of two states: Running or Not Running. When the operating system creates a new process, it creates a process control block for the process and enters that process into the system in the Not Running state. The process exists, is known to the operating system, and is waiting to execute.

Processes that are not running must be kept in some sort of queue, waiting their turn to execute. There is a single queue in which each entry is a pointer to the process control block of a particular process.

Process Creation : When a new process is to be added to those currently being managed, the operating system builds the data structures that are used to manage the process and allocates address space in main memory to the process. This is the creation of a new process.

In a batch environment, a process is created in response to the submission of a job. In an interactive environment, a process is created when a new user attempts to log on. In both cases, the operating system is responsible for the creation of the new process.

Operating system created all processes in a way that was transparent to the user or application program, and this is still commonly found with many contemporary operating systems. When the operating system creates a process at the explicit request of another process, the action is referred to as **process spawning**.

When one process spawns another, the former is referred to as the **parent process**, and the spawned process is referred to as the **child process**.

Process Termination : Any computer system must provide a means for a process to indicate its completion. A batch job should include a Halt instruction or an explicit operating system service call for termination.

A Five-State Model

The queue is a first-in-first out list and the processor operates in **round-robin** fashion on the available processes. Process can be also explained in five state model.

- **Running:** The process that is currently being executed. For this chapter, we will assume a computer with a single processor, so at most one process at a time can be in this state.
- **Ready:** A process that is prepared to execute when given the opportunity.
- **Blocked:** A process that cannot execute until some event occurs, such as the completion of an I/O operation.
- **New:** A process that has just been created but has not yet been admitted to the pool of executable processes by the operating system. Typically, a new process has not yet been loaded into main memory, although its process control block has been created.
- **Exit:** A process that has been released from the pool of executable processes by the operating system, either because it halted or because it aborted for some reason.

The New and Exit states are useful constructs for process management. If a new user attempts to log onto a time-sharing system for execution, the operating system can define a new process.

The process is in the New state means that the operating system has performed the necessary actions to create the process but has not committed itself to the execution of the process.

A process is terminated when it reaches a natural completion point, when it aborts due to an unrecoverable error, or when another process with the appropriate authority causes the process to abort. Termination moves the process to the exit state.

Types of events that lead to each state transition for a process

- **Null → New:** A new process, is created to execute a program
- **New → Ready:** The operating system will move a process from the New state to the Ready state when it is prepared to take on an additional process. Most systems set some limit based on the number of existing processes* or the amount of virtual memory committed to existing processes. This limit assures that there are not so many active processes as to degrade performance
- **Ready → Running:** When it is time to select a new process to run, the operating system chooses one of the processes in the Ready state. This is the job of the scheduler or dispatcher
- **Running → Exit:** The currently running process is terminated by the operating system if the process indicates that it has completed or if it aborts.
- **Running → Ready:** The most common reason for this transition is that the running process has reached the maximum allowable time for uninterrupted execution.
- **Running → Blocked:** A process is put in the Blocked state if it requests something for which it must wait. A request to the operating system is usually in the form of a system service call; that is, a call from the running program to a procedure that is part of the operating system code. It can also request a resource, such as a file or a shared section of virtual memory, that is not immediately available. Or the process may initiate an action, such as an I/O operation, that must be completed before the process can continue. When processes communicate with each other, a process may be blocked when it is waiting for another process to provide data or waiting for a message from another process
- **Blocked → Ready:** A process in the Blocked state is moved to the Ready state when the event for which it has been waiting occurs
- **Ready → Exit:** In some systems, a parent may terminate a child process at any time. Also, if a parent terminates, all child processes associated with that parent may be terminated.
- **Blocked → Exit :** A queuing discipline might be implemented with two queues: a Ready queue and a Blocked queue. As each process is admitted to the system, it is placed in the Ready queue. In the absence of any priority scheme, this can be a Simple first-in-first-out queue. When a

running process is removed from execution, it is either terminated or placed in the Ready or Blocked queue, depending on the circumstances. Any process in the Blocked queue that has been waiting on that event only is moved to the Ready queue i.e. when an event occurs, the operating system must scan the entire blocked queue, searching for those processes waiting on that event. And when the event occurs, the entire list of processes in the appropriate queue can be moved to the Ready state. But if Operating system on priority scheme, then it would be convenient to have a number of Ready queues, one for each priority level. The operating system could then readily determine which is the highest-priority ready process that has been waiting the longest.

Suspended Process:

The processor is so much faster than I/O that it will be common for all of the processes in memory to be waiting for I/O. Thus, even, with multiprogramming, a processor could be idle most of the time. To avoid this main memory could be expanded to accommodate more processes. And another solution is swapping, which involves moving part or all of a process from main memory to disk. When none of the processes in main memory is in the Ready state, the operating system swaps one of the blocked processes out onto disk into a suspend queue. This is a queue of existing processes that have been temporarily kicked out of main memory, or suspended. The operating system then brings in another process from the suspend queue.

With the use of swapping, one other state will increase in the process called the Suspend state. When all of the processes in main memory are in the Blocked state, the operating system can suspend one process by putting it in the Suspend state and transferring it to disk. The space that is freed in main memory can then be used to bring in another process.

Process in the Suspend state was originally blocked on a particular event. When that event occurs, the process is not blocked and is potentially available for execution.

Now the state can be further defined as

- **Ready:** The process is in main memory and available for execution.
- **Blocked:** The process is in main memory and awaiting an event.
- **Blocked/Suspend:** The process is in secondary memory and awaiting an event.
- **Ready/Suspend:** The process is in secondary memory but is available for execution as soon as it is loaded into main memory.
- **Blocked → Blocked/Suspend:** If there are no ready processes, then at least one blocked process is swapped out to make room for another process that is not blocked.

- **Blocked / Suspend → Ready / Suspend:** A process in the Blocked / Suspend state is moved to the Ready/Suspend state when the event for which it has been waiting occurs.
- **Ready / Suspend → Ready:** When there are no ready processes in main memory, the operating system will need to bring one in to continue execution. In addition, it might be the case that a process in the Ready/Suspend state has higher priority than any of the processes in the Ready state.
- **Ready → Ready/Suspend: Normally,** the operating system would prefer to suspend a blocked process rather than a ready one, because the ready process can now be executed, whereas the blocked process is taking up main memory space and cannot be executed.

Process Management :

If the operating system is to manage process and resources, it must have information about the current status of each process and resources. The operating system constructs and maintains tables of information about each entity that it is managing. Types of tables maintained by the operating system: memory, I/O, file, and process.)

Memory tables are used to keep track of both main (real) and secondary (virtual) memory. Some of main memory is reserved for use by the operating system; the remainder is available for use by processes. Processes are maintained on secondary memory using some sort of virtual memory or simple swapping mechanism. The memory tables must include the following information:

- The allocation of main memory to processes
- The allocation of secondary memory to processes
- Any protection attributes of blocks of main or virtual memory, such as which processes may access certain shared memory regions
- Any information needed to manage virtual memory

I/O tables are used by the operating system to manage the I/O devices and channels of the computer system. The operating system also maintains **file tables**. These tables provide information about the existence of files, their location on secondary memory, their current status, and other attributes.

The operating system must maintain **process tables** to manage processes.

Process Control Structures

The operating system must know if it is to manage and control a process. First, it must know where the process is located, and second, it must know the attributes of the process that are necessary for its management. (e.g, process ID and process state.)

Process must include a program or set of programs to be executed. Associated with these programs is a set of data locations for local and global variables and any defined constants. Thus, a process will consist of at least sufficient memory to hold the programs and data of that process. Each process has associated with it a number of attributes that are used by the operating system for process control.

Process Attributes

The process control block information into three general categories:

- Process identification
- Processor state information
- Process control information

With respect to process identification in each process is assigned a unique numeric identifier, which may simply indexed into the primary process table or there must be a mapping that allows the operating system to locate the appropriate table based on the process identifier. When the processes communicate with one another, the process identifier informs the operating system of the destination of the particular communication. When process are allowed to create other processes, identifiers indicate the parent and descendents of each process.

In addition to these process identifiers, a process may be assigned a user identifier that indicates the user responsible for the job.

Processor state information consists of the contents of processor registers. While a process is running, of course, the information is in the registers. When a process is interrupted, all of this register information must be saved so that it can be restored when the process resumes execution. Information in the process control block can be called, **process control information**. This is the additional information needed by the operating system to control and coordinate the various active process.

Modes of Execution

Processors support at least two modes of execution. Certain instructions can only be executed in the more-privileged mode. These would include reading or altering a control register, such as the program status word; primitive I/O instructions; and instructions that relate to memory management. The less-privileged mode is often referred to as the **user mode**, because user programs typically would execute in this mode. The more-privileged mode is referred to as the **system mode, control mode, or kernel mode**.

It is necessary to protect the operating system and key operating system tables, such as process control blocks, from interference by user programs. In the kernel

mode, the software has complete control of the processor and all its instructions, registers, and memory. This level of control is not necessary and for safety is not desirable for user programs.

Process Switching :-

Sometimes, a running process is interrupted and the operating system assigns another process to the Running state and turns control over to that process, which is defined as process switching.

A process switch may occur any time that the operating system has gained control from the currently running process. The main cause is system interrupt i.e. two kinds of system interrupts, one of which is simply referred to as an interrupt, and the other as a trap. Trap is event that is external to and independent of the currently running process, such as the completion of an I/O operation. Interrupt relates to an error or exception condition generated within the currently running process, such as an illegal file access attempt. With an ordinary **interrupt**, control is first transferred to an interrupt handler, then to an operating system

- **Clock interrupt:** The operating system determines whether the currently running process has been executing for the maximum allowable unit of time, i.e **time slice**.
- **I/O interrupt:** The operating system determines what I/O action has occurred. If the I/O action constitutes an event for which one or more processes are waiting, then the operating system moves all of the corresponding blocked processes to the Ready state.
- **Memory fault:** The processor encounters a virtual memory address reference for a word that is not in main memory. The operating system must bring in the block (page or segment) of memory containing the reference from secondary memory to main memory. After the I/O request is issued to bring in the block of memory, the process with the memory fault is placed in a blocked state; the operating system then performs a process switch to resume execution of another process.

With a trap the operating system determines if the error or exception condition is fatal, then the currently running process is moved to the Exit state and a process switch occurs.

The operating system may be activated by a **supervisor call** from the program being executed. For example, a user process is running and an instruction is executed that requests an I/O operation, such as a file open. This call results in a transfer to a routine that is part of the operating system code.

Mode Switching :

1. It sets the program counter to the starting address of an interrupt handler program.

2. It switches from user mode to kernel mode so that the interrupt processing code may include privileged instructions

The interrupt handler is typically a short program that performs a few basic tasks related to an interrupt. If the interrupt relates to an I/O event, the interrupt handler will check for an error condition. If an error has occurred, the interrupt handler may send a signal to the process that originally requested the I/O operation.

Change of Process State A mode switch may occur without changing the state of the process that is currently in the Running state. In that case, the context saving and subsequent restoration involve. However, if the currently running process is to be moved to another state (Ready, blocked, etc.), then the operating system must make substantial changes in its environment. The steps involved in a full process switch are as follows:

1. Save the context of the processor, including program counter and other registers.
2. Update the process control block of the process that is currently in the Running state. This includes changing the state of the process to one of the other states.
3. Move the process control block of this process to the appropriate queue
4. Select another process for execution.
5. Update the process control block of the process selected. This includes changing the state of this process to Running.
6. Update memory-management data structures.

Restore the context of the processor to that which existed at the time the selected process was last switched out of the Running state, by loading in the previous values of the program counter and other registers.

CHAPTER 3 CONCURRENCY CONTROL

Semaphores

The solutions of the critical section problem represented in the section 6.3 is not easy to generalize to more complex problems. To overcome this difficulty, we can use a synchronization tool called a semaphore. A semaphore S is an integer variable that, apart from initialization, is accessed two standard atomic operations: wait and signal. These operations were originally termed P (for wait; from the Dutch *proberen*, to test) and V (for signal; from *verhogen*, to increment). The classical definition of wait and signal are

```
Wait (S) :   while  $S \leq 0$  do no-op;  
            S := S - 1;
```

signal(S): S := S + !;

The integer value of the semaphore in the wait and signal operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of the wait(S), the testing of the integer value of S ($S \leq 0$), and its possible modification ($S := S - 1$), must also be executed without interruption.

Usage

Semaphores can be used to deal with n process critical section problem. The n processes shares the semaphores, mutex (standing for mutual exclusion), initialized to 1.)

```
repeat
    wait (mutex)
        critical section
    signal (mutex);
    remainder section
until false;
```

```
    S1 :
        signal(synch);
in process P1, and the statements
    wait(synch);
    S2
```

in process P₂. Because synch is initialized to 0, P₂ will execute S₂ only after P₁ has invoked signal(synch), which is after S₁.

Implementation

When-a-process-executes the wait operation and finds that the semaphore value is not positive, it must wait. However, rather than busy waiting, the process can block itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then, control is transferred to the CPU scheduler, which selects another process to execute.

A process that is blocked, waiting on a semaphore S, should be resorted when some other process executes a signal operation. The process is restarted by a wake up operation, which changes the process from the waiting state to the ready state.

We define a semaphore as a record:

```

type semaphore = record
    value: integer;
    L: list of process;
end;

```

When a process must wait on a semaphore, it is added to the list of processes.

```

wait(S):    S.value := S.value — 1;
            if S.value < 0
            then begin
                add this process to S.L;
                block;
            end;

```

```

signal(S): S.value := S.value + 1;
            if S.value < 0
            then begin
                remove a process P from S.L;
                wakeup(P);
            end;

```

The block operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls.

If the semaphore value is negative, its magnitude is the number of processes waiting on that semaphore. This fact is a result of the switching of the order of the decrement and the test in the implement of the wait operation. The critical aspect of semaphores is that they are executed atomically.

Deadlocks and Starvation

The implementation of a semaphore with a waiting queue may result in situation where two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes. The event in execution of a signal operation. When such a state is reached, these processes are said to be deadlocked.

Another problem related to deadlocks is indefinite blocking or starvation, a situation where processes wait indefinitely within the semaphore. Indefinite blocking may occur if we add and remove processes from the list associated with a semaphore in LIFO order

Binary Semaphores

A binary semaphore is a semaphore with an integer value that can range only between 0 and 1. A binary semaphore can be simpler to implement than a counting semaphore, depending on the underlying hardware architecture.

Let S be a counting semaphore. To implement it in terms of binary semaphores we need the following data structures:

```
var   S1: binary-semaphore;  
      S2: binary-semaphore;  
      S3: binary-semaphore;  
      C: integer;
```

initially $S1 = S3 = 1$, $S2 = 0$, and the value of integer C is set to the initial value of the counting semaphore S.

The wait operation on the counting semaphore S can be implemented as follows

```
wait(S3);  
wait(S1);  
C := C - 1;  
if C < 0  
  then begin  
    signal(S1);  
    wait(S2);  
  end  
  else signal(S1);  
       signal(S3);
```

The signal operation on the counting semaphore S can be implemented as follows:

```
wait(S1);  
C:=C + 1;  
if C ≤ 0 then signal(S2);  
signal(S1);
```

The S3 semaphore has no effect on signal(S), it merely serializes the wait(S) operations.

1 Interprocess Communication

Cooperating processes can communicate in a shared-memory environment. The scheme requires that these processes share a common buffer pool, and that the code for implementing the buffer be explicitly written by the application programmer. Another way to achieve the same effect is for the operating system -to provide the means for cooperating processes to communicate with each other via an inter-process-communication (IPC) facility.

IPC provides a mechanism to allow processes to communicate and to synchronize their actions. Inter-process-communication is best provided by a message system. Message systems can be defined in many different ways.

Shared-memory and message-system communication schemes are not mutually exclusive, and could be housed simultaneously within a single operating system or even a single process.

Basic Structure

The function of a message system is to allow processes to communicate with each other without the need to resort to shared variables. An IPC facility provides at least the two operations: `send(message)` and `receive(message)`.

Messages sent by a process can be of either fixed or variable size. If only fixed sized messages can be sent, the physical implementation is straight forward. Variable-sized messages require a more complex physical implementation, but the programming task becomes simpler

If processes P and Q want to communicate, they must send messages to and receive messages from each other; a communication link must exist between them. This link can be implemented in a variety of ways

A link is unidirectional only if each process connected to the link can either send or receive, but not both and each link has at least one receiver process connected to it. In addition, there are several methods for logically implementing a link and the send/receive operations

- Direct or indirect communication
- Symmetric or asymmetric communication
- Automatic or explicit buffering
- Send by copy or send by reference
- Fixed-sized or variable-sized messages.

Naming

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

Direct Communication

In the direct-communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the **send and receive** primitives are defined as follows:

`Send (P, message)`. Send a message to process P.
`receive(Q, message)`. Receive a message from process Q.

A communication link has the following properties:

- A link is established automatically between every pair of processes that want to communicate, the processes need to know only each other's identity to communicate:
- A link is associated with exactly two processes.
- Between each pair of processes, there exists exactly one link.;

This scheme exhibits a symmetry in addressing; that is, both the sender and the receiver processes have to name each other to communicate. A variant of this scheme employs asymmetry in addressing. Only the sender names the recipient; the recipient is not required to name the sender. In this scheme, the send and receive primitives are defined as follows:

- `send(P, message)`. Send a message to process P.
- `received, message`. Receive a message from any process; the variable `id` is set to the name of the process with which communication has taken place.

Indirect Communication

With indirect communication, the messages are sent to and received from mailboxes. A mailbox can be viewed abstractly as, an object into which messages can be placed by processes and from which messages can be removed. The send and receive primitives are defined as follows:

- `send (A, message)`. Send a message to mailbox A.
- `received (A, message)`. Receive a message from mailbox A.

In this scheme, a communication link has the following properties:

- A link is established between a pair of processes only if they have a shared mailbox.
- A link may be associated with more than two processes.
- Between each pair of communicating processes, there may be a number of different links, each link corresponding to one mailbox.
- A link may be either unidirectional or bidirectional.

Buffering

A link has some capacity that determines the number of messages that can reside in it temporarily. This property can be viewed as a queue of messages attached to the link.

- **Zero capacity:** The queue has maximum length 0; thus, the link cannot have any messages waiting in it
- **Bounded capacity:** The queue has finite length n ; thus, at most n messages can reside in it.

- **Unbounded capacity:** The queues has potentially infinite length; thus, any number of messages can wait in it.

Exception Conditions

A message system is particularly useful in a distributed environment, where processes may reside at different sites. In such an environment the probability that an error will occur during communication and processing is much larger than in a single-machine environment. In a single-machine environment, messages are usually implemented in shared memory. If a failure occurs, the entire system fails, In a distributed environment, however, message are transferred by communication lines, and the failure of one site (or link) does not necessarily result in the failure of the entire system.

When a failure occurs in either a centralized or distributed system, some error recovery (exception-condition handling) must take place. Let us discuss briefly some of the exception conditions that a system must handle in the context of a message scheme.

Process Terminates

Either a sender or a receiver may terminate before a message is processed. This situation will leave messages that will never be received or processes waiting for messages that will never be sent.

A receiver process P may wait for a message from a process Q that has terminated. If no action is taken, P will be blocked forever. In this case, the system may either terminate P or notify P that Q has terminated.

Lost Messages

A message from process P to process Q may become lost somewhere in the communications network, due to a hardware or communication-line failure. There are three basic methods for dealing with this event:

1. The operating system is responsible for detecting this event and for resending the message.
2. The sending process is responsible for detecting this event and for retransmitting the message, if it wants to do so.
3. The operating system is responsible for detecting this event; it then notifies the sending process that the message has been lost. The sending process can proceed as it chooses.

Scrambled Messages:

The message may be delivered to its destination, but be scrambled on the way This case is similar to the case of a lost message. Usually, the operating system will retransmit the original message. Error checking codes are commonly used to detect this type of error.

In any O.S, process and threads are managed to have:

- **Multiprogramming:** The management of multiple processes within a uni-processor system.
- **Multiprocessing:** The management of multiple processes within a multiprocessor.
- **Distributed processing:** The management of multiple processes executing on multiple, distributed computer systems.

Concurrency including communication among processes, sharing of and competing for resources, synchronization of the activities of multiple processes, and allocation of processor time to processes. Concurrency arises in three different contexts:

- **Multiple applications:** Multiprogramming was invented to allow processing time to be dynamically shared among a number of active application.
- **Structured applications:** As an extension of the principles of modular design and structured programming, some applications can be effectively programmed as a set of concurrent processes.
- **Operating system structure:** The same structuring advantages apply to the systems programmer, and operating systems are themselves often implemented as a set of processes or threads.

Principle of concurrency

In a single-processor multiprogramming system, processes are interleaved in time to yield the appearance of simultaneous execution. Even parallel processing is not achieved, and even though there is a certain amount of overhead involved in switching back and forth between processes, interleaved execution provides major benefits in processing efficiency and in program structuring. In a multiple-processor system, it is possible not only to interleave the execution of multiple processes but also to overlap them. It is assumed, it may seem that interleaving and overlapping represent fundamentally different modes of execution and present different problems. In fact, both techniques can be viewed as examples of concurrent processing, and both present the same problems. The relative speed of execution of processes It depends on activities of other processes, the way in which the operating system handles interrupts, and the scheduling policies of the operating, system.

There are quite difficulties:

1. The sharing of global resources. For example, if two processes both make use of the same global variable and both perform reads and writes on that variable, then the order in which the various reads and writes are executed is critical.
2. It is difficult for the operating system to manage the allocation of resources optimally.
3. It is very difficult to locate a programming error because results are typically not deterministic and reproducible.

Eg:

```
Void echo()
{
    chin = getchar();
    chout = chin;
    putchar(chout);
}
```

This procedure shows the essential elements of a program that will provide a character echo procedure; input is obtained from a keyboard one keystroke at a time. Each input character is stored in variable chin. It is then transferred to variable chout and sent to the display. Any program can call this procedure repeatedly to accept user input and display it on the user's screen.

In a single - processor multiprogramming system supporting a single user. The user can jump from one application to another, and each application uses the same keyboard for input and the same screen for output. Because each application needs to use the procedure echo, it makes sense for it to be a shared procedure that is loaded into a portion of memory global to all applications. Thus, only a single copy of the echo procedure is used, saving space.

The sharing of main memory among processes is useful to permit efficient and close interaction among processes Consider the following sequence:

1. Process P1 invokes the echo procedure and is interrupted immediately after getchar returns its value and stores it in chin. At this point, the most recently entered character, x, is stored in variable chin.
2. Process P2 is activated and invokes the echo procedure, which runs to conclusion, inputting and then displaying a single character, y, on the screen.

3. Process P1 is resumed. By this time, the value x has been overwritten in chin and therefore lost. Instead, chin contains y, which is transferred to chout and displayed.

Thus, the first character is lost and the second character is displayed twice. Because of shared global variable, chin. If one process updates the global variable and then is interrupted, another process may alter the variable before the first process can use its value. However, if only one process at a time may be in that procedure. Then the foregoing sequence would result in the following:

1. Process P1 invokes the echo procedure and is interrupted immediately after the conclusion of the input function. At this point, the most recently entered character, x, is stored in variable chin.
2. Process P2 is activated and invokes the echo procedure. However, because P1 is still inside the echo procedure, although currently suspended, P2 is blocked from entering the procedure. Therefore, P2 is suspended awaiting the availability of the echo procedure.
3. At some later time, process P1 is resumed and completes execution of echo. The proper character, x, is displayed.
4. When P1 exits' echo, this removes the block on P2. When P2 is later resumed, the echo procedure is successfully invoked.

Therefore it is necessary to protect shared global variables. And that the only way to do that is to control the code that accesses the variable.

Race Condition :

A race condition occurs when multiple processes or threads read and write data items so that the final result depends on the order of execution of instructions in the multiple processes.

Suppose that two processes, P1 and P2, share the global variable a. At some point in its execution, P1 updates a to the value 1, and at some point in its execution, P2 updates a to the value 2. Thus, the two tasks are in a race to write variable a. In this example the "loser" of the race (the process that updates last) determines the final value of a.

Therefore Operating System Concerns of following things

1. The operating system must be able to keep track of the various processes
2. The operating system must allocate and deallocate various resources for each active process.

3. The operating system must protect the data and physical resources of each process against unintended interference by other processes.
4. The functioning of a process, and the output it produces, must be independent of the speed at which its execution is carried out relative to the speed of other concurrent processes.

Process Interaction can be defined as

- **Processes unaware of each other**
- **Processes indirectly aware of each other**
- **Processes directly aware of each other**

Concurrent processes come into conflict with each other when they are competing for the use of the same resource.

Two or more processes need to access a resource during the course of their execution. Each process is unaware of the existence of the other processes. There is no exchange of information between the competing processes. However, the execution of one process may affect the behavior of competing processes. If two processes both wish access to a single resource, then one process will be allocated that resource by the operating system, and the other will have to wait.

In the case of competing processes three control problems must be faced. First is the need for **mutual exclusion**. Suppose two or more processes require access to a single nonsharable resource, such as a printer. During the course of execution, each process will be sending commands to the I/O device-receiving status information, sending data, and/or receiving data. Such a resource as a **critical resource**, and the portion of the program that uses it a **critical section** of the program. It is important that only one program at a time be allowed in its critical section.

The enforcement of mutual exclusion creates two additional control problems. One is that of **deadlock**. For example, consider two processes, P1 and P2, and two resources, R1 and R2. Suppose that each process needs access to both resources to perform part of its function. Then it is possible to have the following situation: the operating system assigns R1 to P2, and R2 to P1. Each process is waiting for one of the two resources. Neither will release the resource that it already owns until it has acquired the other resource and performed the function requiring both resources. The two processes are deadlocked.

A final control problem is **starvation**. Suppose that three processes (P1, P2, P3) each require periodic access to resource R. Consider the situation in which P1 is in possession of the resource, and both P2 and P3 are delayed, waiting for that

resource. When P1 exits its critical section, either P2 or P3 should be allowed access to R. Assume that the operating system grants access to P3 and that P1 again requires access before completing its critical section. If the operating system grants access to P1 after P3 has finished, and subsequently alternately grants access to P1 and P3, then P2 may indefinitely be denied access to the resource, even though there is no deadlock situation.

If there are n processes to be executed concurrently. Each process includes (1) a critical section that operates on some resource and (2) additional code preceding and following the critical section that does not involve access to Resource because all processes access the same resource Ra, it is desired that only one process at a time be in its critical section. To enforce mutual exclusion, two functions are provided: entercritical and exitcritical.

Cooperation among Processes by Sharing

The case of cooperation by sharing covers processes that interact with other processes without being explicitly aware of them. For example, multiple processes may have access to shared variables or to shared files or databases. Processes may use and update the shared data without reference to other processes but know that other processes may have access to the same data. Thus the processes must cooperate to ensure that the data they share are properly managed. The control mechanisms must ensure the integrity of the share data.

Because data are held on resources (devices, memory), the control problems of mutual exclusion, deadlock, and starvation are again present.

Example, consider a bookkeeping application in which various data items may be updated. Suppose two items of data a and b are to be maintained in the relationship $a = b$. That is, any program that updates one value must also update the other to maintain the relationship. Now consider the following two processes:

```
P1:
    a = a + 1;
    b = b + 1;
P2:
    b = 2*b;
    a = 2*a;
```

If the state is initially consistent, each process taken separately will leave the shared data in a consistent state. Now consider the following concurrent execution, in which the two processes respect mutual exclusion on each individual data item (a and b) :

```
a = a+1;
b = 2*b;
```

```
b = b+1;  
a = 2*a;
```

At the end of this execution sequence, the condition $a = b$ no longer holds.

Cooperation among Processes by Communication:

In the case of competition, process resources without being aware of the other processes. and, they are sharing values, and although each process is not explicitly aware of the other processes, it is aware of the need to maintain data integrity. When processes cooperate by communication, however, the various processes participate in a common effort that links all of the processes. The communication provides a way to synchronize, or coordinate, the various activities.

Requirements for Mutual- Exclusion :

1. Mutual exclusion must be enforced: Only one process at a time is allowed into its critical section, among all processes that have critical sections for the same resource or shared object.
2. A process that halts in its non critical section must do so without interfering with other processes.
3. It must not be possible for a process requiring access to a critical section to be delayed indefinitely: no deadlock or starvation.
4. When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay.
5. No assumptions are made about relative process speeds or number of processors.
6. A process remains inside its critical section for a finite time only.

Mutual exclusion: Hardware Support

Hardware approaches to mutual exclusion.

1. Interrupt Disabling:

In a uniprocessor machine, concurrent processes cannot be overlapped; they can only be interleaved. Furthermore, a process will continue to run until it invokes an operating system service or until it is interrupted. Therefore, to guarantee mutual exclusion, it is sufficient to prevent a process from being interrupted. This capability can be provided in the form of primitives defined by the system kernel for disabling and enabling interrupts.

eg:

```
while (true)  
(
```

```

        /* disable interrupts*/;
        /* critical section*/;
        /* enable interrupts*/;
        /* remainder */
    )

```

Because the critical section cannot be interrupted, mutual exclusion is guaranteed.

2. Special Machine Instructions

In a multiprocessor configuration, several processors share access to a common main memory. In this case, there is not a master/slave relationship; rather the processors behave independently in a peer relationship. There is no interrupt mechanism between processors on which mutual exclusion can be based.

At a hardware level, access to a memory location excludes any other access to that same location. Processor designers have several machine instructions that carry out two actions atomically, such as reading and writing or reading and testing, of a single memory-location with one instruction fetch cycle. During execution of the instruction, access to the memory location is blocked for any other instruction referencing that location. Typically, these actions are performed in a single instruction cycle.

Test and Set Instruction: The test and set instruction can be defined as follows:

```

boolean testset (int i)
{
    if (i==0)
    {
        i=1;
        return true;
    }
    else, .
    {
        return false;
    }
}

```

The instruction tests the value of its argument *i*. If the value is 0, then the instruction replaces the value by 1 and returns true. Otherwise, the value is not changed and false is returned. A mutual exclusion protocol based on the use of this instruction. The construct `parbegin (P1, P2, . . . , Pn)` means the following: suspend the execution of the main program; initiate concurrent execution of procedures P1, P2, . . . , Pn; when all of P1, P2, . . . , Pn have terminated, resume the main program. A shared variable `bolt` is initialized to 0. The only process that may enter its critical section is one that finds `bolt` equal to 0.

3. Exchange Instruction The exchange instruction can be defined as follows:

```

void exchange (int register, int memory)

```

```

{
    int    temp;
    temp = memory;
    memory = register;
    register = temp;
}

```

The instruction exchanges the contents of a register with that of a memory location. A shared variable bolt is initialized to 0. Each process uses a local variable key that is initialized to 1. The only process that may enter its critical section is one that finds bolt equal to 0. It excludes all other processes from the critical section by setting bolt to 1. When a process leaves its critical section, it resets bolt to 0, allowing another process to gain access to its critical section. Exchange algorithm:

$$\text{bolt} + \sum \text{key}_i = n$$

If bolt = 0, then no process is in its critical section. If bolt = 1, then exactly one process is in its critical section, namely the process whose key value equals 0.

4. Properties of the Machine-Instruction Approach The use of a special machine instruction to enforce mutual exclusion has a number of advantages:

- It is applicable to any number of processes on either a single processor or multiple processors sharing main memory.
- It is simple and therefore easy to verify.
- It can be used to support multiple critical sections; each critical section can be defined by its own variable.

Disadvantages:

- **Busy waiting is employed.**
- **Starvation is possible.**
- **Deadlock is possible.**

Monitors

Semaphores provide a primitive yet powerful and flexible tool for enforcing mutual exclusion and for coordinating processes. It may be difficult to produce a correct program using semaphores. The difficulty is that semWait and semSignal operations may be scattered throughout a program and it is not easy to see the overall effect of these operations on the semaphores they affect.

The monitor is a programming-language construct that provides equivalent functionality to that of semaphores and that is easier to control. The monitor construct has been implemented in a number of programming languages, including Concurrent Pascal, Pascal-Plus, Modula-2, Modula-3, and Java. It has also been implemented as a program library. This allows programmers to put monitor locks on any object.

Monitor with Signal

A monitor is a software module consisting of one or more procedures, an initialization sequence, and local data. The characteristics of a monitor are the following:

1. The local data variables are accessible only by the monitor's procedures and not by any external procedure.
2. A process enters the monitor by invoking one of its procedures.
3. Only one process may be executing in the monitor at a time; any other process that has invoked the monitor is blocked, waiting for the monitor to become available.

A monitor supports synchronization by the use of **condition variables** that are contained within the monitor and accessible only within the monitor. Condition variables are a special data type in monitors, which are operated on by two functions:

- **cwait (c)**: Suspend execution of the calling process on condition c. The monitor is now available for use by another process.
- **csignal (c)**: Resume execution of some process blocked after a cwait on the same condition. If there are several such processes, choose one of them; if there is no such process, do nothing.

Monitor wait and signal operations are different from those for the semaphore. If a process in a monitor signals and no task is waiting on the condition variable, the signal is lost.

Although a process can enter the monitor by invoking any of its procedures, we can think of the monitor as having a single entry point that is guarded so that only one process may be in the monitor at a time. Other processes that attempt to enter the monitor join a queue of processes blocked waiting for monitor availability. Once a process is in the monitor, it may temporarily block itself on condition x by issuing **cwait (x)**; it is then placed in a queue of processes waiting to reenter the monitor when the condition changes, and resume execution at the point in its program following the **cwait (x)** call.

If a process that is executing in the monitor detects a change in condition variable x, it issues **csignal (x)**, which alerts the corresponding condition queue that the condition has changed.

A producer can add characters to the buffer only by means of the procedure **append** inside the monitor; the producer does not have direct access to buffer. The procedure first checks the condition **notfull** to determine if there is space available in the buffer. If not, the process executing the monitor is blocked on that condition.

Mutual Exclusion: Software Approaches

Software approaches can be implemented for concurrent processes that execute on a single processor or a multiprocessor machine with shared main memory. These approaches usually assume elementary mutual exclusion at the memory access level. That is, simultaneous accesses (reading and /or writing) to the same location in main memory are serialized by some sort of memory

Dekker's Algorithm

Dijkstra reported an algorithm for mutual exclusion for two processes.

1. As mentioned earlier, any attempt at mutual exclusion must rely on some fundamental exclusion mechanism in the hardware. The most common of these is the constraint that only one access to a memory location can be made at a time. We reserve a global memory location labeled turn. A process (P0 or P1) wishing to execute its critical section first examines the contents of turn. If the value of turn is equal to the number of the process, then the process may proceed to its critical section. Otherwise, it is forced to wait. Our waiting process repeatedly reads the value of turn until it is allowed to enter its critical section. This procedure is known as **busy waiting**, or **spin waiting**.
2. The problem with the first attempt is that it stores the name of the process that may enter its critical section, when in fact we need state information about both processes. In effect, each process should have its own key to the critical section so that if one fails, the other can still access its critical section. To meet this requirement a Boolean vector flag is defined, with flag [0] corresponding to P0 and flag [1] corresponding to P1. Each process may examine the other's flag but may not alter it. When a process wishes to enter its critical section, it periodically checks the other's flag until that flag has the value false, indicating that the other process is not in its critical section. The checking process immediately sets its own flag to true and proceeds to its critical section. When it leaves its critical section, it sets its flag to false.
3. Because a process can change its state after the other process has checked it but before the other process can enter its critical section, the second attempt failed. If mutual exclusion is guaranteed, using the point of view of process P0. Once P0 has set flag [0] to true, P1 cannot enter its critical section until after P0 has entered and left its critical section. It could be that P1 is already in its critical section when P0 sets its flag. In that case, P0 will be blocked by the while statement until P1 has left its critical section.

In the third attempt, a process sets its state without knowing the state of the other process. Deadlock occurs because each process can insist on its right to enter its critical section; there is no opportunity to back off from this position.

Peterson's Algorithm

Dekker's algorithm solves the mutual exclusion problem but with a rather complex program that is difficult to follow and whose correctness is tricky to prove. The global array variable flag indicates the position of each process with respect to mutual exclusion, and the global variable turn resolves simultaneity conflicts

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

1. P1 has no interest in its critical section. This case is impossible, because it implies flag[1] = false.
2. P1 is waiting for its critical section. This case is also impossible, because if turn = 1, P1 is able to enter its critical section.
3. P1 is using its critical section repeatedly and therefore monopolizing access to it. This cannot happen, because P1 is obliged to give P0 an opportunity by setting turn to 0 before each attempt to enter its critical section.

CHAPTER- 4 **DEADLOCKS**

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a wait state. It may happen that waiting processes will never again change state, because the resources they have requested are held by other waiting processes.

If a process requests an instance of a resource type, the allocation of any instance of the type will satisfy the request. If it will not, then the instances are not identical, and the resource type classes have not been defined properly.

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

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

1. **Request:** If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.
2. **Use:** The process can operate on the resource.
3. **Release:** The process releases the resource

Deadlock Characterization

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

Necessary Conditions

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

1. **Mutual exclusion:** At least one resource must be held in a nonsharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. **Hold and wait :** There must exist a process that is holding at least one resource and is waiting to acquire additional resources that are currently being held by other processes.
3. **No preemption :** Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process, has completed its task.
4. **Circular wait:** There must exist a set $\{P_0, P_1, \dots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1 , P_1 is waiting for a resource that is held by P_2 , ..., P_{n-1} is waiting for a resource that is held by P_n , and P_n is waiting for a resource that is held by P_0 .)

Resource-Allocation Graph

Deadlocks can be described more precisely in terms of a directed graph called a system resource-allocation graph. The set of vertices V is partitioned into two different types of nodes $P = \{P_1, P_2, \dots, P_n\}$ the set consisting of all the active processes in the system; and $R = \{R_1, R_2, \dots, R_k\}$, the set consisting of all resource types in the system.

A directed edge from process P_i to resource type R_j is denoted by $P_i \rightarrow R_j$, it signifies that process P_i requested an instance of resource type R_j and is currently waiting for that resource. A directed edge from resource type R_j to process P_i is denoted by $R_j \rightarrow P_i$ it signifies that an instance of resource type R_j has been allocated to process P_i . A directed edge $P_i \rightarrow R_j$ is called a request edge; a directed edge $R_j \rightarrow P_i$ is called an assignment edge.

When process P_i requests an instance of resource type R_i , a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is instantaneously transformed to an assignment edge. When the process no longer needs access to the resource it releases the resource, and as a result the assignment edge is deleted.

Definition of a resource-allocation graph, it can be shown that, if the graph contains no cycles, then no process in the system is deadlocked. If, on the other hand, the graph contains the cycle, then a deadlock must exist.

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

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

Suppose that process P_3 requests an instance of resource type R_2 . Since no resource instance is currently available, a request edge $P_3 \rightarrow R_2$ is added to the graph. At this point, two minimal cycles exist in the system:

$$\begin{aligned} P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1 \\ P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2 \end{aligned}$$

Processes P_1 , P_2 , and P_3 are deadlocked. Process P_2 is waiting for the resource R_3 , which is held by process P_3 . Process P_3 , on the other hand, is waiting for either process P_1 or process P_2 to release resource R_2 . In addition, process P_1 is waiting for process P_2 to release resource R_1 .

Methods for Handling Deadlocks

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

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

Deadlock avoidance, on the other hand, requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, we can

decide for each request whether or not the process should wait. Each request requires that the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process, to decide whether the current request can be satisfied or must be delayed.

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

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

Deadlock Prevention

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

Mutual Exclusion

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

Hold and Wait

1. When whenever a process requests a resource, it does not hold any other resources. One protocol that be used requires each process to request and be allocated all its resources before it begins execution.
2. An alternative protocol allows a process to request resources only when the process has none. A process may request some resources and use them. Before it can request any additional resources, however it must release all the resources that it is currently allocated

here are two main disadvantages to these protocols. First, resource utilization may be low, since many of the resources may be allocated but unused for a long period. In the example given, for instance, we can release the tape drive" and disk file, and then again request the disk file and printer, only if we can be sure that our data will remain on the disk file. If we cannot be assured that they will, then we must request all resources at the beginning for both protocols.

Second, starvation is possible.

No Preemption

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

Circular Wait

Circular-wait condition never holds if we impose a total ordering of all resource types, and to require that each process requests resources in an increasing order of enumeration.

Let $R = \{R_1, R_2, \dots, R_n\}$ be the set of resource types. We assign to each resource type a unique integer number, which allows us to compare two resources and to determine whether one precedes another in our ordering. Formally, we define a one-to-one function $F: R \rightarrow N$, where N is the set of natural numbers.

Deadlock Avoidance

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

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

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

Safe State

A state is safe if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence. A sequence of processes $\langle P_1, P_2, \dots, P_n \rangle$ is a safe sequence for the current allocation state if, for each P_i the resources that P_i can still request can be satisfied by the currently available

resources plus the resources held by all the P_j , with $j < i$. In this situation, if the resources that process P_i needs are not immediately available, then P_i can wait until all P_j have finished. When they have finished, P_i can obtain all of its needed resources, complete its designated task return its allocated resources, and terminate. When P_i terminates, P_{i+1} can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be unsafe.

Resource-Allocation Graph Algorithm

Suppose that process P_i requests resource R_j . The request can be granted only if converting the request edge $P_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow P_i$ does not result in the formation of a cycle in the resource-allocation graph.

Banker's Algorithm

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

Deadlock Detection

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

- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock

Single Instance of Each Resource Type

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

Several Instances of a Resource Type

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

The algorithm used are :

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

- **Allocation:** An $n \times m$ matrix defines the number of resources of each type currently allocated to each process.
- **Request:** An $n \times m$ matrix indicates the current request of each process. If Request $[i, j] = k$, then process P_i is requesting k more instances of resource R_j .

Detection-Algorithm Usage

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

Recovery from Deadlock

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

Process Termination

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

- **Abort all deadlocked processes:** This method clearly will break the dead – lock cycle, but at a great expense, since these processes may have computed for a long time, and the results of these partial computations must be discarded, and probably must be recomputed
- **Abort one process at a time until the deadlock cycle is eliminated:** This method incurs considerable overhead, since after each process is aborted a deadlock-detection algorithm must be invoked to determine whether a processes are still deadlocked.

Resource Preemption

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

The three issues are considered to recover from deadlock

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

CHAPTE 5 MEMORY MANAGEMENT

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

A program resides on a disk as a binary executable file. The program must be brought into memory and placed within a process for it to be executed. Depending on the memory management in use the process may be moved between disk and memory during its execution. The collection of processes on the disk that are waiting to be brought into memory for execution forms the input queue. i.e. selected one of the process in the input queue and to load that process into memory.

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

- **Compile time:** If it is known at compile time where the process will reside in memory, then absolute code can be generated.
- **Load time:** If it is not known at compile time where the process will reside in memory, then the compiler must generate re-locatable code.
- **Execution time:** If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time.

Dynamic Loading

Better memory-space utilization can be done by dynamic loading. With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a re-locatable load format. The main program is loaded into memory and is executed.

The advantage of dynamic loading is that an unused routine is never loaded.

Dynamic Linking

Most operating systems support only static linking, in which system language libraries are treated like any other object module and are combined by the loader into the binary program image. The concept of dynamic linking is similar to that of dynamic loading. Rather than loading being postponed until execution time, linking is postponed. This feature is usually used with system libraries, such as language subroutine libraries. With dynamic linking, a stub is included in the image for each library-routine reference. This stub is a small piece of code that indicates how to locate the appropriate memory-resident library routing.

Overlays

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

Example, consider a two-pass assembler. During pass 1, it constructs a symbol table; then, during pass 2, it generates machine-language code. We may be able to partition such an assembler into pass 1 code, pass 2 code, the symbol table¹, and common support routines used by both pass 1 and pass 2.

Let us consider

Pass1	70K
Pass 2	80K
Symbol table	20K
Common routines	30K

To load everything at once, we would require 200K of memory. If only 150K is available, we cannot run our process. But pass 1 and pass 2 do not need to be in memory at the same time. We thus define two overlays: Overlay A is the symbol table, common routines, and pass 1, and overlay B is the symbol table, common routines, and pass 2.

We add an overlay driver (10K) and start with overlay A in memory. When we finish pass 1, we jump to the overlay driver, which reads overlay B into memory, overwriting overlay A, and then transfers control to pass 2. Overlay A needs only 120K, whereas overlay B needs 130K

As in dynamic loading, overlays do not require any special support from the operating system.

Logical versus Physical Address Space

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

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

addresses corresponding to these logical addresses is referred to as a physical address space.

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

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

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

The user program deals with logical addresses. The memory-mapping hardware converts logical addresses into physical addressed

Logical addresses (in the range 0 to max) and physical addresses (in the range $R + 0$ to $R + \text{max}$ for a base value R). The user generates only logical addresses.

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

Swapping

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

A variant of this swapping policy is used for priority-based scheduling algorithms. If a higher-priority process arrives and wants service, the memory manager can swap out the lower-priority process so that it can load and execute the higher-priority process. When the higher priority process finishes, the lower-priority process can be swapped back in and continued. This variant of swapping is sometimes called rollout, roll in.

A process is swapped out will be swapped back into the same memory space that it occupies previously. If binding is done at assembly or load time, then the

process cannot be moved to different location. If execution-time binding is being used, then it is possible to swap a process into a different memory space.

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

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

$$\begin{aligned} 100\text{K} / 1000\text{K per second} &= 1/10 \text{ second} \\ &= 100 \text{ milliseconds} \end{aligned}$$

Contiguous Allocation

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

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

Single-Partition Allocation

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

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

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

Multiple-Partition Allocation

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

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

For example, assume that we have 2560K of memory available and a resident-operating system of 400K. This situation leaves 2160K for user processes. FCFS job scheduling, we can immediately allocate memory to processes P_1 , P_2 , P_3 . Holes size 260K that cannot be used by any of the remaining processes in the input queue. Using a round-robin CPU-scheduling with a quantum of 1 time unit, process will terminate at time 14, releasing its memory.

Memory allocation is done using Round-Robin Sequence as shown in fig.

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

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

- First-fit: Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
- Best-fit: Allocate the smallest hole that is big enough. We must search the entire list, unless the list is kept ordered by size. This strategy-produces the smallest leftover hole.

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

External and Internal Fragmentation

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

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

Given N allocated blocks, another $0.5N$ blocks will be lost due to fragmentation. That is, one-third of memory may be unusable. This property is known as the 50-percent rule.

Internal fragmentation - memory that is internal to partition, but is not being used.

Paging

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

The page size like is defined by the hardware. The size of a page is typically a power of 2 varying between 512 bytes and 8192 bytes per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset. If the size of logical address space is 2^m , and a page size is 2^n addressing units (bytes or words), then the high-order $m - n$ bits of a logical address designate the page number, and the n low-order bits designate the page offset. Thus, the logical address is as follows:

page number

page	offset
p	d
$m - n$	n

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

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

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

If process size is independent of page size, we can have internal fragmentation to average one-half page per process.

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

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

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

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

Segmentation:

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

Because of the use of unequal-size segments, segmentation is similar to

dynamic partitioning. In segmentation, a program may occupy more than one partition, and these partitions need not be contiguous. Segmentation eliminates internal fragmentation but, like dynamic partitioning, it suffers from external fragmentation. However, because a process is broken up into a number of smaller pieces, the external fragmentation should be less. Whereas paging is invisible to the programmer, segmentation usually visible and is provided as a convenience for organizing programs and data.

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

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

- Extract the segment number as the leftmost n bits of the logical address.
- Use the segment number as an index into the process segment table to find the starting physical address of the segment.
- Compare the offset, expressed in the rightmost m bits, to the length of the segment. If the offset is greater than or equal to the length, the address is invalid.
- The desired physical address is the sum of the starting physical address of the segment plus the offset.

Segmentation and paging can be combined to have a good result.

Chapter 6 **Virtual Memory**

VIRTUAL MEMORY

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

Virtual memory is the separation of user logical memory from physical memory this separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available.

Virtual memory is commonly implemented by demand paging. It can also be implemented in a segmentation system. Demand segmentation can also be used to provide virtual memory.

Demand Paging

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

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

Hardware support is required to distinguish between those pages that are in memory and those pages that are on the disk using the valid-invalid bit scheme.

Where valid and invalid pages can be checked checking the bit and marking a page will have no effect if the process never attempts to access the pages. While the process executes and accesses pages that are memory resident, execution proceeds normally.

Access to a page marked invalid causes a page-fault trap. This trap is the result of the operating system's failure to bring the desired page into memory.

But page fault can be handled as following :

1. We check an internal table for this process to determine whether the reference was a valid or invalid memory access.
2. If the reference was invalid, we terminate the process. If it was valid, but we have not yet brought in that page, we now page in the latter
3. We find a free frame
4. 4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the illegal address trap. The process can now access the page as though it had always been memory.

Therefore, the operating system reads the desired page into memory and restarts the process as though the page had always been in memory.

The page replacement is used to make the frame free if they are not in used. If no frame is free then other process is called in.

Page Replacement Algorithm

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

1. For a given page size we need to consider only the page number, not the entire address.
2. if we have a reference to a page p, then any immediately following references to page p will never cause a page fault. Page p will be in memory after the first reference; the immediately following references will not fault.

Eg:- consider the address sequence

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,
0104, 0101, 0610, 0102, 0103, 0104, 0104, 0101, 0609, 0102, 0105 and
reduce to 1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

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

FIFO Algorithm

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

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

Optimal Algorithm

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

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

Now consider the same string with 3 empty frames.

The reference to page 2 replaces page 7, because 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14. The reference

to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. Optimal replacement is much better than a FIFO.

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

LRU Algorithm

The FIFO algorithm uses the time when a page was brought into memory; the OPT algorithm uses the time when a page is to be used. In LRU replace the page that has not been used for the longest period of time.

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

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

LRU Approximation Algorithms

Some systems provide no hardware support, and other page-replacement algorithm. Many systems provide some help, however, in the form of a reference bit. The reference bit for a page is set, by the hardware, whenever that page is referenced. Reference bits are associated with each entry in the page table

Initially, all bits are cleared (to 0) by the operating system. As a user process executes, the bit associated with each page referenced is set (to 1) by the hardware.

Additional-Reference-Bits Algorithm

The operating system shifts the reference bit for each page into the high-order or of its 8-bit byte, shifting the other bits right 1 bit, discarding the low-order bit. These 8-bit shift registers contain the history of page use for the last eight time periods. If the shift register contains 00000000, then the page has not been used for eight time periods; a page that is used at least once each period would have a shift register value of 11111111.

Second-Chance Algorithm

The basic algorithm of second-chance replacement is a FIFO replacement algorithm. When a page gets a second chance, its reference bit is cleared and its arrival time is reset to the current time.

Enhanced Second-Chance Algorithm

The second-chance algorithm described above can be enhanced by considering both the reference bit and the modify bit as an ordered pair.

1. (0,0) neither recently used nor modified — best page to replace
2. (0,1) not recently used but modified — not quite as good, because the page will need to be written out before replacement
3. (1,0) recently used but clean — probably will be used again soon
4. (1,1) recently used and modified — probably will be used again, and write out will be needed before replacing it

Counting Algorithms

There are many other algorithms that can be used for page replacement.

- **LFU Algorithm:** The least frequently used (LFU) page-replacement algorithm requires that the page with the smallest count be replaced. This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again.
- **MFU Algorithm:** The most frequently used (MFU) page-replacement algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

Page Buffering Algorithm

When a page fault occurs, a victim frame is chosen as before. However, the desired page is read into a free frame from the pool before the victim is written out. This procedure allows the process to restart as soon as possible, without waiting for the victim page to be written out. When the victim is later written out, its frame is added to the free-frame pool.

When the FIFO replacement algorithm mistakenly replaces a page mistakenly replaces a page that is still in active use, that page is quickly retrieved from the free-frame buffer, and no I/O is necessary. The free-frame buffer provides protection against the relatively poor, but simple, FIFO replacement algorithm.

Chapter -7 I/O SYSTEMS

I/O with computer systems can be roughly grouped into three categories:

- **Human readable:** Suitable for communicating with the computer user. Examples include printers and video display terminals, the latter consisting of display, keyboard, and perhaps other devices such as a mouse.

- **Machine readable:** Suitable for communicating with electronic equipment. Examples are disk and tape drives, sensors, controllers, and actuators.
- **Communication:** Suitable for communicating with remote devices. Examples are digital line drivers and modems. But they differ in the form of application.
- **Data rate:** There may be differences of several orders of magnitude between the data transfer rate.
- **Application:** The use to which a device is put has an influence on the software and policies in the operating system and supporting utilities.
- **Complexity Of control:** A printer requires a relatively simple control interface. A disk is much complex. The effect of these difference on the operating system is filtered to some extent by the complexity of the I/O module that controls the device.
- **Unit of transfer:** Data may be transferred as a stream of bytes or characters.
- **Data representation:** Different data encoding schemes are used by different devices, including difference in character code and parity conventions.
- **Error condition:** The nature of errors, the way in which they are reported, their consequences, and the available range of responses differ widely from one device to another.

I/O Organization

Three techniques for performing I/O:

- **Programmed I/O:** The processor issues an I/O command, on behalf of a process, to an I/O module; that process then busy wait for the operation to be completed before proceeding.
- **Interrupt-driven I/O:** The processor issues an I/O command on behalf of a process, continues to execute subsequent instructions, and is interrupted by the I/O module.
- **Direct memory access (DMA):** A DMA module controls the exchange of data Between main memory and an I/O module.

The Evolution of the I/O Function

The processor directly controls a peripheral device. This is seen in simple microprocessor-controlled devices. A controller, or I/O module is added. The processor uses programmed I/O without interrupts. The processor need not

spend time waiting for an I/O operation to be perforated, thus increasing efficiency. The I/O module is given direct control of memory via DMA. It can now move a block of data to or from memory without involving the processor. The I/O module is enhanced to become a separate processor. The I/O module has a local of its own. With this architecture, a large set of I/O device can be controlled, with minimal processor involvement.

Direct Memory Access:

The DMA unit is capable of mimicking the processor and, indeed, of taking over control of the system bus just like a processor. It needs to do this to transfer data to and from memory over the system bus.

When the processor wishes to read or write a block of data, it issues a command to the DMA module by sending to the DMA module the following information:

- Whether a read or write is requested, using read or write control line between the processor and the DMA module.
- The address of the I/O device involved, communicated on the data line.
- The starting location in memory to read from or write to, communicated on the data lines and stored by the DMA module in its address register.
- The number of the word to be read or written, again communicated via the data lines and stored in the data count register.

The DMA module transfers the entire block of data, one word at a time, directly to or from memory, without going through the processor. When the transfer is complete, the DMA module sends an interrupt signal to the processor. Thus, the processor is involved only at the beginning and end of the transfer.

The DMA module, uses programmed I/O to exchange data between memory and an I/O module through the DMA module.

Design Objectives

Two objectives are considered designing the I/O facility: efficiency and generality. **Efficiency** is important because I/O operations often form a bottleneck in a computing system. I/O devices are extremely slow compared with main memory and the processor. One way to tackle this problem is multiprogramming, which, as we have seen, allows some processes to be waiting on I/O operations while another process is executing. The other major objective is **generality**.

Some parts of the operating system must interact directly with the computer hardware

Generally they follow following layer of involvement

- **Logical I/O:** The logical I/O module deals with the device as a logical resource and is not concerned with the details of actually controlling the device. The logical I/O module is concerned with managing general I/O functions on behalf of user processes, allowing them to deal with the device in terms of a device identifier and simple commands such as open, close, read, write.
- **Device I/O:** The requested operations and data (buffered characters, records, etc.) are converted into appropriate sequences of I/O instructions, channel commands, and controller orders. Buffering techniques may be used to improve utilizations.
- **Scheduling and control:** The actual queuing and scheduling of I/O operations occurs at this layer, as well as the control of the operations. Thus, interrupts are handled at this layer and I/O status is collected and reported. This is the layer of software that actually interacts with the I/O module and hence the device hardware.
- **Directory management:** At this layer, symbolic file names are converted to identifiers that either reference the file directly or indirectly through a file descriptor or index table.
- **File system:** This layer deals with the logical structure of files and with the operations that can be specified by users, such as open, close, read, write. Access rights are also managed at this layer.
- **Physical organization:** Virtual memory addresses must be converted into physical main memory addresses, taking into account the segmentation and paging structure, logical references to files and records must be converted to physical secondary storage addresses, taking into account the physical track and sector structure of the secondary storage device. Allocation of secondary storage space and main storage buffers is generally treated at this layer as well.

Suppose that a user process wishes to read blocks of data from a tape one at a time, with each block having a length of 512 bytes. The data are to be read into a data area within the address space of the user process at virtual location 1000 to 1511. The simplest way would be to execute an I/O command to the tape unit and then wait for the data to become available. There are two problems with this approach. First, the program is hung up waiting for the relatively slow I/O to complete. The second problem is that this approach to I/O interferes with swapping decisions by the operating system. To avoid these overheads and inefficiencies, it is sometimes convenient to perform input transfers in advance of requests being made and to perform output transfers some time after the request is made. This technique is known as buffering. It is sometimes important to make a distinction between two types of I/O devices: block oriented and stream

oriented. A **block-oriented** device stores information in blocks that are usually of fixed size, and transfers are made one-block at a time. Disks and tapes are examples of block-oriented device. A **stream-oriented** device transfers data in and out as a stream of bytes, with no block structure. Terminals, printers, communications ports, mouse and other pointing devices, and most other devices that are not secondary storage are stream oriented.

Single Buffer

The simplest type of support that the operating system can provide is single buffer. When a user process issues an I/O request, the operating system signs a buffer in file system portion of main memory to the operation. For block-oriented devices, the single buffering scheme can be used.

The two main jobs of a computer are I/O and processing. In many cases, the main job is I/O and the processing is merely incidental. For instance, when we browse a web page or edit a file, our immediate interest is to read or type in some information.

The role of operating system in computer I/O is to manage and control I/O operations and I/O devices. Although related topics appear in other chapters, here we bring together the pieces to paint a complete picture. We discuss the I/O services that the operating system provides, and the embodiment of these services in the application I/O interface.

The control of devices connected to the computer is a major concern of operating system designers. These methods form I/O sub-system of the kernel, which separates the rest of the kernel from the complexity of managing I/O devices.

I/O-device technology exhibits two conflicting trends. On one hand, we see increasing standardization of software and hardware interfaces. This trend helps us to incorporate improved device generations into existing computers and operating systems. On the other hand, we see an increasingly broad variety of I/O devices.

I/O Hardware

Computers operate a great many kinds of devices. General types include storage devices (disks, tapes), transmission devices (network cards, modems), and human-interface devices (screen, keyboard, mouse)

A device communicates with a computer system by sending signals over a cable or even through the air. The device communicates with the machine via a connection point termed a port (for example, a serial port). If one or more devices use a common set of wires, the connection is called a bus. When device A has a

cable that plugs into device B, and device B has a cable that plugs into device C, and device C plugs into a port on the computer, this arrangement is called a daisy chain. It usually operates as a bus.

A controller is a collection of electronics that can operate a port, a bus, or a device. A serial-port controller is an example of a simple device controller. It is a single chip in the computer that controls the signals on the wires of a serial port. The SCSI bus controller is often implemented as a separate circuit board (a host adapter) that plugs into the computer. It typically contains a processor, microcode, and some private memory to enable it to process the SCSI protocol messages. Some devices have their own built-in controllers.

An I/O port typically consists of four registers, called the status, control, data-in, and data-out registers. The status register contains bits that can be read by the host. These bits indicate states such as whether the current command has completed, whether a byte is available to be read from the data-in register, and whether there has been a device error. The control register can be written by the host to start a command or to change the mode of a device. For instance, a certain bit in the control register of a serial port chooses between full-duplex and half-duplex communication, another enables parity checking, a third bit sets the word length to 7 or 8 bits, and other bits select one of the speeds supported by the serial port. The data-in register is read by the host to get input, and the data-out register is written by the host to send output. The data registers are typically 1 to 4 bytes. Some controllers have FIFO chips that can hold several bytes of input or output data to expand the capacity of the controller beyond the size of the data register. A FIFO chip can hold a small burst of data until the device or host is able to receive those data.

Polling

Incomplete protocol for interaction between the host and a controller can be intricate, but the basic handshaking notion is simple. The controller indicates its state through the busy bit in the status register. (Recall that to set a bit means to write a 1 into the bit, and to clear a bit means to write a 0 into it.) The controller sets the busy bit when it is busy working, and clears the busy bit when it is ready to accept the next command. The host signals its wishes via the command-ready bit in the command register. The host sets the command-ready bit when a command is available for the controller to execute. For this example, the host writes output through a port, coordinating with the controller by handshaking as follows.

1. The host repeatedly reads the busy bit until that bit becomes clear.
2. The host sets the write bit in the command register and writes a byte into the data-out register.
3. The host sets the command-ready bit.
4. When the controller notices that the command-ready bit is set, it sets the busy bit.

5. The controller reads the command register and sees the write command. It reads the data-out register to get the byte, and does the I/O to the device.
6. The controller clears the command-ready bit, clears the error bit in the status register to indicate that the device I/O succeeded, and clears the busy bit to indicate that it is finished.

The host is busy-waiting or polling: It is in a loop, reading the status register over and over until the busy bit becomes clear. If the controller and device are fast, this method is a reasonable one. But if the wait may be long, the host should probably switch to another task

CHAPTER 8

PRINCIPLES OF I/O SOFTWARE

Interrupts

The CPU hardware has a wire called the interrupt request line that the CPU senses after executing every instruction. When the CPU detects that a controller has asserted a signal on the interrupt request line, the CPU saves a small amount of state, such as the current value of the instruction pointer, and jumps to the interrupt-handler routine at a fixed address in memory. The interrupt handler determines the cause of the interrupt, performs the necessary processing, and executes a return from interrupt instruction to return the CPU to the execution state prior to the interrupt. We say that the device controller raises an interrupt by asserting a signal on the interrupt request line, the CPU catches the interrupt and dispatches to the interrupt handler, and the handler clears the interrupt by servicing the device. Figure 12.3 summarizes the interrupt-driven I/O cycle.

This basic interrupt mechanism enables the CPU to respond to an asynchronous event, such as a device controller becoming ready for service. In a modern operating system, we need more sophisticated interrupt-handling features. First, we need the ability to defer interrupt handling during critical processing. Second, we need an efficient way to dispatch to the proper interrupt handler for a device, without first polling all the devices to see which one raised the interrupt. Third, we need multilevel interrupts, so that the operating system can distinguish between high- and low-priority interrupts, and can respond with the appropriate degree of urgency. In modern computer hardware, these three features are provided by the CPU and by the interrupt-controller hardware.

CPUs have two interrupt request lines. One is the non-maskable interrupt, which is reserved for events such as unrecoverable memory errors. The second interrupt line is maskable: It can be turned off by the CPU before the execution of critical instruction sequences that must not be interrupted. The maskable interrupt is used by device controllers to request service.

This address is an offset in a table called the interrupt vector. This vector contains the memory addresses of specialized interrupt handlers. The purpose of a vectored interrupt mechanism is to reduce the need for a single interrupt handler to search all possible sources of interrupts to determine which one needs service.

The interrupt mechanism also implements a system of interrupt priority levels. This mechanism enables the CPU to defer the handling of low-priority interrupts without masking off all interrupts, and makes it possible for a high-priority interrupt to preempt the execution of a low-priority interrupt:

The interrupt mechanism is also used to handle a wide variety of exceptions, such as dividing by zero, accessing a protected or nonexistent memory address, or attempting to execute a privileged instruction from user mode.

A system call is a function that is called by an application to invoke a kernel service. The system call checks the arguments given by the application, builds a data structure to convey the arguments to the kernel, and then executes a special instruction called a software interrupt, or a trap.

Interrupts can also be used to manage the flow of control within the kernel. If the disks are to be used efficiently, we need to start the next I/O as soon as the previous one completes. Consequently, the kernel code that completes a disk read is implemented by a pair of interrupt handlers. The high-priority handler records the I/O status, clears the device interrupt, starts the next pending I/O, and raises a low-priority interrupt to complete the work. The corresponding handler completes the user level I/O by copying data from kernel buffers to the application space and then by calling the scheduler to place the application on the ready queue.

Application I/O Interfaced

Structuring techniques and interfaces for the operating system enable I/O devices to be treated in a standard, uniform way. For instance, how an application can open a file on a disk without knowing what kind of disk it is, and how new disks and other devices can be added to a computer without the operating system being disrupted

The actual differences are encapsulated in kernel modules called device drivers that internally are custom tailored to each device but that export one of the standard interfaces.

The purpose of the device-driver layer is to hide the differences among device controllers from the I/O subsystem of the kernel, much as the I/O system calls.

Character-stream or block. A character-stream device transfers bytes one by one, whereas a block device transfers a block of bytes as a unit.

Sequential or random-access. A sequential device transfers data in a fixed order that is determined by the device, whereas the user of a random-access device can instruct the device to seek to any of the available data storage locations.

Synchronous or asynchronous. A synchronous device is one that performs data transfers with predictable response times. An asynchronous device exhibits irregular or unpredictable response times.

Sharable or dedicated. A sharable device can be used concurrently by several processes or threads; a dedicated device cannot.

Speed of operation. Device speeds range from a few bytes per second to a few gigabytes per second.

Read-write, read only, or write only. Some devices perform both input and output, but others support only one data direction.

For the purpose of application access, many of these differences are hidden by the operating system, and the devices are grouped into a few conventional types. Operating systems also provide special system calls to access a few additional devices, such as a time-of-day clock and a timer.

The performance and addressing characteristics of network I/O differ significantly from those of disk I/O, most operating systems provide a network I/O interface that is different from the **read-write-seek** interface used for disks.

Clocks and Timers

Most computers have hardware clocks and timers that provide three basic functions:

- Give the current time
- Give the elapsed time
- Set a timer to trigger operation X at time T

These functions are used heavily by the operating system, and also by time-sensitive applications. The hardware to measure elapsed time and to trigger operations is called a programmable interval timer.

Blocking and Non-blocking I/O

One remaining aspect of the system-call interface relates to the choice between blocking I/O and non-blocking (asynchronous) I/O. When an application calls a

blocking system call, the execution of the application is suspended. The application is moved from the operating system's run queue to a wait queue. After the system call completes, the application is moved back to the run queue, where it is eligible to resume execution, at which time it will receive the values returned by the system call.

Some user-level processes need nonblocking I/O.

Kernel I/O Subsystem

Kernels provide many services related to I/O. The services that we describe are I/O scheduling, buffering caching, spooling, device reservation, and error handling.

Scheduling

To schedule a set of I/O requests means to determine a good order in which to execute them. The order in which applications issue system calls rarely is the best choice. Scheduling can improve overall system performance, can share device access fairly among processes, and can reduce the average waiting time for I/O to complete. Operating-system developers implement scheduling by maintaining a queue of requests for each device. When an application issues a blocking I/O system call, the request is placed on the queue for that device. The I/O scheduler rearranges the order of the queue to improve the overall system efficiency and the average response time experienced by applications.

Buffering

A buffer is a memory area that stores data while they are transferred between two devices or between a device and an application. Buffering is done for three reasons. One reason is to cope with a speed mismatch between the producer and consumer of a data stream. Second buffer while the first buffer is written to disk. A second use of buffering is to adapt between devices that have different data transfer sizes. A third use of buffering is to support copy semantics for application I/O.

Caching

A cache is region of fast memory that holds copies of data. Access to the cached copy is more efficient than access to the original.

Caching and buffering are two distinct functions, but sometimes a region of memory can be used for both purposes.

Spooling and Device Reservation

A spool is a buffer that holds output for a device, such as a printer, that cannot accept interleaved data streams. The spooling system copies the queued spool files to the printer one at a time. In some operating systems, spooling is managed by a system daemon process. In other operating systems, it is handled by an in-kernel thread.

Error Handling

An operating system that uses protected memory can guard against many kinds of hardware and application errors.

CHAPTER 9 DISKS

The disk increase in the speed of processors and main memory has far outstripped that for disk access, with processor and main memory speeds increasing by about two orders of magnitude compared to one order of magnitude for disk.

Disk Performance Parameters :

1. When the disk drive is operating, the disk is rotating at constant speed. To read or write, the head must be positioned at the desired track and at the beginning of the desired sector on that track.¹ Track selection involves moving the head in a movable-head system or electronically selecting one head on a fixed-head system. On a movable-head system, the time it takes to position the head at the track is known as **seek time**. When once the track is selected, the disk controller waits until the appropriate sector rotates to line up with the head. The time it takes for the beginning of the sector to reach the head is known as **rotational delay**, or rotational latency. The sum of the seek time, if any, and the rotational delay equals the **access time**, which is the time it takes to get into position to read or write. Once the head is in position, the read or write operation is then performed as the sector moves under the head; this is the data transfer portion of the operation; the time required for the transfer is the **transfer time**.

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

Rotational Delay Disks, other than floppy disks, rotate at speeds ranging from

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

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

where

T = transfer time

b = number of bytes to be transferred

N = number of bytes on a track

r = rotation speed, in revolutions per second

Thus the total average access time can be expressed as

$$T_a = T_s +$$

where T_s is the average seek time.

For the single disk, there will be a number of I/O requests (reads and writes)-from various processes in the queue. This **random scheduling** is useful as a benchmark against which to evaluate other techniques.

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

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

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

Shortest Service Time First The SSTF policy is to select the disk I/O request that requires the least movement of the disk arm from its current position.

Scan With the exception of FIFO, all of the policies described so far can leave some request unfulfilled until the entire queue is emptied. That is, there may always be new requests arriving that will be chosen before an existing request.

C-Scan The C-SCAN policy restricts scanning to one direction only. Thus, when the last track has been visited in one direction, the arm is returned to the opposite end of the disk and the scan begins again. This reduces the maximum delay experienced by new requests,

N-step-SCAN and FSCAN With SSTF, SCAN and C-SCAN, it is possible that the arm may not move for a considerable period of time.

The term cache memory is usually used to apply to a memory that is smaller and faster than main memory and that is interposed between main memory and the processor. Such a cache memory reduces average memory access time by exploiting the principle of locality. The same principle can be applied to disk memory. Specifically, a disk cache is a buffer in main memory for disk sectors. The cache contains a copy of some of the sectors on the disk. When an I/O request is made for a particular sector, a check is made to determine if the sector is in the disk cache. If so, the request is satisfied via the cache. If not, the requested sector is read into the disk cache from the disk. The phenomenon of locality of reference, when a block of data is fetched into the cache to satisfy a single I/O request, it is likely that there will be future references to that same block.

Performance

Once the basic disk methods are selected, there are still several ways to improve performance. Most disk controllers include local memory to form an on-board cache that is sufficiently large to store entire track at a time. Once a seek is performed, the track is read into the disk cache starting at the sector under the disk head. The disk controller then transfers any sector requests to the operating system. Once blocks are made it from the disk controller into main memory, the operating system may cache the blocks there. Some systems maintain a separate section of main for a disk cache, where blocks are kept under the assumption that they will be used again shortly. Other systems treat all physical memory as a buffer pool that is shared by the paging system and the disk-block caching system. A system performing many I/O operations will use most of its memory as a block cache, whereas a system executing many programs will use more memory as paging space. Some systems optimize their disk cache by using different replace algorithms.

Another method of using main memory to improve performance is common on personal computers. A section of memory is set aside and treated as a virtual disk, or RAM disk. In this case, a RAM disk device driver accepts all the standard disk operations, but performs those operations on the memory section, instead of on a disk. All disk operations can then be executed on this RAM disk and, except for the lightning-fast speed, users will not notice a difference. RAM disks are useful only for temporary storage, since a power failure or a reboot of the system will usually erase them. Commonly, temporary files such as intermediate compiler files are stored there.

The difference between a RAM disk and a disk cache is that the contents of the RAM disk are totally user controlled, whereas those of the disk cache are under the control of the operating system. A RAM disk will stay empty until the user creates files there.

Disk Scheduling

One of the responsibilities of the operating system is to use the hardware efficiently. For the disk drives, this means having a fast access time and disk bandwidth. The access time has two major components. The seek time is the time for the disk arm to move the heads to the cylinder containing the desired sector. The rotational latency is the additional time waiting for the disk to rotate the desired sector to the disk head. The disk bandwidth is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer.

The request specifies several pieces of information:

- Whether this operation is input or output
- What the disk address for the transfer is
- What the memory address for the transfer is
- What the number of bytes to be transferred is

If the desired disk drive and controller are available, the request can be serviced immediately. If the drive or controller is busy, any new requests for service will need to be placed on the queue of pending requests for that drive.

FCFS Scheduling

The simplest form of disk scheduling is, of course, first-come, first-served (FCFS). Consider, as an example, a disk queue with requests for I/O to blocks on cylinders 98, 183, 37, 122, 14, 124, 65, 67

In that order. If the disk head is initially at cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 124, 65, and finally to 67, for a total head movement of 640 cylinders.

SSTF Scheduling

It seems reasonable to service all the requests close to the current head position, before moving the head far away to service other requests. This assumption is the basis for the shortest-seek-time-first (SSTF) algorithm. The SSTF algorithm selects the request with the minimum seek time from the current head position. Since seek time increases with the number of cylinders traversed by the head, SSTF chooses the pending request closest to the current head position.

SSTF scheduling is essentially a form of shortest-job-first (SJF) scheduling, it may cause starvation of some requests.

SCAN Scheduling

In the SCAN algorithm, the disk arm starts at one end of the disk, and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk.

The SCAN algorithm is sometimes called the elevator algorithm.

C-SCAN Scheduling

Circular SCAN (C-SCAN) is a variant of SCAN that is designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other servicing requests along the way. When the head reaches the other end, it immediately returns to the beginning of the disk, without servicing any requests on the return trip

LOOK Scheduling

Both SCAN and C-SCAN move the disk arm across the full width of the disk. In practice, neither algorithm is implemented this way. More commonly, the arm goes only as far as the final request in each direction. Then, it reverses direction immediately, without first going all the way to the end of the disk. These versions of SCAN and C-SCAN are called LOOK and C-LOOK

Selection of a Disk-Scheduling Algorithm

Note that the scheduling algorithms described here consider only the seek distances. For Modern disks, the rotational latency can be nearly as large as the average seek time. But, it is difficult for the operating system to schedule for improved rotational latency because modern disks do not disclose the physical location of logical blocks. But the operating system sends a batch of requests to the controller, the controller can queue them and then schedule them to improve both the seek time and the rotational latency.

CHAPTER 11 FILE SYSTEMS

Files and File Systems

The most important parts of an operating system is the file system. The file system provides the resource abstractions typically associated with secondary

storage. The file system permits users to create data collections, called files, with desirable properties, such as the following:

- **Long-term existence:** Files are stored on disk or other secondary storage and do not disappear when a user logs off.
- **Sharable between processes:** Files have names and can have associated access permissions that permit controlled sharing.
- **Structure:** Depending on the file system, a file can have an internal structure that is convenient for particular applications. In addition, files can be organized into hierarchical or more complex structure to reflect the relationships among files.

Any file system provides not only a means to store data organized as files, but a collection of functions that can be performed on files. Typical operations include the following:

- **Create:** A new file is defined and positioned within the structure of files.
- **Delete:** A file is removed from the file structure and destroyed.
- **Open:** An existing file is declared to be "opened" by a process, allowing the process to perform functions on the file.
- **Close:** The file is closed with respect to a process, so that the process no longer may perform functions on the file, until the process opens the file again.
- **Read:** A process reads all or a portion of the data in a file.
- **Write:** A process updates a file, either by adding new data that expands the size of the file or by changing the values of existing data items in the file.

Typically, a file system maintains a set of attributes associated with the file

File Structure

Four terms are use for files

- Field
- Record
- Database

A field is the basic element of data. An individual field contains a single value.

A record is a collection of related fields that can be treated as a unit by some application program.

A file is a collection of similar records. The file is treated as a singly entity by users and applications and may be referenced by name. Files have file names and maybe created and deleted. Access control restrictions usually apply at the file level.

A database is a collection of related data. Database is designed for use by a number of different applications. A database may contain all of the information related to an organization or project, such as a business or a scientific study. The database itself consists of one or more types of files. Usually, there is a separate database management system that is independent of the operating system.

File Management Systems:

A file management system is that set of system software that provides services to users and applications in the use of files. following objectives for a file management system:

- To meet the data management needs and requirements of the user which include storage of data and the ability to perform the aforementioned operations.
 - To guarantee, to the extent possible, that the data in the file are valid.
 - To optimize performance, both from the system point of view in terms of overall throughput.
 - To provide I/O support for a variety of storage device types.
 - To minimize or eliminate the potential for lost or destroyed data.
 - To provide a standardized set of I/O interface routines to use processes.
- To provide I/O support for multiple users, in the case of multiple-user systems

File System Architecture At the lowest level, **device drivers** communicate directly with peripheral devices or their controllers or channels. A device driver is responsible for starting I/O operations on a device and processing the completion of an I/O request. For file operations, the typical devices controlled are disk and tape drives. Device drivers are usually considered to be part of the operating system.

The I/O control, consists of device drivers and interrupt handlers to transfer information between the memory and the disk system. A device driver can be thought of as a translator.

The basic file system needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk.

The file-organization module knows about files and their logical blocks, as well as physical blocks. By knowing the type of file allocation used and the location of the file, the file-organization module can translate logical block addresses to physical block addresses for the basic file system to transfer. Each file's logical blocks are numbered from 0 (or 1) through N, whereas the physical blocks containing the data usually do not match the logical numbers, so a translation is needed to locate each block. The file-organization module also includes the free-space manager, which tracks unallocated and provides these blocks to the file organization module when requested.

The logical file system uses the directory structure to provide the file-organization module with the information the latter needs, given a symbolic file name. The logical file system is also responsible for protection and security.

To create a new file, an application program calls the logical file system. The logical file system knows the format of the directory structures. To create a new file, it reads the appropriate directory into memory, updates it with the new entry, and writes it back to the disk.

Once the file is found the associated information such as size, owner, access permissions and data block locations are generally copied into a table in memory, referred to as the open-file table, consisting of information about all the currently opened files

The first reference to a file (normally an open) causes the directory structure to be searched and the directory entry for this file to be copied into the table of opened files. The index into this table is returned to the user program, and all further references are made through the index rather than with a symbolic name. The name given to the index varies. Unix systems refer to it as a file descriptor, Windows/NT as a file handle, and other systems as a file control block. Consequently, as long as the file is not closed, all file operations are done on the open-file table. When the file is closed by all users that have opened it, the updated file information is copied back to the disk-based directory structure.

File-System Mounting

As a file must be opened before it is used, a file system must be mounted before it can be available to processes on the system. The mount procedure is straight forward. The user is given the name of the device, and the location within the file structure at which to attach the file system (called the mount point).

The operating system verifies that the device contains a valid file system. It does so by asking the device driver to read the device directory and verifying that the directory has the expected format. Finally, the operating system notes in its directory structure that a file system is mounted at the specified mount point. This scheme enables the operating system to traverse its directory structure, switching among file systems as appropriate.

Allocation Methods

The direct-access nature of disks allows us flexibility in the implementation of files. Three major methods of allocating disk space are in wide use: contiguous, linked and indexed. Each method has its advantages and disadvantages.

Contiguous Allocation

The contiguous allocation method requires each file to occupy a set of contiguous blocks on the disk. Disk addresses define a linear ordering on the disk. Notice that with this ordering assuming that only one job is accessing the disk, accessing block $b + 1$ after block b normally requires no head movement.

When head movement is needed, it is only one track. Thus, the number of disk seeks required for accessing contiguously allocated files is minimal.

Contiguous allocation of a file is defined by the disk address and length (in block units) of the first block. If the file is n blocks long, and starts at location b , then it occupies blocks $b, b + 1, b + 2, \dots, b + n - 1$. The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file.

Accessing a file that has been allocated contiguously is easy. For sequential access, the file system remembers the disk address of the last block referenced and, when necessary, reads the next block. For direct access to block i of a file that starts at block b , we can immediately access block $b + i$.

The contiguous disk-space-allocation problem can be seen to be a particular application of the general dynamic storage-allocation First Fit and Best Fit are the most common strategies used to select a free hole from the set of available holes. Simulations have shown that both first-fit and best-fit are more efficient than worst-fit in terms of both time and storage utilization. Neither first-fit nor best-fit is clearly best in terms of storage utilization, but first-fit is generally faster.

These algorithms suffer from the problem of external fragmentation. As files are allocated and deleted, the free disk space is broken into little pieces. External fragmentation exists whenever free space is broken into chunks. It becomes a problem when the largest contiguous chunk is insufficient for a request; storage is fragmented into a number of holes, no one of which is large enough to store the data. Depending on the total amount of disk storage and the average file size, external fragmentation may be either a minor or a major problem.

To prevent loss of significant amounts of disk space to external fragmentation, the user had to run repacking routine that copied the entire file system onto another floppy disk or onto a tape. The original floppy disk was then freed completely, creating one large contiguous free space. The routine then copied the files back onto the floppy disk by allocating contiguous space from this one large hole. This scheme effectively compacts all free space into one contiguous space, solving the fragmentation problem. The cost of this compaction is time. The time cost is particularly severe for large hard disks that use contiguous allocation, where compacting all the space may take hours and may be necessary on a weekly basis. During this down time, normal system operation generally cannot be permitted, so such compaction is avoided at all costs on production machines.

A major problem is determining how much space is needed for a file. When the file is created, the total amount of space it will need must be found and allocated.

The user will normally over estimate the amount of space needed, resulting in considerable wasted space.

Linked Allocation

Linked allocation solves all problems of contiguous allocation. With link allocation, each file is a linked list disk blocks; the disk blocks may be scattered anywhere on the disk.

This pointer is initialized to nil (the end-of-list pointer value) to signify an empty file. The size field is also set to 0. A write to the file causes a free block to be found via the free-space management system, and this new block is written to, and is linked to the end of the file

There is no external fragmentation with linked allocation, and any free block on the free-space list can be used to satisfy a request. Notice also that there is no need to declare the size of a file when that file is created. A file can continue to grow as long as there are free blocks. Consequently, it is never necessary to compact disk space.

The major problem is that it can be used effectively for only sequential access files. To find the *i*th block of a file we must start at the beginning of that file, and follow the pointers until we get to the *i*th block. Each access to a pointer requires a disk read and sometimes a disk seek. Consequently, it is inefficient to support a direct-access capability for linked allocation files.

Linked allocation is the space required for the pointers. If a pointer requires 4 bytes out of a 512 Byte block then 0.78 percent of the disk is being used for pointer, rather than for information.

The usual solution to this problem is to collect blocks into multiples, called clusters, and to allocate the clusters rather than blocks. For instance, the file system define a cluster as 4 blocks and operate on the disk in only cluster units. Pointers then use a much smaller percentage of the file's disk space. This method allows the logical-to-physical block mapping to remain simple, but improves disk throughput (fewer disk head seeks) and decreases the space needed for block allocation and free-list management. The cost of this approach is an increase in internal fragmentation.

Yet another problem is reliability. Since the files are linked together by pointers scattered all over the disk, consider what would happen if a pointer— were lost or damaged. Partial solutions are to use doubly linked lists or to store the file name and relative block number in each block; however, these schemes require even more overhead for each file.

An important variation, on the linked allocation method is the use of a file allocation table (FAT). This simple but efficient method of disk-space allocation is

used by the MS-DOS and OS/2 operating systems. A section of disk at the beginning of each-partition is set aside to contain the table. The table has one entry for each disk block, and is indexed by block number. The FAT is used much as is a linked list. The directory entry contains the block number of the first block of the file. The table entry indexed by that block number then contains the block number of the next block in the file. This chain continues until the last block, which has a special end-of-file value -as the table entry. Unused blocks are indicated by a 0 table value. Allocating a new block to a file is a simple matter of finding the first 0-valued table entry, and replacing the previous end-of-file value with the address of the new block. The 0 is then replaced with the end-of-file value. An illustrative example is the FAT structure of for a file consisting of disk blocks 217, 618, and 339.

Indexed Allocation

Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation. The absence of a FAT, linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and need to be retrieved in order Indexed allocation solves this problem by bringing all the pointers together into one location: the index block.

Each file has its own index block, which is an array of disk-block addresses. The *i*th entry in the index block points to the *i*th block of the file. The directory contains the address of the index block.

When the file is created, all pointers in the index block are set to nil. When the *i*th block is first written, a block is obtained: from the free space manager, and its address- is put in the *i*th index-block entry.

Allocation supports direct access, without suffering from external fragmentation because any free block on he disk may satisfy a request for more space.

Indexed allocation does suffer from wasted space. The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation.

- **Linked scheme.** An index block is normally one disk block. Thus, it can be read and written directly by itself.
- **Multilevel index.** A variant of the linked representation is to use a first-level index block to point to a set of second-level index blocks, which in turn point to the file blocks. To access a block, the operating system uses the first-level index to find a second-level index block, and that block to find the desired data block.

Free-Space Management

Since there is only a limited amount of disk space, it is necessary to reuse the space from deleted files for new files, if possible.

Bit Vector

Free-space list is implemented as a bit map or bit vector. Each block is represented by 1 bit. If the block is free, the bit is 1; if the block is allocated, the bit is 0.

For example consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free, and the rest of the blocks are allocated. The free-space bit map would be

001111001111110001100000011100000

The main advantage of this approach is that it is relatively simple and efficient to find the first free block or n consecutive free blocks on the disk.

The calculation of the block number is

$(\text{number of bits per word}) \times (\text{number of 0-value words}) + \text{offset of first 1 bit}$

Linked List

Another approach is to link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory. This first block contains a pointer to the next free disk block, and so on. Block 2 would contain a pointer to block 3, which would point to block 4, which would point to block 5, which would point to block 8, and so on. Usually, the operating system simply needs a free block so that it can allocate that block to a file, so the first block in the free list is used.

Grouping

A modification of the free-list approach is to store the addresses of n free blocks in the first free block. The first n-1 of these blocks are actually free. The importance of this implementation is that the addresses of a large number of free blocks can be found quickly, unlike in the standard linked-list approach.

Counting

Several contiguous blocks may be allocated or freed simultaneously, particularly when space is allocated with the contiguous allocation algorithm or through clustering. A list of n free disk addresses, we can keep the address of the first free block and the number n of free contiguous blocks that follow the first block[^]. Each entry in the free-space list then consists of a disk address and a count. Although each entry requires more space than would a simple disk address, the overall list will be shorter, as long as the count is generally greater than 1.

CHAPTER 12 SECURITY

Projection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by a subsystem that is malfunctioning

A computer system is a collection of processes and objects. By objects, we mean both hardware objects (such as the CPU, memory segments, printers, disks, tape drives), and software objects (such as files, programs, and semaphore;

Protection domain is process that operates within a protection domain, which specifies the resources that the process may access. Each domain defines a set of objects and .the types of operations that may be invoked on each object. The ability to execute an operation on an object is an access right. A domain is a collection of access ' rights, each of which is an ordered pair <object-name, rights-set>

Domains do not need to be disjoint; they may share access rights. The association between a process and a domain may be either static or dynamic.

A domain can be realized in a variety of ways:

- Each user may be a domain. The set of objects that can be accessed depends on the identity of the user. Domain switching occur when the user is changed — generally when one user logs out and another user logs in.
- Each process may be a domain. In this case, the set of objects that can be accessed depends on the identity of the process. Domain switching corresponds to one process sending a message to another process, and then waiting for a response.
- Each procedure may be a domain. In this case, the set of objects that can be accessed corresponds to the local variables define Domain switching occurs when a procedure call is made.

MULTICS

In the MULTICS system, the protection domains are organized hierarchically into a ring structure. Each ring corresponds to a single domain. The rings are numbered from 0 to 7. Let D_i and D_j be any two domain rings. If $j < i$, then D_j is a subset of D_i . That is, a process executing in domain D_i has more privileges than does a process executing in domain D_j . A process executing in domain D_0 has the most privileges. If there are only two rings, this scheme is equivalent to the monitor-user mode of execution, where monitor mode, corresponds to D_0 and user mode corresponds to D_1 . MULTICS has a segmented address space; each segment is a file.

Access Matrix

Our model of protection can be viewed abstractly as a matrix, called an access matrix. The rows of the access matrix represent domains, and the columns represent objects. Each entry in the matrix consists of a set of access rights. Because objects are defined explicitly by the column, we can omit the object name from the access right. The entry access (i, j) defines the set of operations that a process, executing in domain D_i , can invoke on object D_j .

We consider the access matrix shown in Figure. There are four domains and four objects: three files (F_1, F_2, F_3), and one laser printer. When a process executes in domain D_1 , it can read files F_1 and F_3 . A process executing in domain D_4 has the same privileges as it does in domain D_1 , but in addition, it can also write onto files F_1 and F_3 . Note that the laser printer can be accessed only by a process executing in domain D_2 .

The access-matrix scheme provides us with the mechanism for specifying a variety of policies. More specifically, we must ensure that a process executing in domain D_i can access only those objects specified in row i , and then only as allowed by the access matrix entries.

Policy decisions concerning protection can be implemented by the access matrix. The users normally decide the contents of the access-matrix entries.

Allowing controlled change to the contents of the access-matrix entries requires three additional operations: **copy, owner, and control**.

The Security Problem

The operating system can allow users to protect their resources. We say that a system is secure if its resources are used and accessed as intended under all circumstances. Unfortunately, it is not generally possible to achieve total security.

Security violations of the system can be categorized as being either intentional (malicious) or accidental. Among the forms of malicious access are the following:

- Unauthorized reading of data (theft of information)
- Unauthorized modification of data
- Unauthorized destruction of data j

To protect the system, we must take security measures at two levels:

- **Physical:** The site or sites containing the computer systems must be physically secured against armed or surreptitious entry by intruders.
- **Human:** Users must be screened carefully so that the chance of authorizing a user who then gives access to an intruder (in exchange for a bribe, for example) is reduced.

Security at both levels must be maintained if operating-system security is to be ensured.

Authentication

A major security problem for operating systems is the authentication problem. The protection system depends on an ability to identify the programs and processes that are executing. Authentication is based on one or more of three items: user possession (a key or card), user knowledge (a user identifier and password), and a user attribute (fingerprint) retina pattern, or signature).

Passwords

The most common approach to authenticating a user identity is the use of user passwords. When the user identifies herself by user ID or account name, she is asked for a password. If the user-supplied password matches the password stored in the system.

Password Vulnerabilities

Although there are problems associated with their use, passwords are nevertheless extremely common, because they are easy to understand and use. The problems with passwords are related to the difficulty of keeping a password secret.

There are two common ways to guess a password. One is for the intruder (either human or program) to know the user or to have information about the user. The other way is to use brute force; trying all possible combinations of letters, numbers, and punctuation until the password is found.

Passwords can be either generated by the system or selected by a user. System generated passwords may be difficult to remember, and thus users may commonly write them down

Encrypted Passwords

One problem with all these approaches is the difficulty of keeping the password secret. The UNIX system uses encryption to avoid the necessity of keeping its password list secret. Each user has a password. The system contains a function that is extremely difficult (the designers hope impossible) to invert, but is simple to compute. That is, given a value x , it is easy to compute the function value $f(x)$. Given a function value $f(x)$, however, it is impossible to compute x . This function is used to encode all passwords.

One-Time Passwords

To avoid the problems of password sniffing and shoulder surfing, a system could use a set of paired passwords. When a session begins, the system randomly selects and presents one part of a password pair; the user must supply the other part. In this system, the user is challenged and must respond with the correct answer to that challenge

This approach can be generalized to the use of an algorithm as a password. The algorithm might be an integer function.

In this one-time password system, the password is different in each instance. Anyone capturing the password from one session and trying to reuse it in another session will fail.

The user uses the keypad to enter the shared secret, also known as a personal identification number (PIN). Another variation on one-time passwords is the use of a code book, or one time pad.

Program Threats

In an environment where a program written by one user may be used by another user, there is an opportunity for misuse, which may result in unexpected behavior. There are two common methods. Trojan horses and Trap doors.

Trojan Horse

Many systems have mechanisms for allowing programs written by users to be executed by other users. If these programs are executed in a domain that provides the access rights of the executing user, they may misuse these rights.

A code segment that its environment is called a Trojan horse. The Trojan-horse problem is exacerbated by long search paths. The search path lists the set of directories to search when an ambiguous program name is given. The path is searched for a file of that name and the file is executed. All the directories in the

search path must be secure, or a Trojan horse could be slipped into the user's path and executed accidentally.

A variation of the Trojan horse would be a program that emulates a login program. The emulator stored away the password, printed out a login error message, and exited; the user was then provided with a genuine login prompt. This type of attack can be defeated by the operating system printing a usage message at the end of an interactive session or by a non-trappable key sequence, such as the control-alt-delete combination that Windows NT uses.

Trap Door

The designer of a program or system might leave a hole in the software that only OS is capable of using. A clever trap door could be included in a compiler. The compiler could generate standard object code as well as a trap door, regardless of the source code being compiled.

System Threats

Most operating systems provide a means for processes to spawn other processes.

Worms

A worm is a process that uses the spawn mechanism to clobber system performance. The worm spawns copies of itself, using up system resources and perhaps locking out system use by, all other processes. Since they may reproduce themselves among systems and thus shut down the entire network.

The worm was made up of two programs a grappling hook (also called bootstrap or vector) program and the main program. The grappling hook consisted of 99 lines of C code compiled and run on each machine it accessed. The grappling hook connected to the machine where it originated and uploaded a copy of the main worm onto the "hooked" system. The main program proceeded to search for other machines to which the newly infected system could connect easily.

The attack via remote access was one of three infection methods built into the worm.

Viruses

Another form of computer attack is a virus. Like worms, viruses are designed to spread into other programs and can wreak havoc in a system, including modifying or destroying files and causing system crashes and program malfunctions. Whereas a worm is structured as a complete, standalone program, a virus is a fragment of code embedded in a legitimate program. Viruses are a

major problem for computer users, especially users of microcomputer systems. Even if a virus does infect a program, its powers are limited because other aspects of the system are protected in multi-user. Single-user systems have no such protections and, as a result, a virus has free run.

Viruses are usually spread by users downloading viral programs from public bulletin boards or exchanging floppy disks containing an infection. The best protection against computer viruses is prevention, or the practice of Safe computing.

Threat Monitoring

The security of a system can be improved by two management techniques. One is threat monitoring: The system can check for suspicious patterns of activity in an attempt to detect a security violation.

Another technique is an audit log. An audit log simply records the time, user, and type of all accesses to an object. Networked computers are much more susceptible to security attacks than are standalone systems.

One solution is the use of a firewall to separate trusted and un-trusted systems. A firewall is a computer or router that sits between the trusted and the un-trusted. It limits network access between the two security domains, and monitors and logs all connections. A firewall therefore may need to allow http to pass.

Encryption

Encryption is one common method of protecting information transmitted over unreliable links. The basic mechanism works as follows.

1. The information (text) is encrypted (encoded) from its initial readable form (called clear text), to an internal form (called cipher text). This internal text form, although readable, does not make any sense.
2. The cipher text can be stored in a readable file, or transmitted over unprotected channels.
3. To make sense of the cipher text, the receiver must decrypt (decode) it back into clear text.

Even if the encrypted information is accessed by an unauthorized person or program, it will be useless unless it can be decoded.