DATE:

Unit - III

* Concurrency : Mutual Exclusion And Synchronization *

The central themes of operating system design are all concerned with the management of processes and threads.

* Multiprogramming –

The management of multiple processes within a uniprocessor system.

* Multiprocessing –

The management of multiple process within a multiprocessor.

* Distributed processing –

The management of multiple processes executing on multiple, distributed computer systems. The recent proliferation of clusters is a prime example of this type of system.

Fundamental to all of these areas, and fundamental to OS design, is concurrency. Concurrency encompasses a host of design issues, including communication among processes, sharing of and competing for resources (such as memory, files and I/O access), synchronisation of the activities of multiple processes and allocation of processor time to processes.

* Some key terms related to Concurrency *

Atomic operation –

A sequence of one or more statements that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation.

Critical section –

A section of code within a process that requires access to shared resources and that must not be

executed while another process is in a corresponding section of code.

Deadlock -

A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.

Livelock -

Mutual Exclusion -

The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.

Race Condition -

A situation in which multiple threads or processes read & write a shared data item and the final results depends on the relative timing of their execution.

Starvation -

A situation in which a runnable process is overlooked indefinitely by the scheduler, although it is able to proceed, it is never chosen.

* Process Interaction *

We can classify the ways in which processes interact on the basis of the degree to which they are aware of each other's existence.

* Competition among Processes for Resources -

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 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. We will refer to such a resource as a critical resource, and the portion of the program that uses it a critical section of the program.

* Illustration of Mutual exclusion *

```
/* Process 1 */
void P1
{
while (true) {
 /* preceding code */;
 entercritical (Ra);
 /* critical section */;
 exitcritical (Ra);
 /* following code */;
  }
 }
```

```
/* Process 2 */
void P2
{
while (true) {
/* preceding code */;
 entercritical (Ra);
/* critical section */;
 exitcritical (Ra);
/* following code */;
 }
}
```

...

```
/* Process n */
  void Pn
{
 while (true) {
 /* preceding code */;
   entercritical (Ra);
 /* critical section */;
  exit critical (Ra);
 /* following code */;
  }
}
```

DATE:

* Requirements for Mutual Exclusion *

Any facility or capaability that is to provide support for mutual exclusion should meet the following requirements:

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 noncritical section must do so without interfacing with other processes.

3] It must not be possible for a process requiring access do a critical section to be delayed indefinately: no deadlock or starvation.

4] When no process is in 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 *

Interrupt Disabling -

In a uniprocessor system, concurrent processes can not have overlapped execution; they can only be interleaved. Furthermore, a process will continue to run until it invokes an OS service or until it is interrupted. Therefore to guarantee mutual exclusion, it is sufficient to prevent a process from being interrupted.

A process can then enforce mutual exclusion in the following way -

```
while (true) {
    /* disable interrupts */;
```

```
/* critical section */;
/* enable interrupts */;
/* remainder */;
}
```

* 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 independantly in a peer relationship. There is no interrupt mechanism between processors on which mutual exclusion can be based.

* Compare and Swap Instruction –

The compare & swap instruction, also called a compare & exchange instruction, can be defined as follows.

```
int compare_and_swap (int *word, int testval,
int newval)
{
    int oldval;
    oldval = *word
    if (oldval == testval) *word = newval;
    return oldval;
}
```

* 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. Both the Intel IA-32 architecture (Pentium) and the IA-64 architecture (Itanium) contain an XCHG instruction.

* Semaphores *

* Common Concurrency Mechanism —

1) Semaphore —

An integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic; initialize, decrement and increment. The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process. Also known as a counting semaphore or a general semaphore.

2) Binary Semaphore —

A semaphore that takes on only the values 0 and 1.

3) Mutex —

Similar to a binary semaphore. A key difference between the two is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1).

The fundamental principle is this: Two or more processes can cooperate by means of simple signals, such that a process can be forced to stop at a specified place until it has received a specific signal. Any complex coordination requirement can be satisfied by the appropriate structure of signals. For signaling

special variables called semaphores are used. To transmit a signal via semaphore s, a process executes a premitive semSignal(s). To receive a signal via semaphore s, a process executes the primitive semWait(s); if the corresponding signal has not yet been transmitted, the process is suspended until the transmission takes place.

To achieve the desired effect, we can view the semaphore as a variable that has an integer value upon which only three operations are defined.

1] A semaphore may be initialized to a nonnegative integer value.

2] The semWait operation decrements the semaphore value. If the value becomes negative, then the process executing the semWait is blocked. Otherwise, the process continues execution.

3] The semSignal operation increments the semaphore value. If the resulting value is less than or equal to zero, then a process blocked by a semWait operation, if any, is unblocked.

* A Defination of Semaphore Primitives -

```
struct semaphore {
    int count;
    queueType queue;
};
void semWait (semaphore s)
{
    s.count --;
    if (s.count < 0) {
        /* place this process in s.queue */;
        /* block this process */;
```

DATE :

```
void semSignal (semaphore s)
{
    s.count ++;
    if (s.count <= 0) {
    /* remove a process P from s.queue */;
    /* place process P on ready list */;
    }
}
```

* A defination of Binary semaphore primitives -

```
struct binary - semaphore {
        enum {zero, one} value;
        queueType queue;
};
void semWait B (binary - semaphore s)
{
    if (s.value == one)
        s.value = zero;
    else {
        /* place this process in s.queue */;
        /* block this process */;
    }
}
void semSignal B (semaphore s)
{
    if (s.queue is empty ( )}
        s.value = one;
    else {
        /* remove a process P from s.queue */;
        /* place process P on ready list */;
    }
}
```

The two types of semaphores, the nonbinary semaphore is often referred to as either a counting or semaphore or a general semaphore.

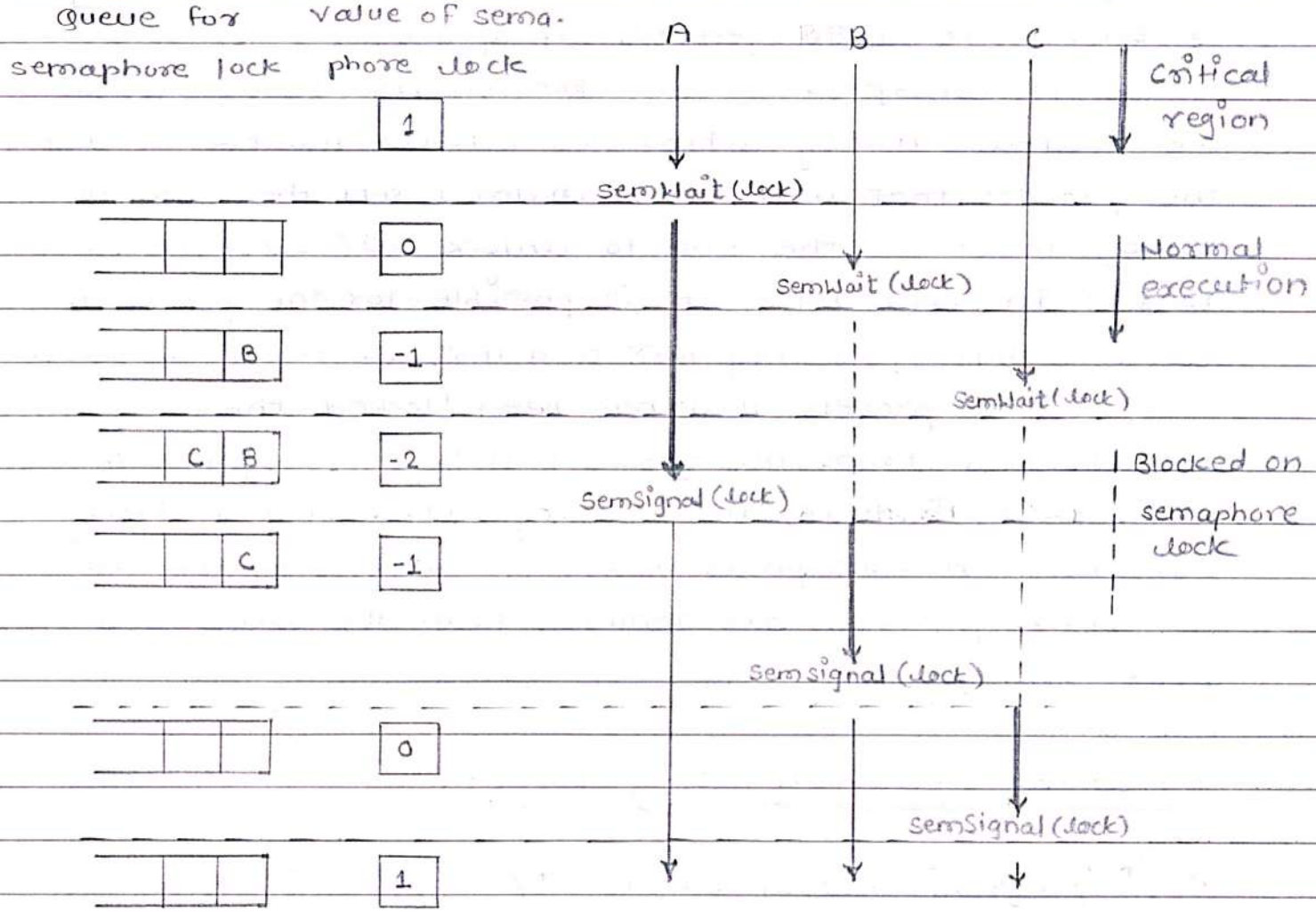
A concept related to the binary semaphore is the mutex. A key difference between the two is that the process that locks the mutex (sets the value to zero) must be the one to unlock it (sets the value to 1). In contrast, it is possible for one process to lock a binary semaphore and for another to unlock it.

The process that has been blocked the longest is released from the queue first; a semaphore whose defination includes this policy is called a strong semaphore. A semaphore that does not specify the order in which processes are removed from the queue is a weak semaphore.

* Mutual Exclusion Using semaphores —

```
/* program mutualexclusion */
const int n = /* number of proceses */;
semaphore s = 1;
void P (int i)
{
  while (true) {
     semWait (s);
     /* critical section */;
     semSignal (s);
     /* remainder */;
  }
}
void main ( )
{
   parbegin ( P(1), P(2), ....., P(n));
}
```

* Mutual Exclusion *



* Note that normal execution can proceed in parallel but that critical regions are serialized.

Fig – Processes Accessing Shared Data Protected by a Semaphore

* The Producer / consumer Problem –

The general statement is this : There are one or more producers generating some type of data (records, characters) and placing these in a buffer. There is a signal consumer that is taking items out of the buffer one at a time.

```
producer :
while (true) {
    /* produce item v */;
    b [in] = v;
    in ++;
}
```

```
consumer :
while (true) {
    while (in <= out)
        /* do nothing */;
    w = b[out];
    out ++;
    /* consumer item w */;
}
```

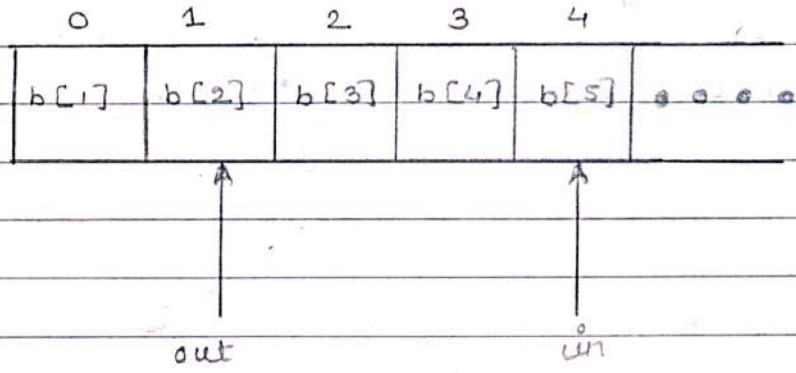


Fig - Infinite Buffer for the Producer / Consumer Problem

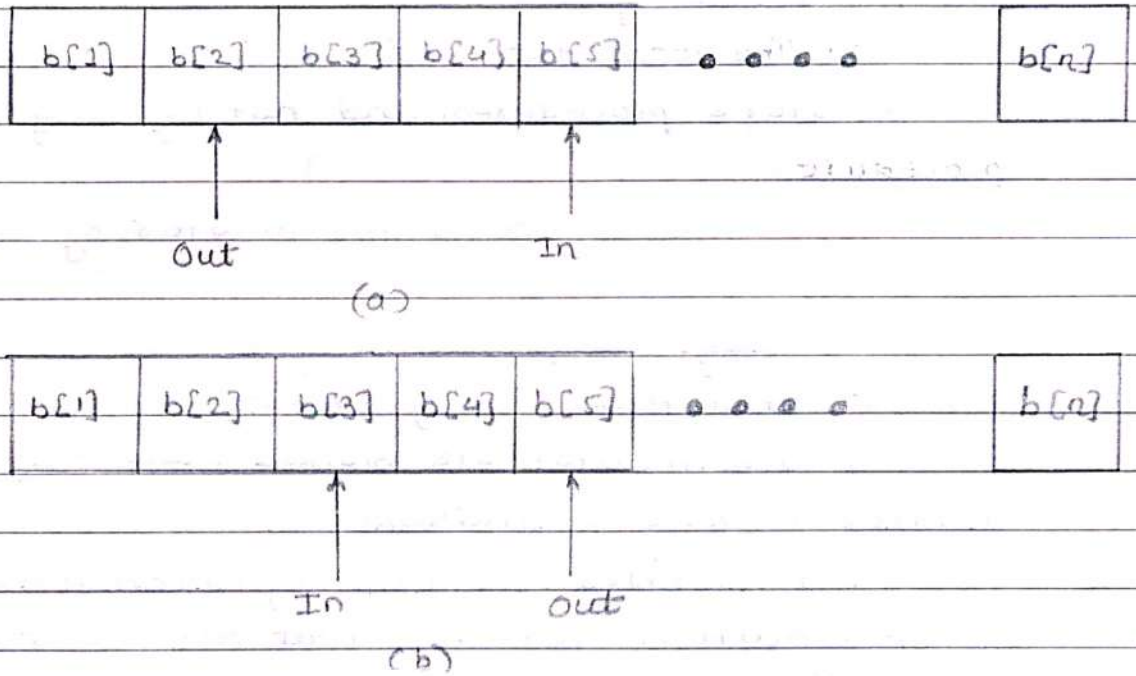


Fig - Finite Circular Buffer for the Producer / consumer problem

The producer and consumer functions can be expressed as follows (variable in and out are initialized to 0 and n is the size of the buffer):

```
producer:
while (true) {
    /* produce item v */
    while ((in + 1) % n == out)
        /* do nothing */;
    b[in] = v;
    in = (in + 1) % n;
}
```

```
consumer:
while (true) {
    while (in == out)
        /* do nothing */;
    w = b[out];
    out = (out + 1) % n;
    /* consume item w */;
}
```

* Monitors *

Monitor with signal –

A monitor is a software module consisting of one or more procedures, an initialization sequence, and local data. The chief 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 processes that have invoked the monitor are 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.

1) cwait (c) :-

Suspend execution of the calling process on condition c. The monitor is now available for use by another process.

2) c signal (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.

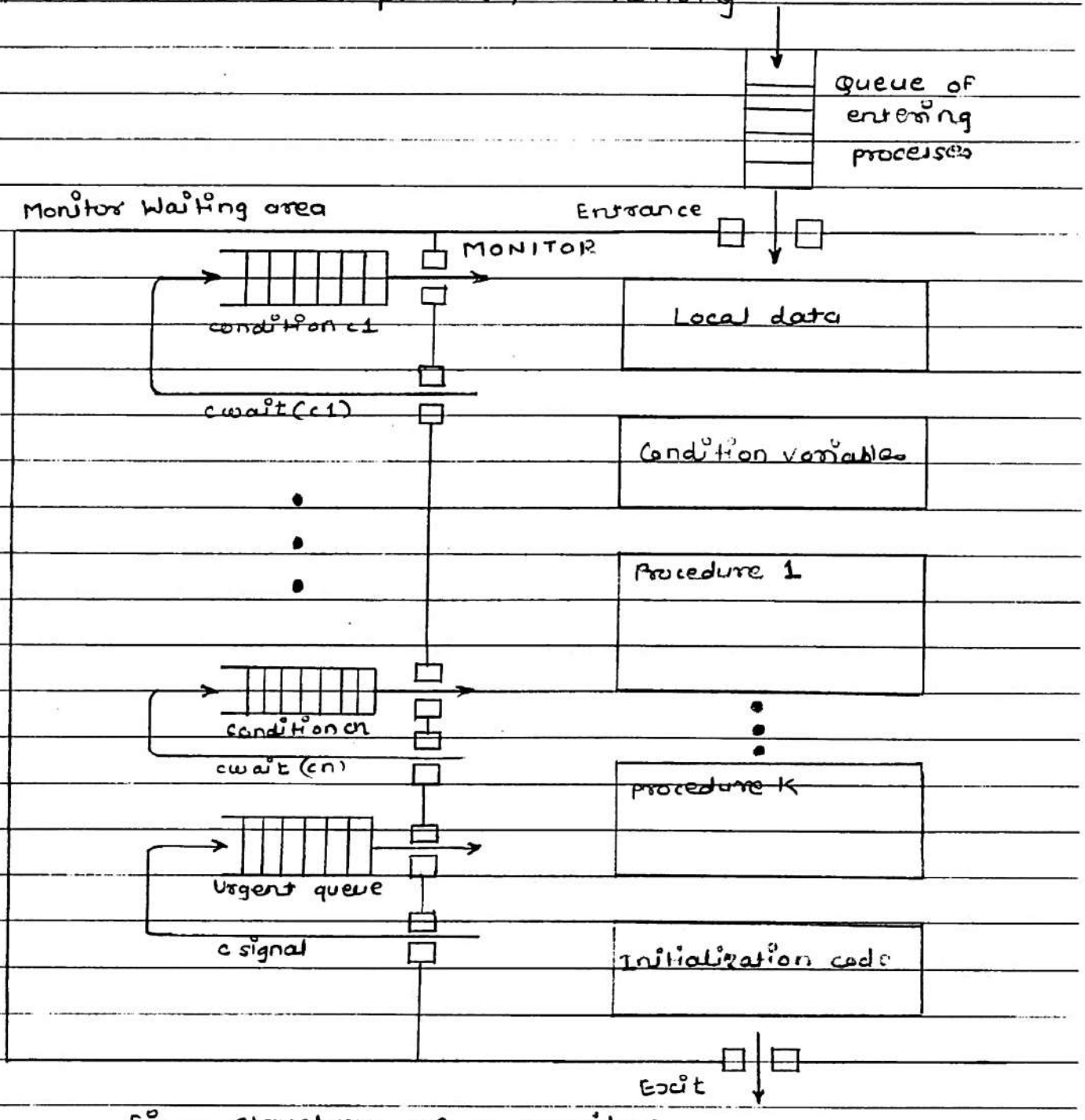


Fig — Structure of a monitor

* Readers / Writers Problem —

The readers / writers problem is defined as follows: There is a data area shared among a number of processes. The data area could be a file, a block of main memory, or even a bank of processor registers. There are a number of processes that only read the data area (readers) and a number that only write to the data area (writers).

The conditions that must be satisfied are as follows:

1] Any number of readers may simultaneously read the file.

2] Only one writer at a time may write to the file.

3] If a writer is writing to the file, no reader may read it.

* Concurrency : Deadlock And Starvation *

* Principles of Deadlock —

Deadlock can be defined as the permanent blocking of a set of processes that either compete for system resources or communicate with each other. A set of processes is deadlocked when each process in the set is blocked awaiting an event (typically the freeing up of some requested resource) that can only be triggered by another blocked processes in the set. Deadlock is permanent because none of the events is ever triggered. Unlike other problems in concurrent process management, there is no efficient solution in the general case.

* The conditions for Deadlock —

Three conditions of policy must be present for a deadlock to be possible.

1) Mutual Exclusion –

Only one process may use a resource at a time. No process may access a resource unit that has been allocated to another process.

2) Hold and wait –

A process may hold allocated resources while awaiting assignment of other resources.

3) No preemption –

No resource can be forcibly removed from a process holding it.

4) Circular wait –

A closed chain of processes exists, such that each process holds atleast one resource needed by the next process in the chain.

Possibility of Deadlock	Existence of deadlock
1] Mutual exclusion	1] Mutual exclusion
2] No preemption	2] No preemption
3] Hold and wait	3] Hold and wait
	4] Circular wait

* Deadlock prevention *

Mutual Exclusion —

In general, the first of the four listed conditions can not be disallowed. If access to a resource requires mutual exclusion, then mutual exclusion may must be supported by the OS. Some resources,

such as files, may allow multiple accesses for reads but only exclusive access for writers. Even in this case, deadlock can occure if more than one process requires write permission.

Hold and wait –

The hold-and-wait condition can be prevented by requiring that a process request all of its required resources at one time and blocking the process until all requests can be granted simultaneously.

No preemption –

This condition can be prevented in several ways: first, if a process holding certain resources is denied a further request, that process must release its original resources and if necessary, request them again together with the additional resource.

Circular Wait –

The circular wait condition can be prevented by defining a linear ordering of resource types.

* Deadlock Avoidance –

– Do not start a process if its demands might lead to deadlock.

– Do not grant an incremental resource request to a process if this allocation might lead to deadlock.

* Determination of a safe state *

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C-A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
0	1	1

Available vector V

a] Initial state

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C - A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
6	2	3

Available vector V

b) P_2 runs to completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C - A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
7	2	3

Available vector V

c) P1 runs to completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

C - A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
9	3	4

Available vector V

d) P_3 runs to completion

Fig – Determination of a safe state

* Determination of an unsafe state *

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
1	1	2

Available vector V

a) Initial state

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix C

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix A

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector R

R1	R2	R3
0	1	1

Available vector V

b) P1 requests one unit each of R1 and R3

Fig – Determination of an Unsafe State

* Deadlock Detection –

Deadlock prevention strategies are very conservative; they solve the problem of deadlock by limiting access to resources and by imposing restrictions on processes.

* Deadlock Detection Algorithm –

1] Mark each process that has a row in the Allocation matrix of all zeros.

2] Initialize a temporary vector W to equal the available vector.

3] Find an index i such that process i is currently unmarked and the ith row of Q is less than or equal to W. That is, $Q_{ik} \leq W_k$, for $1 \leq k \leq m$; if no such row is found, terminate the algorithm.

4] If such a row is found, mark process i and add the corresponding row of the allocation matrix to W. That is, set $W_k = W_k + A_{ik}$, for $1 \leq k \leq m$. Return to step 3.

For Example –

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	0	1

Available vector

Fig – Example of Deadlock Detection

* An Integrated Deadlock strategy –

– Group resources into a number of different resource classes.

– Use the linear ordering strategy defined previously for the prevention of circular wait to prevent deadlocks between resource classes.

– Within a resource class, use the algorithm i.e. most appropriate for that class.

As an example of this technique, consider the following classes of resources.

1) Swappable space –

Blocks of memory on secondary storage for use in swapping processes.

2) Process resources –

Assignable devices, such as tape drives and files.

3) Main memory –

Assignable to processes in pages or segments.

4) Internal resources –

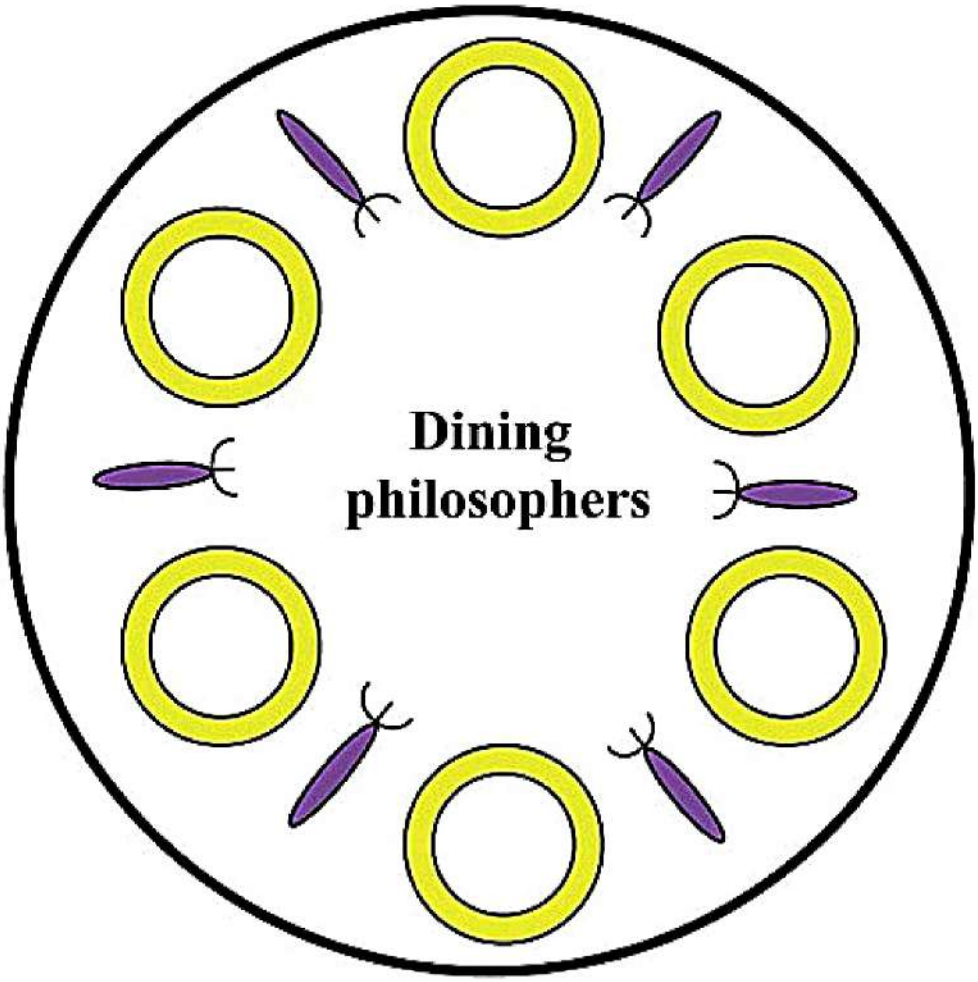
Such as I/O channels.

The order of the preceding list represents the order in which resources are assigned. The order is a reasonable one, considering the sequence of steps that a process may follow during its lifetime. Within each class, the following strategies could be used.

– Swappable space
– Process resources
– Main memory
– Internal resources

* Dining Philosophers Problem –

The eating arrangements are simple : A round table on which is set a large serving bowl of spaghetti, five plates, one for each philosopher, and five forks. A philosopher wishing to eat goes to his or her assigned place at the table and, using the two forks on either side of the plate, takes and eats some spaghetti. The problem : device a ritual (algorithm) that will allow the philosophers to eat. The algorithm must satisfy mutual exclusion (no two philosophers can use the same fork at the same time) while avoiding deadlock & starvation (in this case, the term has literal as well as algorithmic meaning.



DATE:

* solution using semaphores -

* A First solution to the Dining Philosophers Problem -

```
/* program dining philosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
    while (true) {
        think ();
        wait (fork [i]);
        wait (fork [(i+1) mod 5]);
        eat ();
        signal (fork [(i+1) mod 5]);
        signal (fork [i]);
    }
}
void main ()
{
    parbegin (philosopher (0), philosopher (1),
        philosopher (2), philosopher (3),
        philosopher (4));
}
```

* Unix Concurrency Mechanism -

UNIX provides a variety of mechanisms for interprocessor communication and synchronization. Here, we look at the most important of these:

1) Pipes -

When a pipe is created, it is given a fixed size in bytes. When a process attempts to write into the pipe, the write request is immediately executed if there is sufficient room; otherwise the process is blocked. similarly, a reading process is blocked

DATE :

if it attempts to read more bytes than are currently in the pipe; otherwise the read request is immediately executed. The OS enforces mutual exclusion; that is, only one process can access a pipe at a time.

There are two types of pipes: named & unnamed. Only related processes can shared unnamed pipes, while either related or unrelated processes can share named pipes.

2] Messages —

A message is a block of bytes with an accompanying type. UNIX provides msgsnd and msgrcv system calls for processes to engage in message passing. Associated with each process is a message queue, which functions like a mailbox.

3] Shared Memory —

The Fasted form of interprocess communication provided in UNIX is shared memory. This is a common block of virtual memory shared by multiple processes.

4] Semaphores —

The semaphore system calls in UNIX system V are a generalization of the semWait and semSignal primitives: several operations can be performed simultaneously and the increment & decrement operations can be values greater than 1.

5] Signals :-

A signal is a software mechanism that informs a process of the occurrence of asynchronous events. A signal is similar to a hardware interrupt but does not employ priorities. That is, all signals are treated equally; signals that occure ~~that~~ at the

same time are presented to a process one at a time, with no particular ordering.

Unit - IV - Memory Management

* Memory Management Requirements -

While surveying the various mechanisms and policies associated with memory management, it is helpful to keep in mind the requirements that memory management is intended to satisfy. [LIST 93] suggests five requirements.

- Relocation
- Protection
- Sharing
- Logical organisation
- Physical organisation

* Memory Management Terms -

Frame :-

A fixed-length block of main memory.

A fixed-length block of data that resides in secondary memory (such as disk). A page of data may temporarily be copied into a frame of main memory.

A variable-length block of data that resides in secondary memory. An entire segment may temporarily be copied into a available region of main memory (segmentation) or that segment may be divided into pages which can be individually copied into main memory (combined segmentation & paging).

* Addressing Requirements for a process –

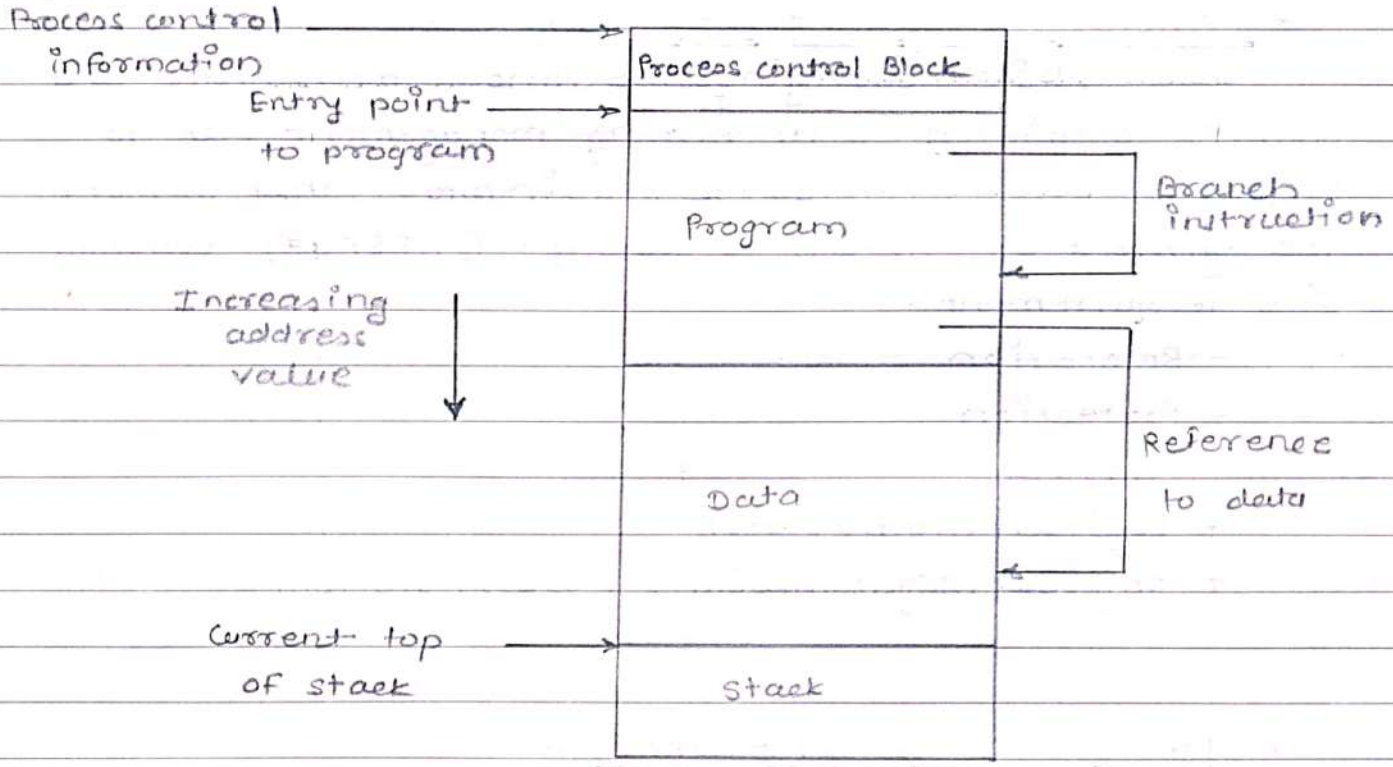


Fig – Addressing Requirements for a process

Note that the memory protection requirement must be satisfied by the processor (hardware) rather than the operating system (software).

* Physical organisation –

The main memory available for a program plus its data may be insufficient. In that case the programmer must engage in practice known as overlaying, in which the program and data are organised in such a way that various modules can be assigned the same region of memory, with a main program responsible for switching the modules in and out as needed. Even with the aid of compiler tools, overlay programming waste programmers time.

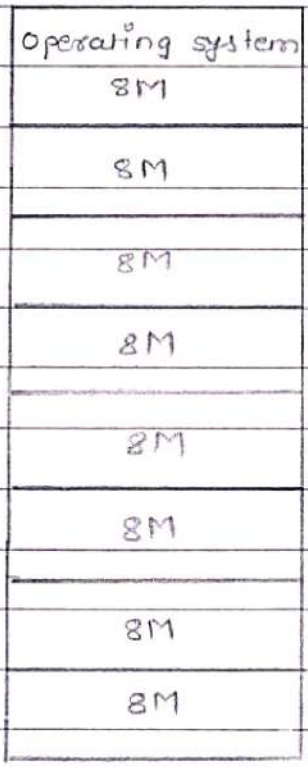
* Memory Partitioning *

Memory Management Techniques –

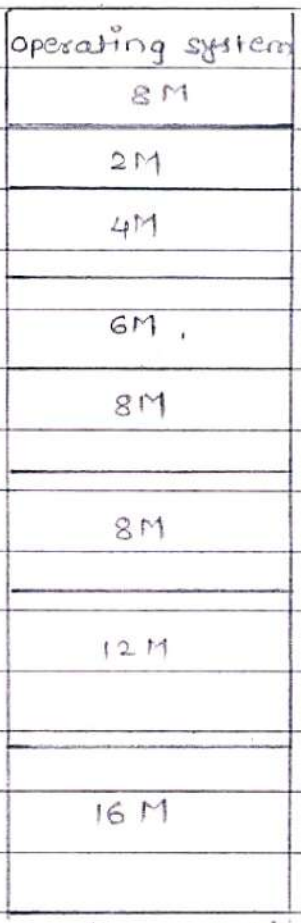
Technique	Description	Strengths	Weaknesses
Fixed partitioning	Main memory is divided into a no. of static partitions at system generation time. A process may be loaded into a partition of equal / greater size.	Simple to implement, little operating system overhead.	Inefficient use of memory due to internal fragmentation; max. no. of active processes is fixed.
Dynamic Partitioning	Partitions are created dynamically, so that each process is loaded into a partition of exactly the same size as that process.	No internal fragmentation; more efficient use of main memory.	Inefficient use of processor due to the need for compaction to counter external fragmentation.
Simple paging	Main memory is divided into a no. of equal size frames. Each process is divided into no. of equal size pages of the same length as frames. a process is loaded by loading all of its pages into available, not necessarily contiguous, frames.	No external fragementation.	A small amount of internal fragementation.
Simple segmentation	Each process is divided into a number of	No internal fragementation;	External fragementation.

Technique	Description	Strengths	Weaknesses
	segments. A process is loaded by loading all of its segments into dynamic partitions that need not be contiguous.	Improved memory utilisation & reduced overhead compared to dynamic partitioning.	
Virtual memory paging	As with simple paging, except that it is not necessary to load all of the pages of a process. Nonresident pages that are needed are brought in later automatically.	No external fragmentation; higher degree of multiprogramming; large virtual address space.	Overhead of complex memory management.
Virtual memory segmentation	As with simple segmentation, except that it is not necessary to load all of the segments of a process. Nonresident segments that are needed are brought in later automatically.	No internal fragmentation; higher degree of multiprogramming; large virtual address space; protection & sharing support	Overhead of complex memory management

* Fixed Partitioning *



a] Equal-size partitions



b) Unequal size partitions

Main memory utilization is extremly inefficient. Any program, no matter how small, occupies an entire partition. In our example, there may be a program whose length is less than 2 Mbytes. yet it occupies an 8-Mbytes partition whenever it is swapped in. This phenomenon, in which there is wasted space internal to a partition due to the fact that the block of data loaded is smaller than the partition, is referred to as internal fragmentation.

* Placement Algorithm –

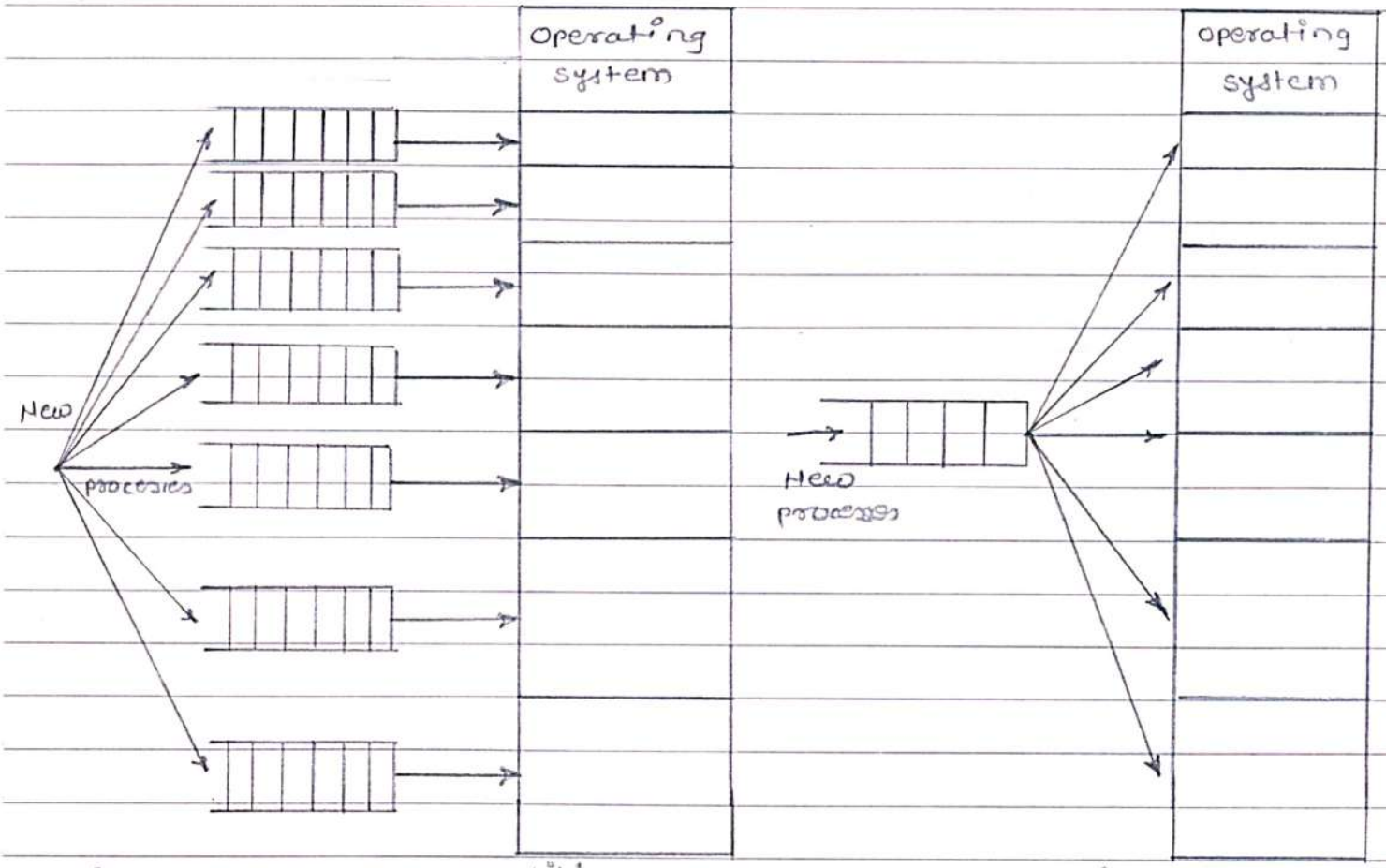


Fig – Memory Assignment for Fixed Partioning

The use of fixed partitioning is almost unknown today. One example of a successful operating system that did use this technique was an early IBM mainframe operating system, OS/MFT (Multiprogramming with a Fixed Number of Tasks.)

* Dynamic Partitioning *

As time goes on, memory becomes more and more fragmented, and memory utilization declines. This phenomenon is referred to as external fragmentation, indicating that the memory that is external to all partitions becomes increasingly fragmented. This is in

contrast to internal fragmentation.

One technique for overcoming external fragmentation is compaction. From time to time, the operating system shifts the processes so that they are contiguous & so that all of the free memory is together in one block.

* Placed Algorithm –

Three placement algorithms that might be considered are best-fit, first-fit and next-fit. All, of course, are limited to choosing among free blocks of main memory that are equal to or larger than the processes to be brought in. Best-fit chooses the block that is closest in size to the request. First-Fit begins to scan memory from the beginning and chooses the first available block that is large enough. Next fit begins to scan memory from the location of the last placement, and chooses the next available block that is large enough.

Fig. shows an example memory configuration after a number of placement and swapping-out operations. The last block that was used was a 22-Mbyte block from which a 14-Mbyte partition was created. Fig. (b) shows the difference between the best-, first-, and next-fit placement algorithms in satisfying a 16-Mbyte allocation request. Best-fit will search the entire list of available blocks & make use of the 18-Mbyte block, leaving a 2-Mbyte fragment. First-fit results in a 6 M-byte fragment, and next-fit results in a 20-Mbyte fragment.

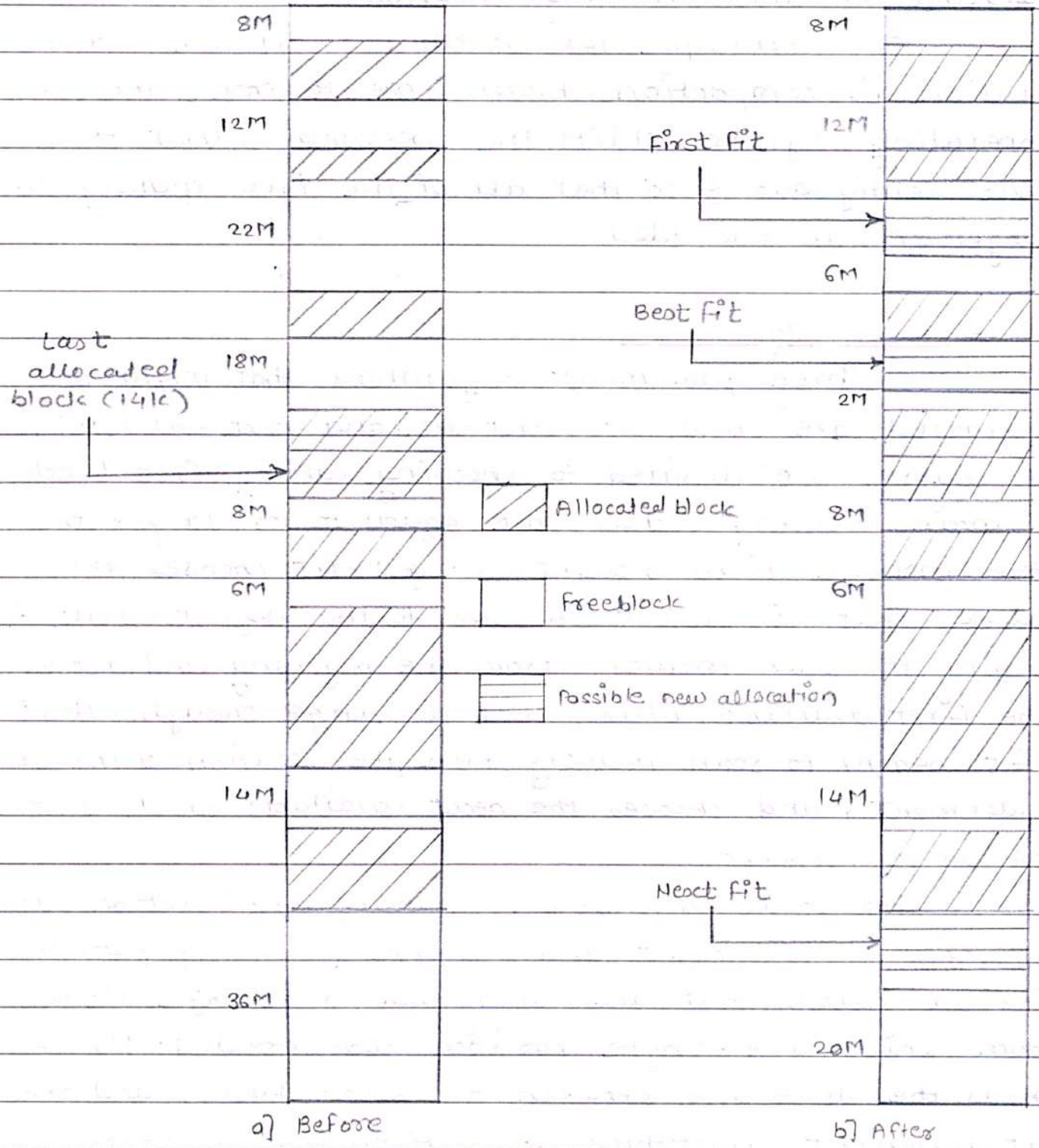


Fig - Example Memory Configuration before & after allocation of 16-Mbyte block

* Buddy System –

Both fixed and dynamic partitioning schemes have drawbacks. A fixed partitioning scheme limits the number of active processes & may use space inefficiently if there is a poor match between available partition sizes & process sizes. A dynamic partitioning scheme is more complex to maintain and includes the overhead of compaction. An interesting compromise is the buddy system ([KNUT 97], [PETE 77]).

In a buddy system, memory blocks are available of size 2^K words, $L \leq K \leq U$,
where

2^L = smallest size block that is allocated.
2^U = Largest size block that is allocated; generally 2^U is the size of the entire memory available for allocation.

Block Fig. gives an example using a 1-Mbyte initial block. The first request, A, is for 100 Kbytes, for which a 128 K block is needed. The initial block is divided into two 512K buddies. The first of these divided into two 256K buddies, and the first of these is divided into two 128K buddies, one of which is allocated to A. The next request B, requires a 256K block. Such a block is already available & is allocated. The process continues with splitting and coalescing occuring as needed. Note that when E is released, two 128K buddies are coalesced into a 256K block, which is immediately coalesced with its buddy.

1-Mbyte block	1M				
Request 100K	A=128K	128K	256K	512K	
Request 240K	A=128K	128K	B=256K	512K	
Request 64K	A=128K	C=64K	64K	B=256K	512K
Request 256K	A=128K	C=64K	64K	B=256K	D=256K	256K
Release B	A=128K	C=64K	64K	256K	D=256K	256K
Release A	128K	C=64K	64K	256K	D=256K	256K
Request 75K	E=128K	C=64K	64K	256K	D=256K	256K
Release C	E=128K	128K	256K	D=256K	256K
Release E	512K			D=256K	256K
Release D	1M				

Fig – Example of a Buddy System

* Relocation –

A logical address is a reference to a memory location independant of the current assignment of data to memory; a translation must be made to a physical address before the memory access can be achieved. A relative address is a particular example of logical address, in which the address is expressed as a location relative to some known point, usually a value in a processor register. A physical address, or absolute address, is an actual location in main memory.

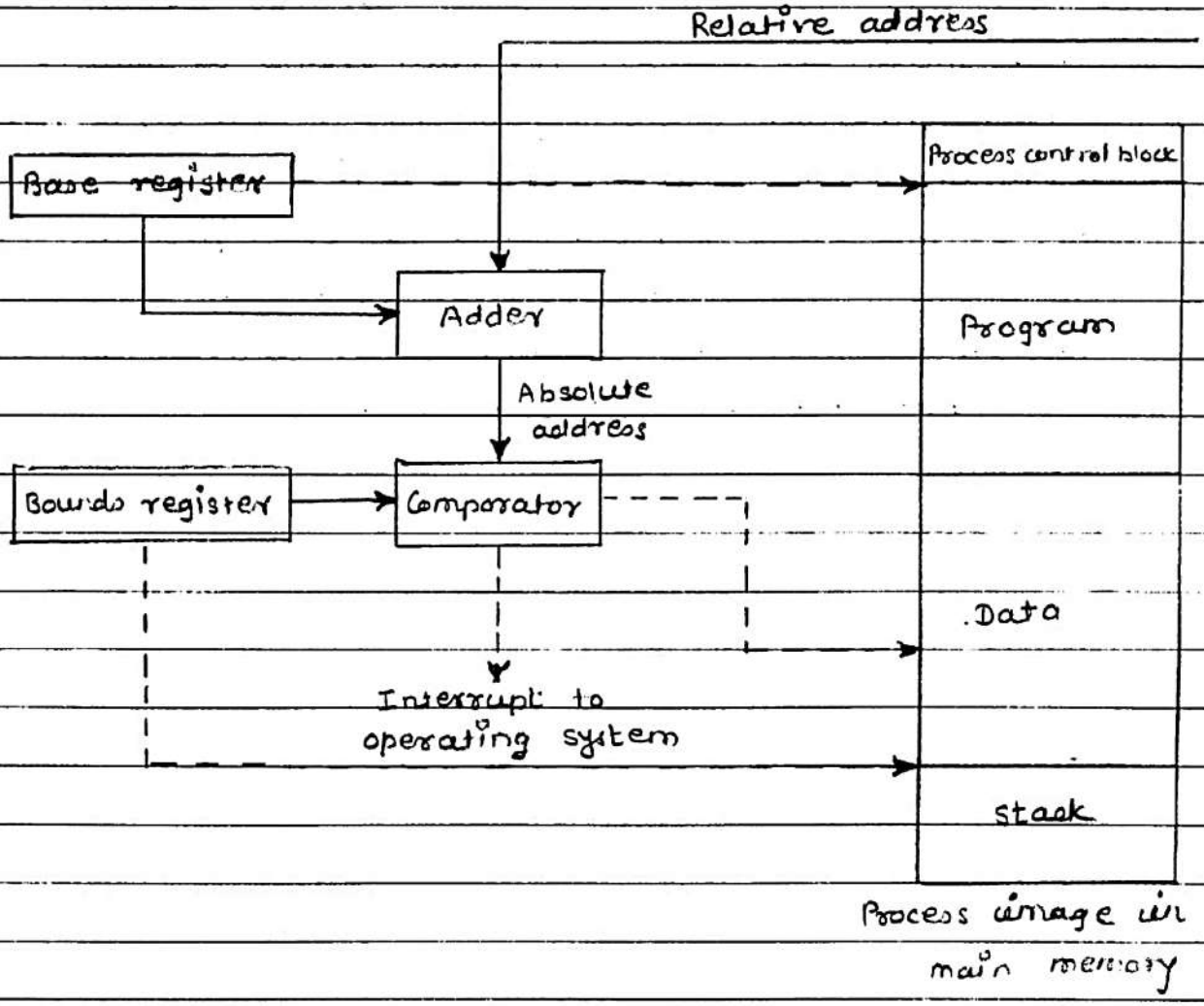


Fig - Hardware support for Relocation

* Paging –

Both unequal fixed-size and variable-size partitions are inefficient in the use of memory; the former results in internal fragmentation, the latter in external fragmentation. Suppose, however, that main memory is partitioned into equal-fixed size chunks that are relatively small, and that each process is also divided into small fixed size chunks of the same size. Then the chunks of a process, known as pages, could be assigned to available chunks of memory, known as frames, or page frames. We show in this section that the wasted space in memory for each process is due to internal fragmentation consisting of only a fraction of the last page of a process. There is no external fragmentation.

The OS maintains a page table for each process. The page table shows a frame location for each page of the process. Within the program, each logical address consists of a page number and an offset within the page.

Frame Number	Main Memory	Main Memory	Main Memory
0		A.0	A.0
1		A.1	A.1
2		A.2	A.2
3		A.3	A.3
4			B.0
5			B.1
6			B.2
7			
8			
9			
10			
11			
12			
13			
14			

a] Fifteen available frames b) Load Process A c) Load process B

	Main memory		Main memory		Main memory
0	A·0	0	A·0	0	A·0
1	A·1	1	A·1	1	A·1
2	A·2	2	A·2	2	A·2
3	A·3	3	A·3	3	A·3
4	B·0	4		4	D·0
5	B·1	5		5	D·1
6	B·2	6		6	D·2
7	C·0	7	C·0	7	C·0
8	C·1	8	C·1	8	C·1
9	C·2	9	C·2	9	C·2
10	C·3	10	C·3	10	C·3
11		11		11	D·3
12		12		12	D·4
13		13		13	
14		14		14	
	d) Load process C		e) Swap out B		f) Load process D

Fig (A) – Assignment of Process to Free Frames

	Process A Page Table		Process B Page Table		Process C Page Table		Process D Page Table		Free frame list
0	0	0	—	0	7	0	4		13
1	1	1	—	1	8	1	5		14
2	2	2	—	2	9	2	6		
3	3			3	10	3	11		
						4	12		

Fig – Data structures for the example of Fig (A) at time epoch (f)

* 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.

Whereas paging is invisible to the programmer, segmentation is usually visible & is provided as a convenience for organising programs and data. Typically the programmer or compiler will assign programs and data to different segments. For purpose of modular programming, the program or data may be further broken down into multiple segments. The principle inconvenience of this service is that the programmer must be aware of the maximum segment size limitation.

* Virtual Memory *

* Virtual Memory Technology —

Virtual memory	A storage allocation scheme in which secondary memory can be addressed as though it were part of main memory. The addresses a program may use to reference memory are distinguished from the addresses the memory system uses to identify physical storage sites and program generated addresses are translated automatically to the corresponding machine addresses. The size of virtual storage

	is limited by the addressing scheme of the computer system & by the amount of secondary memory available & not by the actual number of main storage locations.
Virtual address	The address assigned to a location in virtual memory to allow that location to be accessed as though it were part of main memory.
Virtual address space	The virtual storage assigned to a process.
Address space	The range of memory addresses available to a process.
Real address	The address of a storage location in main memory.

* Hardware And Control Structures –

Comparing simple paging & simple segmentation, on the one hand, with fixed & dynamic partitioning, on the other, we see the foundation for a fundamental breakthrough in memory management. Two characteristics of paging and segmentation are the keys to this breakthrough:

1] All memory references within a process are logical addresses that are dynamically translated into physical addresses at run time. This means that a process may be swapped in and out of main memory such that it occupies different regions of main memory at different times during the course of execution.

2] A process may be broken up into a number of pieces (pages or segments) and these pieces need not be contiguously located in main memory during execution. The combination of dynamic run-time address translation and the use of a page or segment table permits this.

* Paging –

The term virtual memory is usually associated with systems that employ paging, although virtual memory based on segmentation is also used. The use of paging to achieve virtual memory was first reported for the Atlas computer [KILB62] and soon came into widespread commercial use.

* Page Table Structure –

The basic mechanism for reading a word from memory involves the translation of a virtual, or logical, address, consisting of page number and offset, into a physical address, consisting of frame number and offset, using a page table.

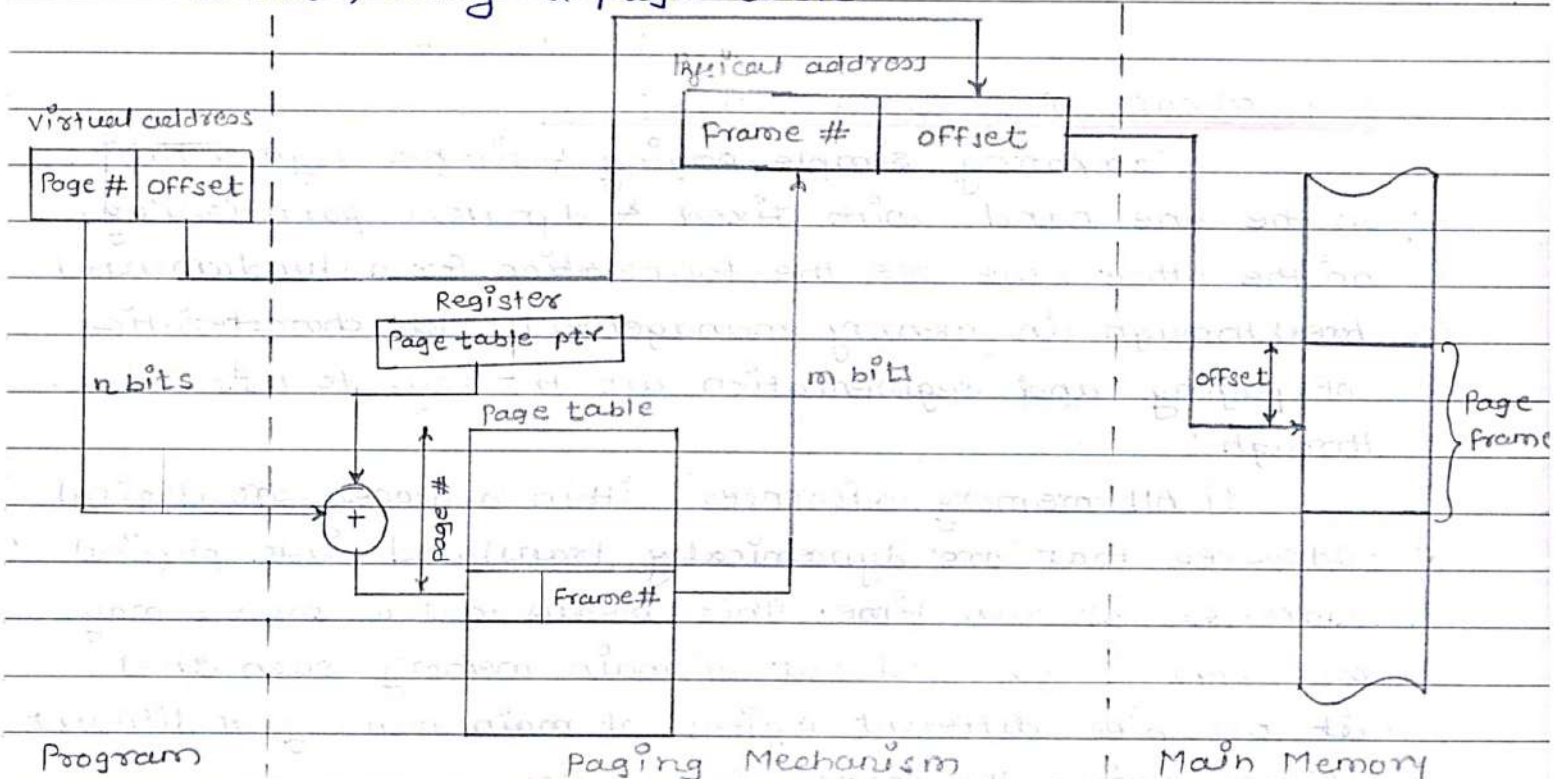


Fig – Address Translation in a Paging System

DATE :

* Translation Lookaside Buffer —

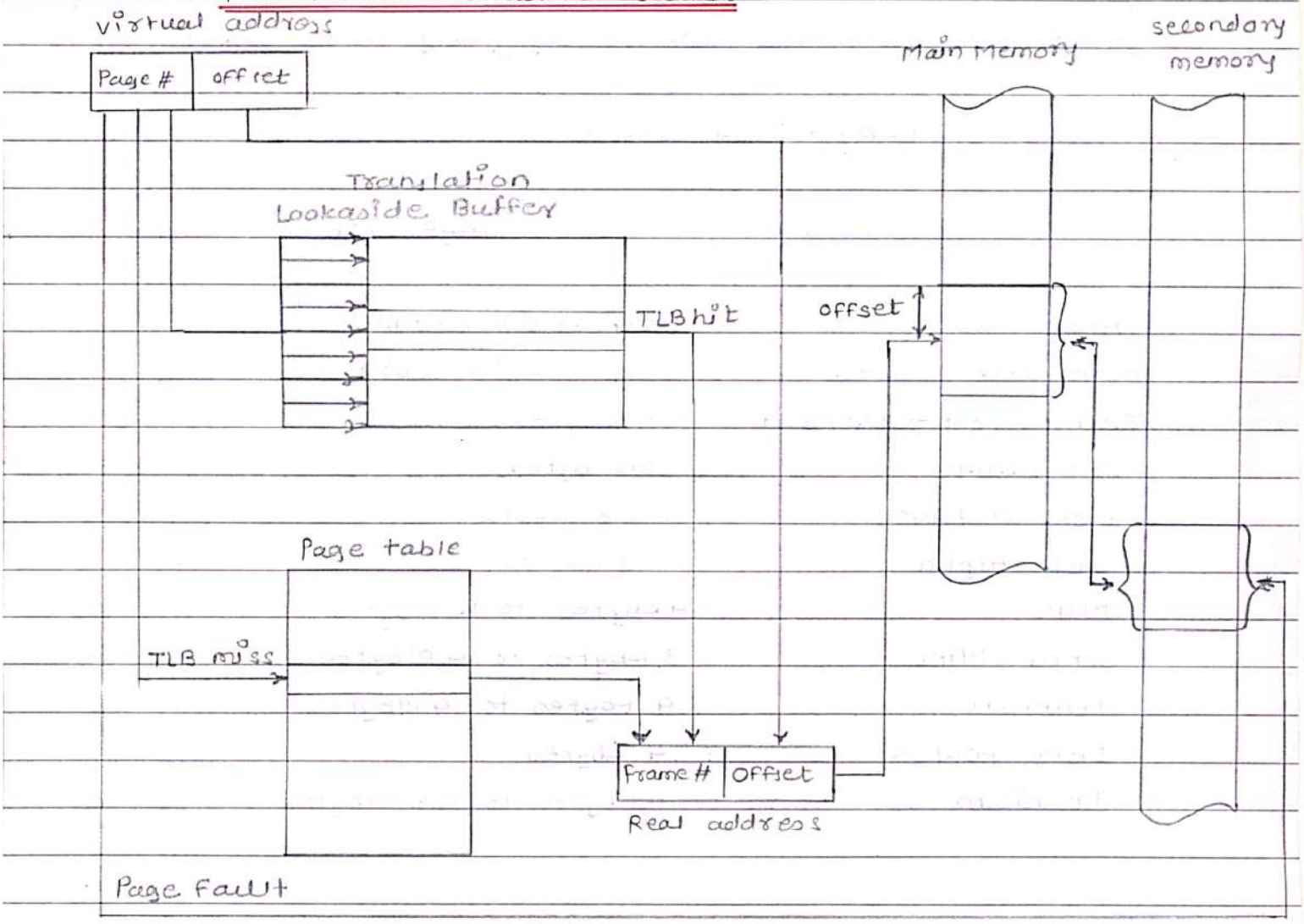


Fig – Use of a Translation Lookaside Buffer

Most virtual memory schemes make use of a special high-speed cache for page table entries, usually called a translation lookaside buffer (TLB). This cache functions in the same way as a memory cache & contains those page table entries that have been most recently used. The organisation of the resulting paging hardware is illustrated in above fig. Given a virtual address, the processor will first examine the TLB. If the desired page table entry is present (TLB hit), then the frame number is retrieved & real address is formed. If the desired page table entry is not found (TLB miss), then the

processor uses the page number to index the process page table & examine the corresponding page table entry.

Table - Example Page sizes

Computer	Page Size
Atlas	512, 48-bit words
Honeywell - Multics	1024, 36-bit words
IBM 370/XA & 370/ESA	4 Kbytes
VAX family	512 bytes
IBM AS/400	512 bytes
DEC Alpha	8 Kbytes
MIPS	4 Kbytes to 16 Mbytes
Ultra SPARC	8 Kbytes to 4 Mbytes
Pentium	4 Kbytes to 4 Mbytes
IBM POWER	4 Kbytes
Itanium	4 Kbytes to 256 Mbytes

* Segmentation -

Virtual Memory Implications -

Segmentation allows the programmer to view memory as consisting of multiple address spaces or segments. Segments may be of unequal, indeed dynamic, size, Memory references consists of a (segment number, offset) form of address.

* Address Translation in a segmentation system -

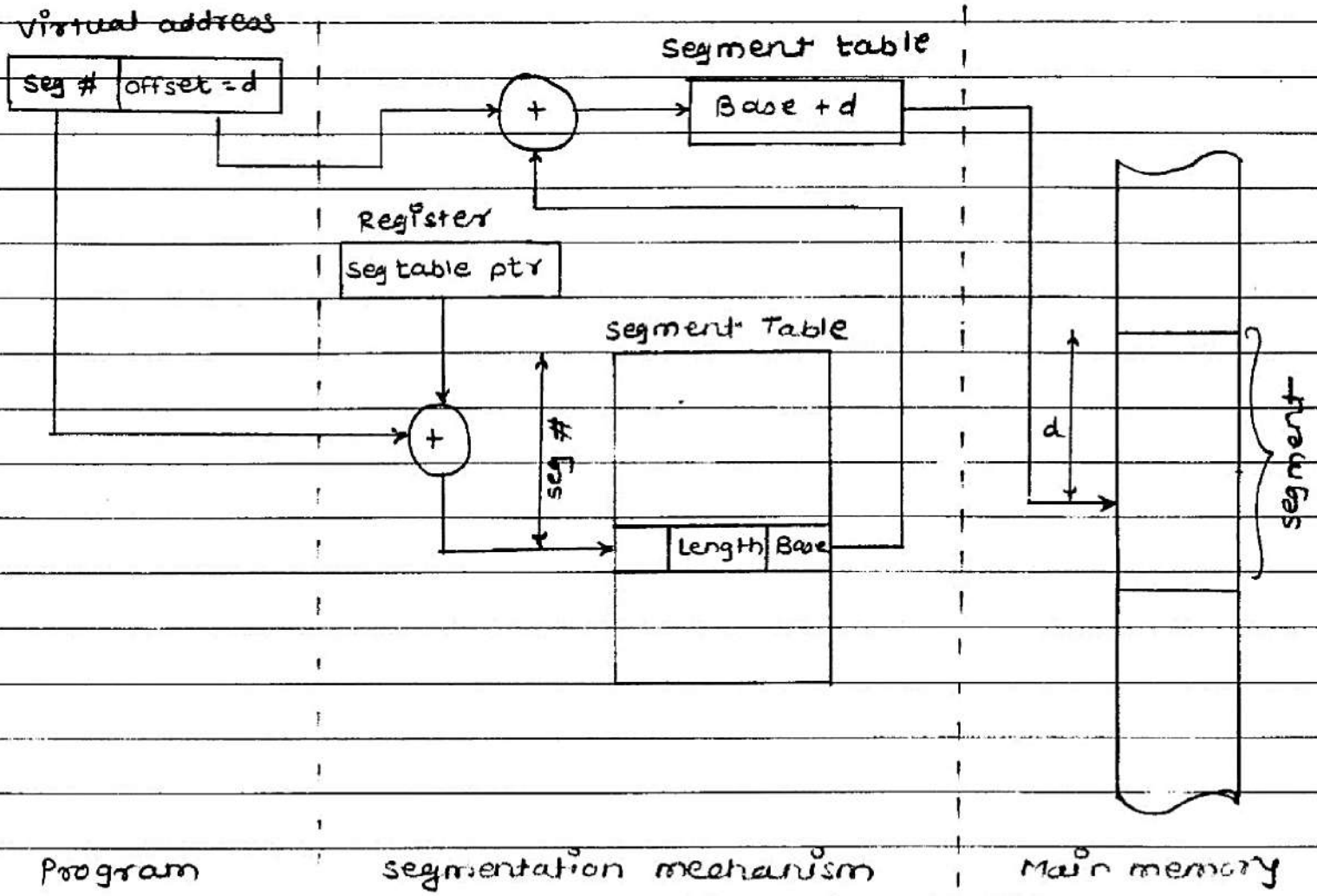


Fig - Address Translation in a segmentation system

* Protection and Sharing *

* Protection Relationships between segments -

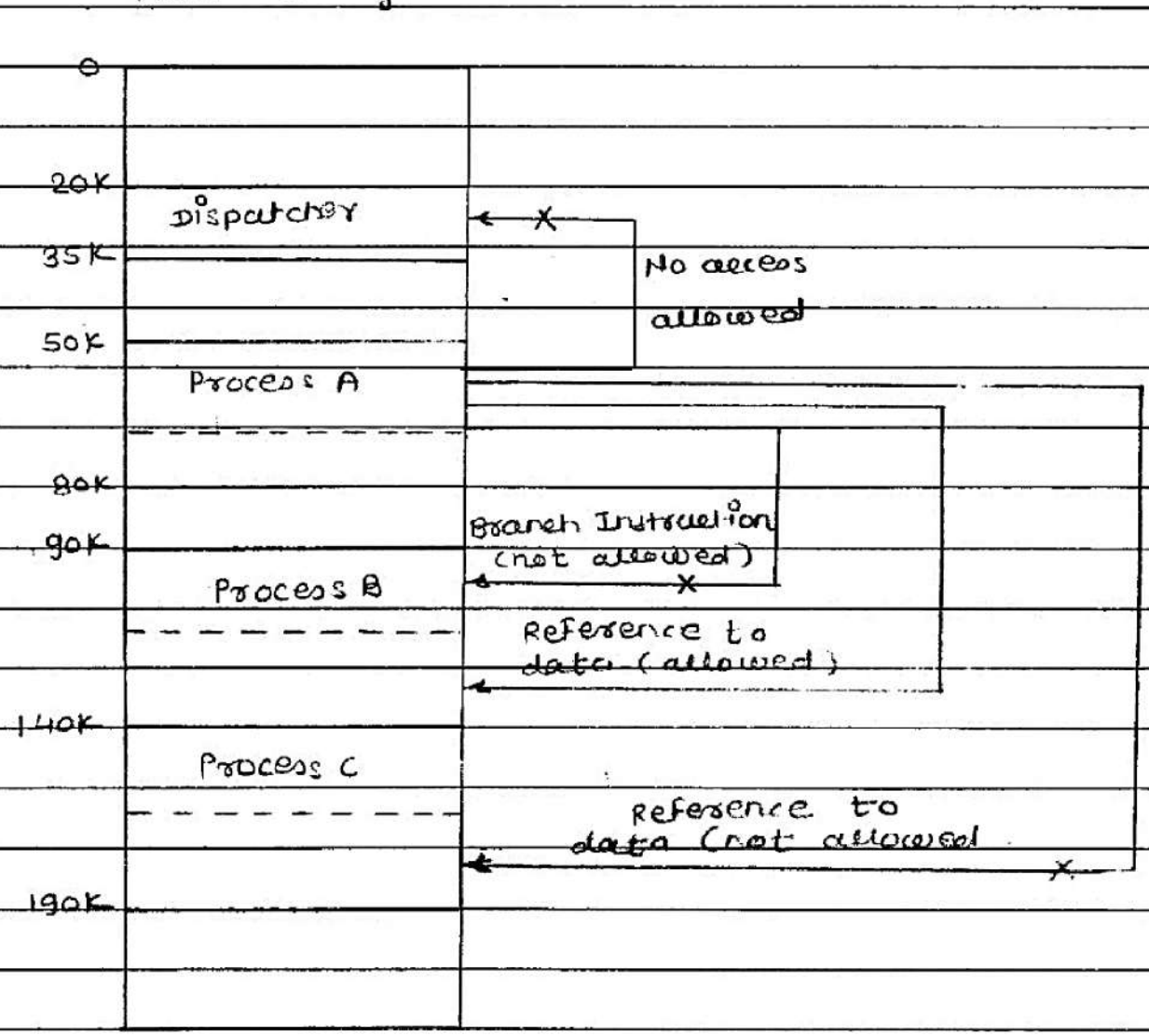


Fig - Protection Relationships between segments.

* Operating System Software —

The design of the memory management portion of an operating system depends on three fundamental areas of choice :

- Whether or not to use virtual memory techniques.
- The use of paging or segmentation or both
- The algorithms employed for various aspects of memory management.

The choice made in first two areas depend on the hardware platform available. Thus, earlier UNIX implementations did not provide virtual memory because the processors on which the system ran did not support paging or segmentation. Neither of these techniques is practical without hardware support for address translation and other basic functions.

* Operating System Policies for Virtual Memory —

Fetch Policy	Resident set Management
Demand	Resident set size
Prepaging	Fixed
Placement Policy	Variable
Replacement Policy	Replacement scope
Basic Algorithms	Global
Optimal	Local
Least recently used (LRU)	Cleaning Policy
First-in-First-out (FIFO)	Demand
Clock	Precleaning
Page buffering	
	Load control
	Degree of multiprogramming

* Replacement Policy –

Basic Algorithms – Regardless of the resident set management strategy, there are certain basic algorithms that are used for the selection of a page to replace. Replacement algorithms that have been discussed in the literature include –

- Optimal
- Least recently used (LRU)
- First-in-first-out (FIFO)
- Clock.

* Behaviour of Four Page Replacement Algorithms –

Page address stream	2	3	2	1	5	2	4	5	3	2	5	2
OPT	2	2	2	2	2	2	4	4	4	2	2	2
		3	3	3	3	3	3	3	3	3	3	3
				1	5	5	5	5	5	5	5	5
					F		F			F		
LRU	2	2	2	2	2	2	2	2	3	3	3	3
		3	3	3	5	5	5	5	5	5	5	5
				1	1	1	4	4	4	2	2	2
					F		F		F	F		
FIFO	2	2	2	2	5	5	5	5	3	3	3	3
		3	3	3	3	2	2	2	2	2	5	5
				1	1	1	4	4	4	4	4	2
					F	F	F		F		F	F
CLOCK →	2*	2*	2*	2*	5*	5* →	5* →	5*	3*	3* →	3* →	3*
		3*	3*	3* →	3	2*	2*	2* →	2 →	2*	2	2*
	→	→		1*	1 →	1	4*	4*	4	4	5*	5*
					F	F	F		F		F	

F = Page fault occurring after the frame allocation is initially filled.

Fig – Behaviour of four page replacement algorithms

* Page Buffering -

Although LRU and the clock policies are superior to FIFO, they both involve complexity & overhead not suffered with FIFO. In addition, there is the related issue that the cost of replacing a page that has been modified is greater than for one that has not, because the former must be written back out to secondary memory.

* Resident set Management -

	Local Replacement	Global Replacement
Fixed Allocation	- Number of frames allocated to process is fixed. - Page to be replaced is chosen from among the frame allocated to that process.	- Not possible
Variable Allocation	- The number of frams allocated to a process may be changed from time to time, to maintain the working set of a process. - Page to be replaced is chosen from among frames allocated to that process.	- Page to be replaced is chosen from all available frames in main memory; this causes the size of resident set of processes to vary.

* LINUX MEMORY MANAGEMENT *

Linux Virtual Memory –

Virtual Memory Addressing – Linux makes use of a three level page table structure, consisting of the following types of table (each individual table is the size of one page):

* Page directory –

An active process has a single page directory that is the size of one page. Each entry in the page directory points to one page of the page middle directory. The page directory must be in main memory for an active process.

* Page middle directory –

The page middle directory may span multiple pages. Each entry in the page middle directory points to one page in the page table.

* Page table –

The page table may also span multiple pages. Each page table entry refers to one virtual page of the process.

* Address Translation in Linux Virtual Memory Scheme.

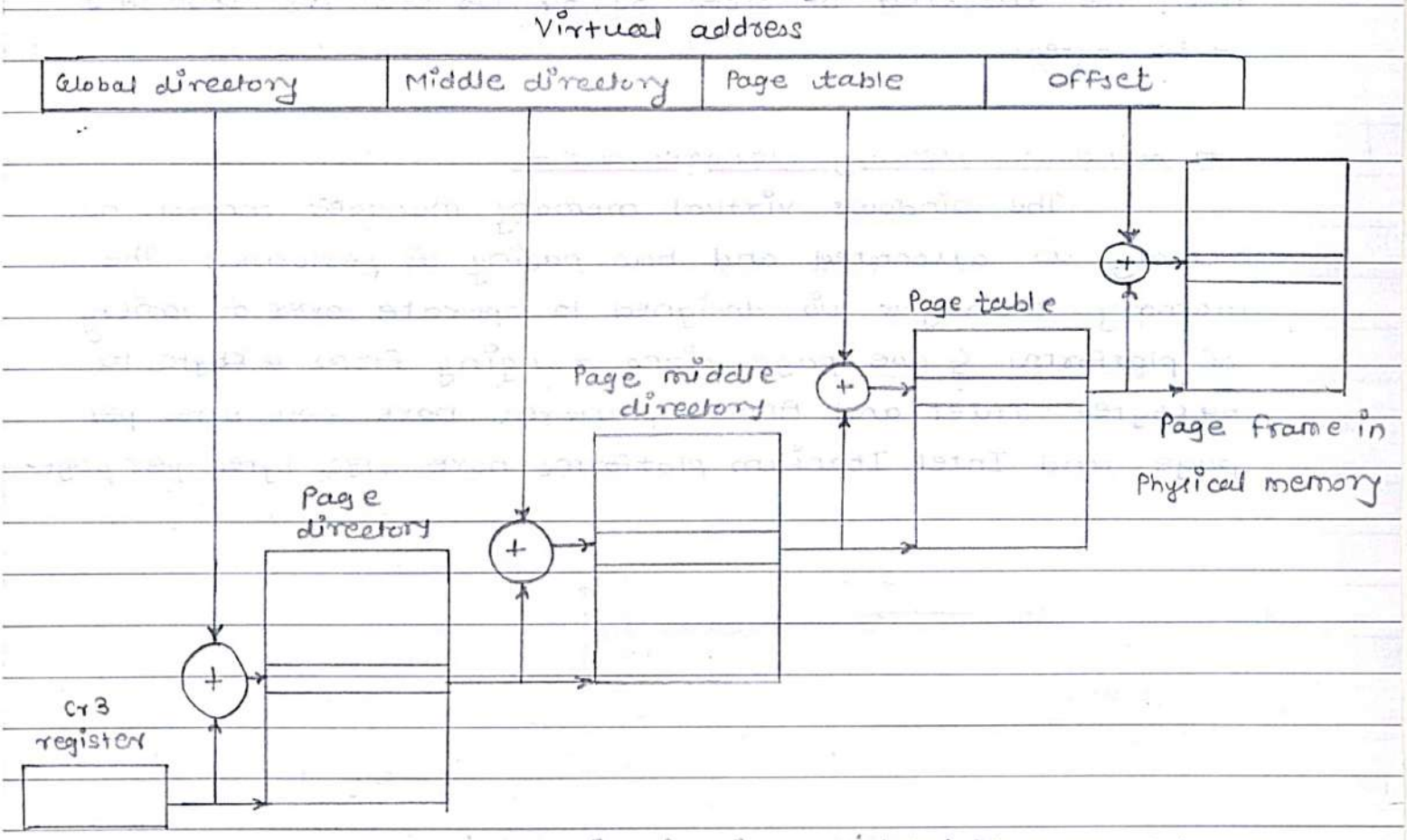


Fig - Address Translation in Linux Virtual Memory Scheme

* Kernel Memory Allocation -

The foundation of kernel memory ~~capability manage~~ allocation for Linux is the page allocation mechanism used for user virtual memory management. As in the virtual memory scheme, a buddy algorithm is used so that memory for the kernel can be allocated & delocated in units of one or more pages. Because the minimum amount of memory that can be allocated in this fashion is one page, the page allocator alone would be inefficient because the kernel requires small short-term memory chunks in odd sizes. To accommodate these small chunks, Linux uses a scheme known as slab allocation [BONW94] within an allocated page. On a Pentium/x86 machine,

the page size is 4 Kbytes, and chunks within a page may be allocated of sizes 32, 64, 128, 252, 508, 2040 and 4080 bytes.

* Windows Memory Management –

The windows virtual memory manager controls how memory is allocated and how paging is performed. The memory manager is designed to operate over a variety of platforms & use page sizes ranging from 4 Kbytes to 64 Kbytes : Intel and AMD64 platforms have 4096 bytes per page and Intel Itanium platforms have 8192 bytes per page.

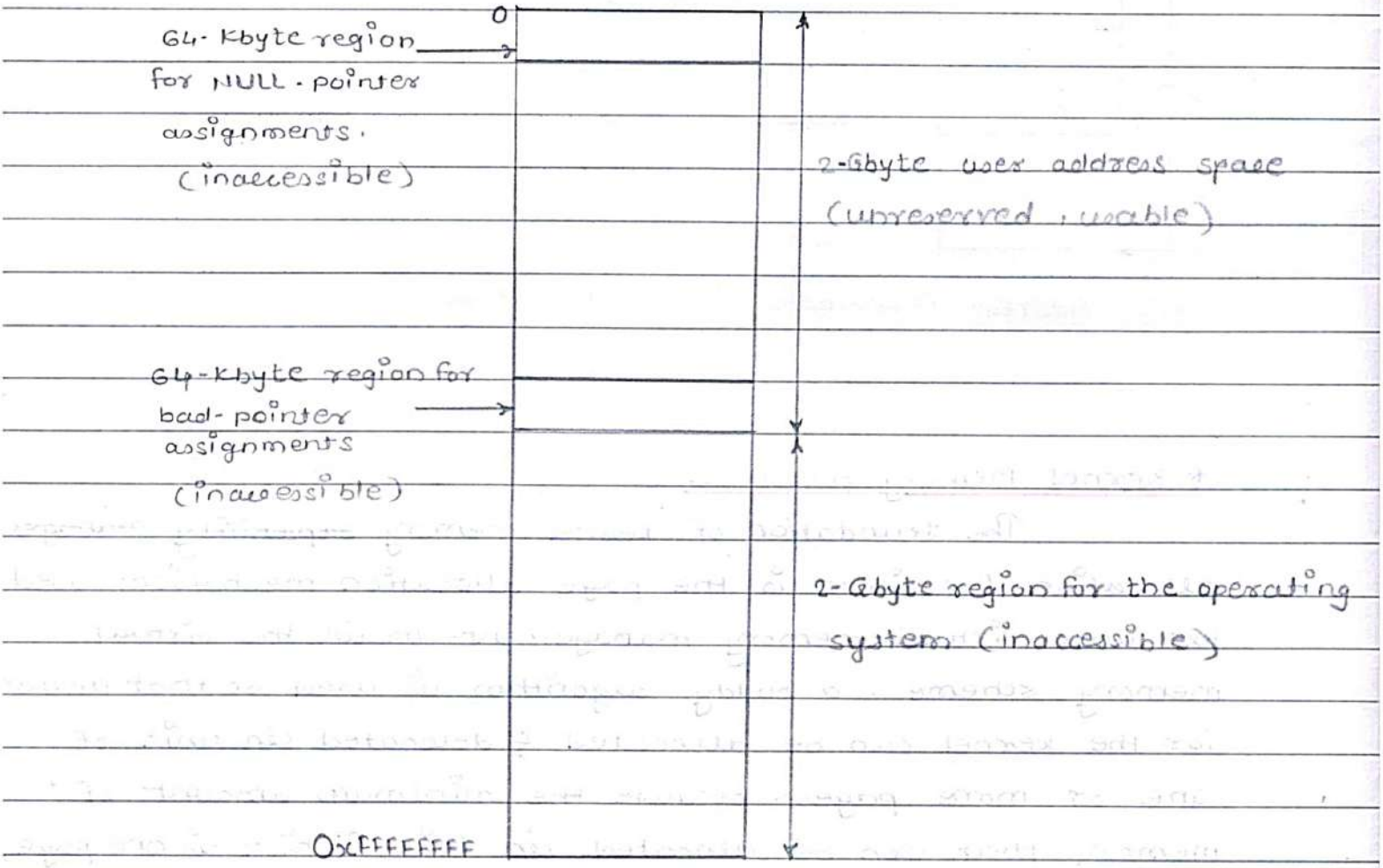


Fig - Windows Default 32-Bit Virtual Address Space

Fig. shows the default virtual address space seen by a normal 32-bit user process. It consists of four regions:

1) 0x00000000 to 0x0000FFFF —

Set aside to help programmers catch NULL-pointer assignments.

2) 0x00010000 to 0x7FFEFFFF —

Available user address space. This space is divided into pages that may be loaded into main memory.

3) 0x7FF0000 to 0x7FFFFFFF —

A guard page inaccessible to the user. This page makes it easier for the operating system to check on out-of bounds pointer references.

4) 0x80000000 to 0xFFFFFFFF —

System address space. This 2-Gbyte process is used for the Windows Executive, kernel, and device drivers.

On 64-bit platforms, 8TB of user address space is available in Windows Vista.

END

UNIT - V

* I/O Management And Disk Scheduling *

* I/O devices :-

External devices that engage in I/O with computers systems can be roughly grouped into three categories :

*** Human readable -**

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

*** Machine readable -**

Suitable for communicating with electronic equipment, Examples are disk drives, USB keys, sensors, controllers and actuators.

*** Communication -**

Suitable for communicating with remote devices. Examples are digital line drivers and modems.

There are great differences across classes and even substantial differences within each class. Among the key differences are the following :

1) Data rate -

There may be differences of several orders of magnitude between the data transfer rates.

2) Application -

The use to which a device is put has an influence on the software and policies in the operating system and supporting utilities.

3) Complexity of control -

A printer requires a relatively simple control interface. A disk is much more complex. The effect of these differences on the operating system is filtered to some extent by the complexity of the I/O module that controls the device.

4) Unit of transfer –

Data may be transferred as a stream of bytes or characters (e.g. terminal I/O) or in larger blocks (e.g disk I/O).

5) Data representation –

Different data encoding schemes are used by different devices, including differences in character code and parity conventions.

6) Error conditions –

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.

* Organization of the I/O Function –

1) Programmed I/O –

The processor issues an I/O command, on behalf of a process, to an I/O module; that process then busy waits for the operation to be completed before proceeding.

Table – I/O Technique –

	No interrupts	Use of interrupts
I/O-to-memory transfer through processor	Programmed I/O	Interrupt-driven I/O
Direct I/O-to-memory transfer		Direct memory access (DMA)

2) Interrupt-driven I/O –

The processor issues an I/O command on behalf of a process. There are two possibilities. If the I/O instruction from the process is nonblocking, then the processor continue

to execute instructions from the process that issued the I/O command. If the I/O instruction is blocking, then the next instruction that the processor executes is from the OS, which will put the current process in a blocked state and schedule another process.

3] Direct memory access (DMA) :-

A DMA module controls the exchange of data between main memory and an I/O module. The processor sends a request for the transfer of a block of data to the DMA module & is interrupted only after the entire block has been transferred.

* Direct Memory Access –

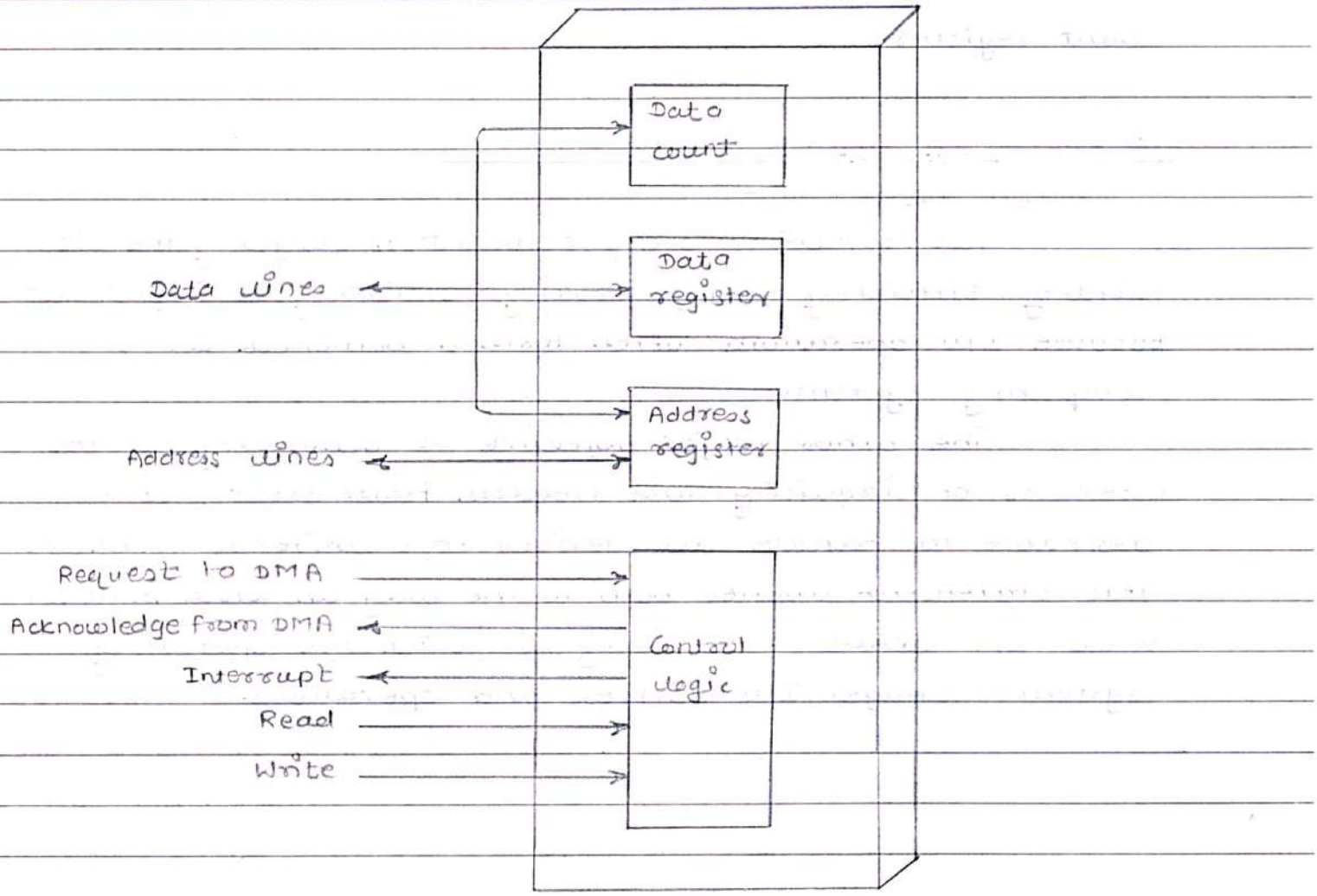


Fig- Typical DMA Block Diagram

The DMA technique works as follows. 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 the read or write control line between the processor and the DMA module.
- The address of the I/O device involved, communicated on the data lines.
- The starting location in memory to read from or write to, communicated on the data lines & stored by the DMA module in its address register.
- The number of words to be read or written, again communicated via the data lines and stored in the data count register.

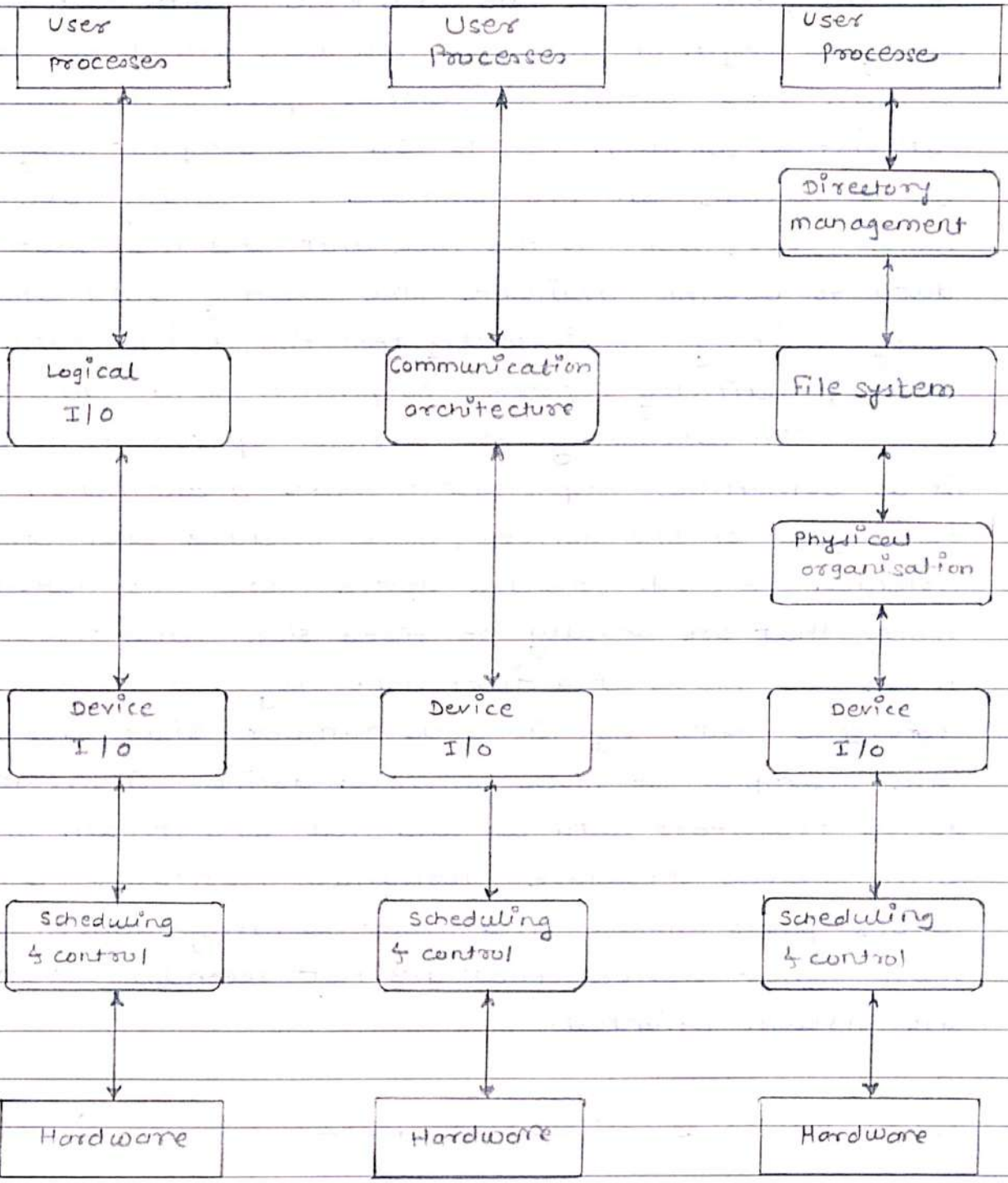
* Operating System Design Issues –

Design Objectives –

Two objectives are paramount in designing the I/O facility: efficiency and generality. Efficiency is important because I/O operations often form a bottleneck in a computing system.

The other major objective is generality. In the interests of simplicity and freedom from error, it is desirable to handle all devices in a uniform manner. This statement applies both to the way in which processes view I/O devices & the way in which the operating system manages I/O devices and operations.

* Logical Structure of the I/O Function –



a] Local peripheral device b] Communications port c) Filesystem

fig - A model of I/O organization

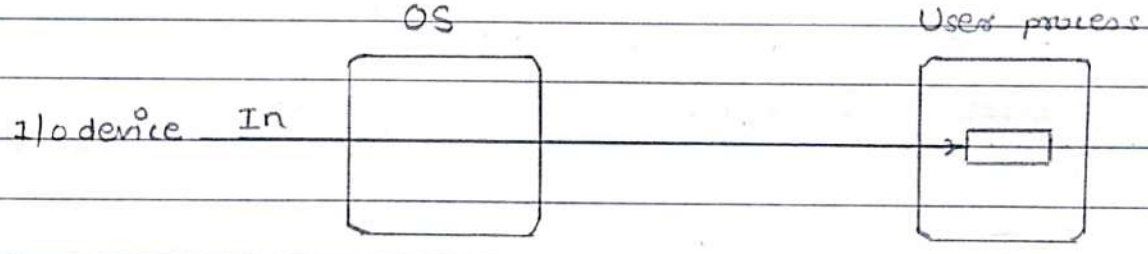
DATE :

* I/O Buffering –

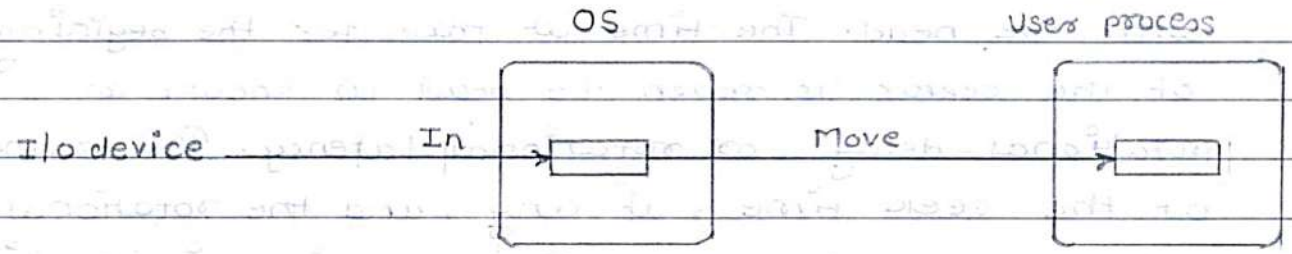
Suppose that a user process wishes to read blocks of data from a disk one at a time, with each block having a length of 512 bytes. The data are to be read into a data area within the address space of the user process at virtual location 1000 to 1511. The simplest way would be to execute an I/O command (something like Read-Block [1000, disk] to the disk unit and then wait for the data to become available. The waiting could either be busy waiting (continuously test the device status) or, more practically, process suspension on an interrupt.

In discussing the various approaches to 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. Generally, it is possible to reference data by its block number. Disks and USB keys are examples of block-oriented devices. 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.

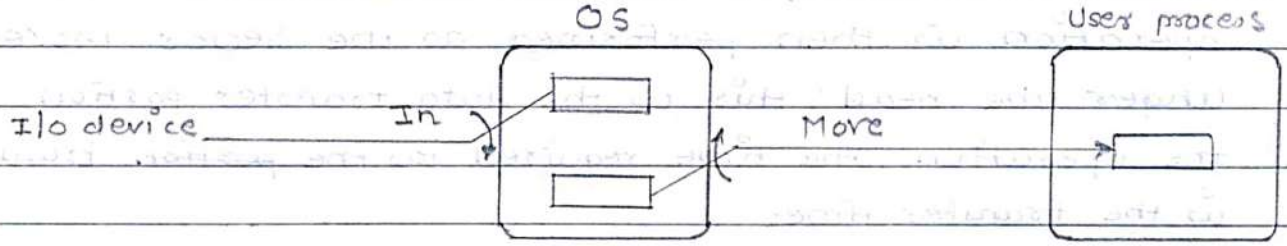
* I/O Buffering Schemes (input) –



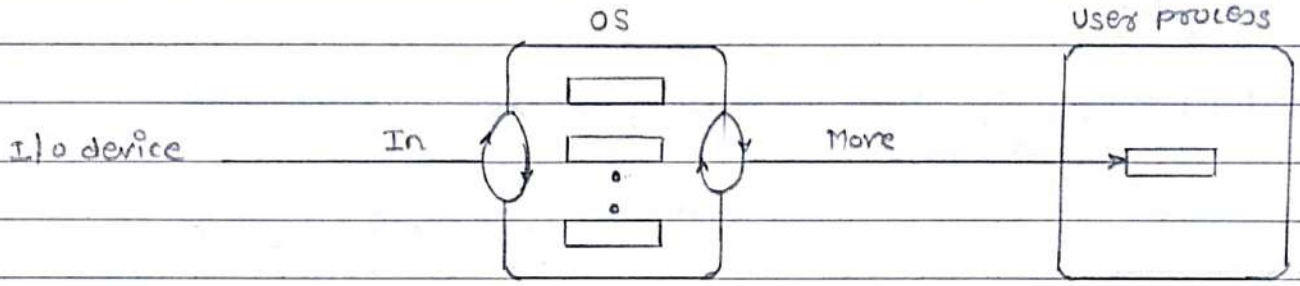
a] no buffering



b) Single buffering



c) Double buffering



d) circular buffering

Fig - I/o Buffering Schemes (input)

* Disk Scheduling *

* Disk Performance Parameters -

When the disk drive is operating, the disk is rotating at constant speed. To read or write, the head must be positioned at the desired track & at the beginning of the desired sector on that track. Track selection involves moving the head in a movable head system or electronically selecting one head on a fixed head system.

On a movable head system, the time it takes to position the head at the track is known as seek time. In either case, once the track is selected, the disk controller waits until the appropriate sector rotates to line up

DATE :

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 portion transfer is the transfer time.

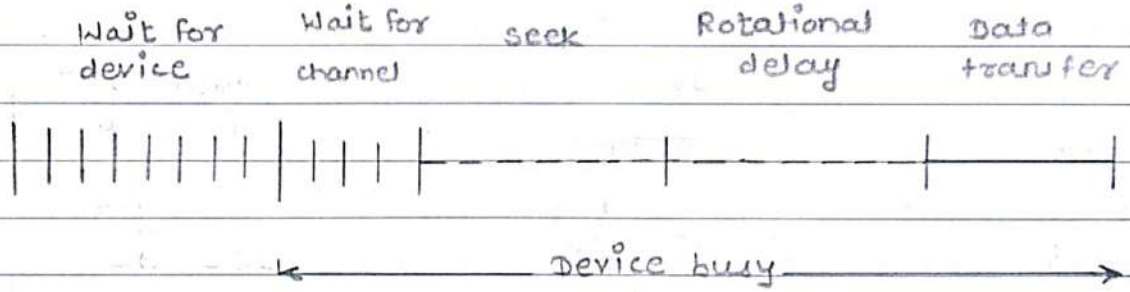
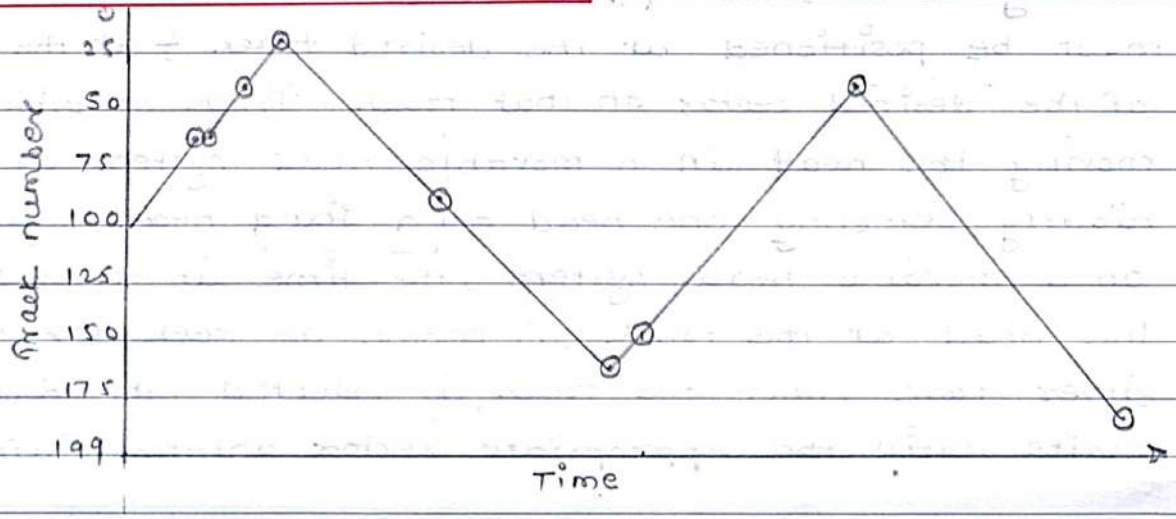


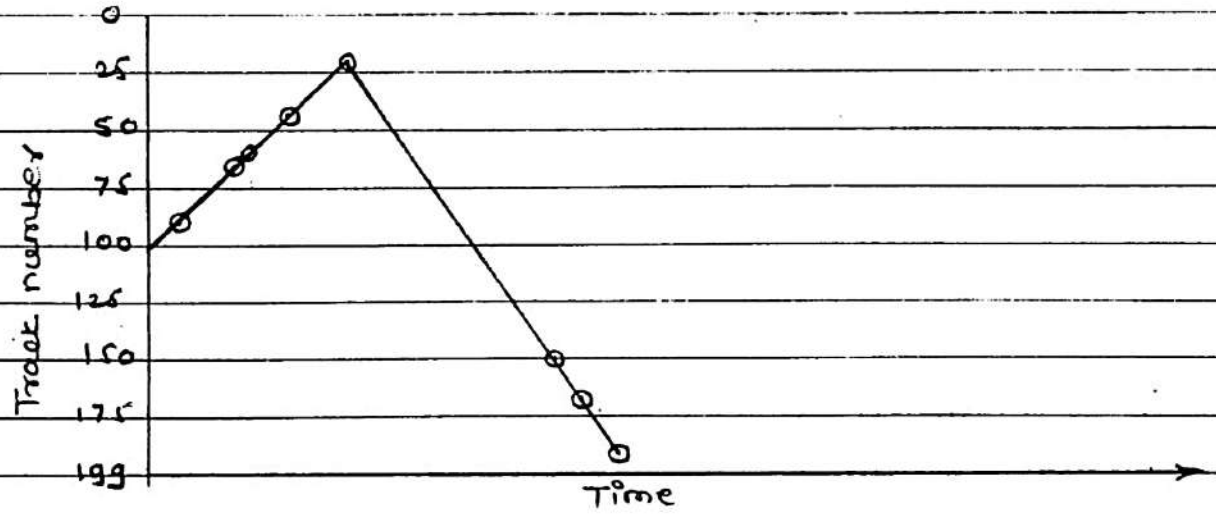
Fig - Timing of a Disk I/O Transfer

* Disk Scheduling Policies -

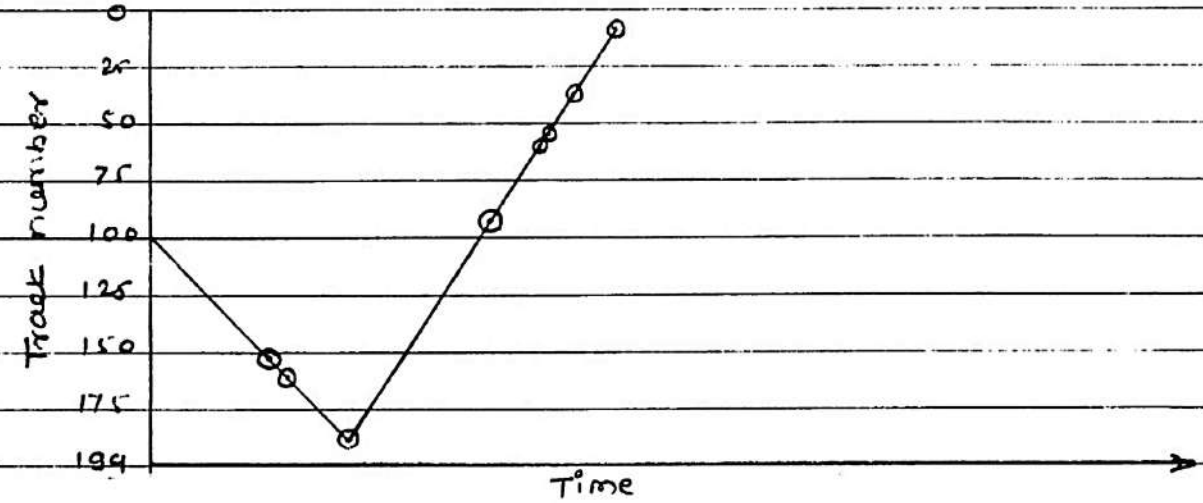
The requested tracks, in the order received by the disk scheduler, are 55, 58, 39, 18, 90, 160, 150, 38, 184.

* First - in - first - out -

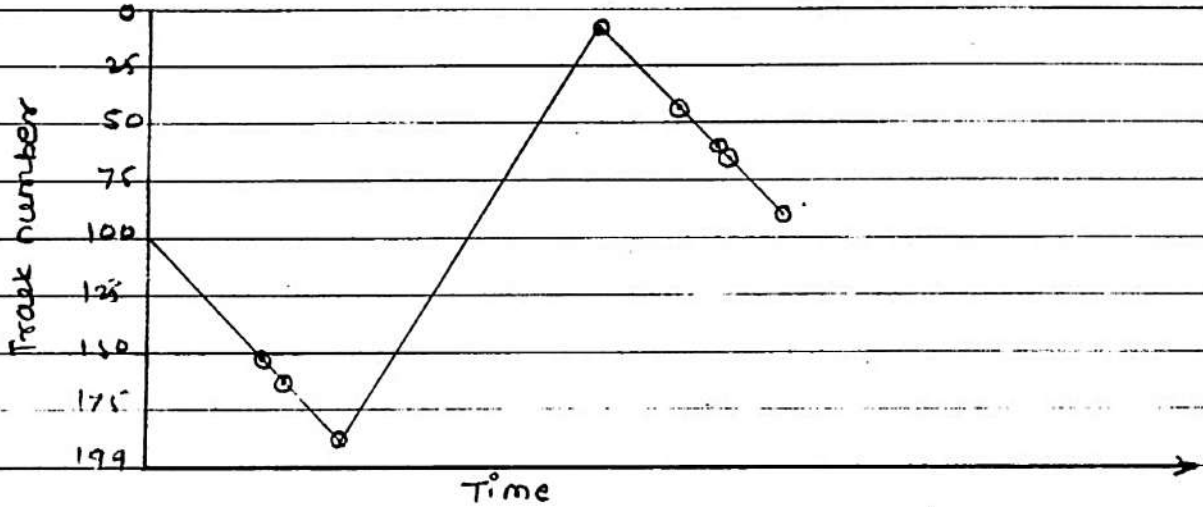




b) SSTF



c) SCAN



d) C-SCAN

Table – Comparison of Disk scheduling Algorithms –

a] FIFO (starting at track 100)		b] SSTF (starting at track 100)		c] SCAN (starting at track 100, in the direction of increasing track number)		d] C-SCAN (starting at track 100, in the direction of increasing track numbers)	
Next track accessed	No. of tracks traversed	Next track accessed	No. of tracks traversed	Next track accessed	No. of tracks traversed	Next track accessed	No. of tracks traversed
55	45	90	10	150	50	150	50
58	3	58	32	160	10	160	10
39	19	55	3	184	24	184	24
18	21	39	16	90	94	18	166
90	72	38	1	58	32	38	20
160	70	18	20	55	3	39	1
150	10	150	132	39	16	55	16
38	112	160	10	38	1	58	3
184	146	184	24	18	20	90	32
Avg. Seek length	55.3	Avg. seek length	27.5	Avg. seek length	27.8	Avg. seek length	35.8

* Table – Disk scheduling algorithm

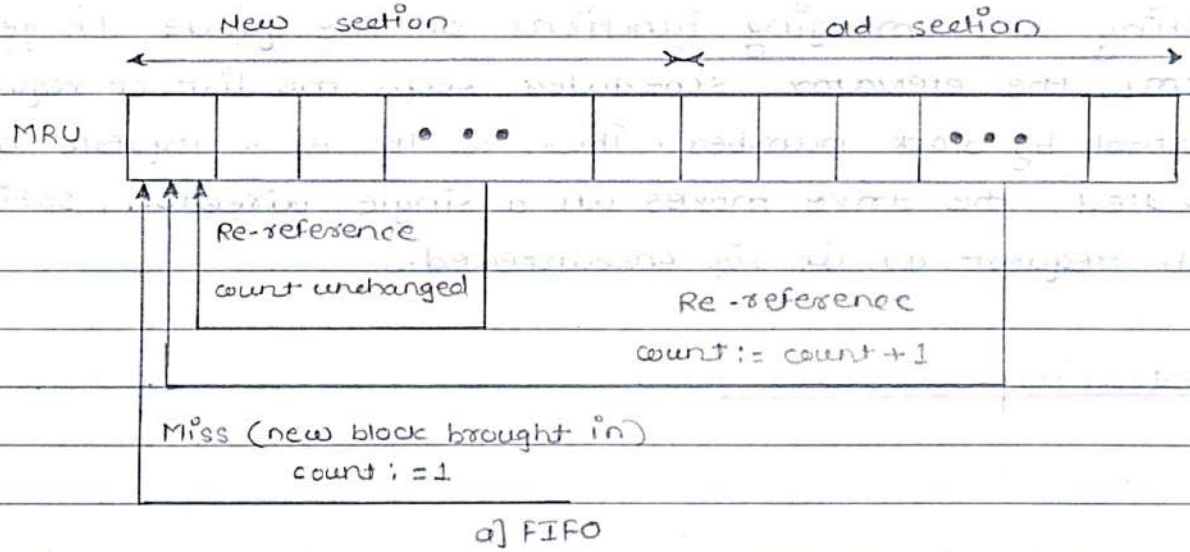
Name	Description	Remarks
Selection according to requestor		
RSS	Random scheduling	For analysis & simulation
FIFO	First in first out	Fairest of them all
PRI	Priority by process	Control outside of disk queue management.
LIFO	Last in first out	Maximize locality & resource utilization.

Name	Description	Remarks
Selection according to requested item		
SSTF SCAN C-SCAN N-Step-SCAN	Shortest service time first Back and forth over disk One way with fast return SCAN of N records at a time	High utilization, small queues Better service distribution Lower service variability Service guarantee
FSCAN	N-step-SCAN with N = queue size at begining of SCAN cycle.	Load sensitive

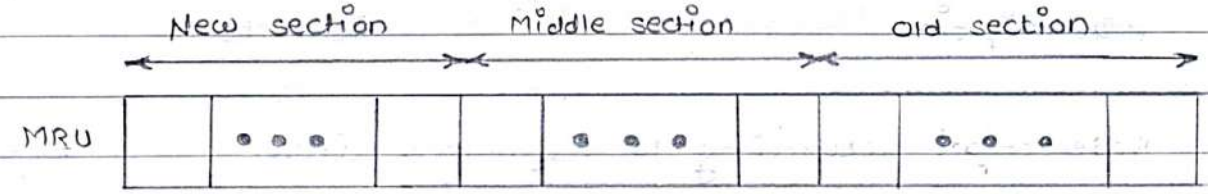
* Disk Cache —

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.

* Design Considerations —



a] FIFO



b] Use of three sections

* Linux I/O –

In general terms, the Linux I/O kernel facility is very similar to that of other UNIX implementation, such as SVR4. The Linux kernel associates a special file with each I/O device driver. Block, character, and network devices are recognized.

* Disk Scheduling –

The default disk scheduler in Linux 2.4 is known as the Linus Elevator, which is a variation on the LOOK algorithm. For Linux 2.6, the Elevator algorithm has been augmented by two additional algorithms: the deadline I/O scheduler and the anticipatory I/O scheduler [LOVE 04].

* The Elevator Scheduler –

The elevator scheduler maintains a single queue for disk read and write requests and performs both sorting and merging functions on the queue. In general terms, the elevator scheduler keeps the list of requests sorted by block number. Thus, as the disk requests are handled, the drive moves in a single direction, satisfying each request as it is encountered.

* Deadline Scheduler –

* Deadline Scheduler -

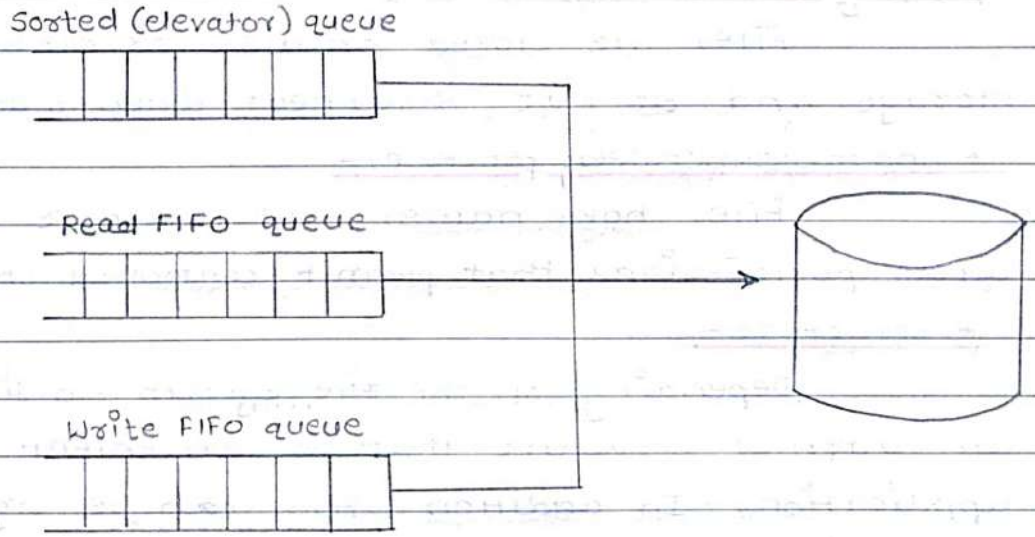


Fig- The Linux Deadline I/O Scheduler

* Anticipatory I/O Scheduler -

The original elevator scheduler & the deadline scheduler both are designed to dispatch a new request as soon as the existing request is satisfied, thus keeping the disk as busy as possible.

* Linux Page Cache -

The kernel maintained a page cache for reads and writes from regular file system files and for virtual memory pages, and a seperate buffer cache for block I/O. For Linux 2.4 and later, there is a single unified page cache that is involved in all traffic between disk and main memory.

* Files and File systems -

From the user's point of view, one of 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-

* Long-term existance -

Files are stored on disk or other secondary storage and do not disappear when a user logs off.

* Shareable 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.

* File Structure -

Four terms are in common use when discussing files.

1) Field -

A field is a basic element of data. An individual field contains a single value, such as an employee's last name, a date or the value of a sensor reading.

2) Record -

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

3) File -

A file is a collection of similar records. The file is treated as a single entity by users and applications and may be referenced by name.

4) Database -

A database is collection of related data.

* 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. [GROS 86] suggests the 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 and from the user's point of view in terms of response time.
- 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 standaradized set of I/O interface routine to user processes.

* File system Architecture -

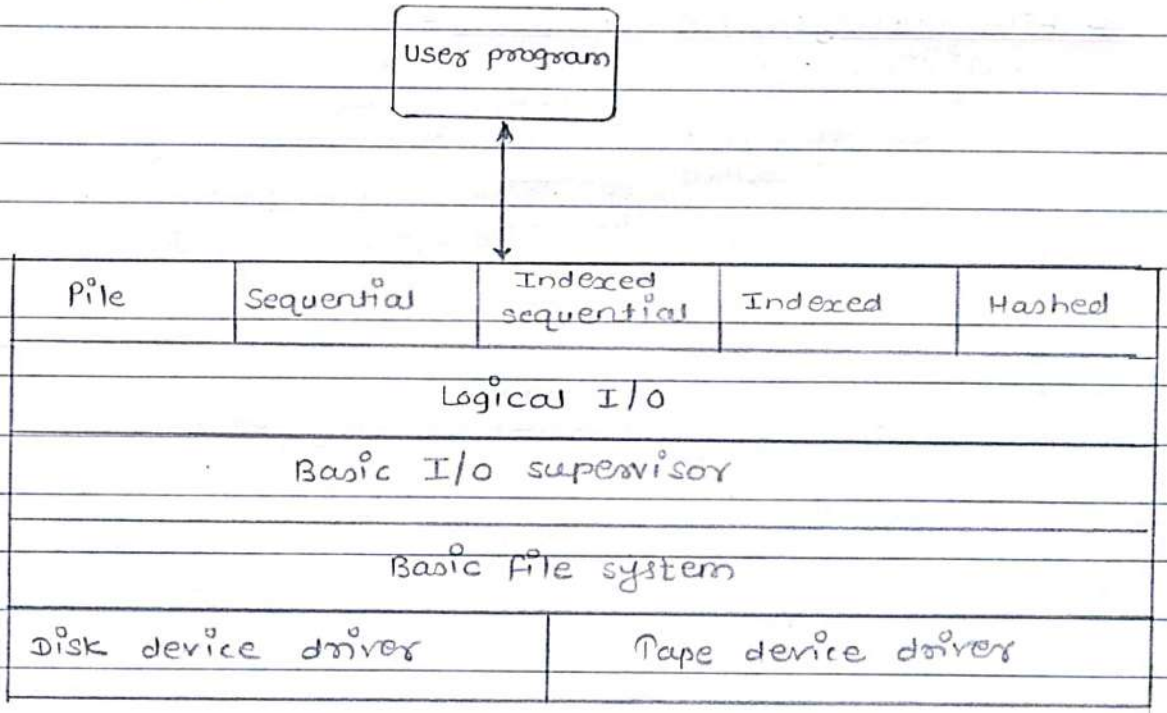


Fig - File system Software Architecture.

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.

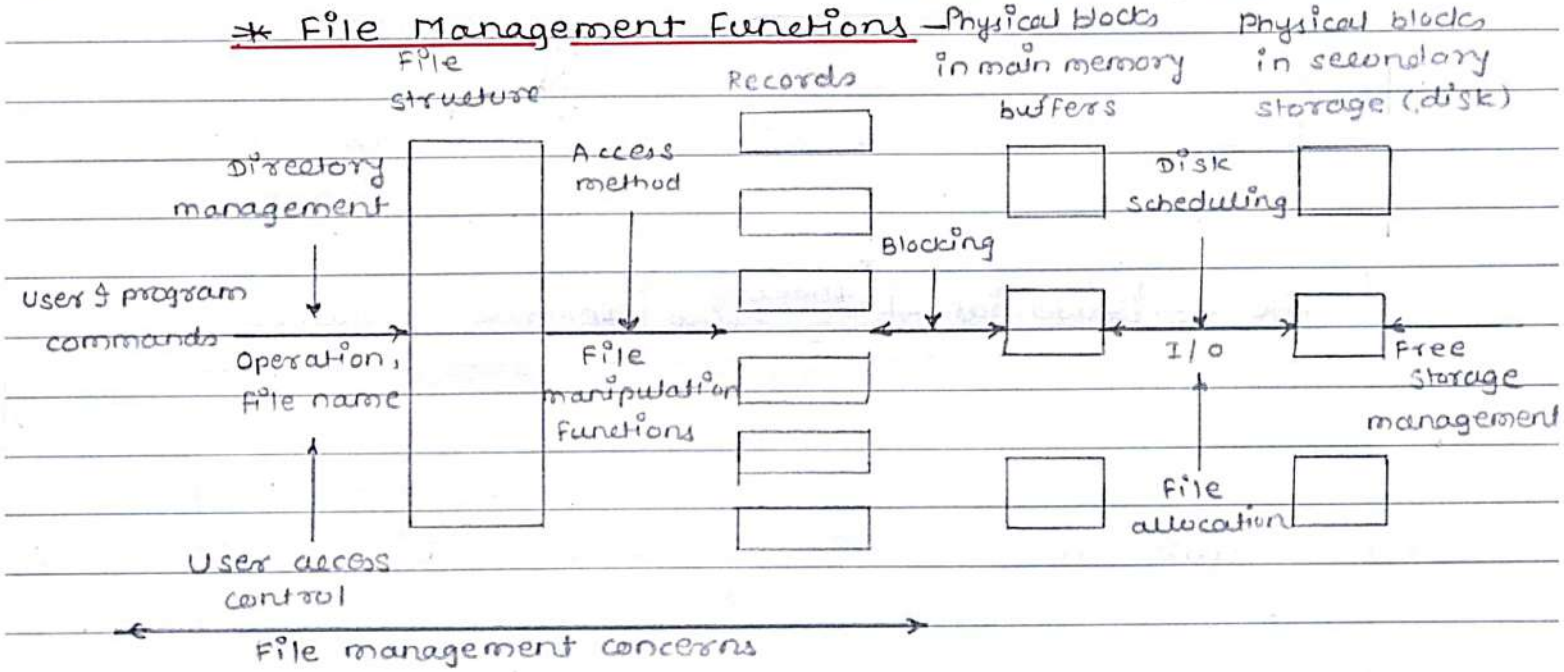
The next level is referred to as the basic file system, or the physical I/O level. This is the primary interface with the environment outside of the computer system.

The basic I/O supervisor is responsible for all file I/O initiation and termination. At this level, control structures are maintained that deal with device I/O, scheduling, and file status.

Logical I/O enables users and applications to access records. Thus, whereas the basic file system deals with blocks of data, the logical I/O module deals with the file records.

The level of the file system closest to the user is often termed the access method. It provides a standard interface between applications and the file systems and devices that hold the data.

* File Management Functions



* File Organization And Access -

In this section, we use term file organization to refer to the logical structuring of the records as determined by the way in which they are accessed. The physical organization of the file on secondary storage depends on the blocking strategy and the file allocation strategy, issues dealt with later.

In choosing a file organization, several criteria are important:

- Short access time
- Ease of update
- Economy of storage
- Simple maintenance
- Reliability.

A file stored on CD-ROM will never be updated, and so ease of update is not an issue.

* The File -

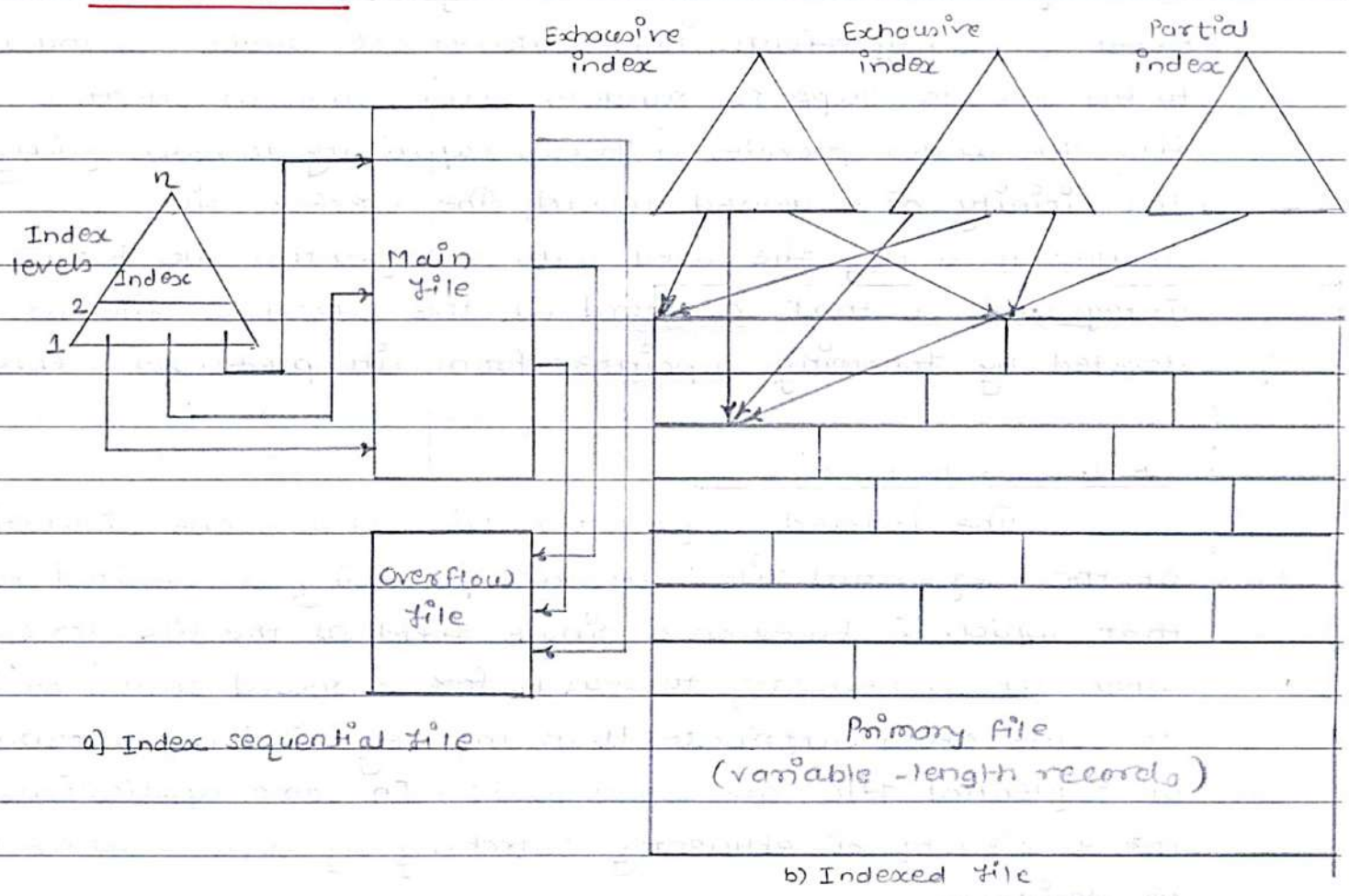


Fig- Common File Organizations

* The Sequential File -

The most common form of file structure is the sequential file. In this type of a file, a fixed format is used for records. All records are of the same length, consisting of the same number of fixed-length fields in a particular order. Because the length & position of each field are known, only the values of fields need to be stored; the field name and length for each field are attributes of the file structure.

An alternative is to organize the sequential file physically as a linked list.

* The Index sequential File -

A popular approach to overcoming the disadvantages of the sequential file is the indexed sequential file. The indexed sequential file maintains the key characteristic of the sequential file; records are organized in sequence based on a key field. Two features are added; an index to the file to support random access, and an overflow file. The index provides a lookup capability to reach quickly the vicinity of a desired record. The overflow file is similar to a log file used with a sequential file but is integrated so that a record in the overflow file is located by following a pointer from its predecessor record.

* The Indexed File -

The indexed sequential file retains one limitation of the sequential file: effective processing is limited to that which is based on a single field of the file, for e.g. when it is necessary to search for a record on the basis of some other attribute than the key field, both forms of sequential file are inadequate. In some applications, the flexibility of efficiently searching by various attributes is desirable.

* The Direct or Hashed File –

The direct, or hashed, file exploits the capability found on disks to access directly any block of a known address. As with sequential and indexed sequential files, a key field is required in each record.

The direct file makes use of hashing on the key value.

* File Directories –

Contents –

Associated with any file management system and collection of files is a file directory. The directory contains information about the files, including attributes, location, and ownership. Much of this information, especially that concerned with storage, is managed by the operating system. The directory is itself a file, accessible by various file management routines. Although some of the information in directories is available to users and applications, this is generally provided indirectly by system routines.

Table – Information Elements of a File Directory.

Basic Information
File name – Name as chosen by creator (user or program). Must be unique within a specific directory.
File type – For e.g. – text, binary, load module, etc.
File organisation – For system that support different organizations.
Address Information
Volume – Indicates device on which file is stored.
Starting Address – Starting physical address on secondary storage (e.g. – cylinder, track & block no. on disk)

Size used – Current size of the file in bytes, words, or blocks
size Allocated – The maximum size of the file.

Access Control Information

Owner – User who is assigned control of this file. The owner may be able to grant / deny access to other users and to change these privileges.

Access Information – A simple version of this element would include the user's name & password for each authorized user.

Permitted Actions – Controls reading, writing, executing, transmitting over a network.

Usage Information

Date created – When file was first placed in directory.

Identity of creator – Usually but not necessarily the current owner.

Date Last Read Access – Date of the last time a record was read.

Identity of last Reader – User who did the reading

Date Last Modified – Date of the last update, insertion or deletion.

Identity of Last Modifier – User who did the Modifying.

Date of Last Back up – Date of the last time the file was backed up on another storage medium.

Current Usage – Information about current activity on the file, such as process / processes that have the file open, whether it is locked by a process, & whether the file has been updated in main memory but not yet on disk.

* Structure -

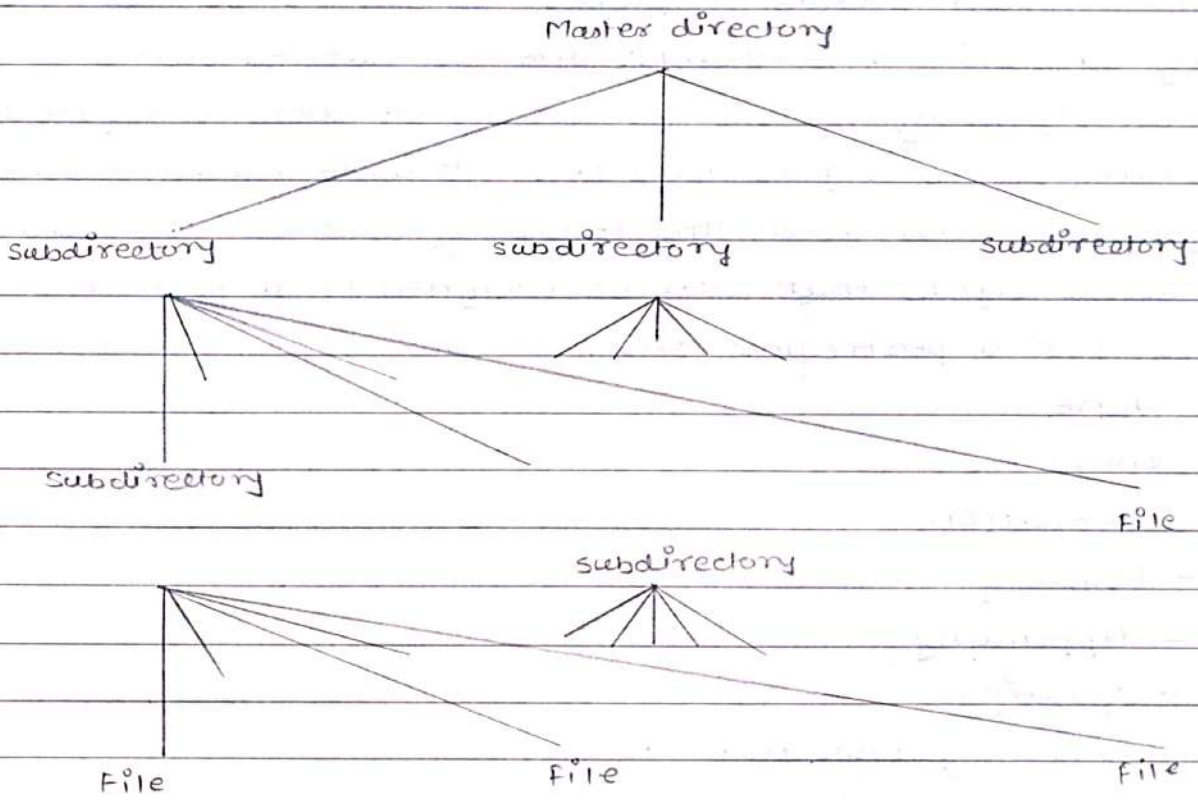


Fig - Tree - structured Directory

* Naming -

Users need to be able to refer to a file by a symbolic name. Clearly, each file in the system must have a unique name in order that file references be unambiguous. On the other hand, it is an unacceptable burden on users to require that they provide unique names, especially in a shared system.

* File Sharing -

In a multiuser system, there is almost always a requirement for allowing files to be shared among a number of users. Two issues arise, access rights and the management of simultaneous access.

Access Rights -

The file system should provide a flexible tool for

allowing extensive file sharing among users. The file system should provide a number of options so that the way in which a particular file is accessed can be controlled. Typically, users or groups of users are granted certain access rights to a file. A wide range of access rights has been used. The following list is representative of access rights that can be assigned to a particular user for a particular file:

- None
- Knowledge
- Execution
- Reading
- Appending
- Updating
- Changing Protection
- Deletion

Access can be provided to different classes of users:

* Specific user:

Individual users who are designated by user ID.

* User groups:

A set of users who are not individually defined. The system must have some way of keeping track of the membership of user groups.

* All:

All users who have access to this system. These are public files.

* Simultaneous Access —

When access is granted to append or update a file to more than one user, the operating system or file management system must enforce discipline. A brute-force approach is to allow a user to lock the entire file when it is to be updated.

* Record Blocking –

Records are the logical unit of access of a structured file, whereas blocks are the unit of I/O with secondary storage. For I/O to be performed, records must be organized as blocks.

Given the size of a block, there are three methods of blocking that can be used.

Fixed blocking –

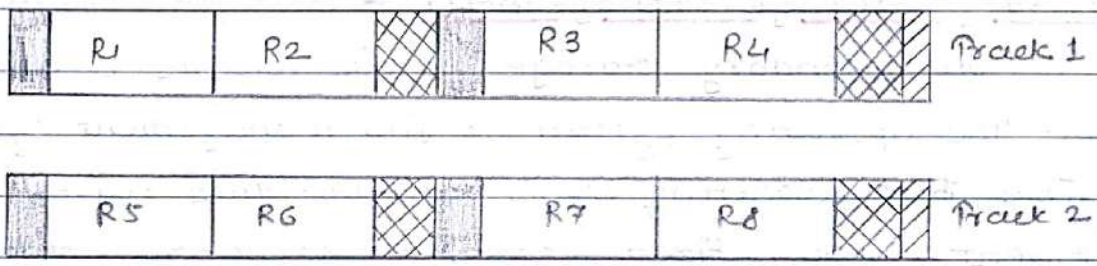
Fixed-length records are used, and an integral number of records are stored in a block. There may be unused space at the end of each block. This is referred to as internal fragmentation.

Variable-length spanned blocking –

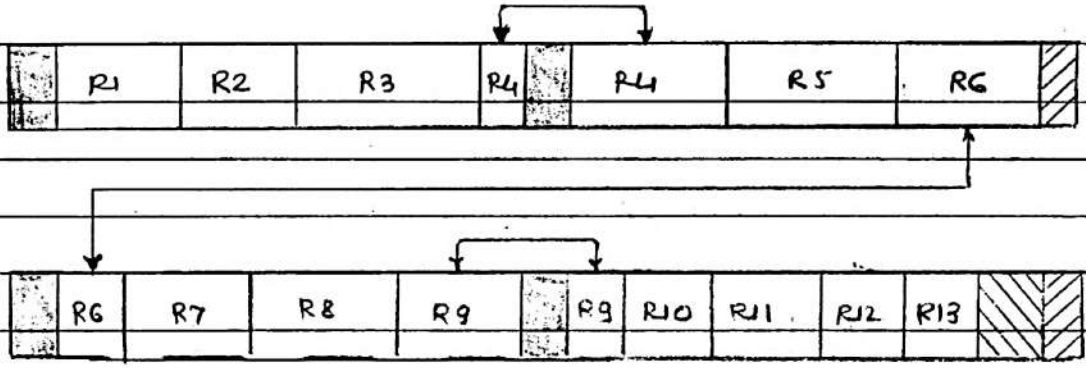
Variable-length records are used and are packed into blocks with no unused space. Thus, some records must span two blocks, with the continuation indicated by a pointer to the successor block.

Variable-length unspanned blocking –

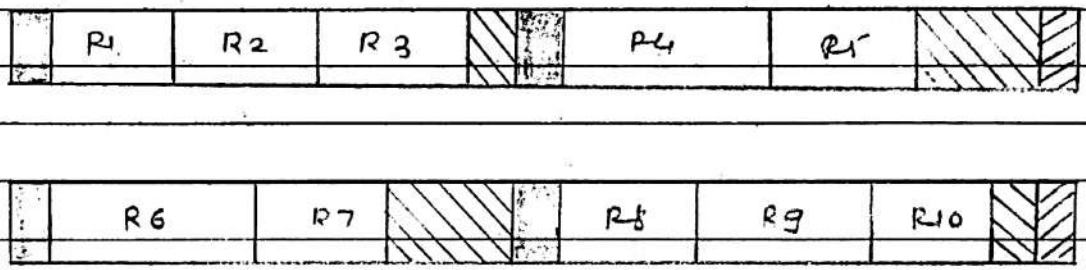
Variable-length records are used, but spanning is not employed. There is wasted space in most blocks because of the inability to use the remainder of a block if the next record is larger than the remaining unused space.



Fixed blocking



Variable blocking : spanned



Variable blocking : unspanned

Data

Waste due to record fit to blocksize

Gaps due to hardware design

Waste due to blocksize constraint from fixed record size

Waste due to block fit to track size

Fig - Record Blocking Methods [MIED67]

* Secondary Storage Management -

On secondary storage, a file consists of a collection of blocks. The operating system or file management system is responsible for allocating blocks to files. This raises two management issues. First, space on secondary storage must be allocated to files and second, it is necessary to keep track of the space available for allocation. We will see that these two tasks are related; that is, the approach taken for free space management. Further, we will see that there is an interaction between file structure

* File Allocation –

Several issues are involved in file allocation:

1] When a new file is created, is the maximum space required for the file allocated at once?

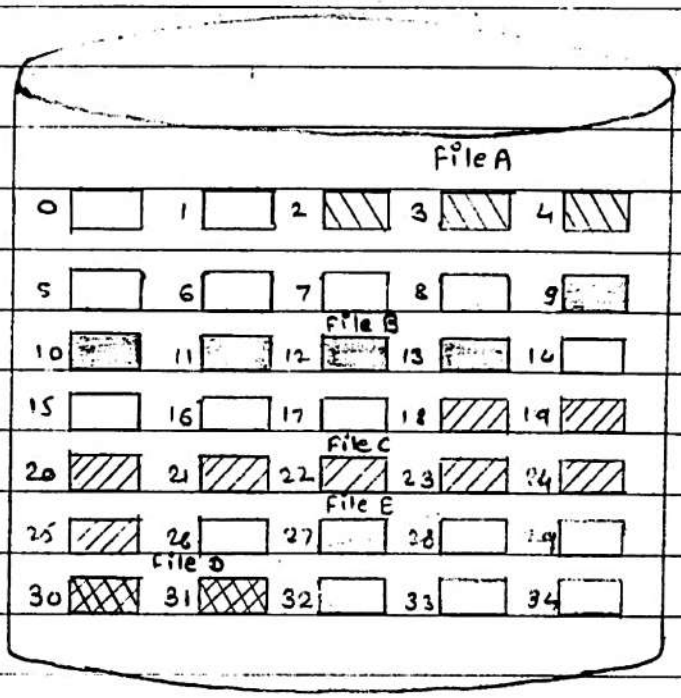
2] Space is allocated to a file as one or more contiguous units, which we shall refer to as portions. That is, a portion is a contiguous set of allocated blocks. The size of a portion can range from a single block to the entire file. What size of portion should be used for file allocation?

3] What sort of data structure or table is used to keep track of the portions asigned to a file? An example of such a structure is a file allocation table (FAT), found on DOS and some other systems.

Table – File Allocation methods –

	Contiguous	Chained	Indexed	
Preallocation?	Necessary	Possible	Possible	
Fixed or variable size portions?	Variable	Fixed blocks	Fixed blocks	Variable
Portion size	Large	Small	Small	Medium
Allocation Frequency	once	Low to high	High	Low
Time to allocate	medium	Long	Short	Medium
File allocation table size	one entry	one entry	Large	Medium

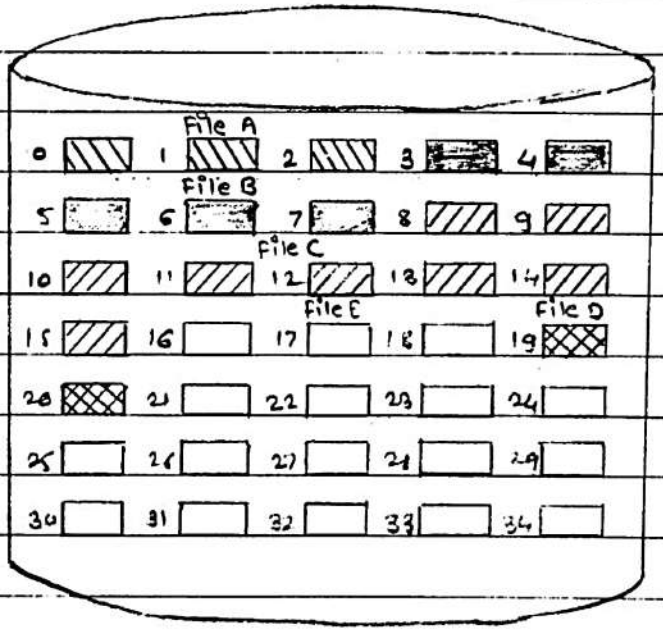
File Allocation Methods –



File Allocation Table

File name	Start Block	Length
File A	2	3
File B	9	5
File C	18	8
File D	30	2
File E	26	3

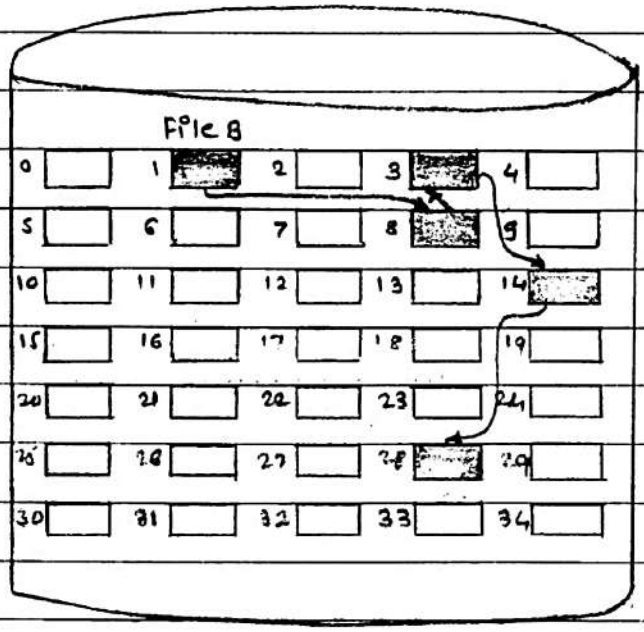
Fig (A) – Contiguous File Allocation



File Allocation Table

File name	Start Block	Length
File A	0	3
File B	3	5
File C	8	8
File D	19	2
File E	16	3

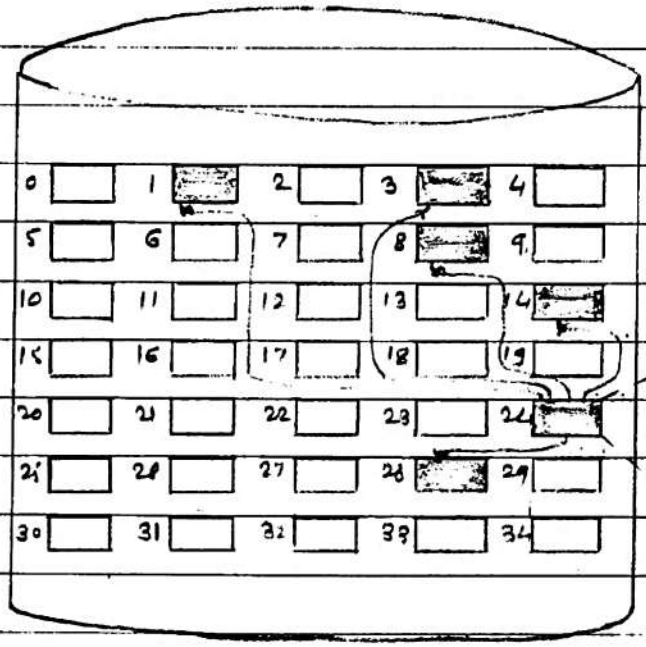
Fig – Contiguous File Allocation (After compaction)



File Allocation Table -

File name	start Block	Length
• • •	• • •	• • •
File B	1	5
• • •	• • •	• • •

Fig – Chained Allocation



File name	Index Block
• • •	• • •
File B	24
• • •	• • •

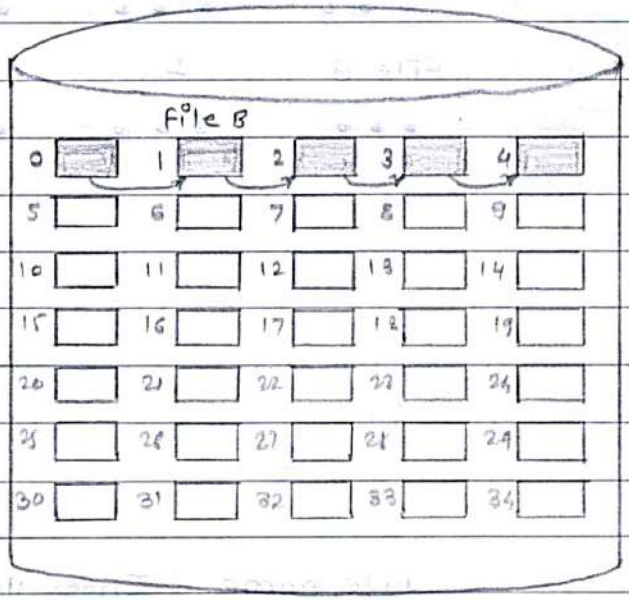
Fig - Indexed Allocation with Block Portions

* Bit Tables –

This method uses a vector containing one bit for each block on the disk. Each entry of a 0 corresponds to a free block, and each 1 corresponds to a block in use. For e.g. for the disk layout of fig. A, a vector of length 35 is needed and would have the following value:

00111000011111000011111111111011000

A bit table has the advantage that it is relatively easy to find one or a contiguous group of free blocks.



File Allocation Table

File name	Start Block	Length
. . .	. . .	. . .
File B	0	5
. . .	. . .	. . .

Fig - Chained Allocation (After Consolidation)

*** Chained Free Portions -**

The free portions may be chained together by using a pointer and length value in each free portion. This method has negligible space overhead because there is no need for a disk allocation table, merely for a pointer to the beginning of the chain & the length of the first portion.

*** Indexing -**

The indexing approach treats free space as a file and uses an index table as described under file allocation.

*** Free Block List -**

In this method, each block is assigned a number sequentially & the list of numbers of all free blocks is maintained in a reserved portion of the disk. Depending

on the size of the disk, either 24 or 32 bits will be needed to store a single block number, so the size of the free block list is 24 or 32 times the size of the corrosponding bit table and thus must be stored on disk rather than in main memory.

* Volumes —

The term volume is used somewhat differently by different operating systems and file management systems, but in essence a volume is a logical disk.

* Linux Virtual File system —

Linux includes a versatile and powerful file handling facility, designed to support a wide variety of file management systems and file structures. The approach taken in Linux is to make use of a virtual file system (VFS), which presents a single, uniform file system interface to user processes. The VFS defines a common file model that is capable of representing any conceivable file system's general feature & behaviour. The VFS assumes that files are objects in a computer's mass storage memory that share basic properties regardless of the target file system or the underlying processor hardware.

VFS is an object-oriented scheme. Because it is written in C, rather than a language that supports object programming (such as C++ or Java), VFS objects are implemented simply as C data structures. Each object contains both data and pointers to file-system-implemented functions that operate on data. The four primary object types in VFS are as follows:

1) Superblock object : Represents a specific mounted file system.

2) Inode object : Represents a specific file.

3) Dentry object : Represents a specific directory entry.

4] File object - Represents an open file associated with a process.

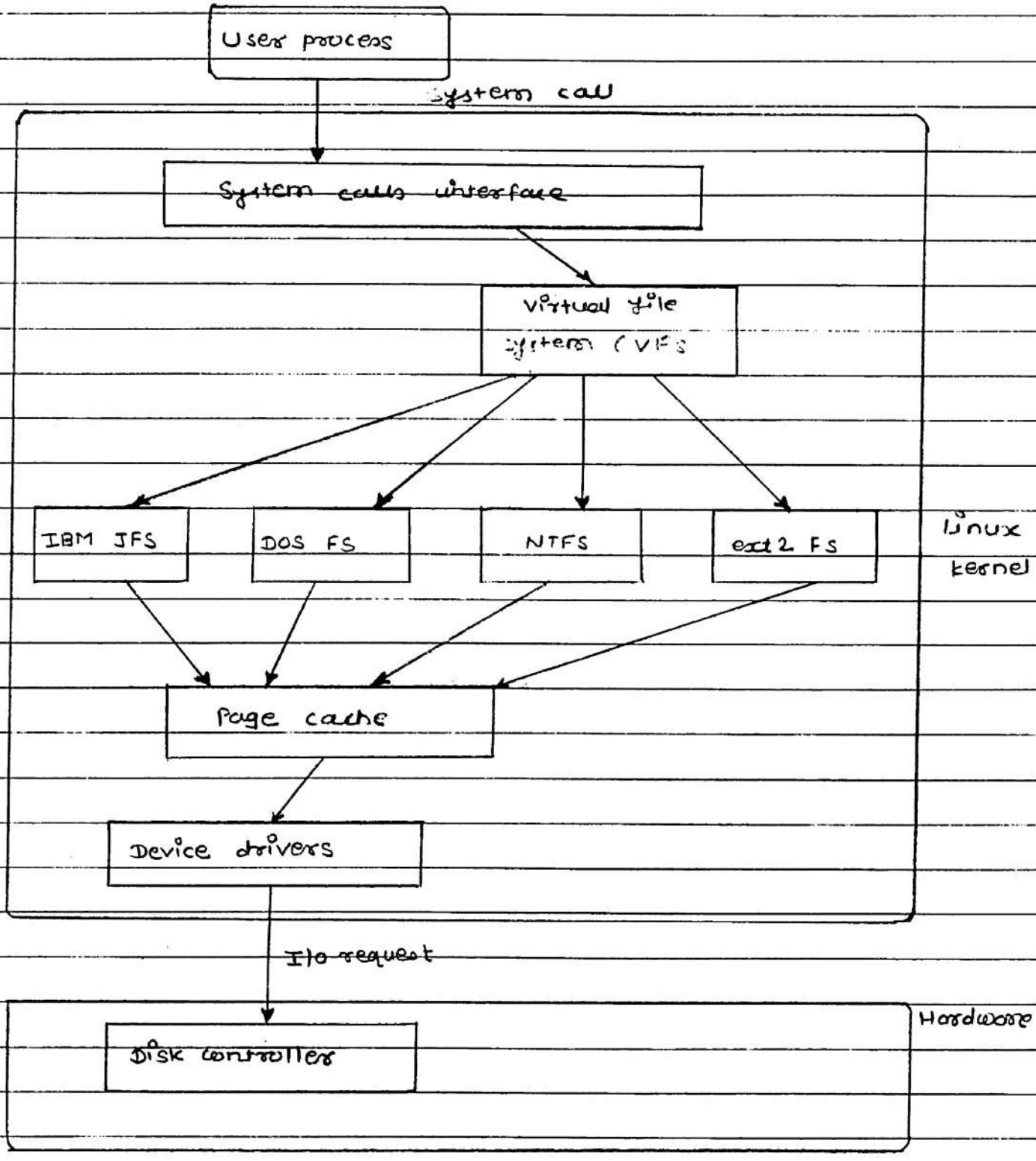


Fig - Linux Virtual File System context

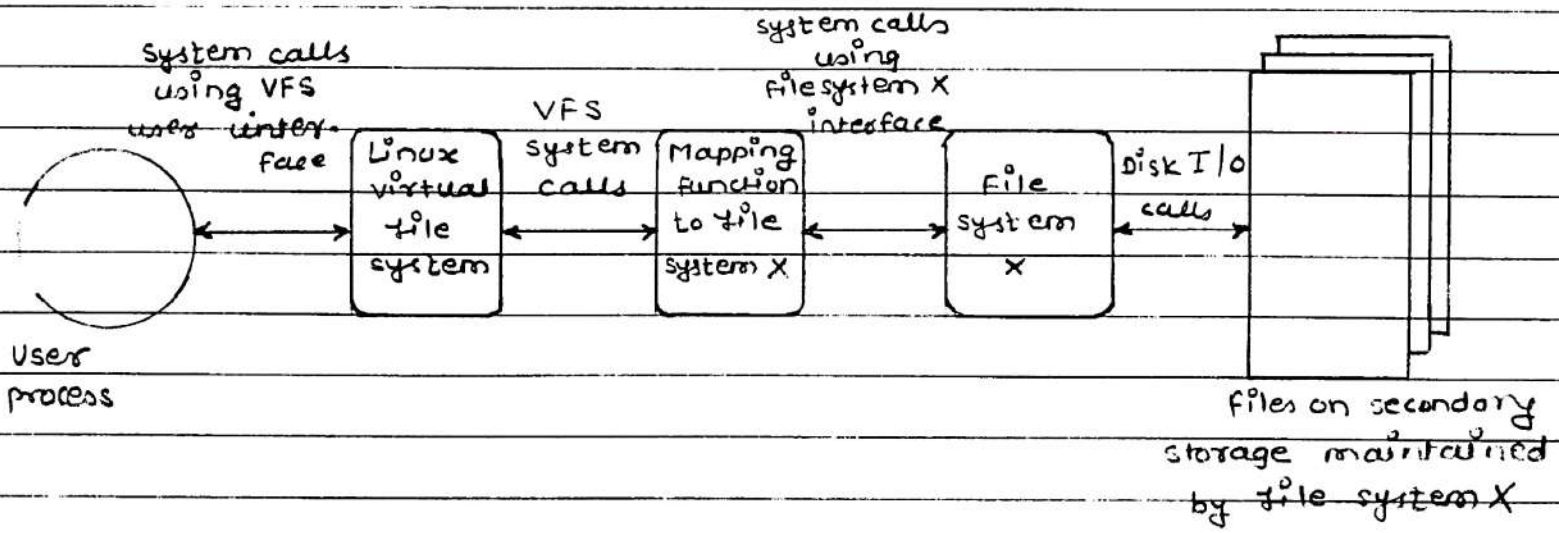


Fig – Linux Virtual file system concept

END

Unit - VI

Linux Kernel Module Programming -

What are kernel modules ?

Kernel modules are piece of code, that can be loaded & unloaded from kernel on demand.

Kernel modules offers an easy way to extend the functionality of base kernel without having to rebuild or recompile the kernel again. Most of the drivers are implemented as a Linux kernel modules. When those drivers are not needed, we can unload only that specific driver, which will reduce the kernel image size.

The kernel module will have a .ko extension. On a normal linux system, the kernel modules will reside inside /lib/modules/<kernel_version>/kernel/directory.

I] Utilities to Manipulate Kernel Modules -

1] Ismod -

List Modules that loaded already.

Ismod command will list modules that are already loaded in the kernel as shown below:

2] insmod -

Invest module into kernel

insmod command will invest a new module into the kernel as shown below

```
# insmod /lib/modules/3.5.0-19-generic/kernel/fs/squashfs/squashfs.ko
# lsmod | grep "squash"
squashfs        35884  0
```

3] modinfo -

Display module info.

modinfo command will display information about a kernel module as shown below.

modinfo /lib/modules/3.5.0-19-generic/kernel/fs/squashfs/squashfs.ko

4] rmmod -

Remove module from kernel.

rmmod command will remove a module from the kernel. You cannot remove a module which is already used by any program.

rmmod squshfs.ko

5) modprobe -

Add or remove modules from the kernel.

modprobe is an intelligent command which will load/unload modules based on the dependancy between modules. Refer to modprobe commands for more detailed examples.

II) Write a Simple Hello World Kernel Module.

1] Installing the linux headers -

You need to instal the linux headers first as shown below. Depending on your distro, use apt-get or yum.

apt-get install build-essential linux-headers-$(uname -r)

2) Hello World Module Source Code -

Next, create the following hello c module in C programming language.

```
#include <linux/module.h> //included for all kernel modules.
# include <linux/kernel.h> // included for KERN-INFO.
# include <linux/init.h> //included for -init and -exit
```

DATE :

```
MODULE_LICENSE ("GPL")
MODULE_AUTHOR ("Lakshmanan");
MODULE_DESCRIPTION ("A simple Hello World module");
static int __init hello_init (void)
{
    printk (KERN_INFO "Hello world!\n");
    return 0;
  // Non-zero return means that the module couldn't be loaded.
}
static void __exit hello_cleanup (void)
{
    printk (KERN_INFO "cleaning up module.\n");
}
module_init (hello_init);
module_exit (hello_cleanup);
```

Warning: All kernel modules will operate on kernel space, a highly privileged mode. So be careful with what you write in a kernel module.

3] Create Makefile to compile kernel Module.

The following makefile can be used to compile the above basic hello world kernel module.

```
obj-m += hello.o
all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
clean:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean.
```

Use the make command to compile hello world kernel module as shown below.

```
# make
make -C /lib/modules/3.5.0-19-generic/build M=/home/
lakshmanan/a modules
make [1]: Entering directory /usr/src/linux-headers-
3.5.0-19-generic'
  CC [M] /home/lakshmanan/a/hello.o
Building modules, stage 2.
MODPOST 1 modules
  CC     /home/lakshmanan/a/hello.mod.o
  LD [M] /home/lakshamanan/a/hello.ko
make [1]: Leaving directory /usr/src/linux-headers-
3.5.0-19-generic'
```

The above will create hello.ko file, which is our sample kernel module.

4] Insert or Remove the Sample Kernel Module –

Now that we have our hello.ko file we can insert this module to the kernel by using insmod command as shown below.

```
# insmod hello.ko
# dmesg | tail -1
[ 8394.731865] Hello world!
# rmmod hello.ko
# dmesg | tail -1
[ 8707.989319] Cleaning up module.
```

When a module is inserted into the kernel, the module_init macro will be invoked, which will call the function hello_init. Similarly, when the module is removed with rmmod, module_exit macro will be invoked, which will call the hello_exit. Using dmesg command, we can see the output from the sample kernel module.

DATE :

Please note that printk is a function which is defined in kernel, & it behaves similar to the printf in the IO library. Remember that you cannot use any of the library functions from the kernel module.

Embedded systems –

The term embedded system refers to the use of electronics & software within a product, as opposed to a general-purpose computer, such as a laptop or desktop system.

Embedded system – A combination of computer hardware & software, & perhaps additional mechanical or other parts, designed to perform a dedicated function. In many cases, embedded systems are part of a larger system or product, as in the case of an antilock braking system in a car.

* Possible Organization of an Embedded system.

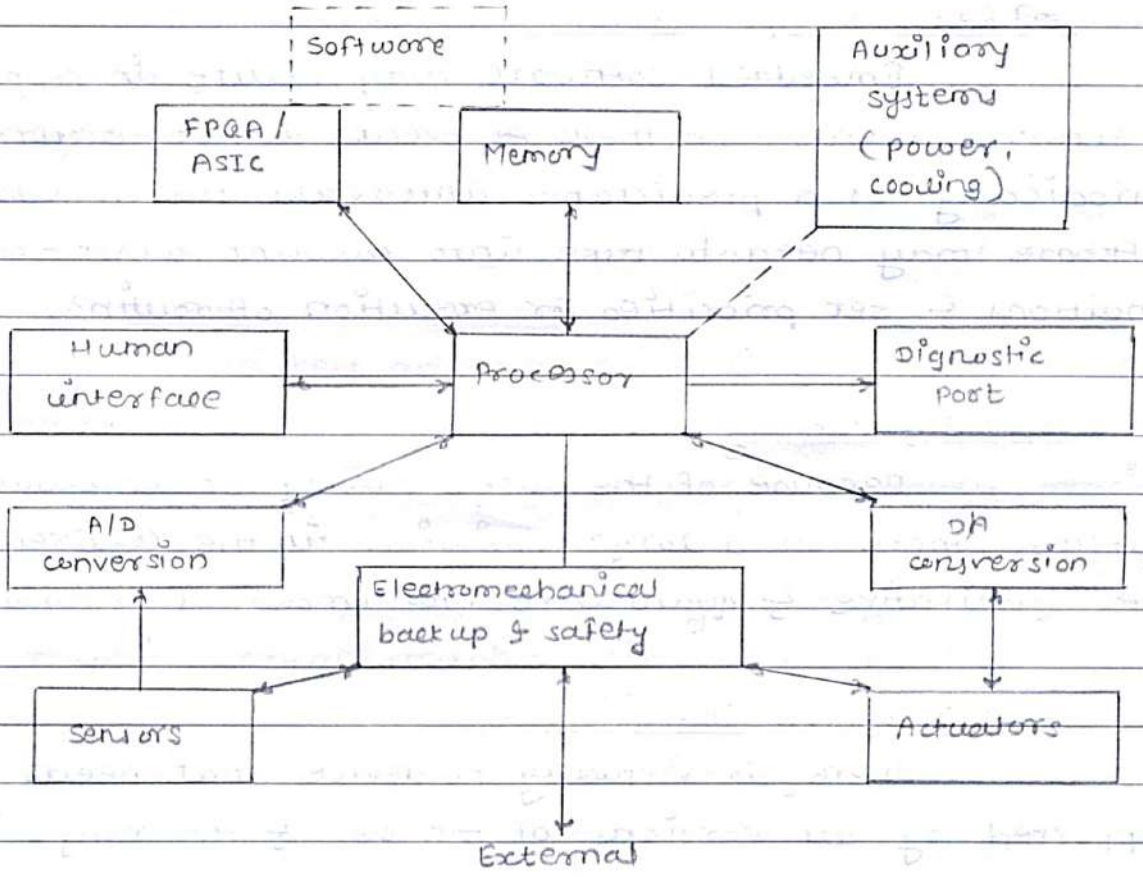


Fig- Possible organization of an Embedded system.

Characteristics of Embedded Operating Systems -

A simple embedded system, with simple functionality may be controlled by a special purpose program or set of programs with no other software. Typically, more complex embedded systems include an OS. Although it is possible in principle to use a general-purpose OS, such as Linux, for an embedded system, constraints of memory space, power consumption & real time requirements typically dictate the use of a special-purpose OS designed for the embedded system environment.

The following are some of the unique characteristics & design requirements for embedded operating systems.

1) Real-time operation -

In many embedded systems, the correctness of a computation depends, in part, on the time at which it is delivered.

2] Reactive operation -

Embedded software may execute in response to external events. If these events do not occur periodically or a predictable intervals, the embedded software may need to take into account worst-case conditions & set priorities for execution of routines.

3] Configurability -

Because of the large variety of embedded systems, there is a large variation in the requirements, both qualitative & quantative, for embedded OS functionality

4] I/O device flexibility -

There is virtually no device that needs to be supported by all versions of the OS, & the range of I/O devices is large.

5] Streamlined protection mechanism -

Embedded systems are typically designed for a limited, well-defined functionality. Untested programs are rarely added to the software. After the software has been configured & tested, it can be assumed to be reliable.

6] Direct use of interrupts -

General-purpose operating systems typically do not permit any user process to use interrupts directly. [MARW06] lists three reasons why it is possible to let interrupts start or stop tasks (e.g. by storing the task's start address in the interrupt vector address table) rather than going through OS interrupt service routines.

Embedded Linux —

Embedded Linux is the usage of the Linux kernel & various open-source components in embeded systems.

Advantages of using Linux in embedded systems.

1) Re-using components -

- The key advantage of Linux & open-source in embedded systems is the ability to re-use components.
- The open source ecosystem already provides many components for standard features, from hardware support to network protocols, going through multimedia, graphic, cryptographic libraries, etc.
- As soon as a hardware device, or a protocol, or a feature is wide-spread enough, high chance of having open-source components that support it.

2] Low Cost -

- Free software can be duplicated on as many devices as you want, free of charge.
- If your embedded system uses only free software you can reduce the cost of software license to zero. Even the development tools are free, unless you choose a commercial embedded Linux edition.

3] Full control -

- With open source, you have the source code for all components in your system.
- Allows unlimited modifications, changes, tuning debugging, optimization, for an unlimited period of time.
- Allows to have full control over the software part of your system.

4] Quality -

- Many open-source components are widely used, on millions of systems.
- Usually higher quality than what an in-house development can produce, or even proprietary vendors.
- Allows to design your system with high-quality components at the foundations.

5] Eases testing of new features -

- Open-source being freely available, it is easy to get a piece of software & evaluate it.
- Allows to easily study several options while making a choice.

6] Community support -

- Open-source software components are developed by communities of developers & users.
- This community can provide a high-quality

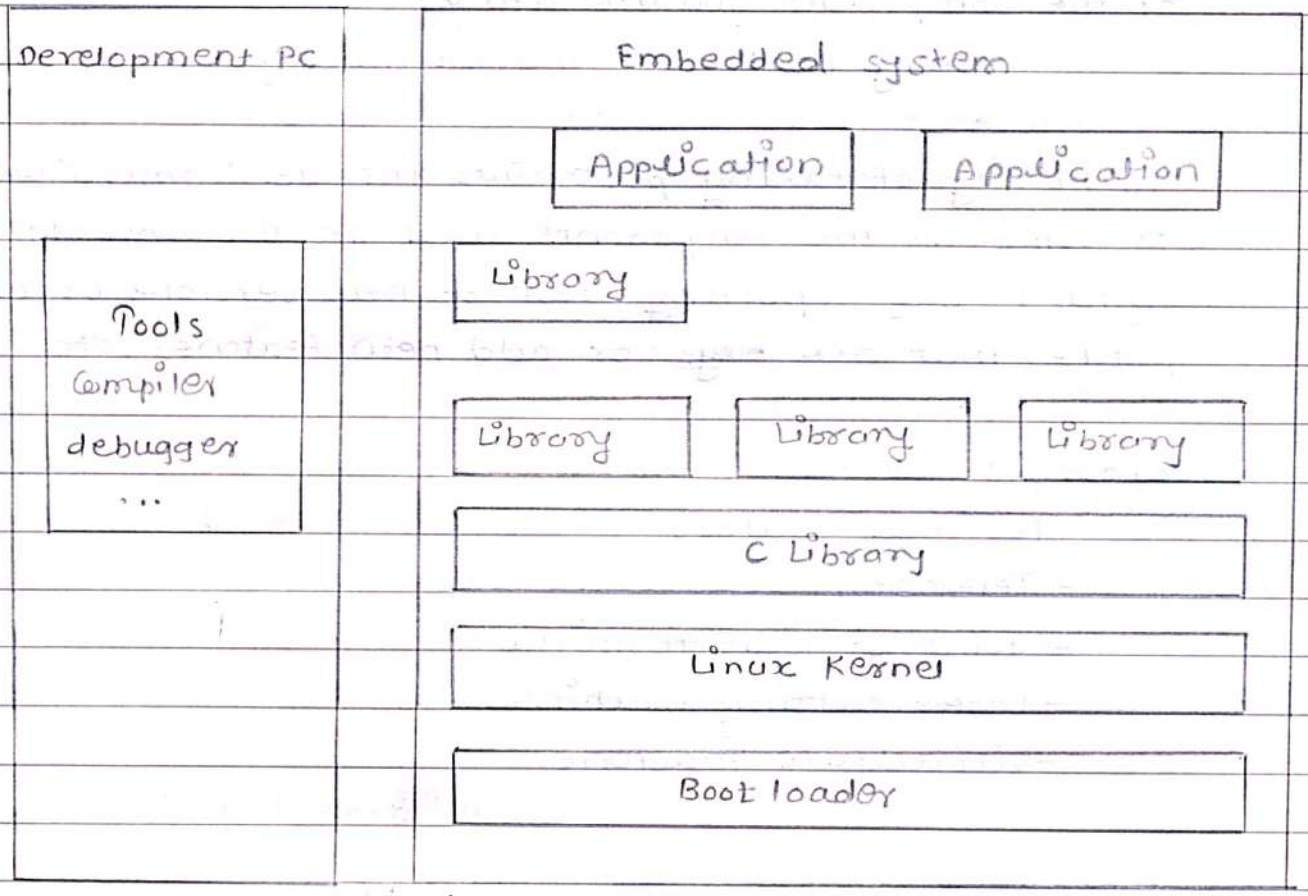
support : you can directly contact the main developers of the component you are using.
- Taking part into the community.

Possibility of taking part into the development community of some of the components used in the embedded systems : bug reporting, test of new versions or features, patches that fix bugs or add new features, etc.

Examples of Embedded Linux –
- Personal routers.
- Television.
- Point of sale terminal
- Laser cutting machine
- Viticulture machine.

Embedded hardware for Linux system –
- The Linux kernel & most other architecture-dependent component support a wide range of 32 & 64 bits architectures.
- x86 and x86-64, as found on PC platforms, but also embedded systems (multimedia, industrial).
- ARM, with hundreds of different SoC (multimedia, industrial)
- PowerPC (mainly real-time, industrial applications)
- MIPS (mainly networking applications)
- Super H (mainly set top box & multimedia applications.
- Blackfin (DSP architecture)
- Microblaze (soft-core for xilinx FPGA)
- Coldfire, SCore, Pile, Xtensa, Cris, FRV, AVR32, M32R.

Embedded Linux system architecture -



Software components -

Cross - compilation toolchain -

Compiler that runs on the development machine, but generates code for the target.

Bootloader -

Started by the hardware, responsible for the basic initialization, loading & executing the kernel.

Linux kernel -

Contains the process & memory management, network stack, device drivers & provides services to user space applications.

C library -

The interface between the kernel & the user space applications.

Libraries & applications - Third party or in-house.

Embedded Linux Work –

several distinct tasks are needed when deploying embedded linux in a product:

Board support Package development

- A BSP contains a bootloader & kernel with the suitable device drivers for the targeted hardware.
- Purpose of our kernel development training.

System integration –

- Integrate all the components, bootloader, kernel, third party libraries & applications & inhouse applications into a working system.
- Purpose of this training.

Development of applications –

- Normal Linux applications, but using specifically chosen libraries.

Application specific operating systems –

Application specific customization became all the more important with the advent of application domains such as multimedia, database, parallel computing etc. which demand high performance & high functionality. Resource management & communication / sharing between applications is to be done by the operating system code running as library routines in each application. Since, these library routines can be made application-specific, the programmer has the flexibility to easily modify them whenever that is necessary for performance.

Need for customized operating systems –

Greater performance –

One of the primary motivation for such a system is greater performance. Problems with traditional operating systems is that they hide information behind high-level abstractions like processes, page table structures, IPC etc.

And hence they provide a virtual machine to the user. Thus they give a fixed implementation to all application thereby making domain specific optimizations impossible General purpose implementations do reduce performance to a great extent & that application specific virtual memory policies can increase application performance.

More Functionality & flexibility -

Further more towards customization is motivated by a need for more functionality. New application domains like multimedia need different sets of services. Incorporating all the services is not a good idea because one may not use number of those service but still have to pay for them in the form of OS overhead

Sophisticated security -

Modern application technologies such as Java require sophisticated security in the form of access controls, that default operating systems may not provide.

Simplicity -

Simple rather than a complex kernel is easy to implement & is efficient. One may draw an analogy to RISC systems for this aspect.

Issues & problems -

- Protection & sharing
- Too much of flexibility.
- Application programmers aren't operating system designers.
- Backward compatibility & Portability.

Extensible Application specific operating systems -

SPIN -

SPIN investigates kernels that allow applications to make policy decisions. All management policies are defined by embedded implementations called spindles. These spindles are dynamically loaded into the kernel.

Merits -

- Secure because of the use of type safe language & compiler checks.
- Good amount of extensibility, relatively with less burden on user.

Demerits -

- High overhead due to cross domain calls.
- Fair sharing across applications is not possible.
- Not reflective i.e. it cannot monitor & modify its own behaviours as it executes.

Aegis - an Exokernel -

Aegis uses the familiar "end-to-end" argument that applies to low-level communications protocols. Maximum opportunity for application-level resource management is assured by designing a thin exokernel. The job of exokernel is to a. track ownership of resources b. perform access control.

Merits -

- Truly extensible & flexible as it exposes the hardware.
- Very efficient.

Demerits -

- Too flexible & hence may lead to portability &

compatibility problems.
- Less secure - it expects a high degree of trust.
- Highly complex software support is needed.

SPACE -

SPACE uses processors, address spaces & a generalized exception mechanism as the basic abstractions. The processor has more than one privilege mode & thus arbitrary number of domains can exist in an address space.

Merits -
- Virtualized resources as the fundamental abstraction & hence more customizable.

Demerits -
- Building applications in terms of spaces, domains, & portals is highly complex & lacks portability.

Synthetix -

This work is a follow on from Synthesis project. It makes use of feedback from the system & use it to dynamically modify the code. The make use of the information from dynamically changing conditions. They use this technique not only for designing OS but also for seamless mobility & few others.

Merits -
- Absolutely no burden on user.
- There are no problems of portability, security etc. in fact it is implemented on HP-UX.

Demerits -
- No Full support for customization.

Kea -

In Kea kernel can be reconfigured through RPC mechanism. RPC in Kea is very general unlike others & eliminate unnecessary steps often made in a general RPC system (like in LRPC). Kea has domains, threads, inter-domain calls & portals as abstractions.

Merits -

- Provides a neat interface through user; user can use usual semantics to configure kernel.

Demerits -

- Huge penalty is incurred because of number of cross domain calls.

Vino -

VINO is a downloadable system, that uses sand-boxing [] to enforce trust & security. Sandboxing is a technique developed to allow embedding of untrusted application code inside trusted programs. Extensions are written in C++ & are compiles by a trusted compiler. This trusted code is then downloaded into the kernel. Each extension in VINO is run in the context of a transaction.

Merits -

- Performance is only slightly worse than unprotected C++.
- Extensions can be implemented in conventional languages.

Demerits -

- Security of the system is heavily dependent on compiler. There is no guard if a malicious user tricks the compiler.

Spring -

Spring is also another micro-kernel based operating system developed to build operating system out of replaceable parts.

Table - Comparison of different application specific operating systems.

Name	Overhead due to cross domain calls	Security	Complexity	Degree of customization & extensibility.	Optimization across applications
SPIN	High	Medium	Medium	High	Low
Aegis	Low	Low	High	High	High
Space	High	Medium	Medium	Medium	Low
Synthetix	Low	High	Low	Low	High
Kea	High	High	Low	Medium	High
Vino	Low	Medium	Low	Medium	High
Spring	High	Medium	Low	Medium	Low
Cache Kernel	Low	High	High	Medium	High
VM	Low	High	High	High	High

Nach OS -

Nach OS is a new teaching operating system & simulation environment. It makes it possible to give assignments that require students to write significant portions of each of the major pieces of a modern operating system: thread management, file systems, multiprogramming, virtual memory & networking. The concepts are necessary to understand the computer systems of today & of the future: concurrency & synchronization, caching & locality,

the trade-off between simplicity & performance, building reliability from unreliable components, dynamic scheduling, the power of a level of translation, distributed computing, layering & virtualization.

Basic services of Nachos OS -

1] Thread management -

Basic working thread system & an implementation of semaphores. The assignment is to implement Mesa-style locks & condition variables using semaphores & then to implement solutions to a number of concurrency problems using these synchronization primitives.

2] File systems -

Real file systems can be very complex artifacts. The UNIX file system, for example, has atleast three levels of indirection - the per-process file descriptor table, the system-wide open file table, & the in-core inode table - before one even gets to disk blocks.

3] Multiprogramming -

The code to create a user address space, load a Nachos file containing an executable image into user memory, and then to run the program is provided. Initial code is restricted to running only a single user program at a time.

4] Virtual memory -

It asks students to replace their simple memory management code from the previous assignment with a true virtual memory system, that is, one that presents to each user program the

abstraction of an (almost) unlimited virtual memory size by using main memory as a cache for the disk. No new hardware or operating system components are provided.

5] Networking –

At the hardware level, we simulate the behaviour a network of workstations, each running Nachos, by connecting the UNIX processes running Nachos via sockets. The Nachos operating system & user programs running on it can communicate with other "machines" running Nachos simply by sending messages into the emulated network the transmission is actually accomplished by socket send & receive.

Introduction to service oriented operating system –

The S(o) OS project addressed future distributed systems on the level of a holistic operating system architecture by drawing from service Oriented Architectures & the strength of grids. This decouples the OS from the underlying resource infrastructure, thus making execution across an almost unlimited number of varying devices possible, independant from the actual hardware.

S(o) OS investigated alternative Operating system Architectures that base on a modular principle, where not only the individual modules may be replaced according to the respective resource specifics, but, more importantly, where also the modules may be distributed according to the executing process requirements. The operating system thus becomes a dynamic service oriented infrastructure in which each executing process may specify the required services which are detected, deployed & adapted on the fly.

S(c)OS examined how code can be segmented in a fashion that allows for concurrent execution over a heterogeneous infrastructure by combining the two approaches above. This principle implicitly supports parallelisation of code in a fashion that reduces the communication overhead. With the information about the code behaviour and dependencies, further optimisation steps can be taken related to the infrastructure's communication layout, cache & memory limitations, resource specific capabilities & restrictions etc. Depending on the size & complexity of the segments, on-the-fly adaption steps may furthermore be taken to increase heterogeneity & in particular to improve performance by exploiting the resource features.

Introduction to Ubuntu Edge OS (Ubuntu Touch) -

Ubuntu Touch (also known as Ubuntu Phone) is a mobile version of the Ubuntu operating system developed by Canonical UK Ltd & Ubuntu Community. It is designed primarily for touchscreen mobile devices such as smartphones & tablet computers.

Canonical released Ubuntu Touch 1.0, the first developer / partner version on 17 october 2013, along with Ubuntu 13.10 that "primarily supports the Galaxy Nexus & Nexus 4 phones, touch user interface & various software frameworks originally developed for Maemo & MeeGo such as ofono as telephony stack, accounts-sso for single sign-on, and Maliit for input. Utilizing libhybris the system can often be used with Linux Kernels used in Android, which makes it easily ported to most recent Android smartphones.

Ubuntu Touch utilizes the same core technologies as the Ubuntu Desktop, so applications designed for the latter platform run on the former & vice versa. Additionally,

Ubuntu Desktop components come with the Ubuntu Touch system; allowing Ubuntu Touch devices to provide a full desktop experience when connected to an external monitor. Ubuntu Touch devices can be equipped with a full Ubuntu session & may change into a full desktop operating system when plugged into a docking station. If plugged the device can use all the features of Ubuntu & user can perform office work or even play ARM-ready games on such device.

Architecture :– GStreamer Playback Pipeline