UNIT 3

PROCESS MANAGEMENT

MEMORY MANAGEMENT

Swapping, Contiguous Memory Allocation, Paging, structure of the Page Table, Segmentation.

VIRTUAL MEMORY MANAGEMENT

Virtual Memory, Demand Paging, Page Replacement Algorithms, Thrashing

DEADLOCK: Principles of deadlock – System Model, Deadlock Characterization, Deadlock Prevention, Detection and Avoidance, Recovery form Deadlock

MEMORY MANAGEMENT

Memory management is the functionality of an operating system which handles or manages primary memory and moves processes back and forth between main memory and disk during execution.

Memory management keeps track of each and every memory location, regardless of either it is allocated to some process or it is free. It checks how much memory is to be allocated to processes. It decides which process will get memory at what time.

It tracks whenever some memory gets freed or unallocated and correspondingly it updates the status.If we define memory as a place where data is stored there are many levels of memory:

- Processor registers
- Primary (or main) memory
- Secondary memory
- Tertiary memory
- Cache memory

SWAPPING

- A process must be in memory to be executed. A process, however, can be swapped temporarily out of memory to a backing store and then brought back into memory for continued execution.
- For example, a round-robin CPU-scheduling algorithm, when a quantum expires the memory manager will start to swap out the process that just finished and to swap another process into the memory space that has been freed (Fig 4.1). In the meantime, the CPU scheduler will allocate a time slice to some other process in memory.
- Ideally, the memory manager can swap processes fast enough that some processes will be in memory, ready to execute, when the CPU scheduler wants to reschedule the CPU. In addition, the quantum must be large enough to allow reasonable amounts of computing to be done between swaps.
- In priority-based scheduling algorithms if a higher-priority process arrives and wants service the memory manager can swap out the lower-priority process and then load and execute the higher-priority process. When the higher-priority process finishes, the lower-priority process can be swapped back in and continued. This variant of swapping is sometimes called **rollout**, **rollin**.

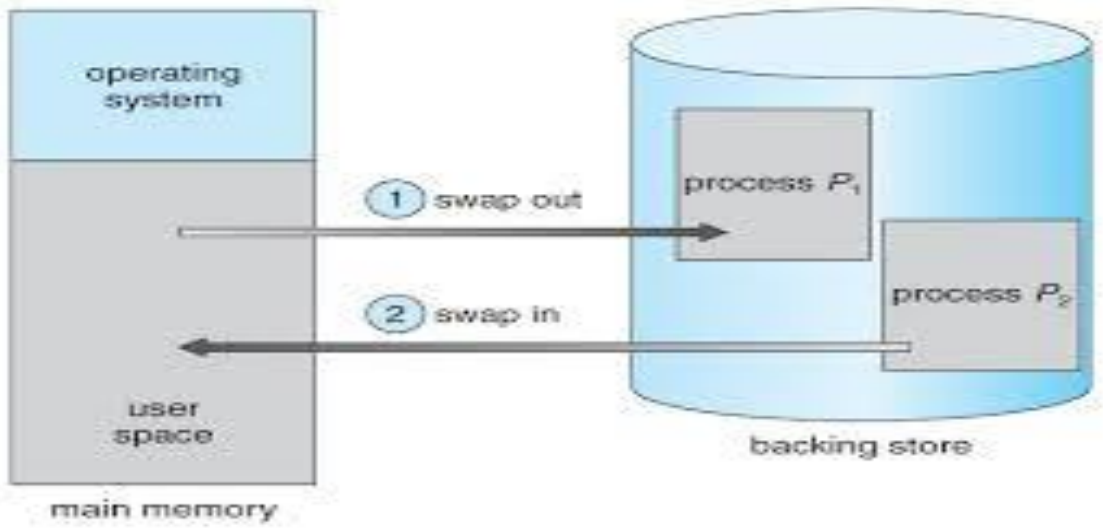


Fig 4.1: Swapping of two processes using a disk as a baking store.

Normally, a process that is swapped out will be swapped back into the same memory space it occupied previously. This restriction is dictated by the method of address binding. If binding is done at assembly or load time, then the process cannot be easily moved to a different location. If execution time binding is being used, however, then a process can be swapped into a different memory space, because the physical addresses are computed during execution time.

Swapping requires a backing store and is a fast disk. It must be large enough to accommodate copies of all memory images for all users, and it must provide direct access to these memory images.

Whenever the CPU scheduler decides to execute a process, it calls the dispatcher. The dispatcher checks to see whether the next process in the queue is in memory. If it is not, and if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process. It then reloads registers and transfers control to the selected process.

The context-switch time in such a swapping system is fairly high. For example, let us assume that the user process is 10 MB in size and the backing store is a standard hard disk with a transfer rate of 40 MB per second. The actual transfer of the 10-MB process to or from main memory takes 10000 KB/40000 KB per second = 1/4 second = 250 milliseconds.

Assuming that no head seeks is necessary, and assuming an average latency of 8 milliseconds, the swap time is 258 milliseconds. Since we must both swap out and swap in, the total swap time is about 516 milliseconds.

The total transfer time is directly proportional to the amount of memory swapped. Swapping is constrained by other factors like if we want to swap a process, it should be completely idle. A process may be waiting for an I/O operation (I/O pending) when we want to swap that process to free up memory. However, if the I/O is asynchronously accessing the user memory

for I/O buffers, then the process cannot be swapped. Transfers between operating-system buffers and process memory then occur only when the process is swapped in. Generally, swap space is allocated as a chunk of disk, separate from the file system, so that its use is as fast as possible.

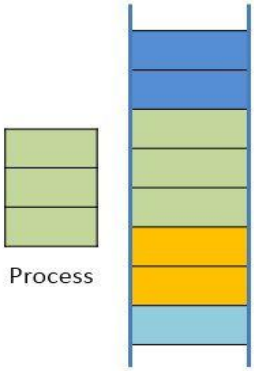
CONTIGUOUS (adjacent) MEMORY ALLOCATION

- Memory is a large array of bytes, where each byte has its own address.
- The memory allocation can be classified into two methods contiguous memory allocation and non-contiguous memory allocation.
- The major difference between Contiguous and Non-contiguous memory allocation is that the **contiguous memory allocation** assigns the consecutive blocks of memory to a process requesting for memory whereas, the **non-contiguous memory allocation** assigns the separate memory blocks at the different location in memory space in a non-consecutive manner to a process requesting for memory.

Definition of Contiguous Memory Allocation

The operating system and the user's processes both must be accommodated in the main memory. Hence the main memory is **divided into two** partitions: at one partition the operating system resides and at other the user processes reside. In usual conditions, the several user processes must reside in the memory at the same time, and therefore, it is important to consider the allocation of memory to the processes.

The Contiguous memory allocation is one of the methods of memory allocation. In contiguous memory allocation, when a process requests for the memory, a **single contiguous section of memory blocks** is assigned to the process according to its requirement.



Contiguous Memory Allocation

The contiguous memory allocation can be achieved by dividing the memory into the fixed-sized **partition** and allocate each partition to a single process only. But this will cause the degree of multiprogramming, bounding to the number of fixed partition done in the memory. The contiguous memory allocation also leads to the **internal fragmentation**. Like, if a fixed sized memory block allocated to a process is slightly larger than its requirement then the left over memory space in the block is called internal fragmentation. When the process residing in the partition terminates the partition becomes available for another process.

In the variable partitioning scheme, the operating system maintains a **table** which indicates, which partition of the memory is free and which occupied by the processes. The contiguous memory allocation fastens the execution of a process by reducing the overheads of address translation.

Memory Mapping and Protection

The issue of memory mapping and protection can be given by using a relocation register, contains the value of the smallest physical address; the limit register contains the range of logical addresses (for example, relocation = 100040 and limit = 74600). With relocation and limit registers, each logical address must be less than the limit register; the MMU maps the logical address dynamically by adding the value in the relocation register. This mapped address is sent to memory (Fig 4.2).

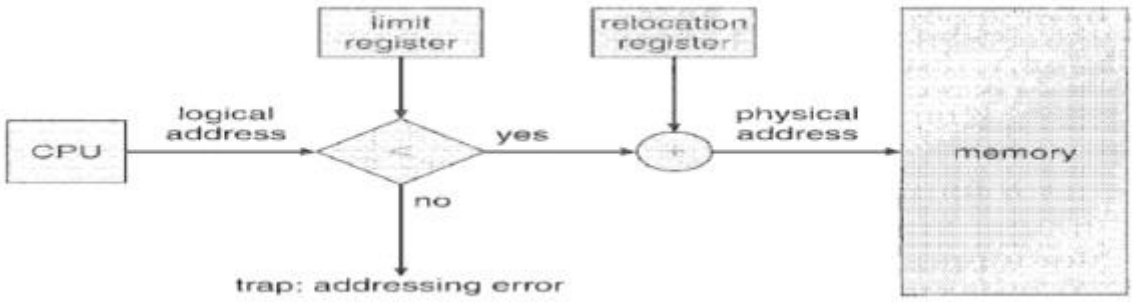


Fig 4.2: Hardware support for relocation and limit registers.

When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Because every address generated by the CPU is checked against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process.

The relocation-register scheme provides an effective way to allow the operating-system size to change dynamically. This flexibility is desirable in many situations.

Memory Allocation

One of the simplest methods for allocating memory is to divide memory into several fixed-sized partitions. Each partition may contain exactly one process. The degree of multiprogramming is given by the number of partitions.

In this multiple partition method, when a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.

In the fixed-partition scheme, the operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes and is considered one large block of available memory, a hole. When a process arrives and needs memory, we search for a hole large enough for this process and we allocate only as much memory as is needed, keeping the rest available to satisfy future requests.

At any given time, we have a list of available block sizes and the input queue. The operating system can order the input queue according to a scheduling algorithm. Memory is allocated to processes until, finally, the memory requirements of the next process cannot be satisfied that is, no available block of memory (or hole) is large enough to hold that process. The operating system can then wait until a large enough block is available, or it can skip down the input queue to see whether the smaller memory requirements of some other process can be met.
In general, at any given time we have a set of holes of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes.

When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole. This procedure is a particular instance of the general dynamic storage allocation problem, which concerns how to satisfy a request of size n from a list of free holes.

There are many solutions to this problem. The first-fit, best-fit, and worst-fit strategies are the ones most commonly used to select a free hole from the set of available holes.
• **First fit**. Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.
• **Best fit**. Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.
• **Worst fit**. Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

Simulations have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization. Neither first fit nor best fit is clearly better than the other in terms of storage utilization, but first fit is generally faster.

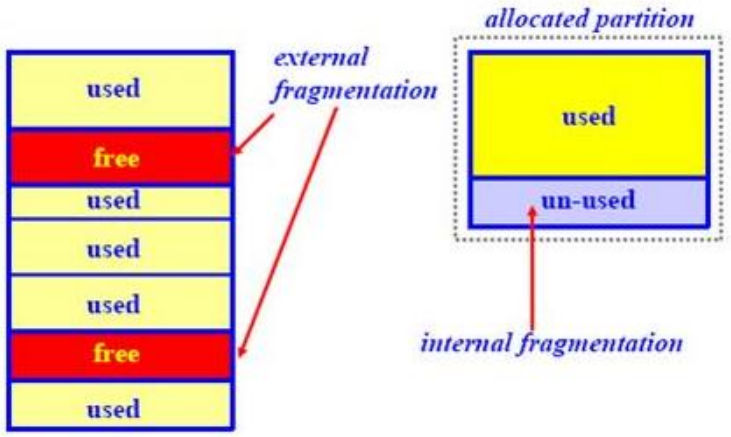
Fragmentation

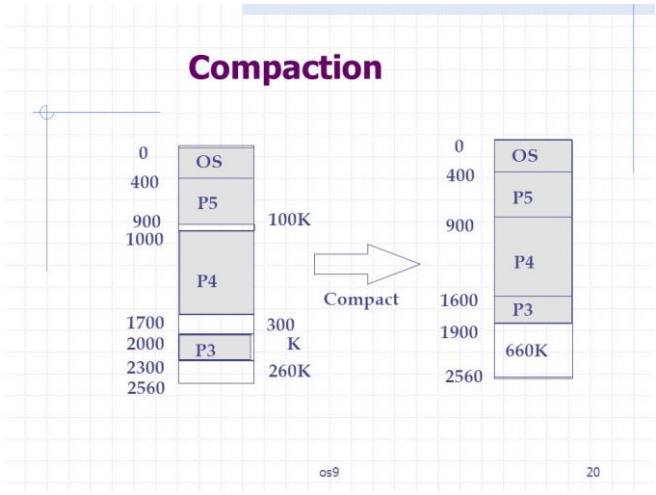
Memory fragmentation can be of two type's internal and external fragmentation.
In internal fragmentation, there is a wasted space internal to each partition due to the fact that the block of data loaded is smaller than the partition. Fig shows the internal fragmentation. Memory that is internal to a partition, but that is not being used. For example, in multiple partition scheme, hole of10524 bytes, suppose that the next process requests 10520 bytes.

If we allocate exactly the requested block, then there is left with a hole of 4 bytes. The overhead to keep track of this hole will be substantially larger than the whole itself.

External fragmentation exists when enough total memory space exists to satisfy a request, but it is not contiguous, storage is fragmented into a large number of small holes. The above figure shows the external fragmentation. There is a hole of 200K and 500K in multiple partition allocation schemes. Next process request is for 700K memory. Actually 700K of memory is free which satisfy the request but hole is not contiguous. So there is an external fragmentation of memory. The selection of first fit versus best fit can affect the amount of fragmentation. Depending on the total amount of memory storage and the average process size, external fragmentation may be either major or minor problem.

One solution to the problem of external fragmentation is compaction. The goal is to shuffle the memory contents so as to place all free memory together in one large block.
Compaction is not always possible, however. If relocation is static and is done at assembly or load time, compaction cannot be done; compaction is possible only if relocation is dynamic and is done at execution time. If addresses are relocated dynamically, relocation requires only moving the program and data and then changing the base register to reflect the new base address. When compaction is possible, we must determine its cost. The simplest compaction algorithm is to move all processes toward one end of memory; all holes move in the other direction, producing one large hole of available memory which can be expensive. Another possible solution to the external-fragmentation problem is to permit the logical address space of the processes to be noncontiguous, thus allowing a process to be allocated physical memory wherever the latter is available.





PAGING

- Paging is a memory-management scheme that permits the physical address space of a process to be noncontiguous.
- Paging reduces the external fragmentation.
- The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called frames and breaking logical memory into blocks of the same size called pages.

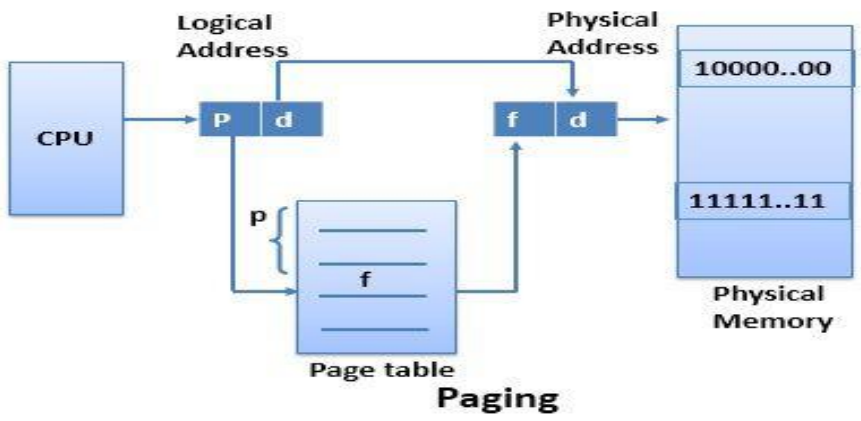


Fig 4.3: Paging hardware.

When a process is loaded into memory, the operating system loads each page into a unused page frame. The page frames used need not be contiguous and the operating system maintains the page table for each process. The page table shows the frame location for each page of the process.

Every address generated by the CPU is divided into two parts: a page number (p) and a page offset (d). In case of a simple partition, a logical address is the location of a word relative to the beginning of the program; the processor translates that into a physical address.

The page number is used as an index into a page table. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit. The paging model of memory is shown in Fig 4.4.

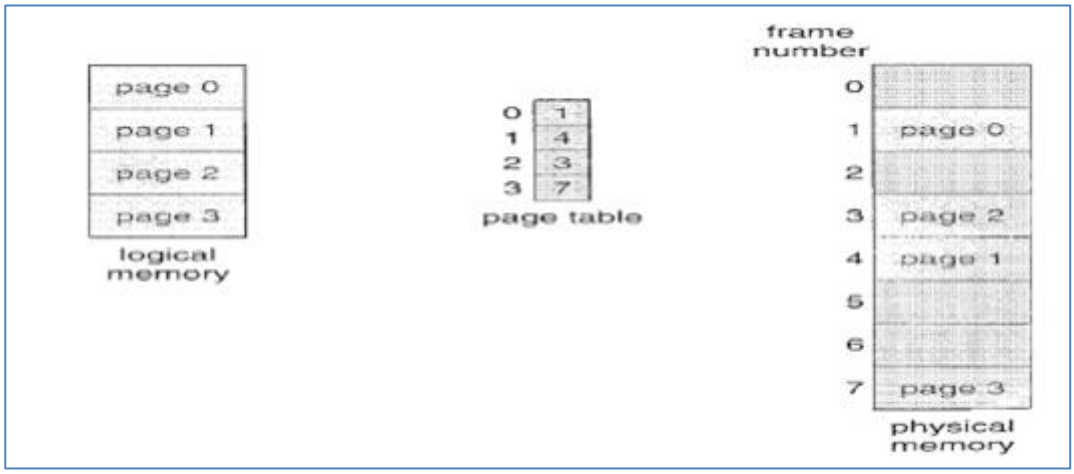


Fig 4.4: Paging model of logical and physical memory.

The logical address consists of page number and page offset:

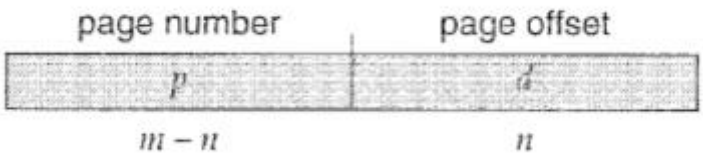


Fig 4.5: Logical address.

Where p is an index into the page table and d is the displacement within the page.
The logic of the address translation process in paged system is shown in the following Fig. 4.6. For example, the virtual address is 03200H. This virtual address is split by hardware into the page number and offset within that page.

High order 12 bit is used as page number i.e. 003H and lower order 12 bit is used for the offset i.e. (200H). Page number is used to index the page table and to obtain the corresponding physical frame

number (FFFH). This value is then concatenated with the offset to produce the physical address (FFF200H) which is used to reference the target item in memory.

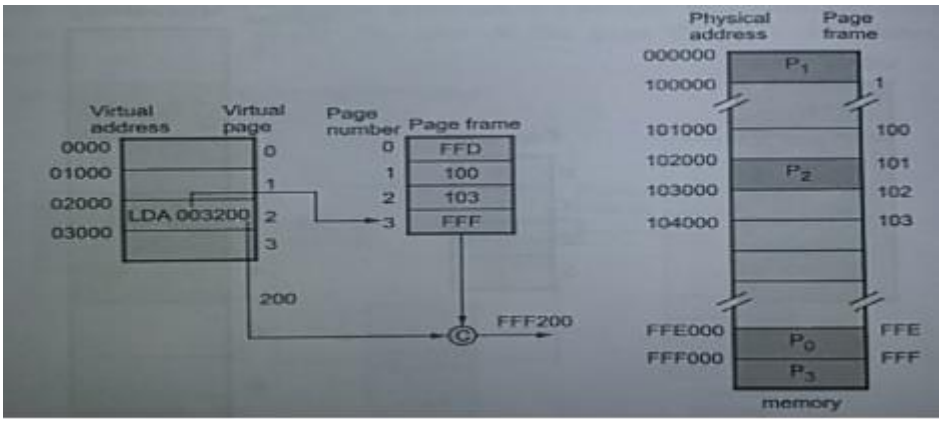


Fig. 4.6: (Virtual address to physical address) Paging

The operating system keeps track of the status of each page frame by means a physical memory map that may be structured as a static table. The logical address space may be the same size as the physical address space or it may be smaller.

If logical address space is small, it prevents one process from monopolizing all of the memory. If the logical address space is larger than the physical address space all pages could not be resident in physical memory. Simple paging has no capability for dealing with references to pages that are not in memory.

Protection and Sharing

Protection bits are used with each page frame for protecting memory in pages.

For eg: protection bits (i.e. access bit) may allow read-only, execute-only, or other restricted forms of access. Every reference physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read-only page. Specification of access rights in paging systems is useful for pages shared by several processes, but it is of much less value inside the boundaries of a given address space. One more bit is generally attached to each entry in the page table a valid-invalid bit.

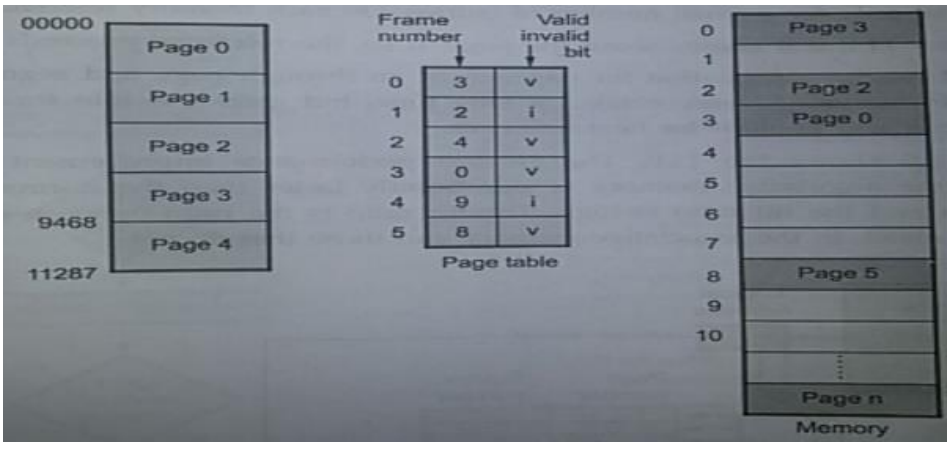


Fig. 4.7: Valid or invalid bit in a page table

When valid bit is set, then the page is in the process logical address space. Thus page is legal. If invalid bit is set, page is not in the process logical address space and illegal page. Illegal addresses are trapped by using the valid-invalid bit.

The operating system sets bits for each page to allow or disallow accesses to that page. A single physical copy of a shared page can be easily mapped into as many distinct address spaces as desired.

Hardware Support for Paging

A hardware mechanism is needed to perform the mapping from each instruction's effective address to the appropriate physical memory location. There must be separate register for each page. These may be actual high speed registers or a special area of physical memory.

Translation Look-aside Buffer (TLB)

Problem with paging is that, extra memory references to access translation tables can slow programs down by a factor of two or three. Too many entries in translation tables to keep them all loaded in fast processor memory. The standard solution to this problem is to use a special, small fast lookup hardware cache, called translation look-aside buffer. The TLB is associative, high-speed memory.

A translation buffer is used to store a few of the translation table entries fast, but only for a small number of entries. On each memory reference

1. First ask TLB if it knows about the page. If so, the reference proceeds fast.

2. If TLB has no information for page, must go through page and segment table to get information. Reference takes a long time, but gives the info for this page to TLB so it will know it for next reference. Fig. 4.8 shows the TLB. The greatest performance improvement is achieved when the associative memory is significantly faster than the normal page table lookup and the hit ratio is high. The hit ratio is the ratio between accesses that find a match in the associative memory and those that do not.

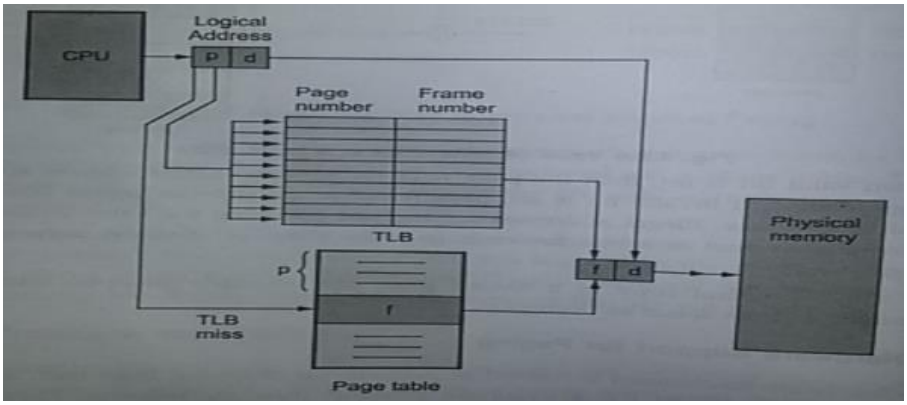


Fig 4.8: Paging with TLB.

Disadvantage of TLB is that if two pages use the same entry of the memory only one of them can be remembered at once. If process is referencing both pages at same time, TLB does not work very well. Fig. 4.9 shows flowchart for use of TLB. If the desired page is not in main memory, a page fault interrupt causes the page fault handling routine to be invoked.

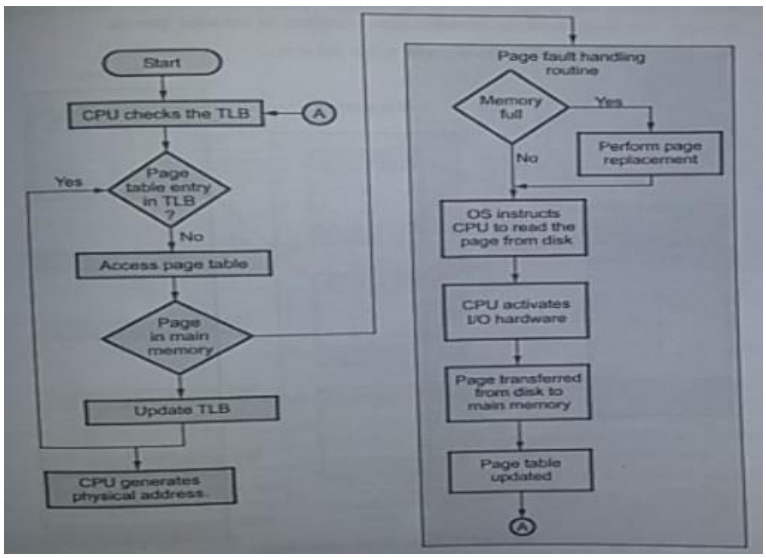


Fig 4.9: Flowchart for TLB

Each entry in the TLB must include the page number as well as the complete page table entry. The processor is equipped with hardware that allows it to simultaneously interrogate a number of TLB entries to determine if there is a match on page number. This technique is referred to as associative mapping.

STRUCTURE OF THE PAGE TABLE

Hierarchical Paging

Most modern computer systems support a large logical address space (2^{32} to 2^{64}). In this system, the page table itself becomes large. Clearly, we would not want to allocate the page table contiguously in main memory. One simple solution to this problem is to divide the page table into smaller pieces. We can accomplish this division in several ways. One way is to use a two-level paging algorithm, in which the page table itself is also paged (Fig 4.10).

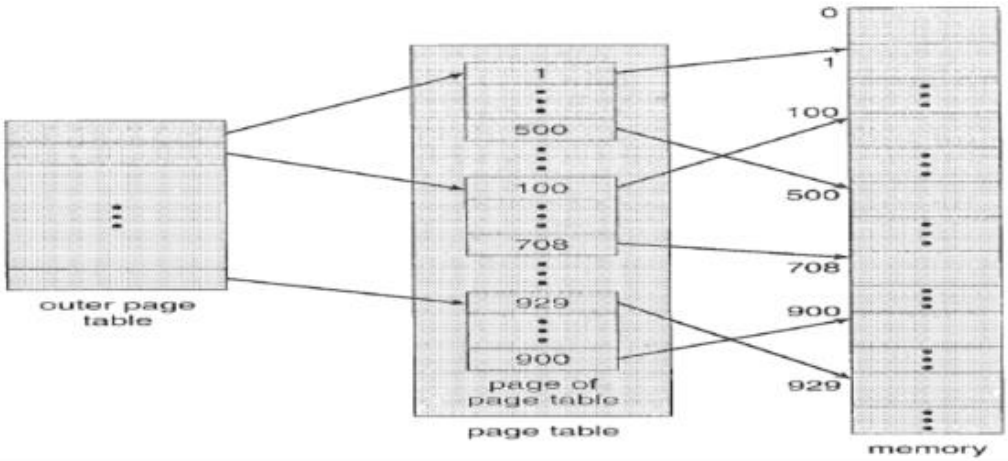


Fig 4.10: A two-level page-table scheme.

Hashed Page Tables

A common approach for handling address spaces larger than 32 bits is to use a hashed page table, with the hash value being the virtual page number. Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions). Each element consists of three fields: (1) the virtual page number, (2) the value of the mapped page frame, and (3) a pointer to the next element in the linked list.

The algorithm works as follows:

- The virtual page number in the virtual address is hashed into the hash table.
- The virtual page number is compared with field 1 in the first element in the linked list.
- If there is a match, the corresponding page frame (field 2) is used to form the desired physical address. If there is no match, subsequent entries in the linked list are searched for a matching virtual page number. This scheme is shown in Fig 4.11.

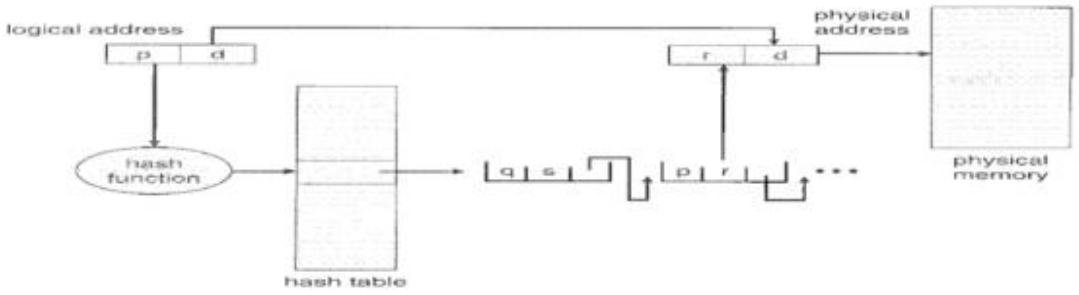


Fig 4.11: Hashed page table.

Inverted Page Tables

As address spaces have grown to 64 bits, the side of traditional page table becomes a problem. Even with two level page tables, the tables can themselves become too large. Inverted page table is used to solve this problem. The inverted page table has one entry for each real page of memory.

A physical page table instead of a logical one. The physical page table is often called as inverted page table. This table contains one entry per page frame. An inverted page table is very good at mapping from physical page to logical page number, but not very good at mapping from virtual page number to physical page number. Fig 4.12 shows the inverted page table.

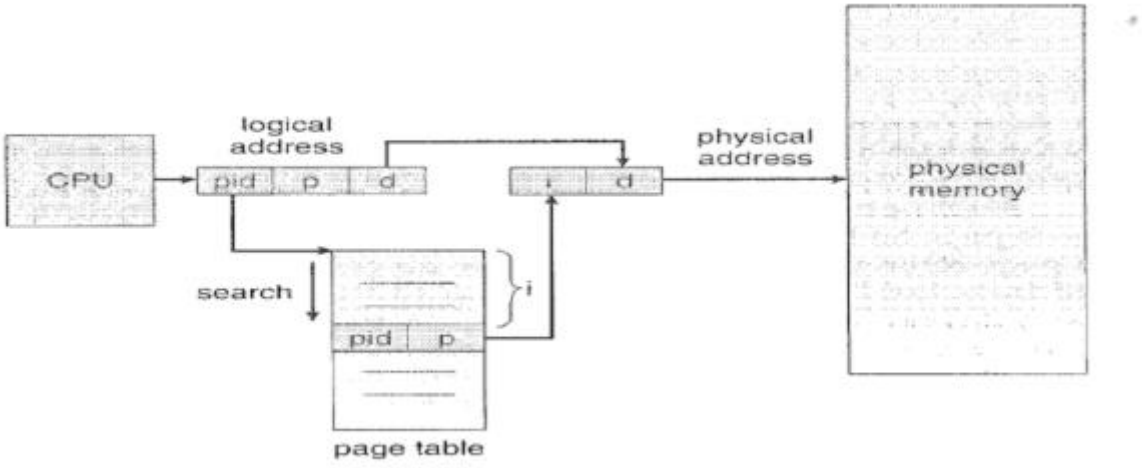


Fig 4.12: Inverted page table.

There is no other hardware or registers dedicated to memory mapping, the TLB ran be quite larger, so that missing entry faults are rare. With an inverted page table, most address translations are handled by the TLB when there is a miss in me translation look aside buffer, the operating system is notified and TLB miss handler is invoked. Hashing is used to speed up the search.

Advantages of Paging:

- Paging eliminates fragmentation.
- Support higher degree of multiprogramming.
- Paging increases memory and processor utilization
- Compaction overhead required for the re-locatable partition scheme is also eliminated.

Disadvantages:

- Page address mapping hardware usually increases the cost of the computer.
- Memory must be used to store various tables like page table, memory map table etc.
- Some memory will still be unused if the number of available block is not sufficient for the address spaces of the jobs to be run.

SEGMENTATION

Segmentation is a memory management scheme. Segmentation divides a program into a number of smaller blocks called segments. A segment can be defined as a logical grouping of information, such as a subroutine, array or data area. Segmentation is variable size. A logical address using segmentation consists of two parts: a segment number and an offset.

Segmentation is similar to dynamic partitioning because of the unequal size segments. Segmentation eliminates internal fragmentation but it suffers from external fragmentation. Segments are formed at program translation time by grouping together logically related items. Segmentation is usually visible and is provided as a convenience for organizing programs and data.

Hardware for Segmentation

A segmented address space can be implemented by using address mapping hardware. Figure shows the segmentation hardware. Segment table is used for mapping the two dimensional user-defined addresses into one dimensional physical addresses. Segment table contains the limit of segment and a segment base. The segment base contains the starting physical address where the segment resides in memory. Segment limit specifies the length of the segment.

A logical address consists of two parts: segment number (s) and offset into that segment (d). Segment number is used as index for the segment table. Each segment table entry would have to give the starting address in the main memory of the corresponding segment.

Consider an address of n + m bits ; where n bits are the segment number bits are the offset. Suppose n = 2 and m = 12, then the maximum segment 212 =4096.

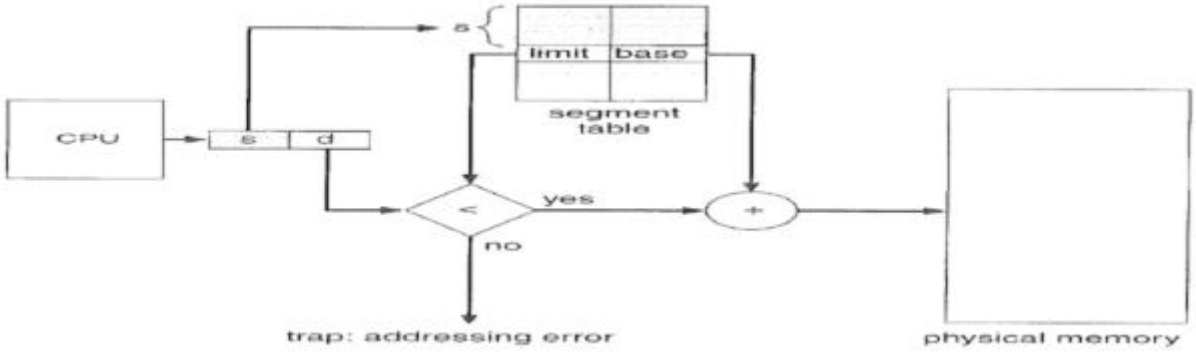


Fig 4.13: Segmentation hardware.

Protection and Sharing

Segments represent a semantically defined portion of the program segments are instructions, whereas other segments are data. The memory mapping hardware will check the protection bits associated with each s table entry to prevent illegal accesses to memory, such as attempts to write into a read only segment. Many common program errors will be detected hardware before they can cause serious damage.

Segments are shared when entries in the segment tables of two different processes point to the same physical location. The sharing occurs at the segment level. Several segments can be shared, so a program composed of several segments can be shared.

Advantages

- Segmentation eliminates fragmentation.
- It provides virtual memory. Allows dynamic segment growth.
- Segmentation assists dynamic linking. Segmentation is visible.

Disadvantages

- Maximum size of a segment is limited by the size of main memory.
- Difficulty to manage variable size segments on secondary storage.

Differences between Segmentation and paging

S No	Segmentation	Paging
1	Program is divided into variable size segments	Program is divided into fixed size pages
2	User (or compiler) is responsible for dividing the program into segments.	Division into pages is performed by the operating system
3	Segmentation is slower than paging	Paging is faster than segmentation
4	Segmentation is visible to the user.	Paging is invisible to the user.
5	Segmentation eliminates internal frag	Paging suffers from internal fragmentation
6	Segmentation suffers from external fragmentation	There is no external fragmentation
7	Processor uses page number, offset to calculate absolute address	Processor uses segment number offset to calculate absolute address
8	OS maintain a list of free holes in main memory	OS must maintain a free frame lie:

VIRTUAL MEMORY

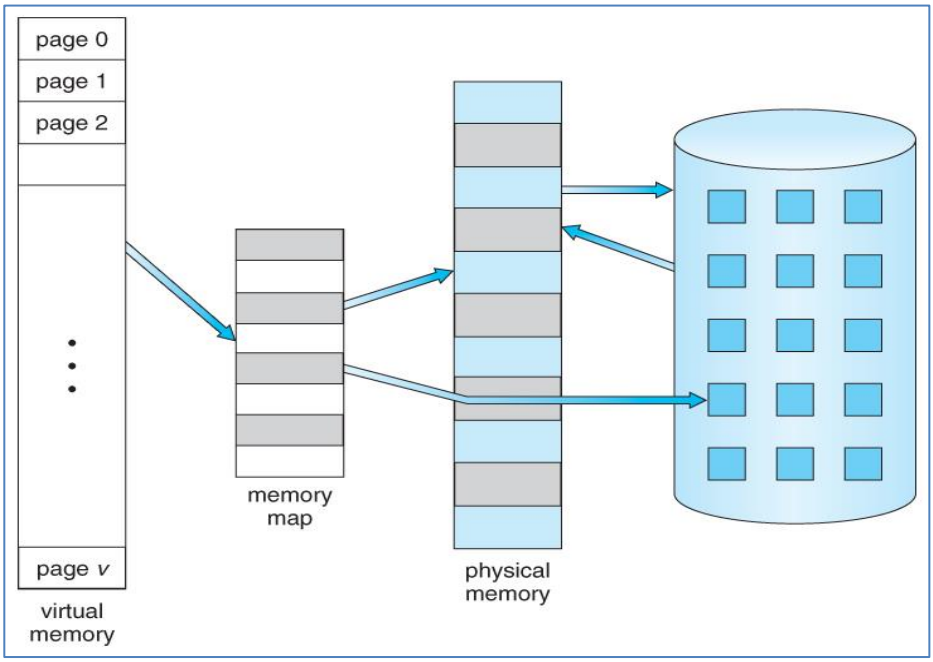


Fig 4.14: Diagram showing virtual memory that is larger than physical memory.

- Virtual memory is a technique which allows the execution of processes that are not completely in memory.

- The main advantage of this scheme is programs can be larger than physical memory.

- Virtual memory involves the separation of logical memory from physical memory.

This separation, allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available (Fig 4.14).

Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available.

The virtual address space of a process refers to the logical (or virtual) view of how a process is stored in memory. Typically, this view is that a process begins at a certain logical address say, address 0 and exists in contiguous memory, as shown in Fig 4.15.

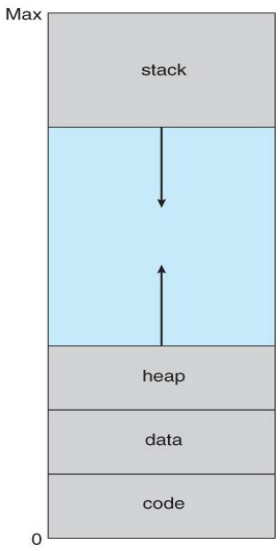


Fig 4.15: Virtual address space.

The physical memory may be organized in page frames and the physical page frames assigned to a process may not be contiguous. It is up to the memory-management unit (MMU) to map logical pages to physical page frames in memory.

Note in Fig 4.15 that we allow for the **heap to grow upward in memory**as it is used for dynamic memory allocation. Similarly, we allow for the **stack to grow downward in memory through successive function calls.**

The large blank space (or hole) between the heap and the stack is part of the virtual address spacebut will require actual physical pages only if the heap or stack grows.

Virtual address spaces that include holes are known as sparse address spaces.
Using a sparse address space is beneficial because the holes can be filled as the stack or heap segments grow or if we wish to dynamically link libraries (or possibly other shared objects) during program execution.

In addition to separating logical memory from physical memory, virtual memory also allows files and memory to be shared by two or more processes through page sharing. This leads to the following benefits:

- **System libraries can be shared by several processes through mapping of the shared object into a virtual address space.** Although each process considers the shared libraries to be part of its virtual address space, the actual pages where the libraries reside in physical memory are shared by all the processes (Fig 4.16). Typically, a library is mapped read-only into the space of each process that is linked with it.

- **virtual memory enables processes to share memory**. Virtual memory allows one process to create a region of memory that it can share with another process. Processes sharing this region consider it part of their virtual address space, yet the actual physical pages of memory are shared, much as is illustrated in Fig 4.16.

- **Virtual memory can allow pages to be shared during process creation with the fork() system call, thus speeding up process creation.**

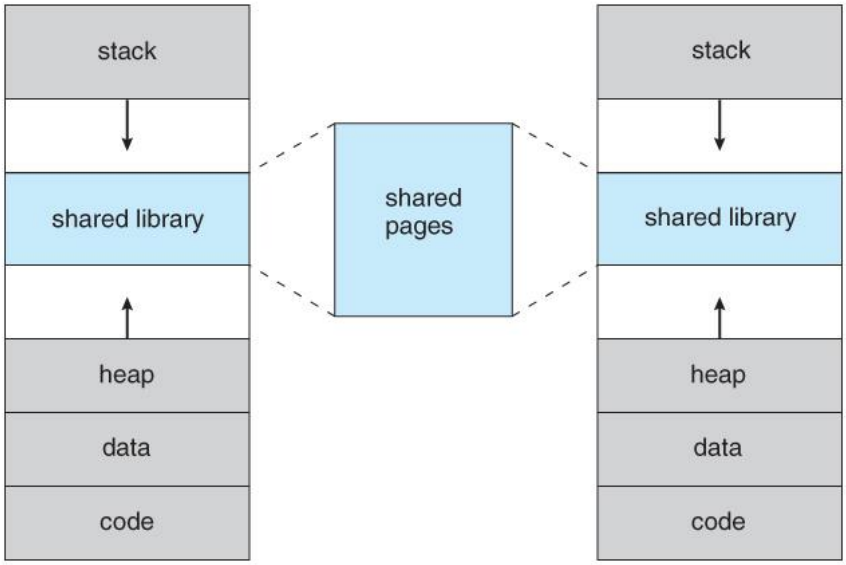


Fig 4.16: Shared library using virtual memory.

DEMAND PAGING

- The Demand Paging is same as Simple Paging.
- But the Main Difference is that in the Demand Paging Swapping is used. Means all the Pages will be in and out from the Memory when they are required.
- When we specify a Process for the Execution then the Processes is stored firstly on the Secondary Memory which is also known as the Hard Disk.

But when they are required then they are Swapped Backed into the Memory and when a Process is not used by the user then they are Temporary Swapped out from the Memory. Means they are Stored on the Disk and after that they are Copied into the Memory.

But when they are required then they are Swapped Backed into the Memory and when a Process is not used by the user then they are Temporary Swapped out from the Memory. Means they are Stored on the Disk and after that they are Copied into the Memory.

Demand Paging is the Concept in which a Process is Copied into the Logical Memory from the Physical Memory when we needs them.

A Process can load either Entire, Copied into the Main Memory or the part of single Process is copied into the Memory so that is only the single Part of the Process is copied into the Memory then this is also called as the <u>Lazy Swapping</u>.

A demand-paging system is similar to a paging system with swapping (Fig 4.17) where processes reside in secondary memory (usually a disk).

When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, however, we use a lazy swapper.

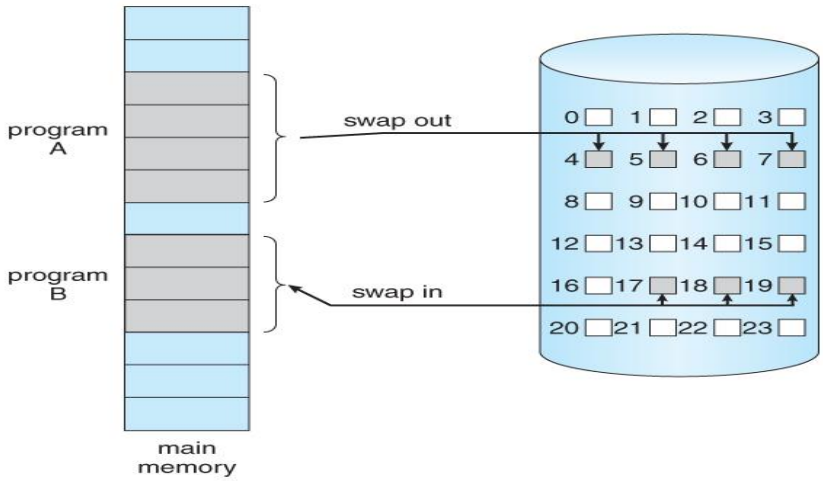


Fig 4.17: Transfer of a paged memory to contiguous disk space.

Basic Concepts

- Hardware support is required to distinguish between the pages that are in memory and the pages that are on the disk.
- The valid-invalid bit scheme can be used for this purpose. This time, however, when this bit is set to "valid," the associated page is both legal and in memory.
- If the bit is set to "invalid," the page either is not valid or is valid but is currently on the disk.
- The page-table entry for a page that is brought into memory is set as usual, but the page-table entry for a page that is not currently in memory is either simply marked invalid or contains the address of the page on disk. This situation is depicted in Fig 4.18.

Notice that marking a page invalid will have no effect if the process never attempts to access that page. Hence, if we guess right and page in all and only those pages that are actually needed, the process will run exactly as though we had brought in all pages. While the process executes and accesses pages that are memory resident, execution proceeds normally.

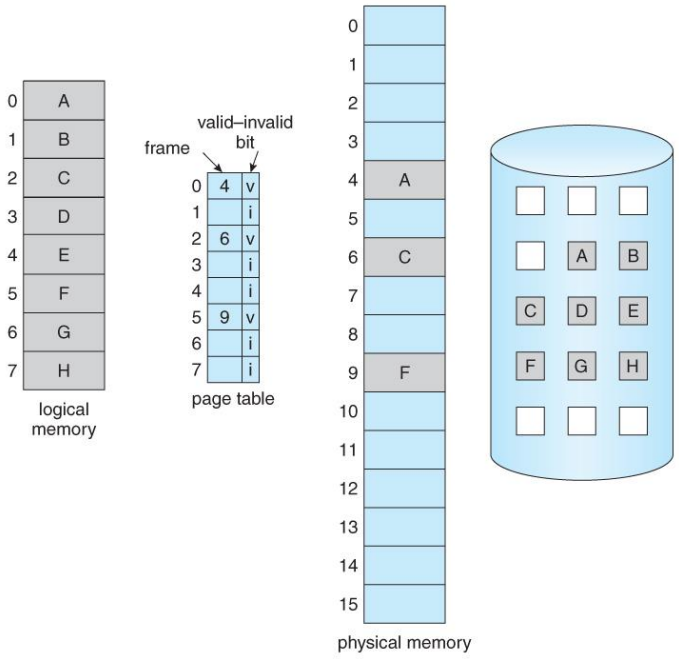


Fig 4.18: Page table when some pages are not in main memory.

But what happens if the process tries to access a page that was not brought into memory?

Access to a page marked **invalid** causes a **page-fault trap**.

The paging hardware, in translating the address through the page table, will notice that the invalid bit is set, causing a trap to the operating system. This trap is the result of the operating system's failure to bring the desired page into memory.

The procedure for handling this page fault is straightforward (Fig 4.19).

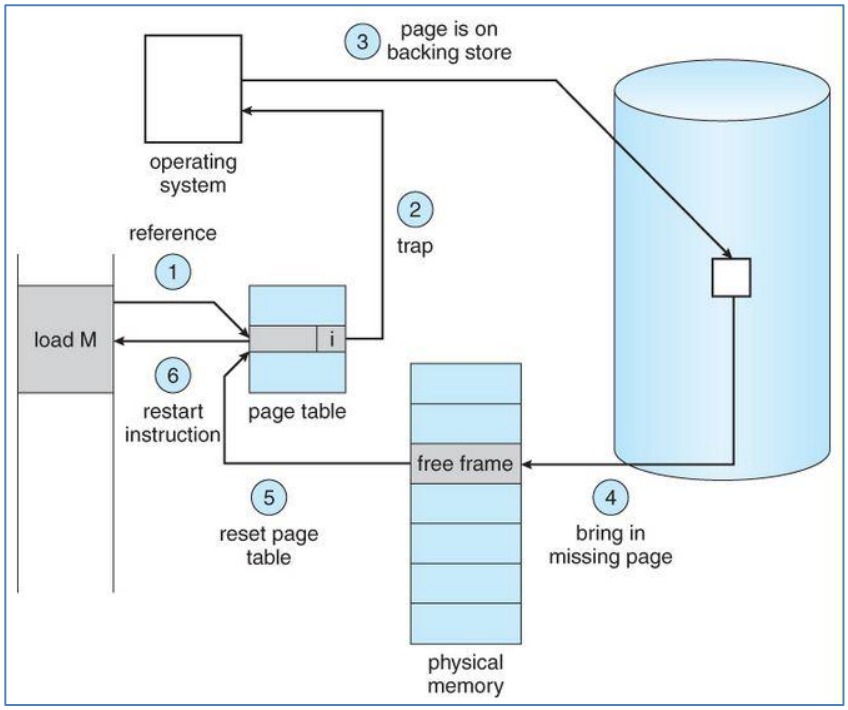


Fig 4.19: Steps in handling a page fault.

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

The hardware to support demand paging is the same as the hardware for paging and swapping:
• **Page table**. This table has the ability to mark an entry invalid through a valid-invalid bit or special value of protection bits.
• **Secondary memory**. This memory holds those pages that are not present in main memory.
The secondary memory is usually a high-speed disk. It is known as the swap device, and the section of disk used for this purpose is known as swap space.

A crucial requirement for demand paging is the need to be able to restart any instruction after a page fault. Because we save the state of the interrupted process when the page fault occurs, we must be able to restart the process in exactly the same place and state, except that the desired page is now in memory and is accessible.

PAGE REPLACEMENT

- Page replacement policy deals with the selection of a page in memory to be replaced when a new page must be brought in. While a user process is executing a page fault occurs.
- The hardware traps to the operating system, which checks its internal tables to see that this page fault is a genuine one rather than an illegal memory access.
- The operating system determines where the desired page is residing on the disk, but then finds that there are no free frames on the free frame list. All memory is in use.
- This is shown in Fig 4.20 when all the frames in main memory are occupied and it is necessary to bring in a new page to satisfy a page fault, replacement policy is concerned with selecting a page currently in memory to be replaced.
- All the policies have as their objective that the page that is removed should be the page least likely to be referenced in the near future.

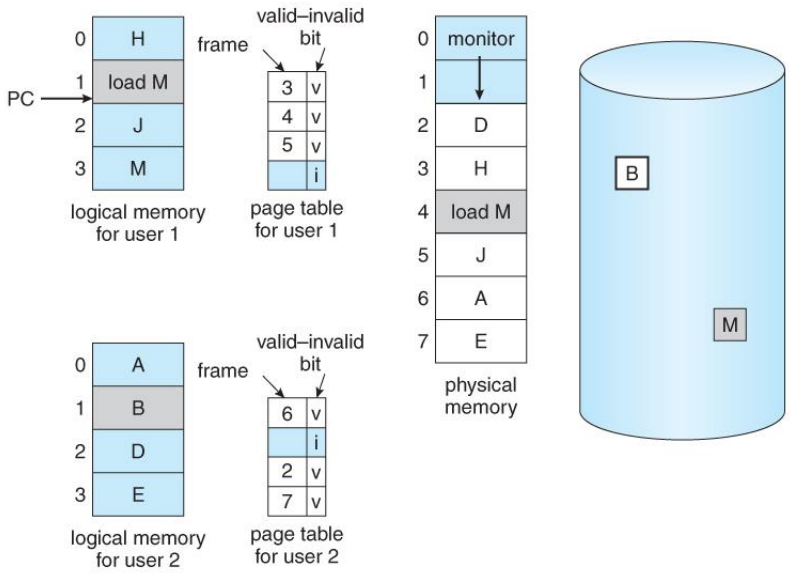


Fig 4.20: Need for page replacements.

Basic Page Replacement

Page replacement takes the following approach. If no frame is free, we find one that is not currently being used and free it. We can free a frame by writing its contents to swap space and changing the page table (and all other tables) to indicate that the page is no longer in memory (Fig 4.21). We can now use the freed frame to hold the page for which the process faulted. We modify the page-fault service routine to include page replacement:

1. Find the location of the desired page on the disk.
2. Find a free frame:
    a. If there is a free frame, use it.
    b. If there is no free frame, use a page-replacement algorithm to select a victim frame.
    c. Write the victim frame to the disk; change the page and frame tables accordingly.
3. Read the desired page into the newly freed frame; change the page and frame tables.
4. Restart the user process.

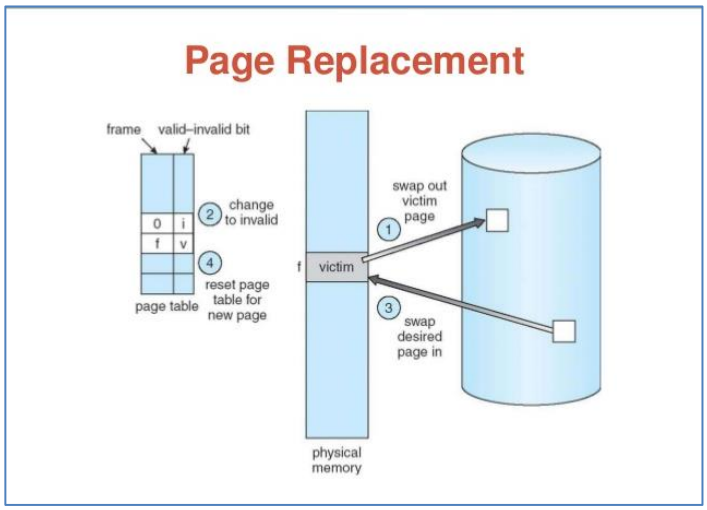


Fig 4.21: Page replacements.

FIFO Page Replacement

- The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm where replacement associates with each page the time when brought into memory. When a page must be replaced, the oldest page is chosen.
- We can create a FIFO queue to hold all pages in memory.
- We replace the page at the head of the queue.
- When a page is brought into memory, we insert it at the tail of the queue.

For our example reference string, our three frames are initially empty.

- The first three references (7, 0, 1) cause page faults and are brought into these empty frames. The next reference (2) replaces page 7, because page 7 was brought in first. Since 0 is the next reference and 0 is already in memory, we have no fault for this reference.
- The first reference to 3 results in replacement of page 0, since it is now first in line. Because of this replacement, the next reference, to 0, will fault.
- Page 1 is then replaced by page 0.
- This process continues as shown in Fig 4.22. Every time a fault occurs, we show which pages are in our three frames. There are 15 faults altogether.

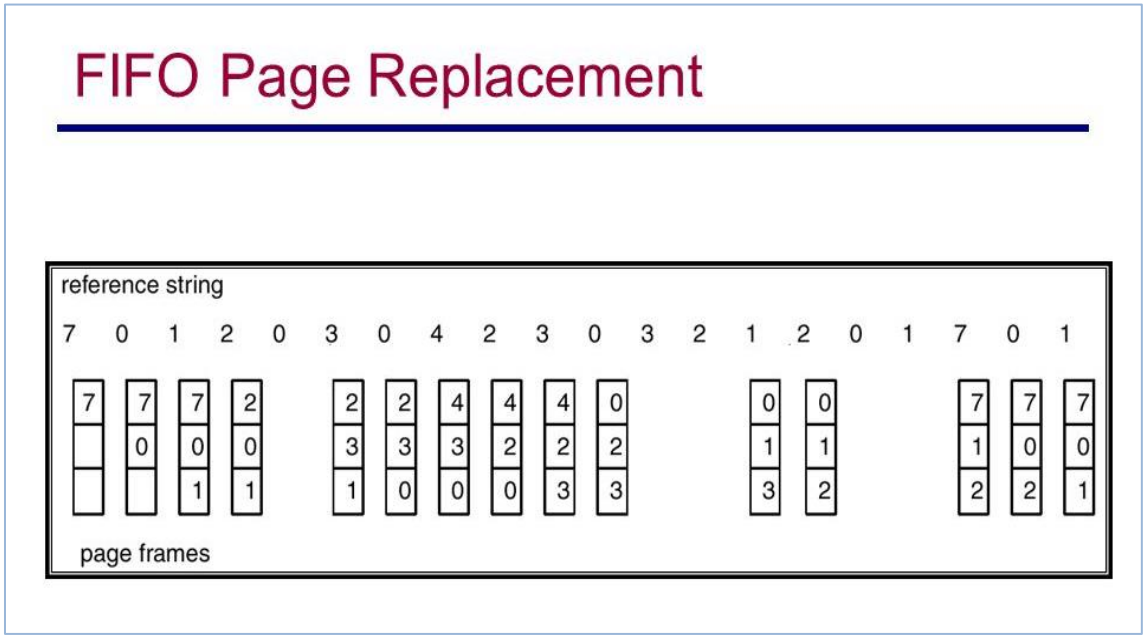


Fig 4.22: FIFO page-replacement algorithm.

- The FIFO page-replacement algorithm is easy to understand and program, its performance is not always good. Notice that, even if we select for replacement a page that is in active use, everything still works correctly.

- After we replace an active page with, a new one, a fault occurs almost immediately to retrieve the active page.
- Some other page will need to be replaced to bring the active page back into memory.
- Thus, a bad replacement choice increases the page-fault rate and slows process execution. It does not, however, cause incorrect execution.

Optimal Page Replacement

One result of the discovery of Belady's anomaly was the search for an optimal page-replacement algorithm. An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly. Such an algorithm does exist and has been called OPT or MIN.

Replace the page that will not be used for the longest period of time. Use of this page-replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames.

For example, on our sample reference string, the optimal page-replacement algorithm would yield nine page faults, as shown in Fig 4.23. The first three references cause faults that fill the three empty frames. The reference to page 2 replaces page 7, because 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With only nine page faults, optimal replacement is much better than a FIFO algorithm, which resulted in fifteen faults.

(If we ignore the first three, which all algorithms must suffer, then optimal replacement is twice as good as FIFO replacement.) In fact, no replacement algorithm can process this reference string in three frames with fewer than nine faults.

Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string. As a result, the optimal algorithm is used mainly for comparison studies.

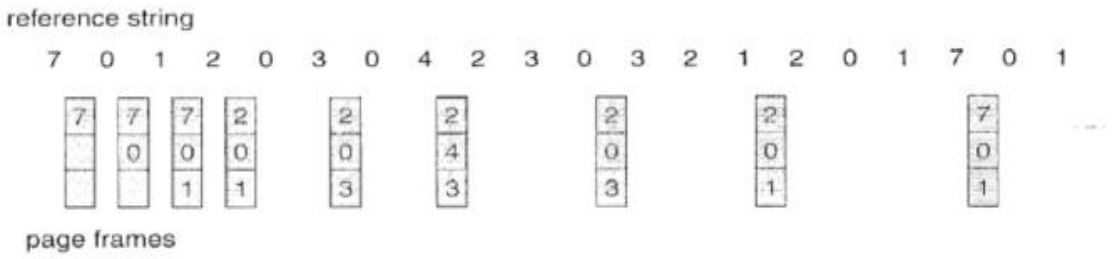


Fig 4.23: Optimal page replacement algorithms.

LRU Page Replacement

LRU replacement associates with each page the time of that page's last use.
When a page must be replaced, LRU chooses the page that has not been used for the longest period of time. We can think of this strategy as the optimal page-replacement algorithm looking backward in time, rather than forward.

The result of applying LRU replacement to our example reference string is shown in Fig 4.24. The LRU algorithm produces 12 faults. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently.
Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used. When it then faults for page 2, the LRU algorithm replaces page 3, since it is now the least recently used of the three pages in memory. Despite these problems, LRU replacement with 12 faults is much better than FIFO replacement with 15.

The LRU policy is often used as a page-replacement algorithm and is considered to be good. An LRU page-replacement algorithm may require substantial hardware assistance to implement. The problem is to determine an order for the frames defined by the time of last use.

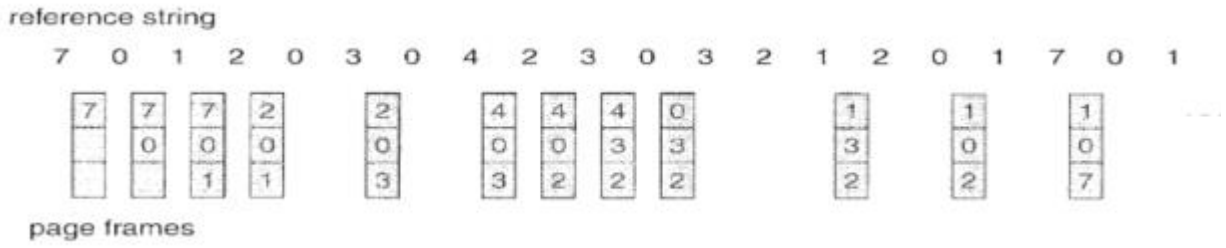


Fig 4.24: LRU page replacement algorithms.

LRU-Approximation Page Replacement

Few computer systems provide sufficient hardware support for true LRU page replacement. Some systems provide no hardware support, and other page replacement algorithms (such as a FIFO algorithm) must be used. Many systems provide some help, however, in the form of a reference bit. The reference bit for a page is set by the hardware whenever that page is referenced (either a read or a write to any byte in the page). Reference bits are associated with each entry in the page table.

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

determine which pages have been used and which have not been used by examining the reference bits, although we do not know the order of use. This information is the basis for many page-replacement algorithms that approximate LRU replacement.

Counting-Based Page Replacement

There are many other algorithms that can be used for page replacement. For example, we can keep a counter of the number of references that have been made to each page and develop the following two schemes.

• The least frequently used (LFU) page-replacement algorithm requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count. A problem arises, however, when a page is used heavily during the initial phase of a process but then is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed. One solution is to shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average usage count.

• The most frequently used (MFU) page-replacement algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.
As you might expect, neither MFU nor LFU replacement is common. The implementation of these algorithms is expensive, and they do not approximate OPT replacement well.

THRASHING

- A process that is spending more time paging than executing is said to be thrashing.
- In other words it means, that the process doesn't have enough frames to hold all the pages for its execution, so it is swapping pages in and out very frequently to keep executing. Sometimes, the pages which will be required in the near future have to be swapped out.

Initially when the CPU utilization is low, the process scheduling mechanism, to increase the level of multiprogramming loads multiple processes into the memory at the same time, allocating a limited amount of frames to each process.

As the memory fills up, process starts to spend a lot of time for the required pages to be swapped in, again leading to low CPU utilization because most of the processes are waiting for pages. Hence the scheduler loads more processes to increase CPU utilization, as this continues at a point of time the complete system comes to a stop.

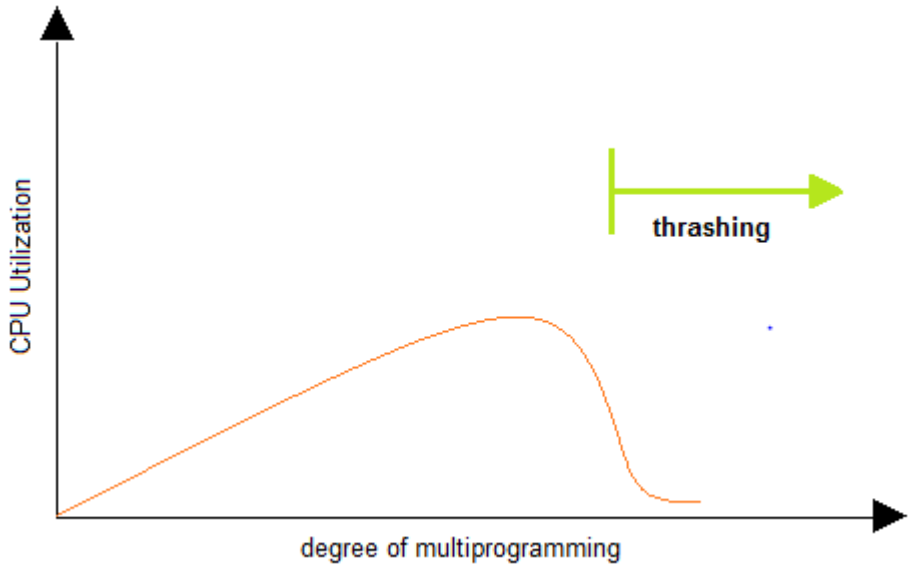


Fig 4.25: Thrashing.

Causes of Thrashing

It results in severe performance problems.

1) If CPU utilization is too low then we increase the degree of multiprogramming by introducing a new process to the system. A global page replacement algorithm is used. The CPU scheduler sees the decreasing CPU utilization and increases the degree of multiprogramming.

2) CPU utilization is plotted against the degree of multiprogramming.

3) As the degree of multiprogramming increases, CPU utilization also increases.

4) If the degree of multiprogramming is increased further, thrashing sets in and CPU utilization drops sharply.

5) So, at this point, to increase CPU utilization and to stop thrashing, we must decrease the degree of multiprogramming.

How to prevent Thrashing

- We must provide a process with as many frames as it needs. Several techniques are used.
- The Working of Set Model (Strategy) It starts by looking at how many frames a process is actually using. This defines the locality model.
- Locality Model It states that as a process executes, it moves from locality to locality.
- A locality is a set of pages that are actively used together.
- A program is generally composed of several different localities which overlap.

DEADLOCKS

When processes request a resource and if the resources are not available at that time the process enters into waiting state. Waiting process may not change its state because the resources they are requested are held by other process. This situation is called deadlock. The situation where the process waiting for the resource i.e., not available is calleddeadlock.

System Model

A system may consist of finite number of resources and is distributed among number of processes. There resources are partitioned into several instances each with identical instances,

A process must request a resource before using it and it must release the resource after using it. It can request any number of resources to carry out a designated task. The amount of resource requested may not exceed the total number of resources available.

A process may utilize the resources in only the following sequences:

1. Request: If the request is not granted immediately then the requesting process must wait it can acquire the resources.

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

3. Release:- The process releases the resource after using it.

Deadlock may involve different types of resources.

For eg:- Consider a system with one printer and one tape drive. Ifa process Pi currently holds a printer and a process Pj holds the tape drive. If process Pi request a tape drive and process Pj request a printer then a deadlock occurs. Multithread programs are good candidates for deadlock because they compete for shared resources.

Deadlock Characterization

Necessary Conditions:

A deadlock situation can occur if the following 4 conditions occur simultaneously in a system

1. Mutual Exclusion:

Only one process must hold the resource at a time. If any other process requests for the resource, the requesting process must be delayed until the resource has been released,

2. Hold and Wait:

A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by the other process.

3. No Preemption:

Resources can't be preempted i.e., only the process holding the resources must release it after the process has completed its task.

4. Circular Wait:

A set {PO.P1.....Pn} of waiting process must exist such that PO is waiting for a resource i.e., held by P1, Pl is waiting for a resource ie., held by P2. Pn-l is waiting for resource held by process Pn and Pn is waiting for the resource i.e., held by P1.

All the four conditions must hold for a deadlock to occur.

Resource Allocation Graph:

- Deadlocks are described by using a directed graph called system resource allocation graph. The graph consists of set of vertices (v) and set of edges (e).
- The set of vertices (v) can be described into two different types of nodes

P={P1,P2,……. Pn} ie, set consisting of all active processes and

R={R1,R2.....Rn} i.e., set consisting of all resource types in the system.

- A directed edge from process Pi to resource type Rj denoted by Pi →Ri indicates that Pi requested an instance of resource Rj and is waiting. This edge is called Request edge.
- A directed edge Ri → Pj signifies that resource Rj is held by process Pi. This is called Assignment edge.
- If the graph contain no cycle, then no process in the system is deadlock. If the graph contains a cycle then a deadlock may exist.
- If each resource type has exactly one instance than a cycle implies that a deadlock has occurred. If each resource has several instances then a cycle do not necessarily implies that a deadlock has occurred.

Methods for Handling Deadlocks:

There are three ways to deal with deadlock problem

- **We can use a protocol to prevent deadlocks ensuring that the system will never enter into the deadlock state.**
- **We allow a system to enter into deadlock state, detect it and recover from it.**
- **We ignore the problem and pretend that the deadlock never occur in the system. This is used by most OS including UNIX.**
- To ensure that the deadlock never occur the system can use either deadlock avoidance or a deadlock prevention.
- Deadlock prevention is a set of method for ensuring that at least one of the necessary conditions does not occur.
- Deadlock avoidance requires the OS is given advance information about which resource a process will request and use during its lifetime.
- If a system does not use either deadlock avoidance or deadlock prevention then a deadlock situation may occur. During this it can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred and algorithm to recover from deadlock.
- Undetected deadlock will result in deterioration of the system performance.

Deadlock Prevention:

- For a deadlock to occur each of the four necessary conditions must hold. If at least one of the three conditions does not hold then we can prevent occurrence of deadlock.

1. **Mutual Exclusion:** This holds for non-sharable resources

Eg:- A printer can be used by only one process at a time.

Mutual exclusion is not possible in sharable resources and thus they cannot be involved in deadlock. Read-only files are good examples for sharable resources. A process never waits for accessing a sharable resource. So we cannot prevent deadlock by denying the mutual exclusion condition in non-sharable resources.

2. **Hold and Wait:** This condition can be eliminated by forcing a process to release all its resources held by it when it request a resource i.e., not available

   * One protocol can be used is that each process is allocated with all of its resources before its start execution.

   Eg:- consider a process that copies the data from a tape drive to the disk, sorts the file Eg:- consider a process that copies the data from a tape drive to the disk, sorts the file and then prints the results to a printer. If all the resources are allocated at the beginning then the tape drive, disk files and printer are assigned to the process. The main problem with this is it leads to low resource utilization because it requires printer at the last and is allocated with it from the beginning so that no other process can use it.

   * Another protocol that can be used is to allow a process to request a resource when the process has none. ie., the process is allocated with tape drive and disk file. It performs the required operation and releases both. Then the process once again request for disk file and the printer and the problem and with this is starvation is possible.

3. **No Preemption:**

   To ensure that this condition never occurs the resources must be preempted. The following protocol can be used.

   * If a process is holding some resource and request another resource that cannot be immediately allocated to it, then all the resources currently held by the requesting process are preempted and added to the list of resources for which other processes may be waiting. The process will be restarted only when it regains the old resources and the new resources that it is requesting.

   * When a process request resources, we check whether they are available or not. If they are available we allocate them else we check that whether they are allocated to some other waiting process. If so we preempt the resources from the waiting process and allocate them to the requesting process. The requesting process must wait.

4. **Circular Wait:**

   The fourth and the final condition for deadlock is the circular wait condition. One way to ensure that this condition never, is to impose ordering on all resource types and each process requests resource in an increasing order.

   Let R={R1.R2.....Rn} be the set of resource types. We assign each resource type with a unique integer value. This will allows us to compare two resources and determine whether one precedes the other in ordering.

Eg:-we can define a one to one function

F:R →N as follows :- F(disk drive)=5

F(printer)=12

F(tape drive)=1

Deadlock can be prevented by using the following protocol:-

* Each process can request the resource in increasing order. A process can request any number of instances of resource type say Ri and it can request instances of resource type Rj only F(Rj) > F(Ri).

* Alternatively when a process requests an instance of resource type Rij, it has released any resource Ri such that F(Ri) >= F(Rj).

If these two protocols are used then the circular wait can't hold.

Deadlock Avoidance:

- Deadlock prevention algorithm may lead to low device utilization and reduces system throughput.
- Avoiding deadlocks requires additional information about how resources are to be requested. With the knowledge of the complete sequences of requests and releases we can decide for each requests whether or not the process should wait.
- For each requests it requires to check the resources currently available. resources that are currently allocated to each processes future requests and release of each process to decide whether the current requests can be satisfied or must wait to avoid future possible deadlock.
- A deadlock avoidance algorithm dynamically examines the resources allocation state to ensure that a circular wait condition never exists. The resource allocation state is defined by the number of available and allocated resources and the maximum demand of each process.

Safe State:

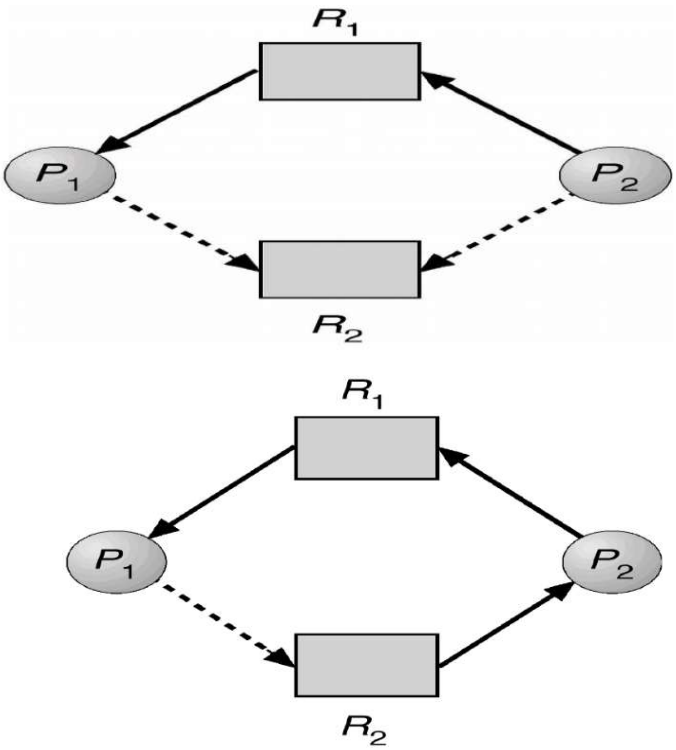
- A state is a safe state in which there exists at least one order in which all the process will run completely without resulting in a deadlock.
- A system is in safe state if there exists a safe sequence.
- A sequence of processes <P1,P2 ...Pn> is a safe sequence for the current allocation state if for each Pi the resources that Pi can request can be satisfied by the currently available resources.
- If the resources that Pi requests are not currently available then Pi can obtain all of its needed resource to complete its designated task.
- A safe state is not a deadlock state.
- Whenever a process request a resource ie., currently available, the system must decide whether resources can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in safe state.
- In this, if a process requests a resource ie., currently available it must still have to wait.Thus resource utilization may be lower than it would be without a deadlock avoidance algorithm.

Resource Allocation Graph Algorithm:

- This algorithm is used only if we have one instance of a resource type.
- In addition to therequest edge and the assignment edge a new edge called claim edge is used.

For eg:- A claim edge Pi→Rj indicates that process Pi may request Rj in future. Theclaim edge is represented by a dotted line

- When a process Pi requests the resource Rj, the claim edge is converted to arequest edge.
- When resource Rj is released by process Pi, the assignment edge Rj → Pi isreplaced by the claim edge Pi →Rj.
- When a process Pi requests resource Rj the request is granted only if converting the request edge Pi>Rj to as assignment edge Rj>Pi do not result in a cycle. Cycle detection algorithm is used to detect the cycle. If there are no cycles then the allocation of the resource to process leave the system in safe state.



Bankers Algorithm

This algorithm is applicable to the system with multiple instances of each resource types, but this is less efficient then the resource allocation graph algorithm.

When a new process enters the system it must declare the maximum number of resources that it may need. This number may not exceed the total number of resources in the system. The system must determine that whether the allocation of the resources will leave the system in a safe state or not. If it is so resources are allocated else it should wait until the process release enough resources.

Several data structures are used to implement the banker's algorithm. Let 'n' be the number of processes in the system and 'm' be the number of resources types. We need the following data structures:

Available: A vector of length m indicates the number of available resources. If Available[i]=k, then k instances of resource type Rj is available.

Max: Ann*m matrix defines the maximum demand of each process if Max[i,j]=k, then Pi may request at most k instances of resource type Rj.

Allocation: An n*m matrix defines the number of resources of each type currently allocated to each process. If Allocation[i,j]=k. then Pi is currently k instances of resource type Rj

Need: An n*m matrix indicates the remaining resources need of each process. If Need [i,j]=k. then Pi may need k more instances of resource type Rj to compute its task.

So Need[i,j]=Max[i,j]-Allocation[i]

Safety Algorithm

This algorithm is used to find out whether or not a system is in safe state or not.

Step 1. Let work and finish be two vectors of length M and N respectively.

Initialize work = available and

Finish[i] = false for i=1.2.3.....0

Step 2. Find i such that both

Finish[i] = false

Need i <= work

If no such i exist then go to step 4

Step 3. Work = work + Allocation

Finish[i]=true

Go to step 2

Step 4. If finish[i]=true for all i, then the system is in safe state.

Resource Request Algorithm:

Let Request(i) be the request vector of process Pi. If $Request_i$ [j] = k. then process Pi wants K instances of the resource type Rj. When a request for resources is made by process Pi the following actions are taken.

1. If $Request_i$ <= $Need_i$ go to step 2 otherwise raise an error condition since the process has exceeded its maximum claim.
2. If $Request_i$ <= Available go to step 3 otherwise Pi must wait. Since the resources are not available.
3. If the system want to allocate the requested resources to process Pi then modify the state as follows

   Available = Available — $Request_i$

   $Allocation_i$= $Allocation_i$+ $Request_i$

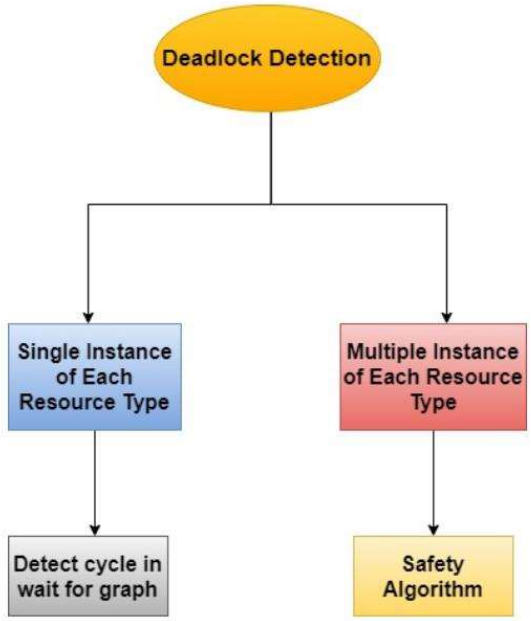
   $Need_i$= $Need_i$— $Request_i$

If the resulting resource allocation state is safe, the transaction is complete and Pi is allocated its resources. If the new state is unsafe then Pi must wait for Request(i) and old resource allocation state is restored.

Deadlock Detection:

If a system does not employ either deadlock prevention or a deadlock avoidance algorithm then a deadlock situation may occur. In this environment the system may provide:

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

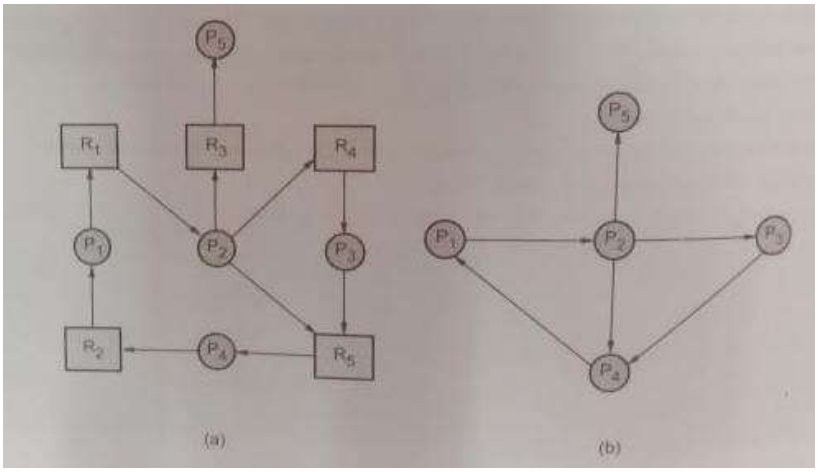


Single Instances of each Resource Type:

If all the resources have only a single instance, then a deadlock-detection algorithm can be defined that mainly uses the variant of the resource-allocation graph and is known as a wait-for graph. This wait-for graph is obtained from the resource-allocation graph by removing its resource nodes and collapsing its appropriate edges.

- An edge from Pi to Pj in wait for graph implies that Pi is waiting for Pj to release a resource that Pi needs.
- An edge from Pi to Pj exists in wait for graph if and only if the corresponding resource allocation graph contains the edges Pi → Rq and Rq → Pj

Deadlock exists with in the system if and only if there is a cycle. In order to detect the deadlock, the system needs to maintain the wait-for graph and periodically system invokes an algorithm that searches for the cycle in the **wait-for** graph.



Resource Allocation Graph with corresponding Wait for Graph

Multiple Instances of Each Resource Type

- The above scheme that is a wait-for graph is not applicable to the resource-allocation system having multiple instances of each resource type. Now we will move towards a deadlock detection algorithm that is is applicable for such systems.
- This algorithm mainly uses several time-varying data structures that are similar to those used in Banker's Algorithm and these are as follows:

  1. Available: It is an **array** of length m. It represents the number of available resources of each type.

  2. Allocation: It is an n x m matrix which represents the number of resources of each type currently allocated to each process.

  3. Request: It is an **n*m** matrix that is used to indicate the request of each process; if Request[i][j] equals to k, then process Pi is requesting k more instances of resource type Ri.

  - Allocation and Request are taken as vectors and referred to as Allocation and Request. The Given detection algorithm is simply used to investigate every possible allocation sequence for the processes that remain to be completed.

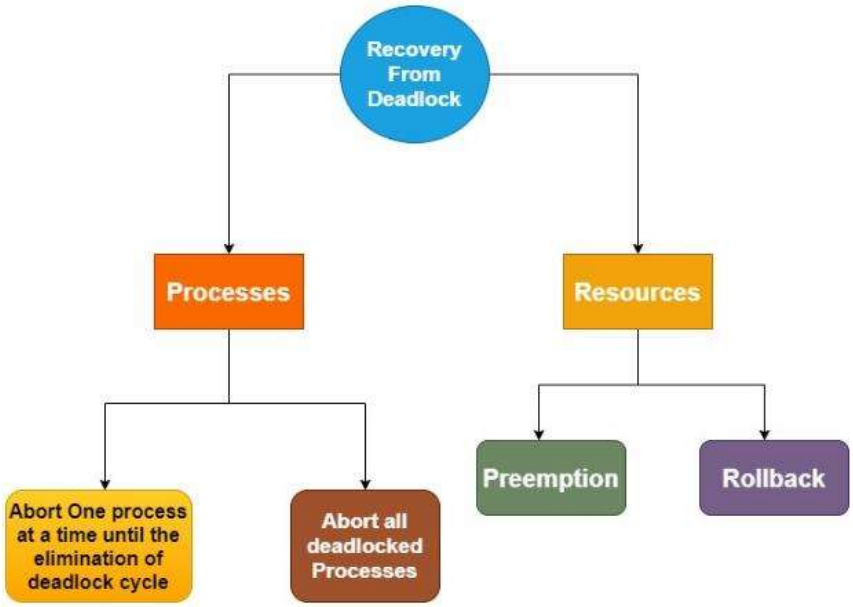
Algorithm:

1. Let Work and Finish be vectors of length **m** and **n**, respectively. Initialize:
   Work = Available
   Finish [i] = False if allocation not equal to **0**
   Finish [i] = True if allocation = 0
2. Find an i such that
   Finish [i] = False
   Request <= Work
3. Work = Work + Allocation
   Finish[i] = true, goto step 2
4. If Finish[i] == false for some i, 0<=i<n, then it means the system is in a deadlocked state otherwise safe state.

Recovery from Deadlock

When a detection algorithm determines that a deadlock exists then there are several available alternatives. There one possibility and that is to inform the operator about the deadlock and let him deal with this problem manually.

Another possibility is to let the system recover from the deadlock automatically. These are two options that are mainly used to break the deadlock.



1. Process Termination

In order to eliminate deadlock by aborting the process, we will use one of two methods given below. In both methods, the system reclaims all resources that are allocated to the terminated processes.

- **Aborting all deadlocked Processes** Clearly, this method is helpful in breaking the cycle of deadlock, but this is an expensive approach. This approach is not suggestible but can be used if the problem becomes very serious. If all the processes are killed then there may occur insufficiency in the system and all processes will execute again from starting.
- **Abort one process at a time until the elimination of the deadlock cycle** This method can be used but we have to decide which process to kill and this method incurs considerable overhead. The process that has done the least amount of work is killed by the Operating system firstly.

- Many factors may affect which process is chosen, including:

a. Least amount of processor time consumed so far.

b. Least amount of output produced so far.

c. Lowest Priority

d. Most estimated time remaining.

e. Least total resources allocated so far.

2. Resource Preemption

If preemption is required to deal with deadlocks then three issues need to be addressed:

1. **Selecting a Victim:** Which resources and which processes are to be preempted?
2. **Rollback:** Backup of each deadlocked process to some previously defined check point and restart all processes. This requires rollback and restart mechanisms are built in to the system.
3. **Starvation:** How do we ensure that starvation will not occur?

Methods for Handling Deadlock:

1. Deadlock Prevention
2. Deadlock Avoidance
3. Deadlock Detection
4. Deadlock Recovery