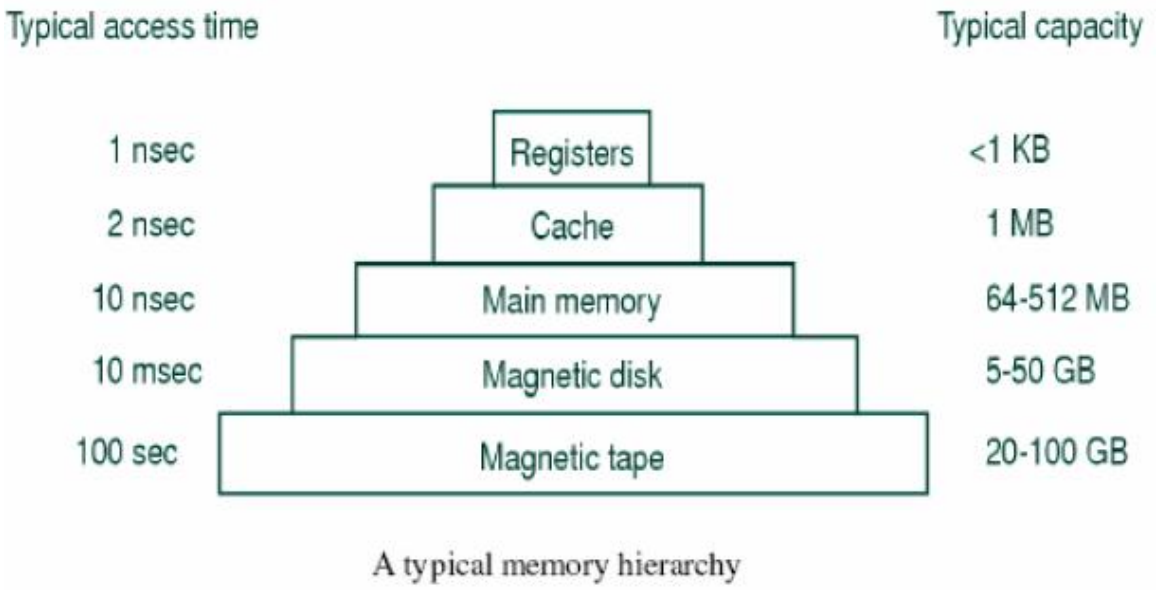
Memory Management

1. Introduction:

Memory is an important resource that must be carefully managed. The part of the operating system that manages the memory hierarchy is called the memory manager. Its job is to keep track of which parts of memory are to processes when they need it and de-allocate it when they are done and to manage swapping between main memory and disk when main memory is too small to hold the processes.

Ideally, every programmer would like is an infinitely large, infinitely fast memory that is also nonvolatile, that is, does not lose its content when the electric power fails. But technology does not provide such memories. Consequently, most computers have a memory hierarchy with a small amount of very fast, expensive cache memory, tens of megabytes of medium-speed, medium price, volatile main memory and hundreds of gigabytes of slow, cheap , nonvolatile disk storage. It is the job of operating system to coordinate how these memories are used.



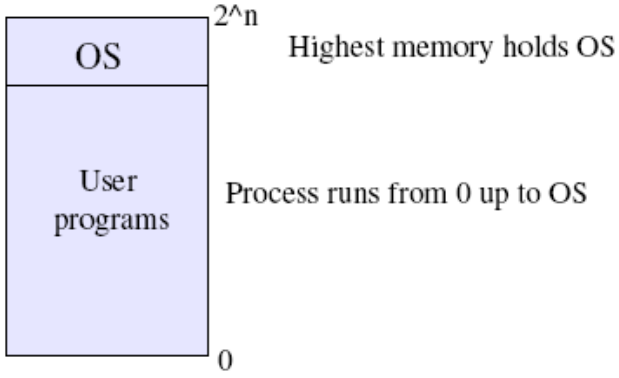
A typical memory hierarchy

2. Continuous and Non-Continuous Storage Allocation

The earliest computing systems require contiguous storage allocation each program had to occupy a single contiguous block of storage location. In non-continuous storage allocation a program is divided into several blocks or segments that may be placed throughout main storage in piece not necessarily adjacent to one another. The benefit of non-continuous is that if main storage has many small holes available instead of a single large hole, then the operating system can often load and execute a program that would otherwise need to wait.

3. Mono-programming without swapping and paging:

The simplest possible memory management scheme is to run just one program at a time, sharing the memory between that program and the operating system. This is called mono-programming model.



When the system is organized in this way, only one process at a time can be running. As soon as user types a command the OS copies the required program form disk to memory and execute. When the process finishes, the OS displays a prompt character and waits for a new command. When it receives the command it loads a new program into memory over-wring first one.

- Used in mainframes, palmtops and early DOS system.

Disadvantages:

- Only one processes can run at a time
- Processes can destroy OS (an erroneous process can crash the entire system)

4. Multiprogramming

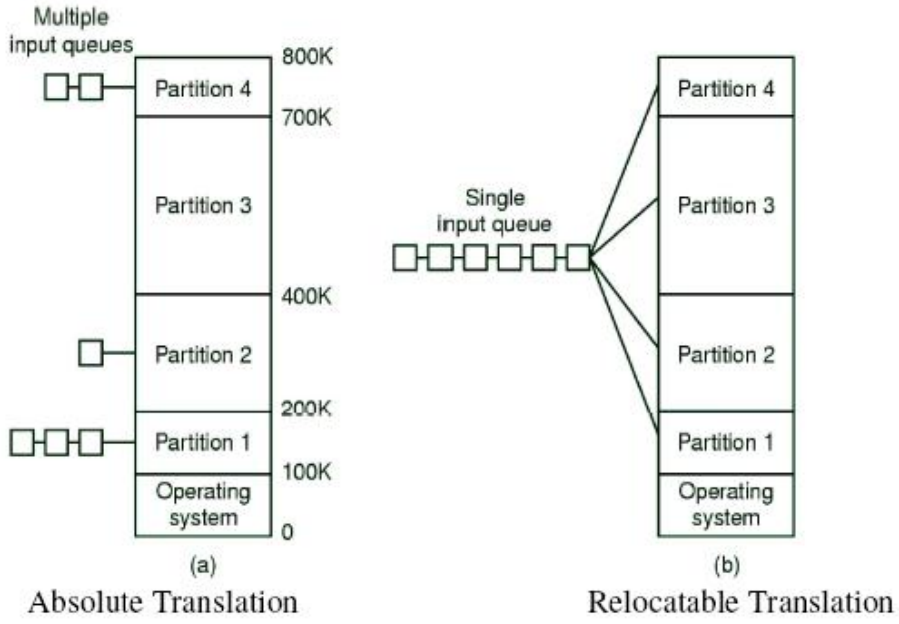
a. Fixed Partition Multiprogramming

Most modern system allows multiple processes to run at the same time. Having multiple process running at once mans that when one process is blocked waiting for input and output to finish, other one can use the CPU. Thus, multiprogramming increases CPU utilization.

In the multiprogramming model, multiple partitions are created to allow multiple user process to reside in memory simultaneously. The partitions are fixes possibly unequal.
When a job arrives, it can be put into the input queue for the smallest partition large enough to hold it. Since the partitions are fixed in this scheme, any space in a partition not used by a job is lost.

Implemented in two ways:

- Dedicated partition for each process (Absolute translation)
- Maintaining a single queue (Relocatable translation)



Absolute Transaction (Maintaining multiple input queues):

In this scheme, separate input queue is maintained for each partition. When a process arrives it is put into the input queue for the smallest partition large enough to hold it.

Problem:
If a process is ready to run and its partition is occupied then that process has to wait even if other partitions are available i.e. wastage of storage.

Relocatable Transaction (Maintaining a single queue):

Another strategy is to maintain a single queue for all partitions whenever a partition becomes free, the job closest to the front of the queue that fits in it could be loaded into the empty partition and run. Since it is undesirable to wastage a large partition on a small job, a different strategy is to search the whole input queue whenever a partition becomes free and pick the largest job that fits. But this algorithm discriminates against small jobs as being unworthy of having a whole partition.

Problem:

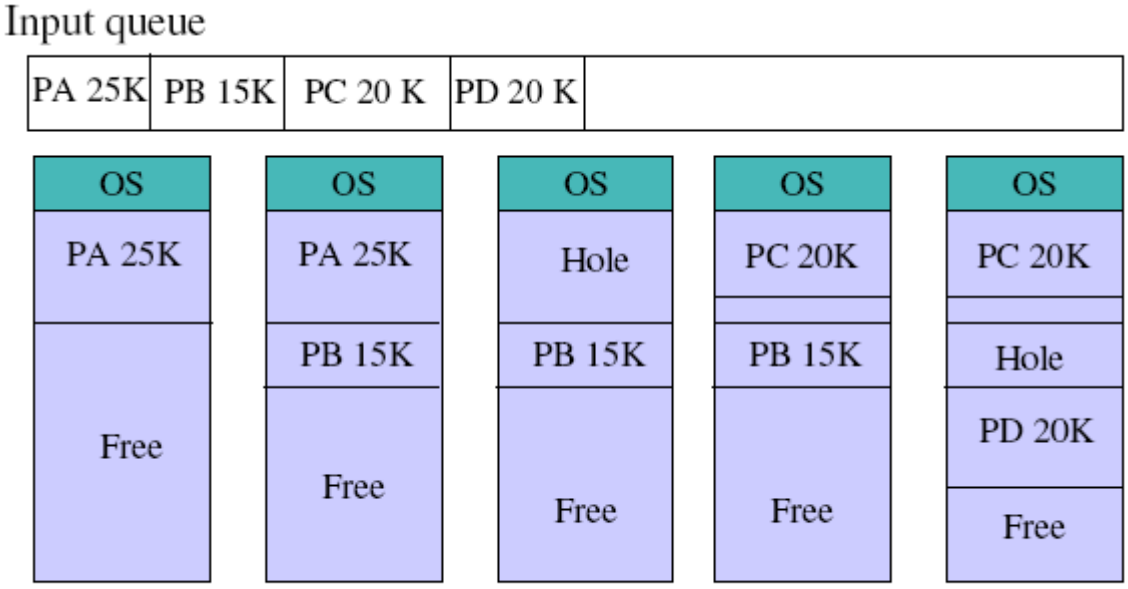
Eliminate the absolute problems but implementation is complex. Wastage of storage when many processes when many process are small.

b. Variable Partition Multiprogramming

In this scheme the numbers of partitions vary with time. When processes arrive, they are given as much storage as they need. When processes finish, they leave holes in main memory. OS fills these holes with another process from the input queue of processes.

Advantages: Process get the exact size partition of their request for memory—no wastage of memory.

Problem: when holes are given to another process they may again partitioned, the remaining holes get smaller eventually becoming too small to hold new processes—wastage of memory.

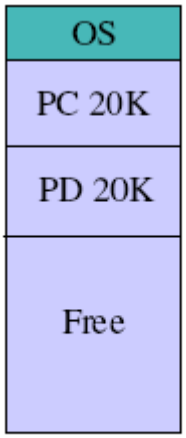


Memory allocation and holes

5. Memory Compaction:

When holes are created in memory, it is possible to combine them all into one big one by moving all the processes downwards as far as possible. This technique is known as memory compaction.

Example from above example:



Memory compaction

Problem: not possible in absolute transaction. If it is possible, it is highly expensive and, it requires a lots of CPU time.

Example: on a 256 MB machine that can copy 4 bytes in 40 nsec, it takes about 2.7 sec to compact all memory.

It stops everything when compaction. So this technique is not used.

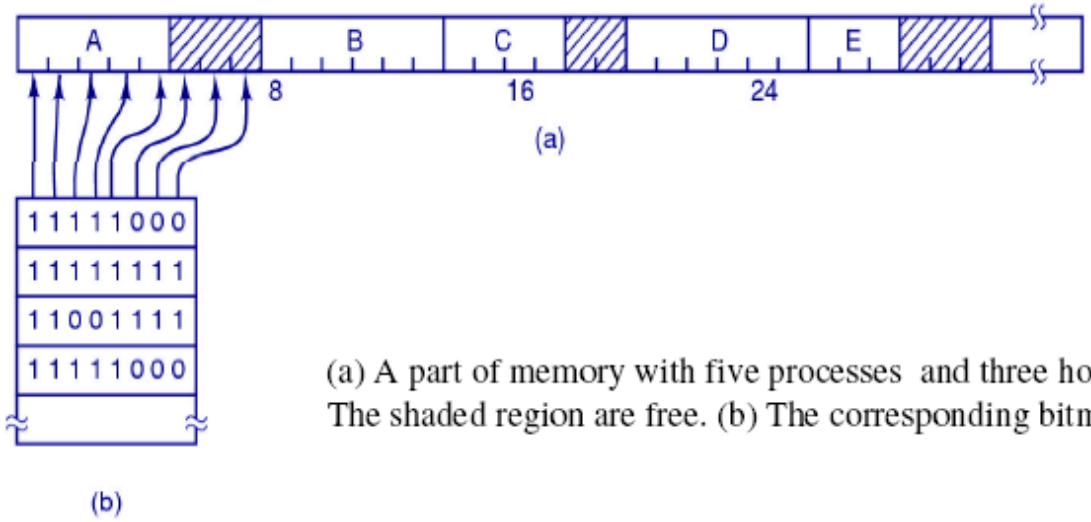
6. Memory Management with bit-map, linked list and buddy system:

When memory is assigned dynamically i.e. variable partition multiprogramming, the operating system must manage it. There are three ways too keep tack of memory usage:

- Bit-Map
- Free list
- Buddy system

6.1 Bit map management:

In bit map management, memory is divided up into allocated units, perhaps as small as a few words and perhaps as large as several kilobytes. Corresponding to each allocation unit is a bit in the bit map, which is 0 if the unit is free and 1 if it is occupied.



(a) A part of memory with five processes and three holes. The shaded region are free. (b) The corresponding bitmap.

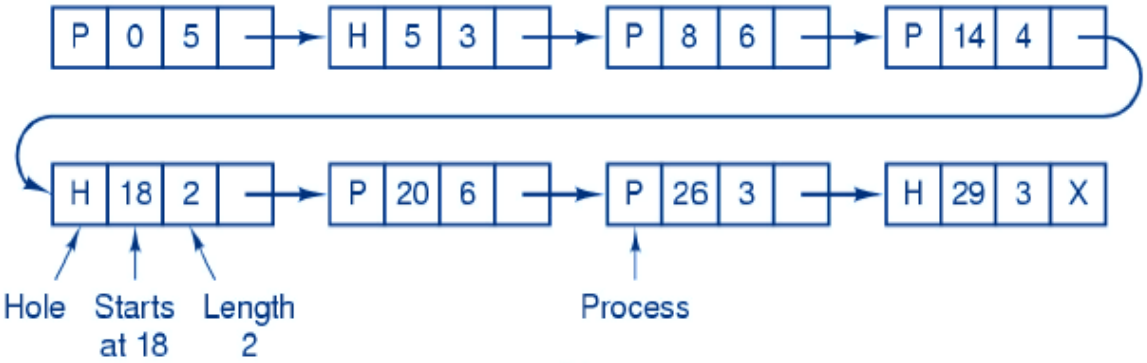
The size of allocation unit is an important design issue. The smaller, the allocation unit, the larger, the bitmap is.

Advantages: a bitmap provides a simple way to keep track of memory words in a fixed amount of memory because the size of the bitmap depends only on the size of memory and size of allocation unit.

Problem: searching a bitmap for a run of given length is a slow operation.

6.2. Linked List Management:

Another way of keeping track of memory is to maintain a linked list of allocated and free memory segments, where a segment is either a hole between two processes or a process.
Each segment in the entry contains an entry indicating the segment size and a pointer to the next segment in the list.



H represents hole and P represents process and the segment list is kept sorted by address. Sorting has an advantage that when a process terminates, updating the list is straightforward. It may be implemented as doubly linked list to make a more convenient search.
When a process terminates, if any neighbors is already hole, merge the holes—called coalescing.

3.6.3 Buddy System:

In a buddy system, memory blocks are available of size 2^k,
$L \le k \le U$, where

2^L - is smallest size block that is allocated.
2^U - is largest size block that is allocated. Generally, 2^U is size of entire memory available for allocation.

Suppose we have the entire block of size 2^U. A request of size s such that $2^{U-1} < s \le 2^u$ is made, then the entire block is allocated. Otherwise, the block is split into two equal buddies of size 2^{U-1}. If the block $2^{U-2} < s \le 2^{U-1}$, then the request is allocated to one of two buddies. Otherwise, one of the buddies is split into half again. This process is continued until the smallest block greater than or equal to s is generated and allocated to the request. At any time, the buddy system maintains a list if holes 2^i. A hole may be removed from the *(i + 1)* list by splitting it in half to create two buddies of the size 2^i in the *i* list. Whenever a pair of buddies on the *i* list both become unallocated, they are removed from that list and coalesced into a single block on the *(i+1)* list.
Example, consider 1- Mbyte initial block.

A request 100 K, B requests 240 K, C request 64 K, D request 256 K and E request 75 K. Show these in buddy system.
See class note.

3.7 Partition Selection Algorithms:

When the processes and holes are kept on a list sorted by address, several algorithms can be used to allocate memory for a newly created process (or an existing process being swapped in from disk).

1. First fit: the memory manager allocates the first hole that is big enough. It stops the searching as soon as it finds a free hole that is large enough. The hole is then broken up into two pieces, one for the process and one for unused memory.
   Advantages: it is a fast algorithm because it searches as little as possible.
   Disadvantages: not good in terms of storage utilization.

2. Next fit: it works the same way as first fit, except that it keeps track of where it is whenever it is whenever it finds a suitable hole. The next time it is called to find a hole, it starts searching the list from the place where it left off last time, instead of always at beginning as first fit does.

3. Best fit: Allocate the smallest hole that is big enough. Best fit searches the entire list and takes the smallest hole that is big enough to hold the new process. Best fit try to find a hole that is close to the actual size needed.
   Advantages: more storage utilization than first fit.
   Disadvantages: slower than first fit because it requires searching whole list at time.
   Creates a tinny hole that may not be used.

4. Worst fit: Allocate the largest hole. It search the entire list, and takes the largest hole, rather than creating a tinny hole it produces the largest leftover hole, which may be more useful.
   Advantages: some time it has more storage utilization than first fit and best fit.
   Disadvantages: not good for both performance and utilization.

7. Virtual memory

It is desirable to be able to execute a process whose logical address space is larger than the available physical address space. The programmer can make such a process executable by restructuring it using overlays, but doing so is generally a difficult programming task. Virtual memory is a technique to allow a large logical address space to be mapped onto a smaller physical memory. Virtual memory allows extremely large process to be run, and also allows the degree of multiprogramming to be raised, increasing CPU utilization. Further, it frees application programmers from worrying about memory availability.

Virtual storage is not a new concept; this concept was devised by fortherigham in 1961 and used in Atlas computer system. But the common use in OS is the recent concept, all microprocessor now support virtual memory. Virtual memory can be implemented by two most common used methods:

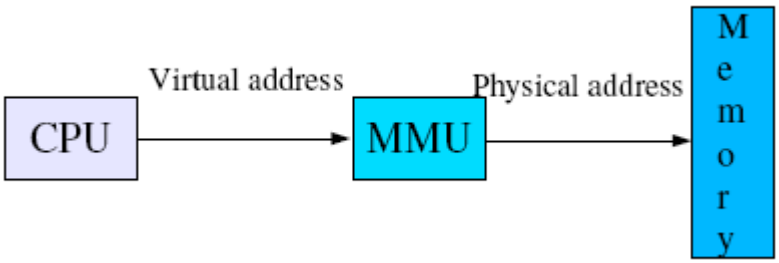
- Paging
- Segmentation

Or mixed strategy of both.

Advantages:

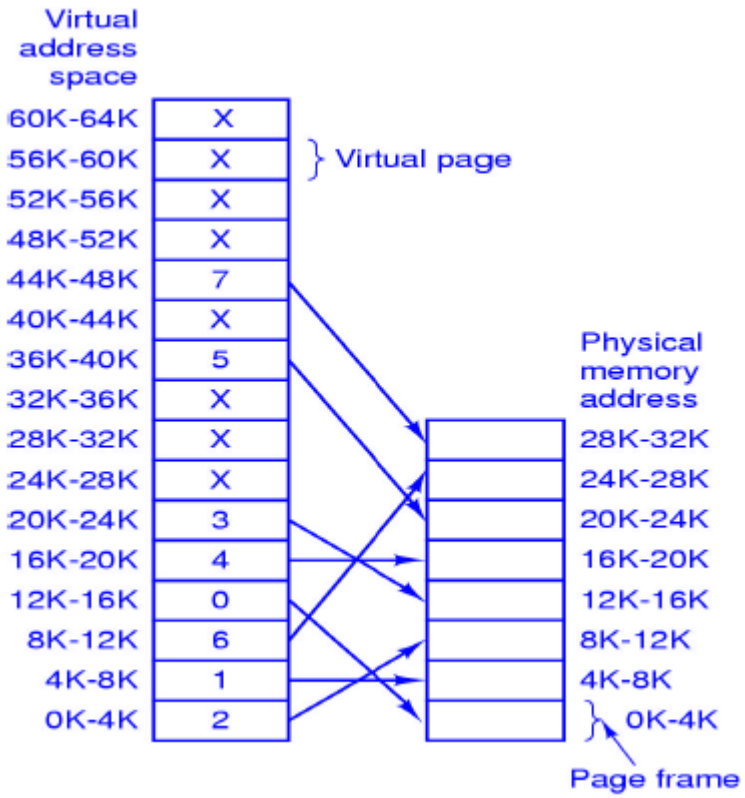
- Users would be able to write programs for an extremely large virtual address space, simplifying the programming task.
- More programs can be run simultaneously, increase in CPU utilization and throughput.
- Less input output would be needed to load or swap each user programs into memory.

MMU: the run time mapping from virtual address to physical address is done by hardware device called memory management unit (MMU).



8. Paging:

The virtual address space (process) is divided into fixed sized blocks called pages and the corresponding same size block in main memory in main memory is called frames. When a process is to be executed, its pages are loaded into any available memory frames from the backing store. The size of pages is determined by the hardware normally from 512 bytes to 64 KB.



In this example, we have a computer that can generate 16-bit addresses from 0 up to 64K. These are the virtual address. This computer however has only 32 KB of physical; so although 64 KB programs can be written, they cannot be loaded into memory in their entirely and run. A complete copy of a program's core image up to 64 KB must be present on the disk. However so that pieces can be brought in as needed.

The pages and page frames are always the same size. In this example, they are 4 KB with 64 KB of virtual address space and 32 KB of physical memory; we get 16 virtual pages and 8 page frames. Transfers between RAM and disks are always in units of page.
The following instruction like:

mov REG, 0

Virtual address 0 is send to MMU. The MMU sees that this virtual address falls in page 0 (0 to 4095), which according to its mapping is page frame 2 (8192 to 12287). Thus the address 0 is transferred to 8192 and output address is 8192.

9. Address Translation

The hardware support for paging (paging hardware) is illustrated in fig:

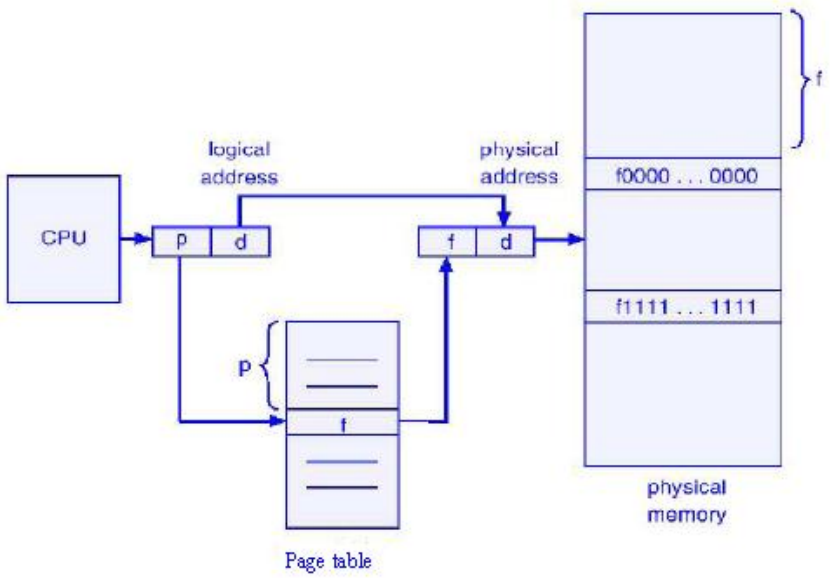


Fig: Paging Hardware

Every address generated by the CPU is divided into two parts: a page number (p) and a page offset (d). the page number is used as an index into a page table. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.

If the size of the logical address space is 2^m and page size is 2^n then the high order m-n bits of logical address designated page number, and n lower order bits designed the page offset.

10. Page Table

For each process, page table stores the number of frame, allocated for each page.

The purpose of the page table is to map virtual pages into page frames. Mathematically the page table is a function with the virtual page number as argument and the physical frame number as result. Using the result of this function the virtual page field in a virtual address can be replaced by a page frame field, thus forming a physical memory address.

i.e. this function of page table can be represented in notation as:

$$page\_frame = page\_table(page\_number)$$

The virtual page number is used as an index into the page table to find the corresponding page frame.

11. Page Table Structure

The exact layout of page table entry is highly machine dependent, but the common structure for 32-bit system is as:

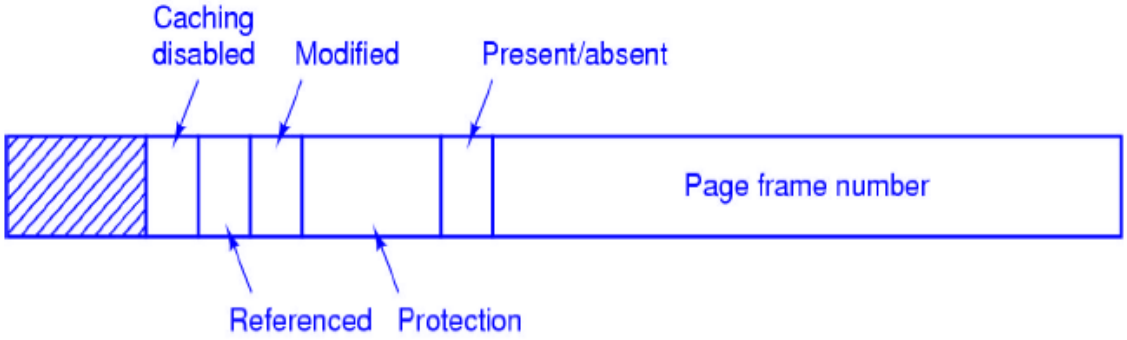


Fig: a typical page entry

Page frame number: the most important field is page frame number; after all, the goal of the page mapping is to locate this value.

Present/Absent bit: if present/absent bit is 1, the entry is valid and can be used (i.e. page is present in RAM). If it is 0, the trap is occurred called page fault.

Protection bit: This bit tells what kinds of access are permitted (i.e. read, write or read only)

Modified bit (Dirty bit): identifiers are changed status of the page since last access; if it is modified then it must be rewritten back to the disk.

Reference bit: it is set whenever a page is referenced, either for reading or writing. Its value is to help the OS choose a page to evict when a page fault occurs.

Cashing disable: it is used for that system where the mapping into device registers rather than memory.

Advantages of paging

- Fast to allocate and free.

Allocation: keep free list of free pages. Grab first page in the list.
Free: add pages to the free list.

- Easy to swap-out memory to disk.

Frame size matches disk page size.
Swap out only necessary pages.
Easy to swap in back from disk.

Disadvantages:

- Additional memory reference

Page table are kept in memory

- Internal fragmentation

Process size does not match allocation size.

12. Page replacement

When a page fault occurs, the operating system has to choose a page to remove from memory to make room for the page that has to be brought in. if the page to be removed has been modified while in memory, it must be rewritten to the disk to bring the disk copy up to date.

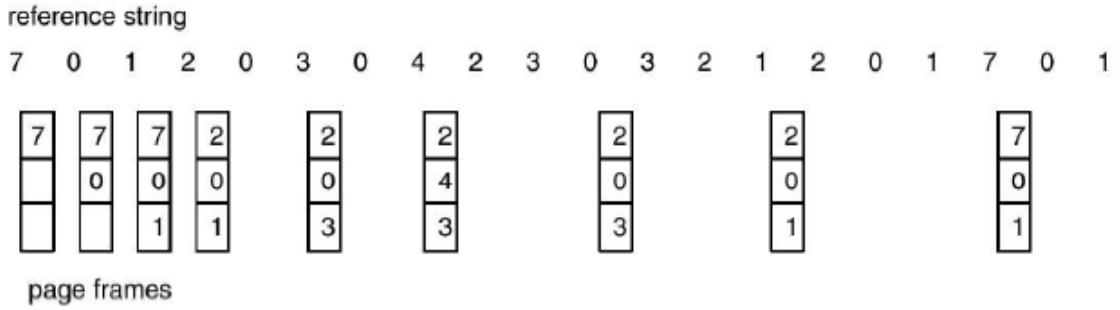
While it would be possible to pick a random page to evict at each page fault, system performances is much better if a page that is not heavily used is chosen. If a heavily used page is removed, it will probably have to be brought back in quickly, resulting in extra overhead. Much work has been done on the subject of page replacement algorithms, both theoretical and experimental.

Principle of optimality: *"To obtain optimal performance the page to replace is one that will not be used for the furthest time in future."*

1. Optimal Page Replacement:

The best possible page replacement algorithm is easy to describe but impossible to implement. OPR, replace the page that will not be used for the longest period of time.
The optimal page replacement algorithm simply says that the page with the highest label should be removed. If one page will not be used for 8 million instructions and another page will not be used for 6 million instructions, remove the former page.
It is used mainly for comparisons study.

Example: for 3 page frames and 8 pages system the optimal page replacement is as:



Advantages:

- An optimal page replacement algorithm, it guarantees the lower possible page fault rate.

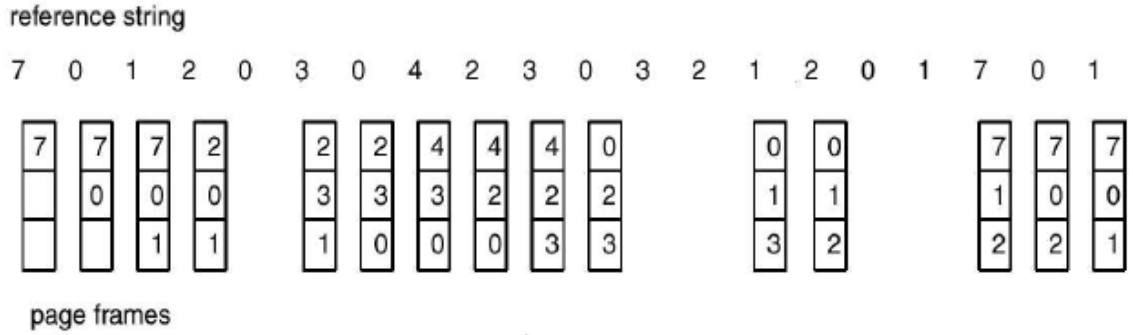
Problem:

- Unrealizable, at the time of the page fault, the OS has no way of knowing when each of the pages will be referenced next.
- Does not exist in real system.
- It is used mainly for comparisons study.

2. First-In-First-Out (FIFO) replacement:

The simplest page replacement algorithm is a FIFO algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When page must be replaced, the oldest page is chosen.
This can also be implemented by using queue of all pages in memory.

Example: for 3 page frames and 8 pages system the FIFO page replacement is as:



For our example, reference string, our three frames are initially empty. The first three reference (7, 0, 1) cause page faults, and are brought into the empty frames. The next reference (2) replaces page 7 because page 7 was brought in first. And so on.

Advantages:

- Easy to understand and program
- Distributes fair chance to all.

Problems:

- FIFO is likely to replace heavily (constantly) used pages and they are still needed for future processing.
- Performance is not always good.

3. Second chance page replacement algorithm

A simple modification to FIFO that avoids the problem of throwing out a heavily used page is to inspect the R (reference) bit of the oldest page. If it is 0, the page both old and unused, so it is replaced immediately. If the R bit is 1, the bit is cleared, the page is put onto the end of the list of pages, and its load time is updated as though it has just arrived in memory. Then the search continues.

Advantages:

- Big improvement over FIFO.

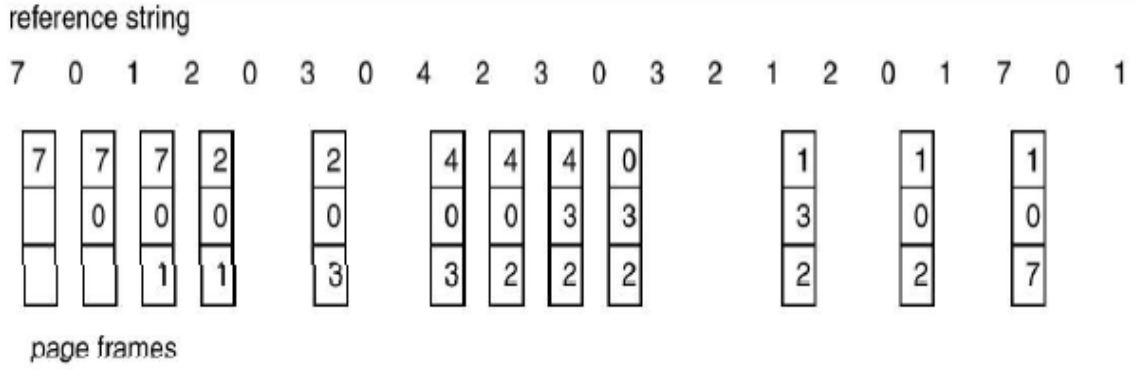
Problems:

- If all pages have been referenced, second chance degenerates into pure FIFO.

4. Least Recently Used (LRU)

"Recent past is a good indicator of the near future"
A good approximation to the optimal algorithm is based on the observation that pages that have been heavily used in the last few instruction will probably be heavily used again in the next future. Conversely, pages that have not been used for ages will probably remain unused for a long time. This idea suggests a realizable algorithm: when page fault occurs, throw out the page that has been unused for the longest time. This strategy is called LRU paging.

It maintains a linked list of all pages in memory with the most recently used page at the front and least recently used page at the rare. The list must be updated on every memory referenced.



Advantages:

- Excellent algorithm
- Efficient as much as optimal algorithm in some cases.

Problems:

- If it is implemented as linked list, updating list is every reference is not a way making system fast!
- This algorithm can be implemented in hardware level, it requires counter bit for each page to store how many times they are used.

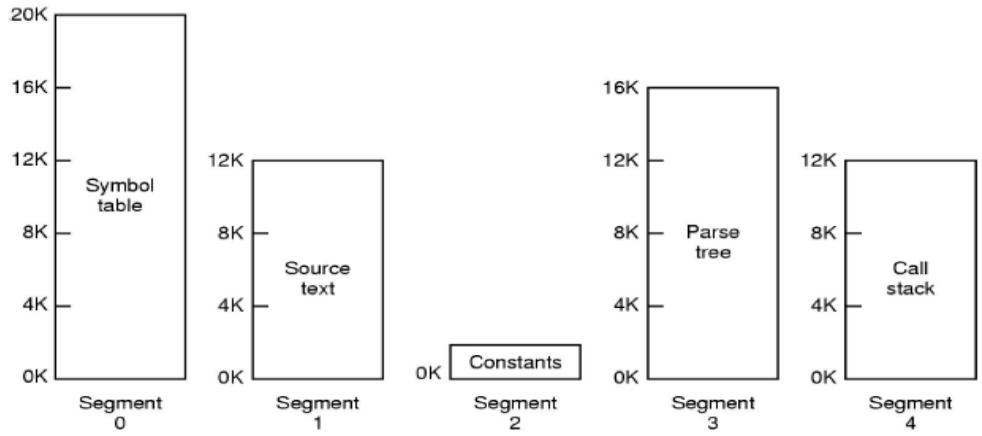
13. Segmentation

Memory management that support variable partitioning and mechanisms with freedom of contiguous memory requirement restriction is called segmentation. Each segment consists of a linear sequence of addresses from 0 to some maximum. The length of each segment may be anything from 0 to the maximum allowed. Different segments may, and usually do, have different lengths. Moreover, segment lengths may change during execution. The length of a stack segment may be increased whenever something is pushed onto the stack and decreased whenever something is popped off the stack. Because each segment constitutes a separate address space, different segments can grow or shrink independently, without affecting each other.

The independent block of the program is a segment such as main program, procedure, functions, methods, objects, local variables, global variable, common blocks, stacks, symbol table, and arrays.

The responsibility for dividing the program into segments lies with user.

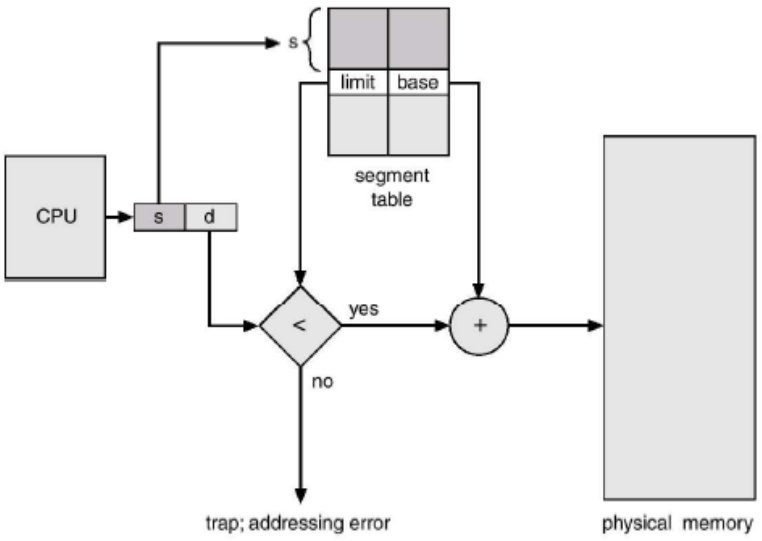
Figure illustrates a segmented memory being used for the compiler tables, five independent segments are shown below:



Segmentation helps in handling data structures that change size during execution and simplifies linking and sharing. It also facilitates providing different protection for different segments. Sometimes segmentation and paging are combined to provide a two-dimensional virtual memory. The MULTICS system and Intel Pentium support segmentation and paging.

Different segments have its own name and size. The different segment can grow and shrink independently, without affecting the others; so the size of segment changed during execution.
For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by segment name. Thus the logical address consists: segment number and offset.

The segment table (like page table but each entry consists limit and base register value) is used to map the logical address to physical address.



Each entry of the segment table has a segment base and segment limit. The segment base contains the starting physical address where the segment reside in memory. Whereas the segment limit specifies the length of the segment.

A logical address consists of two parts: a segment number s and an offset into that segment d. the segment number is used as an index into the segment table. The offset d of the logical address must be between 0 and the segment limit. If it is not, we trap to the OS. If this offset is legal, it is added to the segment base to produce the address in physical memory of the desired bytes. The segment table is thus essentially an array of base-limit register pairs.

Advantages of segmentation

1. It simplifies the handling of data structures that are growing or shrinking.
2. If each procedure occupies a separate segment, with address 0 as its starting address, the linking up of procedures compiled separately is greatly simplified.
3. Segmentation also facilitates sharing procedures or data between several processes.
4. Because each segment forms a logical entry of which the programmer is aware, such as a procedures or an array or stack, different segments can have different form of protection.

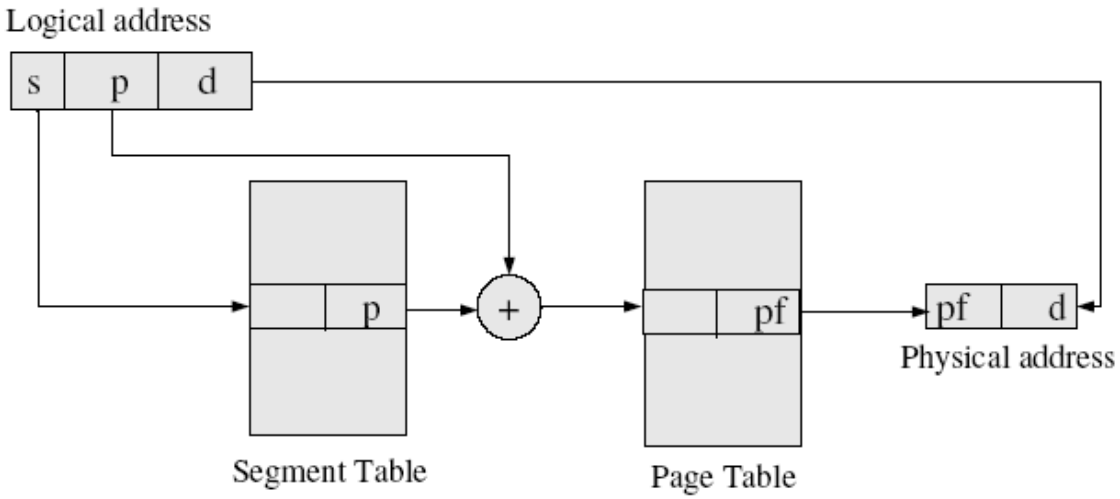
Problems:

The segmentation scheme causes fragmentation; this can be handled by same technique of variable partition memory management.

14. Paging vs. segmentation

Paging	Segmentation
1. Programmer need not be aware that this techniques is used.	Programmer should be aware that this technique is used
2. only one linear address space is present	Many linear address space is present
3. procedures and data can not be distinguished and separately protected	Procedures and data can be distinguished and separately protected
4. tables whose size fluctuates cannot be accommodated easily	The growing and shrinking tables can be handled easily
5 sharing of procedures between process is not fascinated	Procedures can be shared between many user process easily

Segmentation with Paging



If the segments are large, it may be inconvenient or even impossible to keep them in main memory in their entirely. This leads to the idea of paging them, so that only those pages that are actually needed have to be around. Several significant systems have supported paged segment.
Segmentation can be combined with paging to provide the efficiency of paging with the protection and sharing capabilities of segmentation.

As with simple segmentation, the logical address specifies the segment number and the offsets within the segment.

When paging is added, the segment offset is further divided into a page number and page offset. The segment table entry contains the address of the segment's page table.

Segmentation with Paging

Examples:

The Intel Pentium:

The Intel Pentium 80386 and later architecture uses segmentation with paging memory management.
The maximum number of segments per process is 16K, and each segment can be large as 4 GB 32-bit words. The page size is 4K fixed. It use two-level paging scheme.

Multics:

It has 256K independent segments, and each up to 64K 36-bit words. The page size is 1K words or small.

Questions:

1. How fragmentation occurs? Discuss the techniques that manage the fragmentation.

Sol: When a process request memory segment, if the process cannot find the exact size partition in the memory, the fragmentation occurs (in case, the process size is smaller than partition size). When fragmentation occurs, the memory partition is fragmented into small holes which are un usable by other process.

Example let us consider a multiple partition scheme with a hole of 18464 bytes. Suppose that the process requests 18462 bytes. If we allocate exactly the requested block, we are left with a hole of 2 bytes. The general approach is to broke the physical memory into fixed-sized blocks and allocate memory in units of block size. With this approach, the memory allocated to a process may be slightly larger than the requested memory. The difference between these two numbers is fragmentation.

There are four techniques that manage the fragmentation:

- Memory Compaction
- Bitmap management
- Linked list management
- Buddy system

Note: *Explain these four points with example if the question is long (for short question no need to explain the points but shortly you can describe)*

2. Why are page sizes always a power of 2?

Sol: the page size is defined by hardware. However it is typically maintained as power of 2, varying between 512 bytes to 64 KB (power of 2) per page. The selection of power of 2 makes the transaction of logical address into page number and page offset particularly easy. If the size of logical address space is 2^m and if page size is 2^n then high order m-n bits of logical m-n bits of logical address designate the page number and the n lower bits designate page offset.

3. How page fault occurs, Describe action taken by OS in such condition? Write down the function involved in page replacement.

Sol: A field in the PMT (Page Map Table) entry of page P_i indicates whenever the page currently exists in memory. If P_i does not exist in memory when referenced, address translation unit (ATU) raises a missing page interrupt, also called page fault. This transfers control to the virtual memory (VM) handler for necessary actions.

When a page fault occurs during to page P_i the OS finds a free page block in memory and loads P_i in it. This is called a ***page-in*** operation for P_i. if no free block exists in memory, some page P_k existing in the memory is written out onto a secondary memory to free its page block. This is a ***page-out*** operation for P_k.

Page replacement involves the following functions:

- Determine which page is to be removed from the memory.
- Perform a page-out operation

4. Perform a page-in operation.