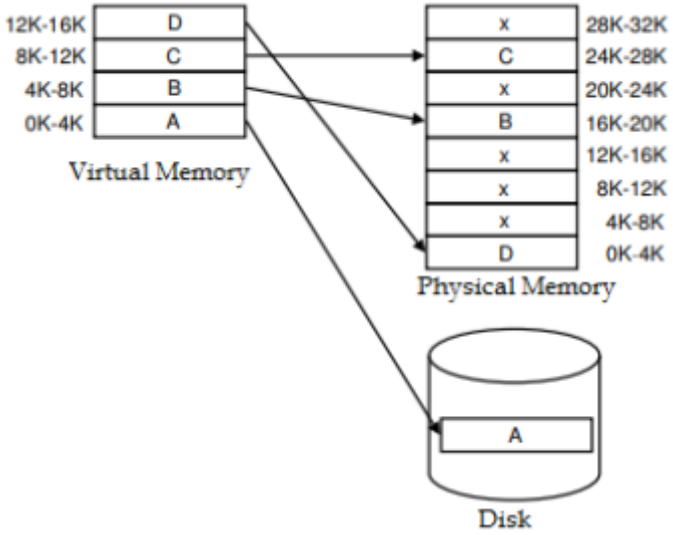
Chapter 3: Memory Management

PART III

II. Non-contiguous allowance

1. Virtual memory

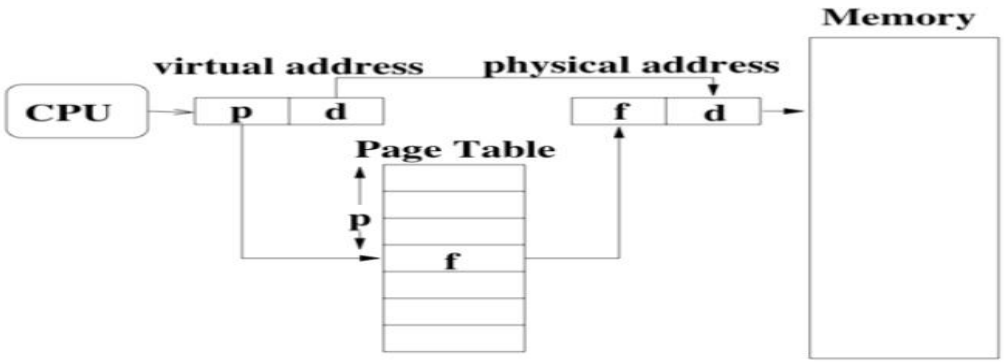
Virtual memory is a technique for running programs that exceed the size of the actual memory. The address space of a process, generated by compilers and linkers, is the virtual memory of the process or logical address space, as shown in the figure below. With proper techniques, as will be seen in this chapter, its size can be much larger than that of physical or real memory.



Example (see figure above) Virtual Address Space and Physical Space: The process in its continuous virtual space consists of four pages: A, B, C and D. Three blocks are located in the physical memory and one is on the disk. It would be too expensive to assign any process a full address space, especially because many use only a small portion of its address space. In general, virtual memory and physical memory are structured in allocation units (pages for virtual memory and frames for physical memory). The size of a page is equal to that of a frame. When a process is running, only part of its address space is in memory. The virtual addresses referenced by the current statement must be translated into physical addresses. This address conversion is performed by hardware management circuits. If this address corresponds to an address in physical memory, the actual address is transmitted to the bus, Otherwise a **page fault** occurs. For example, if the physical memory is 32 KB and the addressing is 16-bit, the logical address space can reach size 216 or 64 KB. The address space is structured into a set of units called **pages** or **segments**, which can be loaded separately into memory. All of its address space (virtual memory) is stored on disk. To run a process, the operating system loads into memory only one or a few units (pages or segments) including the one that contains the beginning of the program. When a process is running, only part of its address space is in main memory. This part is said to be resident. Parts of this space are loaded into main memory on demand. They can be dispersed in central memory.

2. Pure pagination

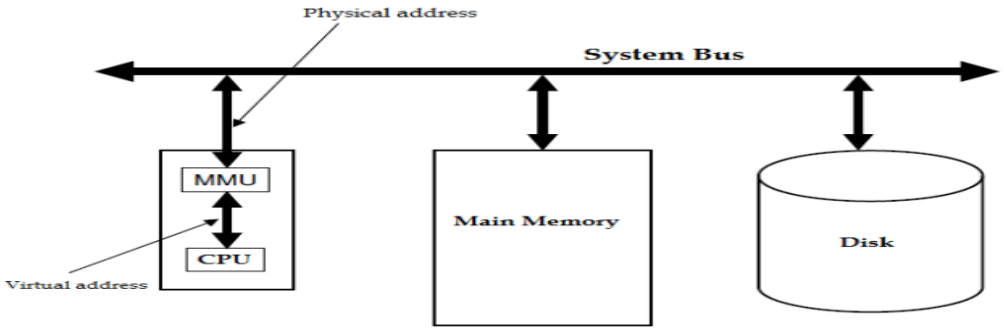
Virtual memory and physical memory are structured into allocation units called **pages** for virtual memory and boxes or **frames** for physical memory. The size of a page is fixed and equal to that of a frame. It varies between 2 KB and 16 KB. 4 KB is a fairly common typical value. A diagram of address translation during **pure pagination** is shown in the figure below. In pagination there is no external fragmentation because all pages are the same size. On the other hand, there can be internal fragmentation if the last page of the logical address space is not full.



3. Memory Management Unit (MMU)

Virtual addresses, generated by compilers and linkers, are pairs consisting of a page number and a relative displacement at the beginning of the page. The virtual addresses referenced by the run-time statement must be converted to physical addresses. This address conversion is performed by the MMU, which is a set of management hardware circuits.

We will consider the transfer unit page. The MMU checks whether the received virtual address matches an address in physical memory, as shown in the figure below. If this is the case, the MMU transmits the actual address on the memory bus, otherwise a page fault occurs. A page fault causes a diversion (or TRAP) whose role is to bring back from disk the missing referenced page. The correspondence between pages and boxes is stored in a table called **Table of Pages (PT).** The number of entries in this table is equal to the number of virtual pages. The **Page Table of** a process must be in whole or in part in central memory when the process is executed. It is necessary for converting virtual addresses to physical addresses.

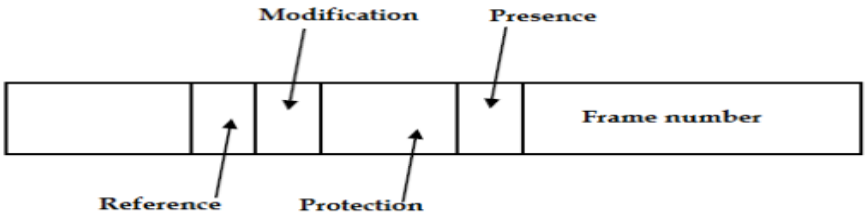


4. Structure of the Page Table

Each entry in the **Page Table** consists of several fields, including:

1. The presence bit.
2. The reference bit (R).
3. Protection bits.
4. The modification bit (M).
5. The frame number corresponding to the page.

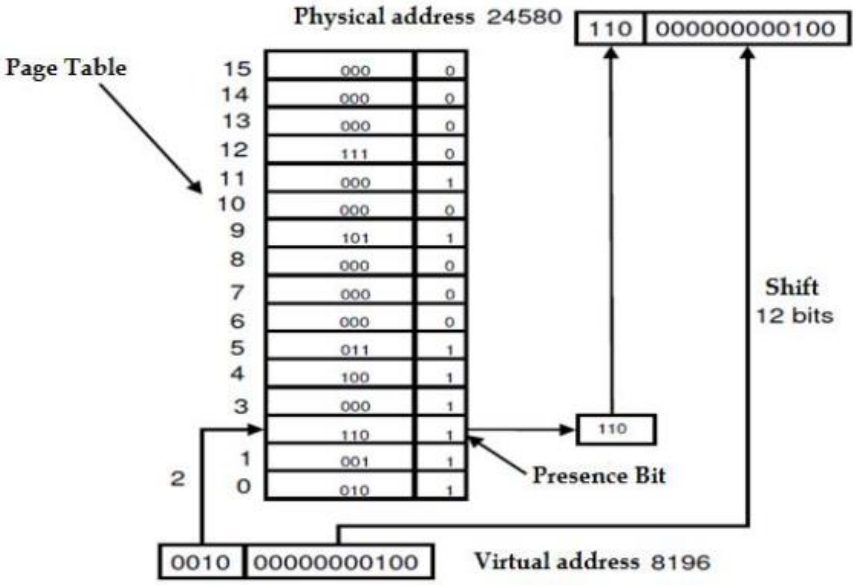
The structure of a typical page table, although machine-dependent, is shown in the figure below. The size varies from machine to machine, but 32-bit is a common size.



5. How an MMU works

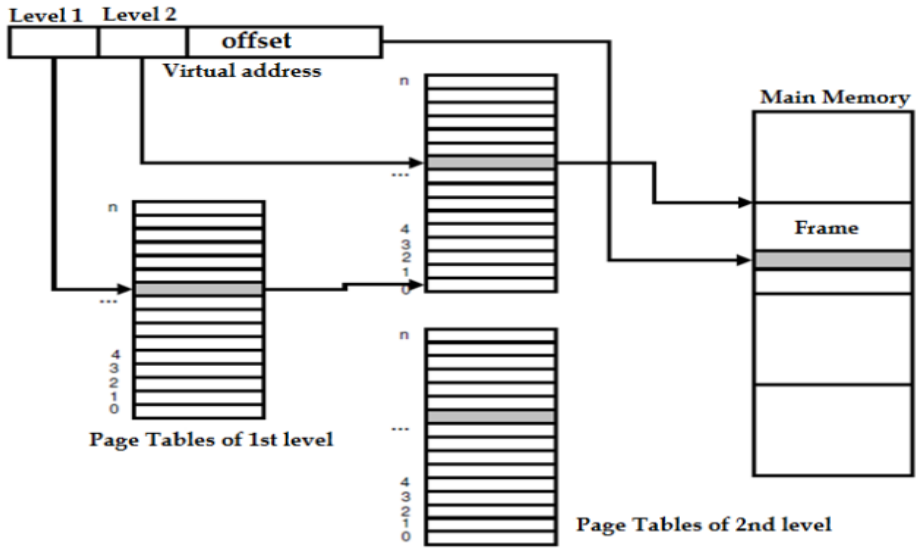
The MMU receives, as input, a virtual address and sends the physical address as an output or causes a diversion. In the example figure below, the virtual address is 16-bit encoded. The 4 bits of high weight indicate the page number, between 0 and 15. The other bits give the offset in the page, between 0 and 4095. The MMU examines the entry in the **Page Table** corresponding to the page number, in this case 2. If the presence bit is 0, the page is not in memory so the MMU causes a diversion.

Otherwise, it determines the physical address by copying in the 3 bits of highest weight the frame number (110) corresponding to the page number (0010), and in the 12 bits of lowest weight of the virtual address. The virtual address 8196 (0010 0000 0000 0100) is then converted to physical address 24580 (1100 0000 0000 0100) as shown in the following figure. The operating system maintains a copy of the process page table, which is used to perform address translation.



6. Multi-level page table

The size of the page table can be very large: for example, we would have more than 1 million entries (220) for 32-bit virtual addressing and 4 KB pages. To avoid having tables that are too large in memory, many computers use multi-level page tables. A two-level page table scheme is shown in the following figure.



For example, a two-level page table, for 32-bit addressing and 4 KB pages, consists of 1024 (1 KB) tables. This makes it possible to load only the necessary 1 KB tables. In this case, a 32-bit virtual address is composed of three fields: a pointer to the 1st level table, a pointer to a 2nd level table, and a 12-bit move in the page.

7. Access to the page table

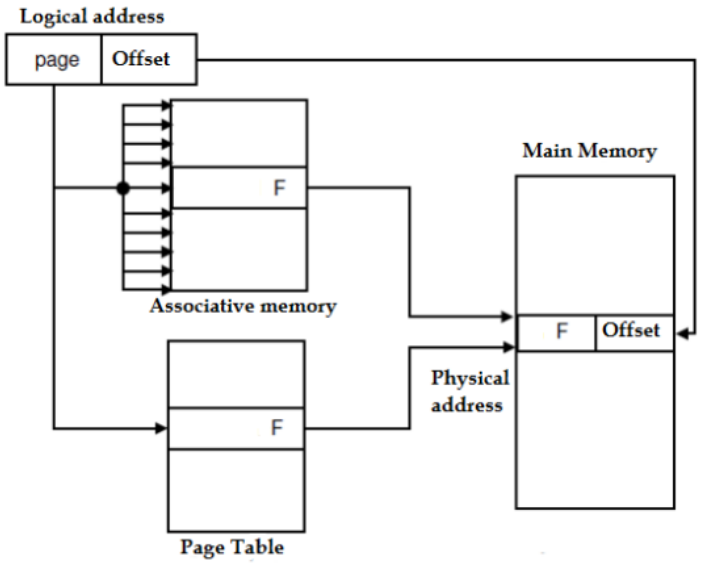
Access to the page table (which is in memory) can be done via an MMU register that points to the page table. This is a slow solution because it requires memory access. When a process moves to the elected state, the address of the page table is loaded into the registry. It has been found that paging can have a big impact on system performance.

8. MMU with associative memory

If the page table is large, the previous solution is not feasible. In order to speed up address translation, the MMU can be equipped with a table of fast machine registers indexed by means of the Virtual page. When a process enters the Chosen state, the operating system loads the process's **Page** Table into the **Register Table** from a copy in central memory. The solution is to equip the MMU with a device called **associative memory**, which is composed of a small number of inputs, normally between 8 and 64. It has been observed that most programs tend to make large references to a small number of pages. Associative memory contains special circuits for accessing addresses that are highly referenced. Associative memory is also called **TLB** Translation Lookside Buffer. This component contains information about the most recently referenced pages. Each entry in the **TLB** is composed of:

1. One bit of validity.
2. A virtual page number.
3. A modification bit (M).
4. Two bits of protection.
5. A frame number.

Address translation using associative memory is shown in the following figure. When a virtual address is presented to the MMU



It first checks whether the number of the virtual page is present in the associative memory, comparing it simultaneously (in parallel) to all entries. If it finds it and the access mode conforms to the protection bits, the box is taken directly from the associative memory (without going through the Table of pages). If the page number is present in the associative memory but the access mode is noncompliant, a protection defect occurs. If the page number is not in the associative memory, the MMU accesses the Page Table at the entry corresponding to the page number. If the presence bit of the found input is 1, the MMU replaces one of the entries in the associative memory with the found input. Otherwise, it causes a page fault.

9. Number of frames allocated to a process

The same number of memory frames can be allocated to each process. For example, if the total memory is 100 pages and there are five processes, each process will receive 20 pages. Frames can also be allocated in proportion to program sizes. If one process is twice as large as another, it will receive double the number of frames. The allocation of frames can be done during loading or on demand during execution.

10. Page replacement algorithms

As a result of a page fault, the operating system must bring the missing page from disk back into memory. If there are no free frames in memory, it must remove a page from memory to replace it by the requested one. If the page to be checked out has been modified since it was loaded into memory, it must be written back to disk. Which page should be removed in order to minimize the number of page faullts?

The choice of the page to replace may be limited to:

- the process pages that caused the page fault (**local allocation)**
- or to all pages in memory (**global allocation).**

In general, the overall allocation produces better results than the local allocation. Page replacement algorithms have been proposed. These algorithms memorize past references to pages. The choice of which page to remove depends on past references. There are several page replacement algorithms: the random replacement algorithm that randomly chooses the victim page (which is only used for comparisons), the optimal algorithm, the FIFO algorithm, the clock algorithm, and the least recently used page algorithm, and many others.

10.1 FIFO Page Replacement

It memorizes in a FIFO discipline queue (first in, first out) the pages present in memory. When a page defect occurs, it removes the oldest, i.e. the one at the front of the line.

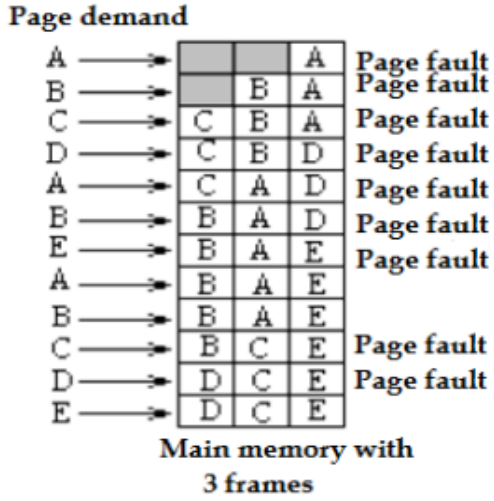
Example

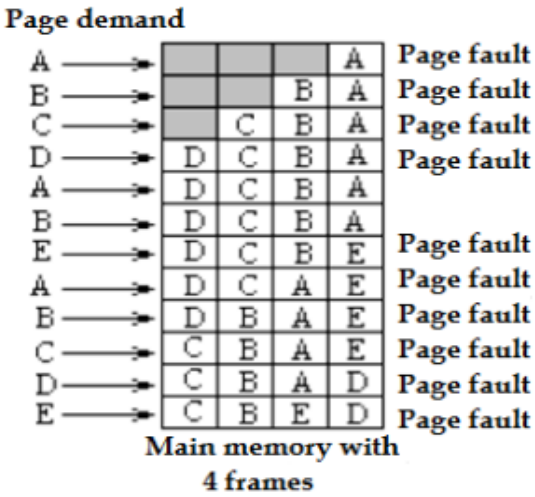
The sequence of references w ={7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1}
With m=3 frames, makes 15 page faults with the FIFO algorithm, as shown in the following table:

m \ ω	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
0	**7**	7	7	**2**	2	2	2	**4**	4	4	**0**	0	0	0	0	0	0	**7**	7	7
1		**0**	0	0	0	**3**	3	3	**2**	2	2	2	2	**1**	1	1	1	1	**0**	0
2			**1**	1	1	1	**0**	0	0	**3**	3	3	3	3	**2**	2	2	2	2	**1**

This algorithm does not take into account the use of each page. For example, at the tenth reference page 0 is removed to be replaced by page 3 and then immediately after the removed page is reloaded. The algorithm is rarely used because there are a lot of page faults.

Belady anomaly: this anomaly, studied by Belady, Nelson, Shedler, is specific to the FIFO strategy. Intuition leads us to think that the more space there is in main memory, the fewer page defects. In reality (this is the anomaly), this is sometimes false. The example below explains this paradox: in a 3- frame memory, there are 9 page faults; In a memory with 4 frames (therefore larger), we meet for the same treatment 10 page faults.





10.2 The optimal page replacement algorithm:

Belady's optimal algorithm is to remove the page that will be referenced as late as possible in the future. This strategy is impossible to implement because it is difficult to predict the future references of a program. Optimal page replacement, however, was used as a baseline for other strategies, as it minimizes the number of page faults.

Example: Let be a system with 3 memory frames and a sequence of references: w ={7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1}. See the following figure:

m \ ω	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
0	**7**	7	7	**2**	2	2	2	2	2	2	2	2	2	2	2	2	2	**7**	7	7
1		**0**	0	0	0	0	0	**4**	4	4	**0**	0	0	0	0	0	0	0	0	0
2			**1**	1	1	**3**	3	3	3	3	3	3	3	**1**	1	1	1	1	1	1

The optimal algorithm therefore makes 9 page faults.

10.3 LRU Least Recently Used Page

The LRU algorithm stores all the pages in memory in a linked list. The most used page is at the top of the list and the least used is at the bottom. When a page fault occurs, the least used page is removed. To minimize searching and editing the linked list, these operations can be performed by the hardware. This algorithm is expensive.

Example

The sequence of references w={7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1}

With m=3 frames, makes 12 page faults with the least recently used page replacement algorithm, as shown in the following table:

m \ ω	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
0	**7**	7	7	**2**	2	2	2	**4**	4	4	**0**	0	0	1	1	1	1	**1**	1	1
1		**0**	0	0	0	0	0	**0**	0	**3**	3	3	3	3	3	**0**	0	0	0	0
2			**1**	1	1	**3**	3	3	**2**	2	2	2	2	2	2	2	2	**7**	7	7

The problem with this algorithm is the difficulty of implementation, which requires hardware support. You need a way to memorize the time each time a page is referenced. We can also use an aging technique, where a register of n bits is associated with each page. The most significant bit is set to 1 each time the page is referenced. Regularly, the bits of this register are shifted to the right. When you have to expel a page, you choose the one with the smallest value. We could also put a page on top of a stack each time it is referenced. We will expel the page at the bottom of the pile.

a. LRU algorithm with mask method:

It is a method of implementing the LRU replacement algorithm according to the mask method.

1. Each page is associated with one byte.
2. The initial value of the mask is one byte (00000000). The high weight bit is set to 1 each time this page is used.
3. At each period (N ms), we make a shift to the right of the byte associated with each page.
4. The victim page is the page that has the smallest mask value at the time of the algorithm launch.

```
LRU algorithm;

Var  find: boolean;
Min: integer;
Begin
      Min← byte-min(number-page-load-MM);
      found ← Unique-test(min);
If (found) then {page is unique}
      Num-page-victim ← Selection(min,list-page-load);
else {the page is not unique}
     Num-page-victim ← Selection(min,list-page-load, FIFO);
End.
Where: list-page-load: is the list of pages loaded in MM.
```

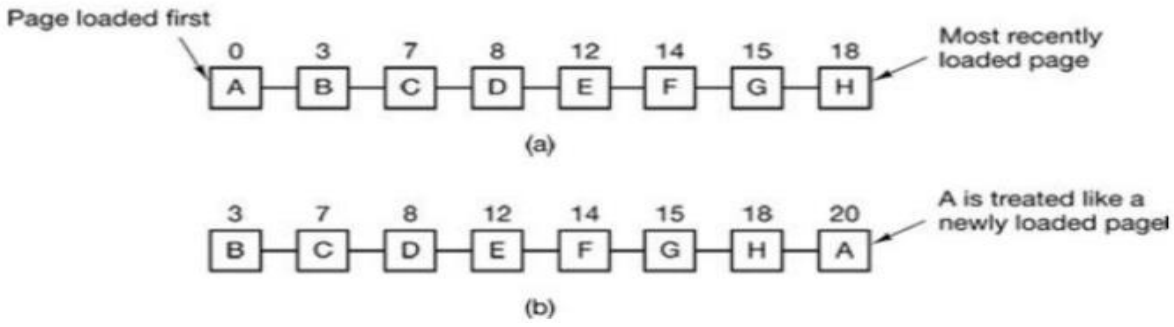
b. Comparison between three methods of LRU implantation

Stack method	Counter method	Mask method
It is difficult to manage	it is easy to update	it is easy but requires a shift (update)
Stack size increases	the number of counters increases with the size of the central memory	by increasing the size of the main memory we will increase the number of the required bytes

10.4 The Second Chance Page Replacement Algorithm

A simple modification can be made to the FIFO algorithm to avoid deleting a commonly used page: inspecting the R bit of the oldest page. If it is set to 0, the page is both old and unused, so it is replaced immediately. If the R bit is at 1, the bit is cleared, the page is placed at the end of the page list, and its loading time is updated as if it had just arrived in memory. The research then continues. The operation of this algorithm, called second chance, is illustrated below we see that the pages from A to H are in a linked list, sorted by their arrival time in memory.

Suppose a page fault occurs at time 20. The oldest page is A, which arrived at time 0, when the process started. If the R bit of page A is at 0, it is ousted from memory: it is either written to disk (if it is modified) or simply abandoned (if it has remained the same as the page on disk). On the other hand, if the R bit is at 1, A is placed at the end of the list and its loading time is updated with the current time (20). The R bit is also set to 0. The search for a suitable page then continues with B.



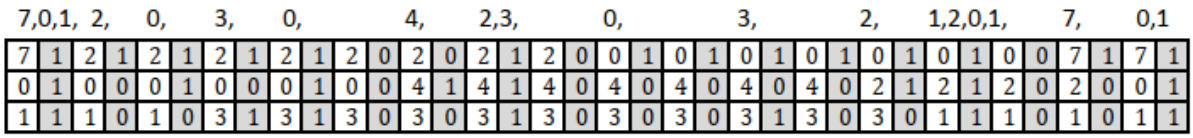
Second chance operation:

(a) Pages sorted in FIFO order.

(b) List of pages when a page fault occurs at time 20 and the R bit of A is at 1.

The numbers above the pages correspond to the time they load. The job of the second chance algorithm is to look for an old page that was not referenced in the previous clock interval. If all pages have been referenced, the second chance algorithm degenerates into a pure FIFO algorithm. To be concrete, imagine that all the pages in the previous figure a have their bit R at 1. The operating system moves eventually, it returns to page A, whose R bit is now at 0. At that point, A is ousted. Thus, this algorithm terminates always. One by one the pages at the end of the list, clearing the R bit each time it adds a page to the end of the list.

Example: The sequence of references w ={7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1} With m=3 frames, makes 11 page faults.

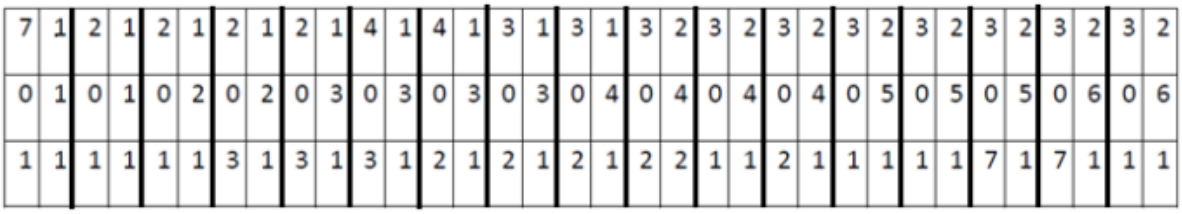


10.5 Replacement of the least frequently used page: LFU(Least frequently used)

We keep a counter that is incremented each time the frame is referenced, and the victim will be the frame with the lowest meter.

Example:

Reference chain: 7-0-1-2-0-3-0-4-2-3-0-3-2-1-2-0-1-7-0-1**.**



There are 13 page faults.

11. Operating System crash

If the system spends more time dealing with page faults than running processes there may be problems with the system collapsing. If the number of processes is too large, the space specific to each will be insufficient; they will then spend their time dealing with page faults. This is called **system crash**. The risk of crash can be reduced by monitoring the number of page faults caused by a process.

If a process causes too many page faults (above an upper limit) it will be allocated more pages; below a lower limit, it will be removed. If there are no more pages available and too many page faults, one of the processes will have to be suspended.

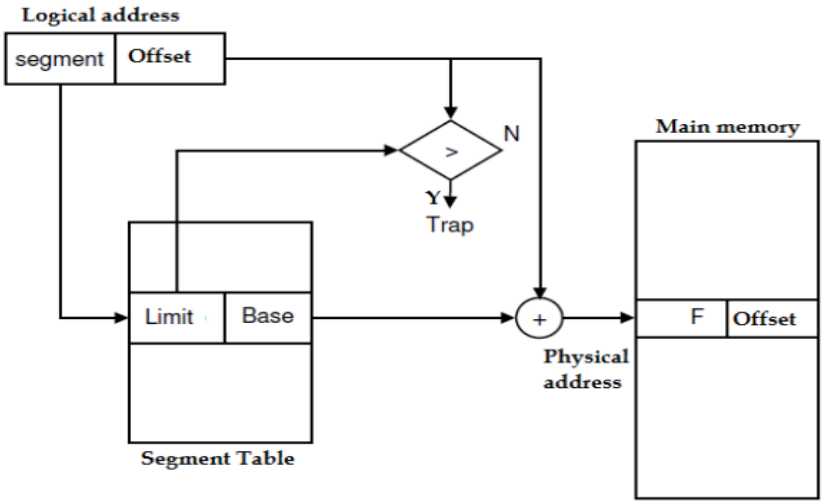
12. Segmentation

In a paginated system, the virtual address space of a process is one-dimensional. In general, a process is composed of a set of logical units:

The different codes: the main program, the procedures, the library functions.

. Initialized data. . Uninitialized data. . Execution stacks.

The idea of the **segmentation** technique is to have a two-dimensional address space. Each logical unit can be associated with an address space called a **segment**. The address space of a process is composed of a set of segments. These segments are of different sizes (external fragmentation). A segmentation address translation scheme is shown in the figure below. Segmentation makes it easy to edit links, as well as share data segments or codes between processes.

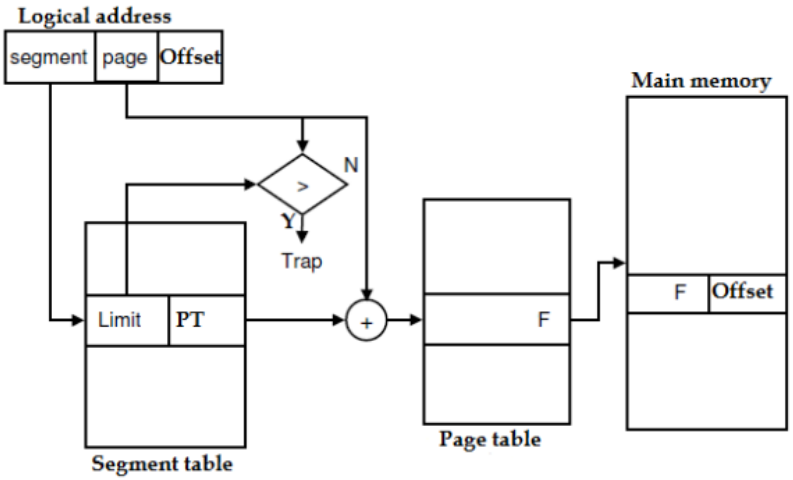


13. Paged segmentation

Segmentation can be combined with pagination. Each segment is composed of a set of pages. The addresses generated by compilers and linkers, in this case, are then triples:

<segment number, page number, Offset>

The paginated segmentation address translation scheme is shown in the figure below.



14. Cache memory

Cache memory is a very short access time memory (10 ns). It costs more. The main memory access time is 100 ns. The cache is placed between the processor and the central memory. In a paging system, when a virtual address is referenced, the system examines whether the page is present in the cache. If so, the virtual address is converted to a physical address. Otherwise, the system locates the page and then fills it back into the cache. The purpose of the cache is to minimize the average access time t.

t = access time + failure rate *failure processing time.