UNIT 6. MEMORY MANAGEMENT

6.1 BACKGROUND

It has been discussed earlier that CPU scheduling is used to increase the utilization of CPU and to improve computer's response to the user. For this, several processes must be kept in the memory. Here we will discuss how memory is managed to achieve this.

Memory consists of a large array of words or bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter. These instructions may cause additional loading/storing from/to specific memory addresses. Due to several reasons, we can ignore *how* a memory address is generated by a program. But, we have to consider only the sequence of memory addresses generated by the running program.

6.1.1 Address Binding

Usually, a program resides on a disk as a binary executable file. The program must be brought into memory and placed within a process for it to be executed.

Most systems allow a user process to reside in any part of the physical memory. Thus, although the address space of the computer starts at 00000, the first address of the user process need not be 00000. This arrangement affects the addresses that the user program can use. In most cases, a user program will go through several steps-some of which may be optional-before being executed as shown in Figure 6.1. Addresses may be represented in different ways during these steps. Addresses in the source program are generally symbolic such as count. A compiler will typically bind these symbolic addresses to relocatable addresses (such as "14 bytes from the beginning of this module"). The linkage editor or loader will in turn bind these relocatable addresses to absolute addresses (such as 74014). Each binding is a mapping from one address space to another.

The binding of instructions and data to memory addresses can be done in following ways:

- **Compile Time:** If we know at compile time where the process will reside in memory, then ***absolute code*** can be generated. After sometime, if this location changes, then it is necessary to recompile this code.
- **Load Time**: If it is not known at compile time where the process will reside in memory, then the compiler must generate ***relocatable code***. In this case, final binding is delayed until load time.
- **Execution Time:** If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this scheme to work.

6.1.2 Logical v/s Physical Address Space

An address generated by the CPU is commonly referred to as a logical address, whereas an address seen by the memory unit-that is, the one loaded into the memory-address register of the memory-is referred to as a physical address. The compile-time and load-time address-binding methods generate identical logical and physical addresses. However, the

execution-time address binding scheme results in different logical and physical addresses. In this case, we usually refer to the logical address as a ***virtual address***.

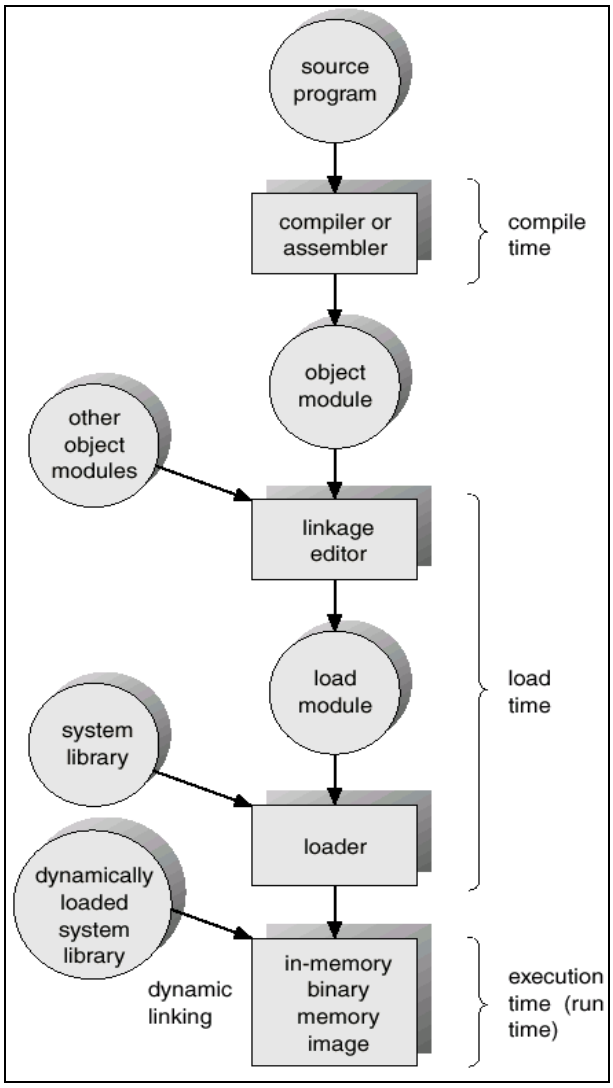


Figure 6.1 Multistep processing of a user program

The set of all logical addresses generated by a program is a logical-address space; the set of all physical addresses corresponding to these logical addresses is a physical-address space. The run-time mapping from virtual to physical addresses is done by a hardware device called the ***memory-management unit (MMU)***.

The value in the base register (also called a ***relocation register***) is added to every address generated by a user process at the time it is sent to memory. For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346 (Figure 6.2). The user program never sees the real physical addresses. The program can create a pointer to location 346, store it in memory, manipulate it, compare it to other addresses-all as the number 346. Only when it is used as a memory address (in an indirect load or store,

perhaps) is it relocated relative to the base register. The user program deals with logical addresses. The memory-mapping hardware converts logical addresses into physical addresses. The final location of a referenced memory address is not determined until the reference is made. We now have two different types of addresses: logical addresses (in the range 0 to *max*) and physical addresses (in the range R + 0 to R + *max* for a base value R). The user generates only logical addresses and thinks that the process runs in locations 0 to max.

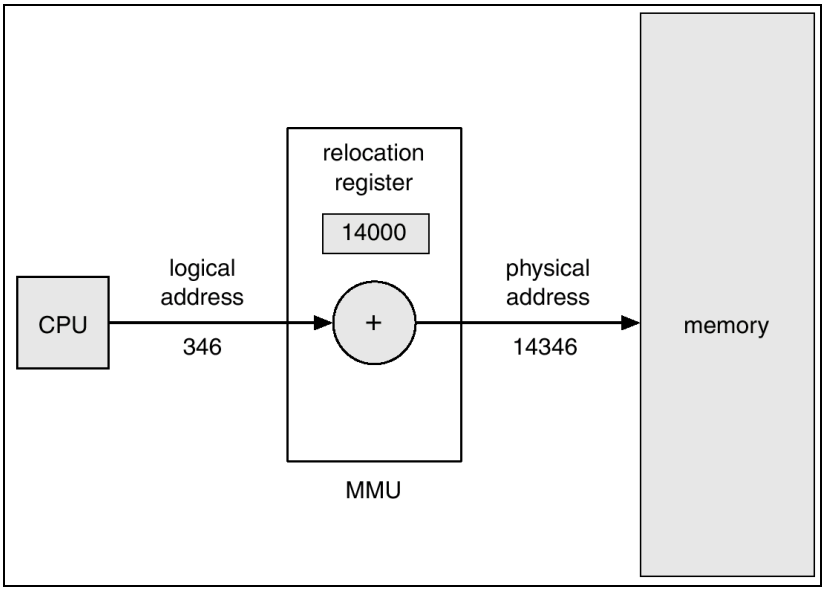


Figure 6.2 Dynamic relocation using a relocation register

6.1.3 Dynamic Loading

We have discussed earlier that the entire program and data of a process must be in physical memory for the process to execute. The size of a process is limited to the size of physical memory. To obtain better memory-space utilization, we can use dynamic loading. With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a relocatable load format. The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded. If not, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change. Then, control is passed to the newly loaded routine.

The advantages of dynamic loading:

- an unused routine is never loaded. Hence, though the total program size is large, the portion used will be much smaller.
- it does not require special support from the operating system

6.1.4 Dynamic Linking and Shared Libraries

Some OS supports static linking, in which language libraries are combined by the loader into the binary program image. The dynamic linking is similar to dynamic loading. Here, linking is postponed until execution time. Small piece of code called ***stub*** is used to locate

the appropriate memory-resident library routine. When this stub is executed, it checks to see whether the needed routine is already in memory. If not, the program loads the routine into memory. Thus, the next time that that code segment is reached, the library routine is executed directly, incurring no cost for dynamic linking. Under this scheme, all processes that use a language library execute only one copy of the library code.

A shared library is a file that is intended to be shared by executable files and further shared object files. Modules used by a program are loaded from individual shared objects into memory at load time or run time, rather than being copied by a linker.

6.2 SWAPPING

A process needs to 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 as shown in Figure 6.3. Such swapping may be necessary in priority based scheduling or round-robin scheduling.

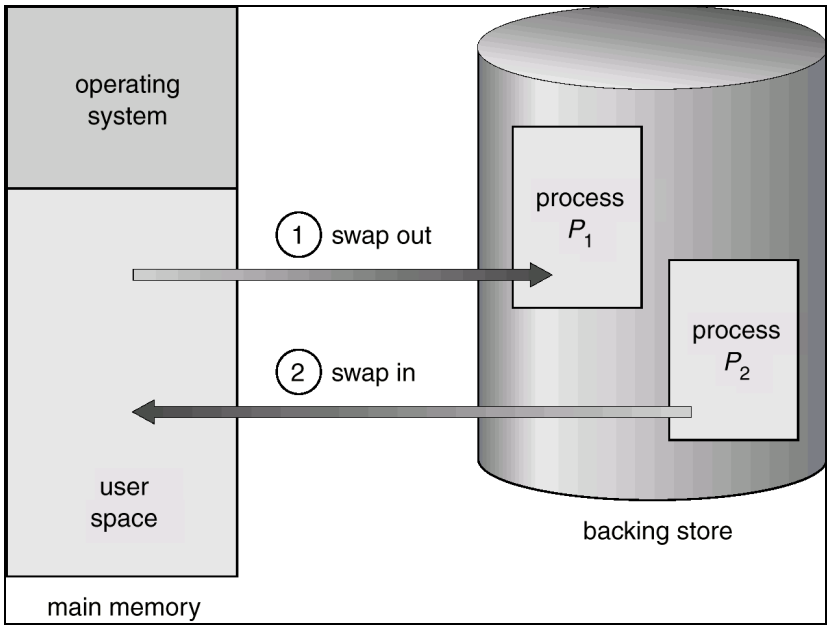


Figure 6.3 Swapping of two processes using a disk as a backing store

Normally a process that is swapped out will be swapped back into the same memory space that 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 moved to different locations. If execution-time binding is being used, 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. The backing store 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. The system maintains a ready queue consisting of all processes whose memory images are on the backing store or in memory and are ready

to run. 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 not, and 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 as normal and transfers control to the selected process.

The context-switch time in such a swapping system is high. If we want to swap a process, it must be idle.

6.3 CONTIGUOUS MEMORY ALLOCATION

The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate different parts of the main memory in the most efficient way possible. This is done using contiguous memory allocation. We usually want several user processes to reside in memory at the same time. Hence, we need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory. In this contiguous memory allocation, each process is contained in a single contiguous section of memory.

6.3.1 Memory Protection

Before discussing memory allocation, we need to discuss the issue of memory protection. ***Memory Protection*** is to protect the OS from user processes, and to protect user processes from one another.

We can provide this protection by using following registers:

- **relocation register** : contains the value of the smallest physical address
- **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 as shown in Figure 6.4.

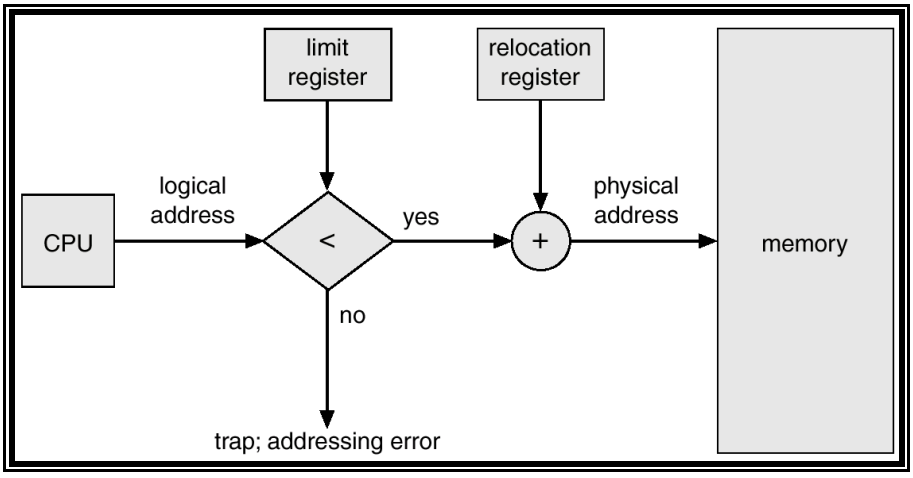


Figure 6.4 Hardware support for relocation and limit registers

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 OS and user programs and data from being modified by this running process.

6.3.2 Memory Allocation

One of the simplest methods for memory allocation is to divide memory into several fixed-sized ***partitions***. Each partition may contain exactly one process. 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. This method is no longer in use.

Another method is a generalization of the fixed-partition scheme. Here, the OS keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes, and is considered as one large block of available memory, ***a hole***. When a process arrives and needs memory, we search for a hole large enough for this process. If we find one, 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.

In general, a set of holes, of various sizes, is scattered throughout memory at any given time. When a process arrives and needs memory, the system searches this set for a hole that is large enough for this process. If the hole is too large, it is split into two: One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. Two adjacent holes are merged to form one larger hole.

How to satisfy a request of size *n* from a list of free holes is a ***dynamic storage-allocation problem***. There are following algorithms:

- **First Fit:** Allocate the first hole that is big enough.
- **Best Fit:** Allocate the smallest hole that is big enough. We must search the entire list, if the list is not ordered by size.
- **Worst Fit:** Allocate the largest hole.

Best fit and first fit are better than worst fit with respect to time and storage utilization. First fit is faster. All the three algorithms suffer from ***external fragmentation***. As processes are loaded and removed from memory, there will be multiple holes. External fragmentation exists when enough total memory space is there to satisfy the request, but it is not contiguous.

6.3.3 Fragmentation

Sometimes, the size of the hole is much smaller than the overhead required to track it. Hence, normally, physical memory is divided into fixed-size block. Then memory is allocated in terms of units. So, sometimes, the memory allocated for a process may be slightly larger than what it requires. Such a difference is known as ***internal fragmentation.*** For example, if the physical memory is divided into block of 4 bytes, and the process

requests for 6 bytes, then it will be allocated with 2 blocks. Hence, the total allocation is 8 bytes leading to 2 bytes of internal fragmentation.

One of the solutions to external fragmentation is ***compaction***: shuffling of memory contents to place all free memory together into one large block. But, compaction is possible only when relocation is dynamic and is done during execution time. That is, if the relocation is static, we cannot apply this technique.

Another solution for external fragmentation is to permit logical-address space of a process to be non-contiguous. This allows a process to be allocated physical memory wherever it is available. To do so, two techniques are there:

- Paging
- Segmentation

These techniques are discussed in the following sections.

6.4 PAGING

Paging is a memory-management scheme that permits the physical-address space of a process to be noncontiguous. Paging avoids the problem of fitting the varying-sized memory chunks onto the backing store.

Paging is commonly used in most OS nowadays. It is implemented in hardware as well as in OS.

6.4.1 Basic Method

Physical memory is broken into fixed-sized blocks called ***frames***. Logical memory is also broken into blocks of the same size called ***pages***. When a process is to be executed, its pages are loaded into any available memory frames from the backing store. The backing store is divided into fixed-sized blocks that are of the same size as the memory frames.

The hardware support for paging is illustrated in Figure 6.5. Every address generated by the CPU is divided into two parts:

- **page number (p)**
- **page offset (d)**

The page number is used as an index into a page table. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit. The paging model of memory is shown in Figure 6.6.

The size of a page is typically a power of 2. If the size of logical-address space is 2m, and a page size is 2n addressing units (bytes or words), then the first m – n bits of a logical address indicate the page number, and the next n bits indicate the page offset.

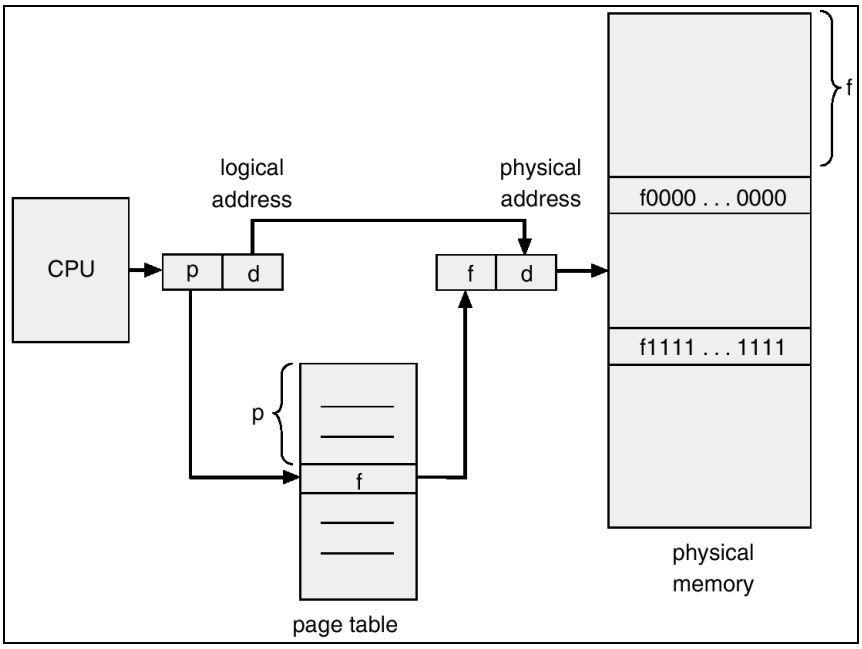


Figure 6.5 Paging hardware

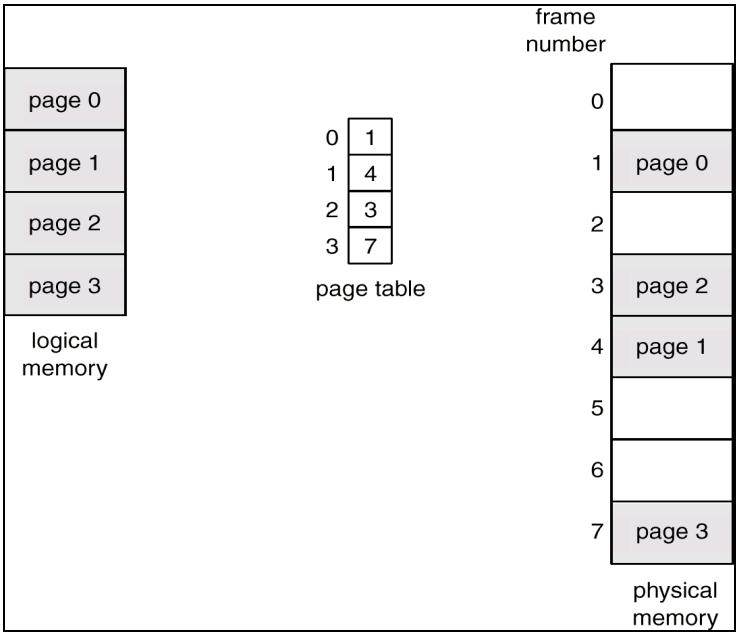


Figure 6.6 Paging model of logical and physical memory

Example:
Consider a page size of 4 bytes and a physical memory of 32 bytes (8 pages) as shown in Figure 6.7. We will see now, how the user's view of memory can be mapped into physical memory:

- Logical address 0 is page 0, offset 0. Indexing into the page table, we find that page 0 is in frame 5. Thus, logical address 0 maps to physical address 20 (= (5 x 4) + 0).
- Logical address 3 (page 0, offset 3) maps to physical address 23 (= (5 x 4) + 3).

- Logical address 4 (page 1, offset 0); according to the page table, page 1 is mapped to frame 6. Thus, logical address 4 maps to physical address 24 (= (6 x 4) + 0).
- Logical address 13 maps to physical address 9.

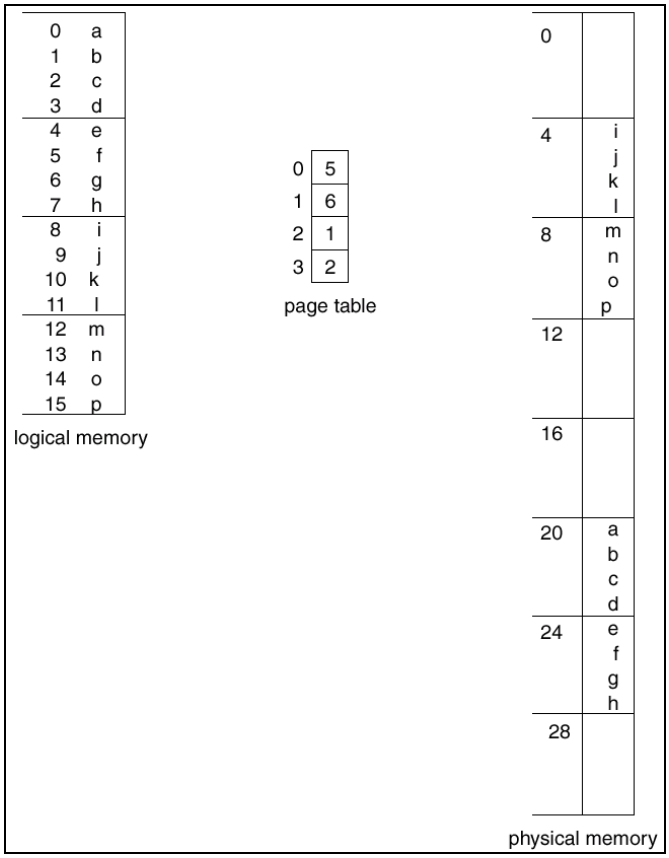


Figure 6.7 Paging Example

When paging is used, there will not be external fragmentation: Any free frame can be allocated to a process that needs it. However, some internal fragmentation will be there. To reduce it, small-sized pages are preferable. But, more number of pages will lead to many entries in page-table, and hence much overhead. So, optimum selection of size of each page is essential.

When a process needs to be executed, its size represented in terms of number of pages is examined. Each page of the process needs one frame. Thus, if the process requires *n* pages, at least *n* frames must be available in memory.

6.4.2 Hardware Support

Each operating system has its own methods for storing page tables. In the simplest case, the page table is implemented as a set of dedicated registers. Another method is: the page table is kept in main memory, and a ***page-table base register (PTBR)*** points to the page table. In this scheme every data/instruction access requires two memory accesses: One for the page table and one for the data/instruction.

This double access problem can be solved by the use of a special fast-lookup hardware cache called ***associative memory*** or ***translation look-aside buffers (TLBs)***. The TLB is associative, high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value. The TLB is used with page tables in the following way:

- The TLB contains only a few of the page-table entries.
- When a logical address is generated by the CPU, its page number is presented to the TLB.
- If the page number is found, its frame number is immediately available and is used to access memory.
- If the page number is not in the TLB (known as a TLB miss), a memory reference to the page table must be made. When the frame number is obtained, we can use it to access memory.

This is shown in Figure 6.8.

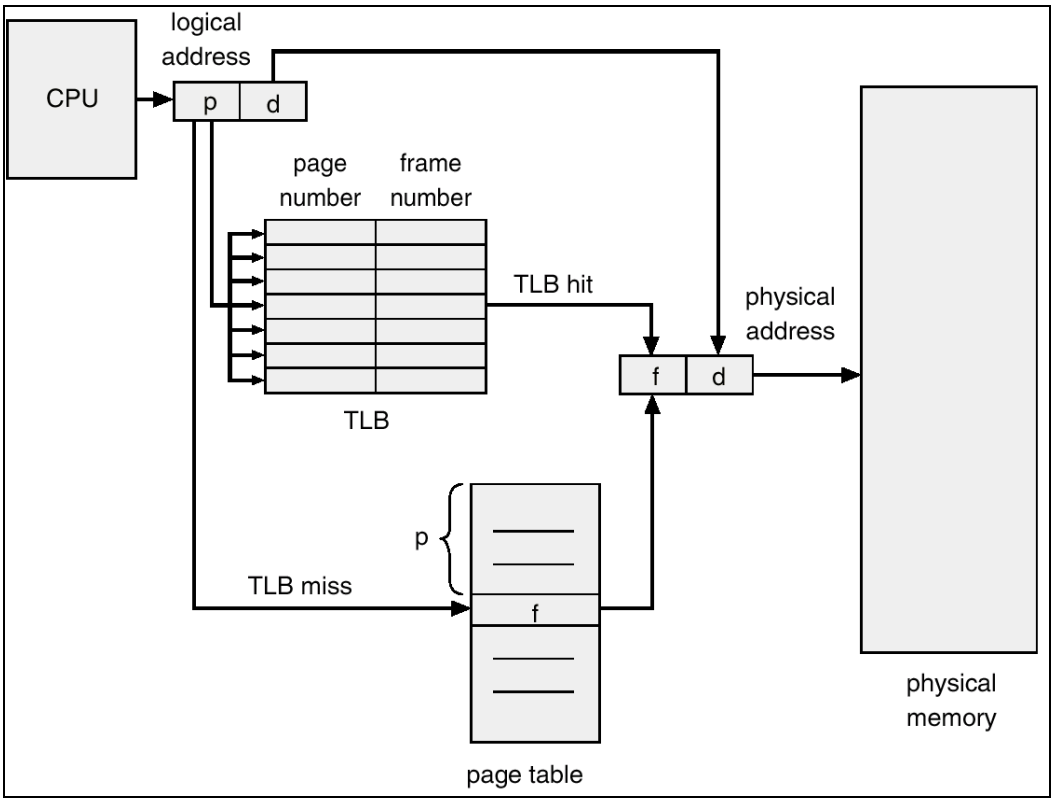


Figure 6.8 Paging hardware with TLB

6.4.3 Protection

In paging technique, protection bits are associated with each frame are kept in the page table. This will help memory protection. One bit can define a page to be read-write or read-only. While referring the page table to get a frame number, the protection bits are also checked. An attempt to write a read-only page causes hardware trap to OS.

Another bit attached to page table is ***valid – invalid bit***. When this bit is set to *valid*, it indicates that the respective page is in the logical address space of the process and hence

it is a legal/valid page. If this bit is set to *invalid*, the page is not in logical address space of the process.

6.4.4 Structure of Page Table

The most common techniques for structuring the page table are:

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Hierarchical Paging: Most modern computer systems support a large logical-address space. In such an environment, the page table itself becomes excessively large. For example, consider a system with a 32-bit logical-address space. If the page size in such a system is 4 KB (2^{12}), then a page table may consist of up to 1 million entries ($2^{32}/2^{12}$). To avoid this problem, one of the solutions is to dividing the page table into smaller pieces. To do so, ***two-level paging algorithm*** can be used. Here, the page table itself is paged as shown in Figure 6.9.

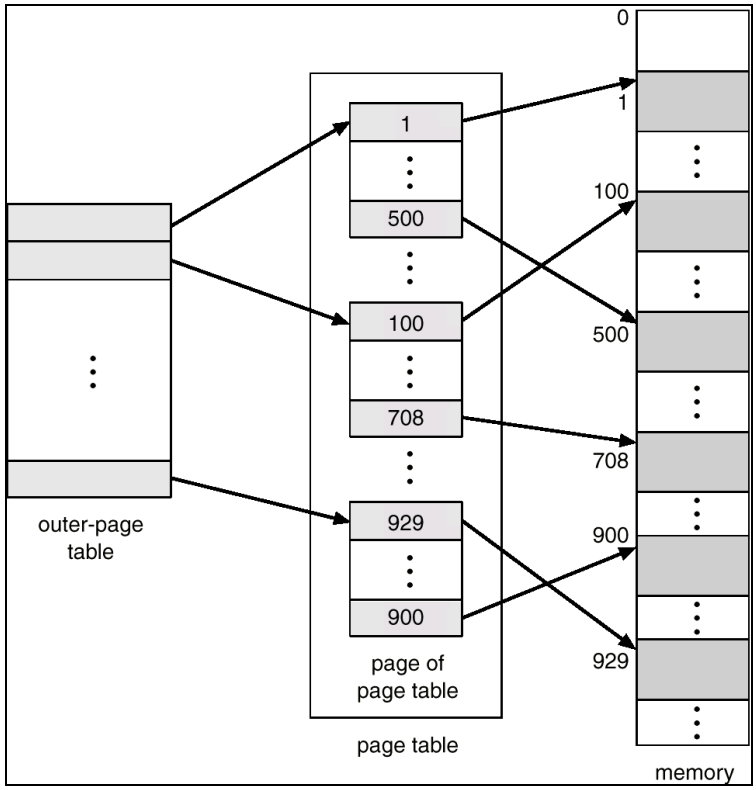


Figure 6.9 A two-level page-table scheme

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 an array of linked lists that hash to the same location (to handle collisions using open-hashing/separate chaining method). Each node in a linked list consists of three fields:

- The virtual page number

- the value of mapped page frame
- a pointer to the next element in the linked list.

The algorithm works as follows:

i. The virtual page number in the virtual address is hashed into the hash table.
ii. The virtual page number is compared to the first field of the first node in the linked list.
iii. If there is a match, the corresponding page frame (second field of a node) is used to form the desired physical address.
iv. If there is no match, subsequent nodes in the linked list are searched for a matching virtual page number.

This scheme is shown in Figure 6.10.

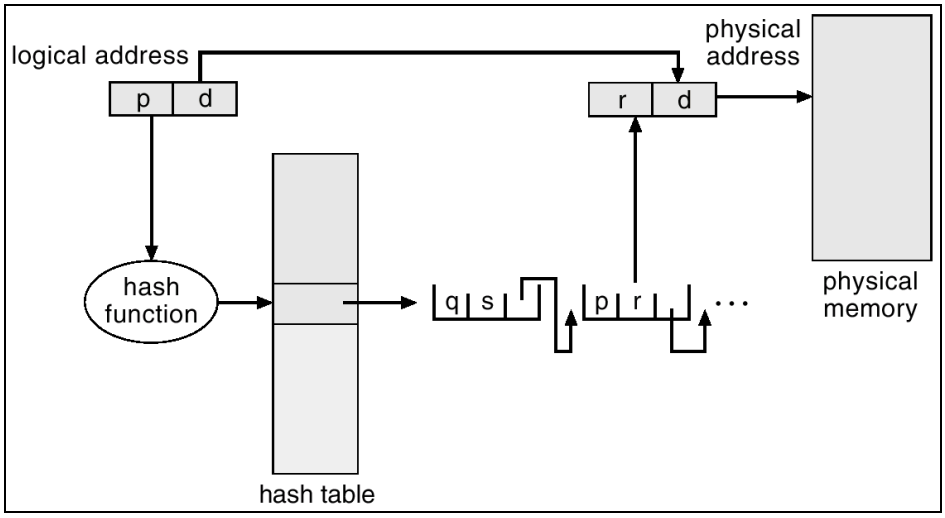


Figure 6.10 Hashed page table

Inverted Page Table: It is obvious that each process has its own page table and every page table has one entry for each page. We know that a page represents logical/virtual addresses. It is the OS which converts the logical addresses into physical addresses later. As, a page table contains millions of entries, it is a burden on OS to covert everything into physical address.

To avoid this problem, inverted page table is used. This table contains one entry for each real physical frame of memory rather than for logical memory. Hence, only one page table is sufficient for the system. As each process has its own address space, another entity specifying ***process identifier (pid)*** in its address – space is used as shown in Figure 6.11. The working is as explained below:

- The page table contains pid and page number.
- The CPU generates logical address containing pid, page number and offset.
- The pair pid and page number is searched for, in the page table.
- If the match is found at i^{th} entry, then the physical address will be <i, offset>
- If the match is not found, illegal address access error occurs.

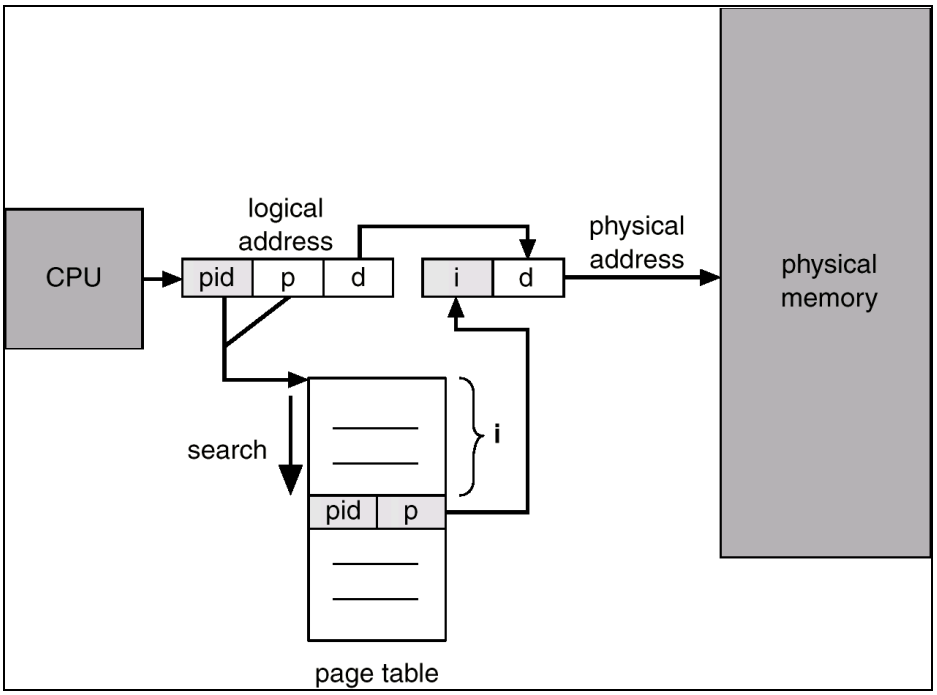


Figure 6.11 Inverted Page Table

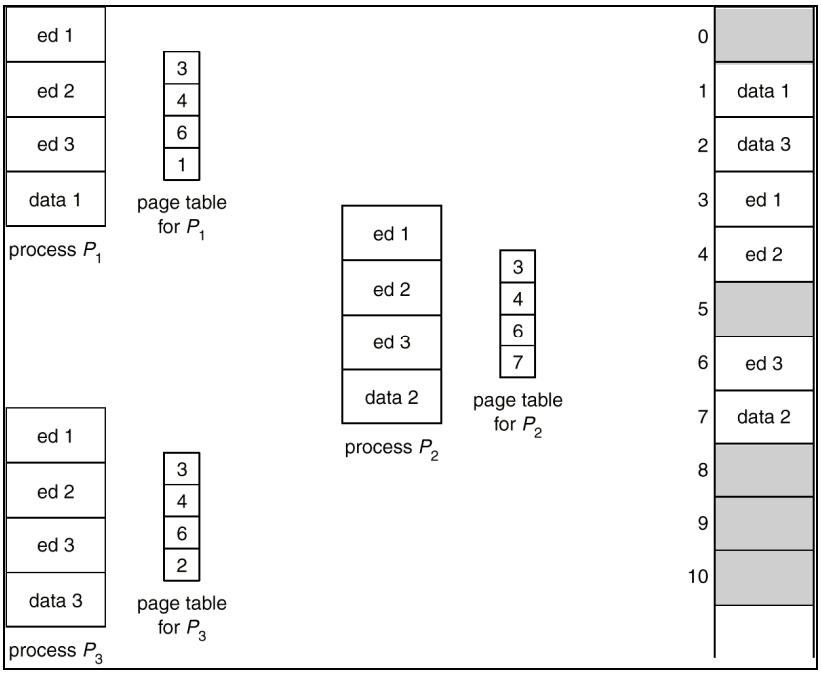


Figure 6.12 Sharing of code in Paging environment

6.4.5 Shared Pages

Paging technique is useful in time-sharing environment. It allows sharing of common code, if possible. Some of the processes have ***re-entrant code***. That is, once they start executing, code will not change. In such situations, pages can be shared. Consider an

example: Assume that 40 users are using a text editor at a time. Let the text editor consists of 150KB of code and 50KB of data space. We need 8000KB for 40 users. As text-editor is reentrant, only one copy of the editor needs to be kept in the physical memory and it can be shared among all the users. But, the data space should be different for every user. This is depicted in Figure 6.12. In this example, size of each page (and also the frame) is taken as 50KB.

Such sharing technique can be applied for compilers, window systems, run-time libraries, database systems etc.

6.5 SEGMENTATION

In paging, we have observed that user view of the memory and the actual physical memory are different. User view of memory is mapped into physical memory.

6.5.1 Basic Method

Users do not think memory as a linear array of bytes. Instead, users view memory as a collection of variable-sized segments, with no necessary ordering among segments as shown in Figure 6.13. For example, we think of a program as a main program with a set of subroutines, procedures, functions, or modules. There may also be various data structures: tables, arrays, stacks, variables, and so on. Each of these modules or data elements is referred to by name. We do not bother which module has is stored before/after other module.

Segmentation is a memory-management scheme that supports user view of memory. A logical-address space is a collection of segments. Each address is specified in terms of the segment number and the offset within the segment.

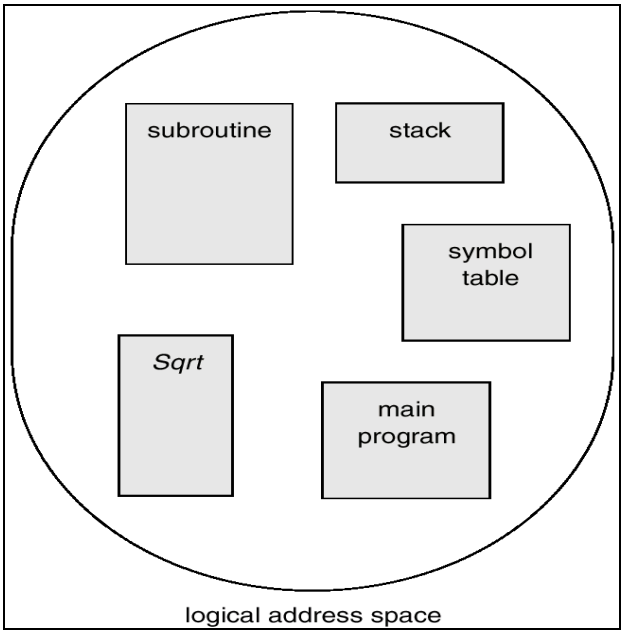


Figure 6.13 User view of a program

6.5.2 Hardware

Here, a two dimensional user-defined addresses are mapped into one-dimensional physical addresses. This mapping is affected by a segment table. Each entry of the segment table has

- *segment base* – contains the starting physical address where the segment resides in memory
- s*egment limit* – specifies the length of the segment

A logical address consists of two parts: a segment number, *s* and an offset *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 (logical addressing attempt beyond end of segment). If this offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte. The segment table is thus essentially an array of base-limit register pairs. The structure is as given in Figure 6.14.

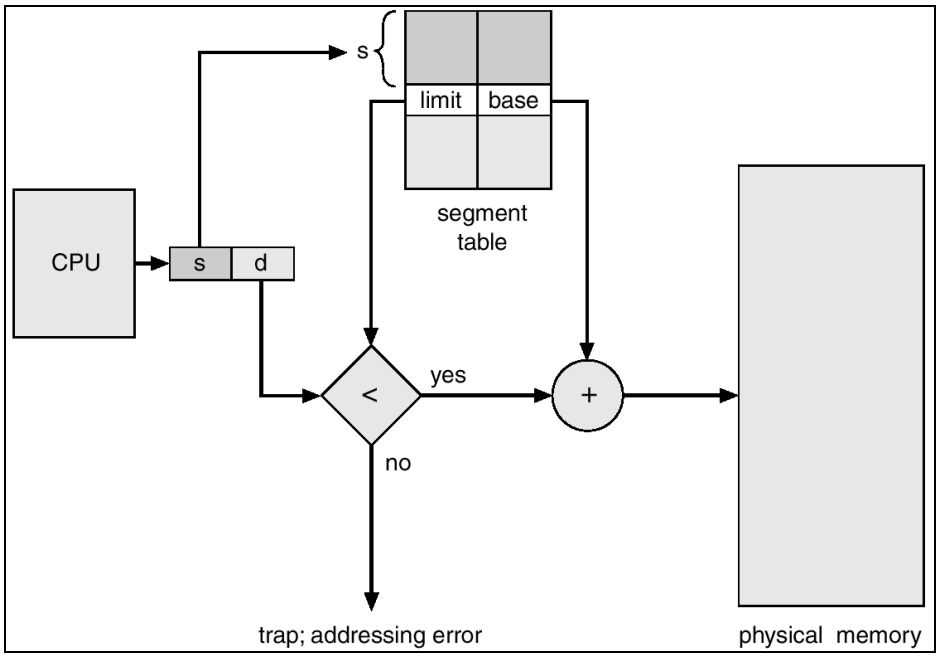


Figure 6.14 Segmentation Hardware

Example: Consider the situation shown in Figure 6.15. We have five segments numbered from 0 through 4. The segment table has a separate entry for each segment. It is giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit). For example, segment 2 is 400 bytes long and begins at the location 4300.

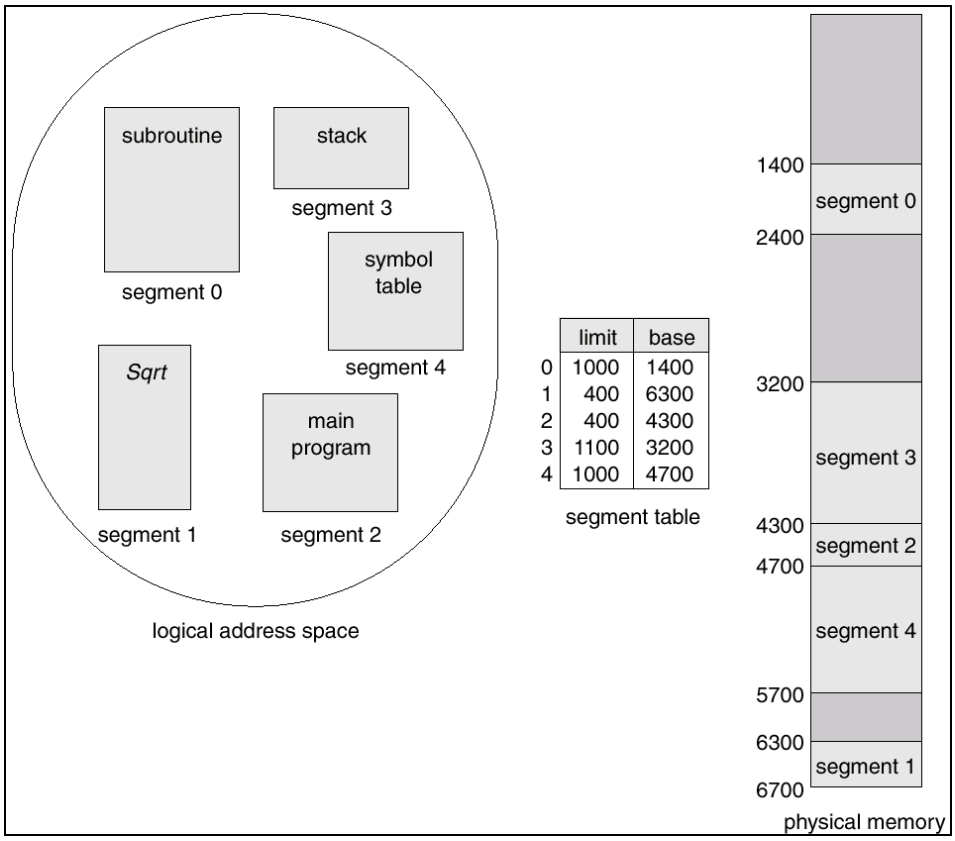


Figure 6.15 Example of Segmentation

6.5.3 Fragmentation

The long-term scheduler must find and allocate memory for all the segments of a user program. This situation is similar to paging *except* that the segments are of variable length; pages are all the same size. Thus, as with the variable-sized partition scheme, memory allocation is a dynamic storage-allocation problem, usually solved with a best-fit or first-fit algorithm. Segmentation may then cause external fragmentation, when all blocks of free memory are too small to accommodate a segment. In this case, the process may simply have to wait until more memory (or at least a larger hole) becomes available, or until compaction creates a larger hole. Because segmentation is by its nature a dynamic relocation algorithm, we can compact memory whenever we want. If the CPU scheduler must wait for one process, because of a memory allocation problem, it may (or may not) skip through the CPU queue looking for a smaller, lower-priority process to run.

6.6 SEGMENTATION WITH PAGING

Both paging and segmentation have advantages and disadvantages. Paging has advantages like –

- No external fragmentation
- Faster allocation

Segmentation has advantages like –

- Sharing
- Protection

These advantages are combined to get better results.

Segmentation with paging is explained as below: Instead of an actual memory location, the segment information includes the address of a page table for the segment. When a program references a memory location the offset is translated to a memory address using the page table. A segment can be extended simply by allocating another memory page and adding it to the segment's page table. Also, the logical address space of a process is divided into two partitions: one consisting of segments that are private to that process, and the other partition consists of segments that can be shared among all the processes.

6.7 DEMAND PAGING

Demand paging is one of the techniques of implementing virtual memory.

(NOTE: Virtual Memory is the separation of user logical memory from physical memory. It allows the execution of processes that may not be completely in memory. Hence, logical address space can therefore be much larger than physical address space. Moreover, it allows address spaces to be shared by several processes and also, allows for more efficient process creation.)

Demand paging is similar to paging system with swapping. Whenever process needs to be executed, only the required pages are swapped into memory. This is called as ***lazy swapping.*** As, the term *swapper* has a different meaning of *'swapping entire process into memory'*, another term ***pager*** is used in the discussion of demand paging.

When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. The pager brings only those necessary pages into memory. Hence, it decreases the swap time and the amount of physical memory needed.

In this scheme, we need to distinguish the pages which are in the memory and pages which are there on the disk. For this purpose, the valid – invalid bit is used. When this bit is set to *valid*, it indicates the page is in the memory. Whereas, the value of bit as *invalid* indicates page is on the disk.

If the process tries to access a page which is not in the memory (means, it is on the disk), ***page fault*** occurs. The paging hardware notices the *invalid* bit in the page table and cause a trap to the OS. This trap is the result of the failure of OS to bring the desired page into memory. This error has to be corrected. The procedure for handling this page fault is as shown in Figure 6.16. The steps are explained below:

1. We check an internal table (usually kept with the process control block) for this process, to determine whether the reference was a valid or invalid memory access.
2. If the reference was invalid, we terminate the process. If it was valid, but we have not yet brought in that page into memory, it is brought now.
3. We find a free frame.
4. We schedule a disk operation to read the desired page into the newly allocated frame.

5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the illegal address trap. The process can now access the page as though it had always been in memory.

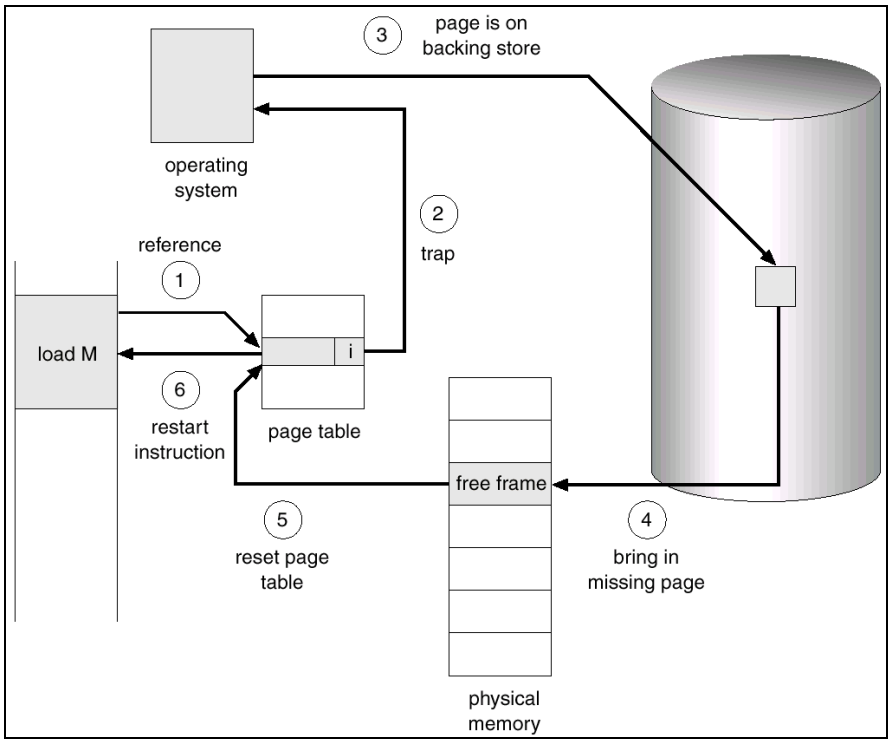


Figure 6.16 Steps in handling page fault

It is important to realize that, because we save the state (registers, condition code, instruction counter etc.) of the interrupted process when the page fault occurs, we can restart the process in exactly the same place and state

In an extreme situation, a process may starts executing with no page in the memory. So, each time an instruction has to be executed, page fault occurs and the required page needs to be brought into the memory. This situation is called as ***pure demand paging***. That is, no page is brought into the memory until it is required.

6.8 PROCESS CREATION

Virtual memory and paging helps in process creation. Two such techniques are discussed here.

6.8.1 Copy-on-Write

Usually, a system call *fork()* can be used to create a child process as a duplicate of its parent. Alternatively, ***copy-on-write*** technique can be used. In this technique, initially, parent and child processes share the same pages. These shared pages are marked as copy-on-write pages. That is, if either process writes to a shared page, a copy of the shared page is created. For example, assume the child process attempts to modify a page

containing portions of the stack; the OS recognizes this as a copy-on-write page. The operating system will then create a copy of this page, mapping it to the address space of the child process. Therefore, the child process will modify its copied page and not the page belonging to the parent process. And, all non-modified pages may be shared by the parent and child processes. Pages that cannot be modified (i.e., pages containing executable code) may be shared by the parent and child. Copy-on-write is a common technique used by several OS when duplicating processes.

6.8.2 Memory-Mapped Files

Memory-mapped file I/O allows file I/O to be treated as routine memory access by *mapping* a disk block to a page in memory. A file is initially read using demand paging. A page-sized portion of the file is read from the file system into a physical page. Subsequent reads/writes to/from the file are treated as ordinary memory accesses. This technique simplifies file access by treating file I/O through memory rather than **read() write()** system calls. It also allows several processes to map the same file allowing the pages in memory to be shared.

6.9 PAGE REPLACEMENT

In a multiprogramming system, there are several processes running. Each of these processes might have been divided into several pages. Physical memory is divided into finite set of frames. Demand paging is applied for bringing the necessary pages from disk to memory. In such a set up, a situation may arise like this:

> A process requests for a page which is not there in the memory. Hence, the page fault occurs. Now, this page has to be swapped into memory from the back store. But, at this moment, no free frames are available.

Now, OS has to make the frames free by swapping out some of the frames to disk and bring newly requested pages into memory. This procedure is called as ***page replacement.***

6.9.1 Basic Scheme

Page replacement takes the following approach. If no frame is free, we find one page that is not currently being used and free it. Necessary changes have to be made in page table to indicate that this page is no longer in memory. We can now use the freed frame to hold the page for which the process faulted. The logic is shown in Figure 6.17. Modified page-fault service routine to include page replacement is as below:

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 page to the disk; change the page and frame tables accordingly.
3. Read the desired page into the (newly) free frame; change the page and frame tables.
4. Restart the user process.

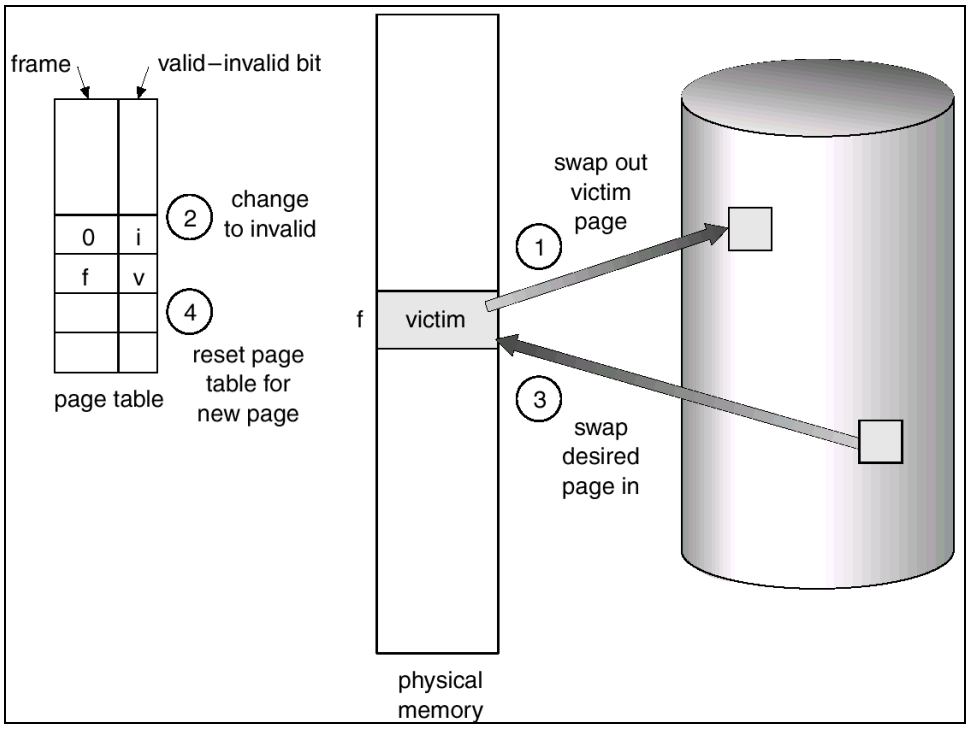


Figure 6.17 Page Replacement

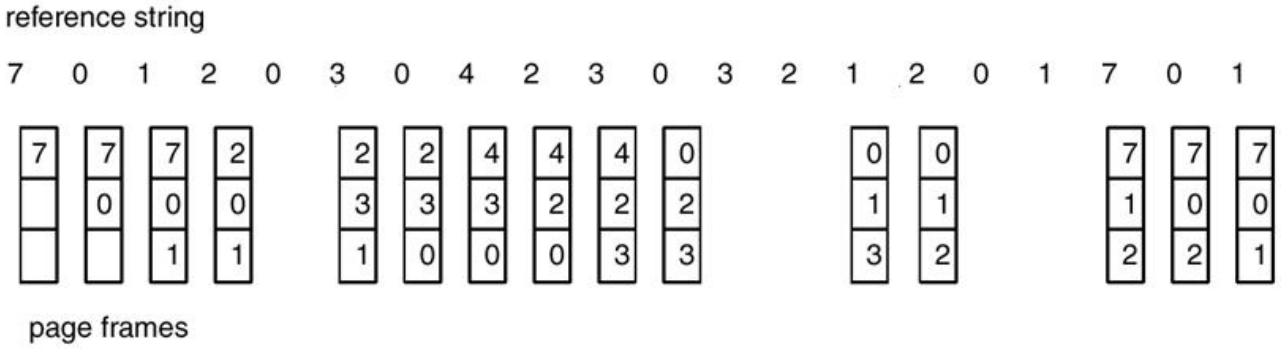
(NOTE: Whenever a page has to be swapped out, the question arises: whether this page has to written back to disk? Writing is necessary if the page has been modified during the process execution. Instead of writing every time a page is being swapped out, a bit called as ***modify bit*** (or ***dirty bit***) is introduced in the hardware. If any modification is done, the bit is set, otherwise not. So, while swapping out a page, the value of this bit is checked and the page is written to disk only if the bit is set. Otherwise, as the copy of the page will always reside in the disk, it need not be written again.)

There are many page replacement algorithms. Every OS adopts its own page-replacement scheme. The algorithm with the lowest page-fault rate must be chosen. Page-replacement algorithm is evaluated by running it on a string of memory references (called as a ***reference string***). Normally, a reference string consists of the numbers indicating the sequence of pages required by the process for execution. For applying any algorithm for page – replacement, we must know the number of frames available.

6.9.2 FIFO Page Replacement

It is the simplest page – replacement algorithm. As the name suggests, the first page which has been brought into memory will be replaced first when there no space for new page to arrive. Initially, we assume that no page is brought into memory. Hence, there will be few (that is equal to number of frames) page faults, initially. Then, whenever there is a request for a page, it is checked inside the frames. If that page is not available, page – replacement should take place.

Example: Consider a reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1
Let the number of frames be 3.
Now, page – replacement according to FIFO algorithm is done as below –



In the above example, there are 15 page faults.

NOTE that, the performance of FIFO page replacement algorithm is not good. Because, the page that has been just swapped out may be needed immediately in the next step. This will cause more page faults.

Sometimes, one may feel that increase in number of frames may decrease the page faults. But this is not true always. For example, consider the reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5. Now, the number of page faults for three frames will be 9 and number of page faults for four frames will be 10. Such an unexpected result is known as ***Belady's anomaly.*** It can be stated as: *the page fault rate may increase as the number of allocated frames increases.*

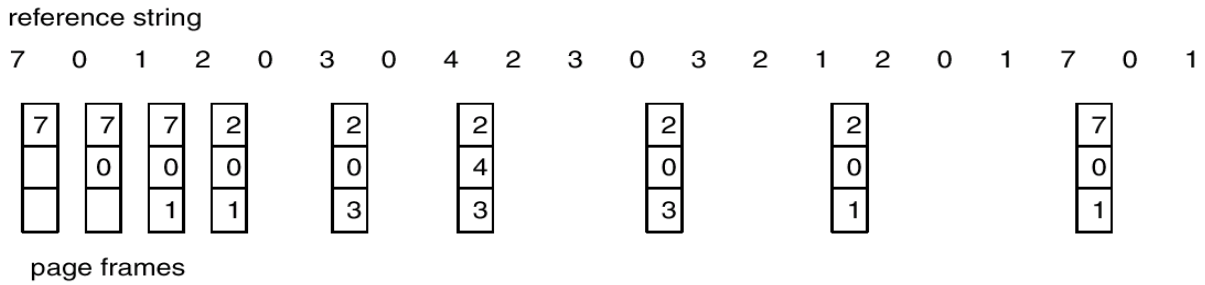
6.9.3 Optimal Page Replacement

An Optimal Page Replacement algorithm (also known as ***OPT*** or ***MIN*** algorithm) do not suffer from Belady's anomaly. It is stated as:

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

This algorithm results in lowest page – faults.

Example:



Here, number of page faults = 9

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

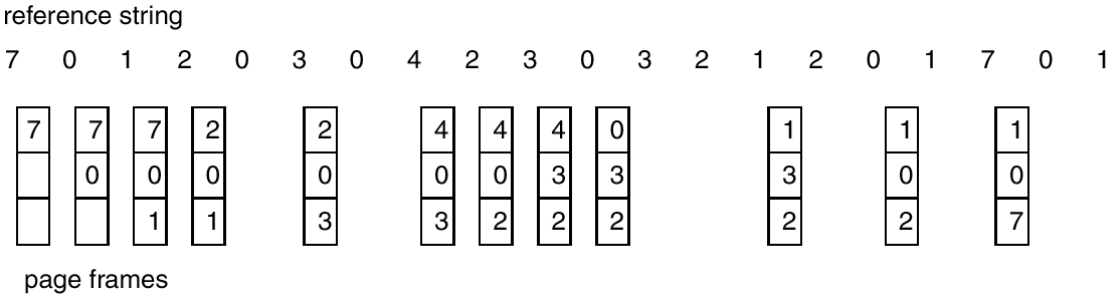
6.9.4 LRU Page Replacement

Least Recently Used page replacement algorithm states that:

Replace the page that has not been used for the longest period of time.

This algorithm is better than FIFO.

Example:



Here, number of page faults = 12

NOTE: The major problem in LRU algorithm is hardware implementation. There are two approaches for hardware implementation:

- **Counters:** Each page table is associated with a *time-of-use* field and CPU is added with a logical clock or counter. When a page needs to be changed, look at the counters to determine which page has to be changed.
- **Stack:** Keep a stack of page numbers. Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the top of the stack is always the most recently used page and the bottom is the least recently used page. To maintain this stack, a doubly linked list can be used.

6.10 ALLOCATION OF FRAMES

Allocation of limited set of free frames to all the processes is challenge. How many frames must be allocated to each process – is a question of interest.

In a single processor system, if only one process has to be executed, all available frames can be allocated to that process. If there are more pages than the frames, and the request is made for a new page, then page – replacement has to be done. But, in case of multiprogramming environment, there will be more than one process in memory at time. Now, the logic used in single-processor system doesn't work.

6.10.1 Minimum Number of Frames

Allocation of frames has various constraints:

- We cannot allocate more than the total number of available frames
- We must allocate at least a minimum number of frames to each process

In most of the cases, as the number of frames allocated to each process decreases, the page fault increases. This will slow down the process execution.

In general, each process should get a minimum number of frames. And this minimum number is defined by the instruction –set architecture. Remember that, when a page fault occurs before an executing instruction is complete, the instruction must be restarted. Consequently, we must have enough frames to hold all the different pages that any single instruction can reference. The maximum number of frames that can be allocated to a process depends on available physical memory.

6.10.2 Global v/s Local Allocation

As multiple processes competing for frames, the page – replacement algorithms can be classified into two categories as explained below:

- **Global Replacement:**
  - It allows a process to select a replacement frame from the set of all frames, including the frames that are being used by other processes.
  - High – priority processes can select a replacement from the lower priority processes.
  - Number of frames allocated for a process may increase/decrease in the course of execution.
  - A process cannot control its own page-fault rate.
  - As a process depends on paging behavior of other processes also, its performance may vary from one execution to other execution.
  - This method results in greater throughput.
- **Local Replacement:**
  - Here, a process can chose a replacement frame from only its own set of allocated frames.
  - Number of frames allocated for a process remains constant in the course of execution.
  - A process depends only on its paging behavior. Hence, the performance is constant.

6.11 THRASHING

If the number of frames allocated to a low-priority process falls below the minimum number required by the computer architecture, we must suspend the execution of that process. We should then page out its remaining pages, freeing all its allocated frames. This provision introduces a swap-in, swap-out level of intermediate CPU scheduling.

Whenever any process does not have enough frames, it will page-fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again. The process continues to fault, replacing pages for which it then faults and brings back in right away. This high paging activity is called ***thrashing***. A process is thrashing if it is spending more time paging than executing.

Thrashing affects the performance of CPU as explained below:

If the CPU utilization is low, we normally increase the degree of multiprogramming by adding a new process to the system. A global page-replacement algorithm is used, and hence, the new process replaces the frames belonging to other processes as well. As the degree of multiprogramming increases, obviously there will be more page faults leading to thrashing. When every process starts waiting for paging rather than executing, the CPU utilization decreases. This problem is shown in Figure 6.18. The effects of thrashing can be limited by using local replacement algorithm.

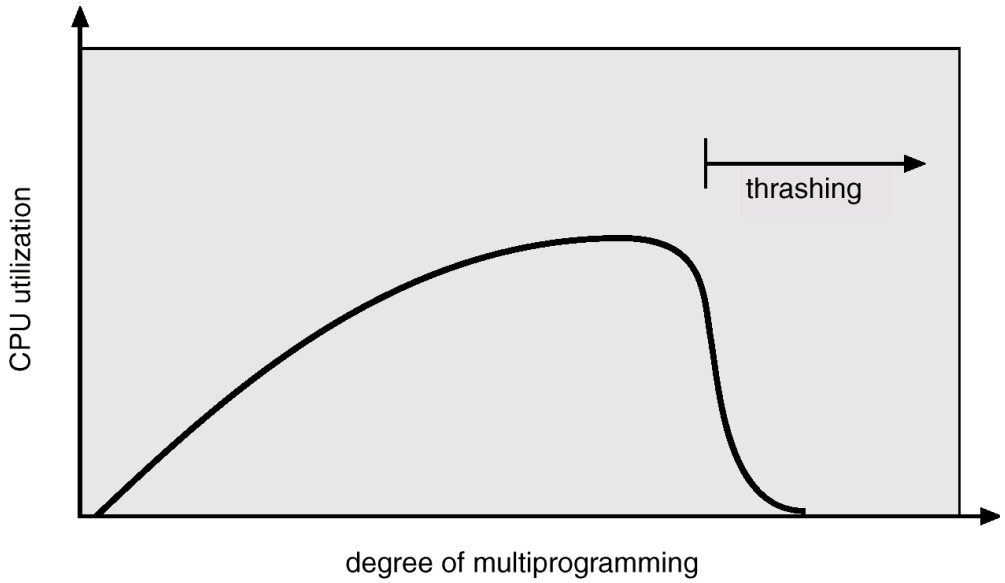


Figure 6.18 Thrashing

****