Modulue III

Main Memory Management Strategies

- Every program to be executed has to be executed must be in memory. The instruction must be fetched from memory before it is executed.
- In multi-tasking OS memory management is complex, because as processes are swapped in and out of the CPU, their code and data must be swapped in and out of memory.

Basic Hardware

Main memory, cache and CPU registers in the processors are the only storage spaces that CPU can access directly.

The program and data must be bought into the memory from the disk, for the process to run. Each process has a separate memory space and must access only this range of legal addresses. Protection of memory is required to ensure correct operation. This prevention is provided by hardware implementation. Two registers are used - a base register and a limit register. The base register holds the smallest legal physical memory address; the **limit register** specifies the size of the range.

For example, if the base register holds 1000 and limit register is 500, then the program can legally access all addresses from 1000 through 1500 (inclusive).

Protection of memory space is done. Any attempt by an executing program to access operating-system memory or other program memory results in a trap to the operating system, which treats the attempt as a fatal error. This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.

The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction. Since privileged instructions can be executed only in kernel mode only the operating system can load the base and limit registers.

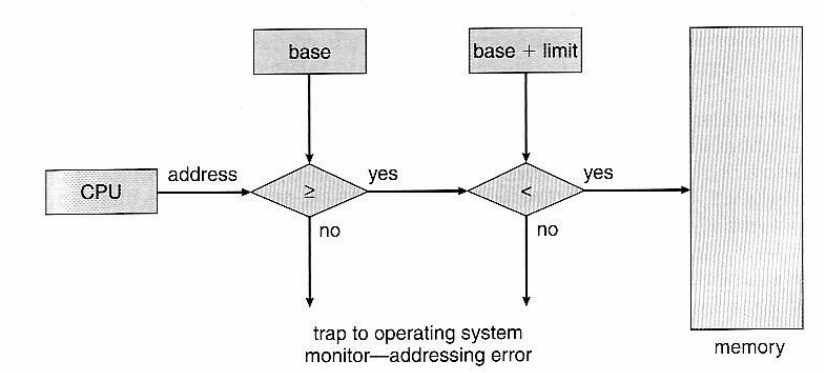


Figure 8.2 Hardware address protection with base and limit registers.

Address Binding

- User programs typically refer to memory addresses with symbolic names such as "i", "count", and "average Temperature". These symbolic names must be mapped or ***bound*** to physical memory addresses, which typically occurs in several stages:
  - **Compile Time** - If it is known at compile time where a program will reside in physical memory, then ***absolute code*** can be generated by the compiler, containing actual physical addresses. However if the load address changes at some later time, then the program will have to be recompiled.
  - **Load Time** - If the location at which a program will be loaded is not known at compile time, then the compiler must generate ***relocatable code***, which references addresses relative to the start of the program. If that starting address changes, then the program must be reloaded but not recompiled.
  - **Execution Time** - If a program can be moved around in memory during the course of its execution, then binding must be delayed until execution time.
- Figure 8.3 shows the various stages of the binding processes and the units involved in each stage:

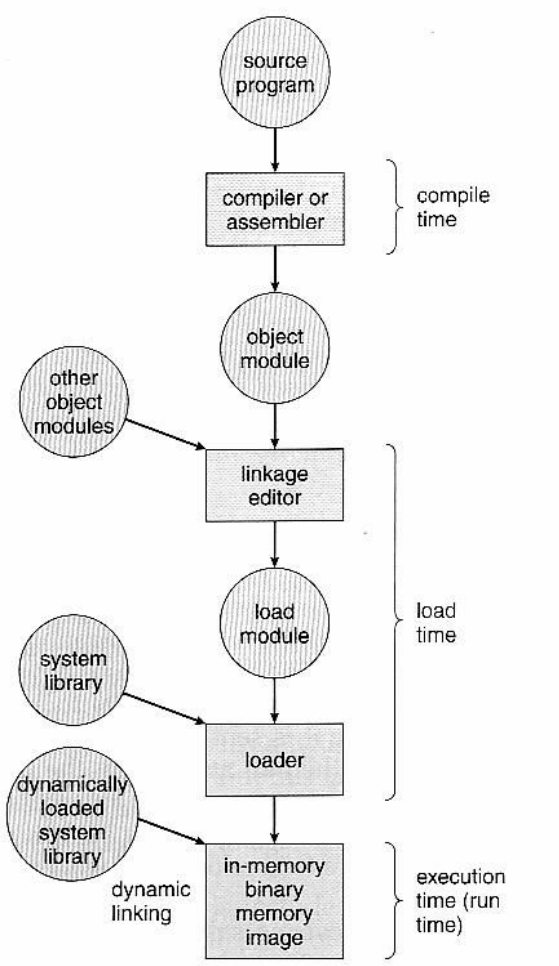


Figure 8.3 Multistep processing of a user program.

Logical Versus Physical Address Space

- The address generated by the CPU is a ***logical address***, whereas the memory address where programs are actually stored is a ***physical address***.
- The set of all logical addresses used by a program composes the ***logical address space***, and the set of all corresponding physical addresses composes the ***physical address space.***
- The run time mapping of logical to physical addresses is handled by the ***memory-management unit, MMU***.
    - The MMU can take on many forms. One of the simplest is a modification of the base-register scheme described earlier.
    - The base register is now termed a ***relocation register***, whose value is added to every memory request at the hardware level.

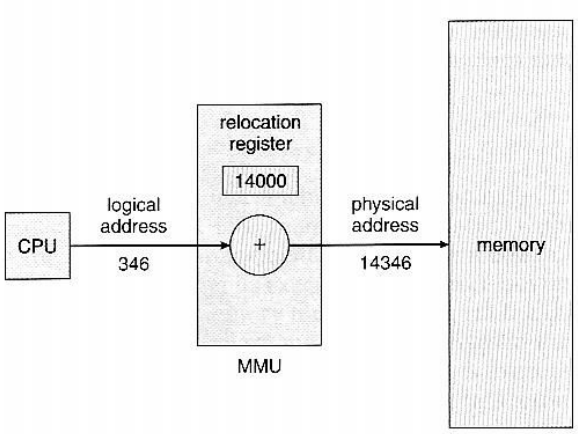


Figure 8.4 Dynamic relocation using a relocation register.

8.1.4 Dynamic Loading

- Rather than loading an entire program into memory at once, dynamic loading loads up each routine as it is called. The advantage is that unused routines need not be loaded, thus reducing total memory usage and generating faster program startup times. The disadvantage is the added complexity and overhead of checking to see if a routine is loaded every time it is called and then loading it up if it is not already loaded.

8.1.5 Dynamic Linking and Shared Libraries

- With ***static linking*** library modules get fully included in executable modules, wasting both disk space and main memory usage, because every program that included a certain routine from the library would have to have their own copy of that routine linked into their executable code.
- With ***dynamic linking***, however, only a stub is linked into the executable module, containing references to the actual library module linked in at run time.

- This method saves disk space, because the library routines do not need to be fully included in the executable modules, only the stubs.
- An added benefit of ***dynamically linked libraries*** (DLLs, also known as ***shared libraries*** or ***shared objects*** on UNIX systems) involves easy upgrades and updates.

8.2 Swapping

- A process must be loaded into memory in order to execute.
- If there is not enough memory available to keep all running processes in memory at the same time, then some processes that are not currently using the CPU may have their memory swapped out to a fast local disk called the ***backing store.***
- Swapping is **the process of moving a process from memory to backing store and moving another process from backing store to memory**. Swapping is a very slow process compared to other operations.
- It is important to swap processes out of memory only when they are idle, or more to the point, only when there are no pending I/O operations. (Otherwise the pending I/O operation could write into the wrong process's memory space.) The solution is to either swap only totally idle processes, or do I/O operations only into and out of OS buffers, which are then transferred to or from process's main memory as a second step.
- Most modern OSes no longer use swapping, because it is too slow and there are faster alternatives available. (e.g. Paging.) However some UNIX systems will still invoke swapping if the system gets extremely full, and then discontinue swapping when the load reduces again. Windows 3.1 would use a modified version of swapping that was somewhat controlled by the user, swapping process's out if necessary and then only swapping them back in when the user focused on that particular window.

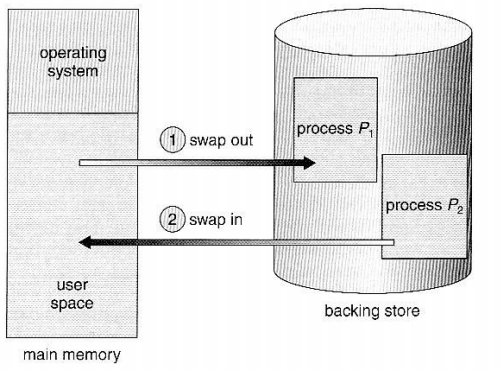


Figure 8.5 Swapping of two processes using a disk as a backing store.

8.3 Contiguous Memory Allocation

- One approach to memory management is to load each process into a contiguous space. The operating system is allocated space first, usually at either low or high memory locations, and then the remaining available memory is allocated to processes as needed. (The OS is usually loaded low, because that is

where the interrupt vectors are located). Here each process is contained in a single contiguous section of memory.

8.3.1 Memory Mapping and Protection

- The system shown in figure below allows protection against user programs accessing areas that they should not, allows programs to be relocated to different memory starting addresses as needed, and allows the memory space devoted to the OS to grow or shrink dynamically as needs change.

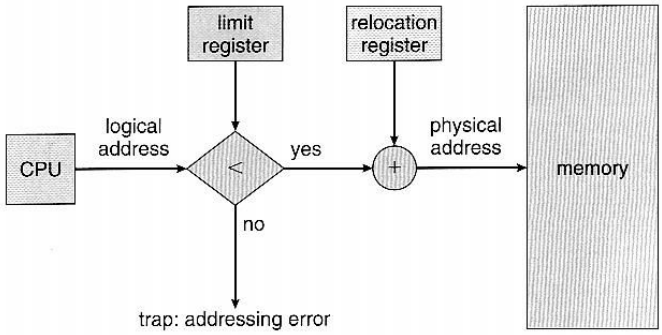


Figure 8.6 Hardware support for relocation and limit registers.

8.3.2 Memory Allocation

- One method of allocating contiguous memory is to divide all available memory into **equal sized** partitions, and to assign each process to their own partition (called as **MFT**). This restricts both the number of simultaneous processes and the maximum size of each process, and is no longer used.
- An alternate approach is to keep a list of unused (free) memory blocks (holes), and to find a hole of a **suitable size** whenever a process needs to be loaded into memory (called as **MVT**). There are many different strategies for finding the "best" allocation of memory to processes, including the three most commonly discussed:
    1. **First fit** - Search the list of holes until one is found that is **big enough** to satisfy the request, and assign a portion of that hole to that process. Whatever fraction of the hole not needed by the request is left on the free list as a smaller hole. Subsequent requests may start looking either from the beginning of the list or from the point at which this search ended.
    2. **Best fit** - Allocate the ***smallest*** **hole that is big enough** to satisfy the request. This saves large holes for other process requests that may need them later, but the resulting unused portions of holes may be too small to be of any use, and will therefore be wasted. Keeping the free list sorted can speed up the process of finding the right hole.
    3. **Worst fit** - Allocate the **largest hole** available, thereby increasing the likelihood that the remaining portion will be usable for satisfying future requests.
- Simulations show that either first or best fit are better than worst fit in terms of both time and storage utilization. First and best fits are about equal in terms of storage utilization, but first fit is faster.

8.3.3. Fragmentation

The allocation of memory to process leads to fragmentation of memory. A hole is the free space available within memory. The two types of fragmentation are –

- External fragmentation – holes present in between the process
- Internal fragmentation - holes are present within the process itself. i.e., There is free space within a process.

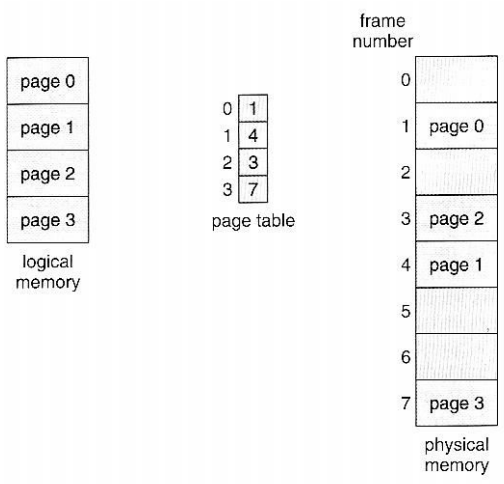
- ***Internal fragmentation*** occurs with all memory allocation strategies. This is caused by the fact that memory is allocated in blocks of a fixed size, whereas the actual memory needed will rarely be that exact size.
- If the programs in memory are relocatable, (using execution-time address binding), then the **external fragmentation** problem can be reduced via ***compaction***, i.e. moving all processes down to one end of physical memory so as to place all free memory together to get a large free block. This only involves updating the relocation register for each process, as all internal work is done using logical addresses.
- Another solution to external fragmentation is to allow processes to use non-contiguous blocks of physical memory- **Paging** and **Segmentation**.

8.4 Paging

- Paging is a memory management scheme that allows processes to be stored in physical memory discontinuously. It eliminates problems with fragmentation by allocating memory in equal sized blocks known as ***pages***.
- Paging eliminates most of the problems of the other methods discussed previously, and is the predominant memory management technique used today.

8.4.1 Basic Method

- The basic idea behind paging is to divide physical memory into a number of equal sized blocks called ***frames***, and to divide a program's logical memory space into blocks of the same size called ***pages.***



- Any page (from any process) can be placed into any available frame.
- The ***page table*** is used to look up which frame a particular page is stored in at the moment. In the following example, for instance, page 2 of the program's logical memory is currently stored in frame 3 of physical memory.

- A logical address consists of two parts: A page number in which the address resides, and an offset from the beginning of that page. (The number of bits in the page number limits how many pages a single process can address. The number of bits in the offset determines the

maximum size of each page, and should correspond to the system frame size.)

- The page table maps the page number to a frame number, to yield a physical address which also has two parts: The frame number and the offset within that frame. The number of bits in the frame number determines how many frames the system can address, and the number of bits in the offset determines the size of each frame.
- Page numbers, frame numbers, and frame sizes are determined by the architecture, but are typically powers of two, allowing addresses to be split at a certain number of bits. For example, if the logical address size is 2^m and the page size is 2^n, then the high-order m-n bits of a logical address designate the page number and the remaining n bits represent the offset.

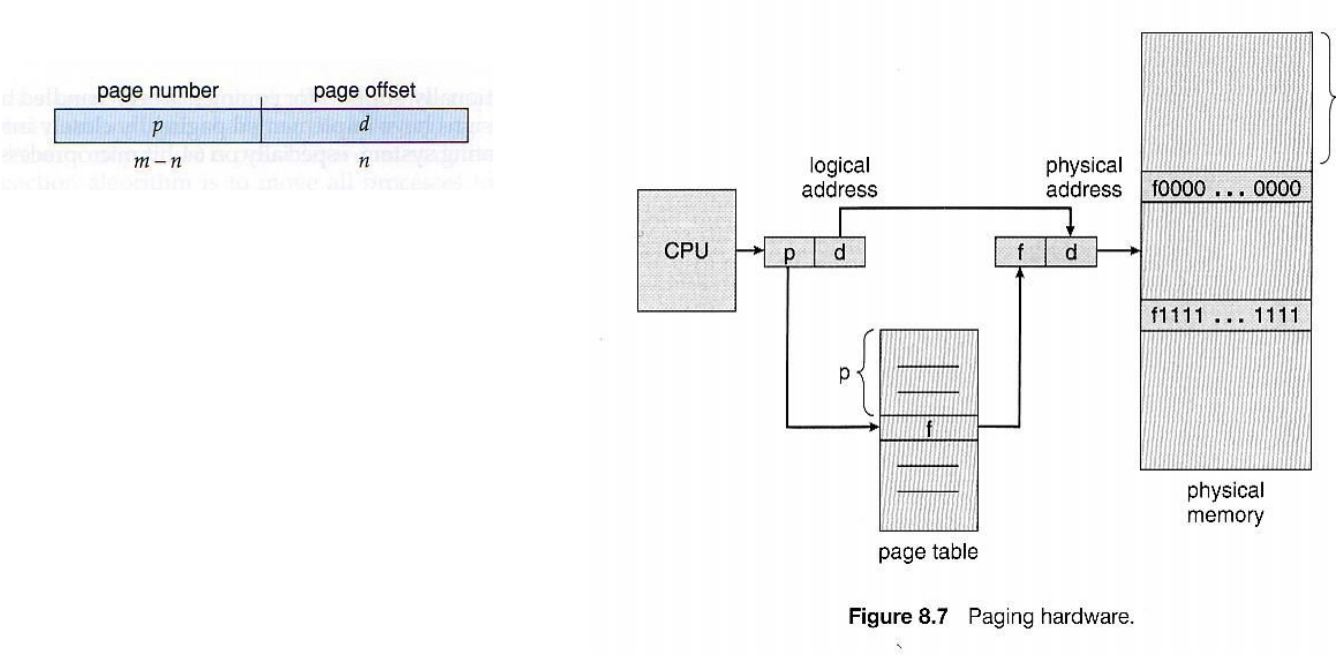


Figure 8.7 Paging hardware.

- Note that paging is like having a table of relocation registers, one for each page of the logical memory.
- There is **no external fragmentation** with paging. All blocks of physical memory are used, and there are no gaps in between and no problems with finding the right sized hole for a particular chunk of memory.
- There is, however, internal fragmentation. Memory is allocated in chunks the size of a page, and on the average, the last page will only be half full, wasting on the average half a page of memory per process.
- Larger page sizes waste more memory, but are more efficient in terms of overhead. Modern trends have been to increase page sizes, and some systems even have multiple size pages to try and make the best of both worlds.

- Consider the following example, in which a process has 16 bytes of logical memory, mapped in 4 byte pages into 32 bytes of physical memory. (Presumably some other processes would be consuming the remaining 16 bytes of physical memory.)

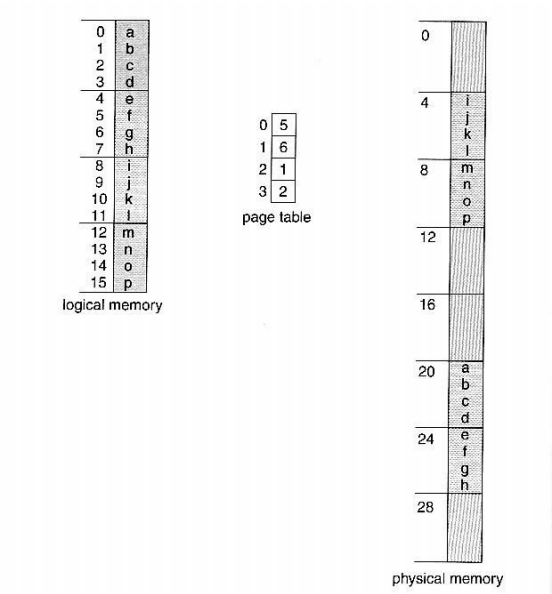


Figure 8.9 Paging example for a 32-byte memory with 4-byte pages.

- When a process requests memory (e.g. when its code is loaded in from disk), free frames are allocated from a free-frame list, and inserted into that process's page table.
- Processes are blocked from accessing anyone else's memory because all of their memory requests are mapped through their page table. There is no way for them to generate an address that maps into any other process's memory space.
- The operating system must keep track of each individual process's page table, updating it whenever the process's pages get moved in and out of memory, and applying the correct page table when processing system calls for a particular process. This all increases the overhead involved when swapping processes in and out of the CPU. (The currently active page table must be updated to reflect the process that is currently running.)

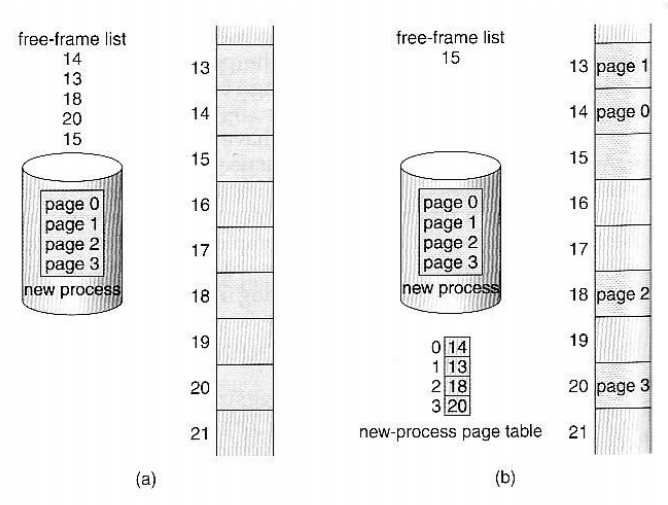
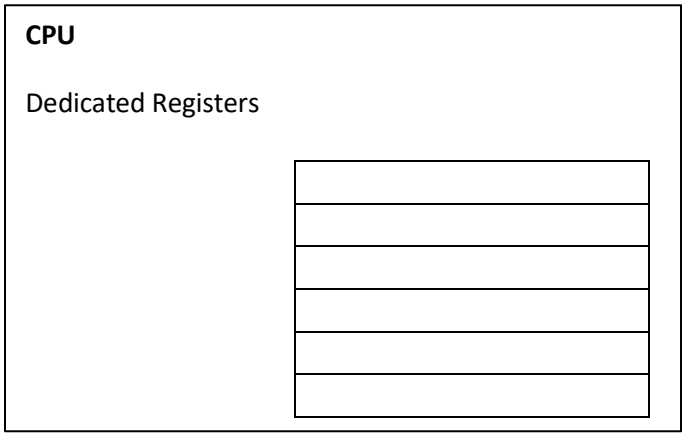


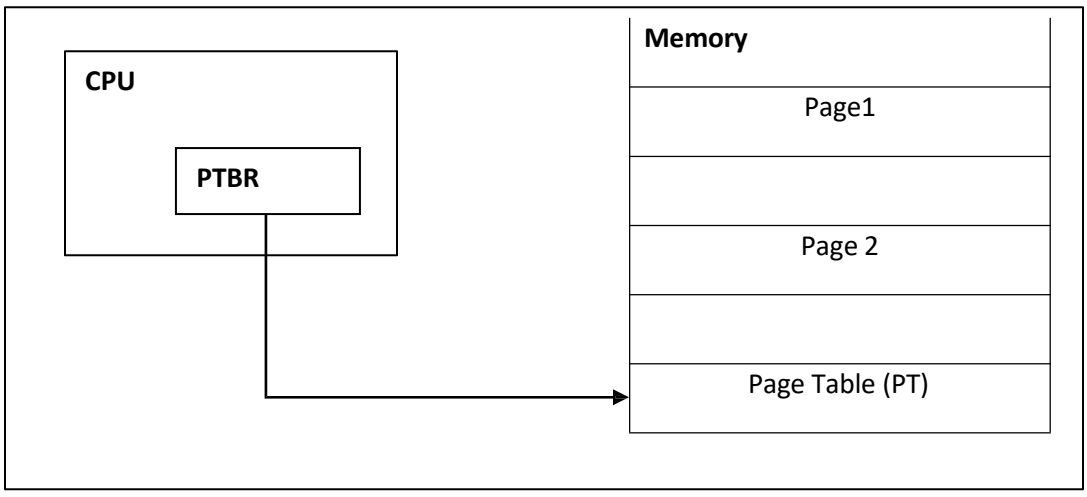
Figure 8.10 Free frames (a) before allocation and (b) after allocation.

8.4.2 Hardware Support

- Page lookups must be done for every memory reference, and whenever a process gets swapped in or out of the CPU, its page table must be swapped in and out too, along with the instruction registers, etc. It is therefore appropriate to provide hardware support for this operation, in order to make it as fast as possible and to make process switches as fast as possible also.
- One option is to use **a set of dedicated registers** for the page table. Here each register content is loaded, when the program is loaded into memory. For example, the DEC PDP-11 uses 16-bit addressing and 8 KB pages, resulting in only 8 pages per process. (It takes 13 bits to address 8 KB of offset, leaving only 3 bits to define a page number.)



- An alternate option is to store the page table in main memory, and to use a single register (called the ***page-table base register, PTBR***) to record the address of the page table is memory.
    - Process switching is fast, because only the single register needs to be changed.
    - However memory access is slow, because every memory access now requires ***two*** memory accesses - One to fetch the frame number from memory and then another one to access the desired memory location.



- The solution to this problem is to use a very special high-speed memory device called the ***translation look-aside buffer, TLB.***
  - The benefit of the TLB is that it can search an entire table for a key value in parallel, and if it is found anywhere in the table, then the corresponding lookup value is returned.
  - It is used as a cache device.
  - Addresses are first checked against the TLB, and if the page is not there (a TLB miss), then the frame is looked up from main memory and the TLB is updated.
  - If the TLB is full, then replacement strategies range from ***least-recently used, LRU*** to random.
  - Some TLBs allow some entries to be ***wired down***, which means that they cannot be removed from the TLB. Typically these would be kernel frames.

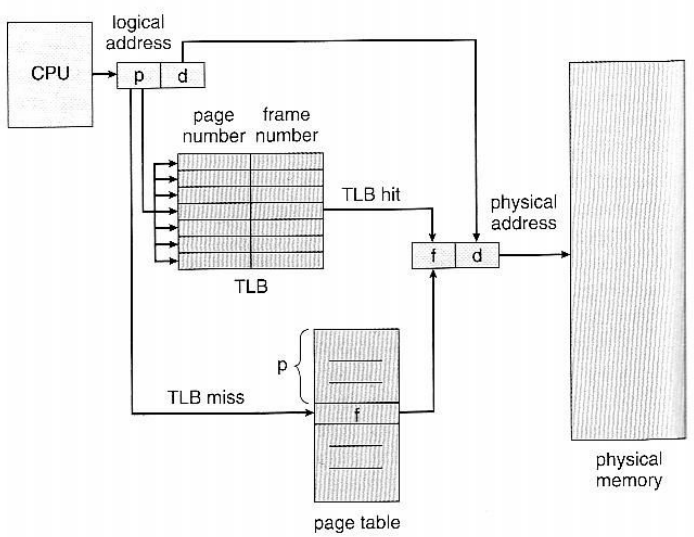


Figure 8.11 Paging hardware with TLB.

- Some TLBs store ***address-space identifiers, ASIDs***, to keep track of which process "owns" a particular entry in the TLB. This allows entries from multiple processes to be stored simultaneously in the TLB without granting one process access to some other process's memory location. Without this feature the TLB has to be flushed clean with every process switch.
- The percentage of time that the desired information is found in the TLB is termed the ***hit ratio***.
- For example, suppose that it takes 100 nanoseconds to access main memory, and only 20 nanoseconds to search the TLB. So a TLB hit takes 120 nanoseconds total (20 to find the frame number and then another 100 to go get the data), and a TLB miss takes 220 (20 to search the TLB, 100 to go get the frame number, and then another 100 to go get the data.) So with an 80% TLB hit ratio, the average memory access time would be:

  0.80 * 120 + 0.20 * 220 = 140 nanoseconds

  for a 40% slowdown to get the frame number. A 98% hit rate would yield 122 nanoseconds average access time (you should verify this), for a 22% slowdown.

8.4.3 Protection

- The page table can also help to protect processes from accessing memory.

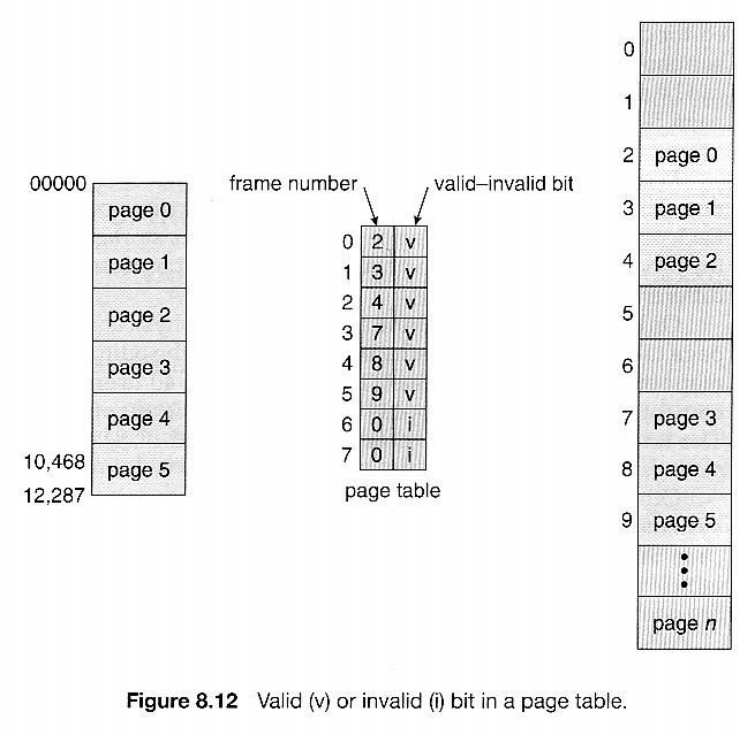


Figure 8.12 Valid (v) or invalid (i) bit in a page table.

- A bit can be added to the page table. Valid / invalid bits can be added to the page table. The valid bit 'V' shows that the page is valid and updated, and the invalid bit 'i' shows that the page is not valid and updated page is not in the physical memory.
- Note that the valid / invalid bits described above cannot block all illegal memory accesses, due to the internal fragmentation. Many processes do not use all the page table entries available to them.
- Addresses of the pages 0,1,2,3,4 and 5 are mapped using the page table as they are valid.
- Addresses of the pages 6 and 7 are invalid and cannot be mapped. Any attempt to access those pages will send a trap to the OS.

8.4.4 Shared Pages

- Paging systems can make it very easy to share blocks of memory, by simply duplicating page numbers in multiple page frames. This may be done with either code or data.
- If code is ***reentrant***(read-only files) that means that it does not write to or change the code in any way. More importantly, it means the code can be shared by multiple processes, so long as each has their own copy of the data and registers, including the instruction register.
- In the example given below, three different users are running the editor simultaneously, but the code is only loaded into memory (in the page frames) one time.
- Some systems also implement shared memory in this fashion.

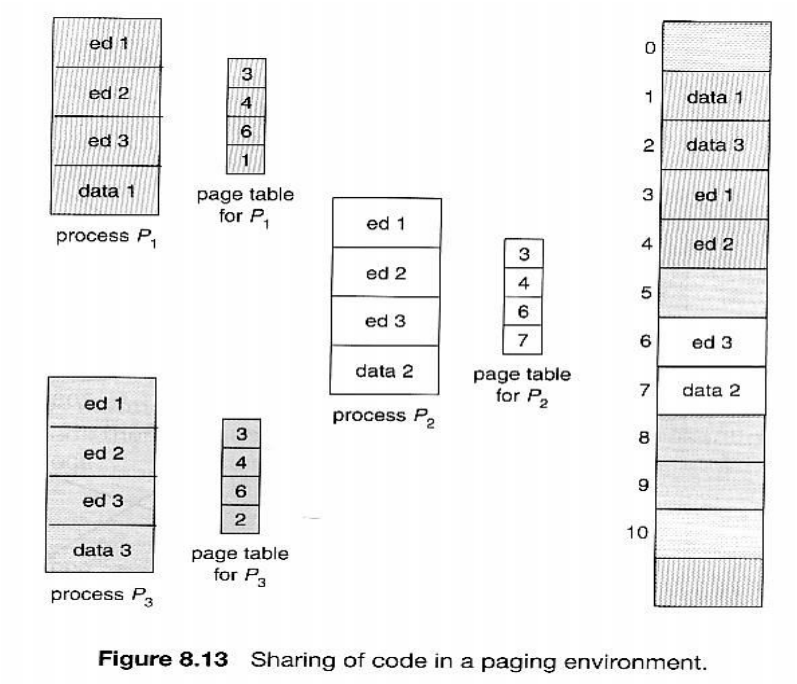


Figure 8.13 Sharing of code in a paging environment.

8.5 Structure of the Page Table

8.5.1 Hierarchical Paging

- This structure supports two or more page tables at different levels (tiers).
- Most modern computer systems support logical address spaces of 2^32 to 2^64.

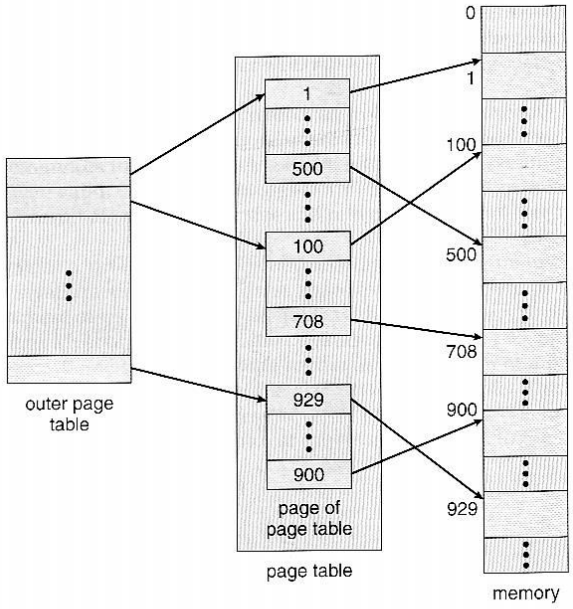
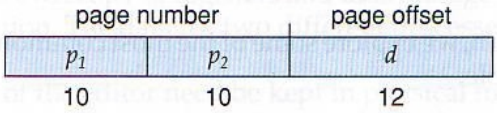
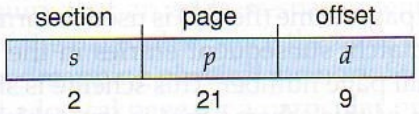


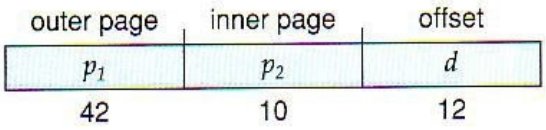
Figure 8.14 A two-level page-table scheme.

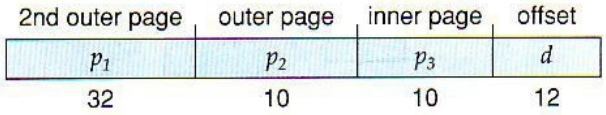


- VAX Architecture divides 32-bit addresses into 4 equal sized sections, and each page is 512 bytes, yielding an address form of:



- With a 64-bit logical address space and 4K pages, there are 52 bits worth of page numbers, which is still too many even for two-level paging. One could increase the paging level, but with 10-bit page tables it would take 7 levels of indirection, which would be prohibitively **slow memory access**. So some other approach must be used.





8.5.2 Hashed Page Tables

- One common data structure for accessing data that is sparsely distributed over a broad range of possible values is with ***hash tables***. Figure 8.16 below illustrates a ***hashed page table*** using chain-and-bucket hashing:

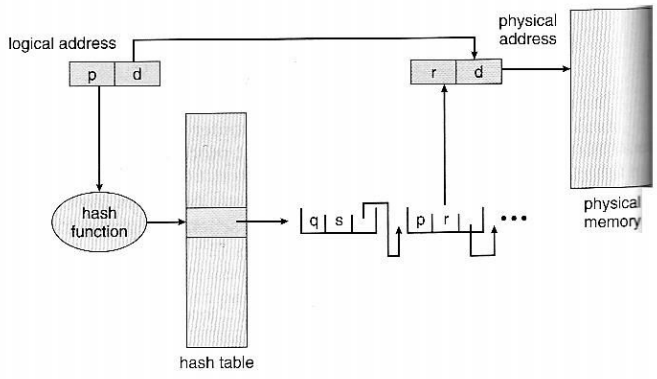


Figure 8.16 Hashed page table.

8.5.3 Inverted Page Tables

- Another approach is to use an ***inverted page table***. Instead of a table listing all of the pages for a particular process, an inverted page table lists all of the pages currently loaded in memory, for all processes. (i.e. there is one entry per ***frame*** instead of one entry per ***page***.)
- Access to an inverted page table can be slow, as it may be necessary to search the entire table in order to find the desired page .
- The 'id' of process running in each frame and its corresponding page number is stored in the page table.

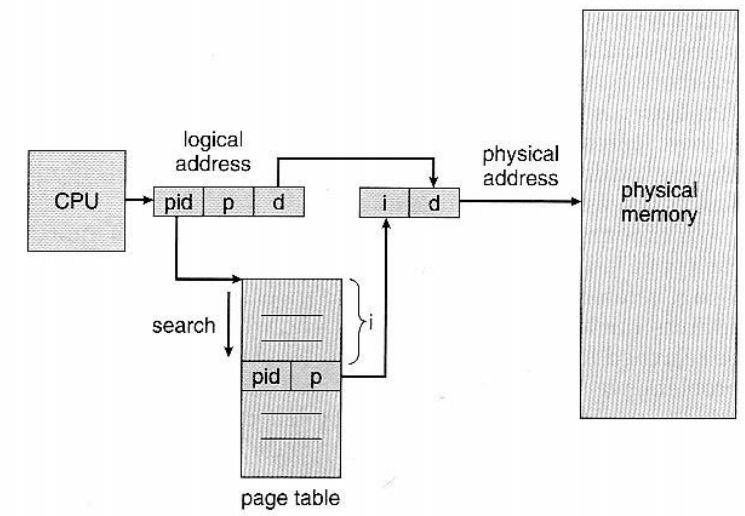


Figure 8.17 Inverted page table.

8.6 Segmentation

- Most users (programmers) do not think of their programs as existing in one continuous linear address space.
- Rather they tend to think of their memory in multiple ***segments***, each dedicated to a particular use, such as code, data, the stack, the heap, etc.
- Memory ***segmentation*** supports this view by providing addresses with a segment number (mapped to a segment base address) and an offset from the beginning of that segment.
- The logical address consists of 2 tuples:

  <segment-number, offset>

- For example, a C compiler might generate 5 segments for the user code, library code, global (static) variables, the stack, and the heap, as shown in Figure 8.18:

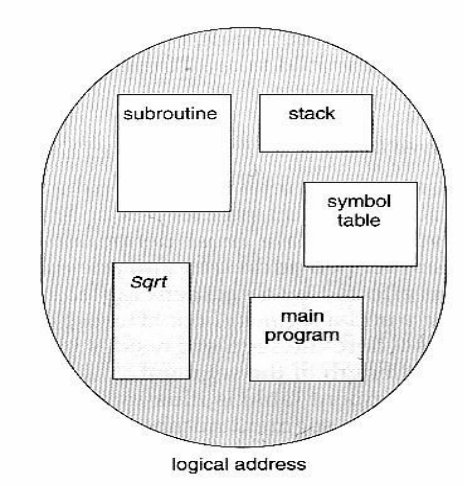


Figure 8.18 User's view of a program.

8.6.2 Hardware

- A ***segment table*** maps segment-offset addresses to physical addresses, and simultaneously checks for invalid addresses.
- Each entry in the segment table has a *segment base* and a *segment limit.* The segment base contains the starting physical address where the segment resides 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 to the segment table. The offset *d* of the logical address must be between 0 and the segment limit. When an 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 and limit register pairs.

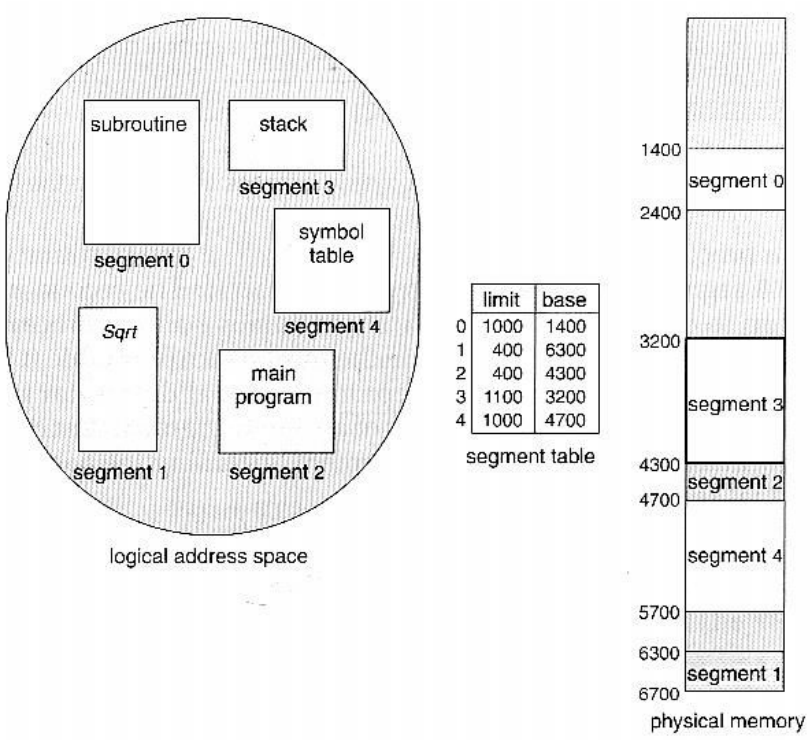


Figure 8.20 Example of segmentation.