STRUCTURE OF PAGE TABLE

1. Hierarchical Paging
2. Hashed Page Tables
3. Inverted Page Tables

Hierarchical Paging

- Most modern computer systems support a large logical address space. Then, the page table itself becomes very large. **It may not be possible to store page tables contiguously.**
- One simple solution to this problem is to **divide the page table into smaller pieces.** We can accomplish this division in several ways. One way is to use a two-level paging algorithm, in which **the page table itself is also paged**

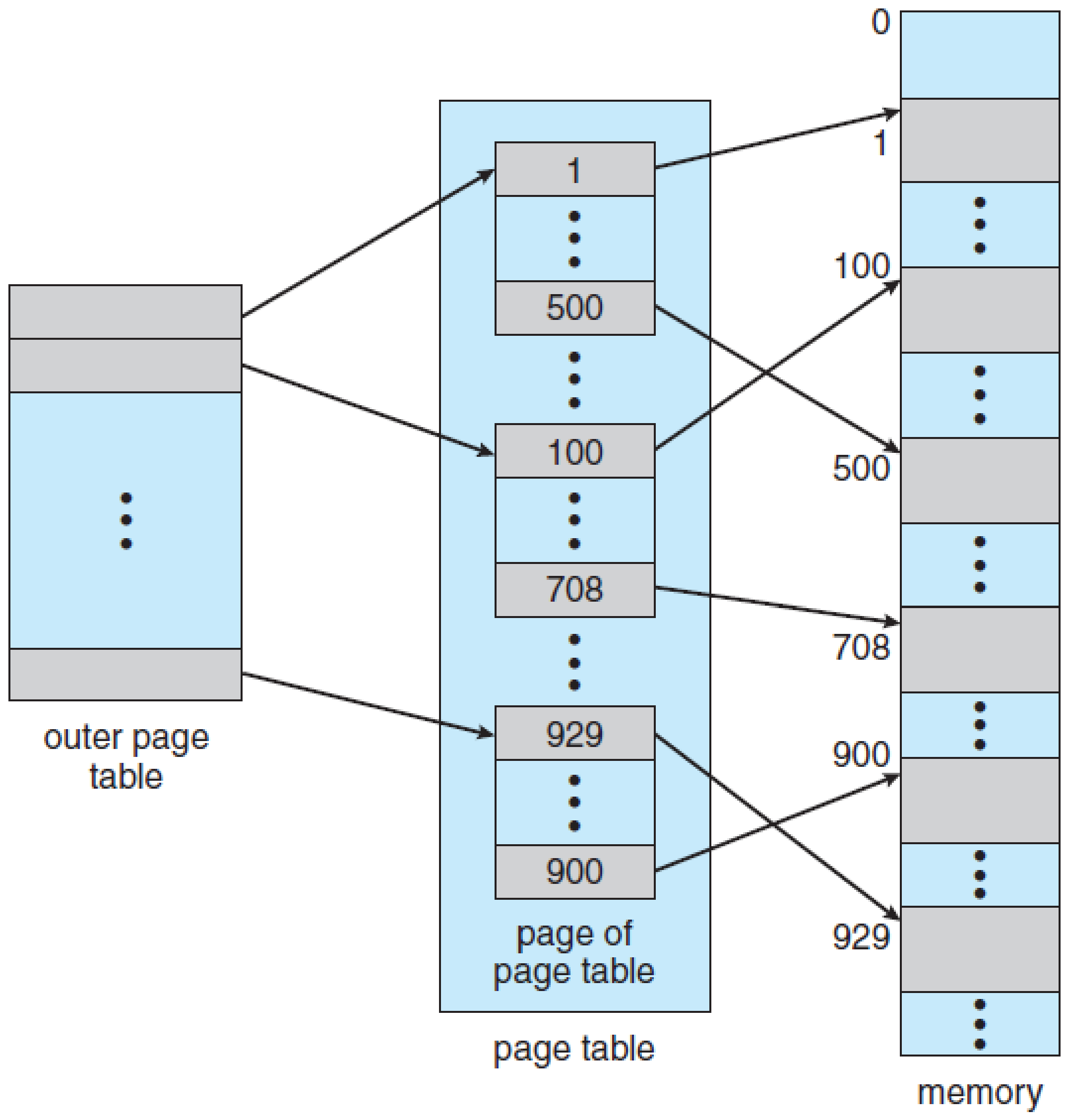
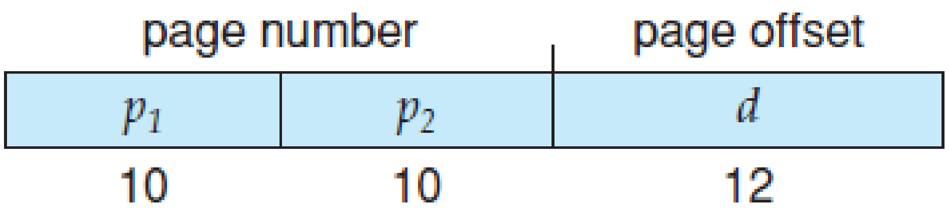


Figure 8.17 A two-level page-table scheme.

- Consider a system with 32 bits logical addressing and 12 bits page addressing. A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits.
- The page number is further divided into a 10-bit page number and a 10-bit page offset. Thus, a logical address is:



where p1 is an index into the outer page table and P2 is the displacement within the page of the outer page table.

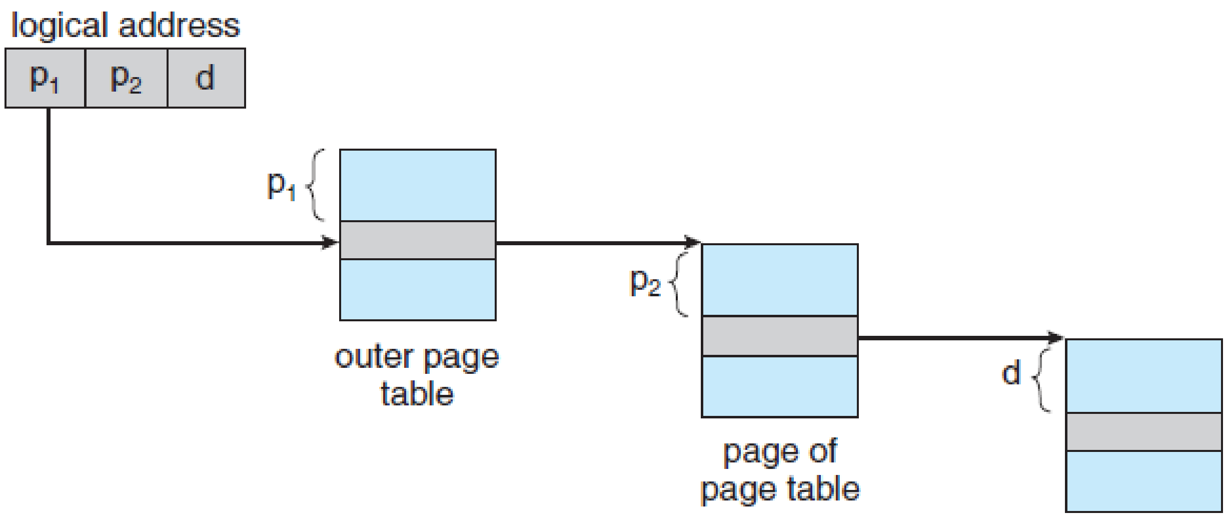


Figure 8.18 Address translation for a two-level 32-bit paging architecture.

- The next step would be a three-level or four-level paging scheme, where the second-level outer page table itself is also paged, and so on.

Hashed Page Tables

- A common approach for handling address spaces larger than 32 bits is to **use a hashed page table with the hash value being the virtual page number**.

- Each entry in the hash table contains a **linked list of elements** that hash to the same location (**to handle collisions**).
- Each element consists of **three fields:**
  (1) The virtual page number
  (2) Mapped page frame
  (3) Pointer next

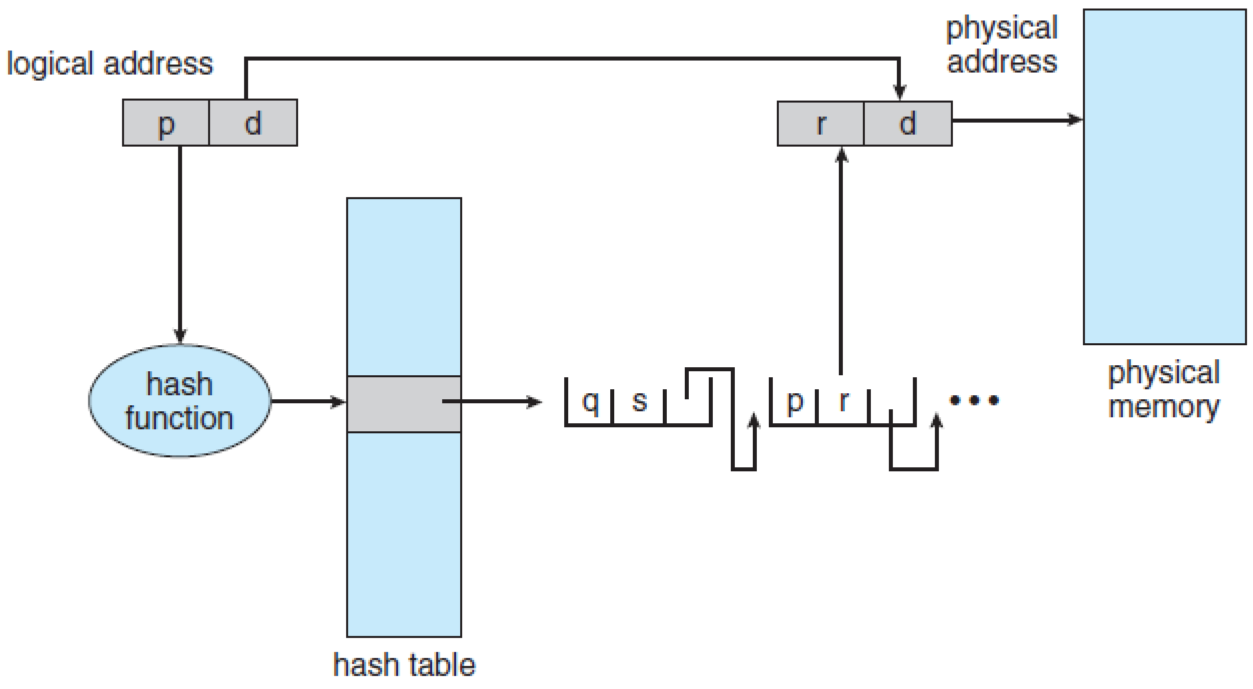


Figure 8.19 Hashed page table.

- The virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared with field1 in the first element in the linked list.
- If there is a match, the corresponding page frame (field2) is used to form the desired physical address.
- If there is no match, subsequent entries in the linked list are searched for a matching virtual page number.

Inverted Page Tables

- Usually, each process has its own page table. In a multiprogramming system, it may not be possible to keep separate page tables for each process. So a **common page table is maintained with process id associated with each entry**
- **Same logical address may be generated for various processes**
- An inverted page table has one entry for each frame of memory. Each entry consists of the virtual address of the page stored in that real memory location; with information about the process that owns the page.

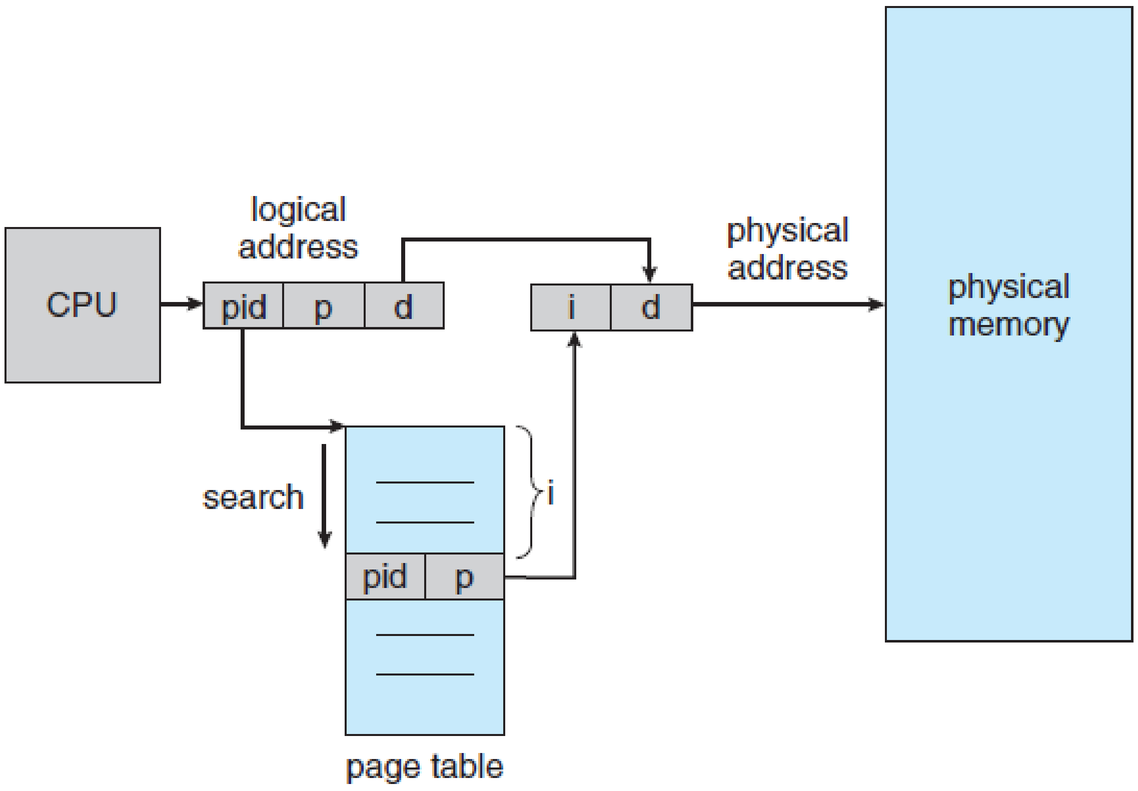


Figure 8.20 Inverted page table.

- **Each virtual address in the system consists of a triple: <process-id, page-number, offset>.**
- <process-id, page-number> is presented to the memory subsystem. The inverted page table is then searched for a match. If a match is found-say, at entry i-then the physical address <i, offset> is generated. If no match is found, then an illegal address access has been attempted.
- Systems that use inverted page tables have **difficulty in implementing shared memory.**
- Shared memory is implemented as multiple virtual addresses (one for each process) that are mapped to one physical address. This standard method cannot be used with inverted page tables; since there is only one entry for a frame and process id is associated with it.

SEGMENTATION

- **Paging does not satisfy the users' view of programs. Pages are scattered through the physical memory and they are equal sized physical partitions irrespective of user concerns**
- **Segmentation is the logic division of programs rather than physical division**

Basic Method

- A program can be made with a set of methods, procedures, or functions. It may also include various data structures: objects, arrays, stacks, variables, and so on. Each of these modules or data elements is referred by name. We talk about "the stack," "the math library," "the main program" etc without caring what addresses in memory these elements occupy.
- Each of these segments is of variable size. Elements within a segment are identified by their offset from the beginning of the segment: Eg: fifth instruction of the Sqrt ().

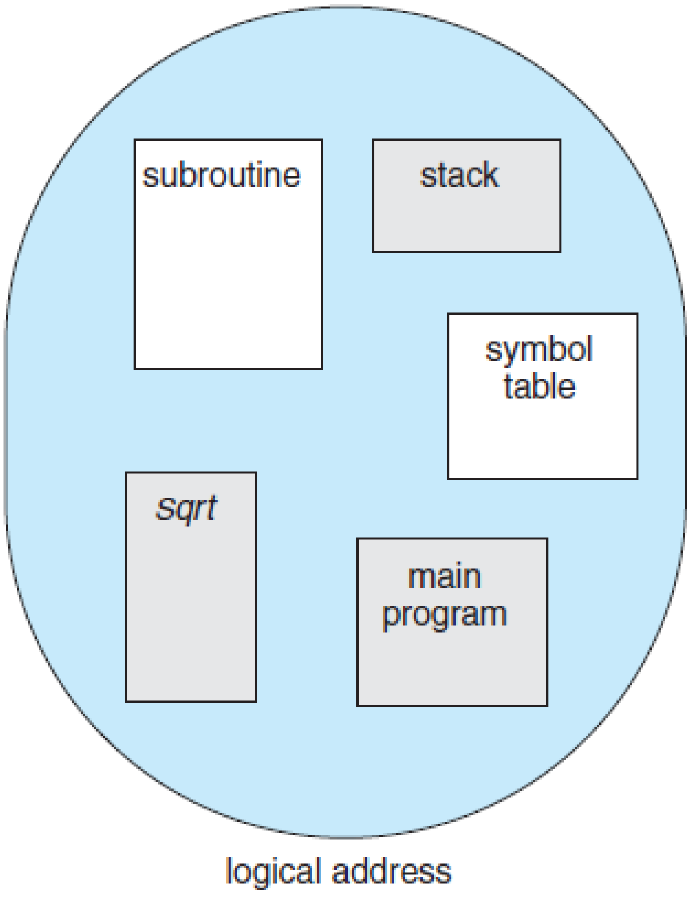


Figure 8.7 Programmer's view of a program.

- Segmentation is a memory-management scheme that supports user view of memory.
- **A logical address space is a collection of segments**.
- Each segment has a name and a length.
- The addresses specify both the segment name and the offset within the segment.
- The user therefore specifies each address by two quantities: **a segment name and an offset.**
- Contrast this scheme with the paging scheme, in which the user specifies only a single address, which is partitioned by the hardware into a page number and an offset, all invisible to the programmer.
- For simplicity of implementation, segments are numbered
- Thus, a logical address consists of a two tuple:
- <segment-number, offset>.
- Normally, when a program is compiled, the compiler automatically constructs segments reflecting the input program.
- A C compiler might create separate segments for the following:
    1. The code
    2. Global variables
    3. The heap, from which memory is allocated
    4. The stacks used by each thread
    5. The standard C library

Hardware

- User can refer to objects in the program by a two-dimensional address, but the actual physical memory is still a one-dimensional sequence of bytes.
- We must define an implementation to map two dimensional user-defined addresses into one-dimensional physical addresses. This mapping is effected by a **segment table**
- 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, and the segment limit specifies the length of the segment.

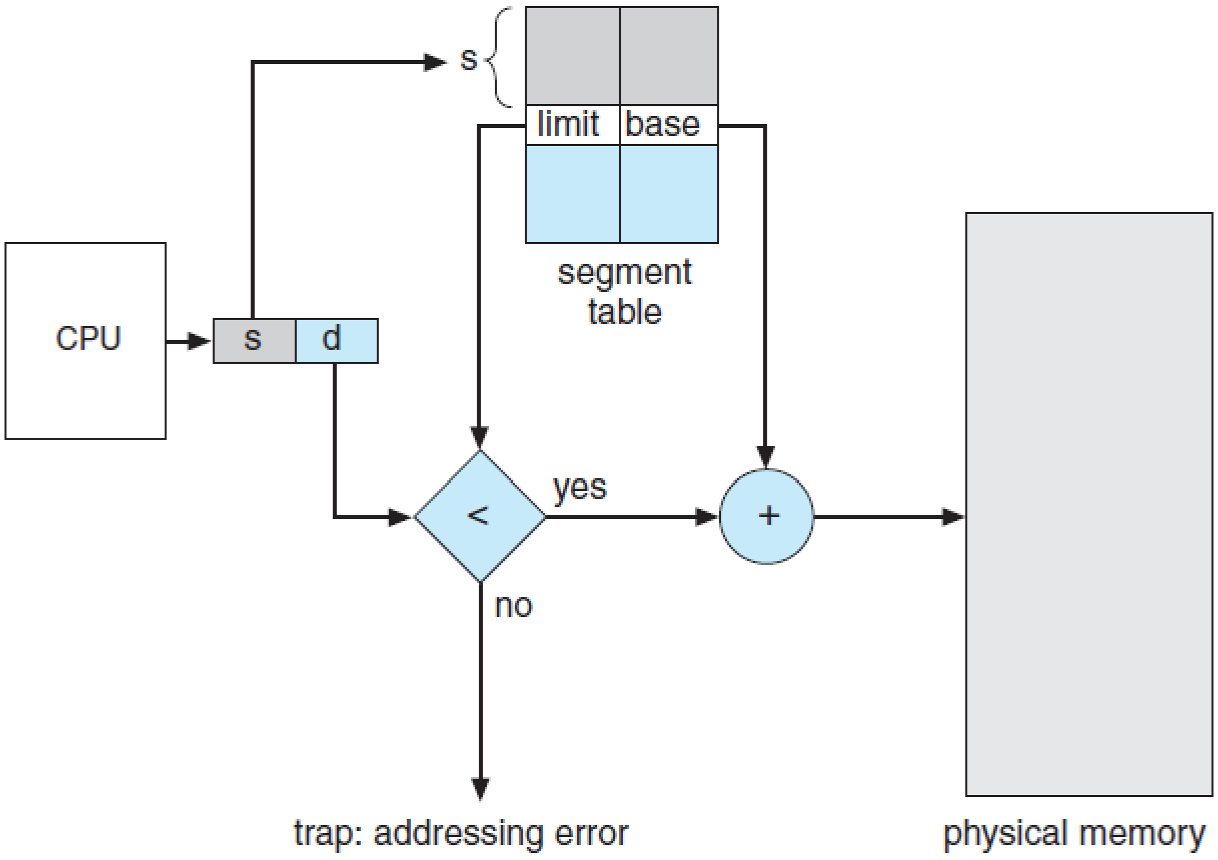
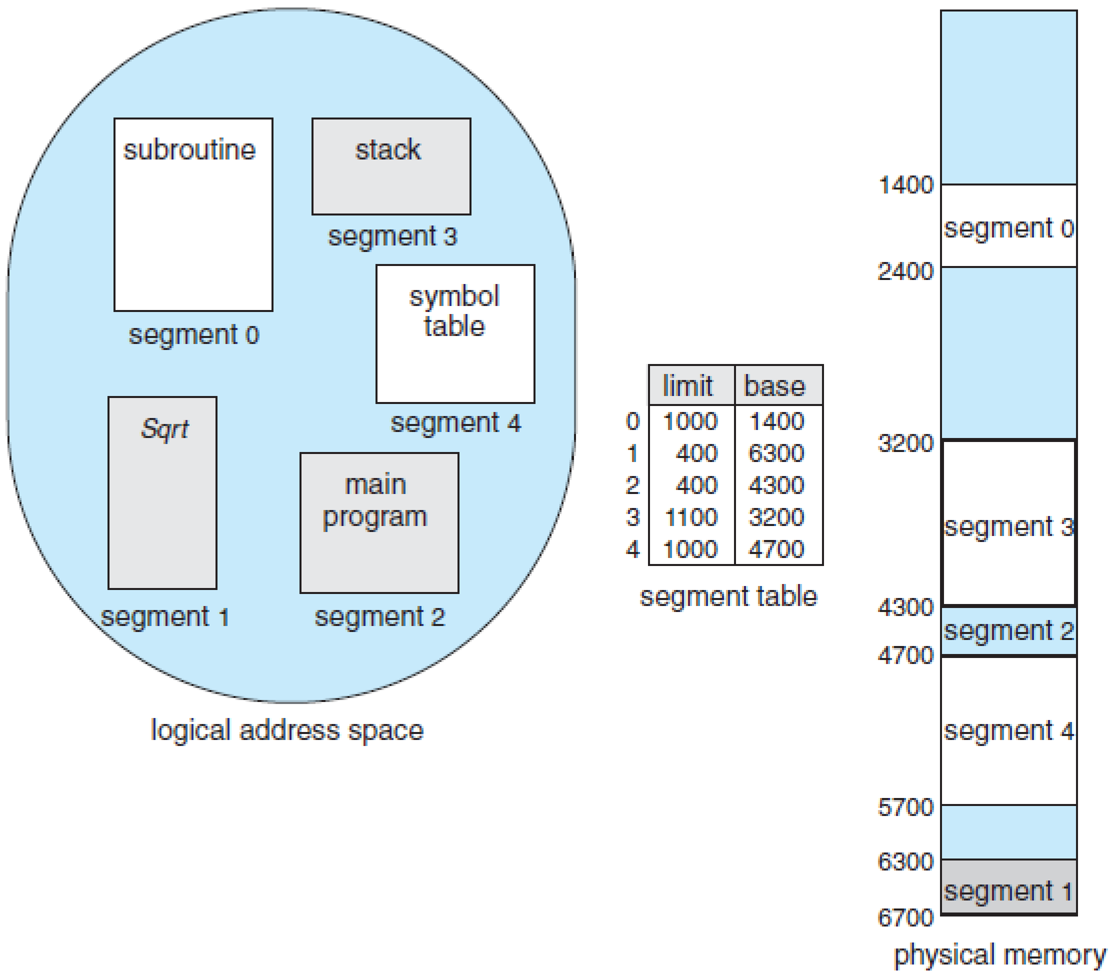


Figure 8.8 Segmentation hardware.

- 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. If it is not, we trap to the OS error
- 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-limit register pairs.



	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

Figure 8.9 Example of segmentation.

- For example, segment 2 is 400 bytes long and begins at location 4300. Thus, a reference to byte 53 of segment 2 is mapped onto location 4300 +53= 4353.

COMPARISON BETWEEN PAGING & SEGMENTATION

Paging

1. Logical Memory is divided into pages and physical memory is divided into frames
2. Division is physical
3. All pages/frames are equal sized
4. There is no external fragmentation (Since all partitions are equal)
5. Internal fragmentations may be there (Since all partitions are equal)
6. Memory manager maintains page tables
7. Page table contains page number, frame number and protection bits
8. CPU/User generate direct logical address
9. Paging does not satisfy users view

Segmentation

1. Logical Memory and physical memory are divided into segments
2. Division is logical
3. Segments are of variable size

4. There may be external fragmentation (Since all partitions are variable sized)
5. Internal fragmentations may not be there (Since partitions are done as per needs)
6. Memory manager maintains segment tables
7. Segment table contains segment limit and base
8. CPU/User generate segment name and displacement as virtual address
9. Segmentation satisfies users view

VIRTUAL MEMORY

- An entire process is to be in memory before it can execute.
- **Virtual memory is a technique that allows the execution of processes that are not completely in memory.**
- One major **advantage** of this scheme is that **programs can be larger than physical memory**.
- Virtual memory **abstracts main memory into an extremely large**, uniform array of storage, separating logical memory as viewed by the user
- Virtual memory also allows processes to share files easily and to implement shared memory.
- **Users would be able to write programs for an extremely large** virtual address space, irrespective of the memory requirement tensions.
- Virtual memory is not easy to implement

- **Dynamic loading** can help to increase the memory utilization, but it generally requires special precautions and extra work by the programmer. But another technique called demand paging is used for virtual memory management

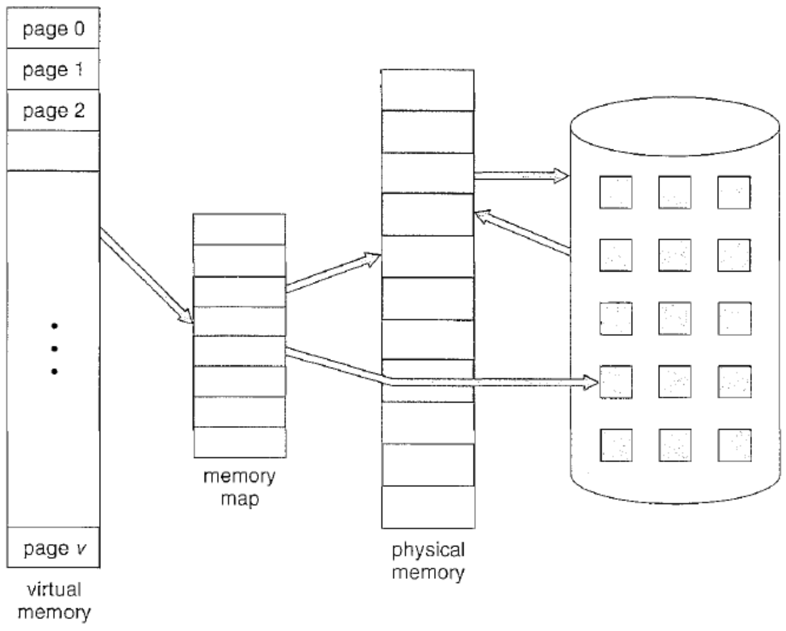


Figure 9.1 Diagram showing virtual memory that is larger than physical memory.

DEMAND PAGING

- **This strategy is to load pages only as they are needed. (on demand)**
- Programs are divided into pages and kept in storage.

- Pages are only loaded when they are demanded during program execution; pages that are never accessed are thus never loaded into physical memory.
- A **demand-paging system is similar to a paging system with swapping and dynamic loading** where processes reside in secondary memory (usually a disk).
- When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, however, we use a **lazy swapper**.
- **A lazy swapper never swaps a page into memory unless that page will be needed.** A pager and swapper work in 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.
- Instead of swapping in a whole process, the pager brings only those pages into memory.
- When **valid-invalid bit is set to "valid” the associated page is both legal and in memory**.
- If the bit is set to **"invalid” the page either is not valid** (that is, not in the logical address space of the process) **or is valid but is currently on the disk, not in memory.**
- While the process executes and accesses pages that are memory resident, execution proceeds normally.

- **But what happens if the process tries to access a page that was not brought into memory? Access to a page marked invalid causes a Page Fault.**
- The paging hardware will cause a trap to the operating system. This trap is the result of the operating system's failure to bring the desired page into memory.
- The procedure for handling this page fault is straightforward

1. Check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an 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, want to swap it in.
3. Find a free frame in memory
4. Schedule a disk operation to read the desired page into the frame.
5. Modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. Restart the instruction that was interrupted by the trap. The process can now access the page as though it had always been in memory.

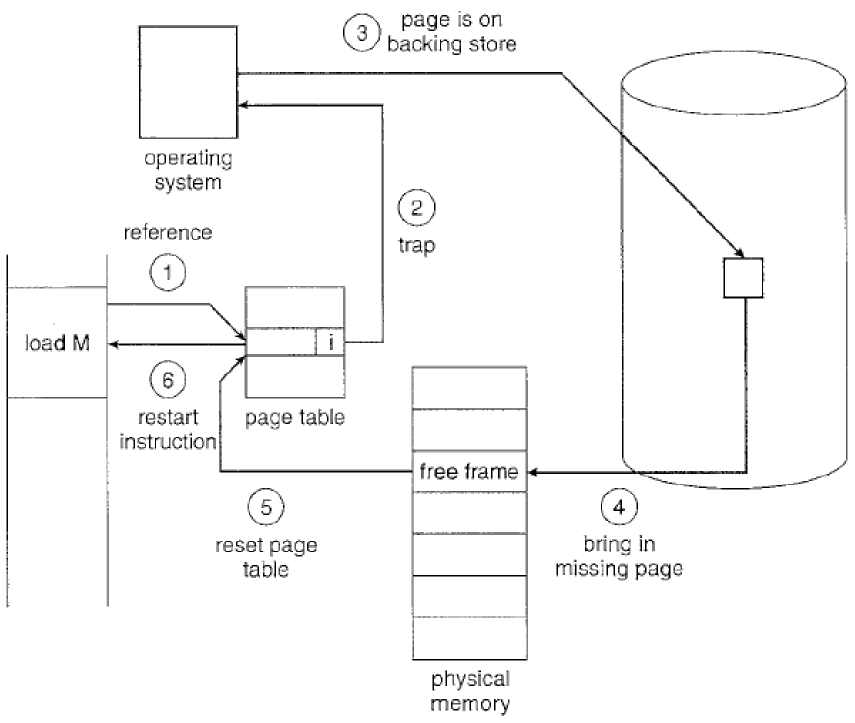


Figure 9.6 Steps in handling a page fault.

- In the extreme case, we can start executing a process with no pages in memory. Pages may be loaded to memory if and only if fault occurs. This scheme is called **pure demand paging**. ie **never bring a page into memory until it is required.**
- The hardware to support demand paging is the same as the hardware for paging and swapping:
- Secondary memory holds those pages that are not present in main memory. The secondary memory is usually a high-speed disk. **It is known as the swap device, and the section of disk used for this purpose is known as swap space**
- A crucial requirement for demand paging is the ability to restart any instruction after a page fault.

- If the page fault occurs on the instruction fetch, we can restart by fetching the instruction again.
- If a page fault occurs while we are fetching an operand, we must fetch and decode the instruction again and then fetch the operand.

Performance of Demand Paging

- Demand paging can significantly affect the performance of a computer system.
- Let's compute the **effective access time** for a demand-paged memory.
- Let the **memory-access time is denoted as ma**.
- As long as we have no page faults, the effective access time is equal to the memory access time.
- If a page fault occurs, we must first read the relevant page from disk and then access the desired location.
- Let **p be the probability of a page fault** ($0 \le p \le 1$). We would expect p to be close to zero-that is, we would expect to have only a few page faults.
- Then
  effective access time = (1 - p) x ma + p x page fault time.
- Page fault leads to 6 extra steps.
- It can be considered as 3 major steps

1. Service the page-fault interrupt.
2. Swap in the page.

3. Restart the process.

- The first and third tasks can be reduced, with careful coding, but data transfer time, step 2 (from disk to memory) is crucial.
- By statistical analysis, the transfer time is close to 8 milliseconds. (A typical hard disk has an average latency of 3 milliseconds, a seek time of 5 milliseconds, and a transfer time of 0.05 milliseconds. Thus, the total paging time is about 8 milliseconds, including hardware and software time.)
- We assume that there is no delay of accessing the devices (disk or frame or page table). If a queue of processes is waiting for the device, we have to add device-queuing time as we wait for the paging device to be free to service our request, increasing even more the time to swap.
- Average memory access time of a system is **200 nano seconds**.

  effective access time = (1 - p) x (200) + p (8 milliseconds)
  (in nano seconds) = (1 - p) x (200) + p x 8,000,000
  = 200 – p x 200 + p x 8,000,000
  = <u>**200 + 7,999,800 x p.**</u>
- Effective access time is directly proportional to the page fault rate
- If one access out of 1,000 causes a page fault, then p = 1/1000 = 0.001

- Then the effective access time is 8,199.8 nano seconds = 8.2 milli seconds
- That is instead of 200 nano seconds (for direct memory access) we need 8200 nano seconds.
- Performance degradation factor is 8200/200 = 41
- If we want to keep performance degradation not more than 10%, then max 200 + (10% of 200) = 200+20 = 220 nanoseconds can be given.
- Then 200 + 7,999,800 x p = 220
  p = 0.0000025
- ie Number of page faults = 1 / 0.0000025 = 4,00,000 per access
- That is, to manage the slowdown due to demand paging at a reasonable level, we can allow fewer than 1 page fault in 4,00,000 memory accesses