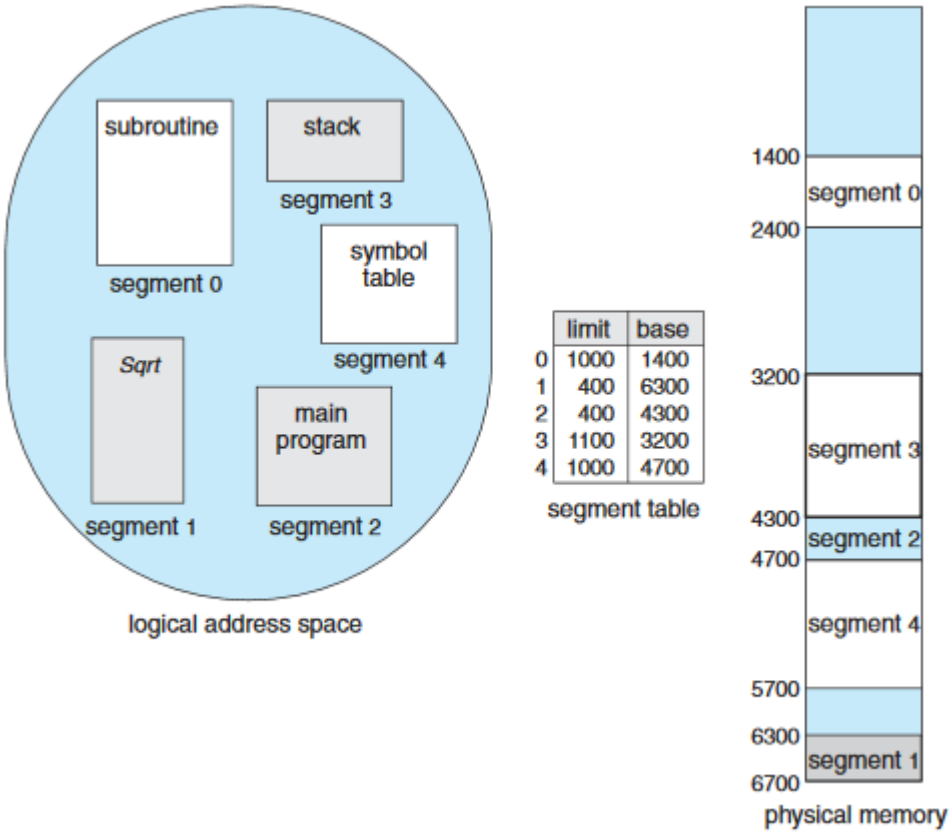
SEGMENTATION

A programmer thinks of it as a main program with a set of methods, procedures, or functions. It may also include various data structures: objects, arrays, stacks, variables, and so on and these modules or data elements is referred to by name. Segments vary in length, and the length of each is intrinsically defined by its purpose in the program. Elements within a segment are identified by their offset from the beginning of the segment. Segmentation is a memory-management scheme that supports the programmer view of memory. A logical address space is a collection of segments.

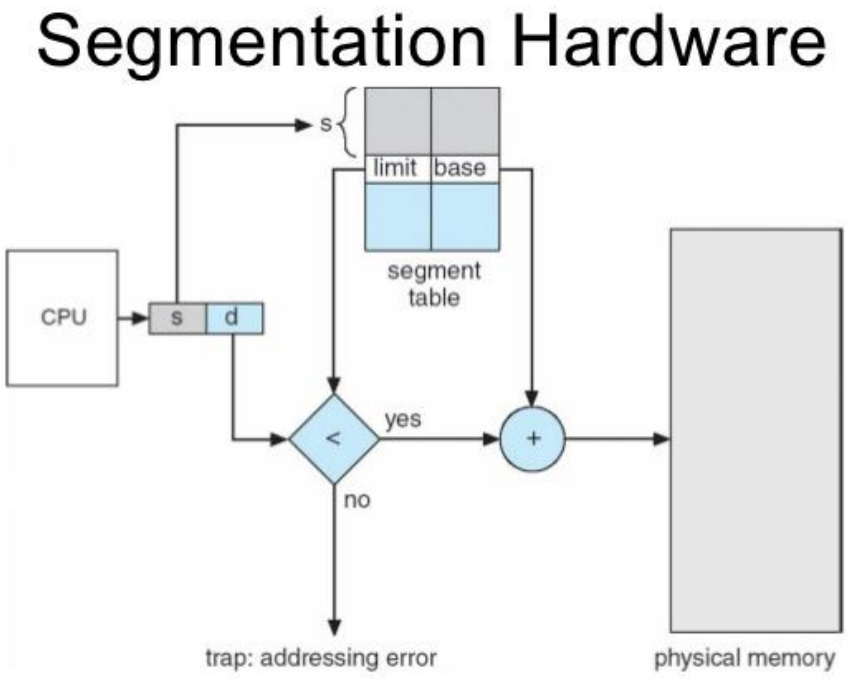


Each segment has a name and a length. The programmer therefore specifies each address by two quantities: a segment name and an offset. For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name. Thus, a logical address consists of a

two tuple: . Normally, when a program is compiled, the compiler automatically constructs segments reflecting the input program.

SEGMENTATION HARDWARE

Segmentation Hardware although the programmer can now refer to objects in the program by a two-dimensional address, the actual physical memory is still, of course, a one dimensional sequence of bytes. Thus, 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



The use of a segment table is illustrated in Figure. 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 operating system (logical addressing attempt beyond end of segment). 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.

Advantages of Segmentation

1. No internal fragmentation
2. Average Segment Size is larger than the actual page size.
3. Less overhead
4. It is easier to relocate segments than entire address space.
5. The segment table is of lesser size as compare to the page table in paging.

Disadvantages

1. It can have external fragmentation.
2. It is difficult to allocate contiguous memory to variable sized partition.
3. Costly memory management algorithms

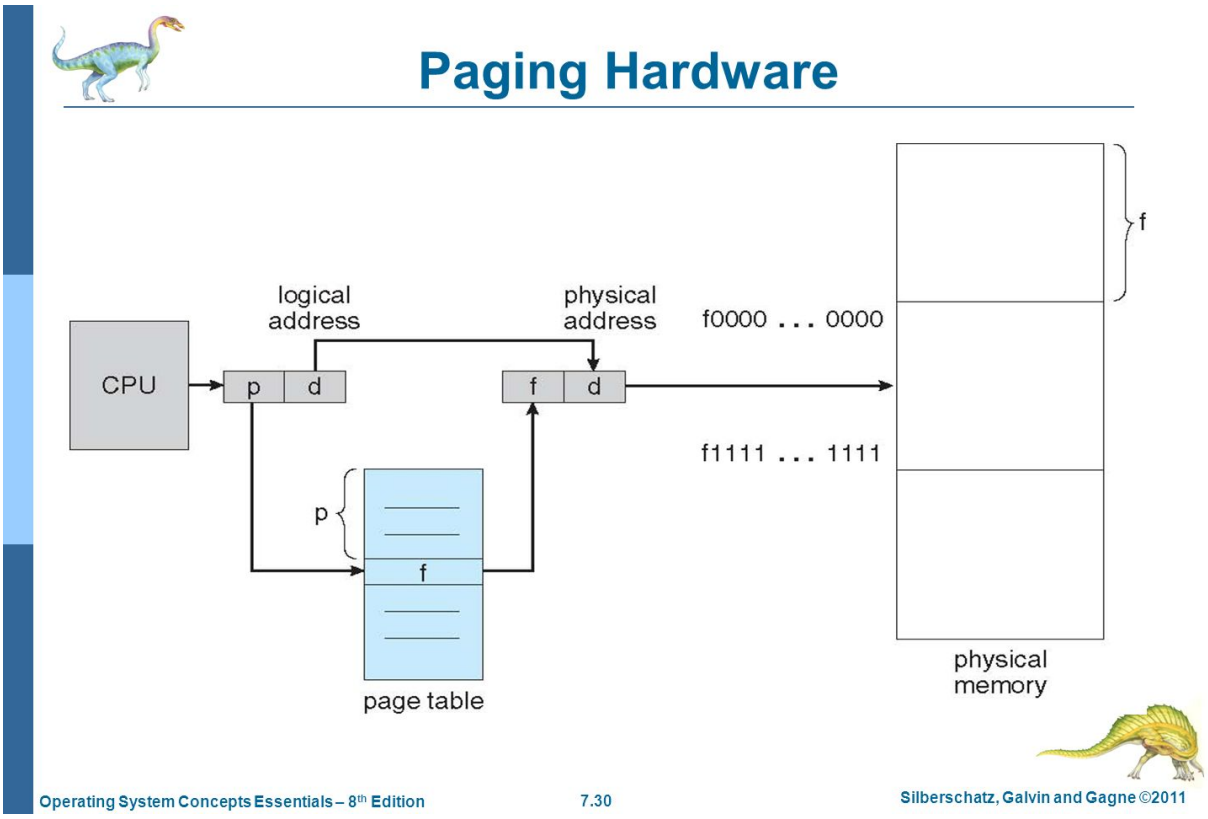
Paging:

Segmentation permits the physical address space of a process to be noncontiguous. Paging is another memory-management scheme that offers this advantage. However, paging avoids external fragmentation and the need for compaction. It also solves the considerable problem of fitting memory chunks of varying sizes onto the backing store. Paging is implemented through cooperation between the operating system and the computer hardware.

Basic Method :

The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called frames and breaking logical memory 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 their source (a file system or the backing store). The backing store is divided into fixed-sized blocks that are the same size as the memory frames or clusters of multiple frames.

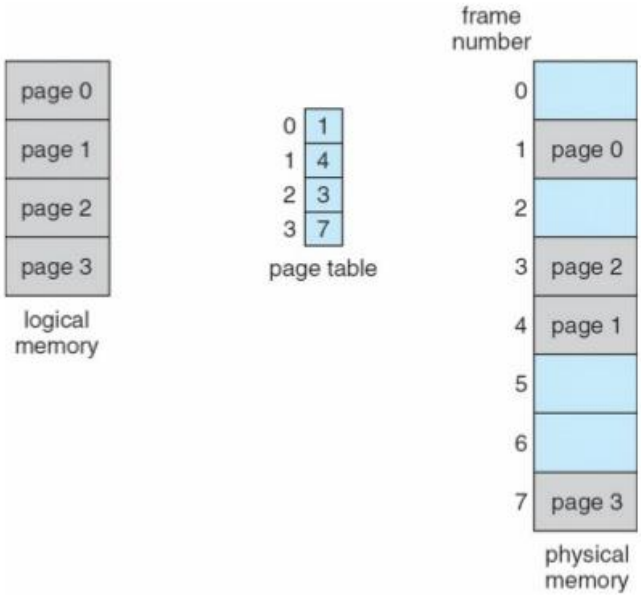
The hardware support for paging is illustrated in Figure:



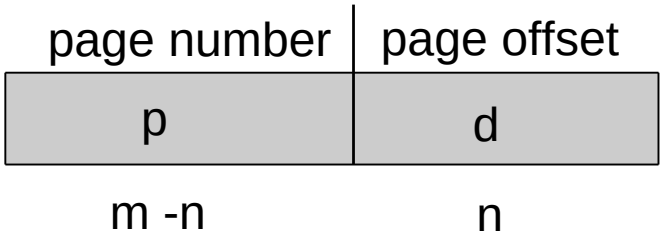
Every address generated by the CPU is divided into two parts: a page number (p) and a page. offset (d). The page number is used as an index into a page table. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical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

The paging model of memory is shown in Figure:

Paging Model of Logical and Physical Memory

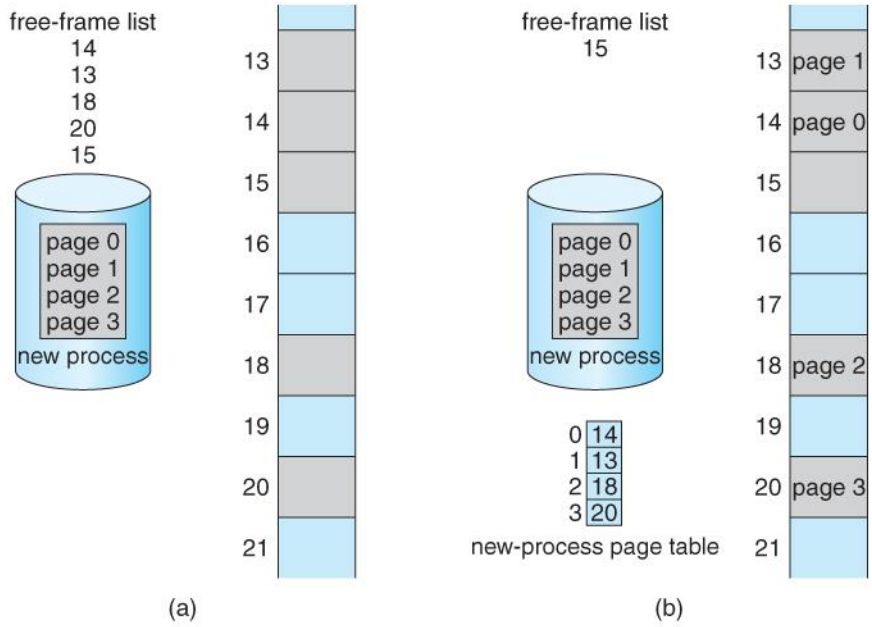


The page size (like the frame size) is defined by the hardware. The size of a page is a power of 2, varying between 512 bytes and 1 GB per page, depending on the computer architecture. The selection of a power of 2 as a page size makes the translation of a logical address into a page number and page offset particularly easy. If the size of the logical address space is 2m, and a page size is 2n bytes, then the high-order m − n bits of a logical address designate the page number, and the n low-order bits designate the page offset. Thus, the logical address is as follows:



where p is an index into the page table and d is the displacement within the page.

Paging itself is a form of dynamic relocationWhen we use a paging scheme, we have no external fragmentation: any free frame can be allocated to a process that needs it. However, we may have some internal fragmentation. If process size is independent of page size, we expect internal fragmentation to average one-half page per process.



Free frames (a) before allocation and (b) after When a process arrives in the system to be executed, its size, expressed in 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. If n frames are available, they are allocated to this arriving process. The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process. The next page is loaded into another frame, its frame number is put into the page table, and so on.

An important aspect of paging is the clear separation between the programmer's view of memory and the actual physical memory. The programmer views memory as one single space, containing only this one

program. The difference between the programmer's view of memory and the actual physical memory is reconciled by the address-translation hardware. The logical addresses are translated into physical addresses. This mapping is hidden from the programmer and is controlled by the operating system.

Since the operating system is managing physical memory, it must be aware of the allocation details of physical memory—which frames are allocated, which frames are available, how many total frames there are, and so on. This information is generally kept in a data structure called a frame table. The frame table has one entry for each physical page frame, indicating whether the latter is free or allocated and, if it is allocated, to which page of which process or processes.

The operating system maintains a copy of the page table for each process, just as it maintains a copy of the instruction counter and register contents. This copy is used to translate logical addresses to physical addresses whenever the operating system must map a logical address to a physical address manually. Paging therefore increases the context-switch time.

Hardware Support:

Each operating system has its own methods for storing page tables. The hardware implementation of the page table can be done in several ways. In the simplest case, the page table is implemented as a set of dedicated registers. These registers should be built with very high-speed logic to make the paging-address translation efficient. Every access to memory must go through the paging map, so efficiency is a major consideration. The CPU dispatcher reloads these registers, just as it reloads the other registers.

The page table is kept in main memory, and a page-table base register (PTBR) points to the page table. Changing page tables requires changing only this one register, substantially reducing context-switch time. The problem with this approach is the time required to access a user memory location. The standard solution to this problem is to use a special, small, fastlookup hardware cache

called a translation look-aside buffer (TLB). The TLB is associative, high-speed memory. Each entry in the TLB consists of two parts: a key (or tag) and a value. When the associative memory is presented with an item, the item is compared with all keys simultaneously. If the item is found, the corresponding value field is returned. The search is fast. If the page number is not in the TLB (known as a TLB miss), a memory reference to the page table must be made. Depending on the CPU, this may be done automatically in hardware or via an interrupt to the operating system. When the frame number is obtained, we can use it to access memory.

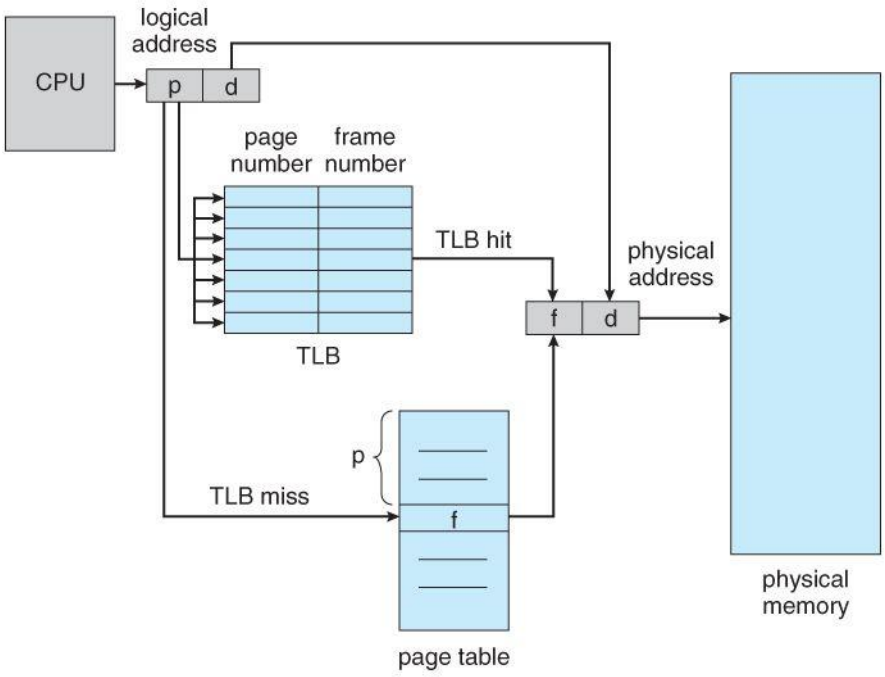


Figure: Paging hardware with TLB

In addition, we add the page number and frame number to the TLB, so that they will be found quickly on the next reference. The percentage of times that the page number of interest is found in the TLB is called the hit ratio.

Protection:

Memory protection in a paged environment is accomplished by protection bits associated with each frame. Normally, these bits are kept in the page table. One bit can define a page to be read–write or read-only. Every reference to memory goes through the page table to find the correct frame number. An attempt to

write to a read-only page causes a hardware trap to the operating system (or memory-protection violation).

One additional bit is generally attached to each entry in the page table: a valid–invalid bit. When this bit is set to valid, the associated page is in the process's logical address space and is thus a legal (or valid) page. When the bit is set to invalid, the page is not in the process's logical address space. Illegal addresses are trapped by use of the valid–invalid bit.

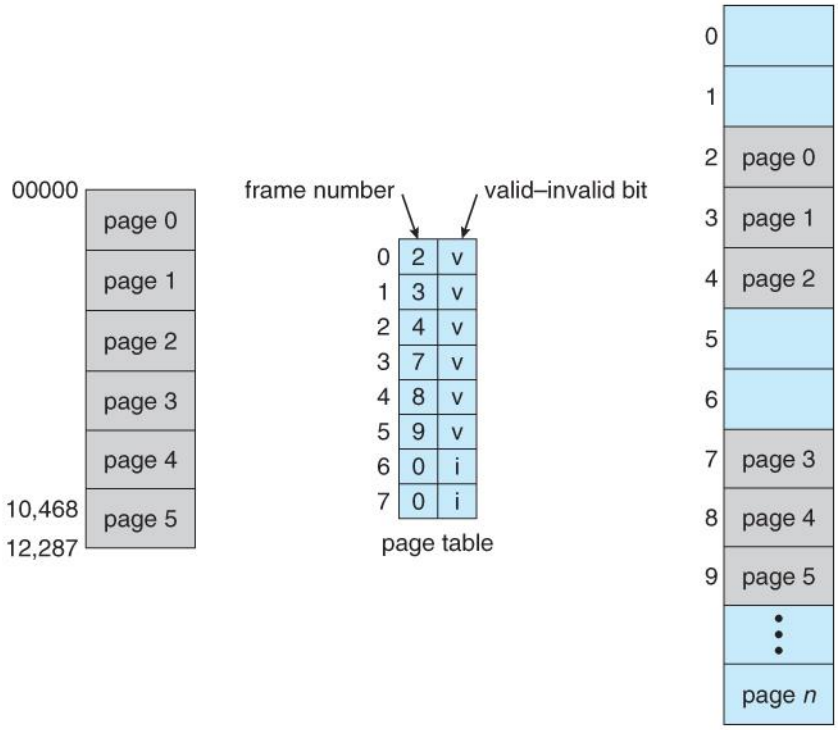


Figure: Valid (v) or invalid (i) bit in a page table.

Shared Pages: An advantage of paging is the possibility of sharing common code. This consideration is particularly important in a time-sharing environment.

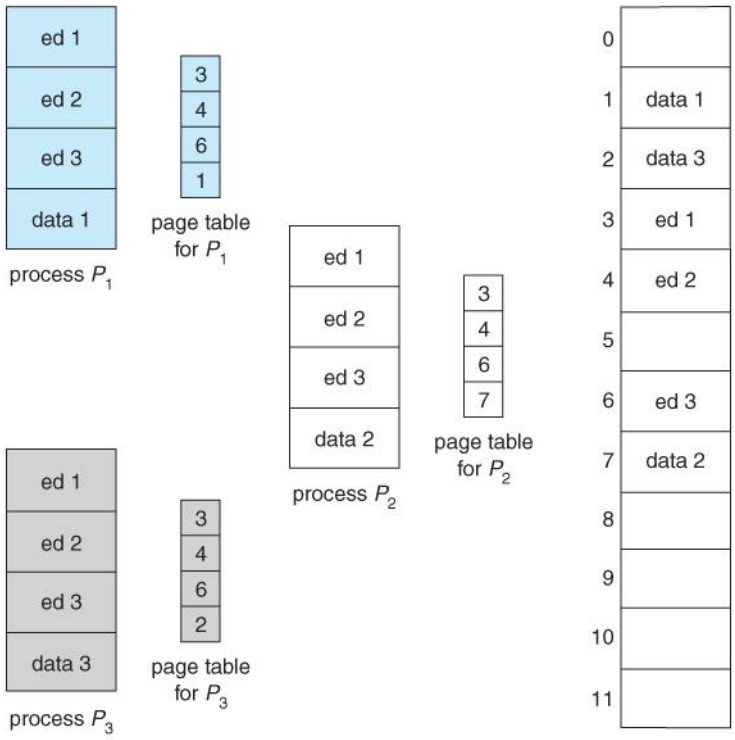


Figure: Sharing of code in a paging environment.