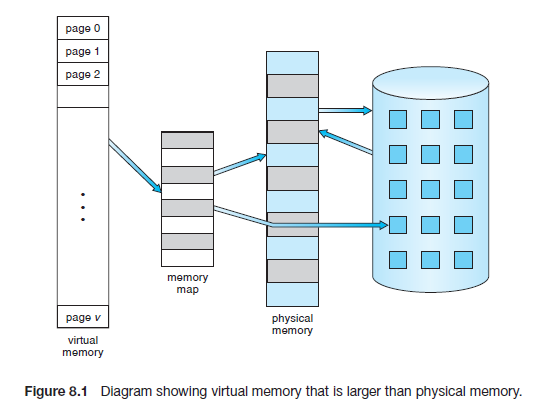
**Virtual Memory**

Virtual memory is a technique that allows the execution of process that may not be completely in memory. The main visible advantage of this scheme is that programs can be larger than physical memory.

Virtual memory is the separation of user logical memory from physical memory this separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available see in Fig 8.1.



Following are the situations, when entire program is not required to load fully

• Programs often have code to handle unusual error conditions. Since these errors seldom, if ever, occur in practice, this code is almost never executed.

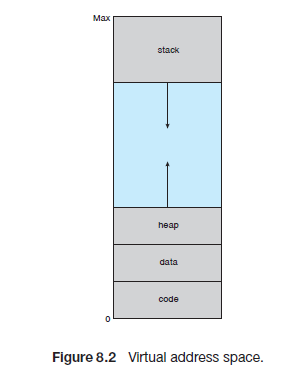
• Arrays, lists, and tables are often allocated more memory than they actually need. An array may be declared 100 by 100 elements, even though it is seldom larger than 10 by 10 elements. An assembler symbol table may have room for 3,000 symbols, although the average program has less than 200 symbols.

• Certain options and features of a program may be used rarely. For instance, the routines on U.S. government computers that balance the budget have not been used in many years.

The ability to execute a program that is only partially in memory would confer many benefits:

* A program would no longer be constrained by the amount of physical memory that is available. Users would be able to write programs for an extremely large virtual address space, simplifying the programming task.
* Because each user program could take less physical memory, more programs could be run at the same time, with a corresponding increase in CPU utilization and throughput but with no increase in response time or turnaround time.
* Less I/O would be needed to load or swap user programs into memory, so each user program would run faster.

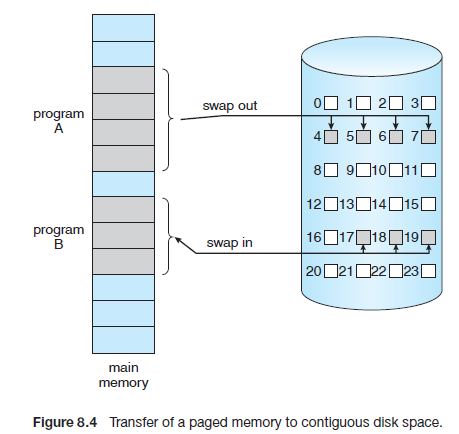
The virtual address space of a process refers to the logical (or virtual) view of how a process is stored in memory. Typically, this view is that a process begins at a certain logical address—say, address 0—and exists in contiguous memory, as shown in Figure 8.2.



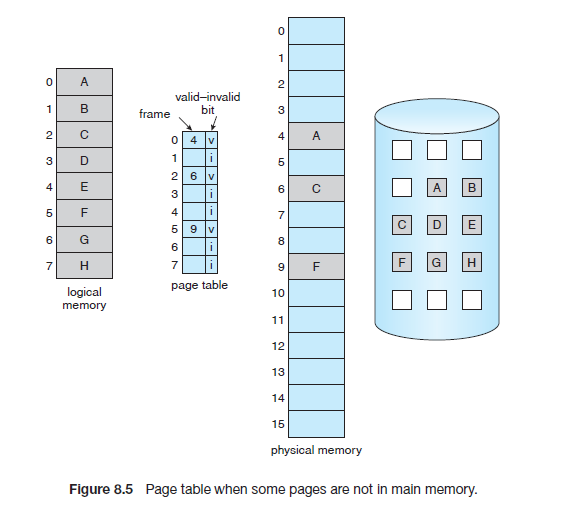
In Figure 8.2 that we allow the heap to grow upward in memory as it is used for dynamic memory allocation. Similarly, we allow the stack to grow downward in memory through successive function calls. The large blank space (or hole) between the heap and the stack is part of the virtual address space but will require actual physical pages only if the heap or stack grows. Virtual address spaces that include holes are known as sparse address spaces. Using a sparse address space is beneficial because the holes can be filled as the stack or heap segments grow or if we wish to dynamically link libraries (or possibly other shared objects) during program execution.

**Demand Paging:** Virtual memory is commonly implemented by demand paging. The strategy is to load pages only as they are needed. This technique is known as demand paging and is commonly used in virtual memory systems.

A demand-paging system is similar to a paging system with swapping see in Figure 8.4, 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. Since we are now viewing a process as a sequence of pages, rather than as one large contiguous address space, use of the term swapper is technically incorrect. A swapper manipulates entire processes, whereas a pager is concerned with the individual pages of a process. We thus use pager, rather than swapper, in connection with demand paging.

****

Hardware support is required to distinguish between those pages that are in memory and those pages that are on the disk using the valid-invalid bit scheme. When this 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. The page-table entry for a page that is brought into memory is set as usual, but the page-table entry for a page that is not currently in memory is either simply marked invalid or contains the address of the page on disk. This situation is depicted in Figure 8.5.



Access to a page marked invalid causes a page-fault trap. 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 shown in figure 8.6.

**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 an invalid memory access.

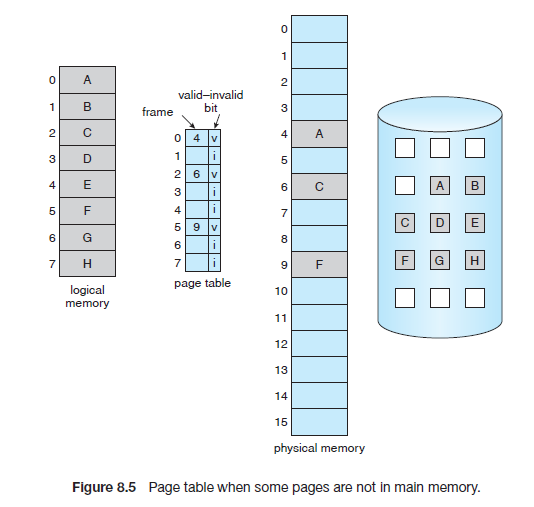
2. If the reference was invalid, we terminate the process. If it was valid, but we have not yet brought in that page, we now page it in.

3. We find a free frame (by taking one from the free-frame list, for example).

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 trap. The process can now access the page as though it had always been in memory.



The hardware to support demand paging is the same as the hardware for paging and swapping:

• Page table. This table has the ability to mark an entry invalid through a valid–invalid bit or a special value of protection bits.

• Secondary memory. This memory holds those pages that are not present in main memory. The secondary memory is usually a high-speed disk.

**Page Replacement Algorithms:**

There are many different page replacement algorithms. We evaluate an algorithm by running it on a particular string of memory reference and computing the number of page faults. The string of memory references is called **reference string.** Reference strings are generated artificially or by tracing a given system and recording the address of each memory reference. The latter choice produces a large number of data.

1. For a given page size we need to consider only the page number, not the entire address.

2. If we have a reference to a page p, then any immediately following references to page p will never cause a page fault. Page p will be in memory after the first reference; the immediately following references will not fault.

Eg:- consider the address sequence 0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103, 0104, 0104, 0101, 0609, 0102, 0105

At 100 bytes per page, this sequence is reduced to the following reference

string:

1, 4, 1, 6,1, 6, 1, 6, 1, 6, 1

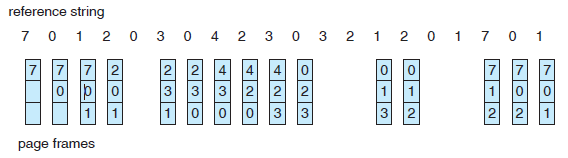
To determine the number of page faults for a particular reference string and page replacement algorithm, we also need to know the number of page frames available. As the number of frames available increase, the number of page faults will decrease.

**FIFO Page Replacement:**

A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. We can create a FIFO queue to hold all pages in memory.

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 for a memory with three frames.



* The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good.
* Notice that the number of faults for four frames (ten) is greater than the number of faults for three frames (nine)! This most unexpected result is known as **Belady’s anomaly**: for some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increases. We would expect that giving more memory to a process would improve its performance. In some early research, investigators noticed that this assumption was not always true. Belady’s anomaly was discovered as a result.

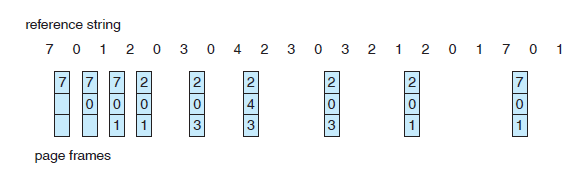
**Optimal Page Replacement:**

An optimal page-replacement algorithm, which has the lowest page-fault rate of all algorithms and will never suffer from Belady’s anomaly. It has been called OPT or MIN. It is simply this:

Replace the page that will not be used

for the longest period of time.

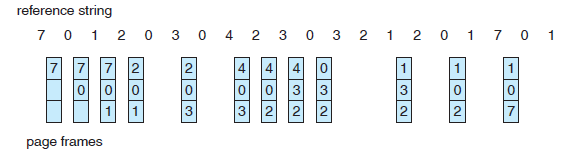
Consider a reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 for a memory with three frames.



The optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

**LRU Page Replacement:**

The FIFO algorithm uses the time when a page was brought into memory; the OPT algorithm uses the time when a page is to be used. In LRU replace the page that has not been used for the longest period of time. LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses that page that has not been used for the longest period of time.

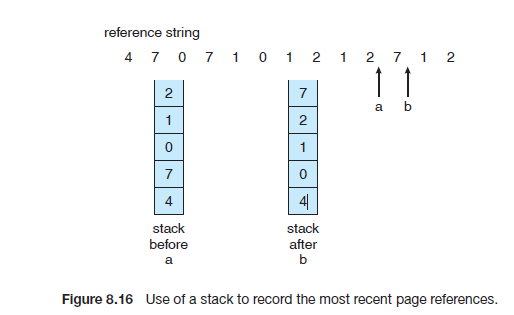


The LRU policy is often used as a page-replacement algorithm and is considered to be good. The major problem is how to implement LRU replacement. An LRU page-replacement algorithm may require substantial hardware assistance. The problem is to determine an order for the frames defined by the time of last use. Two implementations are feasible:

• Counters. In the simplest case, we associate with each page-table entry a time-of-use field and add to the CPU a logical clock or counter. The clock is incremented for every memory reference. Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page. In this way, we always have the “time” of the last reference to each page. We replace the page with the smallest time value. This scheme requires a search of the page table to find the LRU page and a write to memory (to the time-of-use field in the page table) for each memory access. The times must also be maintained when page tables are changed (due to CPU scheduling). Overflow of the clock must be considered.

• Stack. Another approach to implementing LRU replacement is to 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 most recently used page is always at the top of the stack and the least recently used page is always at the bottom (Figure 8.16). Because entries must be removed from the middle of the stack, it is best to implement this approach by using a doubly linked list with a head pointer and a tail pointer. Removing a page and putting it on the top of the stack then requires changing six pointers at worst. Each update is a little more expensive, but there is no search for a replacement; the tail pointer points to the bottom of the stack, which is the LRU page. This approach is particularly appropriate for software or

microcode implementations of LRU replacement.



* Like optimal replacement, LRU replacement does not suffer from Belady’s anomaly.

**LRU-Approximation Page Replacement:**

Some systems provide no hardware support, and other page replacement algorithms (such as a FIFO algorithm) must be used. Many systems provide some help, however, in the form of a reference bit. The reference bit for a page is set by the hardware whenever that page is referenced (either a read or a write to any byte in the page). Reference bits are associated with each entry in the page table. Initially, all bits are cleared (to 0) by the operating system. As a user process executes, the bit associated with each page referenced is set (to 1) by the hardware.

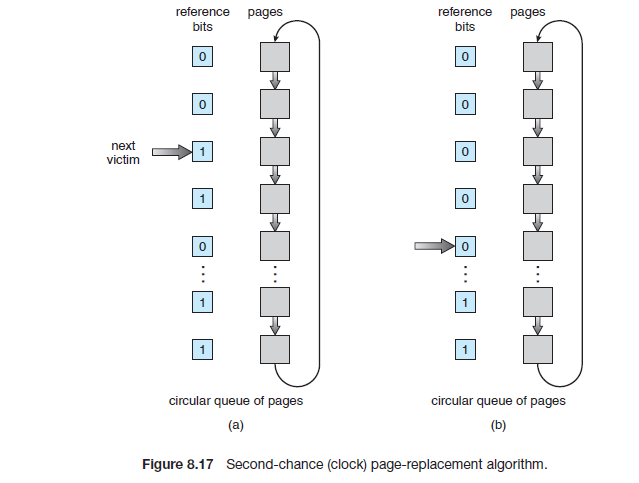
**Additional-Reference-Bits Algorithm:**

The operating system shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right by 1 bit and discarding the low-order bit. These 8-bit shift registers contain the history of page use for the last eight time periods. If the shift register contains 00000000, then the page has not been used for eight time periods; a page that is used at least once each period would have a shift register value of 11111111.

**Second-Chance Algorithm:**

The basic algorithm of second-chance replacement is a FIFO replacement algorithm. When a page has been selected, however, we inspect its reference bit. If the value is 0, we proceed to replace this page; but if the reference bit is set to 1, we give the page a second chance and move on to select the next FIFO page. When a page gets a second chance, its reference bit is cleared, and its arrival time is reset to the current time. Thus, a page that is given a second chance will not be replaced until all other pages have been replaced (or given second chances). In addition, if a page is used often enough to keep its reference bit set, it will never be replaced.

One way to implement the second-chance algorithm (sometimes referred to as the clock algorithm) is as a circular queue. A pointer (that is, a hand on the clock) indicates which page is to be replaced next. When a frame is needed, the pointer advances until it finds a page with a 0 reference bit. As it advances, it clears the reference bits (Figure 8.17). Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position. Notice that, in the worst case, when all bits are set, the pointer cycles through the whole queue, giving each page a second chance.



**Enhanced Second-Chance Algorithm:**

We can enhance the second-chance algorithm by considering the reference bit and the modify bit as an ordered pair. With these two bits, we have the following four possible classes:

1. (0, 0) neither recently used nor modified—best page to replace

2. (0, 1) not recently used but modified—not quite as good, because the

page will need to be written out before replacement

3. (1, 0) recently used but clean—probably will be used again soon

4. (1, 1) recently used and modified—probablywill be used again soon, and

the page will be need to be written out to disk before it can be replaced

**Counting-Based Page Replacement:**

There are many other algorithms that can be used for page replacement. For example, we can keep a counter of the number of references that have been made to each page and develop the following two schemes.

• The least-frequently-used (LFU) page-replacement algorithm requires that the page with the smallest count be replaced. This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again.

• The most-frequently-used (MFU) page-replacement algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

The implementation of these algorithms is expensive.