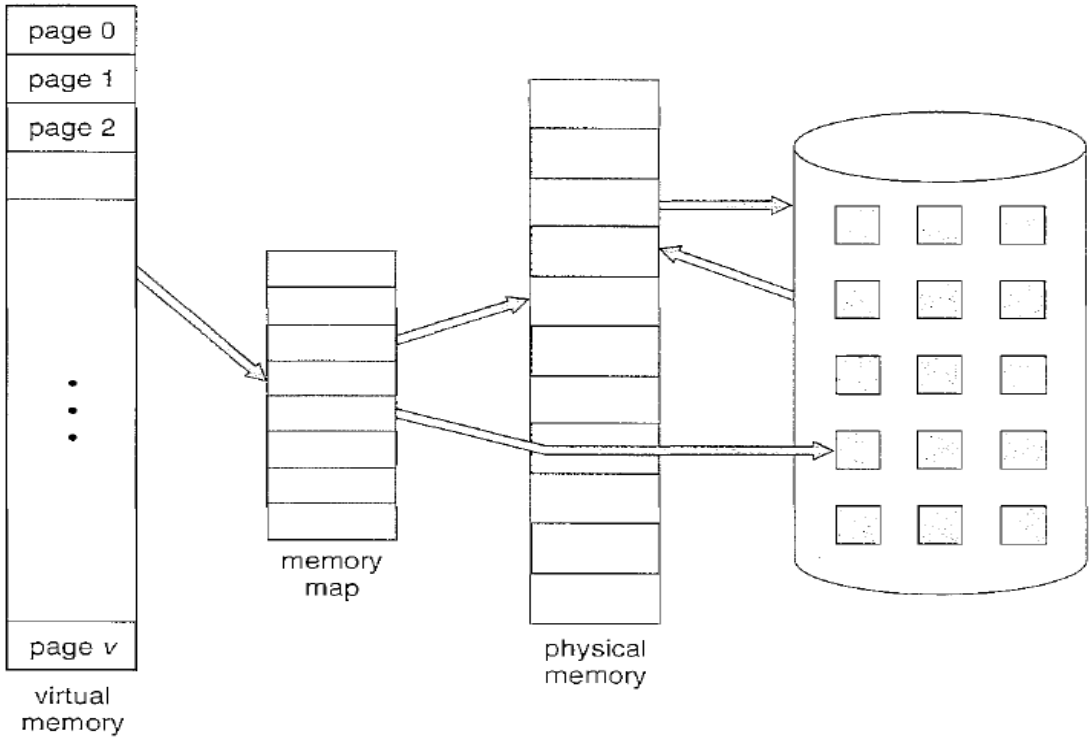
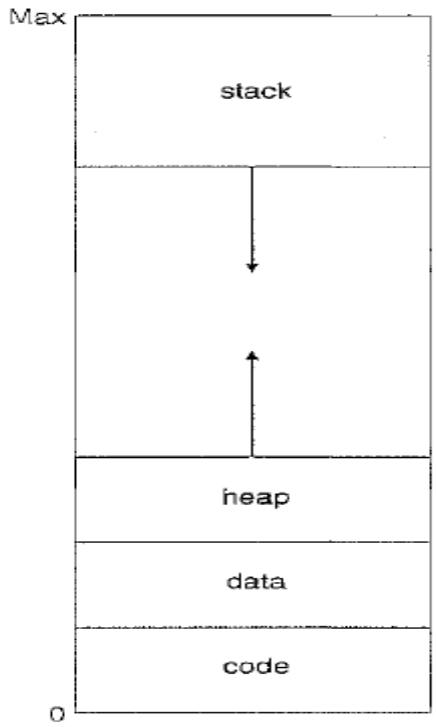
VIRTUAL MEMORY

- 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.
- Further, virtual memory abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the user from physical memory.
- Virtual Memory involves the separation of logical memory as perceived by users from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available (Shown in Figure).
- Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available; she can concentrate instead on the problem to be programmed.



- 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.



DEMAND PAGING

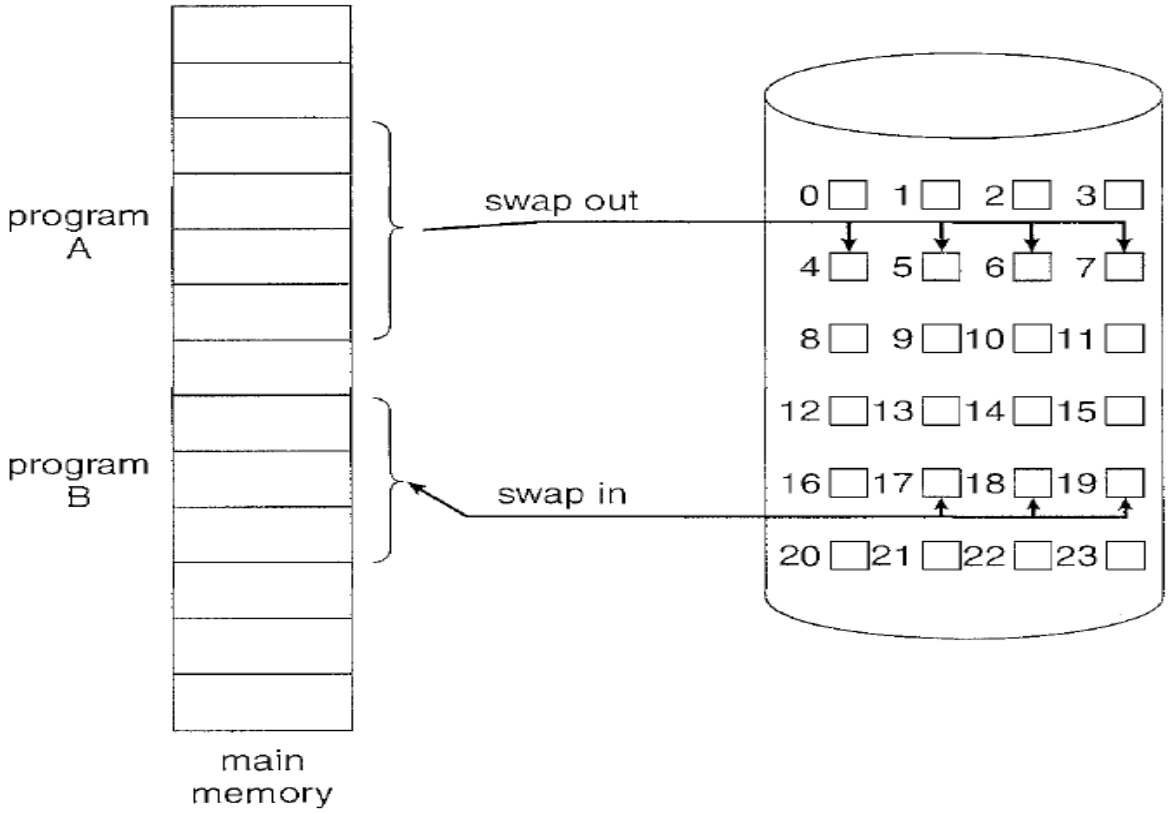
- Consider how an executable program might be loaded from disk into memory. One option is to load the entire program in physical memory at program execution time. However, a problem with this approach is that we may not initially *need* the entire program in memory.

- Suppose a program starts with a list of available options from which the user is to select. Loading the entire program into memory results in loading the executable code for *all* options, regardless of whether an option is ultimately selected by the user or not. An alternative 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.

- With demand-paged virtual memory, 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 (shown in figure below) where processes reside in secondary memory (usually a disk). When we want to execute a process, we swap it into memory.

- Since we are now viewing a process as a sequence of pages, rather than as one large contiguous address space. We thus use *pager,* rather than *swapper (which is used in case of process),* in connection with demand paging.

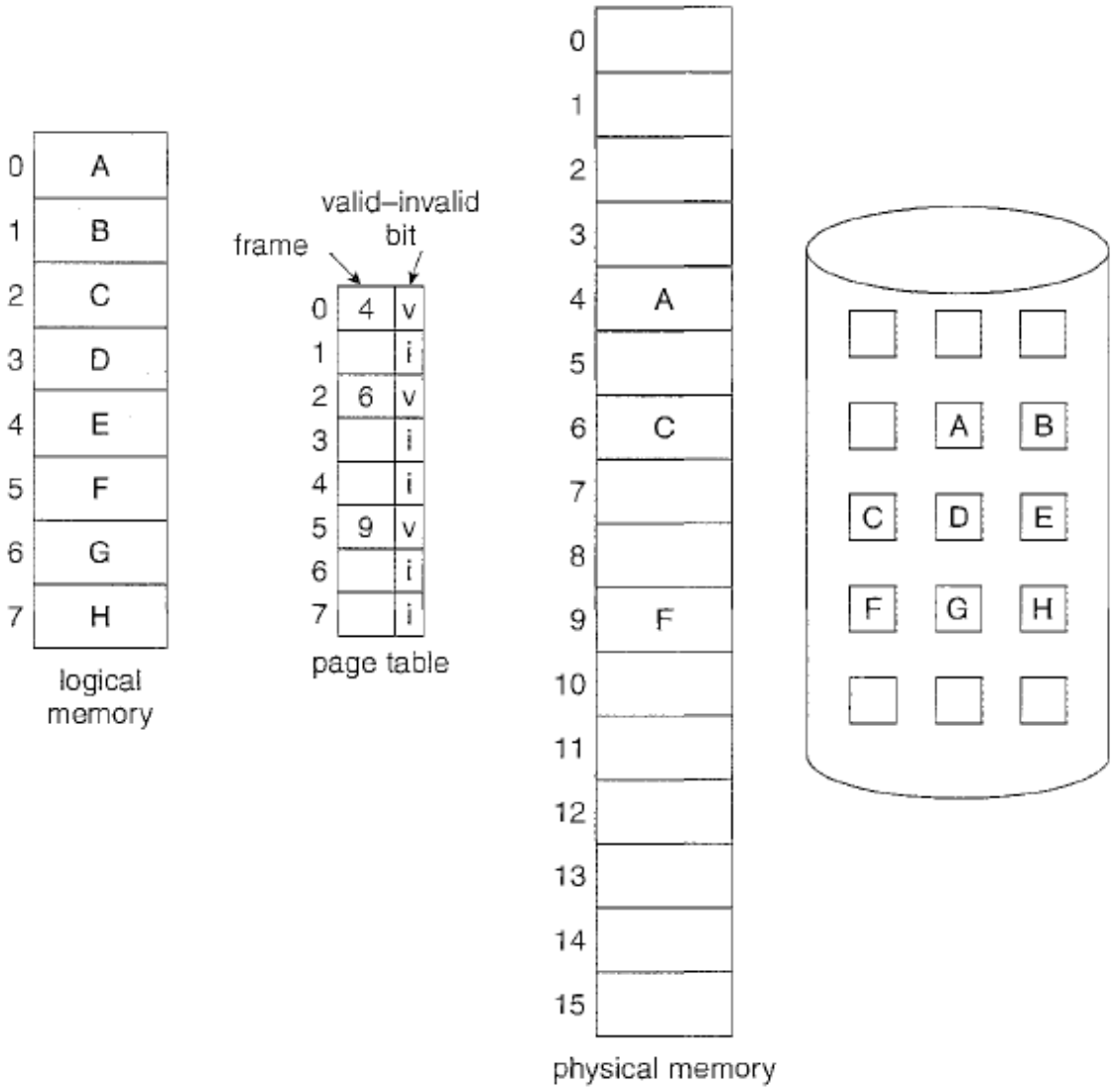


Basic Concepts

- 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. Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.

- With this scheme, we need some form of hardware support to distinguish between the pages that are in memory and the pages that are on the disk. **The valid -invalid bit scheme** can be used for this purpose. When the 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.



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, in translating the address through the page table, will notice that the invalid bit is set, causing 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:
    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.

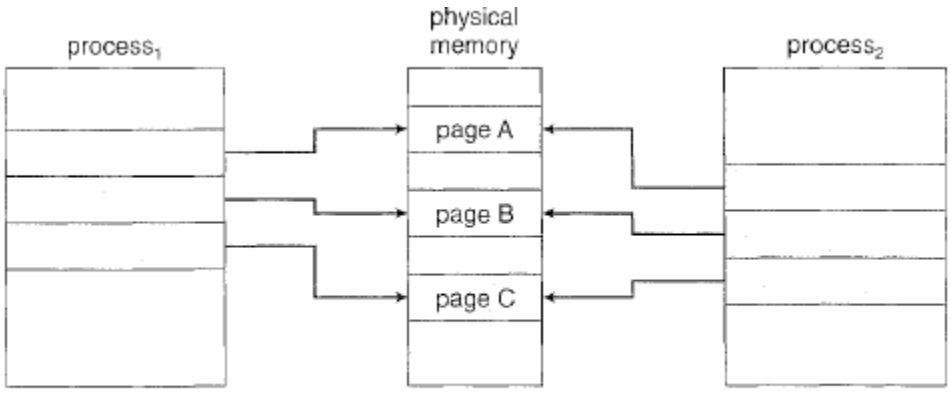
Performance of Demand Paging

Demand paging can significantly affect the performance of a computer system. To see why, let's compute the effective access time for a demand-paged memory.
For most computer systems, the memory-access time, denoted *ma,* ranges from 10 to 200 nanoseconds. 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 word.
Let *p* be the probability of a page fault ($0 <= p <= 1$). We would expect *p* to be close to zero-that is, we would expect to have only a few page faults. The effective access time is then

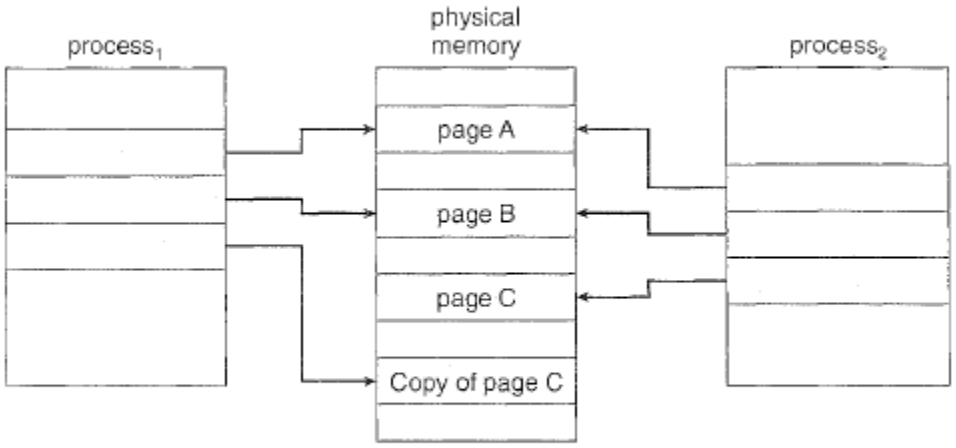
effective access time= $(1 - p)$ x ma + p x page fault time

Copy-On-Write

- As a process can start quickly by merely demand paging in the page containing the first instruction. However, process creation using the fork() system call may initially bypass the need for demand paging by using a technique similar to page sharing.
- This technique provides for rapid process creation and minimizes the number of new pages that must be allocated to the newly created process.
- The fork() system call creates a child process that is a duplicate of its parent. Traditionally, fork() worked by creating a copy of the parent's address space for the child, duplicating the pages belonging to the parent.
- We can use a technique known as **Copy-On-Write** which works by allowing the parent and child processes initially to share the same pages.
- These shared pages are marked as copy-on-write pages, meaning that if either process writes to a shared page, a copy of the shared page is created.
- Copy-on-write is illustrated in Figures below, which show the contents of the physical memory before and after process 1 modifies page c.



Before process 1 modifies page C.



After process 1 modifies page C.

- For example, assume that the child process attempts to modify a page containing portions of the stack, with the pages set to be copy-on-write. The operating system will create a copy of this page, mapping it to the address space of the child process. The child process will then modify its copied page and not the page belonging to the parent process. Obviously, when the copy-on-write technique is used, only the pages that are modified by either process are copied; all unmodified pages can be shared by the parent and child processes.

- When it is determined that a page is going to be duplicated using copy-on- write, it is important to note the location from which the free page will be allocated. Many operating systems provide a **pool** of free pages for such requests.

PAGE REPLACEMENTALGORITHM

Page replacement takes the following approach. If no frame is free, we find one that is not currently being used and free it. We can free a frame by writing its contents to swap space and changing the page table (and all other tables) to indicate that the page is no longer in memory. We can now use the freed frame to hold the page for which the process faulted. We modify the page-fault service routine to include page replacement:

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.

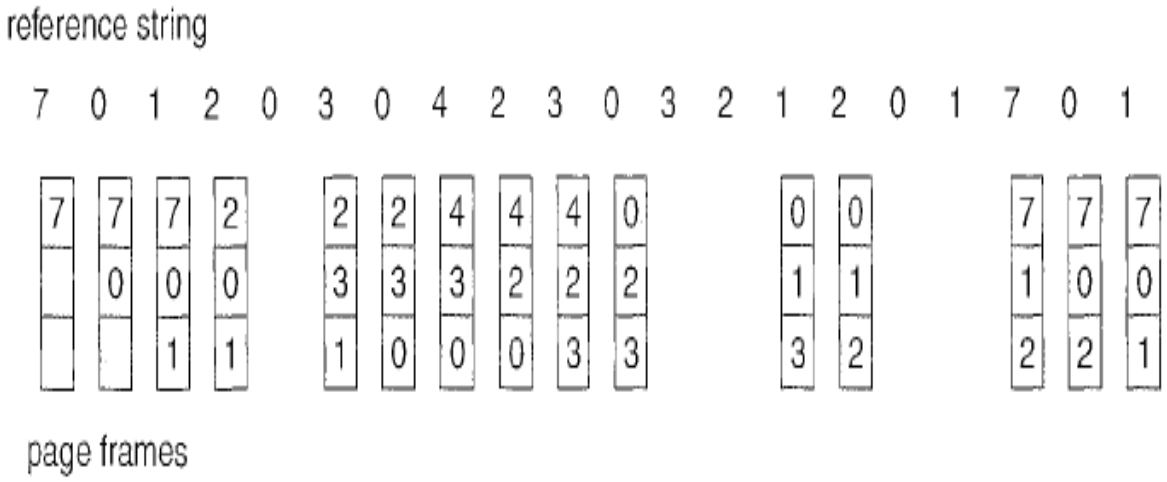
   b. If there is no free frame, use a page-replacement algorithm to select a **Victim Frame.**

   c. Write the victim frame to the disk; change the page and frame tables accordingly.
3. Read the desired page into the newly freed frame; change the page and frame tables.
4. Restart the user process.

Notice that, if no frames are free, *two* page transfers (one out and one in) are required. This situation effectively doubles the page-fault service time and increases the effective access time accordingly.

FIFO PAGE REPLACEMENT ALGORITHM

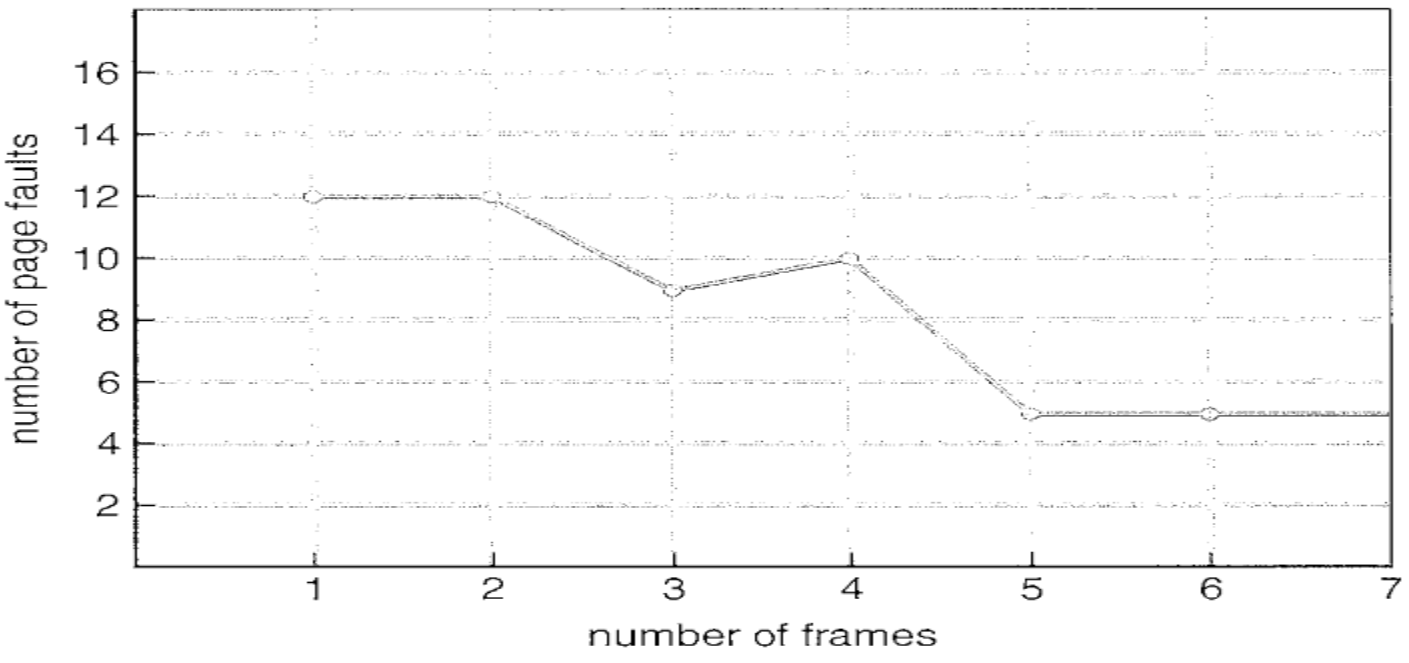
- The simplest page-replacement algorithm is a first-in, first-out (FIFO) algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen.
- Notice that it is not strictly necessary to record the time when a page is brought in. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.



- To illustrate the problems that are possible with a FIFO page-replacement algorithm, we consider the following reference string:

  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- Given Figure shows the curve of page faults for this reference string versus the number of available frames. 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.

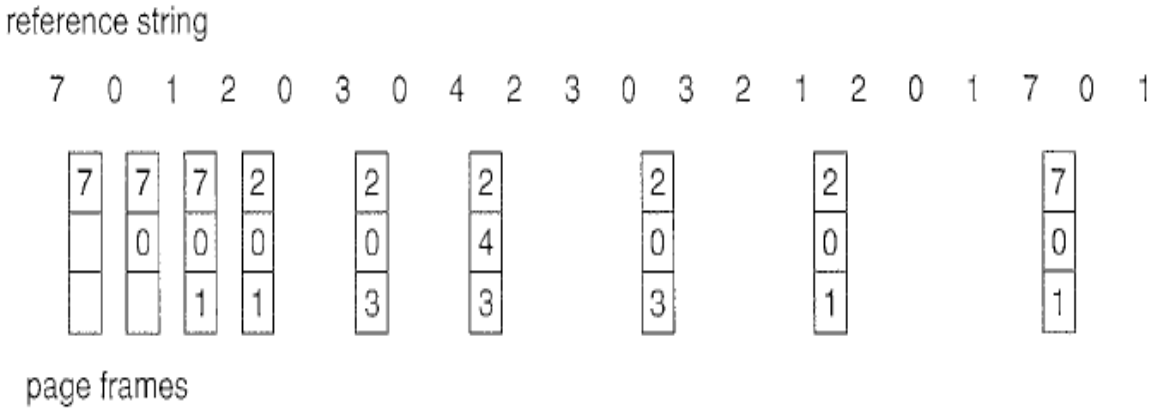


Optimal Page Replacement

One of the discovery due to Belady's Anomaly is Optimal Page Replacement Algorithm- which has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly. Such an algorithm does exist and has been called OPT or MIN. It is simply this:

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

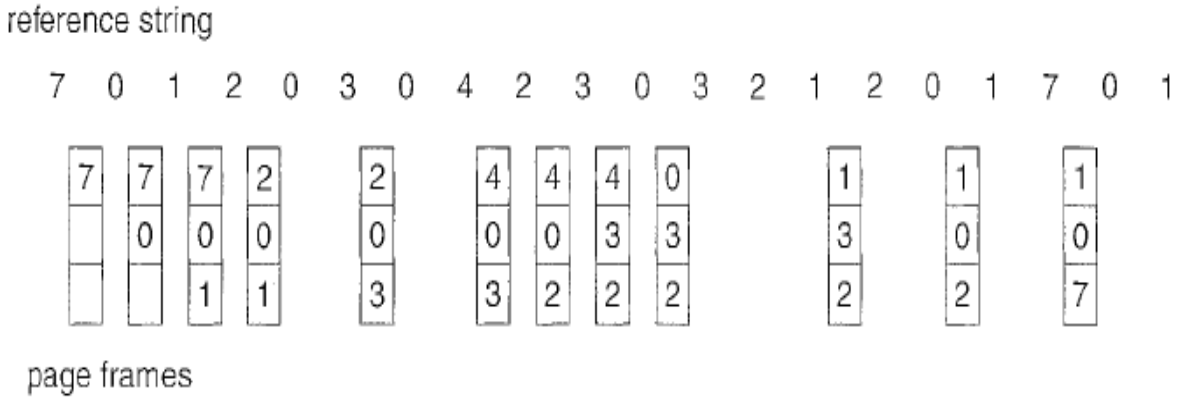
Use of this page-replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames.



LRU Page Replacement Algorithm

In LRU Algorithm the main concern is to use the recent past as an approximation of the near future, then we can replace 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 the page that has not been used for the longest period of time. We can think of this strategy as the optimal page-replacement algorithm looking backward in time, rather than forward.



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. The reason for this selection is that an actively used page should have a large reference count. A problem arises, however, when a page is used heavily during the initial phase of a process but then is never used again. Since it was used heavily, it has a large count and remains in memory even though it is no longer needed. One solution is to shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average usage count.
- **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. As you might expect, neither MFU nor LFU replacement is common. The implementation of these algorithms is expensive, and they do not approximate OPT replacement well.

Page-Buffering Algorithms

- Other procedures are often used in addition to a specific page-replacement algorithm. For example, systems commonly keep a pool of free frames. When a page fault occurs, a victim frame is chosen as before. However, the desired page is read into a free frame from the pool before the victim is written out. This procedure allows the process to restart as soon as possible, without waiting for the victim page to be written out. When the victim is later written out, its frame is added to the free-frame pool.

- An expansion of this idea is to maintain a list of modified pages. Whenever the paging device is idle, a modified page is selected and is written to the disk. Its modify bit is then reset. This scheme increases the probability that a page will be clean when it is selected for replacement and will not need to be written out.

- Another modification is to keep a pool of free frames but to remember which page was in each frame. Since the frame contents are not modified when a frame is written to the disk, the old page can be reused directly from the free-frame pool if it is needed before that frame is reused. No I/O is needed in this case. When a page fault occurs, we first check whether the desired page is in the free-frame pool. If it is not, we must select a free frame and read into it.

Allocation of Frames

How do we allocate the fixed amount of free memory among the various processes? If we have 93 free frames and two processes, how many frames does each process get?

- The simplest case is the single-user system. Consider a single-user system with 128 KB of memory composed of pages 1 KB in size. This system has 128 frames. The operating system may take 35 KB, leaving 93 frames for the user process. Under pure demand paging, all 93 frames would initially be put on the free-frame list. When a user process started execution, it would generate a sequence of page faults. The first 93 page faults would all get free frames from the free-frame list. When the free-frame list was exhausted, a page-replacement algorithm would be used to select one of the 93 in-memory pages to be replaced with the 94th, and so on. When the process terminated, the 93 frames would once again be placed on the free-frame list.

Allocation Algorithms

- The easiest way to split *m* frames among *n* processes is to give everyone an equal share, *m/n* frames. For instance, if there are 93 frames and five processes, each process will get 18 frames. The three leftover frames can be used as a free-frame buffer pool. This scheme is called **Equal Allocation.**

- An alternative is to recognize that various processes will need differing amounts of memory. Consider a system with a 1-KB frame size. If a small student process of 10 KB and an interactive database of 127 KB are the only two processes running in a system with 62 free frames, it does not make much sense to give each process 31 frames. The student process does not need more than 10 frames, so the other 21 are, strictly speaking, wasted.

- To solve this problem, we can use **Proportional Allocation** in which we allocate available memory to each process according to its size. Let the size of the virtual memory for process p_i; be s_i, and define

$$S = \sum s_i .$$

- Then, if the total number of available frames is *m*, we allocate a_i frames to process p_i, where a_i is approximately

$$a_i = s_i / S \times m.$$

We must adjust each *ai* to be an integer that is greater than the minimum number of frames required by the instruction set, with a sum not exceeding *m*.

With proportional allocation, we would split 62 frames between two processes, one of 10 pages and one of 127 pages, by allocating 4 frames and 57 frames, respectively, since

10/137 x 62 ~ 4, and
127/137 X 62 ~57.

In this way, both processes share the available frames according to their "needs," rather than equally.

Global versus Local Allocation

- Another important factor in the way frames are allocated to the various processes is page replacement. With multiple processes competing for frames, we can classify page-replacement algorithms into two broad categories: **Global Replacement** and **Local Replacement**.

- Global replacement allows a process to a replacement frame from the set of all frames, even if that frame is currently allocated to some other process; that is, one process can take a frame from another.

- Local replacement requires that each process select from only its own set of allocated frames.

- For example, consider an allocation scheme wherein we allow high-priority processes to select frames from low-priority processes for replacement. A process can select a replacement from among its own frames or the frames of any lower-priority process. This approach allows a high-priority process to increase its frame allocation at the expense of a low-priority process.

- With a local replacement strategy, the number of frames allocated to a process does not change. With global replacement, a process may happen to select only frames allocated to other processes, thus increasing the number of frames allocated to it

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 that process's execution. 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.

- In fact, look at any process that does not have "enough" frames. If the process does not have the number of frames it needs to support pages in active use, it will quickly 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, replacing pages that it must back in immediately.
- This high paging activity is called **Thrashing**. *A* process is thrashing if it is spending more time paging than executing.

Causes of Thrashing

Consider the following scenario, which is based on the actual behavior of early paging systems.
The operating system monitors CPU utilization. If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system.
A global page-replacement algorithm is used; it replaces pages without regard to the process to which they belong.

Now suppose that a process enters a new phase in its execution and needs more frames. It starts faulting and taking frames away from other processes. These processes need those pages, however, and so they also fault, taking frames from other processes.

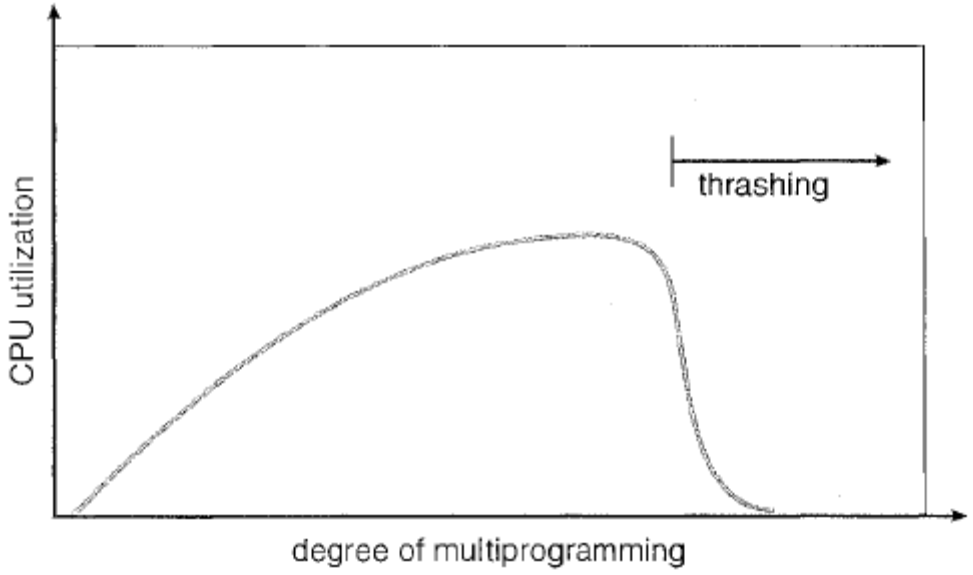
These faulting processes must use the paging device to swap pages in and out. As they queue up for the paging device, the ready queue empties. As processes wait for the paging device, CPU utilization decreases.

The CPU scheduler sees the decreasing CPU utilization and *increases* the degree of multiprogramming as a result.

The new process tries to get started by taking frames from running processes, causing more page faults and a longer queue for the paging device. As a result, CPU utilization drops even further, and the CPU scheduler tries to increase the degree of multiprogramming even more.

Thrashing has occurred, and system throughput plunges. The page fault rate increases tremendously. As a result, the effective memory-access time increases. No work is getting done, because the processes are spending all their time paging.

This phenomenon is illustrated in given Figure in which CPU utilization is plotted against the degree of multiprogramming.

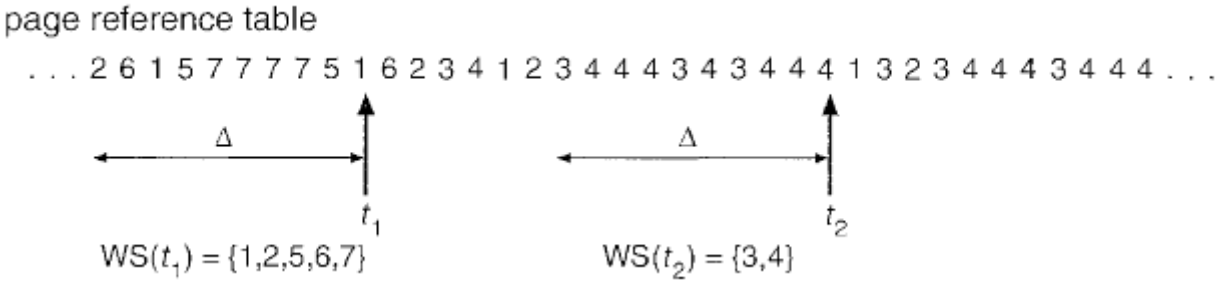


As the degree of multiprogramming increases, CPU utilization also increases, although more slowly, until a maximum is reached. If the degree of multiprogramming is increased even further, thrashing sets in, and CPU utilization drops sharply. At this point, to increase CPU utilization and stop thrashing, we must *decrease* the degree of multiprogramming.

- To prevent thrashing, we must provide a process with as many frames as it needs. But how do we know how many frames it "needs"? There are several techniques. The **working-set strategy** starts by looking at how many frames a process is actually using. This approach defines the locality model of process execution.

Working-Set Model

The Working-Set Model is based on the assumption of locality. This model uses a parameter Δ, to define the working set window. The idea is to examine the most recent Δ pages references. The set of pages in the most recent Δ page references is the **working set.**



If a page is in active use, it will be in the working set. If it is no longer being used, it will drop from the working set Δ time units after its last reference. Thus, the working set is an approximation of the program's locality.

For example, given the sequence of memory references shown in Figure above, if Δ= 10 memory references, then the working set at time t1 is {1, 2, 5, 6, 7}. By time *t2,* the working set has changed to {3, 4}.

The accuracy of the working set depends on the selection of Δ. If $\Delta$is too small, it will not encompass the entire locality; if $\Delta$is too large, it may overlap several localities.

The most important property of the working set, then, is its size. If we compute the working-set size, WSS_i for each process in the system, we can then consider that

$$D = \sum WSS_i,$$

where Dis the total demand for frames.

Each process is actively using the pages in its working set. Thus, process i needs WSS_i frames. If the total demand is greater than the total number of available frames (D > m), thrashing will occur, because some processes will not have enough frames.