Virtual Memory

Virtual memory is a memory management technique that allows the execution of a program, when only part of it is memory resident. Due to this technique, a program with memory requirements larger than physical memory is executed. Virtual memory also allows processes to share files easily and to implement shared memory. Execution of partially resident programs has following benefits:

- 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, degree of multiprogramming is increased, with a corresponding increase in CPU utilization.
- Programs load time is reduced, and system throughput is increased.

Virtual memory is a storage allocation scheme in which secondary memory can be addressed as though it were part of main memory. In virtual memory users can view main memory as an extremely large array of storage and is referred as Virtual Address Space, which refers to the logical (or virtual) view (contiguous with a range 0-FF..FF) of how a process is stored in memory.

Virtual memory is implemented with paging, where physical address space is divided into frames and logical address space is divided into pages. Like in paging, page map table is maintained along with information of memory resident pages at any given time.

Demand Paging

Rather than loading all pages of a program, load pages only as they are needed. With demand-paged virtual memory, pages are loaded only when they are demanded during program execution. A demand-paging system is similar to a paging system with swapping where processes reside on a disk as shown in Figure-1. Rather than swapping the entire process from disk into memory, a page which is required/demanded is moved. The part of OS, which manages transfer of pages from disk to memory is called a *pager*.

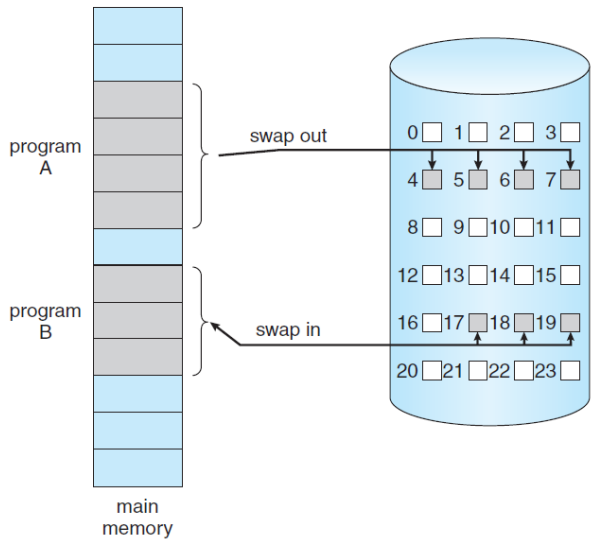


Figure-1: Swapping-in/out pages (page-in and page-out)

When a process is to be placed in memory, 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. A valid bit is used which is part of page table of a process. When this bit is set to "valid," the associated page is in memory. If the bit is set to "invalid," the page is not in memory and is currently on the disk. The page-table entry for a page that is brought into memory is set, 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 as shown in Figure 2.

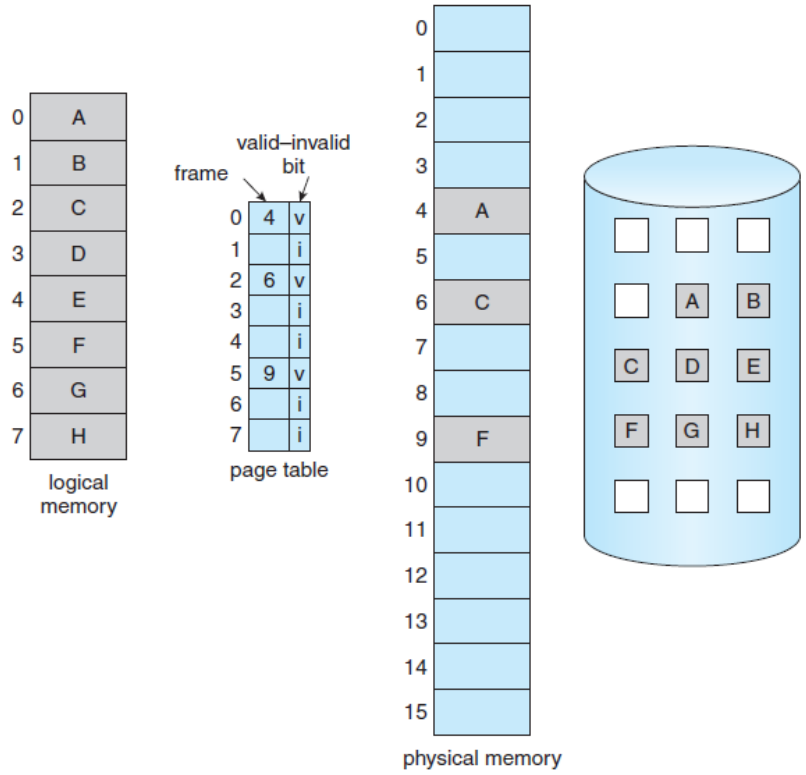


Figure 2: Page table with valid bit

Page Fault

If pager guess rightly and page-in all pages that are needed, the process will run exactly as though we had brought in all pages. While the process executes and access pages that are memory resident, execution proceeds normally, 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 steps for handling this page fault are depicted in Figure 3. The page fault service routine is listed below:

1. Find the location of the desired page on the disk.
2. Find a free frame.
3. Read the desired page into the newly allocated free frame; change the page and frame tables.
4. Continue the user process from where the page fault occurred

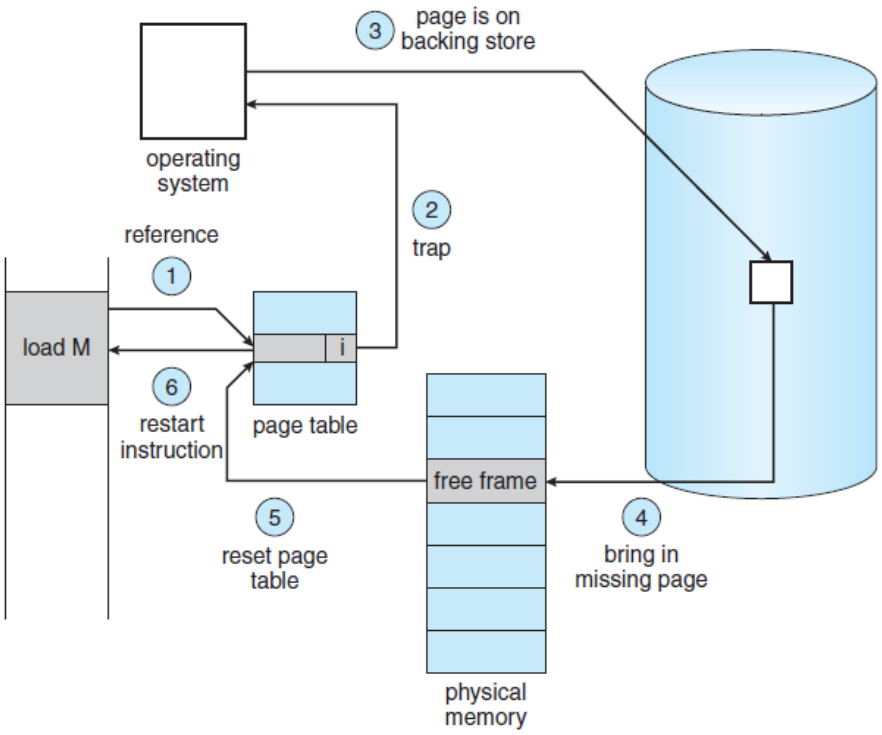


Figure 3: Handling a page fault

In the extreme case, we can start executing a process with no pages in memory. When the operating system sets the instruction pointer to the first instruction of the process, which is on a non-memory-resident page, the process immediately faults for the page. After this page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory. At that point, it can execute with no more faults. This scheme is *pure demand paging*: never bring a page into memory until it is required. A crucial requirement for demand paging is the ability to restart any instruction after a page fault. Because we save the state (registers, condition code, instruction counter) of the interrupted process when the page fault occurs, we must be able to restart the process from the same place and state, except that the desired page is now in memory and is accessible. In most cases, this requirement is easy to meet. A page fault may occur at any memory reference. 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.
By using simple example of paging section, we can illustrate how a page fault occurs, and how a free frame is used to bring in page into memory. The page table is slightly modified to have valid bit entry for each page/frame of the process. In this case if valid bit is '1' it means the page is memory and if valid bit is '0' it means the page is not in memory (no frame is allocated). In our example, 3 pages (0 – 2) are placed in memory. Figure-4 depicts the necessary information. We want to access, 'H' (0111) and Page-3 (11) is indexed to page table, and valid bit is examined, which is '0' means page is not in memory. Page fault occurs. That situation, how a page fault occurs in depicted in Figure-5.

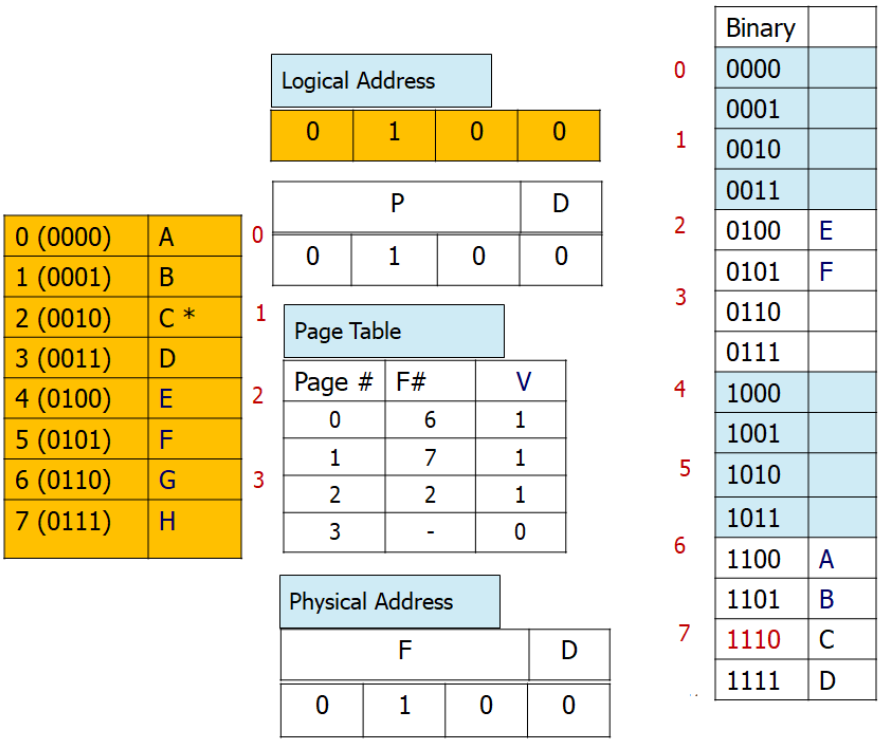


Figure-4: Pages, Page table and frames of a Process

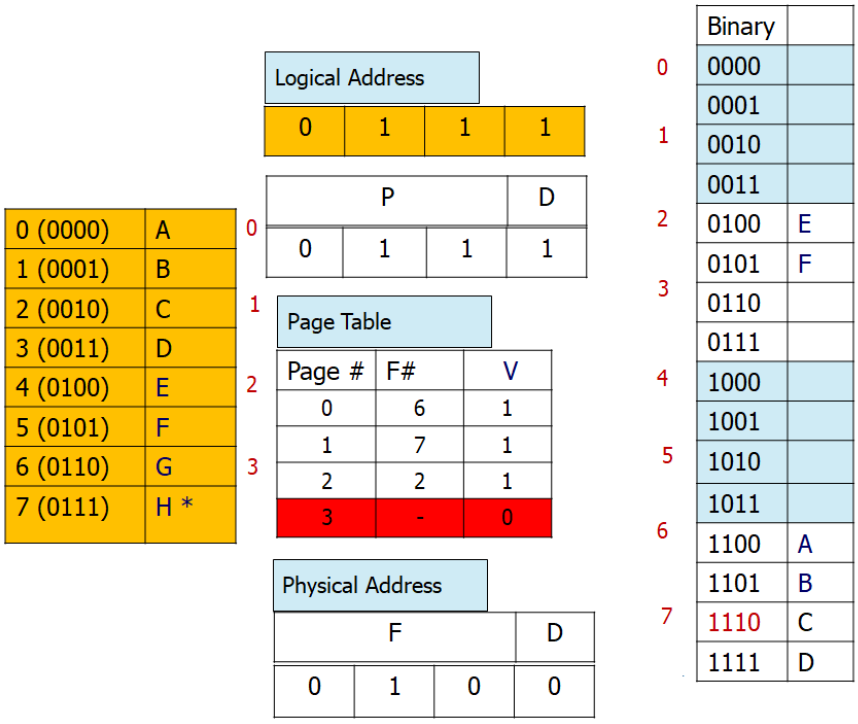


Figure-5: Page fault occurrence

Next, we look for a free frame, which is frame 3, so place page 3 in frame 3, modify the page table. When a page fault occurs, and OS calls page fault service routine, status of the process is changed to ‘Block/Wait’. When the page which has faulted is brought in memory, the status of the process is changed to ‘Ready”. Now the process can continue to access H (where page fault occurs) when it gets CPU and this situation is depicted in Figure-6.

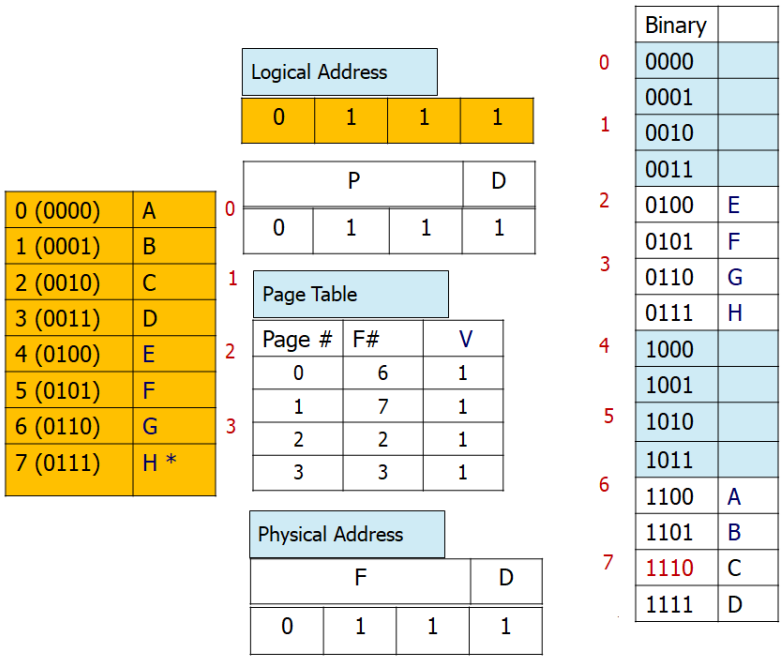


Figure-6: Page table status after page fault serviced

Demand paging only brings pages into main memory when a reference is made to a location on the page. There may be many page faults when process is first started. However, principle of locality suggests that as more and more pages are brought in, most future references will be to pages that have recently been brought in, and page faults should drop to a very low level.

Pre-Paging

Process creation using fork() system call may initially bypass the need for demand paging by using a technique similar to page sharing. This technique provides rapid process creation and minimizes the number of new pages that must be allocated to the newly created process. Another technique allocates a few initial pages from the virtual address space at creation of the process. This is called pre-paging, page(s) is brought in memory before it is demanded. This reduces first few page faults. This scheme exploits the characteristics of most secondary memory devices if pages of a process are stored contiguously in secondary memory. It is more efficient to bring in several pages at one time. Since in this scheme pages other than the one demanded by a page fault are brought in, and if these pages are not referenced, then the memory utilization is compromised which results in more page faults by other processes which may have been allocated those pages.

Copy-on-Write

fork() system call creates a copy of the parent's address space for the child, duplicating the pages belonging to the parent. However many child processes invoke the exec() system call immediately after creation, the copying of the parent's address space may be unnecessary. Instead, a technique called *copy-on-write*, allows 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.

Page Replacement

While describing page fault handling in page fault section, we assume that a free frame is available in the free frame list and is allocated to the process which faulted the page. If there is no free frame available, then a page is replaced. We find one that is not currently being used and free it. We can free a frame by writing its contents to disk and changing the page table 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. The working of page replacement is depicted in Figure-7.

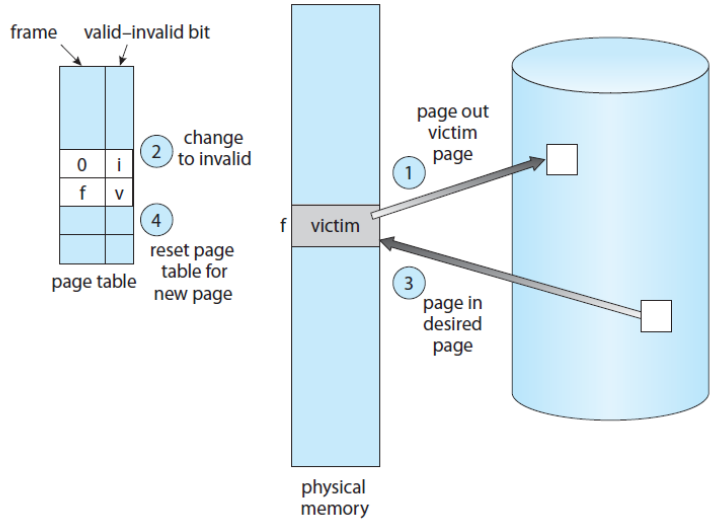


Figure-7.

The page fault service procedure is modified to include page replacement and is listed below:

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. Continue the user process from where the page fault occurred.

When no frames are free, two-page transfers (one out and one in) are required, which doubles the page-fault service time and increases the effective access time accordingly. We can reduce this overhead by using a modify bit (or dirty bit). In this scheme, each page or frame has a modify bit associated with it in the hardware. The modify bit for a page is set by the hardware whenever any byte in the page is written into, indicating that the page has been modified. When a page is selected for replacement, modify bit of that page is examined. If the bit is set, we know that the page has been modified since it was read in from the disk. In this case, we must write the page to the disk. A later reference to that page will cause a page fault. At that time, the page will be brought back into memory, perhaps replacing some other page in the process. If the modify bit is not set, however, the page has not been modified since it was brought into memory. In this case, no need to write the memory page to the disk: it is already there. This technique also applies to read-only pages. Such pages cannot be modified; thus, they may be discarded when desired. This scheme can significantly reduce the time required to service a page fault, since it reduces I/O time by one-half if the page has not been modified.

Page replacement is basic to demand paging. It completes the separation between logical memory and physical memory. With this mechanism, an enormous virtual memory

can be provided for programmers on a smaller physical memory. With demand paging, the size of the logical address space is no longer constrained by physical memory. If we have a user process of thirty pages, we can execute it in ten frames simply by using demand paging and using a page replacement algorithm to find a free frame whenever necessary.

Frame Allocation

For implementing demand paging, frame allocation is managed in an effective manner. With multiple processes in memory, how to allocate the fixed number of free fames among the various processes? Because as the number of frames allocated to each process decreases, the page-fault rate increases, slowing process execution. The minimum number of frames to be allocated to a process for normal instruction execution is defined by the computer architecture. The maximum number is defined by the amount of available physical memory. In between, OS designer have a significant choice in frame allocation. The easiest way to split available *x* frames among *y* processes is to give everyone an equal share, *x/y* frames. This scheme of equal allocation works well for small processes. For example, If we have 75 free frames and five processes, each process will get 15 frames. Though various processes will need differing amounts of memory. Consider a system with a 2KB frame size. If a small student process of 10KB and an interactive database of 120KB are two processes running in a system with 60 free frames, it does not make much sense to give each process 30 frames. The student process does not need more than 5 frames, so the other 25 are, wasted.

To solve this problem, we can use proportional allocation, in which we allocate available memory to each process according to its size. With proportional allocation, we would split 60 frames between two processes, one of 5 pages and one of 60 pages, by allocating 5 frames and 55 frames, respectively, by using following proportional equation:

$5/65 \times 5 \approx 5$, and

$60/65 \times 02 \approx 55$

In this allocation scheme, both processes share the available frames according to their memory needs.

In proportional allocation scheme the ratio of frames can be calculated not on the relative sizes of processes but rather on the priorities of processes or on a combination of size and priority.

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 select 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. In case of local replacement, allocation is fixed/static, and due to global replacement, frames allocation can be increased or decreased. In case of local page replacement, allocation is variable/dynamic. The problem with a global page replacement algorithm is that a process cannot control its own page-fault rate. The set of pages in memory for a process depends not only on the paging behavior of that process but also on the paging behavior of other processes.

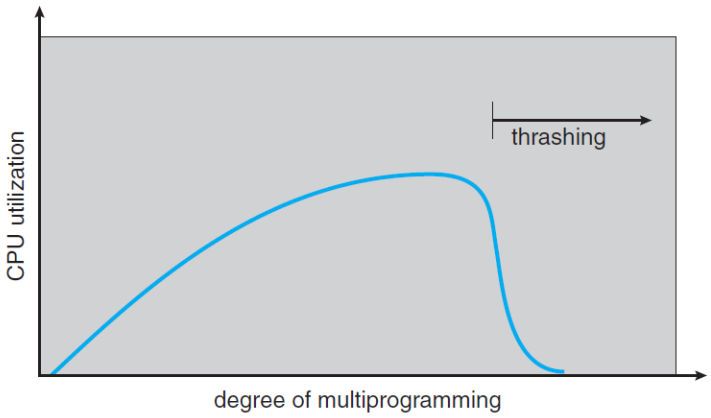
Thrashing

If the number of frames allocated to a process falls below the minimum number required by the architecture, that process's execution must be suspended (process swaped-out). 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 bring back in immediately. This high paging activity is called thrashing. A process is thrashing if it is spending more time paging than executing. Thrashing results in severe performance problems.

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.

We can limit the effects of thrashing by using a local replacement algorithm. With local replacement, if one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash as well.

To prevent thrashing, system must provide a process with as many frames as it needs. But how do we know how many frames it “needs”? 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.

A better strategy that uses the page-fault frequency (PFF) takes a more direct approach. The specific problem is how to prevent thrashing, which has a high page-fault rate. Thus, we want to control the page-fault rate. When it is too high, we know that the process needs more frames. Conversely, if the page-fault rate is too low, then the process may have too many frames. We can establish upper and lower bounds on the desired page-fault rate as shown in Figure-8.

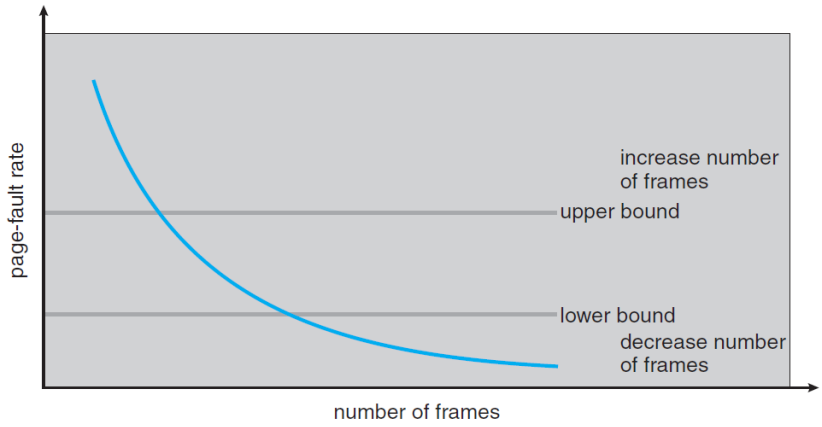


Figure-8: Page fault frequency

If the actual page-fault rate exceeds the upper limit, we allocate the process another frame. If the page-fault rate falls below the lower limit, we remove a frame from the process. Thus, we can directly measure and control the page-fault rate to prevent thrashing.
There may arise a situation where a process has to swapped out. If the page-fault rate increases and no free frames are available, we must select some process and swap it out to backing store. The freed frames are then distributed to processes with high page-fault rates.