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 programmer from physical memory.

This technique frees programmers from the concerns of memory-storage limitations.

Background

An examination of real programs shows us that, in many cases, the entire program is not needed. For instance, consider the following:

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

ii) 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.

iii) Certain options and features of a program may be used rarely.

Even in those cases where the entire program is needed, it may not all be needed at the same time.

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

i) 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.

ii) Each program could take less physical memory. So more programs could be run at the same time. Therefore, this scheme will increase in CPU utilization and throughput, but it will not increase response time or turnaround time.

iii) Less I/O would be needed to load or swap portions of programs into memory, so each program would run faster.

Thus, running a program that is not entirely in memory would benefit both the system and its users.

If a system supports virtual memory, then the overlays have almost disappeared.

Virtual memory is commonly implemented by demand paging.

It can also be implemented in a segmentation system.

Demand segmentation can also be used to provide virtual memory.

Demand Paging

The demand paging technique is commonly used in virtual memory systems.

Here the strategy is to load pages only when they are needed.

With demand-paged virtual memory, pages are loaded only when they are demanded during program execution.

Pages that are never accessed are thus never loaded into physical memory.

The demand paging system is similar to a paging system with swapping.

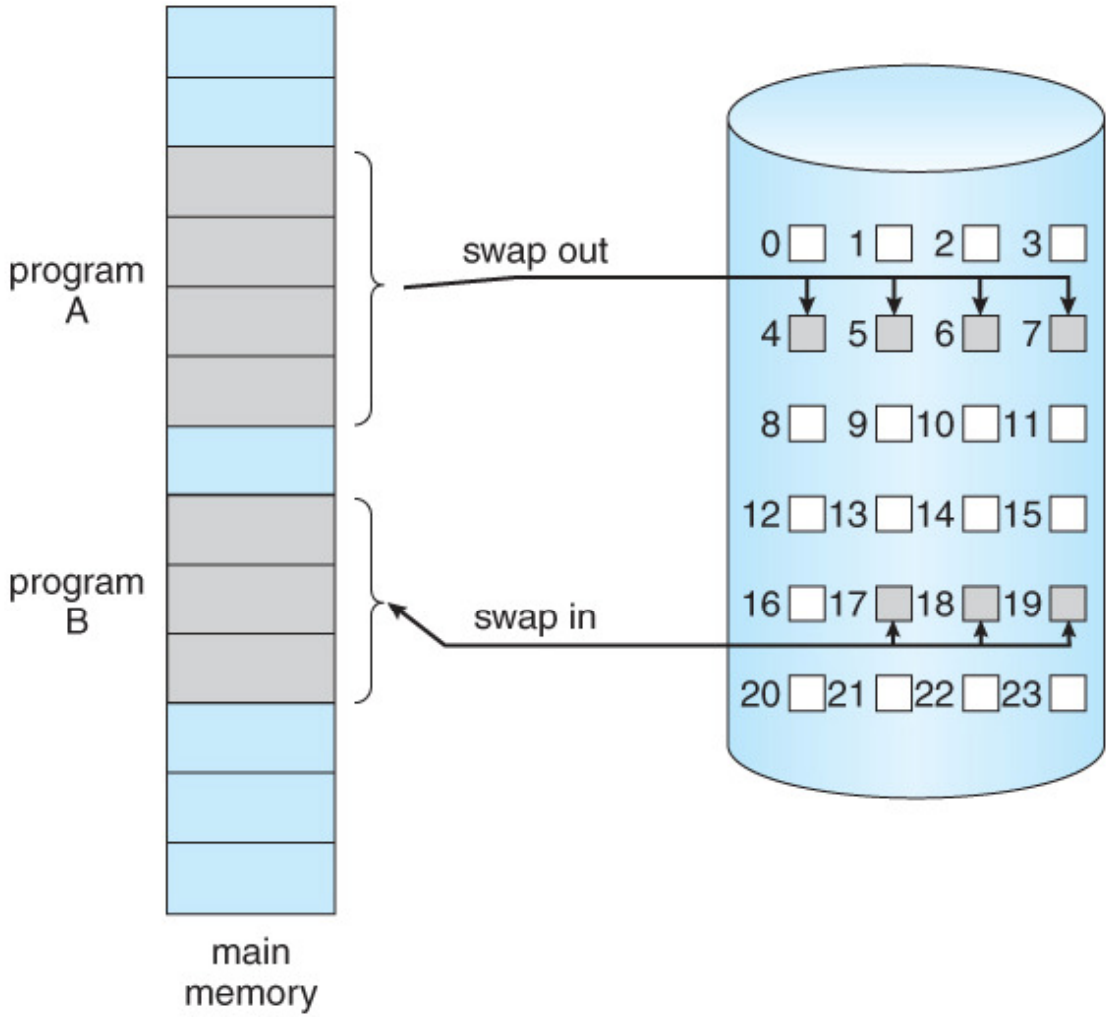

Fig 1: Transfer of a paged memory to contiguous disk space.

Basic Concepts

The general concept behind demand paging is to load a page in memory only when it is needed.

As a result, while a process is executing, some pages will be in memory, and some will be in secondary storage.

Thus, we need some form of hardware support to distinguish between the two.

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 in secondary storage.

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 simply marked invalid.

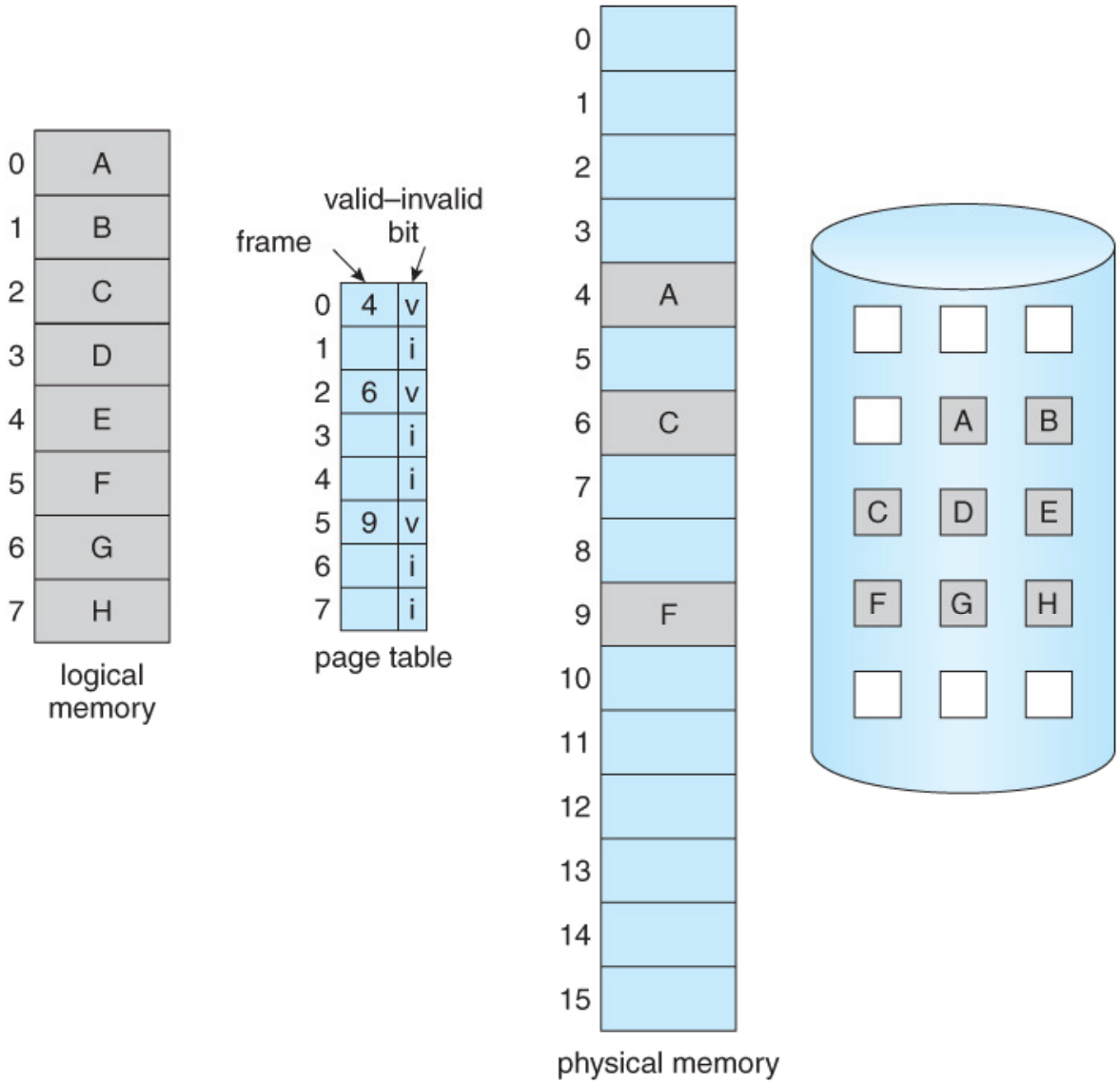

Fig 2: Page table when some pages are not in main memory.

If the process tries to access a page that was not brought into memory, then the access to a page marked invalid causes a ***page fault***.

The procedure for handling this page fault is as follows:

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 secondary storage operation to read the desired page into the newly allocated frame.
5. When the storage 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.

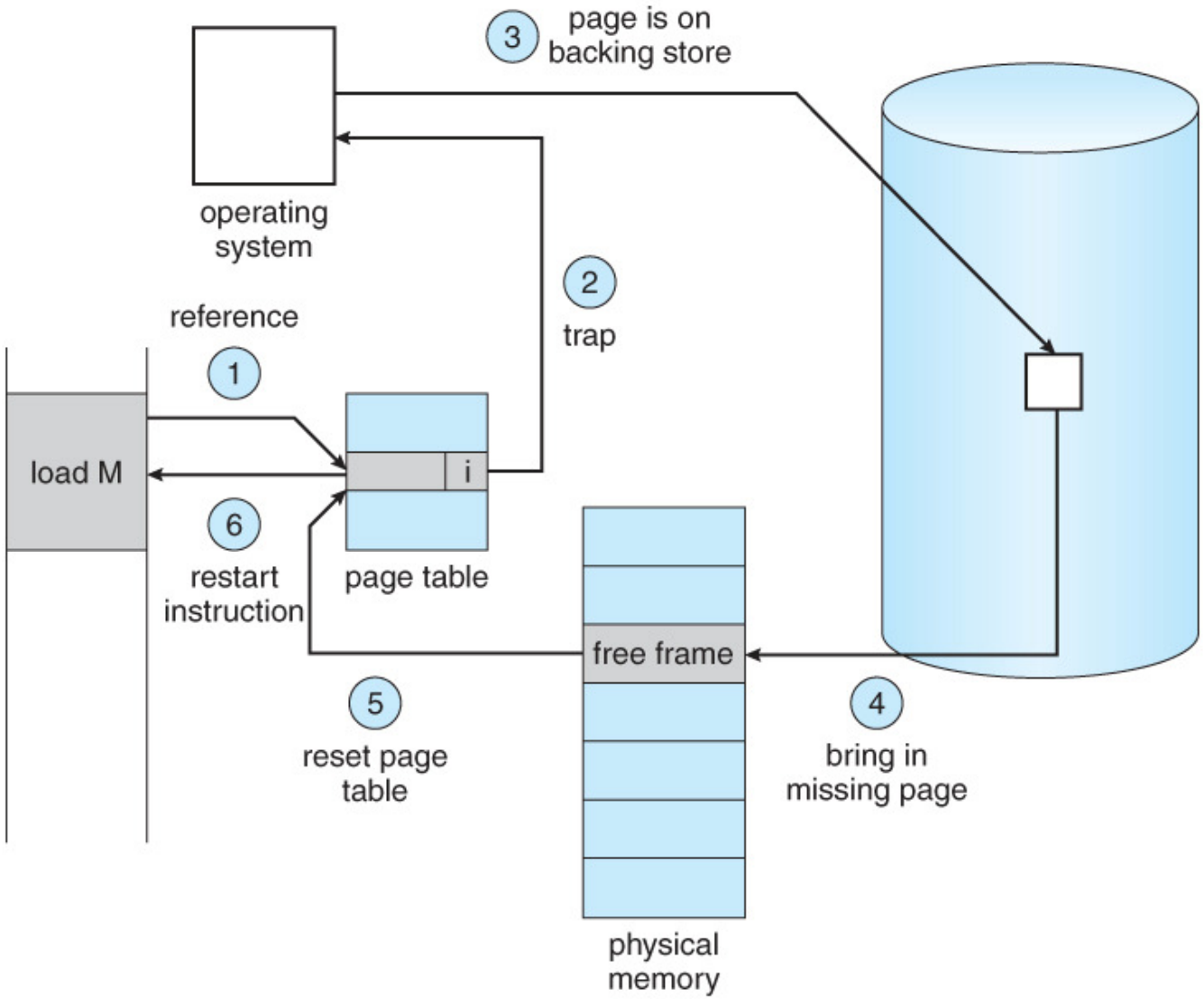

Fig 3: Steps in handling a page fault.

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.

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. It is known as the swap device, and the section of storage used for this purpose is known as swap space.

Performance of Demand Paging

Demand paging can significantly affect the performance of a computer system.

The effective access time for a demand-paged memory can be calculated as follows:

Let p be the probability of a page fault ($0 \leq p \leq 1$); Then,

effective access time $= (1 - p) \times$ ***memory access time*** $+ p \times$ ***page fault time***.

Let an average page-fault service time of 8 milliseconds and a memory access time of 200 nanoseconds, the effective access time in nanoseconds is

Effective access time = $(1 - p) \times (200) + p$ (8 milliseconds)
$= (1 - p) \times 200 + p \times 8{,}000{,}000$
$= 200 + 7{,}999{,}800 \times p$.

It can be observed that the effective access time is directly proportional to the page-fault rate.

If one access out of 1,000 causes a page fault, then

Effective access time$=200 + 7{,}999{,}800 \times p$
$=200 + 7{,}999{,}800 \times 1/1000$ nanosecond
=8199.8 nanosecond

The computer will be slowed down due to demand paging by a factor of
=8199.8/200
=40.99

If we want performance degradation to be less than 10 percent, we need to keep the probability of page faults at the following level:

$$220 > 200 + 7{,}999{,}800 \times p$$
$$20 > 7{,}999{,}800 \times p$$
$$p < 0.0000025$$

That is, to keep the slowdown due to paging at a reasonable level, we can allow fewer that one memory access out of 399,990 to page-fault.

Page Replacement

While a process is executing, a page fault occurs.

The hardware traps to the operating system, which checks its internal table to see that this page fault is genuine one rather than an illegal memory access.

The operating system determines where the desired page is residing on the disk.

If there is a free frame in physical memory, then load that page in that free frame, otherwise we need page replacement scheme.

Basic scheme

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 secondary storage.

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 secondary storage (if necessary); 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 process from where the page fault occurred.

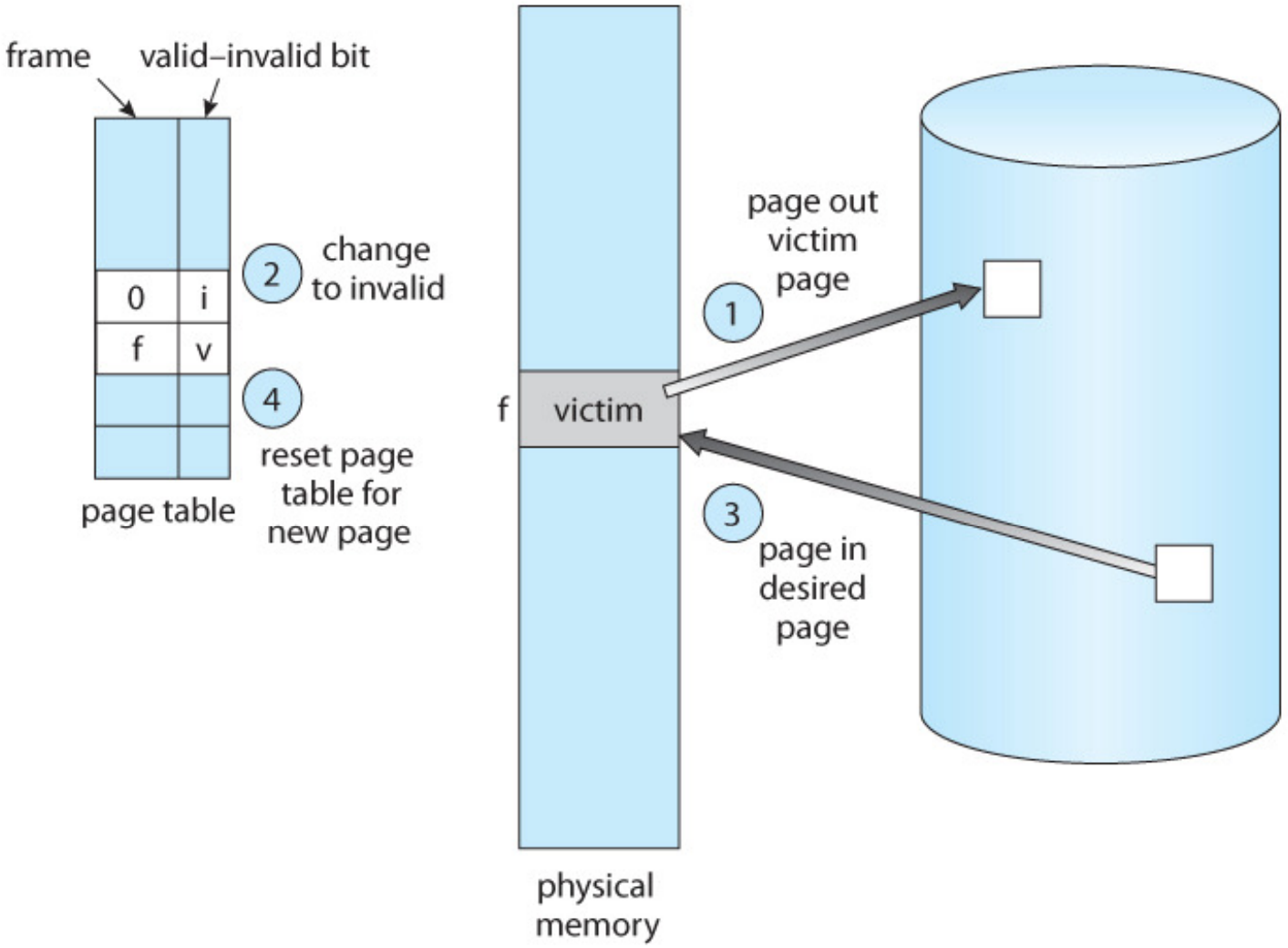

Fig 4: Page replacement.

If no frames are free, two page transfers (one for the page-out and one for the page-in) are required.

This situation effectively 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***).

When this scheme is used, 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 we select a page for replacement, we examine its modify bit.

If the bit is set, we know that the page has been modified since it was read in from secondary storage.

In this case, we must write the page to storage.

If the modify bit is not set, however, the page has not been modified since it was read into memory.

In this case, we need not write the memory page to storage: it is already there.

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.

We must solve two major problems to implement demand paging: we must develop a ***frame-allocation algorithm*** and a ***page-replacement algorithm***.

That is, if we have multiple processes in memory, we must decide how many frames to allocate to each process; and when page replacement is required, we must select the frames that are to be replaced.

Designing appropriate algorithms to solve these problems is an important task, because secondary storage I/O is so expensive.

Even slight improvements in demand-paging methods yield large gains in system performance.

There are many different page-replacement algorithms.

Every operating system probably has its own replacement scheme.

In general, we want the one with the lowest page-fault rate.

We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults. The string of memory references is called a ***reference string***.

First, for a given page size (and the page size is generally fixed by the hardware or system), we need to consider only the page number, rather than the entire address.

For example, consider the reference string with following address sequence:

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103, 0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105

If the page size is 100 bytes, then the reference string can be reduced as:

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

As the number of frames increases, the number of page faults drops to some minimal level.

FIFO Page Replacement

The 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.

Let consider the following example where number of available page frame is 3.

Let the reference string is:

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

According to the FIFO page replacement algorithm, the situation is as follows:

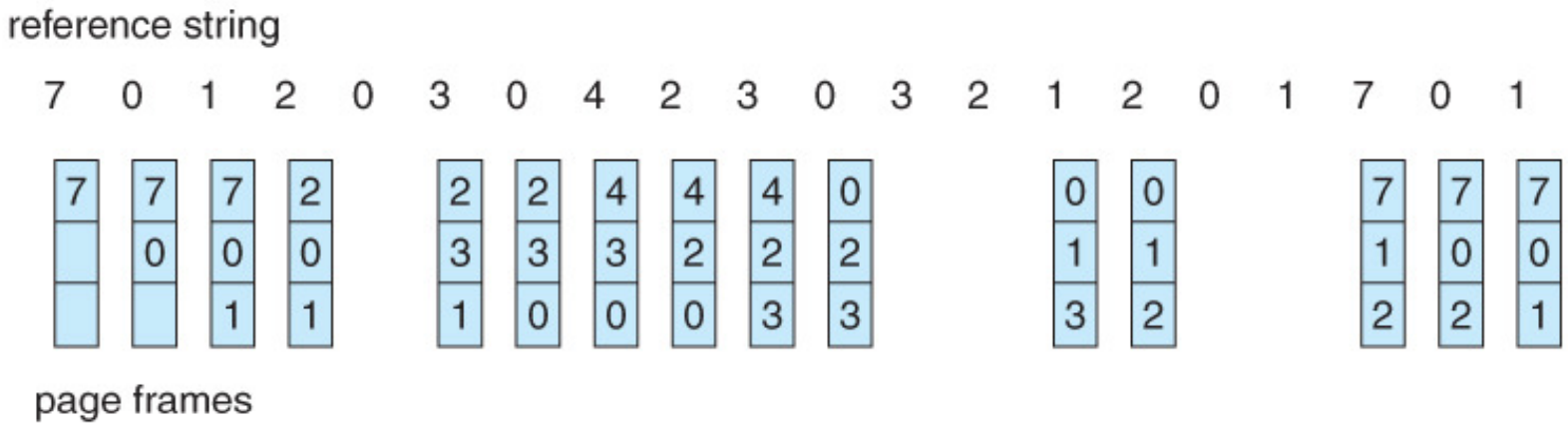

Fig 5: FIFO page-replacement algorithm.

In this example we see that there are total 15 numbers of page fault occur.

However, the performance of this algorithm is not always good.

In this algorithm, we may replace a page that was initialized earlier but is heavily used.

After we page out an active page to bring in a new one, a fault occurs almost immediately to retrieve the active page.

Thus a bad replacement choice increases the page fault rate and shows process execution.

In this algorithm, there is a possibility of **Beladey's anomaly**.

Belady's anomaly: For some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increases.

Optimal Page Replacement

An optimal page-replacement algorithm has the ***lowest page-fault rate*** of all algorithms and will ***never suffer from Belady's anomaly***.

The algorithm is as follows:

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

Let consider the following example where number of available page frame is 3.

Let the reference string is:

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

According to the Optimal page replacement algorithm, the situation is as follows:

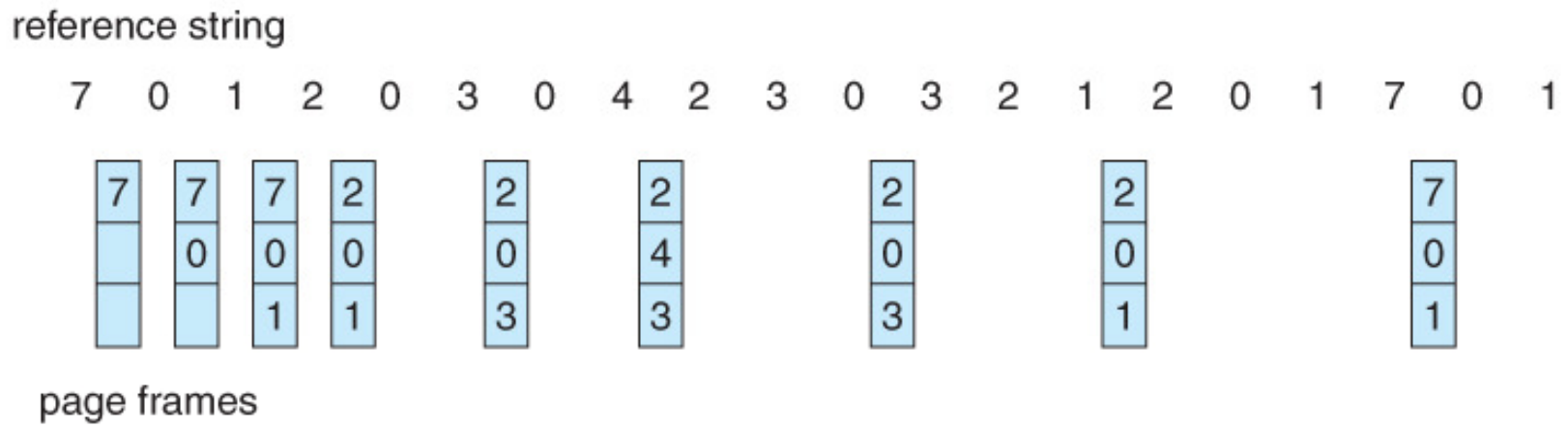

Fig 6: Optimal page-replacement algorithm.

With only 9 page faults, the optimal page replacement algorithm is much better than the FIFO page replacement algorithm, which results in 15 faults.

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

In some situations, the future knowledge of the reference string is not available.

As a result, the optimal algorithm is used mainly for comparison studies.

LRU Page Replacement

In Last Recently Used (LRU) page replacement algorithm the strategy is as follows:

Replace 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.

If we let S^R be the reverse of a reference string S, then the page-fault rate for the OPT algorithm on S is the same as the page-fault rate for the OPT algorithm on S^R.

Similarly, the page-fault rate for the LRU algorithm on S is the same as the page-fault rate for the LRU algorithm on S^R.

Let consider the following example where number of available page frame is 3.

Let the reference string is:

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

According to the LRU page replacement algorithm, the situation is as follows:

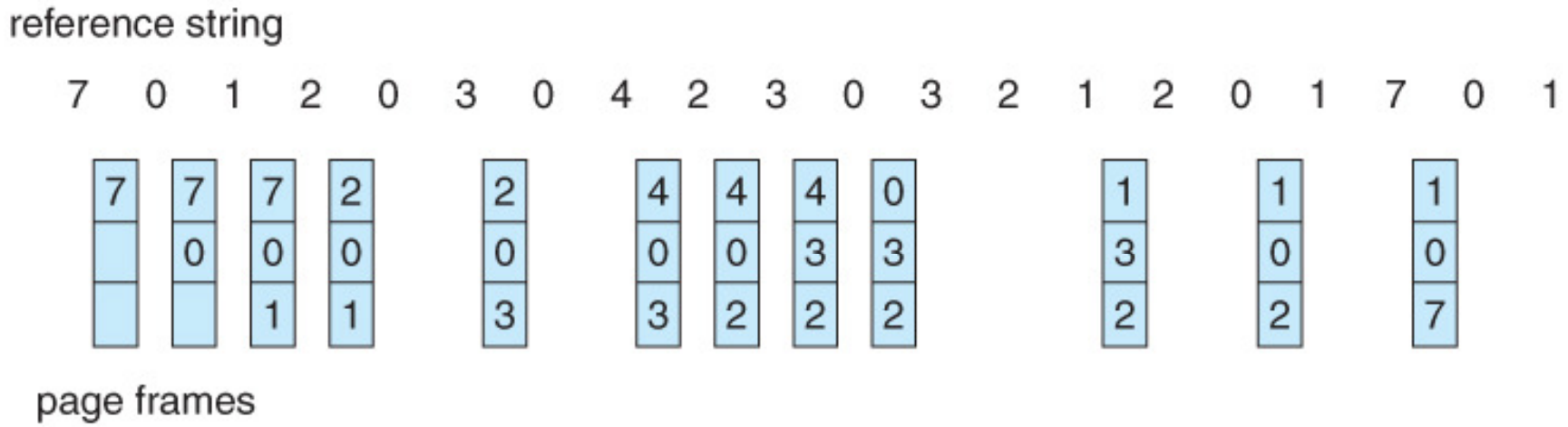

Fig 7: LRU page-replacement algorithm.

With only 12 page faults, the LRU page replacement algorithm is much better than the FIFO algorithm, which results in 15 faults.

This algorithm will ***never suffer from Belady's anomaly***.

Allocation of Frames

The user process can allocate any free frames.

The minimum number of frames per process is defined by the architecture, whereas the maximum number is defined by the amount of available physical memory.

Allocation Algorithms

The easiest way to split *m* frames among *n* processes is to give everyone an equal share, *m*/*n* frames. This scheme is called ***equal allocation***.

The drawback of this technique is that each process will get equal amount of free frames irrespective to their size as well as requirements.

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

$$S = \Sigma s_i.$$

Then, if the total number of available frames is m, we allocate a_i frames to process p_i, then

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

It can be observed that, with either equal or proportional allocation, a high-priority process is treated the same as a low-priority process.

By its definition, however, we may want to give the high-priority process more memory to speed its execution, to the detriment of low-priority processes.

One solution is to use a proportional allocation scheme wherein the ratio of frames depends not on the relative sizes of processes but rather on the priorities of processes or on a combination of size and priority.

Global versus Local Allocation

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.

For example, in case of global replacement, we allow a high priority processes to select frames from low-priority processes for replacement.

A high priority 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.

Thrashing

The high paging activity is called thrashing.

A process is thrashing if it is spending more time paging than executing.

Cause of Thrashing

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.

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.

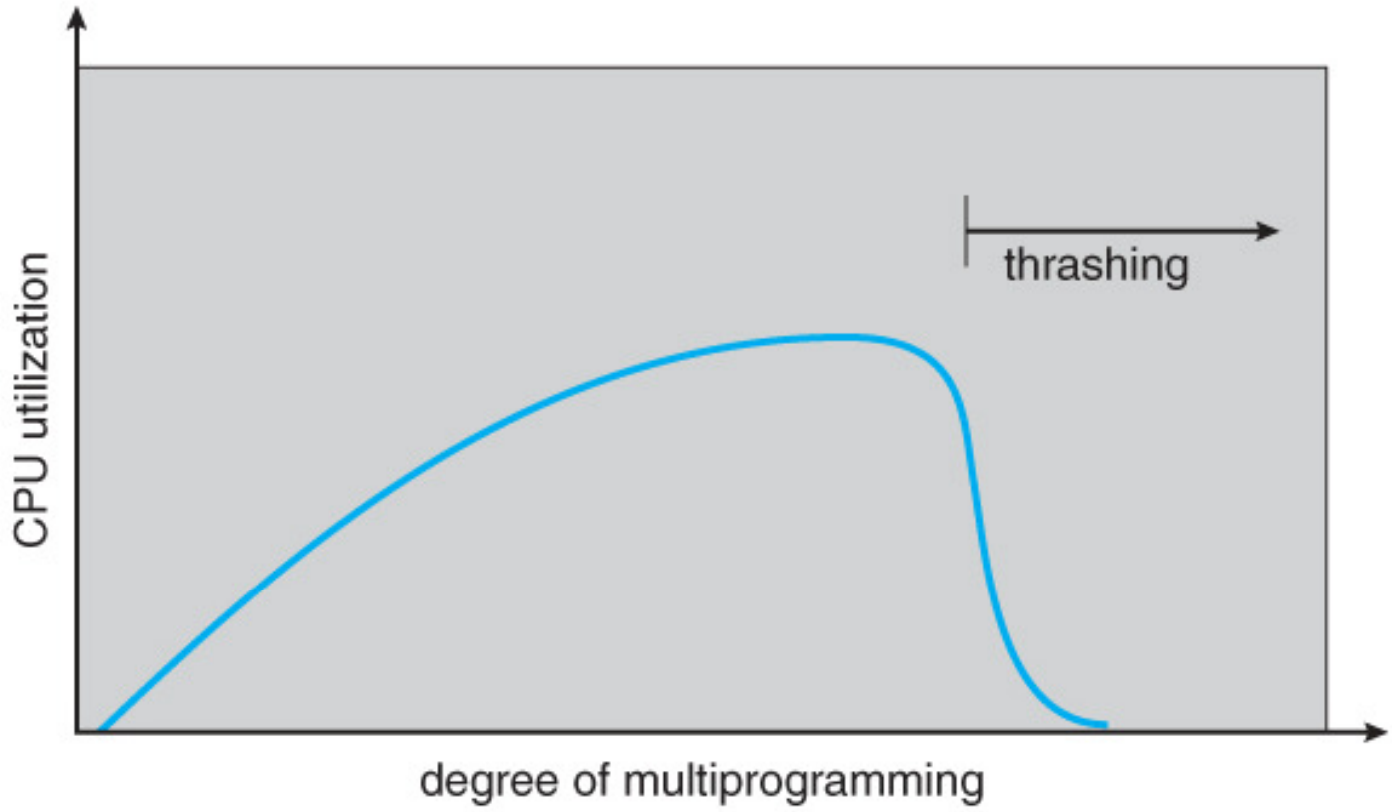

Fig 8: Thrashing

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.

In Fig 8, the CPU utilization is plotted against the degree of multiprogramming.

As the degree of multiprogramming increases, the CPU utilization also increases, although more slowly, until a maximum is reached.

If the degree of multiprogramming is increased 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.

We can limit the effects of thrashing by using a local replacement algorithm (or priority replacement algorithm).

The local replacement requires that each process select from only its own set of allocated frames.

Thus, if one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash as well.

However, the problem is not entirely solved.

If processes are thrashing, they will be in the queue for the paging device most of the time.

The average service time for a page fault will increase because of the longer average queue for the paging device.

Thus, the effective access time will increase even for a process that is not thrashing.

To prevent thrashing, we must know how many frames a process needs.

But it is very difficult to know the frame requirements of a process.