Chapter 9

Virtual Memory

Learning Objectives

After reading this chapter, you should be able to:

- Explain how large programs can be executed on computer systems with a main memory of limited capacity;
- Describe how to simulate a large virtual memory system, on the top of small physical memory, by using a backing store;
- Discuss two virtual-memory-management schemes, namely demand paging and demand segmentation;
- Appreciate the usefulness of the concept of a working set to improve system performance.

9.1 Introduction

There are two extremes in executing an application program: (1) the entire program is loaded into the main memory before the program execution starts, or (2) an instruction and its required operands are loaded into the main memory right before the instruction execution begins. We have assumed the former extreme in presenting memory-management schemes in Chapter 8. There are two visible shortcomings with the former extreme. (1a) We have to bring the entire process address space into the main memory even though the executing process may not reference every entity in the address space. Even if the process does so, it does not do so at the same real time.[≫] (1b) Processes that do not fit entirely into the main memory cannot be executed. There is one visible shortcoming with the latter extreme. (2a) Program execution speed is extremely slow. In this chapter, we present a different kind of memory management technique that helps keep a balance between these two extremes.

We would like to have a memory management scheme that helps us in executing application processes even if they do not entirely fit in the main memory. We would also like to execute many applications simultaneously even when their cumulative memory requirement exceeds the main memory size. The memory management subsystem should be capable of accommodating these two goals without involving

≫ *A process never needs its entire code base and data at any one time.*

applications in the memory management activities. In addition, the execution speed of processes should be acceptably fast.

Virtual memory, a different kind of memory management scheme, aims to achieve the two above-mentioned goals. It holds computations in backing stores and transfers them, in parts, to the main memory according to demand. It thus creates an illusion≫ to users of a very large (main) memory that can accommodate all computations in their entirety. Implementations of virtual memory and their merits and demerits are discussed in this chapter.

≫ *The virtual keyword is borrowed from Physics (actually, Optics): images formed by mirrors and lenses—the images that are not there but behave as if they are there.*

9.2 Virtual-Memory Concepts

The CPU can make a direct reference to a piece of information only if that information is available in the main memory. To execute a process, the CPU needs to have access to the process's program and data in the main memory. However, the process may not execute every instruction in the program and/or access all data. Even if the process does so, it does not do so at the same real time. Consequently, the information that the CPU needs immediately for a process execution must reside in the main memory, and other information pertaining to the process may reside elsewhere without affecting the process execution—at least its immediate execution. In this way, we can have process address spaces that are larger than the physical memory size as long as we can accommodate the immediately required information in the main memory.

If only the immediately required information (i.e., code and data) is kept in the main memory, it would free parts of the memory from a process address space. Other needy processes can effectively use the parts freed by the process. The CPU is not concerned where the missing parts of the address space are stored. As long as the instruction it wants to execute and the operands required for the instruction execution are in the main memory at the appropriate memory locations, the CPU works seamlessly. Some agent has to bring the missing parts in the memory whenever the CPU needs them.

Traditionally, in such situations (where process address space becomes larger than the main memory), application developers will manually do the information transfer between the main memory and a backup storage; the operating system is not involved. In modern systems, this is however done automatically by the operating system without involving applications. We briefly discuss these in the following subsections.

9.2.1 Manual Overlays

...of the allocated memory, we may ...mory location, ...

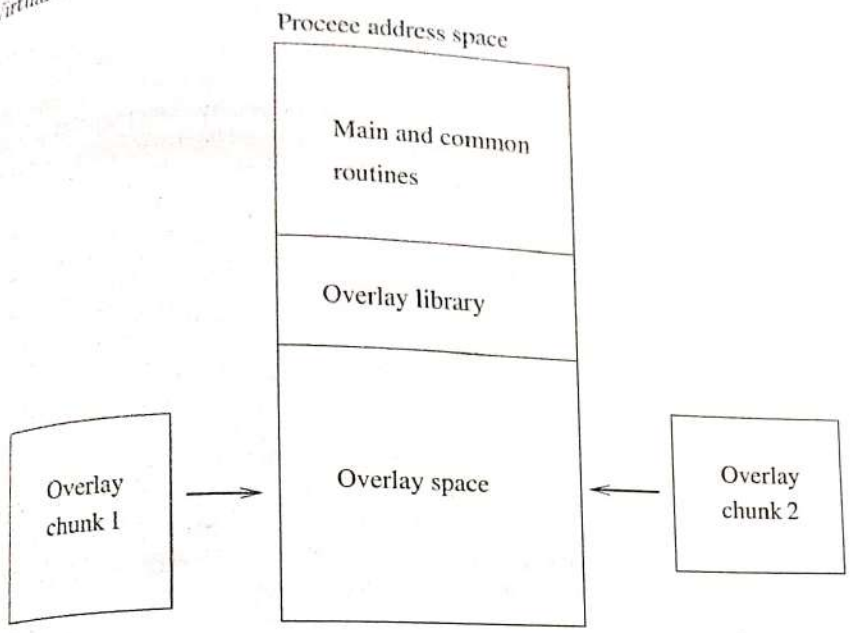

Figure 9.1: *Schematic for overlay*

in the memory. The transfer decision has to be taken by the application, and the operating system is not involved in overlay activities. This we call *manual overlays*. We would like to delegate this work to the operating system.

9.2.2 Need of Automatic Overlays

In early computers, the main memory was quite small. A large part of program development was devoted to managing overlays. Large programs began to be developed with the introduction of higher-level languages such as FORTRAN and Algol. Overlays became a significant problem because application programmers had to do them manually (of course, by careful coding). Naturally, this was not desirable by application programmers. The need of the hour was an automatic transfer mechanism to achieve the overlay feature without involving applications in the memory-management tasks. The motto was that the operating system would solely do the overlay work without any hints from applications. And, in addition, the overlay system must be as efficient as the manual overlays are.[≫]

9.2.3 An Automatic Overlay Infrastructure—A Virtual Memory

To make applications independent of overlays, we need to make certain that they should feel that their entire address spaces reside in the main memory and they behave accordingly. This would make application logic independent of the size of the physical memory. Thereby, we need a large "virtual" memory that could store all processes in their entirety.

≫ *Virtual memory was invented by the designers of the Atlas Computer at the University of Manchester, England, in the 1950s as a solution to the then nagging programming problems such as overlay (i.e., managing data transfer between the main memory and disks), recompilation of programs every time the size of the main memory changed, etc.*

The process address spaces are mapped into the virtual memory and they access the virtual memory as if it is the main memory. Thereby, the limitation on the main memory space becomes transparent to application programmers. The mapping between the virtual memory and the main memory is done by the operating system without the knowledge of applications and processes. Said differently, applications do not include code for overlays.

Figure 9.2 presents an abstract view of the type of a virtual memory. The entire process address spaces reside in the virtual memory. Thereby, applications become independent of the size of the main memory. They run from the virtual memory and, therefore, do not need to do the work of overlay themselves. The virtual memory is a logical entity and its cells/locations are accessed by what we call virtual addresses. These addresses collectively form the virtual address space of the computer system.

Figure 9.2 shows the relationship between the virtual address space and process logical address spaces. The latter are embedded in the former. Each virtual address is a pair (process identifier, private logical address). We always refer to virtual addresses relative to a particular process, and in this book, we naively use the term private or logical address instead of virtual address.

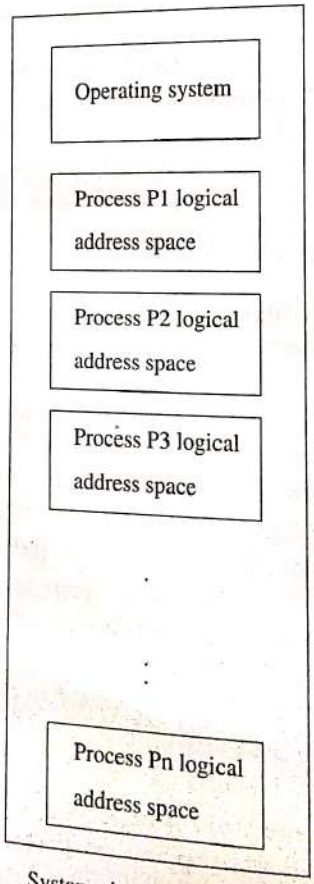

Figure 9.2: *An abstract view of system virtual memory*

9.2.4 Automatic Overlay Management

To implement a virtual memory, the memory system is organized at two levels: (1) a *primary memory* (or directly addressable storage) consisting of only the main memory device, and (2) an *auxiliary memory* (or backup storage) consisting of some secondary storage devices such as disks.[1]

The CPU has direct access to the content of the primary memory, but not to that of the auxiliary memory. To access information in the auxiliary memory, the CPU has to go through the I/O controller of the secondary device that holds the auxiliary memory. Consequently, information retrieval speed from the auxiliary memory is many order magnitude slower compared to that from the primary memory. We need to bridge the speed gap to make virtual memory viable and to satisfy the expectation of computer users.

The information that is not immediately required by the CPU for executions of processes can be moved to, and retained in, the auxiliary memory and brought back into the main memory (from the auxiliary memory) whenever required by the CPU. The main memory holds only the parts of the computations required immediately, and the auxiliary memory all the computations in their entirety.= By so doing, we can accommodate many more computations than the main memory normally does. We can initiate new computations as long as there is space in the auxiliary memory to hold all the computations. The system must also be capable of executing partially loaded computations.

The virtual memory entails managing overlays solely by the operating system. The challenges in implementing a virtual memory are in determining (1) how information is distributed among the two levels of storage devices, (2) how to transfer information from one level to another, (3) when to initiate the transfer, and (4) who initiates the transfer. We need a memory-management scheme that carries out these tasks automatically without the intervention or knowledge of applications. Applications thereby become independent of the size of the main memory, and they do not contain any memory-management functionalities such as overlay. Application programmers are wholly relieved of the burden of memory-management tasks. The processor hardware and the memory-management software of the operating system move missing information into the main memory only when required for processing. In addition, they bind the process logical addresses to their appropriate memory addresses at runtime.=

In summary, the objective is to have a memory-management system that has the following abilities: (1) to load a program into a memory of arbitrary size, (2) to make the program size independent of the availability of the main memory, (3) to run a partially loaded program, (4) to relocate a running program, (5) to vary the amount of physical memory space in use by a given program execution at runtime, and (6) to increase the degree of multiprogramming. In short, it aims to utilize the main memory as effectively as possible.=

[1] For the sake of convenience, we assume that one secondary storage device holds the entire auxiliary memory.

≫ *Here comes a note of caution. A virtual-memory technique simulates a large memory system. It cannot, however, enlarge a process private address space beyond what the processor architecture permits.*

≫ *Without runtime address translation support, implementation of a virtual memory is not possible.*

≫ *One may think that the virtual memory is a simulation of a large RAM using the auxiliary memory, that is, secondary storage devices. We will not discuss the management of the auxiliary memory, that is, swap devices in this book.*

9.3 Memory Organization

We note that the virtual memory is transparent to applications, and they see a large main memory. Although they see a flat view of the main memory, as stated in Section 9.2.4, the operating system designers do not see any more the flat view of memory organization that we had assumed in Chapter 8. The memory system consists of two levels of storage devices: (1) the main memory itself, which is at the primary level and (2) the auxiliary memory consisting of secondary storage device(s).

High-speed online storage devices such as disks are normally used to implement the auxiliary memory; in the present context, they are often called *swap devices* or *backing stores*, and the auxiliary memory is called the system *swap space*. (We use the two terminologies—auxiliary memory and swap space—to mean the same.) The size of the auxiliary memory is determined by the total amount of information that can be stored in swap devices, and it is normally significantly larger than that of the main memory or process address space. The effective size of the memory system is determined by the size of the auxiliary memory.

The auxiliary memory is used only to backup computations, that is, to hold images of logical address spaces. In fact, computations are held in the auxiliary memory in their entirety, and they are brought into the main memory, in parts, whenever the CPU needs them. A process is normally allocated a tiny part of the main memory, and the process sees its entire address space through this tiny allotment (see Fig. 9.3). The tiny space acts as a cache for the process address space. The main memory acts as a "computation cache" of the auxiliary memory, and the virtual-memory-management system is in fact an implementation of this computation cache.

≫ *Maintaining a small set of data in a high-speed device for faster access from a larger data set in a low-speed device is called caching, and the high-speed device is referred to as a cache.*

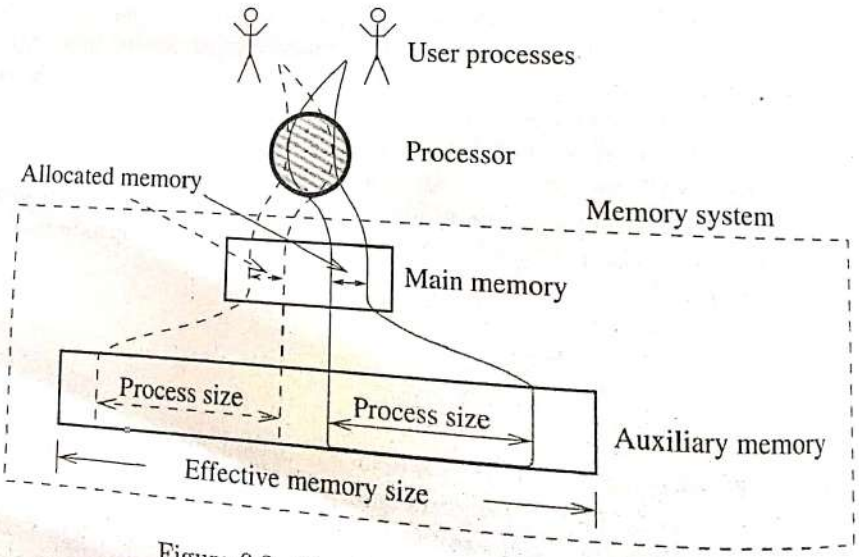

Figure 9.3: *Two-level memory organization*

Computations are automatically transferred, in parts, back and forth between the main memory and the auxiliary memory without the knowledge of applications or executing processes. However, such transfers, if done frequently, may slow down the computation speed considerably. The operating system must avoid random juggling of information between the two devices to assure a higher quality of service

9.4 Virtual-memory Implementation

Virtual-memory management is a technique of implementing the computation cache mentioned in Section 9.3. As in any other cache system, here the main memory holds only parts of a process address space (see Fig. 9.3). The computation cache though is transparent to application processes. Each process thinks that its entire address space is in the main memory and it behaves accordingly.

Application programmers may not worry about how much of physical memory is available in the system or how it is allocated to processes. A process address space can reach the maximum limit supported by the processor architecture irrespective of the actual physical memory allocated to it. Such processors should be able to decode addresses that are longer than the physical memory addresses. Processor MMU should be capable of transforming larger logical addresses into shorter physical addresses. Note that the number of its address pins connected to the system address bus limits the range of physical memory a processor can reference. A process logical address space, however, can be larger than that. The virtual-memory implementation (with help from processor hardware) makes the system appear to have more physical memory than it actually does. In fact, virtual memory creates the illusion[≈] that the system has a very large main memory at its disposal even though the computer actually has a relatively small amount of main memory.

Virtual-memory implementation is a logical layer that sits between application processes and the memory-management subsystem proper (see Fig. 3.2 on page 102). All it does is that it moves, that is, swaps out the information not immediately required by the CPU from the main memory to the auxiliary memory and brings back the swapped out information into the main memory later only when the CPU needs it. It judiciously implements three rules of cache management: fetch, placement, and replacement. A *fetch rule* determines when a piece of information is brought into the main memory from the auxiliary memory.[≈] A *placement rule* determines which parts of the main memory will contain the fetched information. A *replacement rule* determines what information is removed from the main memory to make room for new information. Computer hardware and operating system together carry out these three rules automatically without involving applications or processes. Implementations of these three rules are discussed in the rest of this chapter.

There are two primary tasks of the virtual-memory manager: (1) it propitiously moves information from the auxiliary memory into the main memory when and only when the CPU requires the information, and (2) it binds process logical addresses to the main memory locations that happen to contain the required information.[≈] (Note Figure 9.4 presents a schematic of a typical virtual-memory implementation. that no real implementation is structured as in that model.) The auxiliary memory is hidden from the processes.

As shown in Fig. 9.4, the process address space is automatically folded by the system. Only the immediately required folds are kept in the main memory. Thus, it is possible to run a process that may not entirely fit in the main memory. The system may initiate more program executions than the main memory can hold in their entirety. As long as there is space available in the auxiliary memory, the system may initiate new computations. The size of the auxiliary memory is one factor limiting the degree of multiprogramming. Virtual memory is an implementation of

≈ *The name, which seems to have been borrowed from optics, recalls virtual images formed in mirrors and lenses—images that are not there but behave as if they are.*

≈ *To improve system performance, many operating systems do anticipatory prefetching of information from the auxiliary memory to the main memory. We do not discuss prefetching in this book.*

≈ *There are small devices such as cell phones without any backing store. The modified parts are always kept in the main memory, but other parts may be removed. When needs arise, they are retrieved from their source files.*

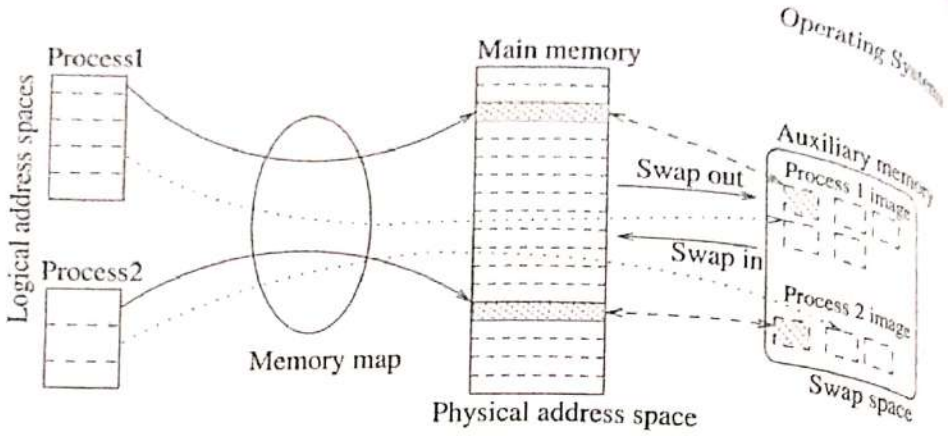

Figure 9.4: *Virtual-memory implementation scheme*

a large simulated RAM, with the memory system performance close to that of a real RAM. How this is done is discussed next.

9.5 Demand Fetching

Unlike in the pure memory-management schemes we discussed in Chapter 8, the entire logical address space of a process may not be loaded into the main memory before the process execution begins. In fact, a process may start its execution even if no part of its address space is in the main memory.= A part of the address space is brought into the main memory when and only when the process attempts to use it for the first time.

≫ *Demand fetching would phenomenally speed up initialization of a program execution.*

The CPU continues a process execution until it finds that some required information (instruction or data) is not available in the main memory. In that case, the CPU would have to suspend the process execution. The part containing the missing information must be brought into the main memory before the process execution can be resumed. The address translation system must know precisely which parts of an address space are currently in the main memory and where exactly they reside there.

When the CPU references a part that is not in the main memory, the address translation unit raises a missing address exception and suspends the process execution. The operating system handles the exception and brings the missing part into the main memory from the auxiliary memory. Once the part is brought into the memory, only then can the execution of the suspended process be resumed from the instruction whose execution caused the exception. The fetch rule here is to bring a part of address space into the main memory only when that part is required for process execution. This is known as *demand fetching*—fetching information from the auxiliary memory on demand. A demand fetch avoids bringing into the main memory the non-obligatory parts of the process address space. If a process never references some parts of its programs, those parts may never be brought into the main memory.[2]

[2] In the previous chapter, we talked about loading a program into the main memory by a loader. Actually, in modern operating systems such as Linux, the loader may not load the program. It instead sets up a mapping between program/library file(s) and process address space. This eliminates the initial load latency. It can though load a tiny part in the main memory. Some

So far, in this chapter we have used "parts of address space" in a generic sense without mentioning what exactly constitute a part. A part can be as small as a single memory location or as large as an entire logical address space. Transferring an entire address space between the main memory and the auxiliary memory is a time consuming operation. In addition, transferring a single word from a swap device takes nearly the same amount of time to transfer a single block of words. Parts are normally a tiny fraction of address space, and are "units" of memory allocation and deallocation and transfer.

Implementation of a virtual memory requires support from the processor hardware and depends on what runtime support the processor provides to the operating system. There are a variety of virtual-memory-management schemes developed in the past. The schemes depend on a number of factors such as whether memory allocation and deallocation units are fixed-size pages or variable-size segments, whether the segments are paged, whether memory units are sharable or non sharable, etc.

We discussed two pure memory-management schemes (namely, segmentation and paging) in Chapter 8. Virtual memory is implemented on the top of both memory-management schemes. They are called demand segmentation and demand paging, respectively. Demand paging is the most common implementation of virtual memory. Nonetheless, a few systems implement demand segmentation. Segment replacement is more problematic than page replacement because segments are of variable size. These two schemes are discussed in the following two sections; we primarily concentrate on demand paging that we present first.

9.6 Demand Paging

Demand paging is a page-based implementation of virtual memory. (We discussed the pure paging scheme in Section 8.6. You may revisit that section first.) Here units of memory allocation and deallocation are fixed-size frames. The main memory is treated as a "page cache" of computations that are held in entirety in the auxiliary memory. The philosophy is to keep in the main memory only those pages of a process currently required for the process's immediate execution requirement. Even if the CPU accesses a single location in a page, the entire page has to be kept in the main memory.[≫] Thus, a "part of address space" is a page for this scheme.

When the operating system wants to execute a process, it brings parts (the few necessary pages) of the process address space into the main memory instead of the entire address space. In the extreme case, the operating system may start executing a process no page of whose address space is in the main memory. This scheme is called *pure demand paging*. In this scheme, a frame allocation for a page is deferred until the process references the page for the first time. The fetch rule is to bring in just one page (that contains the missed reference) at a time. Demand paging removes pages from the main memory only through a replacement algorithm (discussed later in this section).[≫]

The two main questions we need to answer for implementing demand paging are (1) how do we determine which pages of a process address space are not in the main memory and (2) what do we do when a process references a missing page? Another important question is how do we determine which pages of an address

≫ *Some modern devices such as cell phones use a limited form of demand paging to manage their memory: they apply demand paging only on program text, but not on data, and there is no need of a swap device. The auxiliary memory is composed of application storage in flash.*

≫ *Early UNIX systems would swap-out and swap-in the entire process address space.*

operating systems such as Windows XP maintain a small history of application initialization. They bring, i.e., prefetch immediately, required pages before the application execution begins. Other than that, they follow demand fetching.

space definitely need to reside in the main memory. We discuss answers to these three questions systematically in the rest of this section.

9.6.1 Address Translation and Page Fault

The address translation mechanism under demand paging is almost similar to that of the pure paging scheme we discussed in Section 8.6.2 on page 339. As in the pure paging scheme, the operating system maintains a page table for each process, for the purpose of address translation. Each logical address generated by the CPU is translated into a physical memory address by the processor address translation unit using the corresponding page table. The steps involved in the address translation are the same as those in the pure paging scheme except on one count, which we describe below.

We need to keep track of missing pages. Unlike in pure paging, each page descriptor carries some additional information for this purpose. There is a special bit of information in the descriptor, whose value indicates whether the page is present in the main memory. It is called the *presence bit*. (It is different from the valid/invalid bit.) Initially, the bit value is false, indicating that the page is not in the main memory. When the page is brought into the main memory, the bit is set to true. If the page is transferred back to the auxiliary memory (or simply removed from the main memory), the bit is reset to false.

When the CPU references a page, the address translation unit checks the value of the presence bit in the page descriptor. If the bit value is true, the page is in the main memory and the processor can reference the corresponding page frame directly and the address translation proceeds normally. If the bit value is false, the page is absent in the main memory and the address translation unit raises a *missing page* or *page fault exception*. The exception breaks the process execution and the operating system executes a page fault handler routine.[≫]

The page fault handler routine first analyzes whether the address that has caused the exception is valid. If the address falls outside the process address space, then the application may be malfunctioning and the operating system may terminate the process. Otherwise, the page fault handler brings the appropriate page image from the auxiliary memory into the main memory. On the completion of the page transfer, it sets the presence bit in the page descriptor to true to indicate that the page is now in the main memory. The process then resumes from the same machine instruction whose execution caused the page fault exception. This time when the CPU re-executes the same instruction and regenerates the same reference, the translation unit will find the page in the main memory and will not raise a page fault exception for the page. The fetch rule is never to allocate a memory frame for a page and never to load that page image until a page fault exception occurs for that page.

Figure 9.5 presents a typical state of a process's page table. Page 0 is not in the main memory, and hence its presence bit in the page descriptor is set to 0 (false); the base address in the descriptor may point to the auxiliary memory where the page image is available. Page 1 is in the main memory and the presence bit in the page descriptor is set to 1 (true); the base contains the address of the frame that is currently holding the page. A reference to page 0 will cause a page fault, and only then the operating system will bring the page image into a memory frame. Pages that are absent have no effects on the process if it does not attempt to reference them. The address translation proceeds normally for accesses to pages that are

≫ *This exception is not due to programming error, but due to operating system design choices. The operating system alone rectifies the situation and resumes the process from the exception point. We will see in Chapter 12 how exceptions get connected to their respective handlers.*

present in the main memory; as the hardware does the checking of the presence bit, there is no noticeable degradation of the address translation speed.

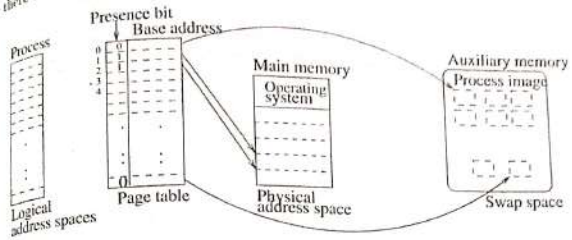

Figure 9.5: *Address binding in demand paging*

9.6.2 The Effect of Page Fault

If the operating system can guess correctly and bring in all the required pages into the main memory before it schedules a process for execution, the execution will proceed normally and will not generate even a single page fault exception. Otherwise, the process generates page fault exceptions, and hence, it sees a significant slowdown in its computation speed if the page fault rate is too high. (The *page fault rate* is defined as the number of page faults per memory reference.)

When the CPU references a page in the main memory, it takes the processor one memory cycle to access any location from the page frame. The access time is typically in the order of nanoseconds. If the page is not in the main memory, the reference causes a page fault; the page image must be brought from the auxiliary memory into the main memory before the process can reference a location from the page. The auxiliary memory access time is typically in the order of milliseconds. Sometimes, due to lack of space in the main memory, we need to flush a frame out of the main memory to the auxiliary memory before we can bring in the required page. This will cause the page fault service time to double. Thus, there is a tremendous increase in the memory reference time upon a page fault. To make the demand paging scheme effective, we have to reduce the page fault rate as much as possible.

There are two parameters to analyze page fault behavior. Suppose the processor requires t time units, on the average, to access a memory cell, and suppose a swap device requires, on the average, T time units to transfer a page between the main memory and the swap device. (The parameter t includes delays due to bus arbitration and physical memory access time. The parameter T includes kernel processing time, waiting time in the device queue, actual transfer time, and device interrupt service time. Usually t is about 50–100 ns, and T 20–30 ms for the disk as "swap device".)

The fault rate of a process depends on what fraction f of the process address space is in the main memory. Let $p(f)$ be the probability that a memory reference would cause a page fault when f fraction of the address space is in the main memory. If f is 1, that is, 100% of the address space is in the main memory, then there is no page fault and every memory reference takes t time units. When $0 < f < 1$, the

average reference time increases from t to approximately about $t + p(f)T$. It is the duty of the demand paging system to keep f as low as possible to effectively utilize the main memory, and at the same time keep $p(f)$ as low as possible to improve the speed of the processes to make virtual-memory simulation viable. Otherwise, if $p(f)$ is too high, the responsiveness of the system might become extremely poor.

The objectives of low f (for higher memory utilization) and low $p(f)$ (for higher process speed) may appear to be conflicting. They are indeed conflicting for a process that generates random and uniformly distributed memory references in its address space. Then $p(f)$, which is approximately $1 - f$, becomes high. The demand paging system may not be able to do much for that process, and the process execution speed will be hopelessly slow for poorer values of f. The system is then forced to keep more and more pages of the process address space in the main memory, causing poorer effective utilization of the memory space. On the other hand, if a process's program is well structured, its references will follow some pattern in small neighbourhoods in the address space, and the system will be able to ensure low page fault rate with only a limited fraction of the process address space in the main memory. This is known as *locality of reference*. The better the locality, the better the expected performance from the demand paging system. We will study the effect of locality on performance in Section 9.6.8 on page 376.

9.6.3 Execution Restart upon Page Fault

When an instruction execution causes a page fault exception, the CPU aborts the instruction execution and suspends the current execution flow of the running process. The CPU resumes the suspended flow by re-executing the same instruction after the page fault service. It is a critical issue for the implementation of any virtual-memory-management scheme.

We note that, upon the page fault service, the process has to be restored to the same state prior to the commencement of the original (faulting) instruction execution. When the exception occurs, the system saves the current state of the process. When the exception service is complete, the system restores the saved state. At this point, the required page is in the main memory, and the presence bit in the page descriptor is set to 1. The process restarts executing the same instruction as if nothing had happened to it. Unfortunately, things may not be as simple for some instruction re-executions as the process state saving may not reflect the "true" state of the instruction execution just before the page fault exception occurred. For instance, the saved state does not contain the state of CPU's internal hardware units, and this might cause applications to malfunction.

The most critical requirement is that the system must be able to restore the process state for re-execution of every instruction even if the instruction is partially executed before the page fault exception has occurred. Note that a page fault may occur at any point in a program execution—instruction fetch, operand read, or result write. If a page fault occurs on an instruction fetch, the same instruction fetch resumes after the exception service. An instruction execution may involve reading some operands from the main memory or CPU registers and writing out values to memory locations or CPU registers. Most instructions write a single value at the end of their executions. If a page fault occurs on an operand read or the final write, there is a need to refetch and re-execute the same instruction after the exception service. (The refetch and re-execution are necessary because the processor does not remember the precise execution state of the instruction.) The difficulty arises with

those instructions that modify multiple data in memory locations and/or CPU registers, and a page fault occurs after partial modifications. Examples include multimove instructions and auto increment/decrement instructions. We need to cancel the effects of the partial execution, before we can re-execute an instruction upon the page fault service.

Let us consider a simple auto decrement instruction execution and its effect on the system. The instruction execution first decrements the value of a register and then uses that value as a memory address to read/write data in the main memory. If a page fault occurs at the memory reference, the value of the register is already decremented. Upon page fault service, when the same instruction is re-executed, it starts with a different value of the register and causes the application to malfunction.

One solution to the above-mentioned problem is that the processor microcode first determines all possible addresses that an instruction execution can modify and ensures that those pages are present in the main memory before the instruction execution begins. Another solution is to keep the original values in some temporary register before updating them, so that upon a page fault those original values can be restored. The number of temporary registers will limit the maximum number of pages that the instructions can modify.

» *Auto increment/decrement instructions are very effective in traversing linear array elements.*

9.6.4 Frame Allocation Policies

Frames are units of memory allocation. Frames are also units of data transfer between the main memory and the auxiliary memory. If a page that is referenced by the CPU is not in the main memory, the hardware page translation unit generates a page fault exception. The page fault exception handler will bring the image of the page from the auxiliary memory into the main memory and update the page descriptor accordingly. Therefore, the foremost task of the page fault handler is to find a free frame for the new page. The placement policy that demand paging follows determines from where a free frame will come.

The allocation policy of the memory manager distributes available frames to competing processes. The question is how many frames have to be allocated to a process. In the extreme case, a process might be able to run with just one page frame allocated to it. On every reference to a new page, the process will generate a page fault exception. Page faults will hopelessly slow down the process execution speed. Another aspect is that some instruction executions need one or more operands from the main memory, and they need multiple pages present in the main memory at the same time. Hence, every process needs a bare minimum number of frames without which its execution may not be possible. The processor architecture defines the minimum number. (If a process cannot be allocated the minimum number of frames, the operating system must suspend the process execution.) Nonetheless, more than the minimum number of frames is needed for a process to reduce its page fault rate. Of course, the more frames allocated to a process, the better for the process.

What fraction of total frames a process gets depends on the system's allocation policy. The policy may be an equal share (i.e., every process gets the same number of frames), or a proportional share (i.e., proportional to their address space size), or some priority-based sharing, or based on immediate requirements of processes to reduce their page fault rates, etc.

When a page fault occurs, the system's placement policy identifies a set of po-

tential frames for the new page. The one that is selected from the set is determined by the system's replacement policy. (We will discuss some replacement algorithms in Section 9.6.5.) Various placement policies have been investigated in literature. A placement scheme can be local, global, or mixed. In a local placement scheme, each process starts with a preallocated number of frames as determined by the allocation policy. The replacements come from its own set of allocated frames. That is, the potential frame in which a new page is loaded comes from its own share of frames. By contrast, in a global placement scheme, any frame (even if it is allocated to another process) can be chosen, that is, the placement set consists of all frames in the system.

- **Local policy**: Memory frames are partitioned and allocated to processes. The size of a partition is system defined and remains fixed until the system allows a change. When needed, the replacement algorithm is applied to the process's own share of frames. A process cannot use any free frames or take away frames from other processes. A shortcoming of this policy is that not all frames in a partition may be required for a process execution, thereby causing a kind of frame wastage and memory underutilization.

- **Global policy**: When needed, the replacement algorithm is applied to all frames. A process can take away frames from other processes.[≫] Thereby, the size of a process's allocated set of frames may expand or shrink. One shortcoming of this policy is that a process cannot control its own page fault rate. (Note especially that in some practical systems kernel frames are never replaced.)

- **Mixed policy**: There are systems that primarily follow the local policy. However, when the needs arise, they go for the global policy. They maintain a global pool of free frames. Frames from individual processes are added to the global pool at regular interval during frame reclamation.[≫] When real needs arise, frames from the global pool are allocated.

≫ *In priority-based systems, to avoid priority inversion and performance degradation, only high-priority processes may be allowed to take away frames from low-priority processes, and not the other way.*

≫ *Some practical systems such as Windows XP use a mixed placement scheme.*

9.6.5 Page Replacement Algorithms

When a process requests a new page, the placement algorithm identifies some frames in one of which the new page would reside. If all these frames are already occupied, one of them needs to be recycled. Otherwise, there is no way the process can run to completion because fewer frames are allocated to pages than required by the process. The one recycled is called a *victim frame*, and the corresponding page(s) that are mapped to the frame are called *victim page(s)*. An algorithm used to victimize a frame/page is called a *page-replacement algorithm*. A replacement algorithm selects a victim from a set of candidate frames (that are decided by the placement policy).

Note that victim frame has a copy in the auxiliary memory. If the victim is an exact copy of its image in the auxiliary memory, the image of the new page blindly replaces the victim. Otherwise, it implies that the victim frame has been modified recently (and the in-memory copy is more recent than its auxiliary memory copy). In this situation, the victim is called a *dirty frame*. The victim frame is written back onto its auxiliary memory copy first, and then replaced. In either case, the victim page is marked absent in the page table (or multiple page tables, if the frame is shared).

The operating system uses a control bit in each page descriptor to determine whether its page frame is to be written back to the auxiliary memory. The control bit is usually called *modified bit* or *dirty bit*. When a page image is brought into a memory frame, the dirty bit in the page descriptor is reset to false. When the page is modified (at any offset in the page), the dirty bit is set to true by the address translation hardware. Once the bit is set, it remains true until the page is discarded or replaced by another one. A victim is written back to the auxiliary memory only if it is dirty.

The objective of a page-replacement algorithm is to minimize the number of page faults. Consequently, it is necessary to maximize the time between page faults. The chief problem of a replacement algorithm is to select the right victims to reduce the page fault rate. The best choice is to victimize those pages that are least likelihood of being reused in the immediate future. This is called the *optimal page replacement algorithm*; it selects a victim whose next reference is farthest in the future. Unfortunately, we rarely have knowledge about the future. In practice, all replacement algorithms are based on some heuristics, a limited amount of immediate past history, or some plausible assumptions.

In the following sub-subsections, we study three simple replacement algorithms, and their merits and demerits. For these algorithms, we assume a local allocation policy, that is, the replacement comes from the frames allocated to the process.

9.6.5.1 First-in-first-out Replacement Algorithm

First-in-first-out (FIFO) is the simplest page-replacement policy. As the name suggests, the page frame that is brought into the process allocation set first is the one chosen as the victim when the process generates the next page fault. This strategy is based on the assumption that programs tend to follow sequences of instructions so that references in the immediate future are most likely be close to present references; so, discard the oldest frame. This scheme is easy to implement. However, its performance may not be satisfactory. For example, a heavily used page may be replaced because the page is brought into the main memory first, and it is returned to the main memory almost immediately.

To understand this replacement algorithm better, consider a process that is five pages long and that makes references to its pages in this sequence: 1,2,3,4,1,2, 5,1,2,3,4,5. Suppose the system allocates three frames to the process. Figure 9.6 presents the fault pattern for the reference string, and each block (containing three small boxes, each representing a memory frame) represents a system state/configuration after each page reference. Since this is a pure demand paging system, the initial configuration is empty, indicating no page of the process is in the memory. As shown in the figure, each block is under-stamped by the page number whose reference occurs in the state. When we have a page fault in a state, "F" marks the top of the corresponding block. For example, in the figure, page 1 is referenced at the initial state, causing a page fault and the given reference string produces nine page faults.

Generally, the page fault rate decreases with an increase in the size of the allocation set. However, in some cases, the FIFO algorithm increases page faults when more frames are allocated to processes. This counterintuitive phenomenon is explained with the aforementioned example. The fault pattern for the same reference string with four frames is shown in Fig. 9.7. The same reference string produces ten page faults! This anomaly was originally observed by Belady and his

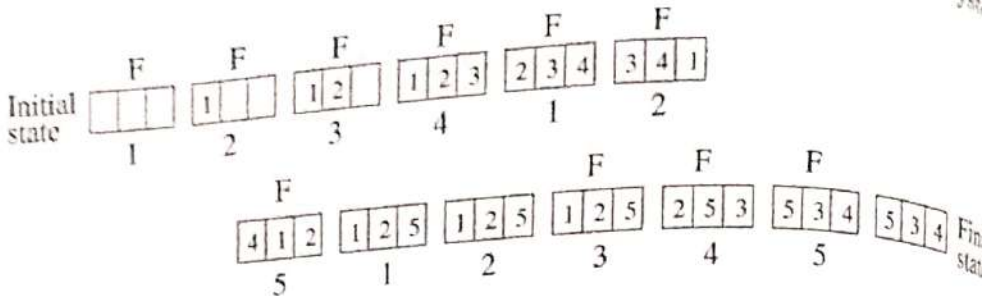

Figure 9.6: *A page fault scenario in FIFO replacement scheme with three fram*

team, and it is known as the *Belady anomaly*.

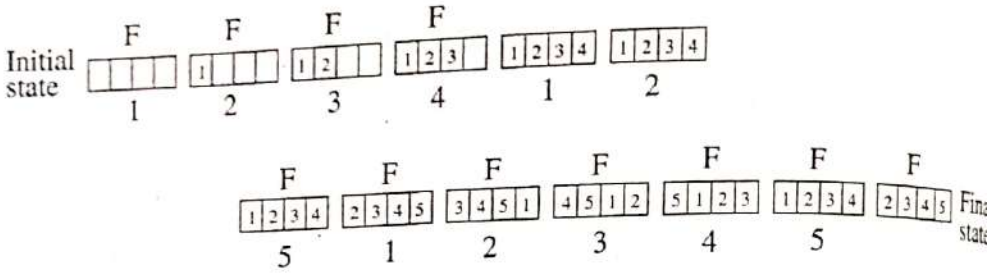

Figure 9.7: *Page fault scenario in FIFO replacement scheme with four frames*

9.6.5.2 Second-chance Replacement Algorithm

This scheme, originally called "Not Recently Used" in UNIX, is a FIFO schem with a difference. It has the simplicity of FIFO, and it avoids replacing heavi used pages. However, it needs more hardware supports. It associates a boolea bit called *reference bit* with each page, and the bit is stored in the page descripto Initially the bit is 0. When a page is referenced (read or written), the hardwa sets the bit to 1. The replacement algorithm replaces the oldest unreferenced pag (i.e., with reference bit 0). If the oldest page has its reference bit as 1, then th algorithm resets the bit to 0 and puts the page at the end of the FIFO queue givin the page a "second chance" (to survive victimization) as if it has arrived by the i.e., becomes the newest page. In that case, the next oldest page is similarly trie for replacement.

Figure 9.8 displays the page fault pattern for this algorithm on the same re erence string noted in the previous sub-subsection with four frames. The value each reference bit for a frame is noted on the top of the small box. Initially all re erence bits are 0. There is a small arrow for each configuration. The frame (sma box) to which the arrow points is the next potential candidate for replacement. its reference bit is 0, it is replaced; otherwise, it gets a second chance to survive an the arrow is moved forward. The given reference string produces 10 page faults.

Although the configurations in Fig. 9.8 are the same as those in Fig. 9.7, in re ality, FIFO and second chance produce different configuration blocks. For example you may extend the given reference string by "2,1" and work out the new string for both FIFO and second chance, and you will see that the last configurations are different.

In addition to reference bits (space overhead), this algorithm has time overhead too. In the worst case, this algorithm takes $O(n)$ time to select a victim page, where

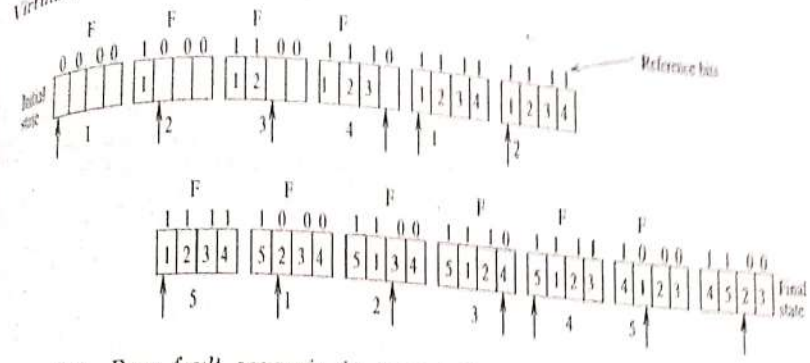

Figure 9.8: *Page fault scenario in second-chance replacement scheme with four frames*

n is the size of the replacement set; but FIFO does the selection in a constant time.

9.6.5.3 Least-recently-used Replacement Algorithm

In the least-recently-used (LRU) scheme, the page not referenced for the longest interval (in the past) is selected as the victim. Figure 9.9 displays the page fault pattern for this algorithm on the same reference string of Section 9.6.5.1 on page 371 with four frames. It produces eight page faults for the reference string.

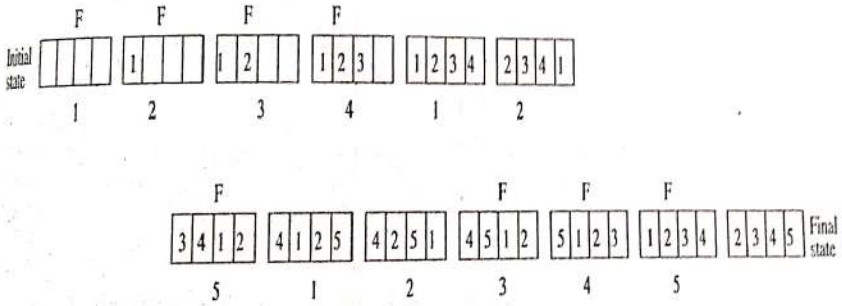

Figure 9.9: *Page fault scenario in LRU-replacement scheme with four frames*

The scheme does not have the defect (Belady anomaly) of the FIFO replacement scheme, and it produces acceptably good results in practice.[≫] However, this scheme is relatively expensive to implement. It requires a timestamp generator or counter to stamp page descriptors on every reference to pages. The counter is incremented every time the CPU makes a memory reference. The algorithm chooses a page frame with the least counter value as the victim.

One drawback of this scheme is that the timestamp structure occupies substantial amount of space in the page table. An execution of the algorithm takes time proportional to an order of the size of the replacement set to find the victim. In addition, every reference causes an update on the timestamp of the referenced page. Another consideration is that the timestamp generator must not overflow.

Operating systems normally do not implement the pure LRU scheme as it needs substantial hardware support and few processors possess it. The systems make some

≫ *LRU is a stack algorithm. A stack algorithm has the following property: when the algorithm is run with more frames, it keeps at least those pages it would have kept if it were run with fewer frames. FIFO is not a stack algorithm.*

≫ *If no hardware support is available, FIFO replacement is the best choice.*

adjustments to the pure LRU policy. Some systems associate a reference bit with each page descriptor.** Initially, the bit is reset to 0 (false) by the operating system. The bit is set to 1 (true) by the hardware when the page is referenced (read or written). At periodic intervals, the operating system makes copies of the reference bits and resets them to 0 (false). The operating system keeps a finite collection of these bit values. When the page-replacement algorithm is executed, it looks through the current and previous copies of reference bits. The pages whose (current) reference bits are true are the ones recently referenced by the CPU. However, the relative order of references cannot be determined. For each page, its reference bit values are organized (with recent ones positioned to the left) in a way that they represent unsigned integer numbers (see Fig. 9.10). One page whose reference bit values represent the lowest number is replaced. (In the figure, page frame 2 will be replaced.) In the case of a tie, either all pages with the lowest number are replaced or one is chosen at random.

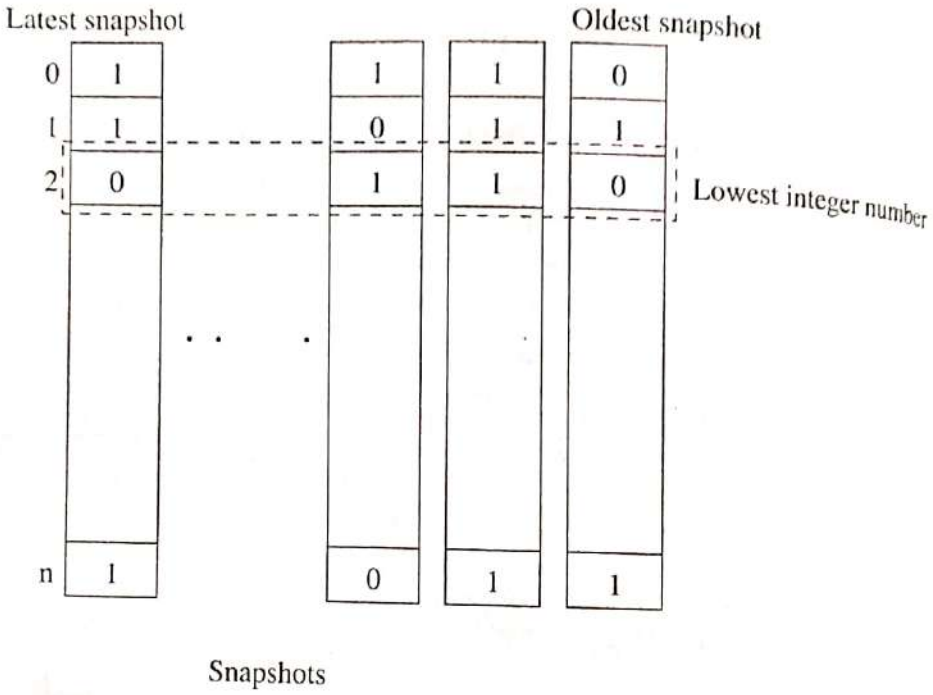

Figure 9.10: *LRU approximation using reference bits*

Let us apply this algorithm on our example reference string (from Section 9.6.5.1 on page 371), and determine how many page faults does it induce. Suppose the operating system clears the reference bits after every four-page references. At the end of the first four-page references, the snapshot is given in Fig. 9.11(a). These four references cause four page faults. The next two references to pages 1 and 2 do not cause page faults. Reference to the page 5 causes a page fault, and we need to victimize a page. At this point, the snapshot is given in Fig. 9.11(b). Either page 3 or 4 can be replaced. Suppose we replace page 3 by page 5. At this point, the snapshot is given in Fig. 9.11(c). The next reference to page 1 does not cause a page fault, and the snapshot configuration remains the same. The next access to page 2 does not cause a page fault, but the next access to page 3 causes one. At this point, the snapshot configuration is in Fig. 9.11(d), and we replace page 4 by page 3, and the resulting snapshot configuration is in Fig. 9.11(e). The next access

to page 4 replaces page 5, and the next access to page 5 replaces page 1. The final configuration is in Fig. 9.11(f). This scheme produces eight-page faults.

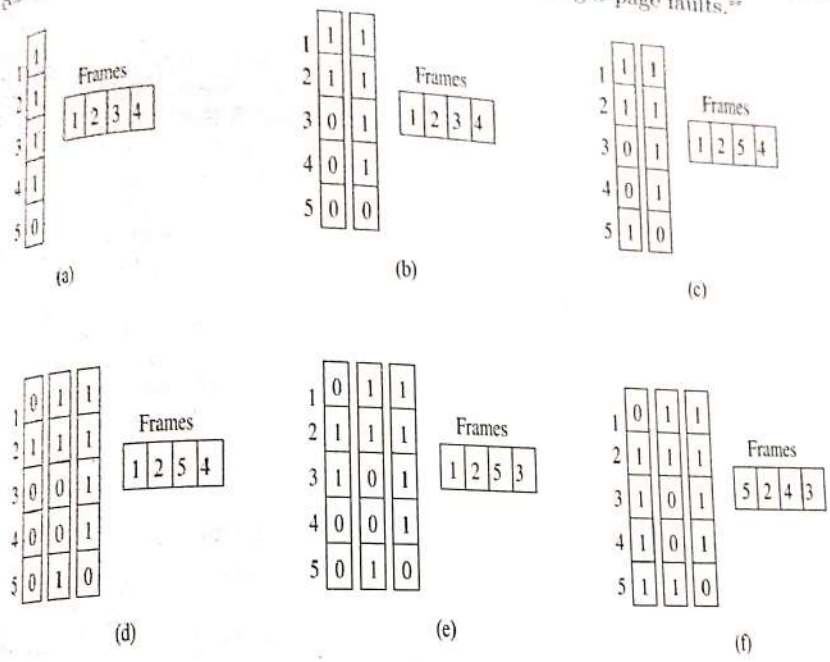

Figure 9.11: *Page fault scenario in modified LRU-replacement scheme with four frames*

There are many other approximations to the pure LRU policy implemented in operating systems. Some algorithms take dirty bits into consideration to determine a victim; they avoid replacing dirty pages as long as possible.

9.6.6 The Constraints on Page-replacement Algorithms

Whatever replacement algorithm is followed, the operating system must not replace a frame that is in use, say by an I/O operation. One solution to this is to do the I/O only in the kernel space whose frames are never replaced in many operating systems. In these systems, there are two levels of data transfer: (1) between the user space and the kernel space and (2) between the kernel space and the I/O devices. An I/O operation causes data to be copied twice, which is time consuming. Some systems do the I/O data transfer directly from the user space. Doing the I/O from the user space is safe if the operating system follows a local replacement scheme, because while a process is doing the I/O, it is blocked and its pages will not be recycled. If the operating system follows a global replacement scheme, it must lock the frames involved in the I/O before initiating it. Otherwise, while a process is doing the I/O on a frame, there is a possibility of the replacement algorithm allocating the same frame to another process that may corrupt the frame. The replacement algorithm must not recycle locked frames. The locks may also be used to pin down frames that must not be recycled.

≫ *In multilevel paging (see Fig. 8.25 on page 342), we can page out inner-level page tables without much harm. In such situation, part of the kernel is also in the swap space and may cause page faults! In some operating systems, there are other parts of the kernel that can also be paged out.*

9.6.7 Page-transfer Management

When a victim selection is complete, we have two tasks at hand depending on whether the victim is dirty. If the victim is not dirty, we just overwrite the victim with the new page image. If the victim is dirty, we first copy the victim image to the auxiliary memory. The copying procedure may cause considerable delay in completing the page fault service, and this needs to be avoided as far as possible.

The operating system normally keeps a pool of free frames to speed up page fault service (see Fig. 9.12). On occurrence of a page fault, the system allocates a free frame from the pool and schedules a page read from the auxiliary memory. At the same time, it does the victim selection, and if the victim is dirty, it schedules a frame write to the auxiliary memory. When the frame write is complete, the system puts the victim in the free frame pool. The process that caused the page fault can resume its execution when the page reading is complete even if the victim write is not complete. In addition, when a swap device becomes idle, the system may schedule (asynchronous) writing of dirty frames and reset their dirty bits. This will increase the probability of non-dirty victim selection when real needs arise.

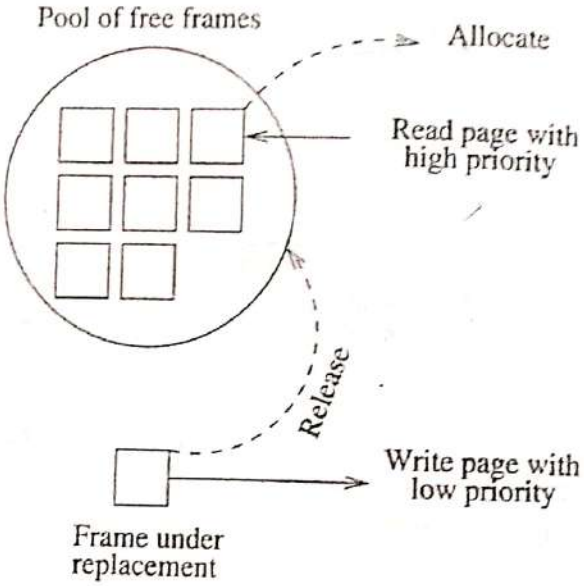

Figure 9.12: *Page-transfer activity for replacing dirty frames*

Some operating systems manage the pool of free frames intelligently. When a frame is put in the pool, the frame content is not destroyed. The pool remembers the process identifier and the page number for each frame. If the process faulted on the page again, the page frame can be obtained from the pool without causing a page read from the auxiliary memory. As no I/O operation is performed, the page fault service will be extremely fast.

9.6.8 Thrashing and Its Remedy

If a page-replacement algorithm is not good or processes are not allocated sufficient frames, the algorithm may select bad victims for replacements. A victim selection is "bad" in the sense that the process references the victim page almost immediately and the system needs to reload the page. Thus, the system may end up doing frequent page transfers between the main memory and the auxiliary memory. This

...may lead to a situation where the swap device becomes congested causing almost all processes to wait on the swap device and rendering the CPU almost idle. This situation (of high page-transfer activity) is called *thrashing*, when the CPU spends most of its time in juggling pages between the main memory and the swap device. A process is said to *thrash* if it spends more time on the swap device than in purposeful computation.

Thrashing causes severe performance degradation. When thrashing occurs, the system's responsiveness may collapse. On the other hand, the utilization of the CPU and non-swap I/O devices decreases. To enhance their utilization, the operating system (actually, a medium-term scheduler)» may allow creation of new computations (or resumption of suspended ones) in the system. These new computations will claim memory frames for themselves, which will aggravate the thrashing situation further. Thereby, thrashing is to be avoided at all costs.

» *The medium-term scheduler is different from the short- and long-term schedulers we discussed in Chapter 5.*

The only solution to the thrashing problem is that the replacement algorithm must not select victims needed immediately by the CPU. Thrashing of a process can only be reduced by keeping in the main memory those pages the process needs immediately for its execution without causing a page fault. To run without throwing itself and the system into thrashing, each process must be assigned frames to hold these pages at the least. If it is not possible to allocate those many frames, the operating system (i.e., the medium-term scheduler) should suspend the process execution and swap out the entire address space to the auxiliary memory, and thus reduce the degree of multiple processes (i.e., multitasking). At any juncture, if a process does not have the minimum number of frames mandated by the processor architecture, the process must be suspended and swapped out.

Experience and experimental studies seem to suggest that for a reasonable replacement algorithm, the fault probability of the algorithm is far more sensitive to the number of allocated frames than the algorithm. Therefore, although the choice of replacement algorithm is important, the choice of allocation size is critical. For most algorithms, the more it is available, the better is the performance. However, the constraint is the limited availability of frames.

Loosely, the minimal collection of pages a process needs to prevent its own thrashing is called its "working set". A good page-replacement scheme would ensure that all ready processes have their working sets in the main memory. The fundamental questions are: (1) how do we know the size of the working sets, and (2) which are the pages in the working sets? We answer these two questions next by defining precisely the working set concept. Before doing so, we introduce two concepts, namely reference string and locality in the next two sub-subsections.

» *The working set is an important concept in implementing virtual memory. Performance of virtual memory depends on how successfully we exploit this concept.*

9.6.8.1 Reference String

A process, in its lifetime, makes a sequence of references to its private address space. (This is irrespective of how many frames are allocated to the process.) From this sequence, we can derive its *reference string* of pages, say $p = p_1, p_2, p_3, \ldots$, where p_i are page numbers. The consecutive page numbers in the sequence need not be distinct. We can evaluate a page-replacement algorithm by feeding the reference string and the number of the allocated frames to the algorithm, and observing the number of page faults the algorithm generates from the given reference string.

9.6.8.2 Reference Locality

In reality, processes do not access entities in their address spaces at random. Each process favors a subset of its pages over an interval, and this set of favored pages changes membership gradually. This is a kind of localized access pattern. A locality here is a collection of pages that are used together by the process. There are two kinds of localities in a program execution, namely, spatial and temporal. These two are discussed in the next paragraph.

Locality in process execution is not unexpected as programs are more modular in recent times.[3] Each function/routine defines its own tiny locality of reference in object-based programming, all functions of an object collectively define locality of the object.[3] A program execution exhibits many different localities based on program structures. A program section consists of a set of functions that cause concentration of references in certain localities in the process address space. For example, the process references a set of variables together. Note that locality is a fuzzy concept, just like we use to mean "spatial neighbourhood" in a township. There is an orthogonal type of locality—temporal locality. If a process accesses a memory location (page) now, when does it again access the same memory location (page)?≡ For example, in a loop, a process executes the same instructions repeatedly. The locality principle is used to predict the immediate future references—which recently accessed instruction/data might be referenced again.

≫ *We may apply Pareto's 80/20 rule: (1) 80% of time is taken by 20% of code and (2) 80% of references go to 20% of address space.*

≫ *Spatial connectivity and temporal connectivity are two aspects of an executable program. Spatial connectivity preserves the address adjacency of consecutive instructions and temporal connectivity preserves the memory residency of consecutive instructions. One of the primary responsibilities of memory-management system is to preserve these connectivities as much as possible to make demand paging an effective scheme.*

9.6.8.3 Working Set

A process in its lifetime moves through different localities. It executes mostly consecutive instructions and/or cycles through a set of instructions stored in a few instruction pages. These instructions reference mostly local variables on the stack ranging a few data pages. If all these pages (containing instructions and data) are in the main memory, there is no noticeable degradation of memory reference time and no appreciable slowdown in the computation speed. This subset of pages is called the *working set* of the process. At any given time, the working set denotes the smallest set of pages that must be present in the main memory to assure some level of program execution speed.

We have already noted in a previous section that there is no advance notice from either the application programmer or the compiler regarding what pages ought to be in the main memory. The operating system needs to determine (or at best, guess) the future page reference patterns depending on the recent reference patterns. Therefore, from the operating system's point of view, the working set of a process is the set of "most recently" referenced pages. Then, for a given reference string of a process, a working set of the process at the k th reference contains all those pages that the process has referenced over the past w references, where w is a positive integer number, as shown in Fig. 9.13. Let $p = p_1, p_2, p_3, \ldots$, be the reference string for a process. For a given w, $w > 0$, if we look backward w page numbers from p_k in the reference string, we get the working set $WS(k, w)$ relative to w at position k. That is, $WS(k, w) = \{p_{k-w+1}, \ldots, p_k\}$, is the content of the window of size w looking backward at reference point k. Here, w is the size of a *working window*. The cardinality of the working set is always less than or equal to w.≡

≫ *The smaller the cardinality of the working set, the stronger the current locality.*

[3] Large objects and large number of member functions can though make reference chaotic. Callback interface (such as garbage collection) software development methodologies add to the chaos considerably.

The working set models the localized pages referenced by the process in the "recent" past; it "approximately" defines its current (spatial) locality in the immediate past. The precision of the locality depends on the size of the working window. The locality is assumed to be nearly the same for the immediate future references—hoping that a very large fraction of next w references from position p_k will be from $WS(k, w)$. If the operating system holds the pages belonging to the working set of the running process in the main memory, the page fault rate can be curtailed with the hope that the process remains in the current locality for some more time. The minimum number of frames required by each process to avoid thrashing is the cardinality of the working set. If we do not allocate this minimum number of frames to the process, it may thrash. A process should not be run if we cannot keep its working set in the main memory.

≫ *There is no hard and fast definition of working set; it is a fuzzy concept.*

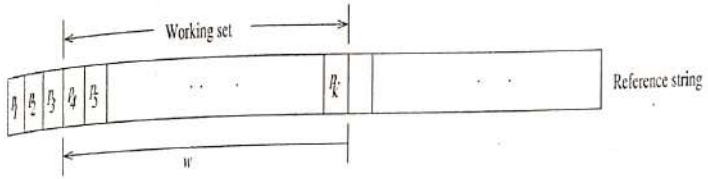

Figure 9.13: *Working set representation*

If a process does not exhibit good locality in its memory references, we have to store the entire address space or a large fraction of it in the main memory to avoid it thrashing. There is no alternative. Virtual memory performs most effectively when processes exhibit good locality, that is, they tend to concentrate their references in small regions of their address spaces. What is the best value of w, the length of working window? If w is too small, pages still useful may be removed from the main memory. This results in higher page-transfer activities. If w is very large, old pages and those not immediately required may remain in the main memory for long and unnecessarily occupying valuable memory space. This reduces effective memory utilization. The value of w then must be a compromise between the two. It has been shown by some evidences that the window size in real time is a value comparable to T, the average page-transfer time between the main memory and the auxiliary memory. That is, we need to hold all those pages that are referenced in the past T time units. However, this is not a hard-and-fast rule; it is a rule of thumb, and we remind you again that working set is a fuzzy concept.

≫ *Working set of a process can be approximated using a timer. At the beginning, the reference bit of all its pages is reset. When the timer interrupts, we know the working set of the process by examining the reference bits.*

If the summation of the cardinalities of working sets of all active processes is greater than the total number of memory frames the system possesses, there is a high probability that the system will soon start to thrash. Thrashing is avoided when loads on the CPU and the swap device are balanced. System throughput is at near optimal level when the virtual-memory system guarantees that each active process has just enough frames to hold its working set. When thrashing does occur, the system must reduce the number of active processes by transferring some of them entirely into the auxiliary memory. The processes that will be put into the auxiliary memory are determined by a medium-term scheduler that controls system workload.

9.7 Demand Segmentation

Demand paging requires a significant amount of support from the hardware even though it is considered the most efficient virtual-memory implementation. Some (old) processor architectures have only segmentation facility on which demand segmentation may be implemented. For example, OS/2 from IBM running on Intel 80286 processor implements demand segmentation because the 80286 processor does not have the paging feature.

A process may not need to have all its segments at the same time in the main memory for its execution. As in demand paging, we use a presence bit in the segment descriptor to indicate whether a segment is present in the main memory. The entire segment is either present or nothing at all. If a segment is in the main memory, the processor can directly reference elements from the segment block. Otherwise, the address translation hardware generates a "segment fault" exception and the process execution is suspended. The operating system brings the segment into the main memory from the auxiliary memory. (If needed, the operating system executes a segment replacement algorithm to make room for the requested segment.) The process resumes its execution from the faulted instruction after the segment fault service. To aid in segment replacement, the system may have a reference bit and a dirty bit in the segment descriptor.

9.8 The Real Illusion

The view of a very large "main memory" that application programmers see is essentially an illusion. The large memory is not real, and it is actually simulated by the memory-management software and processor hardware using a very large auxiliary memory. This is a fact, although applications do not know it or feel it. The memory-management system uses secondary storage devices such as disks to implement the auxiliary memory to hold computations in their entirety. Only a fraction of each computation is held in the main memory. As long as there is space in the auxiliary memory, new computations can be initiated.

The information that the CPU needs immediately must be in the main memory. The memory-management system propitiously moves information between the main memory and the auxiliary memory without the knowledge of applications and processes. Consequently, under demand paging or demand segmentation, unnecessarily large and ill-structured applications may run slower than expected. Such a slowdown is sometimes counterintuitive to naive programmers and theoretical computer scientists. We sometimes can design faster algorithms from using more memory by means of exploiting space-time trade-offs. Such algorithms, however, may not exhibit the expected speeding up under demand paging or demand segmentation.

When program executions exhibit good locality (i.e., they concentrate references in small regions of process address spaces), the virtual memory performs most efficiently. Working set is a good measure of locality. The smaller the working set, the higher is the execution speed because of fewer page faults. Otherwise, the system may envisage thrashing that may completely degrade performance. The best results are seen when working sets are small and slow changing. Unless applications are designed and implemented carefully, process execution speed may slow down considerably. Application designers have been warned!

Summary

This chapter shows
memory. This invol
such as disks. It is
main memory using
application program
They assume that t
application logic is
The virtual-mem
the main memory w
It implements the
agement. It automa
data) between the
applications or proc
This chapter di
and demerits. The
However, we have t
ence bit, reference
these bits is the p
memory or in the
been recently acce
modified since it
accessed, the addr
operating system
into the main me
ment algorithms
this chapter.

One demerit
transferring juggl
than doing purpo
performance. We
of working set. T
keep those pages

Although virt
not, however, ab
limited by the pr
is counterintuitiv
of some algorith
the speeding up

Literature

Paging and virt
the University
Kilburn et al. 19
profound influe
an independent
in the way we u
hardware was l

Summary

This chapter shows how we can execute large applications on limited-sized main memory. This involves simulating a large "virtual memory" using swap devices such as disks. It is actually a marvelous technique of managing the limited-sized main memory using large-sized swap device(s) as computation backup storage. The application programmers are not aware of the limitations of the main memory. They assume that the entire application process resides in the main memory and application logic is not constrained by the size of the physical main memory.

The virtual-memory manager actually implements a "computation cache" in the main memory where computations in their entirety are held on swap devices. It implements the fetch, placement, and replacement rules of typical cache management. It automatically and judiciously transfers information (such as code and data) between the main memory and swap devices on need basis without involving applications or processes.

This chapter discusses a demand paging-based virtual memory and its merits and demerits. The address translation is very similar to the pure paging system. However, we have to maintain additional information in the page descriptors: presence bit, reference bit, and dirty bit to mention a few. The most important among these bits is the presence bit, and it indicates whether the page is in the main memory or in the swap device. The reference bit indicates whether the page has been recently accessed or not. The dirty bit indicates whether the page has been modified since it has been brought into the main memory. If an absent page is accessed, the address translation hardware raises a "page fault" exception. The operating system services the exception and brings the page from the swap device into the main memory. This may require replacing some resident page. Replacement algorithms such as FIFO, second-chance, and LRU have been discussed in this chapter.

One demerit of virtual memory is thrashing, when processes spend more time transferring juggled pages between the main memory and the swap devices rather than doing purposeful computations. Thrashing degrades process as well as system performance. We show how thrashing can be reduced by implementing the concept of working set. The working set models the locality of processes. The system must keep those pages (from the working sets) in the main memory to reduce thrashing.

Although virtual memory helps in increasing the degree of multiprocesses, it is not, however, able to enlarge the process address space. The address space size is limited by the processor architecture. Virtual memory is an illusion and frequently is counterintuitive to application programmers. We may improve the performance of some algorithms by using more memory, but such algorithms may not exhibit the speeding up expected.

Literature

Paging and virtual memory were invented by the Atlas Computer designers at the University of Manchester in 1950s (Denning 1970, 1996; Fotheringham 1961; Kilburn et al. 1961, 1962). They implemented demand paging. Their idea has had a profound influence on memory-management schemes. They postulated "address" as an independent concept distinct from memory location and caused a paradigm shift in the way we understood address spaces at that time. Virtual memory supporting hardware was later implemented in many processor architectures including the IBM

360/80, CDC 7600, Burroughs B6500, GE 645, and Intel 80386 systems. An example of demand paging system is the MULTICS running on GE 645. Linux running on Intel 80386, implements a demand paging-based memory-management system.

Denning (1970) presented a useful survey on virtual memory. This chapter and the previous one are primarily based on his survey and his short note (Denning 1996). Demand segmentation was implemented in the OS/2 operating system of IBM running on Intel 80286 processor (Iacobucci 1988).

Naur (1965) first observed localized access patterns in structured programs. Denning (1968) has also discussed thrashing. He showed that the working set size in real time is a value comparable to the average page-transfer time between the main memory and swap devices. In the same paper, he postulated the working set model.

Belady (1966) studied the FIFO page-replacement algorithm and pointed out that some anomalous situations may occur that would cause more page faults when the number of frames available to a process increases. The reference string used in the Section 9.6.5.1 on page 371 to show that anomaly is taken from Belady et al. (1969). Mattson et al. (1970) pointed out that stack algorithms do not exhibit Belady anomaly. Stack algorithms have the following property: when an algorithm is run with more frames, it keeps at least those frames it would have kept if it were run with fewer frames. The LRU algorithm always keeps the most recently used pages, and hence is a stack algorithm. Thus, LRU-replacement schemes do not have the Belady anomaly defect.

Exercises

1. What is a virtual memory? How is it different from the main memory?
   Answer: A virtual memory is a large memory system that is simulated using a limited-sized main memory and a large swap device. A virtual memory is an illusion and does not exist in reality, but the main memory is real. ⊗

2. Describe some advantages and disadvantages of virtual memory over a non-virtual memory-based system.
   Answer: Advantages: application size is independent of the available quantum of the main memory; we can vary the amount of memory space in use by a given application process at runtime; effective utilization of the main memory is higher. Disadvantages: may slow down program execution speed, especially when the system starts thrashing. ⊗

3. Is there any relation between the size of a virtual memory and that of the main memory or process address space? Justify your answer.
   Answer: No, there is no such relationship. The virtual memory size depends solely on the space available on the swap or backing store. ⊗

4. Can a virtual-memory system enlarge the size of the process address space supported by the processor architecture? Justify your answer.
   Answer: The answer is no. The process address space size is determined by the maximum (logical) address the CPU can generate. It has nothing to do with the size of the main memory or virtual memory. ⊗

5. What is an overlay? What are its merits and demerits?
   Answer: Overlay is a technique of reusing a part of process address space to hold

different code/data at different times. Merit: we can execute large programs that may not fit entirely in the main memory. Demerit: increases code complexity. ⊗

6. What is an auxiliary memory? How is it different from the main memory?
Answer: Auxiliary memory is used to hold computations (aka, application processes) in their entirety. It acts as a backup of computations. The CPU cannot reference an auxiliary memory directly in the same way it can do so the main memory. ⊗

7. How are virtual memory, auxiliary memory, and main memory related to one another?
Answer: The virtual memory logically stores all processes and the operating system; they are physically stored in the auxiliary memory. Parts of the virtual memory are kept in the main memory temporarily on the need basis. ⊗

8. Explain the difference between logical, physical, and virtual addresses.
Answer: Logical address space is process private address space. Logical address spaces are embedded in the virtual address space. Physical address space is the main memory space that partly holds the virtual address space. ⊗

9. What is demand fetching? What is pure demand fetching?
Answer: Demand fetching is bringing a part of process address space into the main memory from the auxiliary memory when that part is referenced by the CPU. In pure demand fetching, a process is started without any part of its address space in the main memory. ⊗

10. What is prefetching? How is it different from demand fetching?
Answer: Prefetching is loading parts of a process address space in the main memory before the process requests these parts. In demand fetching, a process stalls for the missing parts. We can avoid or minimize such stalling in prefetching. ⊗

11. What is demand paging? In what platforms can it be implemented?
Answer: In demand paging, pages of a process are loaded in the main memory when needed. We can implement demand paging in those platforms where the processor architectures have paging scheme with additional support for presence bits. ⊗

12. In a demand paging system, can we always move all unused parts of process address space out of the main memory? Justify your answer.
Answer: Yes, we can. In fact, we can swap out the entire process address space. ⊗

13. What is a page fault? When does it occur? What does the operating system do when a page fault occurs? What effects does the page fault have on the running process?
Answer: When the CPU references a page that is not in the main memory, it produces a page fault exception. The operating system services the exception and loads the missing page in the main memory, and resumes the program execution from the faulting instruction. The process execution speed slows down considerably. ⊗

14. How does the processor hardware know that a page is missing from the main memory?
Answer: By examining the value of the presence bit in the page descriptor. ⊗

15. Explain FIFO page-replacement algorithm and Belady's anomaly.
Answer: **Will not be provided.** ⊗

16. Explain the second-chance page-replacement algorithm.
Answer: **Will not be provided.** ⊗

17. Explain the LRU page-replacement algorithm. Why is it difficult to imple-ment it in its pure form? Explain one approximation to the LRU-replacement algorithm.
Answer: **Will not be provided.** ⊗

18. In the context of page-replacement algorithms, what are stack algorithms? Which of the following are stack algorithms: (i) optimal replacement, (ii) second-chance replacement? Explain which of them can exhibit Belady's anomaly
Answer: An stack algorithm has the following property: when it is run with more pages, it at least keeps those pages that it would have kept if it were run with fewer pages. Optimal replacement is a stack algorithm, but second-chance is not. The latter can exhibit Belady's anomaly. Belady anomaly states that the paging algorithm occasionally perform poorly as the size of memory increases. ⊗

19. What is page-transfer management and why is it needed?
Answer: **Will not be provided.** ⊗

20. Consider the following page reference string of a process: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5, 2, 1. Suppose the process has four frames. The system follows local replacement policy. Draw the frame configurations for the reference string under (i) FIFO and (ii) Second-chance.
Answer: See Figure 9.14.

⊗

21. Consider the following page reference string of a process: 1, 2, 3, 4, 5, 6, 2, 1, 2, 3, 5, 6, 3, 2, 4, 2, 3, 6, 7, 4. If the process is allocated four physical frames, how many page faults would occur if page replacements are done using the (i) FIFO, (ii) LRU, (iii) modified LRU? (Assume that all reference bits are cleared after every five page references for item iii.)
Answer: **Will not be provided.** ⊗

22. Consider the following page reference string of a process: 1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 1, 2, 3, 4, 1, 2, 3, 1, 2, 1. If the process is allocated four frames, how many page faults would occur if page replacements are done using the (i) FIFO, (ii) LRU, and (iii) optimal algorithms?
Answer: **Will not be provided.** ⊗

23. What is the difference between global and local page-replacement strategies? State one advantage of each strategy over the other.
Answer: In a local page replacement scheme, only frames of a process are considered for replacement on page faults from the process; page faults of a process does not affect page-faulting behaviors of other processes. In a global page replacement scheme, any frame can be replaced by any process; memory is more effectively utilized. ⊗

24. What is thrashing? Why does it happen? Why is it bad for processes and the system? How can we limit it?
Answer: **Will not be provided.** ⊗

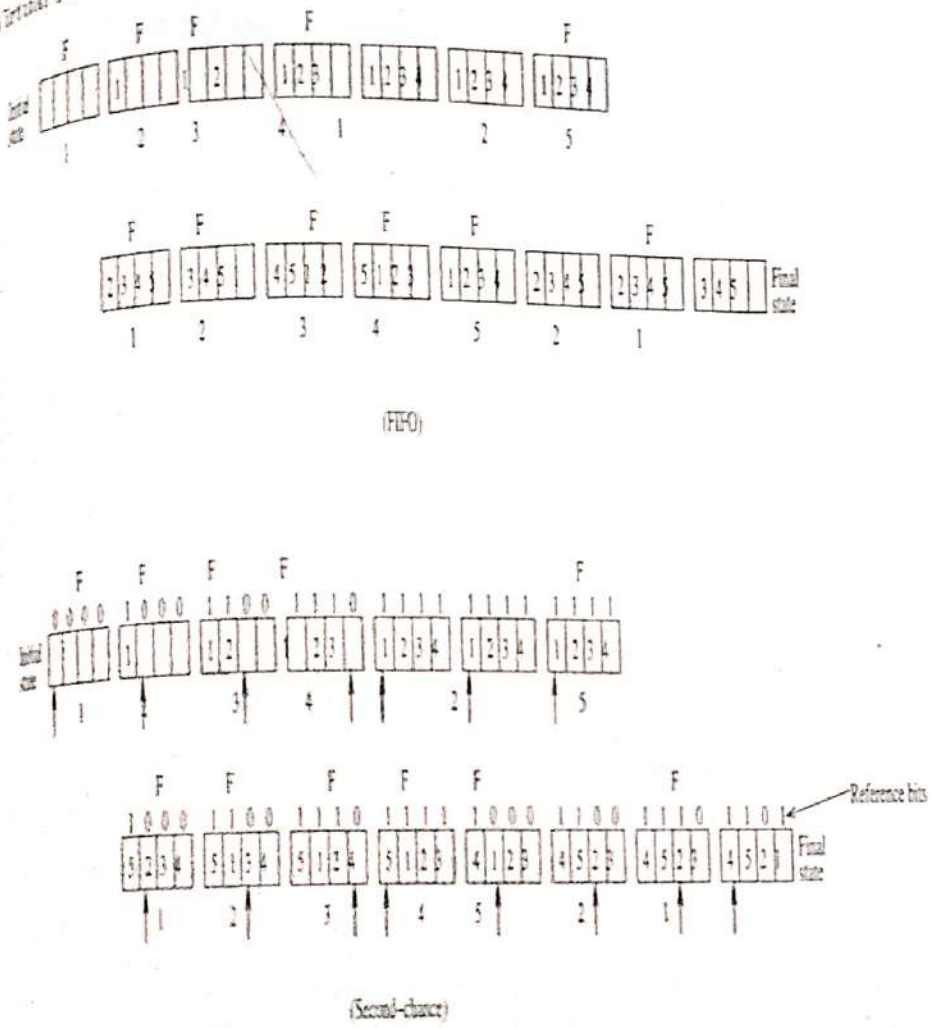

Figure 9.14: *Solution to Exercise 20*

25. When a thrashing situation is detected, what we do to eliminate it?
Answer: If only one or very processes are involved in thrashing, we can try to allocate more pages to the processes to reduce their level of thrashing. Otherwise, we would need to reduce the degree of multiprocessing by swapping out some processes. ⊗

26. What is locality of reference and explain its use. What is working set? What is it used for?
Answer: **Will not be provided.** ⊗

27. While discussing working set, we assumed that each process has only one working set. Suppose we have a system where each process has two working sets, one for instructions and another one for data. Do you think having two working sets is better than having just one? Justify your answer.
Answer: No. The two working sets can be encapsulated in a single bigger working set. ⊗

28. Suppose we have a computer system where normal memory access time is 50 ns on the average. The system's processor is without a TLB. How long does

a reference to a page require to materialize? Assume the page table and all pages are in main memory?
Answer: 50 + 50 = 100 ns. ⊗

29. Suppose we add a TLB in the processor of Exercise 28. The TLB has a hit of 75% of page references. The TLB takes 10 ns for search and another 10 ns on hit to access data. What is the average page reference time? Here too assume the page table and all pages are in the main memory?
Answer: 10 + (0.75(10) + 0.25(50)) + 50 = 10 + 20 + 50 = 80 ns. ⊗

30. Suppose we add a data/instruction cache in the processor of Exercise 29. The cache has a hit of 90% of memory references. The cache takes 10 ns for search and another 10 ns on hit to access data. What is the average page reference time? Here too assume the page table and all pages are in the main memory?
Answer: 10 + (0.75(10) + 0.25(50)) + (10 + 0.9(10) + 0.1(50)) = 10 + 20 + 24 = 54 ns. ⊗

31. Suppose the processor of Exercise 30 is used in a virtual-memory system. We can accommodate only 75% of all pages in the main memory. That is, about 25% of memory references go to disk due to page faults. Suppose page fault service time is 50ms. What is the average page reference time? Here too assume the page table is in the main memory.
Answer: **Will not be provided.** ⊗

32. Suppose we have a demand paging system in which the TLB can hold all the page descriptors of the running process. The normal memory access time is 50 ns on the average. The page fault service time is different in the two situations. It takes 25 ms to service a page fault if an empty frame is available or if the victim frame is not dirty. If the victim frame is dirty, the page fault service time is 50 ms. Assume that the victim frame is dirty 75% of the time. Find the maximum page fault rate for which the effective memory access time remains within 250 ns.
Answer: Let n be the total memory references and p is the page fault rate. Then np references will cause page faults. Out of the np page faults, we select dirty victims $0.75np$ times and clean victims $0.25np$ time. Thus, the total time for the n references is $50n$ nanoseconds plus $(0.25np * 25 + 0.75np * 50)$ milliseconds. This amount should be less than or equal to $250n$ nanoseconds. Thus $50n + (0.25np * 25 + 0.75np * 50) * 1000000 <= 250n$ Or, $1750000p <= 8$

Or, $p <= 8/1750000$. ⊗

33. Consider a demand paging system on a 32-bit machine that has a 20-bit address bus. The system has a page size of 4KB. A process is allocated the first and the last 64MB of its address space of size 2^{32}. How much memory does the system use to hold page tables if (a) the system has one-level paging and (2) two-level paging with 10 bit per level?
Answer: The page size is 4KB. We require 12 bits to span a page. Then we can use the remaining 20 (= 32 − 12) bits for page numbers. Total number of pages in the address space is 2^{20} = 1,048,576. The physical address is 20-bit, and hence we can have at most $2^{(20-12)}$ or 256 frames. We need 8 bits (or one byte) to identify a frame.

(a) The page table size is 1,048,576 bytes.

(b) The page directory has 2^{10} = 1024 entries and they will occupy 1024 bytes or a frame. Each 2nd level page table also has 2^{10} = 1024 entries and the table will