Chapter 8

Virtual Memory

Managing main memory, especially if it needs to be shared among multiple concurrent tasks or even different users, is a challenging problem. While the size of memory has been increasing steadily, and modern CPUs now use long address fields capable of generating very large address spaces, the amount of memory available and actually installed in most computers remains a limiting factor. This situation is not likely to change anytime soon as new applications, especially those using sound, graphics, and video, are capable of utilizing all the memory available. Virtual memory addresses the problems of sharing limited memory through architectural and OS software mechanisms that hide the presence of physical memory and present instead an abstract view of main memory to all applications. In this chapter, we describe the concept of virtual memory and the main implementation schemes.

8.1 Principles of Virtual Memory

The basic idea of virtual memory is to hide the details of the real physical memory available in a given system from the user. In particular, virtual memory conceals the fact that physical memory is not allocated to a program as a single contiguous region and also conceals the actual size of available physical memory. The underlying implementation creates the *illusion* that each user has one or more contiguous address spaces, each beginning at address zero. The sizes of such virtual address spaces may, for most practical purposes, be assumed unlimited. The illusion of such a large amount of memory is created by subdividing the virtual memory into smaller pieces, which can be loaded into physical memory whenever they are needed by the

Figure 8.1: Principles of virtual memory

execution.

The principle is illustrated in Figure 8.1, where different portions of two virtual spaces VM1 and VM2 are mapped onto physical memory. Two distinct portions of VM1 and the beginning portion of VM2 are currently present in main memory and thus accessible by the processor. The address mapping mechanisms, **address_map**, translate logical addresses, generated by the processor, to physical addresses, presented to the actual memory.

Several different forms of virtual memory exist, depending on the view presented to the user and the underlying implementation. From the user's perspective, there are two main organizations. The first presents virtual memory as a single contiguous linear space, corresponding to our conventional view of physical memory. In this organization, virtual memory is considered a large, linearly addressed sequence of n cells (words, bytes, or, less frequently, individual bits), which are referenced using addresses in the range from zero to $n-1$. As with physical memory, n is usually a power of 2, i.e., $n = 2^k$ for some integer k. We call this a **single segment** address space. In the implementation, the virtual memory is subdivided into **fixed-sized** portions, called **pages**, which can be loaded into noncontiguous portions of physical memory, called **page frames**.

A **multiple segment** virtual memory divides the virtual address space into a set of segments, where each segment is a contiguous linear space and can vary in size. A **segment** is a user-defined entity that can be treated as an independent logical unit, for example, a function or a data structure. In common implementations of this second organization, each segment can be loaded as a single unit into a contiguous portion of memory, or it can be subdivided further into fixed-size pages.

For both methods—a single segment or a multiple segment virtual memory—there are several important issues that must be addressed by an implementation:

Address Mapping Mechanisms. There are many different ways to define the conceptual *address_map* function that translates logical addresses to their physical equivalents. Since such a translation must be performed one or more time for ever instruction executed, it is crucial to performance that it is done with minimal overhead. Section 8.2 presents the principal schemes that are used.

Placement Strategies. To access any portion of the virtual memory space, it must reside in main memory. Thus the implementation must provide a placement or memory allocation strategy, which determines *where* to load the needed portions of the virtual memory. When the virtual memory is subdivided into variable-size segments, the memory manager can employ the allocation strategies developed for variable-size partitions, such as first-fit or best-fit, which were already discussed in Section 8.3. With fixed-size pages, placement become greatly simplified, because all memory holes (page frames) are of fixed size and thus a given page will fit into any free frame.

Replacement Strategies. When a new portion of a virtual memory needs to be loaded into main memory and there is not enough free space available, the system must create more space. In multiprogrammed systems, this can be done by swapping out one of the current processes. Alternately, the system could evict only some portion, i.e., a single page or a segment, of one of the currently resident virtual spaces. The particular page or segment removed from executable memory affects performance critically. We will consider various solutions to this problem in Section 8.3.

Load Control. A static loading policy transfers all of a process's virtual memory into main memory prior to execution. But this automatically restricts the number of processes that can be active simultaneously. It may also waste a lot of memory and I/O bandwidth when only a small portion of the virtual memory is actually used. A more flexible approach is to load different portions of virtual memory dynamically, i.e., at the time they are actually needed. This is called **demand paging** or **demand segmentation**, depending on the virtual memory organization. Its main disadvantage is that processes may run very inefficiently if they only have a relatively small amount of memory. Load control addresses the problem of *how much* of a given virtual space should be resident in memory at any given time. Section 8.3.3 discusses the various solutions and trade-offs.

Sharing. There are important reasons for allowing two or more processes to share the same portion (code or data) of main memory. Both paging and segmentation permit sharing. However, since segments correspond to logical entities into which the user chose to divide the program, they are a natural basis for sharing. In contrast, pages represent an arbitrary subdivision of the virtual space, which makes sharing more difficult. The next chapter explores the trade-offs in detail.

Figure 8.2: Form of virtual and physical address

8.2 Implementations of Virtual Memory

We start by describing the architecture of paging systems, present the basics of segmentation, and then discuss the common combination of the two—paging with segmentation. Paging of OS systems tables is treated next. The final topic of the section is the use of translation look-aside buffers for improved performance.

8.2.1 Paging

In a paging implementation, the virtual address space is divided into a sequence of equal-sized contiguous blocks called **pages**; for P pages, these are numbered consecutively p_0, p_1, ..., p_{P-1}. The most common page size is between 1K to 16K bytes. Similarly, the available physical memory is divided into a number of equal-sized contiguous blocks called **page frames**, numbered f_0, f_1, ..., f_{F-1}, where F is the total number of frames. The page size is identical to the page frame size.

A virtual address va is then interpreted as a pair (p, w), where p is a **page number** and w is a **word number**, i.e., an **offset** or **displacement**, within the page p. Let us denote the number of bits used to represent p as $|p|$, and the number of bits used to represent w as $|w|$. A virtual address va is then a string of $|p| + |w|$ bits, resulting in a virtual memory size of $2^{|p|+|w|}$ words. The first $|p|$ bits of va are interpreted as the page number p, while the remaining $|w|$ bits give the word number w. The top portion of Figure 8.2 shows the relationships between the numbers p, w, and their bit-lengths. As an example, a 32-bit address could be divided into $|p| = 22$ and $|w| = 10$, resulting in $2^{22} = 4,194,304$ pages of $2^{10} = 1024$ words each.

A physical memory address pa is similarly interpreted as a pair (f, w), where f is the page **frame number** and w is a word number within the frame f. When pa is a string of $|f|+|w|$ bits, resulting in a physical memory of $2^{|f|+|w|}$ words, the first $|f|$ bits are interpreted as the frame number f, while the remaining $|w|$ bits give the word number w. The bottom portion of Figure 8.2 shows the relationships between the numbers f, w, and their bit-lengths. For example, a 16-bit address could be divided into $|f| = 6$ and $|w| = 10$, resulting in 64 frames of 1024 words each. Note that $|w|$ is the same in both va and pa because the size of a page must be the same as the

Figure 8.3: Paged virtual memory

size of a page frame.

At runtime, any page can reside in any frame of the physical memory. The task of the *address_map* is to keep track of the relationship between pages and their current page frames, and to translate virtual memory addresses $va = (p, w)$ generated by the processor into the corresponding physical memory addresses $pa = (f, w)$, as illustrated in Figure 8.3. Since the w component is the same in both addresses, the translation amounts to finding the frame f holding the page p. Once f is known, the physical address pa is formed in one of the following two ways, depending on the underlying hardware support.

Assume that f and w are kept in separate registers. The physical address can then be computed by the formula $pa = f \times 2^{|w|} + w$, where $2^{|w|}$ is the page size. This computation can be done efficiently in hardware by shifting f to the left by $|w|$ positions and then adding w to it. The second method is even simpler: the two registers holding f and w are simply *concatenated* into one number, pa. For the remainder of this chapter, we will use $f + w$ to denote the addition or the concatenation of the two components f and w.

The remaining question now is how the *address_map* should keep track of the mapping between the page numbers p and the frame numbers f. There are two basic mechanisms to achieve that, one using frame tables and the other using page tables.

Frame Tables

The first approach to maintaining the correspondence between page numbers and frame numbers is to implement a table of length F, where each entry corresponds to a frame and contains the number of the page currently residing in that frame. With multiprogramming, the situation is more complicated because several resident processes can be using the *same* page number, each corresponding to a *different* frame; consequently, the same page number p, belonging to different processes, can be found in more than one entry of the table. Thus, in order to distinguish among the entries, we also need to record the process *id* as part of each entry.

Let us represent this **frame table** as an array $FT[F]$, where each entry $FT[f]$ is a two-component structure: $FT[f].pid$ records the process *id* whose page is stored in frame f, and $FT[f].page$ contains the number of the page

Figure 8.4: Address translation using frame table

stored in that frame. Assuming that each addressed page of the current process is in main memory, the function of the *address_map* can be written conceptually as follows:

```
address_map(id, p, w) {
   pa = UNDEFINED;
   for (f = 0; f < F; f++)
      if (FT[f].pid == id && FT[f].page == p) pa = (f+w);
   return pa;
}
```

A sequential search through the table FT would be too inefficient since the mapping must be performed on every memory reference. One way to make this approach practical is to store the table in an **associative memory**, where the cells are referenced by their *content* rather than their address. A familiar example will clarify the idea. To find a phone number for an individual in a phone directory, it is necessary to search the directory for a *(last name, first name)* match; this task is not difficult *only* because the entries have been sorted in alphabetical order by name. If, however, one starts with a phone number and wishes to find the name or address of the party with that number, the task is hopelessly time-consuming. Storing the telephone book in a general associative memory would permit access of any entry using either a name or an address or a phone number as the search key; that is, any field in an entry can be used and the search occurs by content.

Hardware implementations of associative memories are not quite this general and normally only provide one search field for each entry. But this is sufficient to implement the frame table FT. Each entry of FT can hold the tuple (id, p), where p is a page number belonging to a process identified by id. When a process generates a virtual address (p, w), the concatenation of its id and the page number p is presented to the associative memory, which searches the frame table for a match. The search is performed entirely in hardware by examining all entries $FT[f]$ in parallel. If a match is found, the index f of the matching entry concatenated with the word offset w, which yields the physical memory address $pa = (f, w)$. The entire translation process is illustrated graphically in Figure 8.4.

Case Study:

> Paging can be credited to the designers of the ATLAS computer, who employed an associative memory for the address mapping [Kilburn, et al., 1962]. For the ATLAS computer, $|w| = 9$ (resulting in 512 words per page), $|p| = 11$ (resulting in 2024 pages), and $f = 5$ (resulting in 32 page frames). Thus a 2^{20}-word virtual memory was provided for a 2^{14}-word machine. But the original ATLAS operating system employed paging solely as a means of implementing a large virtual memory; multiprogramming of user processes was not attempted initially, and thus no process id's had to be recorded in the associative memory. The search for a match was performed only on the page number p.

□

In later systems, the frame table organization has gradually been displaced by a page table structure. As will be described in more detail in the next section, a page table keeps track of all frames that hold pages belonging to a given process. Thus a separate page table is needed for each process. But, recently, the virtues of frame tables have been rediscovered in a number of modern systems, including certain IBM and Hewlett-Packard workstations. The main attraction is that only a single frame table needs to be maintained for all processes. Such a frame table is commonly referred to as an **inverted page table**, since it stores the same information as page tables but is sorted by frame number, rather than process id and page number.

The main problem with implementing frame tables is that, due to the increasingly larger memory sizes, frame tables tend to be quite large and cannot be kept in associative memories in their entirety, as was the case with the original ATLAS design. The solution is to use a fast implementation in software, which can be achieved using hash tables. This still requires at least one extra memory access. To minimize this overhead, associative memories are used as **translation look-aside buffers** to bypass the access to the tables most of the time.

Another problem is sharing of pages in main memory. To allow two or more processes to use the same copy of a page in main memory, the frame table would have to be extended to keep track of multiple processes for each frame entry. This further complicates the design of the frame table and increases its cost.

Figure 8.5: Address translation using page tables

Page Tables

A **page table** keeps track of the current locations of all pages belonging to a given process. The p'th entry in the page table identifies the page frame containing that page. In most systems, a special page table register, PTR, contains the starting address of the page table of the currently running process. The translation performed by the function *address_map* is then:

```
address_map(p, w) {
   pa = *(PTR+p)+w;
   return pa;
}
```

$PTR + p$ points to the page table entry for page p; the content $*(PTR + p)$ of this entry is the frame number f. Adding the offset w to f then yields the physical address pa. Figure 8.5 presents this mapping in graphical form.

Example:

Suppose that the page table for the current process is located starting at address 1024 and has the contents as shown in Figure 8.6.

...	...
1024	21504
1025	40960
1026	3072
1027	15360
...	...

Figure 8.6: Page Table Contents

Assuming a page size of 1024 words, the virtual address $(2, 100)$ maps dynamically into $pa = *(1024 + 2) + 100 = *(1026) + 100 = 3072 + 100 = 3172$. □

Note that *two* memory accesses must be made in order to read or write any memory word during a virtual memory operation—one access to the

page table and the second to the actual page. To avoid the first access most of the time, it is common to use translation look-aside buffers to hold the most recently used parts of the page tables in associative memory.

Demand Paging

Paging greatly simplifies the problem of placement, since any page can be loaded into any free page frame. But paging can also be used to solve the problem of limited memory size so that programs larger than the available main memory could be executed without requiring the programmer to specify an overlay structure. This is accomplished by implementing dynamic memory allocation mechanisms, where pages are loaded at the time they are actually needed rather than statically before program execution. This approach is called **demand paging**.

It requires a mechanism that signals a "missing page" when reference is made to a page currently not resident in main memory. Such a missing page event is called a **page fault**. It invokes the operating system; in a lengthy procedure, the OS finds the desired page, loads it into a selected frame, and restarts the process that caused the page fault.

The following extension of the *address_map* function illustrates the principles of demand paging. The function *resident(m)* returns the value *true* if m designates a currently resident page and the value $false$ if the page is missing from memory.

```
address_map(p, w) {
   if ( (resident(*(PTR+p))) ) {
      pa = *(PTR+p)+w;
      return pa; }
   else page_fault;
}
```

If no free frame is available at the time of the page fault, the operating system can either block the faulting process, or it may create a free frame. The latter is accomplished by evicting one of the currently resident pages to secondary memory, or by evicting an entire process, i.e., all its resident pages, to secondary memory. Deciding which page or which process to remove is governed by the load control and page replacement schemes employed.

The main drawback of pure paging is that the user is restricted to a single contiguous address space into which all program and data must be placed in

some sequence. This single space presents some difficulty with dynamic data structures that may grow and shrink at run time. Since they all share the same address space, the user (or the compiler) must decide where to place them so that they have sufficient room to expand without running into each other.

Case Study:

> Unix assigns a single contiguous address space to each process. A Unix process consists of three main parts (segments): (1) the **code** segment, (2) the **data** segment, and (3) the **stack** segment. All three must be placed into the single shared address space. Fortunately, only the last two segments can vary in size—the code segment remains constant. The data segment, which contains the process' variables, can grow and shrink as the program allocates and frees data in heap memory. That is, the area reserved for the heap may be increased or decreased explicitly (using a system call *brk*). The stack segment grows and shrinks as a result of function invocations and returns. To allow both segments to grow independently, Unix places them at opposite ends of the shared address space. That is, one end of the memory contains the (fixed-size) code segment, followed by the data segment. The other and contains the stack. Thus both may expand toward the middle.

□

8.2.2 Segmentation

There are many instances where a process consists of more than the three basic segments: code, data and stack. For example, a process with multiple threads needs a separate stack for each thread, where each stack grows and shrinks independently. Similarly, files may be mapped temporarily into the address space of a process to simplify accessing their contents. (This will be discussed in Chapter 10.) Such files may also grow and shrink as the process executes. Placing multiple dynamically changing entities into a single address space is a difficult problem.

Segmentation solves the problem in an elegant way. It implements virtual memory as a collection of address spaces, each of which is termed a **segment** and may be a different size. This organization permits the system to mirror the organization of a given application by using a separate segment for each logical component, such as a function, a module comprising multiple

functions, an array, a table, or a stack. Each such component has a name by which it is identified by the programmer.

These logical names are translated by the linker or loader into numbers, called **segment numbers**, each of which uniquely identifies one of the segments. An offset within each segment then identifies a specific word. Thus, similar to a page implementation of virtual memory, a virtual address va using segmentation also consists of two components, (s, w), where s is the segment number and w is the offset. The main difference is that segment numbers correspond to logical components of the program while page numbers bear no relationship with the program structure.

There are two different schemes for building a segmented virtual memory. One treats the segment as the basic unit for memory allocation and assumes that memory can be dynamically allocated and relocated in variable size blocks. The second employs paging as a means of allocating space for a segment. That is, each segment is itself subdivided into fixed-size pages to facilitate allocation.

Contiguous Allocation Per Segment

Segmentation was pioneered by the Burroughs Corporation in the 1960's, where it was employed in the B5500 and B6500 computers [Burroughs, 1964, 1967]. These were stack machines oriented toward the efficient compilation and execution of block-structured languages.

Like a page table, a **segment table** is used for each active process to keep track of its current segments. Typically, segments are numbered sequentially with the ith entry in the segment table ($i = 0, 1, ...$) corresponding to segment number i. Each entry in the table contains the starting location of the segment in physical memory. It also contains protection information, to be discussed later in Chapter 13. The segment table itself is treated as a segment by the system. The starting point of the segment table of the currently running process is usually maintained in a dedicated register, the **segment table register**, STR. The STR is analogous to the PTR described in the last section. A virtual address (s, w) is then mapped to a physical memory address pa as follows:

```
address_map(s, w) {
   if ( resident(*(STR+s)) ) {
      pa = *(STR+s)+w;
```

```
            return pa; }
        else segment_fault;
    }
```

The function *resident* serves the same purpose as with demand paging; if the referenced segment is currently not resident in memory, a **segment fault** invokes the operating system to take the necessary actions.

The main advantage of segmentation over paging is that it divides programs into variable-size components that correspond to their logical components, providing, for example, natural protection, debugging, and portability boundaries. Its drawbacks are a complex placement and, in the case of demand segmentation, a complex replacement scheme. Segmentation also suffers from external fragmentation, similar to the simple memory scheme discussed in Section 7.3, resulting in wasted memory space. In recent years, systems using pure paging or paging combined with segmentation have become much more common than those using pure segmentation. As an example, the Intel 286 processor had pure segmentation but later models, such as the 386 or Pentium processor, support a combination of segmentation and paging.

8.2.3 Paging With Segmentation

To provide a multi-segment address space for each process and, at the same time, permit a simple placement algorithm, the principles of paging and segmentation have been combined into one memory management scheme. Under this scheme, memory is organized in variable-size segments, and, from the user's perspective, there is not much difference between pure segmentation and segmentation with paging. From the point of view of the operating system, however, segments are not contiguous portions of memory. Rather, each segment is subdivided into pages of fixed-size.

The implementation of segmentation with paging requires the use of both segment tables and page tables, and results in two levels of indirection in the logical-to-physical address translation. As with pure segmentation, a segment table register STR points to a segment table for the current process. Each of the segment table entries points to the page table for the corresponding segment. Each page table then keeps track of the pages belonging to that segment.

As a consequence of the two levels of indirection, a virtual memory address va is interpreted as a triple (s, p, w), where s is the segment number, p

Figure 8.7: Address translation using segment and page tables

is the page number within the segment, and w is the offset within the page. That is, p and w together specify a word within the segment s. The resulting *address_map*, illustrated graphically in Figure 8.7, performs the following function.

```
address_map(s, p, w) {
   pa = *(*(STR+s)+p)+w;
   return pa;
}
```

This is the simplest form of the address map: it assumes that the segment table, the page tables, and the pages are all resident in memory. If this requirement is relaxed (as is frequently done to permit more efficient utilization of main memory), each reference may potentially produce a page fault. Hence, the *address_map* must be extended to ensure that a page is resident before the access is made.

With the above scheme, each segment table and page table is considered to be a page. Demand paging is attractive under this organization; it permits an efficient use of memory and allows dynamic changes in segment size. The disadvantages of paging with segmentation are the extra memory required for the tables, the overhead of administrating memory in such a complex system, and the inefficiency of two additional memory references at each mapping. Again, fast associative registers are used to avoid extra references (Section 8.2.5).

8.2.4 Paging of System Tables

The sizes of the segment and page tables of a virtual memory system are determined by the lengths of the s and p address components, respectively. Depending on the number of bits uses to represent a virtual address, these tables may become potentially very large. To illustrate the problem, consider a 48-bit address and assume that the page size is 4K, i.e., requiring 12 bits for the offset w. That leaves $48 - 12 = 36$ bits to be split between s and p. One philosophy, assumed by many older systems, is to make s large while keeping p relatively small. For example, keeping p to the same size as w would leave $|s| = 36 - 12 = 24$, resulting in a very large segment table size

Figure 8.8: Address translation with paged segment table

of 2^{24} or 16 MB. Alternatively, we could keep s small while allocating the remaining address bits to p. This would yield a small number of very large segments—a philosophy that has been adopted in many recent systems due to the need to support multimedia applications, which require potentially very large objects. For example, assigning $|s| = 12$ would result in 4K segments per process, each segment consisting of up to 2^{24} or 16 MB of individual pages of 1KB each.

In both of the above scenarios, the resulting tables comprising megabytes of data are too large to be kept permanently resident in main memory. The solution is to subdivide each table into pages and load only those currently needed by the execution. Let us illustrate how this may be accomplished in the case of a large segment table. The s component of the virtual address is divided into two parts, say $s1$ and $s2$. The segment table is then broken into pages of size $2^{|s2|}$, where $|s2|$ is the number of bits comprising $s2$.

To keep track of the pages comprising the segment table, we need a new page table, as shown in Figure 8.8. To avoid confusion between the new page table of the segment table and the actual page tables, we will refer to the former as the **segment directory**. The $s1$ component is used to index this segment directory which, at run time, is pointed to by the register STR. The individual pages comprising the segment table are indexed by $s2$, while p and w are used as before. Assuming that all tables are in memory, the address map for a virtual address $(s1, s2, p, w)$ can be described as:

```
address_map(s1, s2, p, w) {
   pa = *(*(*(STR+s1)+s2)+p)+w;
   return pa;
}
```

Case Studies:

1. *MULTICS.* The s component in the MULTICS operating system is 18 bits long, resulting in a segment table of 2^{18} or 262,144 entries (segments). This table is divided into pages of 1024 words, resulting in $|s2| = 10$ and $|s1| = |s| - |s2| = 8$. Hence the segment directory (i.e., page table of the segment table) has 256 entries.

Figure 8.9: Address translation in Pentium processor: (a) segmentation only, (b) segmentation with paging

2. *Pentium.* The Intel Pentium implements an elegant memory management scheme that can support either pure segmentation or segmentation with paging. The virtual address (called the logical address in the Pentium literature) is 48 bits long, where 2 bits are used for protection, 14 bits are used for the segment number and the remaining 32 bits are the offset within the segment. The segment table is divided into two parts of 4K entries each. One is called the *local descriptor table* (LDT) and contains entries for segments private to the process; the other is called the *global descriptor table* (GDT) and holds descriptors for segments shared by all processes, notably, segments comprising the operating system. Each LDT or GDT entry contains the starting address of the segment, its length, and several other bits that facilitate the address translation and protection.

   Figure 8.9 (a) illustrates the address translation mechanism when pure segmentation is used. The segment number s first selects one of the segments from the segment table (i.e., the LDT or GDT), and a pointer to that segment entry is loaded into one of six special-purpose *segment registers*. Note that this is slightly different from the scheme discussed earlier (e.g. Figure 8.7), where the starting address of the segment table itself is kept in a register (STR) and the segment number s is added to it during each address translation. The starting address (32 bits in length) of the selected segment is fetched from the segment table entry and is added to the 32-bit offset from the virtual address. The resulting address is used to access the word in the segment.

   Figure 8.9(b) illustrates the address translation mechanism when paging is enabled. In this case, the segment descriptor is fetched from the segment table (LDT or GDT) using s as before, and the offset from the virtual address is again added to the segment base address. The resulting 32-bit quantity is however not interpreted as a physical memory address. Instead, it is divided into a 20-bit page number p and a 12-bit offset w within a page. This results in a 4K page size ($2^{12} = 4K$) and a page table size of over 1 million entries (2^{20}). Since this too large to keep in memory, the page table is subdivided further into 4KB pages. That is, p is split into two 10-bit components, $p1$ and

Figure 8.10: A translation look-aside buffer

$p2$; the physical address is then derived as shown in the Figure 8.9(b).

□

8.2.5 Translation Look-aside Buffers

Virtual memory architectures offer a number of advantages over systems that make physical memory visible to programs. However, one main drawback is the increased number of physical memory operations necessary for each virtual address reference. In the simplest case of pure paging or pure segmentation, one additional memory read is necessary to access the page or segment table. If both approaches are combined, two additional memory reads are needed—one to the segment table and a second to the page table. Finally, when segment or page tables are themselves paged, the number of additional references increases to three. In all cases, each reference also possibly generates a page fault.

To alleviate this problem, special high-speed memories, usually called **Translation Look-aside Buffers (TLB)**, are often provided to aid the address translation. The basic idea is to keep the most recent translations of virtual to physical addresses readily available for possible future use. An associative memory is employed as a buffer for this purpose. When a given virtual page is accessed several times within a short time interval, the address translation is performed only during the first reference; subsequent accesses bypass most of the translation mechanisms by fetching the appropriate frame number directly from the associative buffer.

Figure 8.10 shows the organization of such a translation look-aside buffer for a system with both segmentation and paging. The first column contains entries of the form (s, p), where s is a segment number and p is a page number. When a virtual address (s, p, w) is presented to the memory system, the buffer is searched associatively (in parallel) for the occurrence of the pair (s, p). If found, the number f in the second column of the buffer gives the frame number in which the corresponding page currently resides. By adding the offset w to f, the addressed word is found. Note that only one access to main memory is necessary in this case—the segment and page table are bypassed by searching the buffer.

Only when the search for (s, p) fails, need the complete address translation be performed. The resulting frame number is then entered, together

with the pair (s, p), into the buffer for possible future references to the same page. This normally requires the replacement of a current buffer entry. The most common scheme is to replace the *least recently used* such entry, as determined by a built-in hardware mechanism. (More detail on this scheme is given in the next section.)

Note that a translation look-aside buffer is different from a data or program *cache*, which may be used in conjunction with the translation look-aside buffer. The difference is that the buffer only keeps track of recently translated *page numbers*, while a cache keeps copies of the actual recently accessed *instructions* or *data values*.

8.3 Memory Allocation in Paged Systems

One of the main advantage of paging is that it greatly simplifies placement algorithms, since, due to their uniform size, any page can be stored in any free frame in memory. Thus an unordered list of free frames is sufficient to keep track of available space. This is true of both *statically* and *dynamically* allocated memory systems. In the former, all pages belonging to a process must be in memory before it can execute. When not enough free page frames are available, the process must wait; alternatively, we can swap out one or more of the currently resident processes in order to make enough space.

When memory is allocated dynamically, the situation is more complicated. The set of pages resident in memory is constantly changing as processes continue executing. Each process causes new pages to be loaded as a result of page faults. When no free page frames are available, the system can choose to suspend the faulting process or to swap out some other resident process, as in the case of static allocation. But with dynamic allocation it has a third option. It can choose to evict a single page (belonging either to the faulting process or to some other unrelated process) in order to create space for the new page to be loaded.

Deciding which page to evict is determined by the **replacement** policy. An algorithm for page replacement falls into one of two general classes, depending on the choices for eviction candidates:

1. A **global** page replacement algorithm assumes a *fixed number of page frames* shared by all processes. If free frames are needed, the algorithm considers all currently resident pages as possible candidates for eviction, regardless of their owners.

2. A **local** page replacement algorithm maintains a separate set of pages, called the *working set*, for each process. The size of the working set varies over time and thus each process requires a *variable number of page frames*. A local page replacement algorithm automatically evicts those pages from memory that are no longer in a process' current working set.

The next two sections present the most common page replacement algorithms from each category. To evaluate the performance of the various algorithms, we introduce a model that captures the behavior of a paging system independently of the underlying hardware or software implementation. The model represents the execution time trace of a particular program in the form of a **reference string (RS)**:

$$r_0 r_1 ... r_t ... r_T$$

where r_t is the number of the *page* referenced by the program at its tth memory access. The subscripts in the reference string can be treated as time instants. Thus the string represents $T+1$ consecutive memory references by the program.

Assume that the program represented by RS is allocated a main memory area of m page frames, where $1 \leq m \leq n$ and n is the number of *distinct* pages in RS. An allocation policy P defines for each instant of time t the set of pages IN_t that are loaded into memory at time t and the set of pages OUT_t that are removed or *replaced* at time t during the processing of RS.

There are two important measures of goodness for a given replacement policy. The first is the *number of page faults* generated for a given reference string RS. The main reason for adopting this measure is that page faults are very costly and must be kept to a minimum. Each page fault involves a context switch to the operating system, which must analyze the nature of the interrupt, find the missing page, check the validity of the request, find a free frame (possibly by evicting a resident page), and supervise the transfer of the page to a free frame. The page transfer from the disk to memory is the most costly of all these tasks, requiring milliseconds of disk access time. When we compare this to the access time to main memory, which is in the nanoseconds range, we see that the page fault rate must be kept to less than one in several millions of memory accesses to prevent any noticeable degradation in performance due to paging.

The second measure of goodness for a page replacement policy is the *total number of pages*, $IN_{TOT}(RS)$, loaded into memory when RS is processed.

$$IN_{TOT}(RS) = \sum_{k=0}^{T} |IN_t|$$

Note that $IN_{TOT}(RS)$ is not necessarily equal to the number of page faults; it depends on the the size of IN_t, i.e., the number of pages loaded at each page fault. Thus choosing an appropriate value for IN_t is an important consideration in devising efficient replacement algorithms. Loading several pages during one operation is more cost-effective than loading them one at a time. That's because the cost of I/O follows the formula $c_1 + c_2 * k$, where c_1 and c_2 are constants, and k is the amount of data being transfered. c_1 represents the fixed start-up overhead of the I/O operation and $c_2 k$ is the cost directly proportional to the number of bytes transferred. For example, the cost of loading 3 pages at a time is $c_1 + 3 * c_2$, while the cost of loading the same three pages one at a time is $3 * (c_1 + c_2) = 3 * c_1 + 3 * c_2$.

The main problem with loading multiple pages at each page fault is deciding which pages to load, i.e., which pages the program is likely to reference next. If the guess is incorrect, the pages will have been brought into memory in vain and put unnecessary overhead on the memory system.

Case Study:

> Windows 2000 employs the following **clustered paging** policy. When a page fault occurs, it loads not only the page being requested but also several pages immediately *following* the requested page. In the case of a code page, 3-8 additional pages are loaded, depending on the size of physical memory. In the case of data pages, 2-4 additional pages are loaded. The expectation is that execution will exhibit some degree of locality, and thus the pre-loaded pages will be accessed without causing additional page faults.

□

Because of the difficulty to predict the program's future behavior, most paging system implement a *pure demand paging scheme*, where only a *single page*—the one causing the page fault—is brought into memory at each page fault.

In the remainder of this chapter, we will be concerned with only pure demand paging algorithms. Since only a single page is loaded during each

page fault, $|IN_t| = 1$ if a page fault occurs at time t, and $|IN_t| = 0$ if no page fault occurs. Consequently, the total number of pages, $IN_{TOT}(RS)$, is equal to the number of page faults for a given reference string:

$$IN_{TOT}(RS) = \sum_{k=0}^{T} |IN_t| = \textit{number of page faults}$$

We will use this measure in comparing the relative performance of the different page replacement algorithms.

8.3.1 Global Page Replacement Algorithms

During normal operation of a multiprogramming system, pages belonging to different processes will be dispersed throughout the frames of main memory. When a page fault occurs and there is no free frame, the operating system must select one of the resident pages for replacement. This section considered page replacement algorithms where all currently resident pages are considered as possible candidates for replacement, regardless of the processes they belong to.

Optimal Replacement Algorithm (MIN)

For later comparisons and to provide insight into practical methods, we first describe a theoretically **optimal** but unrealizable replacement scheme. It has been proven that the following replacement strategy is optimal in that it generates the smallest number of page faults for any reference string RS [Belady, 1966; Mattson, et at., 1970; Aho, et al., 1971]:

> *Select for replacement that page which will not be referenced for the longest time in the future.*

More formally, this strategy can be expressed as follows. If a page fault requiring a replacement occurs at time t, select a page r such that

$$r \neq r_i, \qquad \text{for } t < i \leq T \tag{8.1}$$

or if such an r does not exist, select

$$\begin{aligned} r &= r_k, \quad \text{such that } t < k \leq T,\ k - t \text{ is a maximum, and} \\ r &\neq r_{k'}, \quad \text{where } t < k' \leq k \end{aligned} \tag{8.2}$$

Expression 8.1 chooses an r that will not be be referenced again in the future. If no such r exists, then the subsequent expression (8.2) chooses the r that is farthest away from t.

Example:

Consider the reference string $RS = cadbebabcd$ and assume the number of page frames to be $m = 4$. Assume further that at time 0, memory contains the pages $\{a, b, c, d\}$. With the optimal replacement algorithm, the memory allocation changes as illustrated in the following table:

Time t	0	1	2	3	4	5	6	7	8	9	10
RS		c	a	d	b	e	b	a	b	c	d
Frame 0	a	a	a	a	a	a	a	a	a	a	d
Frame 1	b	b	b	b	b	b	b	b	b	b	b
Frame 2	c	c	c	c	c	c	c	c	c	c	c
Frame 3	d	d	d	d	d	e	e	e	e	e	e
IN_t						e					d
OUT_t						d					a

Two page faults occur while processing RS. The first, at time 5, causes the replacement of page d, since d will not be referenced for the longest time in the future. For the second page fault, at time 10, any of the pages a, b, or c can be chosen, since none of them will be referenced again as part of RS; we arbitrarily chose page a. □

Random Replacement and the Principle of Locality

Program reference strings are virtually never known in advance. Thus practical page replacement algorithms must be devised which can make the necessary decisions without *a priori* knowledge of the reference behavior. The simplest scheme one might consider is a **random** selection strategy where the page to be replaced is selected using a random number generator. This strategy would be useful if no assumption about the program's behavior could be made.

Fortunately, most programs display a pattern of behavior called the **Principle of Locality**. According to this principle, a program that references a location at some point in time is likely to reference the same location and locations in its immediate vicinity in the near future. This statement is quite intuitive if we consider the typical execution patterns of a program:

1. Except for branch instructions, which in general constitute an average of only some 10% of all instructions, program execution is sequential. This implies that, most of the time, the next instruction to be fetched will be the one immediately following the current instruction.

2. Most iterative constructs (i.e., for-loops and while-loops) consist of a relatively small number of instructions repeated many times. Hence computation is confined to a small section of code for the duration of each iteration.

3. A considerable amount of computation is spent on processing large data structures such as arrays or files of records. A significant portion of this computation requires sequential processing; thus consecutive instructions will tend to reference neighboring elements of the data structure. (Note, however, that the way data structures are stored in memory plays an important role in locality behavior. For example, a large two-dimensional array stored by columns but processed by rows might reference a different page on each access.)

Since moving pages between main memory and secondary memory is costly, it is obvious that a random page replacement scheme will not yield as effective an algorithm as one that takes the principle of locality into consideration.

First-In/First-Out Replacement Algorithm (FIFO)

A FIFO strategy always selects the page for replacement that has been resident in memory for the longest time. In terms of implementation, the algorithm views the page frames as slots of a FIFO queue. A new page is appended to the end of the queue, and, at a page fault, the page at the head of the queue is selected for replacement.

More precisely, assume a physical memory size of m page frames. Let the identity of all resident pages be stored in a list, $P[0]$, $P[1]$, ..., $P[m-1]$, such that $P[i+1 \ \% \ m]$ has been loaded after $P[i]$ for all $0 \leq i < m$. A pointer k is also maintained to index the most recently loaded page, $P[k]$. On a page fault, page $P[k+1 \ \% \ m]$ is replaced and k is incremented by 1 (modulo m).

Example:

For the reference string RS = *cadbebabcd* and physical memory size $m = 4$, we obtain the following changes in memory occupancy:

Time t	0	1	2	3	4	5	6	7	8	9	10
RS		c	a	d	b	e	b	a	b	c	d
Frame 0	→a	→a	→a	→a	→a	e	e	e	e	→e	d
Frame 1	b	b	b	b	b	→b	→b	a	a	a	→a
Frame 2	c	c	c	c	c	c	c	→c	b	b	b
Frame 3	d	d	d	d	d	d	d	d	→d	c	c
IN_t						e		a	b	c	d
OUT_t						a		b	c	d	e

We have assumed that the pages resident in memory at time 0 had been loaded in the order a, b, c, d; i.e., a is the oldest and d is the youngest resident. This causes a to be replaced first (at time 5) when the first page fault occurs. The pointer, k, represented by the small arrow, is advanced to point to b. This page is replaced at the next page fault at time 5, and so on. The algorithm results in a total of 5 page faults for the given string RS. □

The main attraction of the FIFO replacement strategy is its simple and efficient implementation: a list of m elements and a pointer, incremented each time a page fault occurs, is all that is required. The list does not even have to be implemented explicitly; the pointer can simply cycle through the physical page frames in an ascending or descending address order.

Unfortunately, the FIFO strategy assumes that pages residing the *longest* in memory are the least likely to be referenced in the future and, as a consequence, it exploits the principle of locality only to some degree. It will favor the more recently loaded pages, which are likely to be referenced again in the near future. However, it is unable to deal with the fact that the program may return to pages referenced in the more distant past and start using those again. The FIFO algorithm will remove these pages because it only considers their absolute age in memory, which is unaffected by access patterns. Thus the principle of locality is often violated. Moreover, FIFO can also exhibit a strange, counterintuitive behavior, called **Belady's anomaly** [Belady_et_al,1969]: increasing the available memory can result in more page faults! (See exercises.)

Least Recently Used Replacement Algorithm (LRU)

The LRU algorithm has been designed to fully comply with the principle of locality, and it does not suffer from Belady's anomaly. If a page fault

occurs and there is no empty frame in memory, it will remove that page which has not been referenced for the longest time. To be able to make the correct choice, the algorithm must keep track of the relative order of all references to resident pages. Conceptually, this can be implemented by maintaining an ordered list of references. The length of this list is m, where m is the number of frames in main memory. When a page fault occurs, the list behaves as a simple queue: the page at the head of the queue is removed and the new page is appended to the end of the queue. Hence the queue length m remains constant. To maintain the chronological history of page references, the queue must, however, be reordered upon each reference—the referenced page is moved from its current position to the end of the queue, which gives it the greatest chance to remain resident.

Example:

The following table shows the current memory contents for the reference string $RS = cadbebabcd$ and $m = 4$ as in the previous examples. Below the memory, the queue is shown at each reference.

Time t	0	1	2	3	4	5	6	7	8	9	10
RS		c	a	d	b	e	b	a	b	c	d
Frame 0	a	a	a	a	a	a	a	a	a	a	d
Frame 1	b	b	b	b	b	b	b	b	b	b	b
Frame 2	c	c	c	c	c	e	e	e	e	e	d
Frame 3	d	d	d	d	d	d	d	d	d	c	c
IN_t						e				c	d
OUT_t						c				d	e
Queue end	d	c	a	d	b	e	b	a	b	c	d
	c	d	c	a	d	b	e	b	a	b	c
	b	b	d	c	a	d	d	e	e	a	b
Queue head	a	a	b	b	c	a	a	d	d	e	a

We assume that the pages were referenced in the order a, b, c, d, resulting in a being at the head and d at the end of the queue at time 0. During the next four references, the queue is reordered as each of the currently referenced pages is moved to the end of the queue (indicated by arrows). At time 5, a page fault occurs, which replaces page c—the current head of the queue—with the new page e. The latter is appended to the end of the queue while the former is removed. The next three references do not cause any page faults, however, the queue is reordered on each reference as shown. The next page fault at time 9 causes the removal of page d which, at that time, was the head of the queue and thus the least recently referenced one. Similarly,

the page fault at time 10 replaces the least recently referenced page e, thus bringing the total number of page faults produced by this algorithm to 3. □

The implementation of LRU as a software queue is impractical due to the high overhead with reordering the queue upon each reference. Several methods implemented directly in hardware to reduce this overhead have been developed. One implements a form of **time stamping**. Each time a page is referenced, the current content of an internal clock maintained by the processor is stored with the page frame. At a page fault, the page with the lowest time stamp is the one that has not been referenced for the longest time, and this is selected for replacement.

A similar scheme employs a **capacitor** associated with each memory frame. The capacitor is charged upon each reference to the page residing in that frame. The subsequent exponential decay of the charge can be directly converted into a time interval which permits the system to find the page which has not been referenced for the longest time.

Yet another technique uses an **aging register** R of n bits for each page frame:

$$R = R_{n-1}R_{n-2}\ldots R_1R_0$$

On a page reference, R_{n-1} of the referenced page is set to 1. Independently, the contents of all aging registers are periodically shifted to the right by one bit. Thus, when interpreted as a positive binary number, the value of each R decreases periodically unless the corresponding page is referenced, in which case the corresponding R contains the largest number. Upon a page fault, the algorithm selects the page with the smallest value of R to be replaced, which is the one that has aged the most.

Second-Chance (Clock) Replacement Algorithm

Due to the high cost of implementing LRU directly, even in hardware, a more economical alternative was developed. The Second-Chance Replacement scheme approximates the LRU scheme at a fraction of its cost. The algorithm maintains a circular list of all resident pages and a pointer to the current page in much the same way as the FIFO scheme described earlier. In addition, a bit u, called the **use** bit (or *reference* bit or *access* bit), is associated with each page frame. Upon each reference, the hardware automatically sets the corresponding use bit to 1.

To select a page for replacement the Second-Chance algorithm operates as follows. If the pointer is at a page with $u = 0$, then that page is selected for replacement and the pointer is advanced to the next page. Otherwise the

use bit is reset to 0 and the pointer is advanced to the next page on the list. This is repeated until a page with $u = 0$ is found, which in principle could take a complete pass through all page frames if all use bits were set to 1.

The name Second-Chance refers to the fact that a page with $u = 1$ gets a second chance to be referenced again before it is considered for replacement on the next pass. The algorithm is also frequently called the **Clock Replacement Algorithm**, because the pointer that cycles through the circular list of page frames may be visualized as scanning the circumference of a dial of a clock until a page with $u = 0$ is found.

Example:

For $RS = cadbebabcd$ and $m = 4$, the memory occupancy under the Second-Chance Algorithm changes as follows:

Time t	0	1	2	3	4	5	6	7	8	9	10
RS		c	a	d	b	e	b	a	b	c	d
Frame 0	→a/1	→a/1	→a/1	→a/1	→a/1	e/1	e/1	e/1	e/1	→e/1	d/1
Frame 1	b/1	b/1	b/1	b/1	b/1	→b/0	→b/1	b/0	b/1	b/1	→b/0
Frame 2	c/1	c/1	c/1	c/1	c/1	c/0	c/0	a/1	a/1	a/1	a/0
Frame 3	d/1	d/1	d/1	d/1	d/1	d/0	d/0	→d/0	→d/0	c/1	c/0
IN_t						e		a		c	d
OUT_t						a		c		d	e

The current value of the use bit is shown as a zero or one following each page. Initially, we have assumed all pages to have the use bit set. At the time of the first page fault (time 5), the pointer is at page a. Its use bit is reset and the pointer is advanced to the next page, b. b's use bit is also reset and the pointer is advanced to c. The same reset operation is applied to c and then to d, thus reaching the page a once more. Unless a has been referenced again in the meantime, its use bit is zero, causing that page to be replaced by the new page e. The pointer is advanced to page b. The next page fault at time 7 does not replace b because its use bit has again been set (at time 6). The search algorithm only resets $b's$ use bit and proceeds by replacing the following page c. The last two page faults occurs when c and d are re-referenced at time 9 and 10, respectively. The total number of page faults is 4. □

The Second-Chance algorithm approximates LRU in that frequently referenced pages will have their use bit set to one and thus will not be selected for replacement. Only when a page has not been referenced for the duration of a *complete* cycle of the pointer through all page frames, will it be replaced.

Before	After
u w	u w
1 1	0 1
1 0	0 0
0 1	0 0 *
0 0	select

Table 8.1: Changes of u and w bits under Second-Chance algorithm

The Third-Chance Algorithm

A replaced page must be written back onto secondary memory if its contents have been changed during its last occupancy of main memory; otherwise, the replaced page may simply be overwritten. A page that has been written into is frequently referred to as a **dirty** page. To be able to distinguish between the two types of pages, the hardware provides a **write** bit, w, associated with each page frame. When a new page is loaded, the corresponding write bit is 0; it is set to 1 when the information in that page is modified by a store instruction.

The pair of bits u (use bit) and w (write bit), associated with each page frame, is the basis for the Third-Chance algorithm. As was the case with the Second-Chance algorithm, a circular list of all pages currently resident in memory, and a pointer, are maintained. When a page fault occurs, the pointer scans the list, until it finds a page with both bits u and w equal to 0; this page is selected for replacement. Each time the pointer is advanced during the scan, the bits u and w are reset according to the rules shown in Table 8.1.

One additional complication is introduced by the fact that the two combinations $(0, 1)$ and $(1, 0)$ both yield the same combination $(0, 0)$. But in the first case, the page has been modified while in the second it has not. The algorithm must record this difference so that, prior to replacement, the modified page is written back onto secondary memory. The asterisk in Table 5-2 is used to indicate the modification, and can be implemented by an additional bit maintained for each frame.

The name Third-Chance derives from the fact that a page that has been written into will not be removed until the pointer has completed *two* full scans of the list. Thus, compared to a page that has not been modified,

it has one additional chance to be referenced again before it is selected for removal.

Example:

For this algorithm, in addition to specifying the reference string, $RS = cadbebabcd$, and the memory size, $m = 4$, we also need to know which references are write requests. For the sake of this example, we assume that the references at time 2, 4, and 6 are write requests, as indicated by the superscript w. The memory changes are then as follows:

Time t	0	1	2	3	4	5	6	7	8	9	10
RS		c	a^w	d	b^w	e	b	a^w	b	c	d
Frame 0	→a/10	→a/10	→a/11	→a/11	→a/11	a/00*	a/00*	a/11	a/11	→a/11	a/00*
Frame 1	b/10	b/10	b/10	b/10	b/11	b/00*	b/10*	b/10*	b/10*	b/10*	d/10
Frame 2	c/10	c/10	c/10	c/10	c/10	e/10	e/10	e/10	e/10	e/10	→e/00
Frame 3	d/10	d/10	d/10	d/10	d/10	→d/00	→d/00	→d/00	→d/00	c/10	c/00
IN_t						e				c	d
OUT_t						c				d	b

At time 1, page c is read, resulting in no changes. At time 2, page a is written, resulting in the corresponding w bit to be set. At time 3, the read operation produces no change, while the write at time 4 sets the corresponding w of b. Note that the pointer has not moved because there were no page faults. At the time of the first page fault, the pointer is at page a. The algorithm scans the pages while resetting the u and w bits as follows: reset u bit of a; reset u bit b; reset u bit of c; reset u bit of d; reset w bit of a and remember that the page is dirty (asterisk); reset w bit of b and remember that page as dirty. The next page c has now both bits equal to 0 and is replaced by the new page e, while the pointer is advanced to the next page d. Note that this search required almost two full passes of the pointer through the page frames.

The next three references (6 through 8) do not cause page faults but only modify the u and w bits of the pages a and b according to the reference type. (Note that the asterisk on page a is removed since the page has again been written into and thus its w bit is set.) The last two references replace pages d and b using the same principles, thus bringing the total number of page faults to 3. □

Case Study:

Berkeley Unix uses a form of the second-chance algorithm presented above. The main difference is that the selection of a page to evict from memory is not done at the time of a page fault. Rather, a page fault always uses a frame from the list of currently free frames. If this list is empty, the process blocks until a free frame becomes available.

Creating free page frames is the responsibility of a dedicated process, called the **page daemon**. This process wakes up periodically to check if there are enough free frames, as defined by a system constant. If not, the daemon creates free frames using the second-chance algorithm as follows. During the first pass over all frames, the daemon sets the *use bit* of all frames to zero. During the second pass, it marks all frames whose use bit is still zero, i.e., those not referenced since their examination during the first pass, as tentatively free. Such frames may be used to service a page fault when no actually free frames exist. At the same time, if a tentatively free frame is referenced again by the process, it may be reclaimed without a page fault by removing it from the tentatively free frame list.

□

8.3.2 Local Page Replacement Algorithms

Measurements of paging behavior indicate that each process requires a certain minimum set of pages to be resident in memory at all times in order to run efficiently; otherwise the page fault rate becomes unacceptably high. This condition, referred to as **thrashing**, will be discussed in the context of load control in Section 8.3.3. Furthermore, the size of this minimal set of resident pages changes dynamically as the process executes. These observations led to the development of several page replacement methods that attempt to maintain an *optimal* resident set of pages for each active process. These schemes are strictly local since the page replacement algorithm must distinguish between pages belonging to different processes. Thus a page fault caused by a given process will never be resolved by reducing the resident set of any other process.

Optimal Page Replacement Algorithm (VMIN)

Similar to the optimal replacement algorithm MIN, used as a global page replacement algorithm (Section 8.3.2), there exists an optimal replacement

algorithm, VMIN, which varies the number of frames according to the program's behavior [Prieve and Fabry, 1976]. As with MIN, it is unrealizable because it requires advance knowledge of the reference string.

VMIN employs the following page replacement rule, invoked after each reference. Consider a page reference at time t. If that page is currently not resident, it results in a page fault and the page is loaded into one of the free frames. Regardless of whether or not a page fault occurs, the algorithm then looks ahead in the reference string RS. If that page is *not* referenced again in the time interval $(t, t+\tau)$, where τ is a system constant, then the page is removed. Otherwise it remains in the process's resident set until it is referenced again. The interval $(t, t+\tau)$ is frequently referred to as a **sliding window**, since at any given time the resident set consists of those pages visible in that window. This contains the currently referenced page plus those pages referenced during the τ future memory accesses. Thus the actual window size is $\tau + 1$.

Example:

We use the reference string, $RS = ccdbcecead$. Let the size of the sliding window be defined by $\tau = 3$. Assume that there are enough free page frames to accommodate the set of pages that need to be resident. We do not show which page resides in which frame, since this is not the concern of a local replacement policy; instead, we show, using check marks, the pages of RS that are resident in memory at each reference. At time 0, the only resident page is page d. The following table then shows the changes in the resident set.

Time t	0	1	2	3	4	5	6	7	8	9	10
RS	d	c	c	d	b	c	e	c	e	a	d
Page a	–	–	–	–	–	–	–	–	–	√	–
Page b	–	–	–	–	√	–	–	–	–	–	–
Page c	–	√	√	√	√	√	√	√	–	–	–
Page d	√	√	√	√	–	–	–	–	–	–	√
Page e	–	–	–	–	–	–	√	√	√	–	–
IN_t		c			b		e			a	d
OUT_t					d	b			c	e	a

At time 0, page d is referenced. Since it is referenced again at time 3, which is within the interval $(0, 0+\tau) = (0, 0+3) = (0, 3)$, it is not removed. The first page fault occurs at time 1, which causes the page c to be loaded into a free frame; the set of resident pages now consists of two pages, c and d. At time 2 and 3, both pages c and d are still retained because they

are referenced again within the current window. Page d is removed from memory at time 4, since it is not referenced again within $(4, 4+3) = (4, 7)$. Instead, the faulting page b is loaded into one of the free frames. But b falls out of the current window at the very next time point 5 and is removed, while c continue to be resident. The next page fault at time 6 brings in page e, which remains resident until its next reference at time 8. The last two references bring in the pages a and d respectively.

The total number of page faults for VMIN in this example is 5. The resident set size varies between 1 and 2 and thus at most two page frames are occupied at any time. By increasing τ, the number of page faults can arbitrarily be reduced, of course at the expense of using more page frames. □

The Working Set Model (WS)

The Working Set model [Denning, 1968, 1980] attempts a practical approximation to VMIN and employs a similar concept of a sliding window; however, the algorithm does not look *ahead* in the reference string (since this information is not available), but looks *behind*. It relies heavily on the principle of locality discussed previously, which implies that, at any given time, the amount of memory required by a process in the near future may be estimated by considering the process's memory requirements during its recent past.

According to the model, each process at a given time t has a working set of pages $W(t, \tau)$, defined as the set of pages referenced by that process during the time interval $(t-\tau, t)$, where τ is again a system-defined constant.

The memory management strategy under the Working Set model is then governed by the following two rules:

1. At each reference, the current working set is determined and only those pages belonging to the working set are retained in memory.

2. A process may run if and only if its entire current working set is in memory.

Example:

Let $RS = ccdbceceead$, $\tau = 3$, and the initial resident set at time 0 contain the pages a, d, and e, where a was referenced at time $t = 0$, d was referenced at time $t = -1$, and e was referenced at time $t = -2$. The following table shows the working set at each reference. As with the previous example of

VMIN, we assume that a list of free page frames is available to accommodate the current working set. If not enough page frames were available, a load control mechanism, discussed in Section 8.3.3, would decide which process to deactivate to create more space.

Time t	0	1	2	3	4	5	6	7	8	9	10
RS	a	c	c	d	b	c	e	c	e	a	d
Page a	√	√	√	√	–	–	–	–	–	√	√
Page b	–	–	–	–	√	√	√	√	–	–	–
Page c	–	√	√	√	√	√	√	√	√	√	√
Page d	√	√	√	√	√	√	√	–	–	–	√
Page e	√	√	–	–	–	–	√	√	√	√	√
IN_t		c			b		e			a	d
OUT_t			e		a			d	b		

The first page fault occurs at time 1 and, as a result, the corresponding page c is loaded into one of the available page frames. The other three currently resident pages a, d, e are still visible in the window $(1-3,1) = (-2,1)$ and thus are retained. At time 2, page e falls out of the current window $(2-3,2) = (-1,2)$ and is removed. At time 4, the page fault brings in page b; this takes the place of page a, which fell out of the current window $(4-3,4) = (1,4)$. The next page fault at time 6 brings in page e, while the currently resident pages b, d, and c remain resident as part of the current working set defined by $(6-3,6) = (3,6)$. Over the next two references, the working set shrinks to only 2 pages e and c and grows again to four pages as a result of the last two page faults at times 9 and 10.

The total number of page faults for this algorithm is also 5. The working set size fluctuates between 2 and 4 page frames as the reference string is processed. □

While conceptually very attractive, the Working Set strategy is difficult to implement in its full generality. One problem is estimating the appropriate window size τ, which is a crucial parameter of the algorithm. This is usually performed empirically, by varying τ until the highest performance is achieved. Except for rare cases when anomalous behavior has been observed [Franklin, Graham, Gupta, 1978], increasing the window size reduces the paging activity of each process. The trade-off is, of course, a reduced number of processes that may be run concurrently.

A more serious problem with a pure Working Set approach is the large overhead in implementation since the current working set may change with each reference. To alleviate this problem, special hardware may be provided

to keep track of the working set. Alternately, approximations of the Working Set algorithm have been developed which permit the working set to be reevaluated less frequently. These generally involve some combination of reference bits and/or time stamps, which are examined and modified at fixed intervals determined by a periodic interrupt.

One such scheme uses aging registers similar to those used to approximate the LRU algorithm (Section 8.3.1). The left-most bit of an aging register is set to 1 whenever the corresponding page is referenced. Periodically, every δ memory references (where δ is a system constant), a time-out interrupt is generated and the contents of all aging registers are shifted to the right. Thus an aging register gradually decreases in value unless the page is referenced. When the aging register reaches 0 (or some other system-defined threshold value), the page is removed from the working set of the process because it fell out of the sliding window; the size of the window is $\delta * n$, where n is the number of bits comprising the aging registers.

Example:

For example, assume 3-bit aging registers and a time-out interval of 1000 references. The aging register of a page referenced at time t will be '100' at time t, '010' at time $t + 1000$, '001' at time $t + 2000$, and it will reach '000' at time $t + 3000$, at which time the page will be removed. This effectively approximates a working set with a window size $3 * 1000 = 3000$. □

A similar approximation can be implemented with a single use bit and a time stamp associated with each frame. The use bit is turned on by the hardware whenever the page is accessed. The use bits and time stamps are examined periodically at least every δ instructions using a timeout interrupt. When the use bit of the frame is found to be 1, it is reset and the time of this change is recorded with the frame as its time-stamp. When the bit is found 0, the time since it has been turned off, denoted as t_{off}, is computed by subtracting the value of the time stamp from the current time. The t_{off} time keeps increasing with each timeout unless the page has been referenced again in the meantime, causing its use bit to be set to 1. The time t_{off} is compared against a system parameter, t_{max}. If $t_{off} > t_{max}$, the page is removed from the working set, i.e., the frame is released.

Case Study

The Virtual Memory Operating System (VMOS) employs the above scheme of time stamps and use bits, but implements a further refinement of the basic scheme. It maintains two system thresholds, t_{max}

and t_{min}. If $t_{off} > t_{max}$, the page is removed from the working set as before. If $t_{min} < t_{off} < t_{max}$, the page is removed *tentatively* from the working set by putting the frame on a special tentatively-released list. When a new page frame is needed and there are no free frames available, one of the tentatively released pages is used. Otherwise the faulting process (or some other process) must be swapped out and its page frames added to the free page frame list. This two-tier system guarantees preferential treatment to pages which have not yet exceeded t_{max} but have not been used for a significant length of time (at least t_{min}).

□

Page Fault Frequency Replacement Algorithm (PFF)

One of the main objectives of any page replacement scheme must be to keep the number of page faults to a minimum. In the Working Set model described above, this objective was accomplished indirectly by adjusting the working set size. The Page Fault Frequency algorithm takes a direct approach by actually measuring the page fault frequency in terms of time intervals between consecutive page faults. These times are then used to adjust the resident page set of a process at the time of each page fault. The faulting page is of course loaded into memory, increasing the size of the resident set of pages. The following rule then guarantees that the resident set does not grow unnecessarily large:

> *If the time between the current and the previous page fault exceeds a critical value τ, all pages not referenced during that time interval are removed from memory.*

Here is a more formal specification. Let t_c be the time of the current page fault and t_{c-1} the time of the previous page fault. Whenever $t_c - t_{c-1} > \tau$, where τ is a system parameter, all pages not referenced during the interval (t_{c-1}, t_c) are removed from memory. Consequently, the set of resident pages at time t_c is described as:

$$resident(t_c) = \begin{cases} RS(t_{c-1}, t_c), & \text{if } t_c - t_{c-1} > \tau \\ resident(t_{c-1}) + RS(t_c), & \text{otherwise} \end{cases}$$

where $RS(t_{c-1}, t_c)$ denotes the set of pages referenced during the interval (t_{c-1}, t_c) and $RS(t_c)$ is the page referenced at time t_c (and found missing from the resident set).

The main advantage of the Page Fault Frequency algorithm over the Working Set model is efficiency in implementation. The resident set of pages is adjusted only at the time of a page fault instead of at each reference.

Example:

We use the same reference string, $RS = ccdbcecead$, as before and set the parameter τ to 2. Let the resident set at time 0 consist of the pages a, d, and e. The following table shows the set of resident pages at each reference.

Time t	0	1	2	3	4	5	6	7	8	9	10
RS		c	c	d	b	c	e	c	e	a	d
Page a	√	√	√	√	–	–	–	–	–	√	√
Page b	–	–	–	–	√	√	√	√	√	–	–
Page c	–	√	√	√	√	√	√	√	√	√	√
Page d	√	√	√	√	√	√	√	√	√	–	√
Page e	√	√	√	√	–	–	√	√	√	√	√
IN_t		c			b		e			a	d
OUT_t					a,e					b,d	

The first page fault occurs at time 1. Let's assume that the previous page fault occurred recently and so no pages are removed at this time. The next page fault occurs at time 4. Since $t_c = 4$ and $t_{c-1} = 1$, the condition $t_c - t_{c-1} > \tau$ is true, and thus all pages not referenced during the time interval (1,4) are removed; these are the pages a and e. The next page fault occurs at time 6. No pages are removed at this time because $6 - 4 = 2$ is not greater then τ. But pages b and d are removed at the next page fault at time 9, because the condition is again true. No pages are removed at the last page fault at time 10. The Page Fault Frequency algorithm produces a total of 5 page faults for the given string RS, while the number of resident pages fluctuates between 3 and 4. □

Case Study:

Windows 2000 implements a page replacement scheme that combines the features of several of the algorithms discussed above, notably, the working set model and the second chance algorithm.

The replacement is *local* in that a page fault will not cause the eviction of a page belonging to another process. Thus the system maintains a *current working set* for each process. The size of the working set, however, is not determined automatically using a sliding window. Instead,

a *minimum* and a *maximum* size is assigned by the system. Typical values for the minimum are 20-50 page frames, depending on the size of physical memory; typical values for the maximum are 45-345 page frames.

At each page fault, the working set of a process is increased by adding the referenced page to the set, until the maximum is reached. At that time (unless memory is plentiful, which is determined by a system constant), a page must be removed from the working set to accommodate the new page.

Whenever memory becomes scares—which is measured by the number of free frames currently available—the working sets of some processes must be decreased. This is accomplished by a routine called the **working set manager**, which is invoked periodically (every second). It examines processes that have more than their minimum of pages and orders these processes according to their size, idle time, and other criteria to select the most suitable candidates for working set reduction.

Once a process has been selected, the system must choose which page(s) to remove. For this it uses a variation of the clock algorithm, which approximates the LRU policy. Each page frame has a use bit, u, and a counter, *cnt*. The use bit (called access bit in Windows 2000) is set by the hardware whenever the page is accessed. When looking for a page to evict from the working set, the working set manager scans the use bits of the pages within the working set. During each pass, it performs the following operations:

```
if (u == 1) {u = 0; cnt = 0}
else cnt++
```

At the end of the pass it evicts the page with the highest values of *cnt*. Thus as long as a page is being referenced frequently, its *cnt* value will be low and it will not be evicted.

Another interesting feature of the page replacement scheme is that the eviction of a page from a working set gradual. First, the selected page frame is placed on one of two lists: one holds the *tentatively* removed pages that have been modified; the other holds *tentatively* removed read-only pages. But the pages remain in memory and, if referenced

again, can quickly be removed from the list, without causing a page fault. They are used to satisfy page faults only when the list of actually free pages is empty.

□

8.3.3 Load Control and Thrashing

When implementing a virtual memory management scheme, one of the main problems is to maintain a balance between the degree of multiprogramming and the amount of paging (i.e., page faults) that occurs. On the one hand, we would like to keep as many processes running concurrently as possible to keep CPU utilization high. On the other hand, since all active processes must compete for the same pool of memory frames, their resident page sets get smaller as the multiprogramming level increases, resulting in more page faults. **Load control** refers to the policy that determines the number and the type of processes that should be active concurrently, thus competing for memory, CPU time, and other critical resources. The goal is to maximize overall performance.

Three basic question must be answered for a load control mechanism:

1. How to decide when to increase or decrease the degree of multiprogramming?
2. Which of the currently active tasks should be swapped out if the degree of multiprogramming must be reduced?
3. When a new process is created or a previously suspended process is reactivated, which of its pages should be loaded into memory as its current resident set?

The following sections address each of these questions in turn.

Choosing the Degree of Multiprogramming

In the case of local page replacement schemes, such as the Working Set model or the Page Fault Frequency algorithm, the first question is answered by the replacement method itself. Since each process has a well-defined resident set, the system can increase the degree of multiprogramming only to the point where all memory is allocated. Only when the current resident sets shrink due to changing patterns in process behavior, or when one or more

Figure 8.11: Processor utilization

processes are deactivated, can new processes be added to the currently active set. Conversely, if a process's resident set expands, it may be necessary to swap out one or more of the currently active processes in order to create the necessary space.

With global page replacement, the degree of multiprogramming is not determined automatically. Thus explicit load control criteria must be provided to guarantee that each active process has some minimum number of resident pages. With too many active processes the page fault rate can increase to the point where most of the system's effort is expended on moving pages between main and secondary memory. Such a condition is termed **thrashing**.

Figure 8.11 illustrates the problem of thrashing. The solid curve shows the effect of multiprogramming on CPU utilization for a given fixed size memory. The horizontal axis gives the degree of multiprogramming, N, i.e., the number of active processes sharing main memory. The vertical axis shows the corresponding CPU utilization. Initially, the curve rises rapidly toward high CPU utilization but it slows down as more processes are added. Eventually, it drops down again toward zero. It is this latter phase that is referred to as thrashing and is the direct result of too many processes competing for a fixed number of frames. Since each process is able to execute only a few instructions before it encounters a non-resident page, all processes are blocked most of the time waiting for their page to be moved into main memory. As a result, the CPU is idle most of the time. Its main activity becomes the housekeeping related to paging, while the amount of useful work being accomplished is minimal.

The goal of an effective load control policy must then be to keep the level of multiprogramming at the point where the CPU utilization is highest, i.e., at point N_{max} in the figure. Unfortunately, CPU utilization varies at run time due to a number of circumstances and is difficult to monitor accurately. Thus we need some other measurable criterion that would tell us at any given point in time whether CPU utilization has reached its attainable peek.

A number of schemes have been proposed and studied to solve this problem. Since the main objective is to minimize the amount of paging, most criteria for determining the optimal degree of multiprogramming are based on the rate of paging. One example is the **L=S criterion**, where L is the mean time between faults and S is the mean page fault service time, i.e.,

the time it takes to replace a page in main memory. Intuitively, when L is much larger than S, the paging disk is underutilized. In contrast, when S exceeds L, there are more page faults than the paging disk can handle. When L approaches S, both the paging disk and CPU generally reach their maximum utilization.

The dashed curve in Figure 8.11 shows the L/S ratio—the utilization of the paging disk—which depresses the CPU utilization when the degree of multiprogramming is too high. Thus to achieve the highest overall performance, the **L=S criterion** criterion selects the point $N_{L=S}$ as the desired level of multiprogramming. Note that this is slightly higher than the optimum, N_{max}, but the deviation is usually tolerable for the purposes of load control. The curve is based on extensive measurements in actual computer systems and is valid under general conditions that apply to almost all real multiprogramming systems [Denning, 1980].

Another basis for choosing the level of multiprogramming, referred to as the **50% criterion** [Denning et al., 1976], also relies on measuring the page fault rate. Using this criterion, the system maintains a level of multiprogramming such that the paging disk is busy 50% of the time. This is based on the observation that CPU utilization tends to be highest when the utilization of the paging device is approximately 50%. Both the $L = S$ and the 50% criteria are closely related.

Yet another criterion, called **Clock Load Control**, has been proposed by Carr [1981]. It is applicable to schemes which employ a pointer to scan the list of page frames, when searching for a page to be replaced. The method is based on the rate at which the pointer travels around the page frame list. If this rate is low, one of the following two conditions is true:

1. The page fault rate is low, resulting in few requests to advance the pointer, or

2. The mean pointer travel between page faults is small, indicating that there are many resident pages that are not being referenced and are readily replaceable.

In both cases the level of multiprogramming might usefully be increased. Conversely, if the travel rate of the pointer increases past a certain threshold value, the level of multiprogramming should be decreased.

Figure 8.12: Lifetime curve of a program

Choosing the Process to Deactivate

When the degree of multiprogramming needs to be decreased, the system must choose which of the currently active processes to deactivate. Some of the possible choices for selecting a process to evict are the following:

- the *lowest priority process*: follow the same rules as the process scheduler and thus try to maintain a consistent scheduling policy
- the *faulting process*: eliminate the process that would be blocked anyway while waiting for its page to be loaded
- the *last process activated*: the most recently activated process is considered the least important
- the *smallest process*: the smallest process is the least expensive to swap in and out
- the *largest process*: free up the largest number of frames

In general, it is not possible to determine the superiority of one choice over another since each depends on many other policies and system parameters, including the scheduling methods used in a given system. Hence the decision will rely upon the intuition and experience of the system's designer, as well as the particular application.

Prepaging

In a system with static allocation, all pages belonging to an active process are loaded before the process is permitted to run. In a dynamically allocated system, pages are loaded only as a result of page faults. The latter method works well when the process already has some reasonable set of pages in memory since page faults do not occur too frequently. Before this set is established, however, the page fault rate will be high. This will be the case when a newly created process is started or when an existing process is reactivated as a result of a satisfied request.

The paging behavior can be captured by a process **lifetime curve**, which shows how the mean time between page faults increases with the mean set

of pages resident in memory [Denning, 1980]. The shape of this lifetime curve, shown in Figure 8.12, is of particular interest. It shows that, as long as the resident set of pages is small, page faults occur frequently. When the set reaches a certain size, the page fault rate decreases dramatically and, consequently, the mean time between page faults increases accordingly. This continues until the curve reaches a *knee* beyond which the benefits of adding more pages to the resident set begin to diminish. This suggests that a process should not be started or reactivated with an empty or a very small resident set. Rather, a set of pages should be **prepaged** at the time of activation, which can be done more efficiently than loading one page at a time.

Theoretically, the optimal size of the prepaged set is given by the point at which the life time curve reaches its knee. In practice, however, this value is difficult to estimate. Thus for a newly created process, prepaging is rarely used. In the case of an existing process, i.e., one that has been blocked temporarily while waiting for an I/O completion or some other event, the obvious choice is the set of pages that were resident just before the process was suspended. This is particularly attractive with schemes based on the working set model, where a list of currently resident pages is explicitly maintained.

8.3.4 Evaluation of Paging

Systems that employ paging have two main advantages over non-paged systems. The first is a very simple placement strategy. Since the basic unit of memory allocation is a fixed-size page, programs and data need not occupy contiguous areas of memory. Thus, if a request for k pages of memory is made and there are at least k free frames, the placement decision is straight forward—any frame may be allocated to any page.

The second advantage is the elimination of external fragmentation, which occurs with variable memory partitions. That's because page frames form a uniform sequence without any gaps. However, the problem of fragmentation does not disappear entirely. Because the sizes of programs and data are rarely multiples of the page size, the last page of each virtual memory space is generally only partially filled. This problem of **internal** fragmentation occurs with any scheme using fixed memory partitions, including paging. The magnitude of the problem depends on the page size relative to the size of the virtual memory spaces (e.g. segments) that are mapped onto these pages.

Many early experiments were carried out to study the dynamic behavior

Figure 8.13: Qualitative program behavior under paging: (a) percentage of referenced pages, (b) portion of each page actually used, (c) effect of page size on page fault rate, (d) effect of memory size on page fault rate

of programs under paging [Belady, 1966; Coffman and Varian, 1968; Baer and Sager, 1972; Gelenbe, Tiberio and Boekhorst, 1973; Rodriguez-Rosell, 1973; Opderbeck and Chu, 1974; Lenfant and Burgevin, 1975; Sadeh, 1975; Spirn, 1977; Gupta and Franklin, 1978]. The most significant parameters that were varied were the page size and the amount of available memory. The results obtained from these independent studies were generally consistent with one another. They can be summarized by the diagrams shown in Figure 8.13; each illustrates the behavior of an individual process under different conditions.

1. Figure 8.13(a) shows how pages are referenced by a process over time. The curve rises sharply at the beginning, indicating that the process requires a certain percentage of its pages within a very short time period after activation. After a certain working set is established, additional pages are demanded at a much slower rate. For example, about 50% of a process's total number of pages were referenced on the average during a single quantum in a typical time-sharing operation.

2. Usage within a page is illustrated in Figure 8.13(b). It shows that a relatively small number of instructions within any page are executed before control moves to another page. For page sizes of 1024 words, less than 200 instructions were executed before another page was referenced in most of the samples investigated.

3. Figure 8.13(c) describes the effect of page size on the page fault rate. Given a fixed size of main memory, the number of page faults increases as the page size increases; that is, a given memory within many small pages has fewer faults than the same memory filled with larger pages.

4. The effect of memory size on the page fault rate appears in Figure 8.13(d). For a given fixed page size, the page fault rate rises exponentially as the amount of available memory decreases. The point labeled W represents a minimum amount of memory required to avoid thrashing, i.e., a page fault rate that causes CPU to be grossly underutilized as it needs to wait for pages to be transfered into memory.

These results provide important guidelines for the design of paging systems. Figure 8.13(a), which is a direct consequence of the principle of locality, underscores the importance of prepaging. When a process is reactivated, for example, after an I/O completion, it should not be restarted with just the current page. This would cause it to generate a large number of page faults before it was able to reestablish its working set. Prepaging can load these pages much more efficiently than pure demand paging.

Figures 8.13(b) and (c) address the question of page size. Both suggest that page size should be *small*. In the case of (b), smaller page sizes would eliminate many of the instructions that are not referenced, but must be brought into memory as part of the larger pages. An additional argument in favor of a small page size, not reflected in the figures, is that the amount of memory wasted due to internal fragmentation is also reduced. Smaller pages provide a finer resolution, and thus result in a closer fit for the variable size of program and data segments.

There are, however, important counter arguments in support of a relatively *large* page size. First, larger page sizes require smaller page tables to keep track of a given amount of virtual memory, which reduces memory overhead. Large pages also mean that main memory is divided into a smaller number of physical frames; thus, the cost of the hardware necessary to support paging is less. Finally, the performance of disks is dominated by the seek time and the rotational delay, while the actual data transfer time is negligible. As a result, transferring a contiguous block of data is much more efficient than transferring the same amount of data distributed over multiple smaller blocks. This too favors a larger page size.

In older systems, pages were generally in the range from 512 bytes to 4K bytes (for example, the IBM 360/370). Due to the increases in CPU speed and other technological advances, the trend has been toward increasingly larger page sizes. Many contemporary systems use page sizes of 16K bytes or even as large as 64K bytes.

Figure 8.13(d) underscores the importance of effective load control. It illustrates that there is a critical amount of memory a process needs to run efficiently; if this is not provided, the process' paging behavior deteriorates rapidly, resulting in thrashing, which only waste the system's resources.

Contents