

Memory Management

- To provide a detailed description of various ways of organizing memory hardware
- To discuss various memory-management techniques, including paging and segmentation
- To provide a detailed description of the Intel Pentium, which supports both pure segmentation and segmentation with paging
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Register access in one CPU clock (or less)
- Main memory can take many cycles
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation

Base and Limit Registers

A pair of **base** and **limit** registers define the logical address space

Binding of Instructions and Data to Memory

- Address binding of instructions and data to memory addresses can happen at three different stages
- **Compile time:** If memory location known a priori, **absolute code** can be generated; must recompile code if starting location changes
- **Load time:** Must generate **relocatable code** if memory location is not known at compile time
- **Execution time:** Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., base and limit registers)

Multistep Processing of a User Program

- The concept of a logical address space that is bound to a separate **physical address space** is central to proper memory management
- **Logical address** – generated by the CPU; also referred to as **virtual address**
- **Physical address** – address seen by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme

Memory-Management Unit (MMU)

Hardware device that maps virtual to physical address

In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory

The user program deals with *logical* addresses; it never sees the *real* physical addresses

Dynamic relocation using a relocation register**Dynamic Loading**

- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No special support from the operating system is required implemented through program design

Dynamic Linking

- Linking postponed until execution time
- **Small piece of code, *stub*, used to locate the appropriate memory-resident library routine**
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system needed to check if routine is in processes' memory address
- Dynamic linking is particularly useful for libraries
- System also known as **shared libraries**

Swapping

A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution

n**Backing store** – fast disk large enough to accommodate copies of all memory images for all users;
must provide direct access to these memory images

n**Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed

nMajor part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped

nModified versions of swapping are found on many systems (i.e., UNIX, Linux, and

Windows) System maintains a **ready queue** of ready-to-run processes which have memory images on

Contiguous Allocation

Main memory usually into two partitions:

Resident operating system, usually held in low memory with interrupt vector

User processes then held in high memory

nRelocation registers used to protect user processes from each other, and from changing operating- system code and data

Base register contains value of smallest physical address

- **Firs** Limit register contains range of logical addresses – each logical address must be less than the limit register

Hardware Support for Relocation and Limit Registers

- Multiple-partition allocation
- Hole – block of available memory; holes of various size are scattered throughout memory
- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Operating system maintains information about:
 - a) allocated partitions
 - b) free partitions (hole)
- **Best-fit:** Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size
Produces the smallest leftover hole
- **Worst-fit:** Allocate the *largest* hole; must also search entire list
- Produces the largest leftover hole
- First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Fragmentation

- **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation** – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- Reduce external fragmentation by **compaction**
- Shuffle memory contents to place all free memory together in one large block
- Compaction is possible *only* if relocation is dynamic, and is done at execution time.
- I/O problem
 - Latch job in memory while it is involved in I/O
 - Do I/O only into OS buffers

Paging

Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available

Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8,192 bytes)

Divide logical memory into blocks of same size called **pages**

nKeep track of all free frames

To run a program of size n pages, need to find n free frames and load program

Set up a page table to translate logical to physical addresses

Internal fragmentation

Address Translation Scheme

Address generated by CPU is divided into

Page number (p) – used as an index into a *pagetable* which contains base address of each page in physical memory

For given logical address space 2^m and page size 2^n

Structure of the Page Table

In this section, we explore some of the most common techniques for structuring the page table.

1. Hierarchical Paging

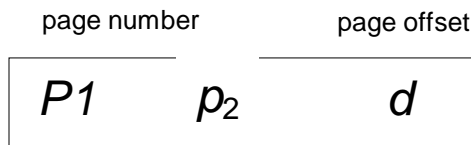
Most modern computer systems support a large logical address space (2^{32} to 2^{64}). In such an environment, the page table itself becomes excessively large.

For example, consider a system with a 32-bit logical address space. If the page size in such a system is 4 KB (2^{12}), then a page table may consist of up to 1 million entries ($2^{32}/2^{12}$). Assuming that each entry consists of 4 bytes, each process may need up to 4 MB of physical address space for the page table alone. Clearly, we would not want to allocate the page table contiguously in main memory.

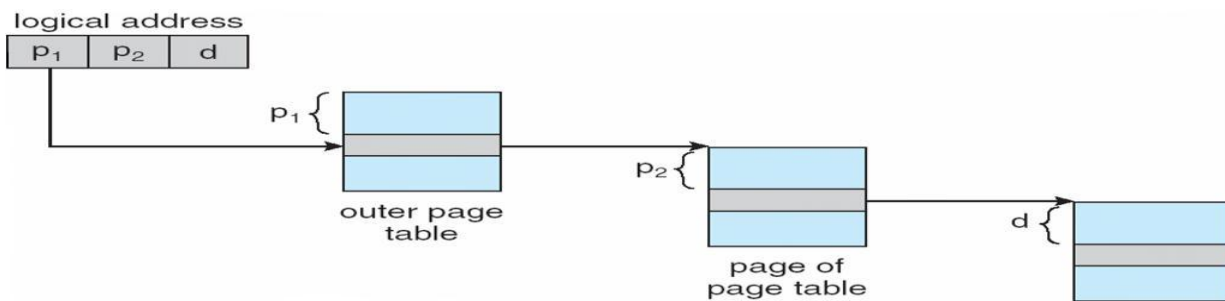
One simple solution to this problem is to divide the page table into smaller pieces. We can accomplish this division in several ways. One way is to use a two-level paging algorithm, in which the page table itself is also paged. Remember our example of a 32-bit machine

A two-level page-table scheme.

with a page size of 4 KB. A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits. Because we page the page table, the page number is further divided into a 10-bit page number and a 10-bit page offset. Thus, a logical address is as follows:



where P_1 is the page number of the outer page table and p_2 is the displacement within the page of the outer page table. The address-translation method for this architecture is shown in Figure. Because address translation works from the outer page table inward, this scheme is also known as a forward-mapped page table.



Address translation for a two-level 32-bit paging architecture.

For a system with a 64-bit logical-address space, a two-level paging scheme is no longer appropriate. To illustrate this point, let us suppose that the page size in such a system, is 4 KB (2^{12}). In this case, the page table consists of up to 2^{52} entries. If we use a two-level paging scheme, then the inner page tables can conveniently be one page long, or contain 2^{10} 4-byte entries. The addresses look like this:

The outer page table consists of 2^{42} entries, or 2^{44} bytes. The obvious way to avoid such a large table is to divide the outer page table into smaller pieces. This approach is also used on some 32-bit processors for added flexibility and efficiency.

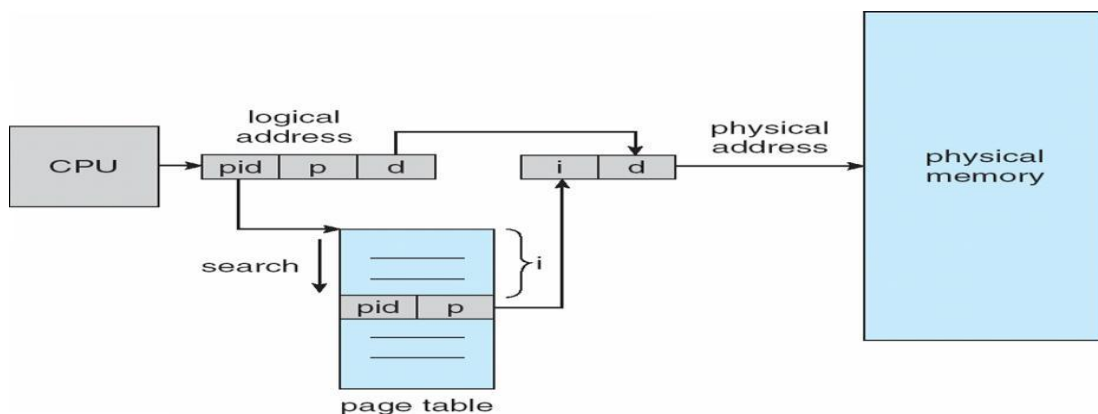
We can divide the outer page table in various ways. We can page the outer page table, giving us a three-level paging scheme. Suppose that the outer page table is made up of standard-size pages (2^{32} entries, or 2^{12} bytes); a 64-bit address space is still daunting: The outer page table is still 2^{32} bytes in size.

2. Hashed Page Tables

A common approach for handling address spaces larger than 32 bits is to use a **hashed** page table, with the hash value being the virtual page number. Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions). Each element consists of three fields: (1) the virtual page number, (2) the value of the mapped page frame, and (3) a pointer to the next element in the linked list.

The algorithm works as follows: The virtual page number in the virtual address is hashed into the hash table. The virtual page number is compared with field 1 in the first element in the linked list. If there is a match, the corresponding page frame (field 2) is used to form the desired physical address. If there is no match, subsequent entries in the linked list are searched for a matching virtual page number. This scheme is shown in Figure.

To solve this problem, we can use an **inverted page table**. An inverted page table has one entry for each real page (or frame) of memory. Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page. Thus, only one page table is in the system, and it has only one entry for each page of physical memory. Figure shows the operation of an inverted page table.



Inverted Page Table Architecture

Each virtual address in the system consists of a triplet $\langle \text{process-id, page-number, offset} \rangle$. Each inverted page-table entry is a pair $\langle \text{process-id, page-number} \rangle$ where the process-id assumes the role of the address-space identifier. When a memory reference occurs, part of the virtual address, consisting of $\langle \text{process-id, pagenumber} \rangle$, is presented to the memory subsystem. The inverted page table is then searched for a match. If a match is found—say, at entry i —then the physical address $\langle i, \text{offset} \rangle$ is generated. If no match is found, then an illegal address access has been attempted.

Although this scheme decreases the amount of memory needed to store each page table, it increases the amount of time needed to search the table when a page reference occurs. Because the inverted page table is sorted by physical address, but lookups occur on virtual addresses, the whole table might need to be searched for a match. This search would take far too long.

5. Segmentation

Segmentation is a memory-management scheme that supports this user view of memory. A logical address space is a collection of segments. Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. The user therefore specifies each address by two quantities: a segment name and an offset. For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name. Thus, a logical address consists of a *two tuple*: $\langle \text{segment-number, offset} \rangle$.

Address mapping is effected by a segment table. Each entry in the segment table has a *segment base* and a *segment limit*. The segment base contains the starting physical address where the segment resides in memory, whereas the segment limit specifies the length of the segment. The use of a segment table is illustrated in Figure.

Segmentation hardware.

A logical address consists of two parts: a segment number, s , and an offset into that segment, d . The segment number is used as an index to the segment table. The offset d of the logical address must be between 0 and the segment limit. If it is not, we trap to the operating system (logical addressing attempt beyond end of segment).

When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte. The segment table is thus essentially an array of base-limit register pairs.

VIRTUAL MEMORY MANAGEMENT

1.Virtual memory

Virtual memory is a technique that allows the execution of processes that are not completely in memory. One major advantage of this scheme is that programs can be larger than physical memory. This technique frees programmers from the concerns of memory limitations.

The entire process to be executed must be in physical memory is reasonable, but infact the entire program is not needed. For instance, consider the following:

- Programs often have code to handle unusual error conditions. Since these errors rarely occur in practice, this code is almost never executed.
- Arrays, lists, and tables are often allocated more memory than they actually need.
- Certain options and features of a program may be used rarely.

Even in those cases where the entire program is needed, it may not all be needed at the same time. The ability to execute a program that is partially in memory would provide many benefits:

- A program would no longer be constrained by the amount of physical memory that is available.
- Because each user program could take less physical memory, more programs could be run at the same time, with a corresponding increase in CPU utilization and throughput but with no increase in response time or turnaround time.
- Less I/O would be needed to load or swap each user program into memory, so each user program would run faster.

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

Virtual memory also allows files and memory to be shared by two or more processes Through page sharing. This leads to the following benefits:

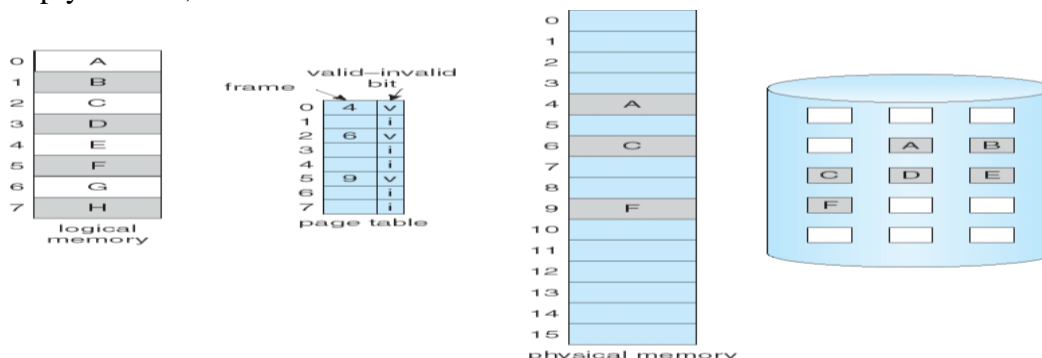
- System libraries can be shared by several processes through mapping of the shared object into a virtual address space.
- Similarly, virtual memory enables processes to share memory. Virtual memory allows one process to create a region of memory that it can share with another process.
- Virtual memory can allow pages to be shared during process creation with the forkO system call, thus speeding up process creation.

2.Demand Paging

Load the entire program in physical memory at program execution time is called pre paging. Load pages only as they are needed are known as demand paging and is commonly used in virtual memory systems. With demand-paged virtual memory, pages are only loaded when they are demanded during program execution

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

When this bit is set to "valid/" the associated page is both legal and in memory. If the bit is set to "invalid," the page either is not valid or is valid but is currently on the disk. The page-table entry for a page that is brought into memory is set as usual, but the page-table entry for a page that is not currently in memory is either simply marked, invalid .



Page table when some pages are not in main memory.

Notice that marking a page invalid will have no effect if the process never attempts to access that page. But if the process tries to access a page that was not brought into memory, it causes a page-fault trap. The procedure for handling this page fault is.

Steps in Handling Page Replacement

1. We check an internal table for this process to determine whether the reference was a valid or an invalid memory access.
2. If the reference was invalid, we terminate the process. If it was valid, but we have not yet brought in that page, we now page it in.
3. We find a free frame (by taking one from the free-frame list, for example).
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the trap.

The process can now access the page as though it had always been in memory.

In the extreme case, we can start executing a process with *no* pages in memory. When the operating system sets the instruction pointer to the first instruction of the process, which is on a disk, the page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory. This scheme is pure demand paging.

If the page fault occurs on the instruction fetch, we can restart by fetching the instruction again. If a page fault occurs while we are fetching an operand, we must fetch and decode the instruction again and then fetch the operand

Performance of Demand Paging

Demand paging can significantly affect the performance of a computer system. As long as we have no page faults, the effective access time is equal to the memory access time. If, however, a page fault occurs, we must first read the relevant page from disk and then access the desired word.

Let p be the probability of a page fault ($0 \leq p \leq 1$). We would expect p to be close to zero—that is, we would expect to have only a few page faults. The effective access time is then

$$\text{Effective access time} = (1 - p) \times ma + p \times \text{page fault time.}$$

To compute the effective access time, we must know how much time is needed to service a page fault. A page fault causes the following sequence to occur:

1. Service the page-fault interrupt.
2. Read in the page.
3. Restart the process.

The first and third tasks can be reduced, with careful coding, to several hundred instructions. These tasks may take from 1 to 100 microseconds each. The page-switch time, however, will probably be close to 8 milliseconds and a memory-access time of 200 nanoseconds, then the effective access time is

$$\begin{aligned} \text{Effective access time} &= (1 - p) \times (200) + p (8 \text{ milliseconds}) \\ &= (1 - p) \times 200 + p \times 8.00(1000) \\ &= 200 + 7,999,800 \times p. \end{aligned}$$

We see that the effective access time is directly proportional to the page-fault rate. The computer will be slowed down because of demand paging.

Copy-on-Write

Process creation using the `fork ()` system call may initially bypass the need for demand paging by using a technique similar to page sharing. The `fork()` system call creates a child process as a duplicate of its parent. We can use a technique known as copy-on-write, which works by allowing the parent and child processes initially to share the same pages. These shared pages are marked as copy-on-write pages, meaning that if either process writes to a shared page, a copy of the shared page is created. Copy-on-write is illustrated in Figures which show the contents of the physical memory before and after process 1 modifies page C.

For example, assume that the child process attempts to modify a page containing portions of the stack, with the pages set to be copy-on-write. The operating system will then create a copy of this page, mapping it to the address space of the child process. The child process will then modify its copied page and not the page belonging to the parent process. Note, that only pages that can be modified need be marked as copy-onwrite. Pages that cannot be modified (pages containing executable code) can be shared by the parent and child.

Many operating systems provide a **pool** of free pages for such requests. Operating systems typically allocate these pages using a technique known as zero-fill-on-demand.

3. Page Replacement

While a user process is executing, a page fault occurs. The operating system finds that there are *no* free frames on the free-frame list; all memory is in use. Then we follow the most common solution: page replacement. Page replacement takes the following approach.

1. Find the location of the desired page on the disk.
2. Find a free frame:
 - a. If there is a free frame, use it.
 - b. If there is no free frame, use a page-replacement algorithm to select victim frame.
 - c. Write the victim frame to the disk; change the page and frame tables accordingly.
3. Read the desired page into the newly freed frame; change the page and frame tables.
- 4. Restart the user process.**

Page Replacement

4. Page Replacement Algorithms

If we have multiple processes in memory, we must decide how many frames to allocate to each process. Further, when page replacement is required, we must select the frames that are to be replaced. There are many different page-replacement algorithms. We want the one with the lowest page-fault rate. We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults. The string of memory references is called a **reference string**. the number of frames available increases, the

number of page faults decreases. We illustrate several page-replacement algorithms. We use the reference string **7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1** for a memory with three frames.

FIFO Page Replacement

It is the simplest page-replacement algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory, we insert it at the tail of the queue.

For our example reference string, our three frames are initially empty. The first three references (7,0,1) cause page faults and are brought into these empty frames. The next reference (2) replaces page 7, because page 7 was brought in first. Since 0 is the next reference and 0 is already in memory, we have no fault for this reference. The first reference to 3 results in replacement of page 0, since it is now first in line. Because of this replacement, the next reference, to 0, will fault. Page 1 is then replaced by page 0. This process continues as shown in Figure. Every time a fault occurs, we show which pages are in our three frames. There are 15 faults altogether.

The FIFO page-replacement algorithm is easy to understand and program. But if we select for replacement a page that is in active use, everything still works correctly. After we replace an active page with a new one, a fault occurs almost immediately to retrieve the active page. Some other page will need to be replaced to bring the active page back into memory. Thus, a bad replacement choice increases the page-fault rate and slows process execution. To illustrate the problems, we consider the following reference string:

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

Notice that the number of faults for four frames (ten) is *greater* than the number of faults for three frames (nine)! This most unexpected result is known as Belady's **anomaly**: For some page-replacement algorithms, the page-fault rate may *increase* as the number of allocated frames increases.

Optimal Page Replacement

An optimal page-replacement algorithm has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly. Such an algorithm does exist and has been called OPT or MIN. It is simply this:

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

Use of this page-replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames. For example, on our sample reference string, the optimal page-replacement algorithm would yield nine page faults, as shown in Figure.

The first three references cause faults that fill the three empty frames. The reference to page 2 replaces page 7, because 7 will not be used until reference 18, whereas page 0 will be used at 5, and page 1 at 14. The reference to page 3 replaces page 1, as page 1 will be the last of the three pages in memory to be referenced again. With only nine page faults, optimal replacement is much better than a FIFO algorithm, which resulted in fifteen faults.

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

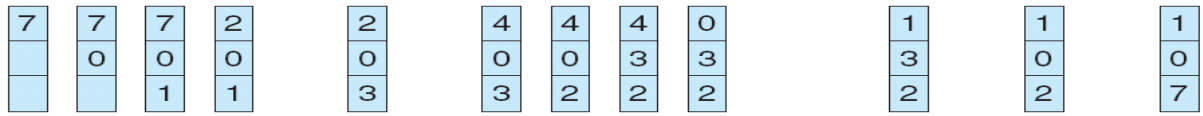
LRU Page Replacement

This approach is the least-recently-used (LRU) algorithm. In this, it uses the recent past as an approximation of the near future, then we can replace the page that *has not been used* for the longest

period of time. LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time.

reference string

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



page frames

The result of applying LRU replacement to our example reference string is shown in Figure. The LRU algorithm produces 12 faults. Notice that the first 5 faults are the same as those for optimal replacement. When the reference to page 4 occurs, however, LRU replacement sees that, of the three frames in memory, page 2 was used least recently. Thus, the LRU algorithm replaces page 2, not knowing that page 2 is about to be used. When it then faults for page 2, the LRU algorithm replaces page 3, since it is now the least recently used of the three pages in memory. Despite these problems, LRU replacement with 12 faults is much better than FIFO replacement with 15.

The major problem is *how* to implement LRU replacement. An LRU page-replacement algorithm may require substantial hardware assistance. The problem is to determine an order for the frames defined by the time of last use. Two implementations are feasible:

- **Counters.** In this, we associate with each page-table entry a time-of-use field and add to the CPU a logical clock or counter. The clock is incremented for every memory reference. Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page. In this way, we always have the "time" of the last reference to each page. We replace the page with the smallest time value.
- **Stack.** Another approach to implementing LRU replacement is to keep a stack of page numbers. Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the most recently used page is always at the top of the stack and the least recently used page is always at the bottom.

Like optimal replacement, LRU replacement does not suffer from Belady's anomaly.

Additional-Reference-Bits Algorithm

We can gain additional ordering information by recording the reference bits at regular intervals. We can keep an 8-bit byte for each page in a table in memory. At regular intervals (say, every 100 milliseconds), a timer interrupt transfers control to the operating system. The operating system shifts the reference bit for each page into the high-order bit of its 8-bit byte. These 8-bit shift registers contain the history of page use for the last eight time periods. If the shift register contains 00000000, for example, then the page has not been used for eight time periods; a page that is used at least once in each period has a shift register value of 11111111. A page with a history register value of 11000100 has been used more recently than one with a value of 01110111.

The number of bits of history can be varied. In the extreme case, the number can be reduced to zero, leaving only the reference bit itself. This algorithm is called the second-chance page replacement algorithm.

Second-Chance Algorithm

The basic algorithm of second-chance replacement is a FIFO replacement algorithm. When a page has been selected, however, we inspect its reference bit. If the value is 0, we proceed to replace this page; but if the reference bit is set to 1, we give the page a second chance and move on to select the next FIFO page. When a page gets a second chance, its reference bit is cleared, and its arrival time is reset to the current time. Thus, a page that is given a second chance will not be replaced until all other pages have been replaced

One way to implement the second-chance algorithm is as a circular queue. A pointer indicates which page is to be replaced next. When a frame is needed, the pointer advances until it finds a page with a 0 reference bit. Once a victim page is found, the page is replaced, and the new page is inserted in the circular queue in that position.

Second-Chance (clock) Page-Replacement Algorithm

Enhanced Second-Chance Algorithm

We can enhance the second-chance algorithm by considering the reference bit and the modify bit as an ordered pair. With these two bits, we have the following four possible classes:

1. (0, 0) neither recently used nor modified—best page to replace
2. (0, 1) not recently used but modified—not quite as good, because the page will need to be written out before replacement
3. (1, 0) recently used but clean—probably will be used again soon
4. (1,1) recently used and modified—probably will be used again soon, and the page will be need to be written out to disk before it can be replaced

Each page is in one of these four classes. When page replacement is called for, we use the same scheme as in the clock algorithm; but instead of examining whether the page to which we are pointing has the reference bit set to 1, we examine the class to which that page belongs. We replace the first page encountered in the lowest nonempty class. Notice that we may have to scan the circular queue several times before we find a page to be replaced.

Counting-Based Page Replacement

There are many other algorithms that can be used for page replacement. For example, we can keep a counter of the number of references that have been made to each page and develop the following two schemes.

- The **least frequently used (LFU) page-replacement algorithm** requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count.
- The **most frequently used (MFU) page-replacement algorithm** is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

As you might expect, neither MFU nor LFU replacement is common. The implementation of these algorithms is expensive.

5. Allocation of Frames

We discuss the issue of allocate the fixed amount of free memory among the various processes. The minimum number of frames per process is defined by the architecture, the maximum number is defined by the amount of available physical memory. There are two allocation algorithms used.

Allocation Algorithms

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

An alternative is to recognize that various processes will need differing amounts of memory. We can use **proportional** allocation, in which we allocate available memory to each process according to its size. Let the size of the virtual memory for process p_i be s_i and define

$$S = \sum s_i$$

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

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

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

In this way, both processes share the available frames according to their "needs," rather than equally. In both equal and proportional allocation, of course, the allocation may vary according to the multiprogramming level. If the multiprogramming level is increased, each process will lose some frames to provide the memory needed for the new process. Conversely, if the multiprogramming level decreases, the frames that were allocated to the departed process can be spread over the remaining processes.

In either equal or proportional allocation, a high-priority process is treated the same as a low-priority process. By its definition, however, we may want to give the high-priority process more memory to speed its execution, to the detriment of low-priority processes. One solution is to use a proportional allocation scheme wherein the ratio of frames depends not on the relative sizes of processes but rather on the priorities of processes or on a combination of size and priority.

Global versus Local Allocation

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

Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame is currently allocated to some other process; that is, one process can take a frame from another. Local replacement requires that each process select from only its own set of allocated frames.

A process can select a replacement from among its own frames or the frames of any lower-priority process. This approach allows a high-priority process to increase its frame allocation at the expense of a low-priority process.

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

One problem with a global replacement algorithm is that a process cannot control its own page-fault rate. The set of pages in memory for a process depends not only on the paging behavior of that process but also on the paging behavior of other processes. Therefore, the same process may perform quite differently.

Under local replacement, the set of pages in memory for a process is affected by the paging behavior of only that process. Local replacement might hinder a process, however, by not making available to it other, less used pages of memory.

Thus, global replacement generally results in greater system throughput and is therefore the more common method.

6. Thrashing

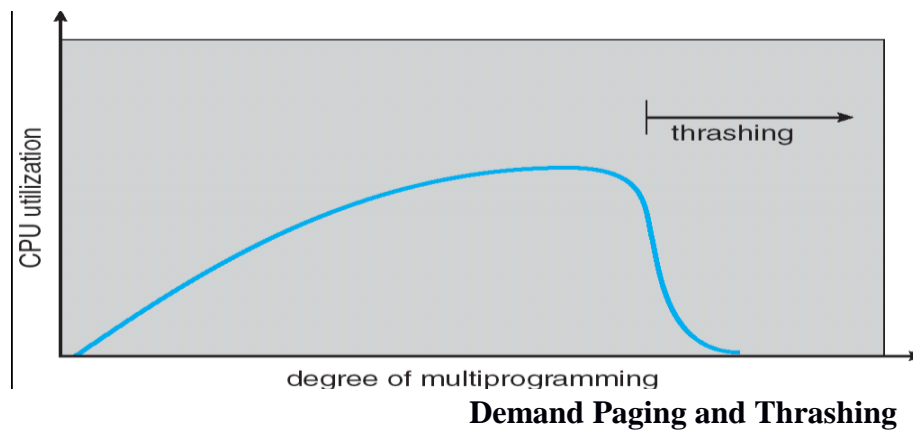
If the number of frames allocated to a low-priority process falls below the minimum number required by the computer architecture, we must suspend that process's execution. We should then page out its remaining pages, freeing all its allocated frames. This provision introduces a swap-in, swap-out level of intermediate CPU scheduling.

If the process does not have the number of frames it needs to support pages in active use, it will quickly page-fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately.

This high paging activity is called **thrashing**. A process is thrashing if it is spending more time paging than executing.

Cause of Thrashing

Thrashing results in severe performance problems. If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system. A global page-replacement algorithm is used; it replaces pages without regard to the process to which they belong. Now suppose that a process enters a new phase in its execution and needs more frames. It starts faulting and taking frames away from other processes. These processes need those pages, however, and so they also fault, taking frames from other processes. These faulting processes must use the paging device to swap pages in and out. As they queue up for the paging device, the ready queue empties. As processes wait for the paging device, CPU utilization decreases.



This phenomenon is illustrated in Figure in which CPU utilization is plotted against the degree of multiprogramming. As the degree of multiprogramming increases, CPU utilization also increases, until a maximum is reached. If the degree of multiprogramming is increased even further, thrashing sets in, and CPU utilization drops sharply. At this point, to increase CPU utilization and stop thrashing, we must *decrease* the degree of multiprogramming.

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

To prevent thrashing, we must provide a process with as many frames as it needs.