**Virtual Memory**

Virtual memory is a computing technique enabling the execution of processes that aren't entirely stored in physical memory. A significant advantage of this approach is the ability to accommodate programs larger than the available physical memory capacity.

**Background**

Memory-management algorithms are essential because instructions must reside in physical memory for execution. One approach to meeting this requirement involves placing the entire logical address space in physical memory, although dynamic linking can alleviate this but often requires extra effort by programmers. However, this necessity limits program size to physical memory capacity, which is unfortunate as many programs do not require their entire codebase simultaneously.

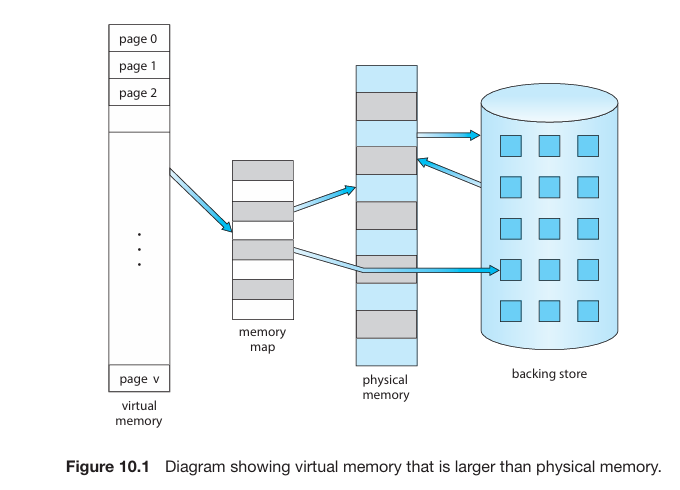
Examples include error-handling code rarely executed, inefficient memory allocation for arrays and lists, and unused program features. Thus, optimizing memory usage can enhance system efficiency by only loading necessary code and data into memory when needed.

Executing programs that are only partially in memory offers several advantages:

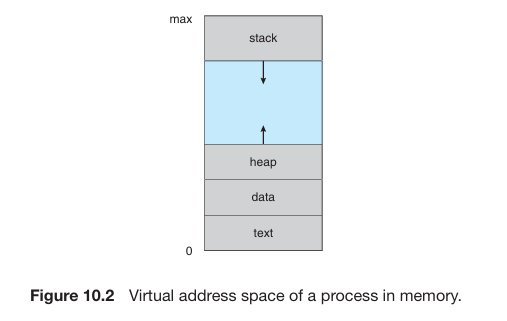
1. **Increased Virtual Address Space**: Programs would not be restricted by the amount of physical memory available, allowing users to develop applications for larger virtual address spaces. This simplifies programming tasks.
2. **Enhanced System Performance**: Since each program could consume less physical memory, more programs could run simultaneously, leading to higher CPU utilization and throughput without increasing response time or turnaround time.
3. **Improved Efficiency**: With less need for input/output operations to load or swap portions of programs into memory, each program would run faster, contributing to overall system efficiency.

Overall, executing programs partially in memory benefits both the system and its users by allowing for greater flexibility, improved performance, and efficient resource utilization.

**Virtual** **memory** involves the separating of logical memory, as perceived by developers, from physical memory. This allows for the provision of a much larger virtual memory space than the available physical memory. Virtual memory simplifies programming by relieving programmers from concerns about physical memory constraints, enabling them to focus on solving the problem at hand.

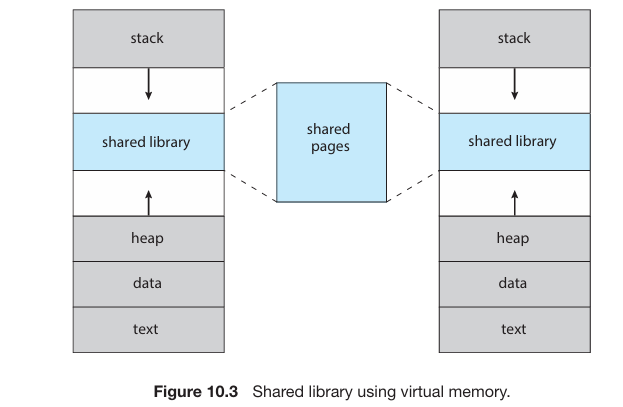


The **virtual address space** of a process represents the logical layout of the process in memory, typically starting at address 0 and appearing as contiguous memory. However, physical memory is organized into page frames, which may not be contiguous. The *memory management unit* (MMU) maps logical pages to physical page frames.



In the virtual address space, the heap grows upward in memory as dynamic memory allocation is used, while the stack grows downward through successive function calls. The space between the heap and stack, termed a *hole*, is part of the virtual address space but only requires physical pages if the heap or stack expands. Such virtual address spaces with holes are called **sparse address spaces**.

Using sparse address spaces is advantageous because holes can be filled dynamically as the stack or heap grows, or when dynamically linking libraries or shared objects during program execution.



Virtual memory not only separates logical memory from physical memory but also facilitates sharing files and memory among multiple processes through page sharing. This yields several benefits:

1. **Sharing System Libraries**: Virtual memory allows multiple processes to share system libraries, like the standard C library, by mapping the shared object into their virtual address spaces. Although each process perceives the library as part of its virtual address space, the actual physical pages where the libraries reside are shared by all processes.
2. **Shared Memory**: Processes can share memory regions using virtual memory. A process can create a region of memory that it shares with another process. Each process considers this shared region part of its virtual address space, but the physical pages of memory are shared between them.
3. **Process Creation Efficiency**: Pages can be shared during process creation using the fork () system call, which accelerates process creation by avoiding the need to duplicate memory pages.

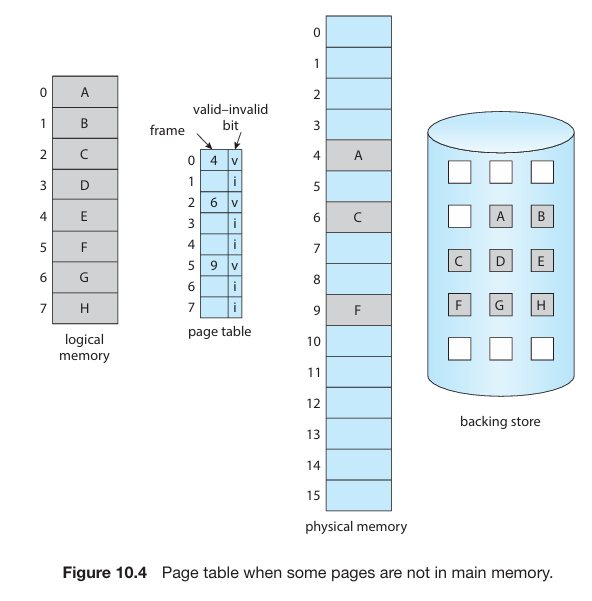
**Demand Paging**

When loading an executable program from secondary storage into memory, one option is to load the entire program into physical memory at program execution time. However, this approach may result in loading unnecessary parts of the program into memory, leading to inefficiency, especially if not all parts of the program are immediately needed.

An alternative strategy is **demand** **paging**, commonly used in virtual memory systems. With demand paging, pages of the program are loaded into memory only when they are needed during program execution. This means that pages that are never accessed are not loaded into physical memory, thus conserving memory resources. Demand paging is similar to a paging system with swapping, where processes reside in secondary memory. It highlights one of the primary benefits of virtual memory: by loading only the necessary portions of programs into memory, memory utilization is more efficient.

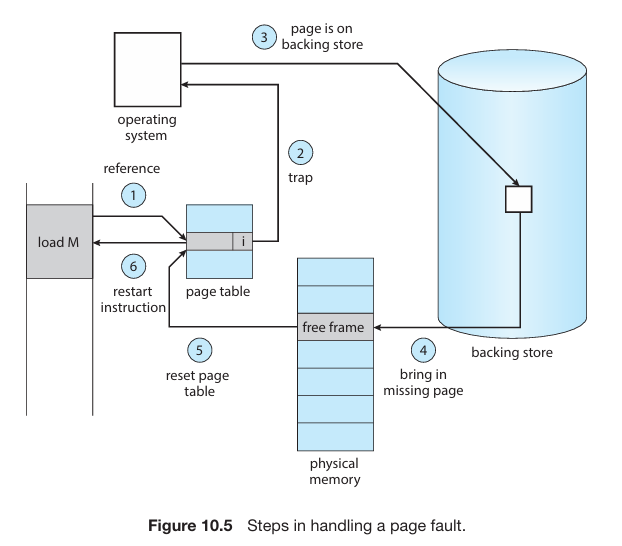
**Basic Concept**

Demand paging involves loading a page into memory only when it's needed, resulting in some pages being in memory while others reside in secondary storage during process execution. Hardware support is required to differentiate between pages in memory and those in secondary storage. The valid-invalid bit scheme described in Section 9.3.3 is commonly used for this purpose.



In demand paging, when the valid bit is set to "valid," the associated page is both legal and in memory. If the bit is set to "invalid," it indicates that the page is either not valid (not in the logical address space of the process) or valid but currently in secondary storage. When a page is brought into memory, its page-table entry is set as usual. However, if a page is not currently in memory, its page-table entry is marked as invalid. This marking has no effect unless the process attempts to access that page.

When a process attempts to access a page that has not been brought into memory, it triggers a **page** **fault**. The paging hardware, while translating the address through the page table, detects that the invalid bit is set for the requested page. This detection results in a trap being generated to the operating system. The page fault trap signifies the operating system's inability to bring the desired page into memory.



Steps for handling page fault are as follows:

1. The system checks an internal table associated with the process control block to determine whether a memory access is valid or invalid.

2. If the memory access is invalid, the process is terminated. If the access is valid but the required page is not yet in memory, the system proceeds to page it in.

3. A free frame is located, often by taking one from the free-frame list.

4. A secondary storage operation is scheduled to read the desired page into the newly allocated frame.

5. Once the storage read is complete, the system updates the internal table kept with the process and the page table to indicate that the page is now in memory.

6. The instruction that was interrupted by the trap is restarted, allowing the process to access the page as if it had always been in memory.

In pure demand paging, a process can start execution with no pages in memory. When the operating system sets the instruction pointer to the first instruction of the process, which may reside on a non-memory resident page, the process immediately faults for the page. The operating system then brings this page into memory, allowing the process to continue execution.

The process continues to execute, faulting as necessary, until every page it needs is in memory. Once all required pages are in memory, the process can execute without further faults. This scheme ensures that pages are brought into memory only when they are required, exemplifying the concept of demand paging.

Theoretical concerns suggest that some programs could potentially trigger multiple page faults per instruction execution, as they may require access to several new pages of memory for both instructions and data. However, empirical analysis of running processes indicates that this scenario is highly unlikely. Most programs exhibit locality of reference, meaning they tend to access memory locations that are close to each other, resulting in reasonable performance with demand paging.

The hardware required to support demand paging is similar to that for paging and swapping:

* **Page table**: This table includes features such as a valid–invalid bit or special protection bits to mark entries as invalid.
* **Secondary memory**: Pages not currently in main memory are stored in secondary memory, typically a high-speed disk or non-volatile memory (NVM) device. This secondary storage is known as the swap device, and the dedicated section for this purpose is referred to as swap space.

A crucial requirement for demand paging is the ability to restart any instruction after a page fault. This necessitates saving the state of the interrupted process when the fault occurs, allowing the process to resume execution from the exact same place and state once the desired page is brought into memory.

In most cases, meeting this requirement is straightforward. If a page fault occurs during instruction fetch, the instruction can simply be fetched again upon restart. Similarly, if a fault occurs while fetching an operand, the instruction needs to be fetched, decoded, and operands fetched again.

In a worst-case scenario, such as with a three-address instruction like "ADD the content of A to B, placing the result in C", multiple steps are involved in executing the instruction. If a page fault occurs while storing the result in C (due to C being in a page not currently in memory), the process must fetch the desired page, update the page table, and restart the instruction. This involves re-fetching the instruction, decoding it, fetching the operands again, and performing the operation again. However, the repeated work is minimal (less than one complete instruction), and it is necessary only when a page fault occurs.

The major difficulty in demand paging arises when one instruction may modify multiple locations, particularly when dealing with instructions like the IBM System 360/370 MVC (move character) instruction, which can move up to 256 bytes from one location to another, possibly overlapping.

If either the source or destination block of such an instruction straddles a page boundary, a page fault might occur after the move is partially completed. Moreover, if the source and destination blocks overlap, and the source block has been modified, simply restarting the instruction becomes problematic.

To address this issue, two solutions can be employed:

1. **Pre-computation and Access**: In this solution, the microcode computes and attempts to access both ends of both blocks before any modification are made. If a page fault is going to occur, it will happen at this step, before anything is modified. Once it's confirmed that all relevant pages are in memory, the move operation can proceed without fear of page faults.

2. **Temporary Registers**: Alternatively, temporary registers can be used to hold the values of overwritten locations. If a page fault occurs during the instruction execution, all the old values are written back into memory before the trap occurs. This action effectively restores memory to its state before the instruction was started, allowing the instruction to be repeated without adverse effects.

**Free Frame List**

When a page fault occurs, the operating system is tasked with bringing the desired page from secondary storage into main memory. To handle page faults efficiently, most operating systems maintain a free-frame list, which is a pool of available frames that can be allocated to satisfy such requests. These free frames are also utilized when stack or heap segments of a process expand.



Operating systems typically employ a technique called "zero-fill-on-demand" to allocate free frames. This involves zeroing out the contents of a frame before allocating it, erasing any previous data. This practice is crucial for security reasons, as it prevents sensitive information from leaking if a frame is reassigned without being cleared.

During system startup, all available memory is placed on the free-frame list. As free frames are requested, such as during demand paging, the size of the free-frame list decreases. Eventually, the list may reach zero or fall below a certain threshold, indicating a need to replenish the free-frame list by reclaiming memory from processes or other sources.

**Performance of Demand Paging**

Demand paging can significantly affect the performance of a computer system. The effective access time for a demand-paged memory system can be computed by considering the memory access time (ma), typically around 10 nanoseconds. When there are no page faults, the effective access time equals the memory access time. However, if a page fault occurs, the system must retrieve the relevant page from secondary storage before accessing the desired word.

Let p represent the probability of a page fault (0 ≤ p ≤ 1). Normally, p is expected to be close to zero, indicating only a few page faults. The effective access time in such a scenario can be calculated as:

Effective access time = (1 − p) × ma + p × page fault time.

Servicing a page fault involves a sequence of steps that occur when a page referenced by a process is not in memory and needs to be brought in from secondary storage. The steps involved in handling a page fault are as follows:

1. Trap to the operating system

2. Save the registers and process state

3. Determine that the interrupt was a page fault

4. Check if the page reference was legal and locate the page in secondary storage

5. Issue a read from the storage to a free frame

a. Wait in a queue until the read request is serviced

b. Wait for the device seek and/or latency time

c. Begin the transfer of the page to a free frame

6. While waiting, allocate the CPU core to some other process

7. Receive an interrupt from the storage I/O subsystem (I/O completed)

8. Save the registers and process state for the other process (if step 6 is executed)

9. Determine that the interrupt was from the secondary storage device

10. Correct the page table and other tables to show that the desired page is now in memory

11. Wait for the CPU core to be allocated to this process again

12. Restore the registers, process state, and new page table, and then resume the interrupted instruction.

Not all steps listed are necessary in every case. Step 6, for example, assumes that the CPU is allocated to another process during I/O operations, which can increase the time to resume the page-fault service routine once the I/O transfer is complete but will allow multiprogramming to maintain CPU utilization.

There are three major task components in the page-fault service time:

1. Servicing the page-fault interrupt.

2. Reading in the page.

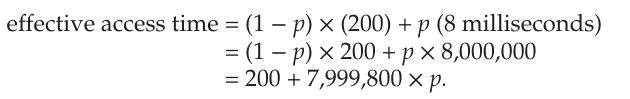
3. Restarting the process.

The first and third tasks can be reduced to several hundred instructions each, with careful coding, and may take from 1 to 100 microseconds each.

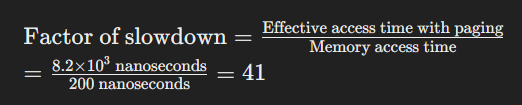
In the case of using HDDs as the paging device:

* The page switch time is likely close to 8 milliseconds.
* This includes hardware and software time, considering an average hard disk latency of 3 milliseconds, seek time of 5 milliseconds, and transfer time of 0.05 milliseconds.
* Additionally, queuing time may need to be considered if a queue of processes is waiting for the device, further increasing the time to page in.

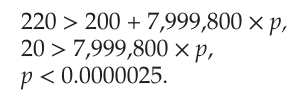
Given an average page-fault service time of 8 milliseconds and a memory access time of 200 nanoseconds, the effective access time can be calculated as follows:



The effective access time is directly proportional to the **page-fault rate**. For example, if one access out of 1,000 causes a page fault, the effective access time becomes 8.2 microseconds, resulting in a slowdown by a factor of 40 (approximated) due to demand paging.



To keep the performance degradation to less than 10%, the probability of page faults (p) should be kept below 0.0000025. In other words, to maintain reasonable performance, less than one memory access out of 399,990 should result in a page fault.



In conclusion, it's crucial to keep the page-fault rate low in a demand-paging system to prevent a significant increase in effective access time, which can dramatically slow down process execution.

1 second (s) = 1,000 milliseconds (ms) = 1,000,000 microseconds (µs) = 1,000,000,000 nanoseconds (ns)

So, when converting between these units: To convert from a larger unit to a smaller unit (e.g., seconds to milliseconds), you multiply by 1,000. To convert from a smaller unit to a larger unit (e.g., nanoseconds to microseconds), you divide by 1,000.

Demand paging involves efficiently managing swap space, which is an essential aspect of virtual memory systems. Here are the key points summarized:

1. **Swap Space Utilization**:

* I/O to swap space is generally faster than to the file system due to larger block allocation and simpler allocation methods.
* One approach to enhance paging throughput is to copy an entire file image into swap space at process startup, allowing subsequent demand paging from swap space. However, this method has the disadvantage of initial image copying.

2. **Alternative Approaches**:

* Another approach, used by operating systems like Linux and Windows, is to initially demand-page from the file system and write replaced pages to swap space. This ensures only necessary pages are read from the file system, with subsequent paging from swap space.
* Some systems limit swap space usage by demand paging binary executable files directly from the file system. When page replacement occurs, these frames can be overwritten, and pages can be read from the file system again if needed. This approach uses the file system itself as the backing store.

3. **Anonymous Memory and Compromise**:

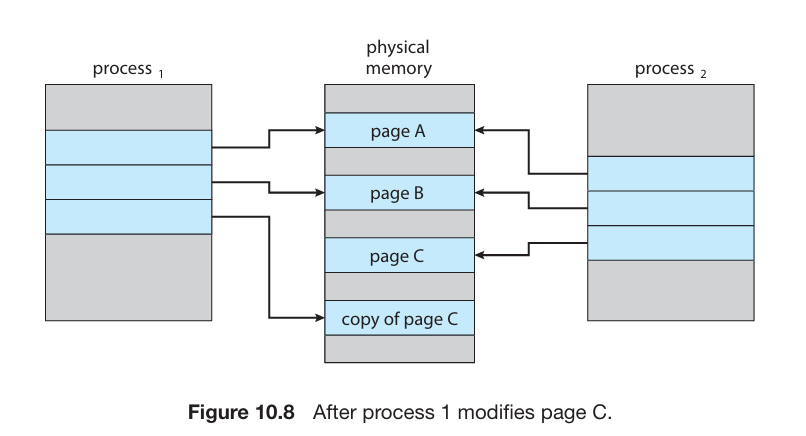
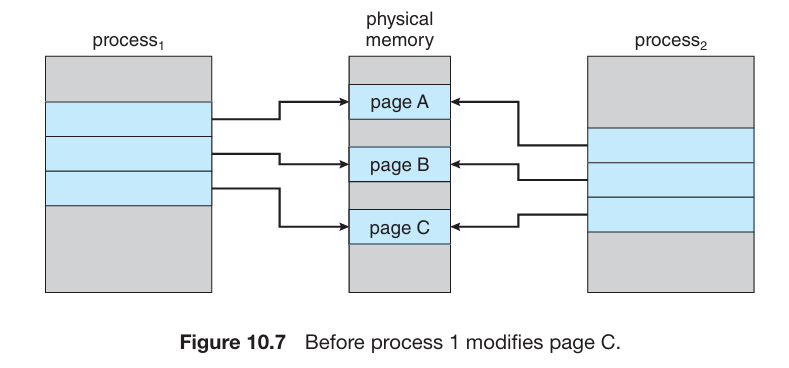
* Swap space is still used for pages not associated with a file, such as the stack and heap for a process (anonymous memory).
* This method serves as a compromise between demand paging from the file system and using swap space effectively. It's employed by systems like Linux and BSD UNIX, utilizing the file system as the primary backing store and swap space for anonymous memory pages.

**Copy on Write**

The traditional `fork () ` system call creates a child process by duplicating the parent's address space, which involves copying all pages belonging to the parent. However, considering that many child processes immediately invoke the `exec () ` system call, duplicating the address space may be unnecessary. Instead, a technique called copy-on-write is used; allowing the parent and child processes to initially share the same pages.

These shared pages are marked as copy-on write pages, meaning that if either process write to a shared page, a copy of the page is created. This technique ensures that only modified pages are copied, while unmodified pages are shared between the parent and child processes. Copy-on-write is commonly employed by operating systems like Windows, Linux, and macOS to optimize memory usage and process creation.

For example, if a child process attempts to modify a page containing portions of the stack with copy-on-write pages set, the operating system will obtain a frame from the free-frame list and create a copy of the page. This copy is then mapped to the address space of the child process. The child process can then modify its copied page without affecting the parent process's page. Only pages that are modified by either process are copied, while unmodified pages are shared between the parent and child.



**Page Replacement**

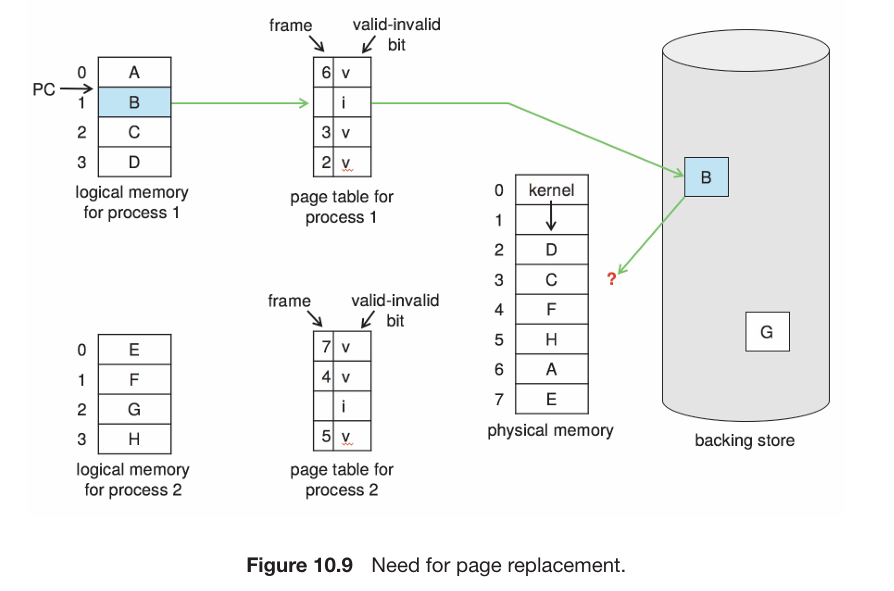
In our earlier discussion about page-fault rates, we assumed each page only faults once when it's first accessed. But sometimes, a process only uses half of its pages. Demand paging helps save resources by not loading unused pages. This means we can run more processes simultaneously, improving system efficiency.

For example, with forty memory frames, instead of running four processes needing ten frames each (some unused), we could run eight processes, doubling our multiprogramming efficiency.

However, increasing multiprogramming means allocating more memory. If we run six processes, each supposed to use ten pages but actually using only five, we have spare frames. This boosts CPU utilization and throughput but risks running out of memory if all processes suddenly need all their pages.

Consider that system memory isn't solely used for holding program pages; a significant portion is also consumed by I/O buffers. This can complicate memory-placement algorithms. Deciding the balance between allocating memory to program pages and I/O buffers is a tough challenge. Some systems allocate a fixed percentage of memory for I/O buffers, while others let both processes and the I/O subsystem compete for system memory.

Over-allocation of memory becomes apparent when a process is running, and a page fault happens. The operating system locates the required page on secondary storage but realizes there are no available frames on the free-frame list; all memory is already in use. This scenario is illustrated in Figure 10.9, where the absence of free frames is depicted by a question mark.

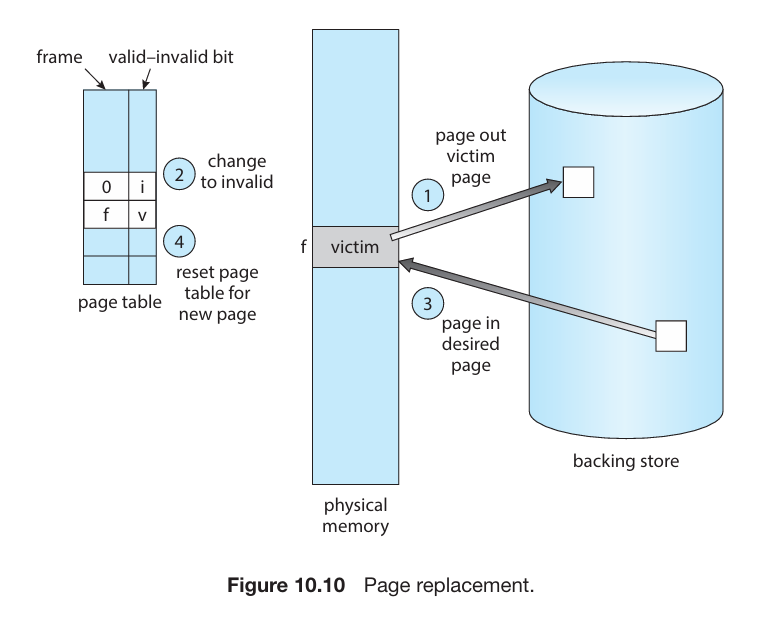


The operating system has a few options in this situation. It could choose to terminate the process, but this contradicts the aim of demand paging, which seeks to enhance system utilization and throughput. Ideally, users shouldn't be aware that their processes are running on a paged system; paging should be transparent to them. Thus, termination isn't the preferred choice.

Alternatively, the operating system could resort to standard swapping, swapping out a process entirely to free up its frames and reduce the level of multiprogramming. However, most modern operating systems have moved away from standard swapping due to the overhead of copying entire processes between memory and swap space. Instead, they typically combine swapping pages with **page** **replacement** algorithms.

**Basic Page Replacement**

Page replacement follows a specific approach: when no frame is free, it looks for one that is not currently in use and frees it up. This is achieved by writing its contents to swap space and updating the page table (along with other related tables) to indicate that the page is no longer in memory (as shown in Figure 10.10). The freed frame can then be utilized to hold the page for which the process experienced a fault.



We enhance the page-fault service routine to incorporate page replacement as follows:

1. Locate the desired page on secondary storage.

2. Find a free frame:

a. If there's a free frame, use it.

b. If no free frame is available, employ a page-replacement algorithm to choose a victim frame.

c. Write the contents of the victim frame to secondary storage if necessary and update the page and frame tables accordingly.

3. Read the desired page into the newly freed frame and update the page and frame tables.

4. Resume the process from where the page fault occurred.

It's worth noting that if no frames are free, two page transfers are necessary: one for swapping out the victim page and another for bringing in the desired page. This effectively doubles the page-fault service time and consequently increases the effective access time accordingly.

We can mitigate this overhead by employing a **modify** bit, also known as a *dirty* bit. In this scheme, each page or frame in hardware is associated with a modify bit. This bit is set by the hardware whenever any byte in the page is written into, indicating that the page has been altered.

When selecting a page for replacement, we inspect it’s modify bit. If the bit is set, it signifies that the page has been modified since it was loaded from secondary storage. In such cases, we need to write the page back to storage. Conversely, if the modify bit is not set, it indicates that the page hasn't been modified since it was loaded into memory. In this scenario, we can skip writing the memory page to storage since it's already present there. This approach is also applicable to read-only pages, such as pages containing binary code. Since these pages cannot be altered, they can be discarded as needed.

By utilizing this scheme, we can notably reduce the time required to service a page fault. Specifically, it reduces I/O time by half if the page hasn't been modified.

Demand paging, facilitated by page replacement, plays a crucial role in separating logical memory from physical memory. This enables the provision of a vast virtual memory for programs, even with limited physical memory. Without demand paging, logical addresses directly map to physical addresses, with all process pages residing in physical memory. However, demand paging frees logical address space from physical memory constraints. For example, a process with twenty pages can execute using just ten frames via demand paging. If a modified page needs replacement, its content is moved to secondary storage. When later accessed, a page fault occurs, bringing the page back into memory, possibly replacing another page in the process.

To implement demand paging, we need to tackle two significant challenges: developing a frame-allocation algorithm and a page-replacement algorithm.

1. **Frame-allocation algorithm**: This involves deciding how many memory frames to allocate to each process when multiple processes are in memory simultaneously.

2. **Page-replacement algorithm**: When page replacement is necessary due to memory constraints, we must select which frames to replace.

Designing effective algorithms to address these challenges is crucial because I/O operations with secondary storage are costly. Even small enhancements in demand-paging methods can result in substantial improvements in system performance.

When selecting a page-replacement algorithm, the goal is to choose one with the lowest page-fault rate. We evaluate algorithms by running them on a **reference** **string**—a sequence of memory references. This string can be artificially generated using a random-number generator or traced from an actual system, recording the address of each memory reference. Tracing a system generates a vast amount of data, so we can reduce it by considering only the page number instead of the entire address. Additionally, if there's a reference to a page 'p', subsequent references to 'p' won't cause a page fault because 'p' will already be in memory.

For example, given the address sequence:

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,

0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105

With a page size of 100 bytes, this sequence can be reduced to:

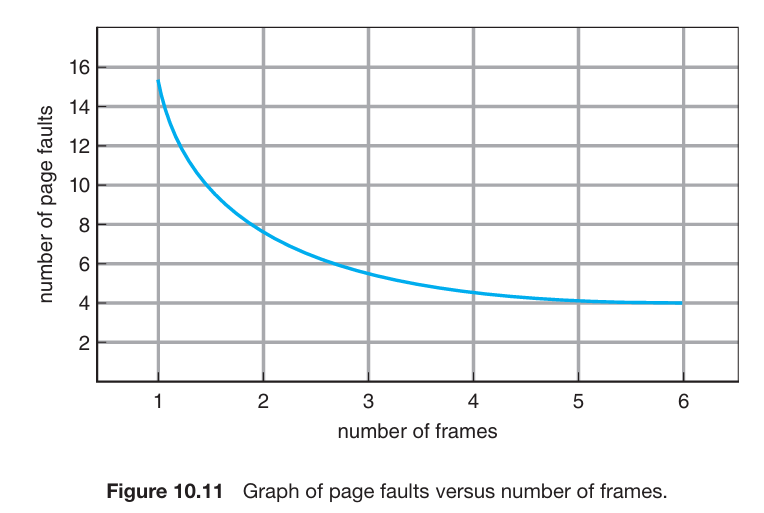
1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

This simplified reference string allows us to evaluate page-replacement algorithms effectively.

To determine the number of page faults for a specific reference string and page-replacement algorithm, we also need to know the number of page frames available. Clearly, as the number of frames increases, the number of page faults decreases.

For instance, using the previous reference string, if we had three or more frames available, we would only encounter three faults—one for the initial reference to each page. Conversely, with only one frame available, we would experience a replacement with every reference, resulting in eleven faults.

In general, we expect a curve like the one shown in Figure 10.11. As the number of frames increases, the number of page faults decreases to a certain minimal level. Adding physical memory increases the number of frames, hence improving system performance.



**FIFO Page Replacement**

The FIFO (First-In, First-Out) page-replacement algorithm is straightforward: it associates with each page the time it was brought into memory, and when a replacement is needed, the oldest page in memory is chosen. Instead of recording the time when a page is brought in, we can utilize a FIFO queue to hold all pages in memory. The page at the head of the queue is replaced, and when a new page is brought into memory, it's inserted at the tail of the queue.

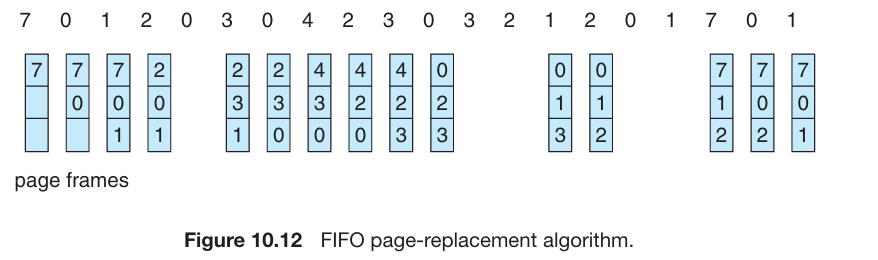
While the FIFO algorithm is easy to understand and implement, its performance isn't always optimal. The page replaced might be an initialization module no longer needed or a heavily used variable initialized early and constantly in use.

Even if a page selected for replacement is actively used, the system still functions correctly. After replacing an active page with a new one, a fault occurs almost immediately to retrieve the active page, necessitating another page replacement. Thus, a poor replacement choice increases the page-fault rate and slows process execution, but it doesn't cause incorrect execution.

Consider the reference string:

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

With the FIFO scheme, we will have 15 page faults:



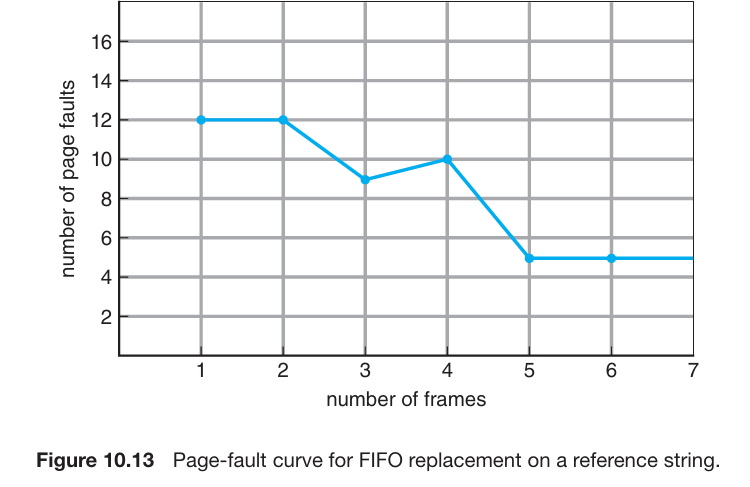


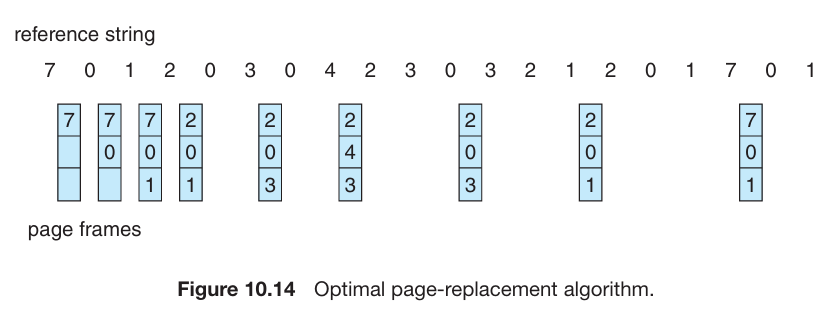
Figure 10.13 illustrates the curve of page faults for a specific reference string plotted against the number of available frames. Interestingly, it's observed that the number of faults for four frames (ten) is higher than the number for three frames (nine). This surprising outcome is termed **Belady’s** **anomaly**: for certain page-replacement algorithms, the page-fault rate might actually increase as the number of allocated frames increases.

**Optimal Page Replacement**

Belady’s anomaly led to the quest for an optimal page-replacement algorithm—one with the lowest page-fault rate among all algorithms and immune to Belady’s anomaly. Such an algorithm does exist and is known as OPT or MIN. It's remarkably simple: replace the page that will not be used for the longest period of time.

Using the OPT (or MIN) page-replacement algorithm ensures the lowest achievable page-fault rate for a fixed number of frames.

Using the same reference string as before, we get fewer than nine page faults:



Regrettably, implementing the optimal page-replacement algorithm is challenging because it necessitates future knowledge of the reference string. Consequently, the optimal algorithm is primarily utilized for comparison studies. For example, it can be beneficial to ascertain that while a new algorithm isn't optimal, it's within 12.3 percent of optimal at its worst and within 4.7 percent on average.

**Least Recently Used (LRU) Page Replacement**

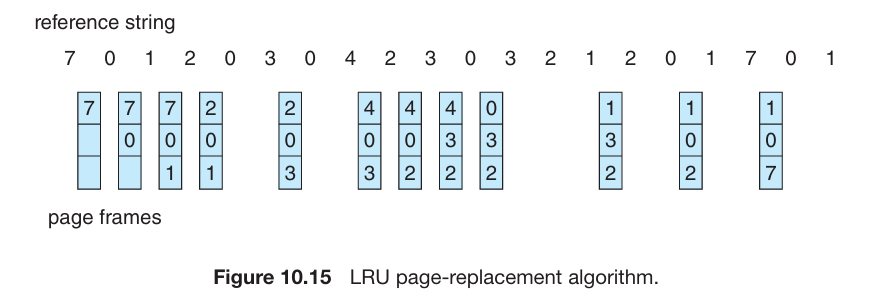
If implementing the optimal algorithm isn't feasible, perhaps an approximation is possible. The primary difference between the FIFO and OPT algorithms—aside from their forward or backward time perspective—is that FIFO relies on when a page was brought into memory, whereas OPT uses when a page is to be used.

By considering the recent past as an approximation of the near future, we can replace the page that hasn't been used for the longest period of time. This strategy is known as the **least recently used (LRU) algorithm**.

The LRU (Least Recently Used) replacement algorithm associates with each page the time of its last use. When a page needs replacement, LRU selects the page that hasn't been used for the longest period of time. This strategy can be seen as the optimal page-replacement algorithm looking backward in time instead of forward.

Interestingly, if we consider the reverse of a reference string 'S' as 'SR', the page-fault rate for the OPT algorithm on 'S' is the same as that for OPT on 'SR'. Similarly, the page-fault rate for the LRU algorithm on 'S' is identical to that for LRU on 'SR'.

Using the same reference string as before, we get 12 page faults using LRU:



The Least Recently Used (LRU) policy is commonly utilized as a page-replacement algorithm due to its effectiveness. However, implementing LRU replacement can pose significant challenges.

The main issue lies in determining the order of frames based on the time of last use. This task may require substantial hardware assistance to accurately track the usage history of each page in memory.

There are two ways to tackle this problem: Counters and stack.

To implement the Least Recently Used (LRU) page-replacement algorithm, we can utilize **counters**. In the simplest setup, each page-table entry is associated with a *time-of-use field*. Additionally, the CPU is equipped with a logical clock or *counter*, which is incremented for every memory reference.

Whenever a reference to a page occurs, the contents of the clock register are copied to the time-of-use field in the corresponding page-table entry. This way, we always have the "time" of the last reference to each page. Subsequently, we replace the page with the smallest time value, indicating it was least recently used.

However, this scheme necessitates searching the page table to find the LRU page and performing a write to memory for each memory access (to update the time-of-use field in the page table). Furthermore, the times must be maintained when page tables are altered due to CPU scheduling. Additionally, overflow of the clock must be taken into consideration.

Another approach to implementing LRU replacement is by maintaining a **stack** of page numbers. Whenever a page is referenced, it is removed from its current position in the *stack* and placed on the top. Consequently, the most recently used page resides at the **top** of the stack, while the least recently used page remains at the **bottom**.

To facilitate efficient removal and insertion of pages, it's best to implement this approach using a *doubly linked list* with a head pointer and a tail pointer. This allows for easy removal and insertion of pages from both ends of the stack. While each update may be slightly more costly due to the need to modify pointers, there's no need for a search for replacement—the tail pointer always points to the bottom of the stack, representing the LRU page.

This method is particularly suitable for software or microcode implementations of LRU replacement, where hardware assistance might be limited.

LRU (Least Recently Used) replacement, like optimal replacement, doesn't suffer from Belady’s anomaly. Both are part of a category of page-replacement algorithms known as stack algorithms, which are immune to Belady’s anomaly.

A stack algorithm ensures that the set of pages in memory for 'n' frames is always a subset of the set of pages that would be in memory with 'n+1' frames. For LRU replacement, this means that the set of pages in memory consists of the 'n' most recently referenced pages. If the number of frames is increased, these 'n' pages will still be the most recently referenced and will remain in memory.

Both implementations of LRU (Least Recently Used) replacement rely on hardware assistance beyond the standard Translation Look aside Buffer (TLB) registers. Updating the clock fields or stack for every memory reference is essential for these implementations.

If we were to use interrupts for every memory reference to allow software to update such data structures, it would significantly slow down the system. In fact, it could slow down every process by a factor of at least ten. This level of overhead for memory management would be intolerable for most systems.

**LRU Approximation Page Replacement**

Many computer systems lack sufficient hardware support for true LRU (Least Recently Used) page replacement. In fact, some systems provide no hardware support at all, requiring the use of alternative page-replacement algorithms such as FIFO. However, many systems offer some assistance in the form of a reference bit. This bit, associated with each entry in the page table, is set by the hardware whenever the corresponding page is referenced, either by a read or writes operation.

Initially, all reference bits are cleared (set to 0) by the operating system. As a process executes, the hardware sets the reference bit for each page that is referenced (set to 1). After some time, by examining these reference bits, we can determine which pages have been used and which have not, although we don't know the order of use. This information forms the basis for many page-replacement algorithms that approximate LRU replacement.

**Additional Reference Bits Algorithm**

To gain additional ordering information, we can record reference bits at regular intervals. For each page, we can maintain an 8-bit byte in memory, forming a table.

At fixed intervals, such as every 100 milliseconds, a timer interrupt triggers the operating system to shift the reference bit for each page into the high-order bit of its 8-bit byte. The other bits are shifted right by 1 bit, discarding the low-order bit. These 8-bit shift registers store the history of page use for the last eight time periods.

11000100 is 196 in decimal for page 1 whereas 01110111 is 119 for page 2. Hence page 2 is least recently used and will be replaced.

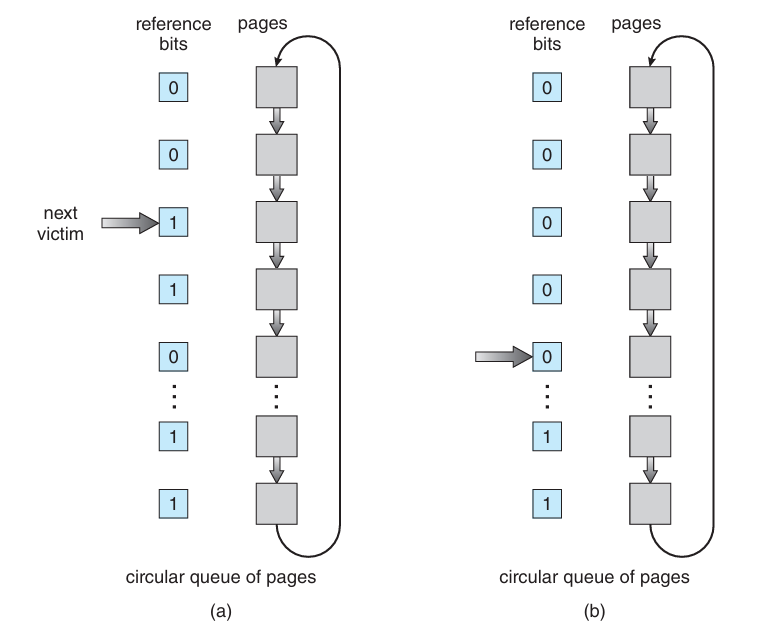
Interpreting these 8-bit bytes as unsigned integers, the page with the lowest number represents the LRU (Least Recently Used) page and can be replaced. However, the numbers are not guaranteed to be unique. We can replace all pages with the smallest value or use the FIFO method to select among them.

The number of bits of history included in the shift register can vary, depending on available hardware. It's chosen to optimize updating speed. In the extreme case, the number of bits can be reduced to zero, retaining only the reference bit itself. This algorithm is known as the *Second Chance page-replacement algorithm*.

**Second Chance Algorithm**

The basic algorithm of second-chance replacement is a variation of FIFO replacement. When selecting a page for replacement, we first inspect its reference bit. If the bit is 0, we proceed to replace the page. However, if the reference bit is set to 1, indicating recent use, we give the page a second chance and move on to the next page in the FIFO sequence.

When a page receives a second chance, its reference bit is cleared, and its arrival time is reset to the current time. Consequently, a page with a second chance won't be replaced until all other pages have been replaced or given second chances. Furthermore, if a page is used frequently enough to maintain its reference bit set, it will never be replaced.



One way to implement the second-chance algorithm, also known as the clock algorithm, is as a circular queue. A pointer, resembling a hand on a clock, indicates the next page for replacement. As the pointer advances, it clears reference bits until it finds a page with a reference bit of 0. Once a victim page is found, it's replaced, and the new page is inserted in the circular queue at that position.

In the worst-case scenario, when all bits are set, the pointer cycles through the entire queue, giving each page a second chance. It clears all reference bits before selecting the next page for replacement. If all bits are set, the second-chance replacement algorithm degenerates to FIFO replacement.

**Enhanced Second Chance Algorithm**

To enhance the second-chance algorithm, we can consider both the reference bit and the modify bit (described in Section 10.4.1) as an ordered pair, creating four possible classes:

1. (0, 0): Neither recently used nor modified—best page to replace.

2. (0, 1): Not recently used but modified—not as good, as 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 need to be written out to secondary storage before it can be replaced.

When page replacement is required, we follow the same scheme as in the clock algorithm. 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 then replace the first page encountered in the lowest non-empty class. Note that we may have to scan the circular queue several times before finding a page to be replaced.

The major difference between this algorithm and the simpler clock algorithm is that here we give preference to pages that have been modified in order to reduce the number of I/O required.