What is fragmentation and what are the differences between internal and external fragmentation?

Defining and Differentiating Fragmentation

Fragmentation occurs when memory or disk space is broken into smaller, unusable pieces. This can happen due to various memory management techniques used by the operating system. There are two main types of fragmentation: internal and external.

Internal fragmentation occurs when a process is allocated more memory than it needs. The extra space within the allocated unit is wasted, as it cannot be used by other processes. For example, if memory is allocated in fixed-sized blocks of 2,048 bytes, and a process needs 72,766 bytes, it will be allocated 36 blocks, resulting in 962 bytes of internal fragmentation. This type of fragmentation is common in systems that use fixed-sized allocation units, such as single-partition schemes and paging.

External fragmentation, on the other hand, happens when enough total memory space exists to satisfy a request, but the available spaces are not contiguous. The free memory is scattered throughout the system in small, unusable pieces. For example, imagine a system with 100 MB of free memory, but the largest contiguous block is only 20 MB. A process requesting 30 MB of memory would be unable to run, even though there is technically enough free space. This issue arises from the dynamic allocation and deallocation of variable-sized memory chunks, as seen in multi-partition schemes and segmentation.

Here's a table summarizing **three key differences between internal and external fragmentation:**

Feature	Internal Fragmentation	External Fragmentation
Cause	Allocating a fixed-size memory block larger than required	Scattering of free memory into small, non-contiguous pieces
Impact	Wasted space within allocated units	Wasted space between allocated units

Mitigation	Using smaller allocation units or variable-sized units	Compaction or using non-contiguous memory allocation schemes like paging and segmentation

Compare first-fit , worst fit and best-fit strategies for disk space allocation, with their advantages and disadvantages.

Disk Space Allocation Strategies: First-Fit, Worst-Fit, and Best-Fit

The sources discuss three strategies for allocating disk space to satisfy requests from processes: first-fit, worst-fit, and best-fit. All three strategies are used to select a free hole (a contiguous block of available space) from the set of free holes on the disk.

Here's a breakdown of each strategy with its advantages and disadvantages:

First-Fit

How it works: The first-fit strategy allocates the first hole that is large enough to satisfy the request. The search for a suitable hole can begin at the beginning of the set of holes or where the previous first-fit search left off.

Advantages:

- **Simplicity**: This algorithm is relatively simple to implement, making it computationally inexpensive.
- **Speed**: Since the search stops as soon as a sufficiently large hole is found, first-fit can be faster than other strategies, particularly if suitable holes are found early in the search.

Disadvantages:

- **External fragmentation**: First-fit can lead to external fragmentation as it may leave small, unusable holes between allocated blocks. Over time, these small holes can accumulate and reduce the overall efficiency of disk space utilization.

Best-Fit

How it works: The best-fit strategy allocates the smallest hole that is big enough to satisfy the request. This necessitates searching the entire list of free holes unless the list is ordered by size.

Advantages:

- **Minimizes leftover space**: Best-fit aims to minimize the size of the leftover hole, potentially reducing the amount of external fragmentation compared to first-fit.

Disadvantages:

- **Slower search**: Searching the entire list of free holes can take longer than first-fit, especially if the list is not sorted by size.
- **External fragmentation**: While best-fit may leave smaller leftover holes, it still contributes to external fragmentation.

Worst-Fit

How it works: The worst-fit strategy allocates the largest hole to satisfy the request. This also involves searching the entire list of free holes, unless sorted by size.

Advantages:

- **Larger leftover holes**: Worst-fit leaves the largest leftover holes, which might be more readily usable for subsequent requests compared to the smaller remnants from best-fit.

Disadvantages:

- **Inefficient use of space**: By allocating the largest holes first, worst-fit tends to quickly break down large free spaces into smaller ones, leading to poorer overall disk space utilization and increased external fragmentation.
- **Slower search**: Like best-fit, worst-fit generally requires searching the entire list of free holes.

General Observations and Recommendations

- Simulations have indicated that, in terms of decreasing time and storage utilization, both **first-fit and best-fit perform better than worst-fit**.

- While neither first-fit nor best-fit is definitively superior in terms of storage utilization, **first-fit is usually faster**.

What is the need of Page replacement? Consider the following reference string 7,0,1,2,0,3, 0, 4, 2, 3, 0, 3,2, 1,2,0, 1, 7, 0, 1 Find the number of Page Faults with FIFO, Optimal Page replacement and LRU with four free frames which are empty initially. Which algorithm gives the minimum number of page faults?

Page replacement is a critical memory management technique used in operating systems that implement **demand paging**. Demand paging allows a process to execute even if its entire memory image is not loaded into the main memory at once. This separation of logical memory (what the process sees) from physical memory (the actual RAM) is the foundation of **virtual memory**, which enables us to run programs larger than the available physical memory.

However, as multiple processes run concurrently and demand pages from secondary storage (like a hard disk), the available **page frames** in the main memory may become exhausted. At this point, the OS needs a way to decide which existing page in memory should be **replaced** (swapped out to disk) to make room for the newly requested page. This is where page replacement algorithms come into play.

Without page replacement, the OS would have to terminate the requesting process or swap out an entire process to free up frames, both of which are detrimental to system performance and user experience. Page replacement allows us to keep more processes in memory, potentially increasing **multiprogramming** and overall **CPU utilization**.

Page Replacement Algorithms in Action

Let's analyze the given reference string:

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

using three common page replacement algorithms: FIFO, Optimal, and LRU, with **four** initially empty frames.

FIFO (First-In, First-Out)

FIFO replaces the page that has been in memory the longest.

Reference	Frame 1	Frame 2	Frame 3	Frame 4	Page Fault?
7	7				Yes
0	7	0			Yes
1	7	0	1		Yes
2	7	0	1	2	Yes
0	0	0	1	2	No
3	0	3	1	2	Yes
0	0	3	1	2	No
4	0	3	1	4	Yes
2	2	3	1	4	Yes
3	2	3	1	4	No
0	2	3	0	4	Yes
3	2	3	0	4	No
2	2	3	0	4	No
1	2	3	1	4	Yes
2	2	3	1	4	No
0	2	3	1	0	Yes
1	2	3	1	0	No
7	2	7	1	0	Yes
0	2	7	1	0	No
1	2	7	1	0	No

Total Page Faults: 10

Optimal Page Replacement

Optimal replacement replaces the page that will not be used for the longest time in the future. This requires knowing the future reference string, making it impractical for real-world systems but useful as a benchmark.

Reference	Frame 1	Frame 2	Frame 3	Frame 4	Page Fault?
7	7				Yes
0	7	0			Yes
1	7	0	1		Yes
2	7	0	1	2	Yes
0	7	0	1	2	No
3	7	0	1	3	Yes
0	7	0	1	3	No
4	7	0	4	3	Yes
2	7	0	4	2	Yes
3	7	0	4	2	No
0	7	0	4	2	No
3	7	0	4	2	No
2	7	0	4	2	No
1	7	0	4	1	Yes
2	7	0	4	1	No
0	7	0	4	1	No
1	7	0	4	1	No
7	7	0	4	1	No
0	7	0	4	1	No
1	7	0	4	1	No

Total Page Faults: 7

LRU (Least Recently Used)

LRU replaces the page that has not been used for the longest period of time. It approximates the optimal algorithm by looking at past usage.

Reference	Frame 1	Frame 2	Frame 3	Frame 4	Page Fault?
7	7				Yes
0	7	0			Yes
1	7	0	1		Yes
2	7	0	1	2	Yes
0	0	0	1	2	No
3	0	3	1	2	Yes
0	0	3	1	2	No
4	0	3	4	2	Yes
2	0	3	4	2	No
3	0	3	4	2	No
0	0	3	4	2	No
3	0	3	4	2	No
2	0	3	4	2	No
1	1	3	4	2	Yes
2	1	3	4	2	No
0	1	3	0	2	Yes
1	1	3	0	2	No
7	1	7	0	2	Yes
0	1	7	0	2	No

1	1	7	0	2	No

Total Page Faults: 8

Comparing the Algorithms

As expected, the **optimal page replacement algorithm yields the minimum number of page faults (7)**. LRU, which attempts to approximate optimal replacement, performs reasonably well with 8 page faults. FIFO, while simple, results in the most page faults (10) in this scenario.

Explain how paging supports virtual memory. With a neat diagram. Explain how logical address is translated into physical address

Paging is a memory management technique that forms the cornerstone of virtual memory systems. Let's explore how it works.

Paging and Virtual Memory

Virtual memory allows a process to have a logical address space that can be larger than the available physical memory. This is achieved by dividing both the **logical address space** (what the process sees) and the **physical address space** (the actual RAM) into fixed-size blocks called **pages** and **frames**, respectively.

- **Pages** are chunks of the logical address space.
- **Frames** are chunks of the physical memory.

When a process needs to execute, only a subset of its pages are loaded into any available memory frames. The rest of the pages reside on secondary storage (typically a hard disk), often in a dedicated area called **swap space**.

Demand Paging

The core principle of **demand paging** is to load pages from secondary storage into main memory only when they are needed. This is in contrast to loading the entire process image into memory at once.

1. **Page Table:** Each process has a **page table** that acts as a mapping between its logical pages and the corresponding physical frames in memory. The page table entries can indicate whether a page is:
    - **Valid:** Residing in a physical frame.
    - **Invalid:** Not currently in memory or not part of the process's logical address space.
2. **Page Faults:** When a process tries to access a page that is not currently in memory (marked as invalid in the page table), a **page fault** occurs. This triggers a trap to the operating system.
3. **Handling Page Faults:** The operating system handles a page fault by:
    - Locating the desired page on the disk.
    - Finding a free frame or, if none are available, selecting a victim frame using a **page replacement algorithm**.
    - Loading the desired page into the free frame from disk.
    - Updating the page table to reflect the new mapping.
4. **Restarting the Instruction:** Once the required page is in memory, the instruction that caused the page fault is restarted, and the process continues execution.

Address Translation: Logical to Physical

Figure 8.10 from the sources visually illustrates the hardware support for paging.

[Image of Figure 8.10: Paging hardware]

1. **Logical Address:** The CPU generates a **logical address** for each memory access.
2. **Page Number and Offset:** The logical address is split into two parts:
    - **Page Number:** Used as an index into the page table to find the corresponding frame number.
    - **Page Offset:** Specifies the location of the desired data within the page.

3. **Page Table Lookup:** The page number is used to index into the **page table**, retrieving the base address of the corresponding frame in physical memory.
4. **Physical Address Calculation:** The **physical address** is formed by combining the base address of the frame (from the page table) with the page offset. This address is then sent to the memory unit.

Figure 8.11 from the sources provides a model of the paging system showing how logical pages are mapped to non-contiguous physical frames through the page table.

[Image of Figure 8.11: Paging model of logical and physical memory]

This address translation mechanism is crucial for demand paging to function correctly. It allows the operating system to place pages of a process anywhere in physical memory and maintain a consistent logical view for the process.

Key Advantages of Paging

1. **Processes Larger than Physical Memory:** Programs can be larger than the available physical memory, as only the necessary pages are loaded.
2. **Increased Multiprogramming:** By loading only a subset of each process's pages, more processes can reside in memory concurrently, potentially improving CPU utilization.
3. **Simplified Programming:** Programmers are relieved from the complexities of memory management, as the operating system handles the mapping between logical and physical addresses.
4. **Memory Protection:** Each process has its own page table, preventing it from accessing memory outside its allocated pages, enhancing system stability and security.
5. **Efficient Memory Sharing:** Processes can share physical memory by mapping different virtual pages to the same physical frame. This is beneficial for shared libraries and inter-process communication.

Consider an executable problem that has to be loaded from disk to memory. Suggest a technique that loads pages only when they are needed and discuss the suggested technique in detail.

Demand Paging: A Technique for Loading Pages Only When Needed

The technique that loads pages into memory only when they are required is called **demand paging**. This method is a fundamental aspect of virtual memory systems and allows processes to execute even if their entire logical address space is not resident in physical memory.

How Demand Paging Works

- **Lazy Swapper:** Demand paging employs a "lazy swapper" that avoids loading pages into memory unless they are absolutely necessary. This contrasts with traditional swapping, where entire processes are moved between main memory and secondary storage.
- **Valid-Invalid Bits:** Demand paging leverages the **valid–invalid bit** in page table entries to differentiate between pages that are currently in memory and those that reside on disk.
    - A **valid** bit indicates that the corresponding page is both legal and present in a physical frame.
    - An **invalid** bit signifies that the page is either not part of the process's logical address space or is valid but currently residing on disk.
- **Page Faults:** When a process attempts to access a page marked as invalid in its page table, a **page fault** occurs. This triggers a trap to the operating system, signaling that the requested page needs to be brought into memory.

Steps in Handling a Page Fault

1. **Validate Memory Access:** The operating system verifies whether the memory access that caused the page fault is valid within the process's logical address space. If the access is invalid, the process is terminated.
2. **Locate Free Frame:** A free frame is allocated from the free-frame list or obtained through page replacement if no free frames are available.

3. **Initiate Disk Read:** A disk operation is scheduled to read the required page from secondary storage (swap space) into the newly allocated frame.
4. **Update Page Table:** Once the disk read is complete, the page table entry for the requested page is updated to mark it as valid and to reflect its mapping to the corresponding physical frame.
5. **Restart Instruction:** The instruction that caused the page fault is restarted, and the process continues as if the page had always been in memory.

Advantages of Demand Paging

1. **Reduced I/O and Memory Usage:** By loading only the necessary pages, demand paging minimizes disk I/O and the amount of physical memory required. Pages that are never accessed are not loaded at all, leading to more efficient use of resources.
2. **Support for Large Processes:** Processes can be larger than the available physical memory, as their entire logical address space does not need to be resident in RAM simultaneously.
3. **Increased Degree of Multiprogramming:** Demand paging enables a higher degree of multiprogramming by allowing more processes to reside in memory concurrently. This can improve CPU utilization by reducing the time spent waiting for pages to be swapped in.

Overall, demand paging is a powerful technique that enables efficient memory management by loading pages into memory only when they are required. It contributes to increased system performance, the ability to run large processes, and a higher degree of multiprogramming. Careful management of swap space and techniques to minimize page faults are essential for maximizing the effectiveness of demand paging.

For the following page reference string

2,3,0,1,5,4,0,1,2,5,4,3,4,2,3,5

Compute the number of page faults following the:

1. LRU page replacement policy with 3 frames
2. Optimal page replacement policy with 4 frames.

Calculating Page Faults for LRU and Optimal Page Replacement

This response will calculate the number of page faults for the given page reference string using both the LRU (Least Recently Used) and Optimal page replacement policies.

LRU Page Replacement with 3 Frames

The LRU algorithm replaces the page that has not been used for the longest period. Here's a step-by-step breakdown of how it works with the given reference string and 3 frames:

Reference String: 2, 3, 0, 1, 5, 4, 0, 1, 2, 5, 4, 3, 4, 2, 3, 5

Frames: 3

Reference	Frame 1	Frame 2	Frame 3	Page Fault	Explanation
2	2	-	-	Yes	Page 2 is not in memory and a frame is available.
3	2	3	-	Yes	Page 3 is not in memory and a frame is available.
0	2	3	0	Yes	Page 0 is not in memory and a frame is available.
1	1	3	0	Yes	Page 1 is not in memory, and page 2 is the least recently used.

5	1	5	0	Yes	Page 5 is not in memory, and page 3 is the least recently used.
4	1	5	4	Yes	Page 4 is not in memory, and page 0 is the least recently used.
0	0	5	4	Yes	Page 0 is not in memory, and page 1 is the least recently used.
1	0	1	4	Yes	Page 1 is not in memory, and page 5 is the least recently used.
2	0	1	2	Yes	Page 2 is not in memory, and page 4 is the least recently used.
5	0	5	2	Yes	Page 5 is not in memory, and page 1 is the least recently used.
4	0	5	4	Yes	Page 4 is not in memory, and page 2 is the least recently used.
3	3	5	4	Yes	Page 3 is not in memory, and page 0 is the least recently used.
4	3	5	4	No	Page 4 is already in memory.
2	2	5	4	Yes	Page 2 is not in memory, and page 3 is the least recently used.
3	2	3	4	Yes	Page 3 is not in memory, and page 5 is the least recently used.

5	2	3	5	Yes	Page 5 is not in memory, and page 4 is the least recently used.

Total Page Faults for LRU with 3 Frames: 14

Optimal Page Replacement with 4 Frames

The Optimal algorithm replaces the page that will not be used for the longest period in the future. While not practically implementable, it serves as a benchmark. Here's how it works with the given reference string and 4 frames:

Reference String: 2, 3, 0, 1, 5, 4, 0, 1, 2, 5, 4, 3, 4, 2, 3, 5

Frames: 4

Reference	Frame 1	Frame 2	Frame 3	Frame 4	Page Fault	Explanation
2	2	-	-	-	Yes	Page 2 is not in memory and a frame is available.
3	2	3	-	-	Yes	Page 3 is not in memory and a frame is available.
0	2	3	0	-	Yes	Page 0 is not in memory and a frame is available.
1	2	3	0	1	Yes	Page 1 is not in memory and a frame is available.
5	2	3	0	5	Yes	Page 5 is not in memory and a frame is available.
4	2	3	4	5	Yes	Page 4 is not in memory, page 2 will not be used for the longest time in the future.
0	2	3	4	5	No	Page 0 is already in memory.

1	2	3	4	5	No	Page 1 is already in memory.
2	2	3	4	5	No	Page 2 is already in memory.
5	2	3	4	5	No	Page 5 is already in memory.
4	2	3	4	5	No	Page 4 is already in memory.
3	2	3	4	5	No	Page 3 is already in memory.
4	2	3	4	5	No	Page 4 is already in memory.
2	2	3	4	5	No	Page 2 is already in memory.
3	2	3	4	5	No	Page 3 is already in memory.
5	2	3	4	5	No	Page 5 is already in memory.

Total Page Faults for Optimal with 4 Frames: 6

Conclusion

As expected, the Optimal page replacement algorithm results in fewer page faults compared to the LRU algorithm for the same reference string. This highlights the theoretical efficiency of Optimal replacement. However, it's crucial to remember that Optimal replacement requires future knowledge of the reference string, making it impractical for real-world systems. LRU, while not optimal, is a more practical choice and generally provides good performance due to the principle of locality of reference.

Demonstrate with an example how memory management is achieved through swapping in UNIX systems.

Swapping is a memory management technique where processes are moved between main memory and a backing store, typically a fast disk. This allows a system to run more processes than can fit in memory simultaneously, increasing the degree of multiprogramming.

Example: Imagine a UNIX system with 2GB of RAM and three processes (A, B, and C), each requiring 1GB of memory. Without swapping, only two processes could run concurrently. With swapping, all three processes can be loaded, but only two will reside in RAM at a time.

1. **Process Loading:** Initially, processes A and B are loaded into memory, while process C resides in the swap space on disk.
2. **Swap Out:** When process C needs to run, the operating system selects either A or B to be swapped out to disk, freeing up space in RAM.
3. **Swap In:** Process C is then swapped in from disk to the newly available RAM space and begins execution.
4. **Context Switching:** This swapping in and out of processes continues as needed, allowing all three processes to execute despite limited RAM.

Important Considerations for Swapping

- **Swap Time:** Moving processes between memory and disk can be time-consuming, especially for large processes. Factors like transfer rate and process size impact the overall swap time.
- **Modified Pages:** When swapping out a process, only modified pages need to be written back to disk, potentially saving time.
- **Reduced Usage in Modern Systems:** While swapping is an important concept, it's not commonly used in its standard form in modern operating systems. It's often replaced or supplemented by more sophisticated techniques like demand paging and virtual memory.

Modern Adaptations of Swapping

While not used in its pure form, variations of swapping are implemented in systems like UNIX, Linux, and Windows. These adaptations aim to mitigate the performance impact of traditional swapping. Some common variations include:

- **Threshold-based Swapping:** Swapping is activated only when free memory falls below a certain threshold and deactivated when it rises again.
- **Partial Swapping:** Instead of entire processes, only portions or specific pages are swapped, reducing the time overhead.

It is worth noting that mobile operating systems like iOS and Android generally do not support swapping. These systems use demand paging from the file system and may reclaim read-only pages when memory is constrained. They might terminate processes to free up memory if necessary, sometimes saving application states to flash memory for quick restart.

Consider the following page reference string:

1, 2, 3, 4, 2, 1, 4, 5, 7, 1, 2, 3, 3, 6, 3, 2, 4, 2, 3, 6.

Assuming demand paging with four frames, how many page faults would occur for the following replacement algorithms (3 marks each)?

- LRU
- FIFO

Page Faults in Demand Paging

This response will calculate the number of page faults for the given page reference string using the LRU and FIFO page replacement algorithms.

LRU Page Replacement with 4 Frames

The LRU (Least Recently Used) algorithm replaces the page that hasn't been used for the longest time. With 4 frames and the reference string 1, 2, 3, 4, 2, 1, 4, 5, 7, 1, 2, 3, 3, 6, 3, 2, 4, 2, 3, 6, the page faults are as follows:

Reference	Frame 1	Frame 2	Frame 3	Frame 4	Page Fault	Explanation
1	1	-	-	-	Yes	Initial page fault, page 1 loaded.
2	1	2	-	-	Yes	Initial page fault, page 2 loaded.
3	1	2	3	-	Yes	Initial page fault, page 3 loaded.
4	1	2	3	4	Yes	Initial page fault, page 4 loaded.
2	1	2	3	4	No	Page 2 already in memory.
1	1	2	3	4	No	Page 1 already in memory.
4	1	2	3	4	No	Page 4 already in memory.
5	5	2	3	4	Yes	Page 5 not in memory, page 1 replaced as it's least recently used.
7	5	7	3	4	Yes	Page 7 not in memory, page 2 replaced as it's least recently used.
1	5	7	1	4	Yes	Page 1 not in memory, page 3 replaced as it's least recently used.
2	5	7	1	2	Yes	Page 2 not in memory, page 4 replaced as it's least recently used.
3	5	7	1	2	No	Page 3 already in memory.

3	5	7	1	2	No	Page 3 already in memory.
6	6	7	1	2	Yes	Page 6 not in memory, page 5 replaced as it's least recently used.
3	6	7	1	3	Yes	Page 3 not in memory, page 2 replaced as it's least recently used.
2	6	7	2	3	Yes	Page 2 not in memory, page 1 replaced as it's least recently used.
4	6	7	2	4	Yes	Page 4 not in memory, page 3 replaced as it's least recently used.
2	6	7	2	4	No	Page 2 already in memory.
3	6	7	2	3	No	Page 3 already in memory.
6	6	7	2	3	No	Page 6 already in memory.

Total Page Faults for LRU with 4 Frames: 13

FIFO Page Replacement with 4 Frames

In the FIFO (First-In, First-Out) algorithm, the oldest page in memory is replaced. The reference string is processed using 4 frames, similar to the previous example, but using the FIFO rule for replacement.

Reference	Frame 1	Frame 2	Frame 3	Frame 4	Page Fault	Explanation
1	1	-	-	-	Yes	Page 1 loaded into an empty frame.

2	1	2	-	-	Yes	Page 2 loaded into an empty frame.
3	1	2	3	-	Yes	Page 3 loaded into an empty frame.
4	1	2	3	4	Yes	Page 4 loaded into an empty frame.
2	1	2	3	4	No	Page 2 is already in a frame.
1	1	2	3	4	No	Page 1 is already in a frame.
4	1	2	3	4	No	Page 4 is already in a frame.
5	5	2	3	4	Yes	Page 5 is not in a frame. Page 1 is replaced because it was loaded first.
7	5	7	3	4	Yes	Page 7 is not in a frame. Page 2 is replaced because it was loaded next.
1	5	7	1	4	Yes	Page 1 is not in a frame. Page 3 is replaced because it was loaded next.
2	5	7	1	2	Yes	Page 2 is not in a frame. Page 4 is replaced because it was loaded next.
3	3	7	1	2	Yes	Page 3 is not in a frame. Page 5 is replaced

						because it was loaded next.
3	3	7	1	2	No	Page 3 is already in a frame.
6	3	6	1	2	Yes	Page 6 is not in a frame. Page 7 is replaced because it was loaded next.
3	3	6	1	2	No	Page 3 is already in a frame.
2	3	6	1	2	No	Page 2 is already in a frame.
4	4	6	1	2	Yes	Page 4 is not in a frame. Page 3 is replaced because it was loaded next.
2	4	6	1	2	No	Page 2 is already in a frame.
3	4	6	1	3	Yes	Page 3 is not in a frame. Page 2 is replaced because it was loaded next.
6	4	6	1	3	No	Page 6 is already in a frame.

Total Page Faults for FIFO with 4 Frames: 14

What is a page fault and how it is handled by the operating system?

A **page fault** occurs when a process tries to access a page that is not currently in main memory. In other words, the page the process needs is marked as "invalid" in the page table. This triggers a trap to the operating system, signifying the OS failed to bring the needed page into memory.

Handling a Page Fault

The operating system follows these steps when a page fault occurs:

1. **Verify Access:** The OS checks if the memory access was valid using an internal table, likely stored within the process control block. An invalid access results in process termination.
2. **Locate Page:** If the access is valid but the page is missing, the OS determines the page's location on the disk.
3. **Find a Free Frame:**
    - If a free frame exists, the OS uses it.
    - If not, a page-replacement algorithm selects a "victim" frame to be freed. The contents of this victim frame are written to the swap space (usually a designated disk area), and relevant tables are updated to reflect the page's absence from memory.
4. **Read Desired Page:** The OS schedules a disk operation to read the required page from the disk into the newly freed frame.
5. **Update Tables:** Once the disk read completes, the internal table and page table are updated to mark the page as present in memory.
6. **Restart Instruction:** The interrupted instruction that caused the page fault is restarted, and the process continues as if the page was always in memory.

Key Points and Considerations

- **Pure Demand Paging:** A strategy where pages are brought into memory only when needed. This can lead to a large number of page faults initially, but minimizes memory usage.
- **Modify (Dirty) Bit:** A hardware bit associated with each page or frame that indicates if the page has been modified since being loaded. It helps optimize page replacement. If the modify bit is set, the page must be written back to disk before being replaced. Otherwise, no disk write is needed.
- **Performance Impact:** Page faults introduce overhead due to disk I/O. The effective access time increases significantly if the page-fault

rate is high. This can lead to thrashing, where a process spends more time paging than executing, causing performance degradation.

- **Restarting Instructions:** Demand paging necessitates restarting instructions after a page fault. The operating system must ensure the process can resume from the exact point of interruption, even if the instruction involved modifications to multiple memory locations.
- **I/O Interlock and Page Locking:** During I/O operations involving virtual memory, pages involved in the transfer may need to be locked into memory to prevent them from being replaced. This is crucial because an I/O operation targeting a specific virtual address might fail if that page is swapped out before the operation completes.

The address generated by CPU is called as the logical address. Demonstrate the address translation scheme that converts the logical address into physical address.

Address Translation from Logical to Physical Address

The **logical address** is generated by the CPU and is translated to a **physical address** that the memory unit sees. The **memory-management unit (MMU)** is responsible for the run-time mapping from logical/virtual addresses to physical addresses.

Address Binding

A user program typically goes through several steps before being executed. During these steps, addresses can be represented in different ways.

- **Symbolic addresses** in the source program (e.g., variable count) are bound to **relocatable addresses** by a compiler (e.g., "14 bytes from the beginning of this module").
- **Relocatable addresses** are bound to **absolute addresses** by a linkage editor or loader (e.g., 74014).

Address Spaces

- **Logical Address Space:** The set of all logical addresses generated by a program.

- **Physical Address Space:** The set of all physical addresses corresponding to the logical addresses.

In execution-time address binding, the logical and physical address spaces are different.

Relocation Register

The MMU uses a **relocation register** to map logical addresses to physical addresses. The relocation register contains a base value that is added to every logical address generated by a user process to produce the physical address.

- Example: If the base address in the relocation register is 14000, then an attempt to access logical address 346 is mapped to physical address 14346 (14000 + 346).

Note: The user program only deals with logical addresses and never sees the actual physical addresses. The memory-mapping hardware hides the address translation from the user.

Why Use Logical Addresses?

The concept of logical addresses bound to a separate physical address space is crucial for memory management. This separation:

- Allows a user program to be placed in any part of physical memory.
- Facilitates **dynamic relocation**, which is essential for memory compaction (a technique to reduce external fragmentation).
- Enables **virtual memory**, allowing a program to use a larger logical address space than the available physical memory.

Memory gets splitted into number of partitions when assigned to processes. Demonstrate the concept from the above statement, its types and discuss its Causes.

Memory Partitioning

The statement "Memory gets splitted into a number of partitions when assigned to processes" refers to the concept of **memory partitioning**, a

technique used by operating systems to manage and allocate portions of main memory to multiple processes.

Types of Memory Partitioning

- **Fixed Partitioning:**
    - Memory is divided into a predetermined number of fixed-size partitions.
    - Each partition can hold only one process at a time.
    - The degree of multiprogramming (number of processes running concurrently) is limited by the number of partitions.
    - **Example:** IBM OS/360 (MFT - Multiprogramming with a Fixed number of Tasks).
- **Variable Partitioning:**
    - The sizes of partitions are variable and adjusted dynamically as processes arrive and leave.
    - The operating system maintains a table or list of available memory blocks (holes) and allocates memory to processes based on their requirements.
    - Offers more flexibility than fixed partitioning and can better accommodate processes with varying memory needs.
    - **Example:** IBM OS/360 (MVT - Multiprogramming with a Variable number of Tasks).

Causes of Memory Partitioning

The primary reason for partitioning memory is to support **multiprogramming**, which allows multiple processes to reside in memory simultaneously. This practice significantly improves CPU utilization and system throughput. The sources provide insights into the factors leading to the need for memory partitioning:

- **Limited Physical Memory:** Early computer systems had limited physical memory. Partitioning allowed the system to run multiple programs concurrently, even if they couldn't fit entirely in memory.

- **Resource Sharing:** Partitioning enables the operating system to isolate processes from each other, preventing them from interfering with each other's memory. This isolation is essential for system stability and security.
- **Concurrency and Parallelism:** Modern operating systems strive for concurrency and parallelism, running multiple processes seemingly at the same time. Memory partitioning is fundamental to achieving this goal.
- **Supporting Virtual Memory:** Partitioning is a prerequisite for implementing virtual memory, a technique that allows processes to have a larger logical address space than the available physical memory.
- **Supporting Multiprocessing:** In multiprocessing systems, multiple CPUs can access memory concurrently. Memory partitioning helps manage access and ensures proper allocation for each process running on the different CPUs.

Explain the following terms in brief 1)Virtual Memory, 2)Compaction 3)Thrashing 4)Belady's Anomaly

Understanding Key Memory Management Concepts

- **Virtual Memory:** Virtual memory is a memory management technique that allows a process to execute even if it is not completely in memory. This technique decouples the logical address space from the physical address space, giving the illusion of a much larger memory space than what is physically available. It enables programs to be larger than physical memory and provides a mechanism for processes to share files and implement shared memory efficiently. Demand paging is a common implementation of virtual memory, where pages are loaded into physical memory only when they are needed, as discussed in our previous conversation.
- **Compaction:** Compaction is a memory management technique used to address external fragmentation. External fragmentation occurs when enough free memory space exists to satisfy a request, but it is not contiguous. Compaction involves shuffling the memory

contents to consolidate the free space into a larger block, allowing larger processes to be loaded. Compaction is only possible if relocation is dynamic and done at execution time.

- **Thrashing:** Thrashing occurs when a process spends more time paging than executing. It arises when a process does not have enough frames to accommodate its working set, the set of pages it is actively using. This leads to a constant cycle of page faults and replacements, resulting in severe performance degradation. If CPU utilization drops too low, the operating system may try to increase the degree of multiprogramming, which can exacerbate thrashing if insufficient memory is available.
- **Belady's Anomaly:** Belady's Anomaly is a counterintuitive phenomenon where increasing the number of allocated frames to a process can sometimes lead to an increase in page faults. This anomaly occurs with certain page replacement algorithms, like FIFO (First-In, First-Out), while other algorithms, such as LRU (Least Recently Used), are not susceptible to it. Belady's Anomaly highlights that not all page replacement algorithms behave intuitively, and a seemingly beneficial change (like adding more memory) can have unintended consequences.

Explain the data structures used for Demand paging in UNIX.

Data Structures in UNIX Demand Paging

Demand paging in UNIX, as in many other operating systems, relies on several key data structures to manage the virtual-to-physical memory mapping and handle page faults efficiently. While the sources do not directly list the data structures used for demand paging in UNIX, they offer relevant information that allows us to infer those structures:

- **Page Table:** The page table is the central data structure for mapping virtual addresses to physical addresses. In UNIX systems with demand paging, the page table has entries that indicate whether a particular virtual page is resident in memory or on disk. Each entry in the page table typically includes:

    - **Frame Number:** If the page is resident in memory, this field points to the physical frame containing the page.
    - **Present/Absent Bit:** This bit indicates whether the page is currently in memory (present) or on disk (absent). A page fault occurs when a process tries to access a page marked as absent.
    - **Modified/Dirty Bit:** This bit indicates whether the page has been modified since it was last loaded from disk. If a page has been modified, it must be written back to disk before it can be replaced.
    - **Protection Bits:** These bits specify the access permissions (read, write, execute) for the page.
    - **Other Control Information:** May include information for page replacement algorithms, such as access time or reference bits.
- **Frame Table:** The frame table keeps track of the allocation status of each physical frame. Each entry in the frame table might indicate:
    - **Free/Allocated:** Whether the frame is currently free or allocated to a page.
    - **Process ID:** If allocated, the ID of the process to which the frame is allocated.
    - **Page Number:** The virtual page number mapped to this frame.
    - **Other Information:** May include information about the page's location in the swap space on disk.
- **Swap Space:** Swap space on the hard disk is used to store pages that have been swapped out of physical memory. It is typically a dedicated partition or a file on the file system. The sources mention that:
    - "The secondary memory is usually a high-speed disk. It is known as the swap device, and the section of disk used for this purpose is known as swap space."

The organization and management of the swap space can vary depending on the specific UNIX implementation. Data structures related to swap space management might include:

- **Swap Map:** A data structure to track the allocation and availability of blocks in the swap space.
- **Swap-Space Descriptors:** Information about the swap devices or files used for paging.

Three storage solutions problems to the are dynamic first-fit, storage best-fit allocation and worst- fit. Consider a memory block which has partitions of the following sizes and in the following order: 46 Kbytes ,200 Kbytes ,350 Kbytes. Determine Kbytes, in which 60 partition Kbytes, a and 55 679 Kbyte process should be placed under each scheme above. Justify your choice of partition in each case.

The query presents a scenario of dynamic storage allocation with three common strategies: first-fit, best-fit, and worst-fit. Let's break down each strategy and determine the appropriate partition for a 55 KB process given the available memory blocks: 46 KB, 200 KB, 350 KB, 60 KB, and 679 KB.

First-Fit Allocation

Definition: The first-fit strategy allocates the first available hole that is large enough to accommodate the process.

Application:

- The first hole encountered is 46 KB. This hole is too small.
- The next hole is 200 KB. This hole is large enough to hold the 55 KB process.

Decision: The 55 KB process would be placed in the **200 KB** partition.

Justification: First-fit prioritizes speed. It stops searching as soon as a suitable hole is found, even if there might be a better fit later.

Best-Fit Allocation

Definition: The best-fit strategy allocates the smallest hole that is big enough for the process. This aims to minimize external fragmentation.

Application:

- The holes are examined to find the smallest hole that can accommodate 55 KB:
    - 46 KB: Too small
    - 200 KB: Large enough
    - 350 KB: Large enough
    - 60 KB: Large enough
    - 679 KB: Large enough
- The 60 KB partition is the smallest hole that can fit the process.

Decision: The 55 KB process would be placed in the **60 KB** partition.

Justification: Best-fit aims to minimize the size of the leftover hole, making it potentially more useful for allocating smaller processes in the future.

Worst-Fit Allocation

Definition: The worst-fit strategy allocates the largest hole available. The idea is that the leftover space will be larger and potentially more useful.

Application:

- The largest available hole is the 679 KB partition.

Decision: The 55 KB process would be placed in the **679 KB** partition.

Justification: Worst-fit aims to maximize the size of the remaining hole after allocation. However, simulations have shown that this strategy is generally less efficient than first-fit and best-fit in terms of both time and storage utilization.