

DEPARTMENT OF COMPUTER APPLICATIONS

MODULE 4

TOPIC 1 — MEMORY-MANAGEMENT BACKGROUND

Definition

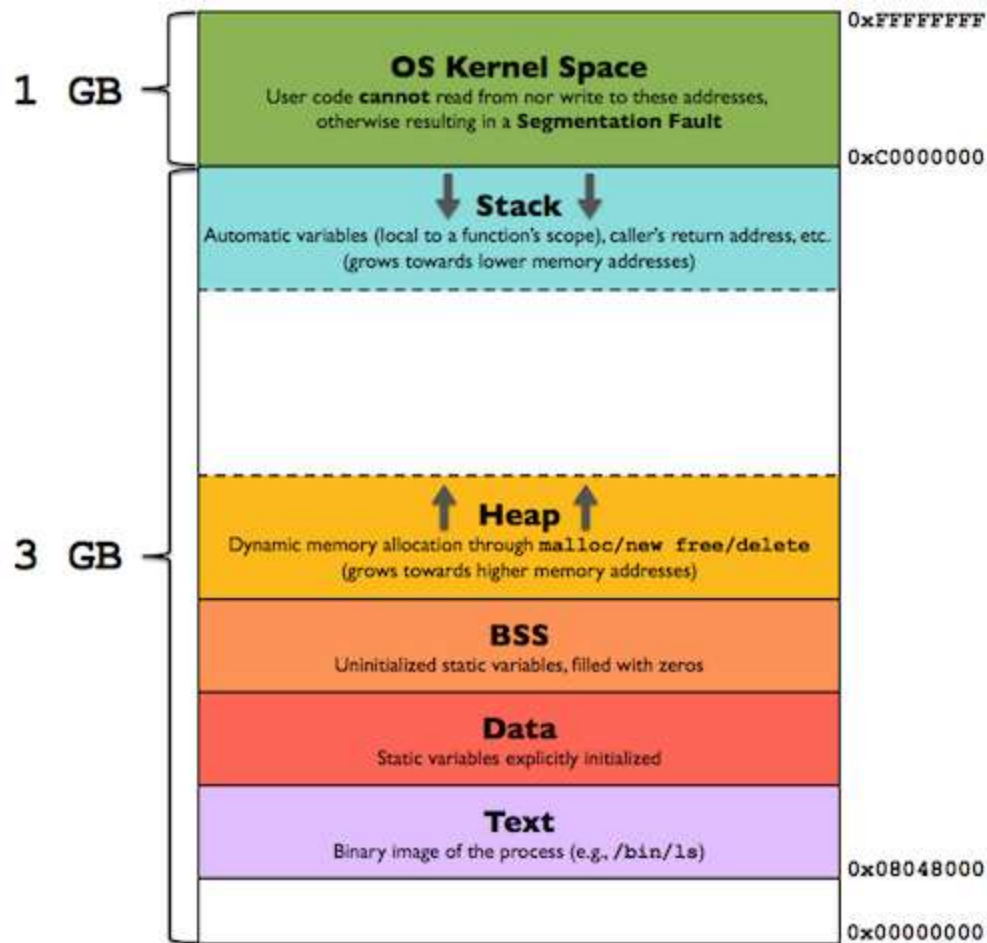
Memory management is the function of an operating system that handles the allocation and deallocation of memory space to processes. It ensures that each process receives the memory it requires while maintaining system protection and efficiency. This system keeps track of memory usage and prevents processes from interfering with each other's memory space. Efficient memory management improves utilization, performance, and process execution speed.

Introduction

Every program executed in a computer system requires memory for code, data, stack, and heap. Main memory (RAM) acts as a large array of bytes, each with its own address, and the OS must manage it safely and efficiently. A CPU can only access instructions and data located in main memory; thus, memory plays a central role in process execution. However, memory is limited, and multiple processes must share it simultaneously, requiring careful allocation. Memory management also includes protection so that processes cannot access each other's memory. The OS maintains data structures like page tables and segment tables to translate logical addresses to physical ones. Early systems used contiguous memory allocation, but modern systems use paging and segmentation to minimize fragmentation. Memory management also interacts with virtual memory, allowing systems to run programs larger than physical RAM. The goal is to maximize CPU utilization while minimizing memory waste. Thus, memory management is essential for efficient multiprogramming and stable system operation.

Diagram (Memory Layout in OS)

DEPARTMENT OF COMPUTER APPLICATIONS



Explanation

In any computer system, memory is a critical resource required by all executing programs. The CPU fetches instructions directly from main memory, which necessitates efficient management. Memory management begins with the concept of **address spaces**, where each process is given its own logical address range. These logical addresses must be mapped to physical addresses using hardware such as the **Memory Management Unit (MMU)**.

One of the earliest memory techniques was **contiguous allocation**, where each process occupied a single, continuous block of memory. Though simple, it suffered from external fragmentation and limited flexibility. To solve this, modern systems adopt **non-contiguous allocation** methods like paging and segmentation.

DEPARTMENT OF COMPUTER APPLICATIONS

The operating system maintains tables that store the mapping between logical and physical addresses. For paging systems, the page table maps page numbers to frame numbers, while in segmentation, segment tables map segments to physical memory. These mappings allow processes to be loaded into any available frames rather than requiring contiguous blocks.

Memory protection is enforced using base and limit registers or page table entries. This prevents processes from accessing each other's memory, ensuring stability and security. Every memory reference undergoes address translation, often using hardware support to minimize overhead.

Multiprogramming depends heavily on memory management because multiple processes must reside in memory simultaneously. When memory is full, operating systems may temporarily move processes to disk, a technique known as **swapping**. Demand paging and virtual memory extend these capabilities, allowing systems to execute programs larger than the available memory.

Memory management also involves handling internal and external fragmentation. Paging eliminates external fragmentation but may suffer from internal fragmentation, whereas segmentation is more flexible but reintroduces external fragmentation. The OS must balance these trade-offs while ensuring performance.

In summary, memory-management background covers all fundamental principles required to understand advanced memory techniques like paging, segmentation, and virtual memory.

Example

Imagine a process requiring 120 KB of memory.

On a simple contiguous allocation system, if memory has a free block of only 100 KB followed by another free block of 50 KB, the process cannot fit despite enough total free space.

This results in **external fragmentation**.

In contrast, with paging (say page size = 4 KB), the process is divided into 30 pages, which can be placed into any 30 free frames.

DEPARTMENT OF COMPUTER APPLICATIONS

This allows the process to run without needing contiguous memory blocks.

Modern OSs like Linux use paging to solve such allocation problems.

Conclusion

Memory-management background provides the foundation for understanding how the OS allocates, protects, and manages memory.

It highlights the need for efficient address translation, protection mechanisms, and allocation strategies.

These principles lead to advanced techniques such as paging, segmentation, and virtual memory.

A strong understanding of this topic is essential for studying process execution and system performance.

Thus, memory management forms the backbone of modern multiprogramming operating systems.

SWAPPING

Definition

Swapping is a memory-management technique in which processes are temporarily moved between main memory and secondary storage. The OS swaps out a process to disk when memory is full and swaps it back in when required. This allows more processes to be executed than the available physical memory can hold. Swapping increases the degree of multiprogramming by managing memory dynamically.

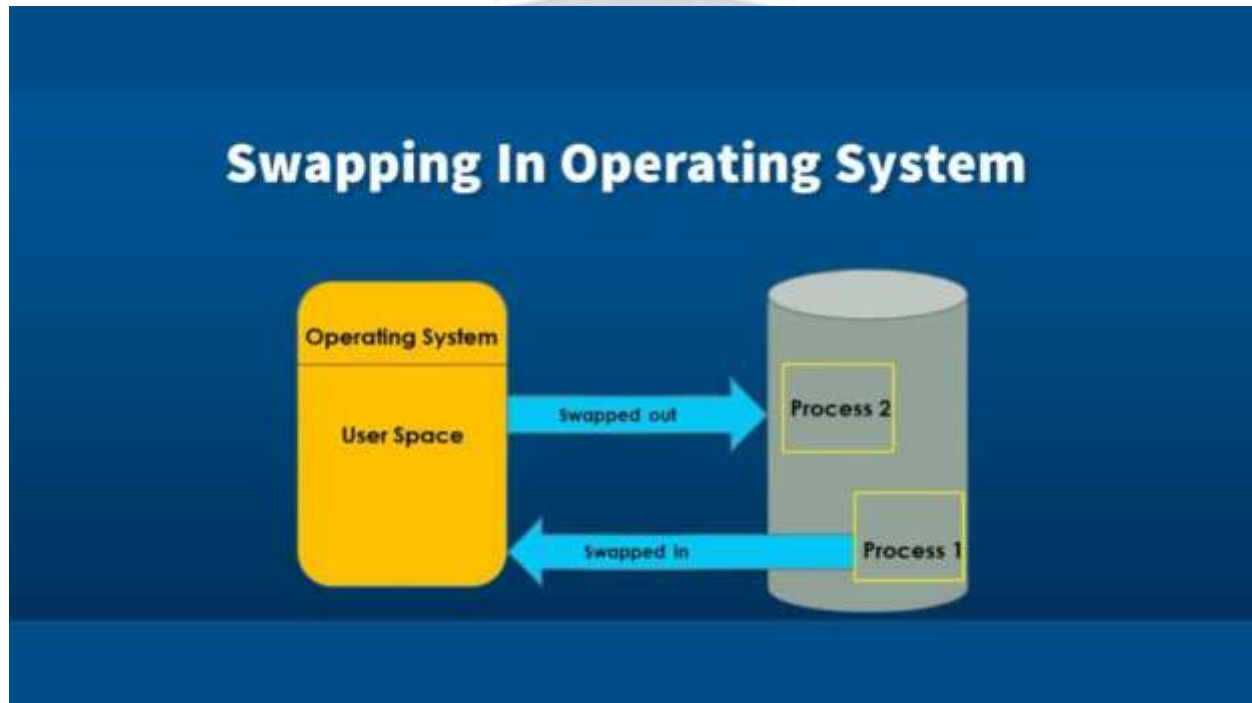
Introduction

Swapping was originally designed for early multiprogramming systems to overcome limited physical memory. It allows the operating system to remove entire processes from memory and store them on disk when space is needed. Later, when the CPU schedules that process again, the OS brings it back into memory. This mechanism makes it possible for many processes to exist even if total RAM is small. Swapping helps maintain system responsiveness when multiple programs compete for memory. Although modern systems rely more on paging and virtual memory, swapping remains an important backup mechanism. Linux and Unix-based systems still use swapping to handle memory pressure.

DEPARTMENT OF COMPUTER APPLICATIONS

situations. Swapping requires disk space (swap area) reserved specifically for this purpose. However, swapping may cause delays because disk access is slower than RAM. Thus, swapping must be used carefully to avoid performance degradation.

Diagram (Swapping Process)



Explanation (40 lines)

Swapping is a technique in which the operating system temporarily moves a process from main memory to a designated swap area on disk. When the process is not currently needed by the CPU or when memory becomes insufficient, it is swapped out to free space for other processes. Later, if the CPU schedules that process again, the OS swaps it back into memory.

A swap operation involves two major steps: **swap-out**, when a process is removed from memory and written to disk; and **swap-in**, when it is brought back to RAM. The time required for swapping depends

DEPARTMENT OF COMPUTER APPLICATIONS

heavily on the speed of the backing store, because transferring large memory images between RAM and disk takes significant time.

Swapping increases the degree of multiprogramming by allowing more programs to remain in a “ready” state, even if not all of them can fit in physical memory at the same time. However, swapping is expensive because it requires entire process images, including code, stack, and data, to be moved. This leads to long delays, particularly in systems with large processes.

Early operating systems used swapping heavily as their primary memory management technique. Modern systems, however, use **paging** and **demand paging**, but still retain swapping as a fallback mechanism when memory becomes critically low. Linux, for example, uses a combination of demand paging and swap partitions or swap files to manage memory pressure.

One issue with swapping is related to **memory fragmentation**. When processes of different sizes enter and exit main memory, free memory may become scattered into small holes. Swapping combined with contiguous allocation makes fragmentation more likely. Modern paging techniques reduce this problem.

Another factor is the **context of swapped-out processes**. The OS must save the program counter, CPU registers, and entire memory image. It must also update process states to reflect that the process is swapped out and not ready for execution until reloaded.

The OS must also manage timing: it should avoid swapping out processes that are actively waiting for I/O because they may need to access memory quickly. For this reason, I/O-bound processes are rarely swapped out.

Despite being slower, swapping remains useful in systems under high memory load, providing a safety buffer and preventing process termination.

Example

- Suppose a computer has 4 GB RAM and five processes, each needing 1.5 GB.

DEPARTMENT OF COMPUTER APPLICATIONS

- Only two processes can fit into memory at a time.
- When Process P1 finishes CPU execution temporarily, the OS swaps it out to disk.
- Process P3 is swapped into RAM to begin execution.
- Later, when P1 is scheduled again, P3 may be swapped out and P1 swapped back in.
- This allows all five processes to make progress even though total RAM is insufficient.

Conclusion

Swapping is a fundamental memory-management method that allows systems to handle more processes than the RAM can hold. Although slower due to disk access, it ensures continuity of operations under memory pressure. Swapping remains relevant in modern systems as a backup technique alongside paging. It increases multiprogramming levels and prevents memory exhaustion. Overall, swapping contributes to stable and flexible system performance.

CONTIGUOUS MEMORY ALLOCATION

Definition

Contiguous memory allocation is a memory-management technique in which each process is allocated a single continuous block of physical memory. The operating system assigns memory in such a way that a process occupies adjacent memory locations. This technique was used in early operating systems due to its simplicity. It requires careful management to avoid memory wastage and fragmentation.

Introduction

In early computer systems, memory management was relatively simple because only one process executed at a time. As multiprogramming systems evolved, multiple processes needed to reside in memory simultaneously. Contiguous memory allocation was one of the first techniques developed to manage this requirement. In this approach, each process is loaded into a contiguous section of main

DEPARTMENT OF COMPUTER APPLICATIONS

memory. The operating system keeps track of which parts of memory are free and which are occupied. This method is easy to implement and supports fast memory access. However, it introduces problems such as fragmentation and inefficient memory utilization. To overcome these issues, different allocation strategies were developed. Despite its limitations, contiguous allocation helps in understanding the basics of memory management. It forms the foundation for more advanced techniques like paging and segmentation.

Explanation

In contiguous memory allocation, each process is assigned a single block of memory large enough to hold its entire execution image, including code, data, stack, and heap.

The operating system maintains information about free memory blocks using data structures such as a free list. When a process arrives, the OS searches the free list to find a hole large enough to satisfy the request. Several allocation strategies are used to select a suitable memory hole.

First Fit allocates the first hole that is large enough.

Best Fit allocates the smallest hole that is sufficient, reducing leftover space.

Worst Fit allocates the largest available hole, leaving sizable free blocks behind.

Over time, as processes are loaded and removed, memory becomes fragmented.

External fragmentation occurs when free memory is split into many small holes scattered throughout memory. Even if the total free memory is sufficient, a process may not be allocated because no single block is large enough.

Another problem is **internal fragmentation**, where allocated memory may be slightly larger than required, leaving unused space within the allocated block. To reduce fragmentation, the OS may use **compaction**, which shifts processes in memory to combine scattered free spaces into one large block. However, compaction is expensive and requires dynamic relocation. Contiguous allocation also relies on protection mechanisms such as base and limit registers. These registers ensure that a process accesses only its allocated memory region. While contiguous allocation is simple and efficient for small systems, it is not suitable for modern large-scale multiprogramming environments.

DEPARTMENT OF COMPUTER APPLICATIONS

Example

- Consider a system with 100 MB of main memory available for user processes.
- Process P1 requires 20 MB, P2 requires 30 MB, and P3 requires 25 MB.
- These processes are loaded sequentially into memory in contiguous blocks.
- When P2 terminates, a 30 MB hole is created between P1 and P3.
- If a new process P4 requires 35 MB, it cannot be allocated despite sufficient total free memory.
- This situation demonstrates external fragmentation.
- Using compaction could solve the problem, but it introduces overhead.

Conclusion

Contiguous memory allocation is one of the simplest memory-management techniques. It allows fast memory access and easy implementation. However, fragmentation and inefficient memory utilization limit its usefulness. Modern operating systems prefer paging and segmentation to overcome these drawbacks. Nevertheless, understanding contiguous allocation is essential for grasping advanced memory-management concepts.

PAGING

Definition

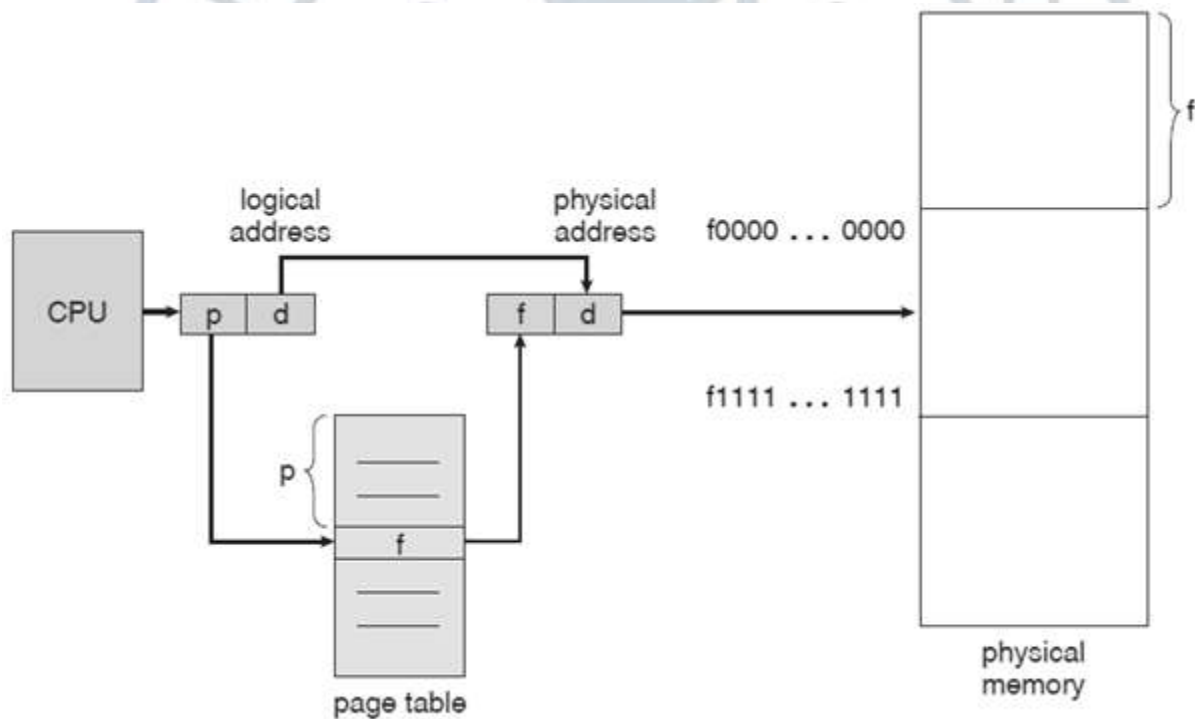
Paging is a memory-management technique in which logical memory is divided into fixed-size blocks called pages and physical memory is divided into fixed-size blocks called frames. The operating system maps pages to frames using a page table. Paging allows processes to be stored non-contiguously in physical memory. It eliminates the problem of external fragmentation.

Introduction

DEPARTMENT OF COMPUTER APPLICATIONS

Paging was introduced to overcome the limitations of contiguous memory allocation. In contiguous allocation, a process requires a single continuous block of memory, which leads to fragmentation problems. Paging removes this requirement by allowing a process to be split into fixed-size pages. These pages can be placed anywhere in available physical memory frames. The operating system uses a page table to maintain the mapping between pages and frames. Paging simplifies memory allocation and improves memory utilization. It allows multiple processes to coexist efficiently in main memory. Paging is widely used in modern operating systems such as Linux and Windows. It forms the foundation for advanced concepts like virtual memory and demand paging. Thus, paging is a core memory-management technique in modern systems.

Diagram (Paging Concept)



Explanation

DEPARTMENT OF COMPUTER APPLICATIONS

In a paging system, the logical address generated by the CPU is divided into two parts: the page number and the page offset. The page number is used as an index into the page table, which contains the frame number corresponding to that page. The frame number, combined with the offset, forms the physical address. Paging allows a process's pages to be scattered throughout physical memory. Because all pages and frames are of the same size, paging avoids external fragmentation entirely. However, paging may still suffer from internal fragmentation when a page is not fully utilized.

The page table is maintained by the operating system for each process. To improve performance, a special hardware cache called the **Translation Lookaside Buffer (TLB)** stores recent page-to-frame translations. This significantly reduces memory-access time. Paging supports protection and sharing. Page-table entries include protection bits such as read, write, and execute permissions. Shared pages allow multiple processes to share common code, such as system libraries.

Paging also simplifies process swapping because pages can be moved individually rather than as a whole process. It enables virtual memory, allowing programs to execute even when not fully loaded into memory. However, paging introduces overhead due to address translation and page-table maintenance.

Modern systems use multi-level page tables to handle large address spaces efficiently. Overall, paging provides a flexible and efficient solution to memory management.

Example

- Assume a system with a page size of 4 KB.
- A logical address of 16,500 bytes is generated by the CPU.
- The page number is calculated as $16,500 \div 4,096 = 4$.
- The offset is $16,500 \bmod 4,096 = 116$.
- The page table maps page 4 to frame 12.
- Thus, the physical address is frame 12 with offset 116.
- This address is then accessed in main memory.

DEPARTMENT OF COMPUTER APPLICATIONS

Conclusion

Paging is an efficient memory-management technique that eliminates external fragmentation. It allows non-contiguous allocation of process memory. Paging supports protection, sharing, and virtual memory implementation. Despite some overhead, it is widely used in modern operating systems. Understanding paging is essential for studying advanced memory-management concepts.

STRUCTURE OF PAGE TABLE

Definition

The page table is a data structure used by the operating system to map logical page numbers to physical frame numbers. Each process has its own page table maintained by the operating system. It enables address translation from logical to physical memory. The structure of the page table directly affects memory access efficiency.

Introduction

Paging requires a mechanism to keep track of where each page of a process is stored in physical memory. This mechanism is provided by the page table. As programs and address spaces grew larger, simple page tables became inefficient. Large page tables consume significant memory and increase access time. To address this problem, different page-table structures were developed. Modern operating systems use advanced page-table organizations to improve performance. These structures reduce memory overhead and speed up address translation. Hardware support plays a crucial role in page-table management. The design of the page table is closely linked to system architecture. Understanding page-table structures is essential for understanding virtual memory systems.

DEPARTMENT OF COMPUTER APPLICATIONS

Logical Page No	Frame No
0	14
1	6
2	20
3	43

Explanation

A page table consists of a set of page-table entries (PTEs), each corresponding to a page in the logical address space. Each entry contains the frame number where the page is stored, along with control bits. Common control bits include valid/invalid bit, protection bits, reference bit, and dirty bit.

The simplest structure is the **single-level page table**, where each page number directly indexes the table. However, for large address spaces, single-level page tables become very large and inefficient.

To overcome this, **multi-level page tables** are used. In this structure, the page table is broken into smaller tables, reducing memory usage. Only the required portions of the page table are kept in memory.

Another structure is the **hashed page table**, commonly used for large address spaces such as 64-bit systems. Here, the virtual page number is hashed to a location in a hash table. This structure improves performance for sparse address spaces.

The **inverted page table** stores one entry per physical frame instead of per virtual page. Each entry contains the process ID and page number currently using that frame. This greatly reduces memory overhead but increases lookup complexity.

To speed up address translation, most systems use a **Translation Lookaside Buffer (TLB)**. The TLB is a small, fast cache that stores recent page-table entries. If a TLB hit occurs, the frame number is obtained quickly. On a TLB miss, the system accesses the page table in memory. Each structure

DEPARTMENT OF COMPUTER APPLICATIONS

involves trade-offs between memory usage, speed, and complexity. The operating system chooses the most suitable structure based on system requirements.

Example

- Consider a 32-bit system with a page size of 4 KB.
- The logical address is divided into a 20-bit page number and a 12-bit offset.
- A single-level page table would require 2^{20} entries per process.
- Using a two-level page table significantly reduces memory usage.
- Only the necessary second-level tables are allocated.
- This makes address translation more efficient and scalable.

Conclusion

The structure of the page table is vital for efficient memory management. Different page-table organizations address the challenges of large address spaces. Advanced structures reduce memory overhead and improve translation speed. The use of TLBs further enhances performance.

Page-table design is a key factor in modern operating systems.

SEGMENTATION

Definition

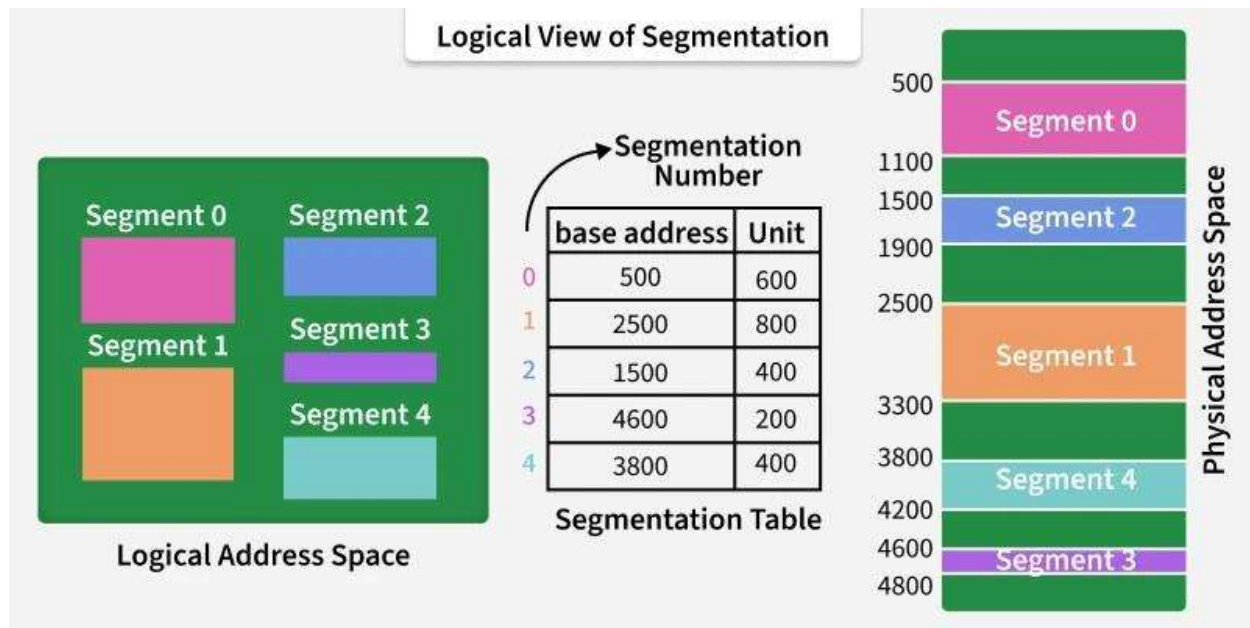
Segmentation is a memory-management technique that divides a program into variable-sized logical units called segments. Each segment represents a logical part of a program such as code, data, stack, or heap. Segmentation supports the programmer's view of memory organization. It provides both memory protection and sharing at the segment level.

Introduction

Paging divides memory into fixed-size units, which may not match the logical structure of programs. Segmentation was introduced to reflect the logical divisions within a program. Programs are naturally composed of different sections, each with different sizes and access requirements. Segmentation allows these sections to be treated as separate entities. Each segment has its own length and protection attributes. The operating system maintains a segment table to manage segments. Logical addresses in segmentation consist of a segment number and an offset. Segmentation enables fine-grained memory protection and sharing. It is particularly useful for modular programming. However, segmentation can suffer from external fragmentation.

Diagram (Segmentation Concept)

DEPARTMENT OF COMPUTER APPLICATIONS



Explanation

In segmentation, a program is divided into multiple segments, each representing a logical unit.

The operating system maintains a **segment table** for each process. Each segment-table entry contains a base address and a limit. The base specifies the starting physical address of the segment. The limit specifies the length of the segment.

When the CPU generates a logical address, it consists of two parts: the segment number and the offset. The segment number is used to index the segment table. The offset is checked against the segment limit to ensure protection. If the offset exceeds the limit, a segmentation fault occurs. Segmentation allows different protection levels for different segments.

For example, code segments can be marked read-only, while data segments allow read and write access. Segmentation also supports sharing by allowing multiple processes to share the same segment. Shared libraries are commonly implemented using segmentation.

DEPARTMENT OF COMPUTER APPLICATIONS

However, because segments are of variable size, segmentation can suffer from external fragmentation. As segments are allocated and deallocated, free memory may become scattered. Compaction may be required to combine free spaces, which introduces overhead.

Segmentation provides a natural and flexible memory model. Some systems combine segmentation with paging to eliminate fragmentation while preserving logical structure. In such systems, each segment is divided into pages. This approach combines the advantages of both techniques.

Overall, segmentation emphasizes logical memory organization rather than physical placement.

Example

Consider a program with four segments: code, data, stack, and heap. The code segment is read-only and shared among multiple processes. The data segment is writable and private to each process. When the program accesses a variable, the CPU generates a logical address with a segment number and offset. The segment table provides the base and limit. If the offset is within the limit, the physical address is computed. Otherwise, a segmentation fault occurs.

Conclusion

Segmentation provides a logical and flexible view of memory management. It allows protection and sharing at a logical level. However, external fragmentation limits its efficiency. Modern systems often combine segmentation with paging. Understanding segmentation is essential for advanced memory-management concepts.

VIRTUAL MEMORY MANAGEMENT

(BACKGROUND)

Definition

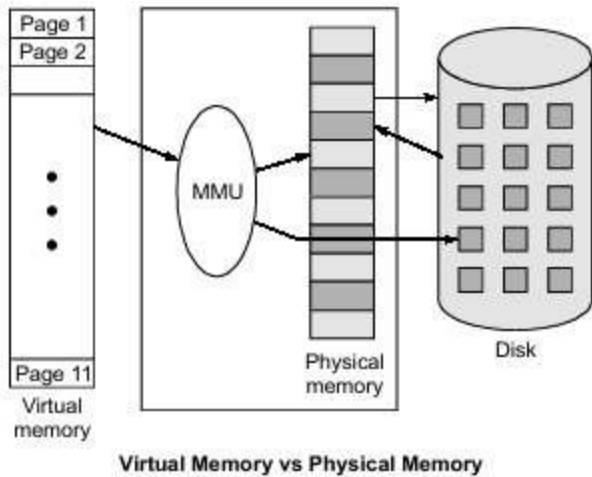
Virtual memory is a memory-management technique that allows programs to execute without being fully loaded into physical memory. It provides an abstraction of a large, contiguous address space to each process. Virtual memory separates logical memory from physical memory. It enables efficient use of main memory and supports multiprogramming.

Introduction

Modern programs often require more memory than is physically available in main memory. Early systems could not execute such programs due to limited RAM. Virtual memory was introduced to overcome this limitation. It allows a program to be partially loaded into memory while the remaining parts reside on secondary storage. The operating system loads only the required portions of a program when needed. This technique improves memory utilization and system performance. Virtual memory also allows multiple processes to share memory efficiently. It increases the degree of multiprogramming. Processes can execute independently with the illusion of having a large memory space. Thus, virtual memory plays a vital role in modern operating systems.

Diagram (Virtual Memory Concept)

DEPARTMENT OF COMPUTER APPLICATIONS



Explanation

Virtual memory works by separating the logical address space of a process from the actual physical memory. The logical address space is the range of addresses a process believes it can access. Physical memory, however, is limited and shared among multiple processes.

The operating system uses hardware support such as the **Memory Management Unit (MMU)** to translate logical addresses into physical addresses. If the required data is not present in main memory, a **page fault** occurs. The operating system then retrieves the required data from secondary storage.

Virtual memory allows only a portion of a program to be resident in memory at any given time. This reduces memory requirements and allows more programs to run concurrently. It also enables efficient CPU utilization by reducing idle time.

One major advantage of virtual memory is the ability to run programs larger than physical memory. Another advantage is increased memory utilization because unused pages do not occupy RAM. Virtual memory also simplifies program loading and linking.

DEPARTMENT OF COMPUTER APPLICATIONS

Virtual memory systems typically use **paging** as the underlying mechanism. Demand paging loads pages only when they are accessed. This avoids unnecessary memory allocation.

However, virtual memory introduces overhead due to page-table maintenance and disk access. If not managed properly, it can lead to excessive page faults. This may result in performance degradation.

Despite these challenges, virtual memory is essential for modern operating systems. It provides flexibility, efficiency, and scalability.

Example

Consider a system with 4 GB of RAM and a program that requires 8 GB of memory. Without virtual memory, the program cannot execute. With virtual memory, only the active portions of the program are loaded into RAM. The remaining parts are stored on disk. When the program accesses a missing portion, a page fault occurs. The OS loads the required page into memory. Thus, the program runs successfully despite limited RAM.

Conclusion

Virtual memory is a powerful memory-management technique that overcomes physical memory limitations. It improves system performance and resource utilization. By separating logical and physical memory, it enables large and complex applications to run efficiently. Virtual memory forms the basis for demand paging and page replacement techniques. It is a fundamental concept in modern operating systems.

DEMAND PAGING

Definition

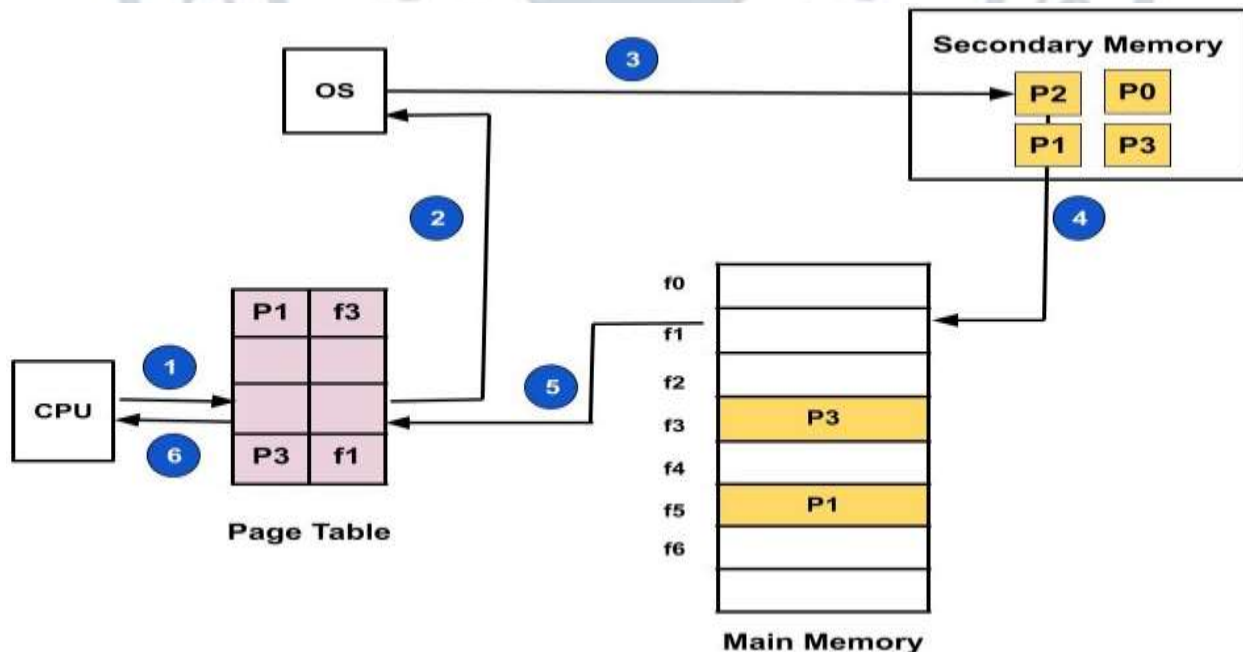
DEPARTMENT OF COMPUTER APPLICATIONS

Demand paging is a virtual memory technique in which pages are loaded into main memory only when they are required. Pages are brought from secondary storage to memory upon a page fault. This approach avoids loading unused pages into memory. Demand paging improves memory utilization and system efficiency.

Introduction

In early systems, entire programs were loaded into memory before execution. This approach wasted memory when large portions of programs were never used. Demand paging was introduced to solve this inefficiency. It allows a program to start execution with only a few pages in memory. Additional pages are loaded dynamically as they are accessed. This technique significantly reduces memory requirements. Demand paging enables higher levels of multiprogramming. It allows the system to run more processes simultaneously. The technique relies on the principle of locality. Demand paging is widely used in modern operating systems.

Diagram (Demand Paging Operation)



DEPARTMENT OF COMPUTER APPLICATIONS

Explanation

In demand paging, the operating system initially loads only a subset of a process's pages into memory. Each page-table entry includes a valid/invalid bit. If the bit indicates invalid, the page is not currently in memory. When the CPU references such a page, a page fault occurs. The operating system then suspends the process and invokes the page fault handler. The handler determines the location of the page on secondary storage. A free frame is selected, and the page is loaded into that frame. The page table is updated to mark the page as valid. The instruction that caused the fault is then restarted.

Demand paging relies heavily on the concept of locality of reference. Programs tend to access a small subset of their pages repeatedly. Thus, most page references are satisfied from memory.

The performance of demand paging depends on the page-fault rate. A low page-fault rate results in efficient execution. A high page-fault rate increases disk I/O and slows the system.

Demand paging requires hardware support, including page tables and MMU. It also needs fast secondary storage. Page replacement algorithms play an important role when memory is full. Demand paging allows efficient use of memory resources. It also simplifies program execution by eliminating unnecessary memory allocation.

Example

- Consider a program with 100 pages.
- Only the first 10 pages are loaded initially.
- When the program accesses page 15, a page fault occurs.
- The operating system loads page 15 from disk into memory.
- The program then continues execution.
- Unused pages remain on disk.
- This reduces memory usage and improves efficiency.

DEPARTMENT OF COMPUTER APPLICATIONS

Conclusion

Demand paging is a key technique in virtual memory systems. It loads pages only when needed, conserving memory. This approach improves system performance and scalability. Demand paging works effectively with locality of reference. It is essential for modern operating systems.

COPY-ON-WRITE

Definition

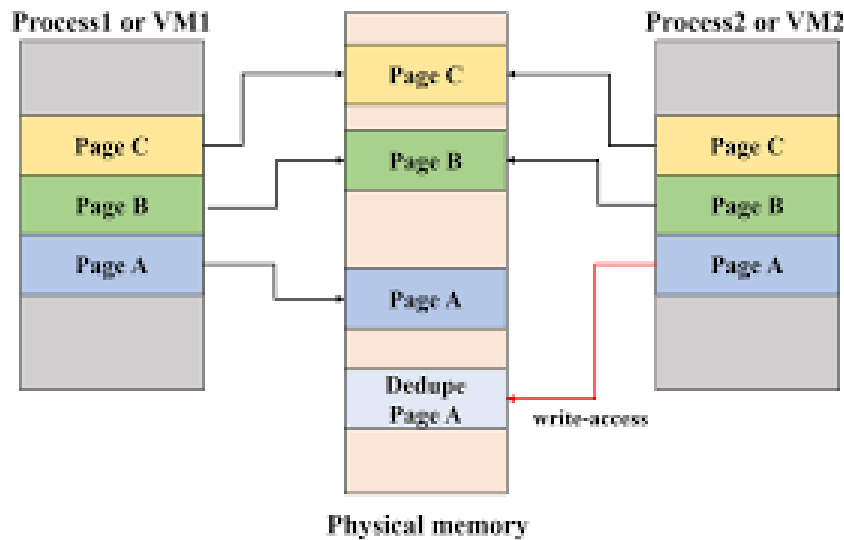
Copy-on-Write is a memory-management optimization technique used in virtual memory systems. It allows multiple processes to share the same physical memory pages initially. A page is copied only when a process attempts to modify it. This technique reduces memory usage and improves process creation efficiency.

Introduction

In multiprogramming environments, processes often share common code and data. Creating separate copies of memory pages for each process can waste memory. Copy-on-Write was introduced to optimize memory usage in such situations. It is commonly used during process creation, especially with the `fork()` system call. Initially, both parent and child processes share the same memory pages. These shared pages are marked as read-only. As long as neither process modifies the page, no duplication occurs. When a write operation is attempted, the OS creates a private copy. This mechanism significantly improves system performance. Copy-on-Write is widely used in modern operating systems like Linux.

Diagram (Copy-on-Write Mechanism)

DEPARTMENT OF COMPUTER APPLICATIONS



Explanation

Copy-on-Write works by delaying the duplication of memory pages until modification is required. When a process is created using `fork()`, the child process initially shares the parent's address space. Both processes reference the same physical pages. The operating system marks these pages as read-only.

If either process attempts to write to a shared page, a page fault occurs. The OS identifies that the fault is due to a write operation on a read-only shared page. The OS then allocates a new physical frame. The contents of the original page are copied into the new frame. The page table of the writing process is updated to point to the new page. The original shared page remains unchanged for the other process.

This approach avoids unnecessary copying of pages that may never be modified. It greatly reduces memory overhead during process creation. Copy-on-Write also reduces process startup time.

It is especially effective when the child process immediately executes a new program using `exec()`. COW requires hardware support for page protection. The page-table entries include protection bits to detect write attempts. The MMU generates a page fault when a write is attempted on a shared page.

DEPARTMENT OF COMPUTER APPLICATIONS

Copy-on-Write improves overall system efficiency. However, it introduces slight overhead during the first write operation. Despite this, its benefits outweigh the costs. It is an essential optimization in modern virtual memory systems.

Example

- A parent process creates a child process using `fork()`.
- Both processes share the same code and data pages.
- Neither process modifies the memory initially.
- No extra memory is used.
- If the child modifies a variable, a page fault occurs.
- The OS creates a copy of that page for the child.
- The parent continues using the original page.
- Thus, memory is duplicated only when necessary.

Conclusion

Copy-on-Write is an efficient memory-management optimization technique. It minimizes memory duplication and speeds up process creation. By copying pages only when modified, it conserves system resources. COW is widely used in modern operating systems. It plays a critical role in efficient virtual memory management.

PAGE REPLACEMENT

Definition

Page replacement is a memory-management technique used when a page fault occurs and no free frame is available in memory. The operating system selects a page to remove from memory to make space for

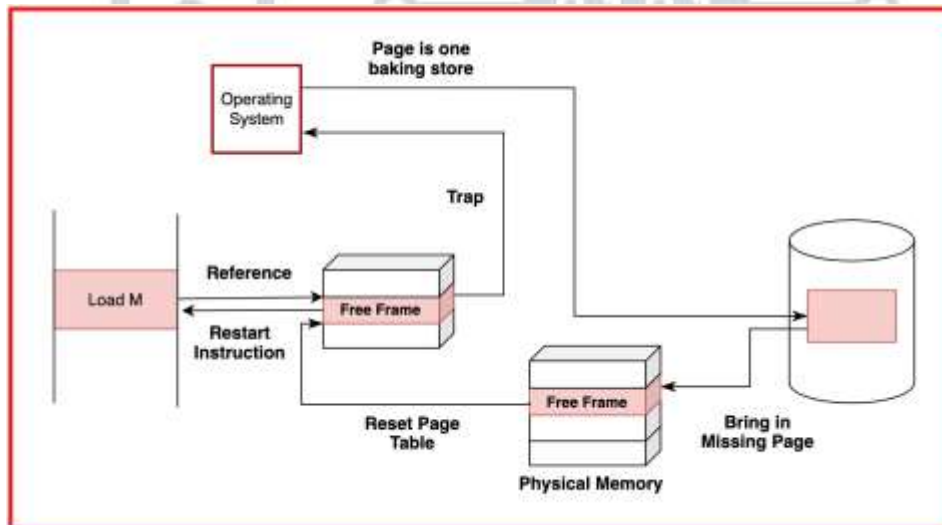
DEPARTMENT OF COMPUTER APPLICATIONS

the required page. The goal is to minimize the number of page faults. Page replacement is a key component of virtual memory systems.

Introduction

In a demand-paged system, pages are loaded into memory only when required. As more pages are accessed, physical memory may become full. When a page fault occurs and no free frames are available, the OS must replace an existing page. Choosing which page to replace is critical for system performance. An inefficient replacement decision can cause frequent page faults. Page replacement algorithms are designed to make optimal decisions. These algorithms attempt to predict future page usage. They are evaluated based on their page-fault rate. Different algorithms are suitable for different workloads. Page replacement plays a crucial role in overall system efficiency.

Diagram (Page Replacement Scenario)



Explanation

DEPARTMENT OF COMPUTER APPLICATIONS

Page replacement occurs when a page fault happens and memory is already full. The operating system must choose one page in memory to be replaced. If the selected page has been modified, it must be written back to disk before replacement.

The performance of page replacement depends on the chosen algorithm. The **Optimal Page Replacement** algorithm replaces the page that will not be used for the longest time in the future. Although it provides the lowest page-fault rate, it is not implementable in practice.

First-In First-Out (FIFO) replaces the oldest page in memory. FIFO is simple but may suffer from Belady's anomaly.

Least Recently Used (LRU) replaces the page that has not been used for the longest time. It performs better than FIFO but requires additional overhead.

Second-Chance Algorithm improves FIFO by considering a reference bit. Pages with a reference bit set are given a second chance.

Clock Algorithm is an efficient implementation of the second-chance algorithm. It uses a circular list of frames and a pointer.

Page replacement algorithms must balance performance and implementation complexity. They rely on hardware support such as reference and dirty bits. Local replacement affects only the faulting process. Global replacement allows any process's pages to be replaced. Effective page replacement reduces page faults and improves system throughput.

Example

- Consider a system with three frames and page reference string:
- 7, 0, 1, 2, 0, 3, 0, 4.
- Using FIFO, page 7 is replaced first, followed by 0 and 1.
- This results in several page faults.

DEPARTMENT OF COMPUTER APPLICATIONS

- Using LRU, the least recently used page is replaced.
- This typically results in fewer page faults.
- Thus, algorithm choice impacts performance.

Conclusion

Page replacement is essential in virtual memory systems. The choice of replacement algorithm significantly affects performance. Efficient algorithms minimize page faults and disk access. Page replacement enables systems to run large programs efficiently. It is a core component of modern operating systems.

ALLOCATION OF FRAMES

Definition

Allocation of frames refers to the method by which physical memory frames are distributed among competing processes. It determines how many frames each process is allowed to use. Proper frame allocation ensures efficient memory utilization and system performance. It is a critical aspect of virtual memory management.

Introduction

In a multiprogramming environment, multiple processes share limited physical memory. Each process requires a certain number of frames to execute efficiently. If too few frames are allocated, frequent page faults occur. If too many frames are allocated, memory is wasted. The operating system must decide how to divide available frames among processes. Different allocation strategies have been developed to handle this problem. These strategies aim to balance fairness, efficiency, and performance. Frame allocation also affects page replacement behavior. The choice of allocation policy impacts overall system throughput. Thus, frame allocation plays a vital role in memory management.

DEPARTMENT OF COMPUTER APPLICATIONS

Explanation

Frame allocation determines how many physical frames are assigned to each process. The simplest method is **equal allocation**, where each process receives the same number of frames. This method is easy to implement but does not consider process size. **Proportional allocation** assigns frames based on the size of each process. Larger processes receive more frames than smaller ones. This approach provides better performance for memory-intensive programs. **Priority allocation** assigns frames based on process priority. Higher-priority processes receive more frames. Lower-priority processes may suffer more page faults. Another important concept is **minimum number of frames**. Each process requires a minimum number of frames to execute correctly. This minimum depends on the instruction set architecture.

Frame allocation is also related to page replacement scope. In **global replacement**, a process can replace pages belonging to other processes. This improves overall throughput but may cause unfairness. In **local replacement**, a process can replace only its own pages. This provides fairness but may reduce system efficiency.

The OS must also consider the total number of available frames. If too many processes are active, frame allocation becomes tight. This can lead to thrashing.

Efficient frame allocation helps reduce page faults and improve CPU utilization. It works closely with page replacement algorithms. Together, they ensure optimal memory performance.

Example

- Assume a system has 100 frames and three processes.
- Process P1 requires 40 pages, P2 requires 30 pages, and P3 requires 30 pages.
- Using equal allocation, each process receives about 33 frames.
- Using proportional allocation, frames are distributed as 40, 30, and 30 respectively.

DEPARTMENT OF COMPUTER APPLICATIONS

- Proportional allocation reduces page faults for larger processes. Thus, allocation strategy affects system performance.

Conclusion

Allocation of frames is a fundamental component of virtual memory management. It determines how efficiently physical memory is shared among processes. Different allocation strategies provide different trade-offs. Proper frame allocation minimizes page faults and improves system performance. It works together with page replacement to ensure efficient memory utilization.

THRASHING

Definition

Thrashing is a condition in a virtual memory system where the CPU spends most of its time handling page faults rather than executing processes. It occurs when processes do not have enough frames to support their working sets. Thrashing leads to excessive paging activity and poor system performance. It is a serious problem in memory-management systems.

Introduction

In a multiprogramming environment, several processes compete for limited physical memory. When too many processes are active, each process may receive too few frames. As a result, pages required for execution are frequently not present in memory. This causes a high rate of page faults. Each page fault requires disk access, which is slow compared to memory access. The CPU remains idle while waiting for pages to be loaded from disk. As CPU utilization drops, the operating system may mistakenly increase multiprogramming. This further worsens the situation. The system becomes overloaded with paging operations. This condition is known as thrashing.

Explanation

DEPARTMENT OF COMPUTER APPLICATIONS

Thrashing occurs when the working set of a process exceeds the number of frames allocated to it. The **working set** is the set of pages a process needs to execute efficiently. If these pages are not resident in memory, page faults occur frequently. When page faults increase, the operating system spends more time swapping pages between disk and memory. Disk I/O increases dramatically, slowing down the system. The CPU remains underutilized because it waits for page transfers.

A common cause of thrashing is excessive multiprogramming. When the OS loads too many processes into memory, frame allocation per process becomes insufficient. Another cause is poor frame allocation and page replacement policies.

Thrashing can be detected by monitoring the page-fault rate. A sudden increase in page faults accompanied by low CPU utilization indicates thrashing.

Several techniques are used to prevent thrashing. The **Working Set Model** ensures that each process is allocated enough frames to hold its working set. If sufficient frames are not available, the process is suspended.

The **Page Fault Frequency (PFF)** approach monitors page-fault rates. If the rate is too high, the OS allocates more frames or reduces multiprogramming.

Using **local page replacement** also helps reduce thrashing by preventing processes from stealing frames from others. Reducing the degree of multiprogramming is the most effective solution. Thrashing severely degrades performance and must be avoided for system stability.

Example

Consider a system with 10 frames and four processes. Each process requires at least 4 frames to execute efficiently. The total required frames become 16, exceeding available memory. As a result, processes continuously replace each other's pages. Page faults occur frequently. Disk I/O increases sharply. CPU utilization drops close to zero. The system enters a thrashing state.

DEPARTMENT OF COMPUTER APPLICATIONS

Conclusion

Thrashing is a critical problem in virtual memory systems. It occurs when processes are allocated insufficient memory frames. The system becomes dominated by paging activity rather than useful work. Proper frame allocation and controlled multiprogramming prevent thrashing. Working set and page-fault frequency models help maintain system stability.

