Memory Management

OPERATING SYSTEMS

Sercan Külcü | Operating Systems | 16.04.2023

Contents

Chapter 8:
Memory Management

1 Introduction

Welcome to the chapter on memory management in operating systems! Memory management is a crucial component of any operating system as it involves the management of a system's primary memory. This chapter will provide a detailed understanding of memory management, its definition, and the reasons why it is significant.

The chapter will start with an overview of the goals of the chapter, followed by a discussion of memory management, its importance, and how it impacts the overall performance of an operating system. We will also discuss the different types of memory and their roles in the memory management process. Additionally, we will delve into the memory hierarchy and how it affects the performance of an operating system.

By the end of this chapter, you will have a comprehensive understanding of memory management in operating systems and its crucial role in ensuring optimal system performance. So, let's dive in!

1.1 Definition and importance of memory management

Memory management is a crucial part of operating systems that deals with the management of computer memory. Memory management is responsible for the efficient and effective allocation and de-allocation of memory to processes and programs. The memory management subsystem of an operating system must ensure that the right process

gets the required amount of memory at the right time. This chapter will introduce the concept of memory management and discuss its importance in operating systems.

Memory management is the process of controlling and coordinating the use of computer memory to allow the efficient execution of programs and the sharing of memory among multiple processes. The main tasks of memory management include allocation, deallocation, protection, and sharing of memory.

Memory management is essential in operating systems for several reasons. The following are some of the key reasons why memory management is critical:

- Efficient Use of Memory: Memory management is essential for the efficient use of memory. It ensures that memory is allocated only to those processes that need it, and the unused memory is released to other processes that require it. This helps in maximizing the available memory resources and improving the overall performance of the system.
- Protection of Memory: Memory management is crucial for the protection of memory from unauthorized access. It ensures that each process can only access the memory allocated to it and not interfere with other processes' memory space. This helps in preventing programs from corrupting each other's data or code.
- Sharing of Memory: Memory management is essential for enabling the sharing of memory among multiple processes. Shared memory allows multiple processes to access the same memory area, which can improve communication and coordination among the processes.
- Virtual Memory: Memory management is also responsible for the implementation of virtual memory, which is a technique used to provide the illusion of a larger main memory than is physically available. Virtual memory enables programs to use more memory

than is physically present by swapping parts of the program between the main memory and the hard disk.

Memory management is a crucial part of operating systems that ensures the efficient and effective use of memory resources. The importance of memory management cannot be overstated, as it plays a significant role in the overall performance and reliability of the system. In the following chapters, we will explore various memory management techniques and strategies used by operating systems to manage memory effectively.

1.2 Overview of the goals of the chapter

Memory management is a fundamental concept in operating systems that involves the management of the computer's primary memory. The primary memory is a volatile storage area that temporarily stores the data and instructions that are being processed by the CPU. This chapter will provide an overview of the goals of memory management and discuss the different techniques and strategies employed by operating systems to achieve these goals.

The primary goals of memory management are to provide a convenient and efficient way for processes to access and use the system's memory, while also ensuring that the memory is utilized in the most optimal manner possible. Achieving these goals requires careful planning and coordination by the operating system, which must maintain an accurate record of which processes are using which parts of the memory at any given time.

To achieve these goals, the operating system employs various techniques such as memory allocation, memory protection, memory sharing, and memory compaction. Memory allocation is the process of reserving memory space for a process to use. Memory protection is the mechanism that ensures that a process can only access memory areas that have been allocated to it. Memory sharing is the technique that

allows multiple processes to share a single memory region. Finally, memory compaction is the process of eliminating fragmentation in the memory, which can occur when the memory is allocated and deallocated frequently.

This chapter will examine each of these techniques in more detail, as well as explore the advantages and disadvantages of each approach. It will also discuss the different memory allocation strategies such as paging and segmentation, and their impact on memory management. Additionally, this chapter will describe various memory management algorithms, such as page replacement algorithms, and their performance characteristics.

Overall, the goals of memory management are to ensure that the memory is utilized efficiently, that processes have access to the memory they require, and that the memory is protected from unauthorized access. The techniques and strategies employed by operating systems to achieve these goals are constantly evolving and improving as technology advances, and it is essential for operating systems developers to keep abreast of these developments to ensure that their systems remain efficient, reliable, and secure.

1.3 Background

When you run a program on your computer, it is first brought from the disk into the main memory and then placed within a process. The CPU can only directly access the main memory and registers. Therefore, the program must be loaded into the main memory for the CPU to execute it.

The memory unit only sees a stream of addresses and read or write requests. It does not know anything about the content of the data being transferred. This means that the CPU has to issue an address and a request type to access a specific location in the memory.

Register access is incredibly fast and can be accomplished in one CPU clock cycle or less. On the other hand, accessing the main memory can take many cycles, which can cause a stall in the CPU. This delay can be a significant problem in high-performance computing.

To overcome this issue, the cache sits between the main memory and CPU registers. The cache is a small amount of high-speed memory that holds frequently accessed data. When the CPU needs data that is not currently in the registers, it checks the cache. If the data is present in the cache, it can be quickly accessed, avoiding the need to access the main memory. This process is called caching.

Protection of memory is crucial to ensure correct operation. Without memory protection, one program could access the memory of another program and corrupt its data, causing unexpected results. This can be especially problematic in a multi-user system where multiple users are running programs simultaneously. To prevent this from happening, the operating system implements memory protection mechanisms that ensure that each program can only access its allocated memory.

In conclusion, bringing a program from the disk into memory and placing it within a process for execution is a critical process in the operation of an operating system. Main memory and registers are the only storage that the CPU can access directly. The cache sits between the main memory and the CPU registers to improve performance, and memory protection is essential to ensure correct operation.

2 No memory abstraction

Before the advent of memory abstraction, computer programs simply saw physical memory as a continuous set of addresses, each corresponding to a cell containing a certain number of bits. When a program executed an instruction to move data from one location to

another, the computer simply moved the contents of the specified physical memory location to the destination register.

While this model may seem straightforward, it has several drawbacks. First and foremost, it exposes programs to low-level details of the underlying hardware, making them harder to write and debug. Programs that rely on physical memory addresses can be hard to port to different hardware platforms or operating systems, and they may break if the physical layout of memory changes.

Moreover, physical memory can be shared among multiple programs running concurrently, which can lead to conflicts and data corruption. Without a memory abstraction layer to manage access to memory, programs can inadvertently overwrite each other's data, leading to unpredictable results.

To address these issues, modern operating systems provide a variety of memory abstractions, such as virtual memory, memory protection, and memory allocation. These abstractions hide the underlying physical memory layout from programs, provide isolation and protection between different programs, and allow programs to allocate and release memory dynamically as needed.

2.1 Running multiple programs

In the absence of memory abstraction, running multiple programs on a computer might seem impossible. However, it is possible to run multiple programs sequentially with the help of swapping. Swapping refers to the process of saving the entire contents of memory to a disk file, then bringing in and running the next program.

When a program is run, it occupies a certain amount of physical memory. When another program needs to be run, the operating system saves the current program's memory to disk and loads the new program's memory into physical memory. This allows the computer to

appear as though it is running multiple programs simultaneously, even though only one program is actually in memory at any given time.

Swapping is not an ideal solution, as it can be slow and inefficient. However, in the absence of memory abstraction, it is the only way to run multiple programs on a computer. As computer technology advanced, memory abstraction was introduced, allowing for more efficient and streamlined management of multiple programs running in memory simultaneously.

2.2 Static relocation

The core problem with the no-memory-abstraction approach is that two programs may reference the same physical memory, leading to conflicts and unpredictable behavior. What we want instead is for each program to have its own private set of addresses that are local to it. This can be achieved through the use of memory abstraction.

Memory abstraction allows programs to reference logical memory addresses that are mapped to physical memory addresses by the operating system. This mapping is done on a per-process basis, meaning that each process has its own private set of logical addresses that are mapped to physical memory.

One early solution to the problem of running multiple programs without memory abstraction was the IBM 360's static relocation technique. When loading a program into memory, the operating system would modify the program on the fly by adjusting its memory references. This allowed the program to reference logical memory addresses instead of physical memory addresses, but required extra processing time and added complexity to the loading process.

Modern operating systems use more sophisticated memory abstraction techniques that are more efficient and transparent to the programmer. For example, virtual memory allows each process to have its own virtual

address space that is mapped to physical memory by the operating system. This allows programs to reference logical memory addresses that are independent of the physical memory layout, making it possible to run multiple programs simultaneously without conflicts.

In conclusion, memory abstraction is a crucial concept in modern operating systems that enables multiple programs to run simultaneously without conflicts. While early computers lacked memory abstraction, techniques such as static relocation were developed to work around this limitation. Today, virtual memory provides a powerful and efficient way to abstract memory and allow multiple programs to run without interference.

In summary, while the lack of memory abstraction may have been acceptable in the early days of computing, modern operating systems provide sophisticated memory abstractions to improve program reliability, portability, and security.

3 The Address Space

The address space is the view of memory that a running program has in the system. It contains all of the memory state of the program, including the code, data, and stack. For example, when you write a program, the instructions that make up the program code have to live somewhere in memory. They are stored in the address space of the program.

In addition to the code, the address space also contains data. This can include variables, arrays, and other data structures that are used by the program. The address space also includes a stack, which is used by the program to keep track of function calls and to allocate local variables.

The address space is an abstraction because the memory used by a program is not necessarily contiguous or physically contiguous in the system. The OS uses virtual memory to map the address space of a program to the physical memory of the system. This allows the OS to

allocate memory dynamically as needed, and to protect the memory of one program from being accessed by another.

When a program accesses memory, it does so using virtual addresses. These addresses are translated by the OS into physical addresses that correspond to locations in the physical memory of the system. The translation process is transparent to the program, which sees only its virtual address space.

Understanding the concept of the address space is crucial for understanding how memory is managed by the operating system. It allows programs to access memory in a way that is independent of the physical memory layout of the system, and it allows the OS to protect the memory of one program from being accessed by another.

3.1 Logical address space

A logical address space is the set of addresses that a process can use to reference its memory. This space is defined by a pair of base and limit registers, which specify the starting address and the size of the memory region that the process can access. The base register contains the starting address of the memory region, and the limit register contains the size of the memory region.

When a CPU generates a memory access in user mode, it must check that the address is within the boundaries defined by the base and limit registers for that particular user. This check ensures that the process does not access memory outside its address space, which can cause the system to crash or behave unpredictably.

The hardware address protection mechanism enforces this check by ensuring that every memory access generated by the CPU is verified against the base and limit registers for that user. If the access falls outside the limits, the CPU will raise an exception, which the operating system will handle.

The hardware address protection mechanism is a critical component of modern operating systems, as it provides a robust and efficient way to protect processes from accessing memory outside their address space. It ensures that each process is isolated from other processes and can only access its allocated memory region, improving system stability and security.

In conclusion, the base and limit registers define the logical address space for each process in the system. The CPU must check every memory access generated in user mode to ensure that it falls within the base and limit registers for that particular user. The hardware address protection mechanism enforces this check and ensures that each process is isolated from other processes and can only access its allocated memory region, improving system stability and security.

3.2 Address binding

When a program is on disk, it is in a state of readiness to be brought into memory for execution. These programs are stored in an input queue, waiting for the operating system to load them into memory. Without support, these programs must be loaded into address oooo, which can be inconvenient for the system, as the first user process physical address always ends up being oooo.

However, there are ways to ensure that this is not always the case. For example, the operating system can use a technique known as base and bounds registers, which allows the programs to be loaded into different regions of memory. This technique defines a range of addresses that a program can access, and the programs can be loaded into any available memory space within that range.

Furthermore, addresses are represented in different ways at different stages of a program's life. For instance, source code addresses are usually

symbolic, representing the name of a variable or a function. These addresses are not specific to a particular memory location.

Once the source code is compiled, the addresses are bound to relocatable addresses, which are relative to the beginning of the module. For example, an address may be expressed as "14 bytes from the beginning of this module." The linker or loader will then bind these relocatable addresses to absolute addresses, such as 74014, which represents the specific memory location.

Each binding maps one address space to another, allowing the operating system to keep track of the different address spaces used by a program throughout its life cycle. This mapping enables the operating system to manage the memory effectively, ensuring that different programs do not conflict with each other.

In conclusion, programs on disk are loaded into memory for execution from an input queue. The use of base and bounds registers allows the programs to be loaded into different regions of memory. Addresses are represented in different ways at different stages of a program's life, starting with symbolic names for variables and functions, moving to relocatable addresses, and finally binding to absolute addresses. Each binding maps one address space to another, allowing the operating system to manage the memory effectively.

3.3 Logical vs physical address spaces

A logical address, also known as a virtual address, is generated by the CPU. It is the address that a program sees when it is running, and it is independent of the physical memory location where the data is stored. The physical address, on the other hand, is the address that is seen by the memory unit. It is the actual location of the data in physical memory.

In compile-time and load-time address-binding schemes, the logical and physical addresses are the same. However, in execution-time

address-binding schemes, the logical and physical addresses differ. At this stage, the operating system maps the logical addresses generated by the program to their corresponding physical addresses.

The logical address space is the set of all logical addresses generated by a program. It represents the entire range of addresses that the program can access, regardless of whether or not they are actually present in physical memory. The physical address space, on the other hand, is the set of all physical addresses generated by a program. It represents the actual range of addresses where the data is stored in physical memory.

The separation of logical and physical address spaces enables the operating system to manage memory more effectively. By using virtual memory, the operating system can provide each process with its own logical address space, regardless of the amount of physical memory available. This allows the operating system to execute multiple programs concurrently, even if they require more memory than is physically available.

In conclusion, the concept of a logical address space that is bound to a separate physical address space is crucial to proper memory management. The logical address is generated by the CPU and is independent of the physical memory location, while the physical address is the actual location of the data in physical memory. The logical address space represents the entire range of addresses that the program can access, while the physical address space represents the actual range of addresses where the data is stored. This separation enables the operating system to manage memory effectively and execute multiple programs concurrently.

3.4 Swapping

As computer systems became more complex and the number of processes running concurrently increased, it became necessary to

develop strategies to deal with memory overload. One such strategy is swapping, which involves bringing each process into memory in its entirety, running it for a period of time, and then putting it back on the disk. When a process is not running, it can be stored on the disk, freeing up memory for other processes to use.

Swapping is a relatively simple approach to managing memory, but it can be slow and inefficient. Every time a process is swapped in or out, there is an overhead associated with reading from or writing to the disk, which can slow down the system. Additionally, if the system is heavily loaded with many processes, there may not be enough disk space to hold all of the swapped-out processes, leading to performance degradation.

To address some of these issues, another approach to memory management, known as paging, was developed. Instead of swapping entire processes in and out of memory, paging divides memory into fixed-sized blocks, known as pages, and maps these pages to corresponding blocks on the disk, known as page frames. When a process needs to access a page that is not currently in memory, the page is brought in from disk and placed in a free page frame.

Paging can be more efficient than swapping, as it allows processes to share pages that are common across multiple processes. It also allows for more fine-grained control over memory allocation, as pages can be allocated and deallocated on demand. However, paging also introduces overhead in the form of page faults, which occur when a process tries to access a page that is not in memory, and must be fetched from disk.

In summary, swapping and paging are two approaches to managing memory overload in computer systems. While swapping is a simple and straightforward approach, it can be slow and inefficient. Paging, on the other hand, allows for more fine-grained control over memory allocation, but introduces additional overhead in the form of page faults. Both approaches have their strengths and weaknesses, and the choice of which approach to use will depend on the specific requirements of the system in question.

3.5 Memory compaction

In a system that uses swapping to deal with memory overload, multiple processes are brought into memory, run for a period, and then moved back to disk to free up memory. This can lead to multiple holes or unused regions in memory where processes were loaded and then swapped out. To optimize memory usage, it is possible to combine all these unused regions into one big hole by moving all the processes down as far as possible. This technique is called memory compaction.

However, memory compaction requires a lot of CPU time and may not be worth the effort on larger machines. For instance, on a 16-GB machine that can copy 8 bytes in 8 nanoseconds, it would take approximately 16 minutes to move all the processes downward. Therefore, memory compaction is usually not performed except in rare cases when the system is running out of memory and a significant amount of memory could be recovered by performing this operation.

In summary, memory compaction is a technique used to optimize memory usage in a system that uses swapping to deal with memory overload. By combining all the unused regions into one big hole, more memory can be made available for use. However, this technique can be time-consuming and is usually only performed in rare cases when the system is running out of memory.

3.6 Managing free memory

Managing free memory is a crucial task for the operating system, especially in dynamic memory allocation. Two common methods for keeping track of memory usage are bitmaps and free lists.

A bitmap is a data structure that consists of a series of bits, where each bit represents a block of memory. If the bit is set to 1, it means that the corresponding block of memory is currently in use; if it is set to 0, the

block of memory is free. Bitmaps are easy to use and require minimal overhead, but they are not very efficient for large memory systems.

A free list is a linked list that contains all the free blocks of memory in the system. Each block of memory has a header that contains the size of the block and a pointer to the next free block. When a process requests memory, the operating system searches the free list for a block that is large enough to satisfy the request. When a block is allocated, it is removed from the free list. When the block is freed, it is added back to the free list. Free lists require more overhead than bitmaps, but they are more efficient for large memory systems.

In Linux, there are specific memory allocators used for managing free memory. The buddy allocator divides memory into power-of-two-sized blocks and maintains a free list for each block size. When a block is allocated, the allocator finds the smallest free block that is large enough to satisfy the request and splits it into two halves. When a block is freed, the allocator merges it with its buddy (the adjacent block of the same size) if it is also free. The slab allocator, on the other hand, allocates and frees fixed-sized memory chunks called slabs. Each slab contains one or more objects of the same size, and the allocator maintains a free list for each slab.

Managing free memory is a complex task, but it is essential for ensuring the efficient use of system resources. By using bitmaps, free lists, or specific memory allocators, the operating system can ensure that memory is allocated and deallocated in an efficient and organized manner.

3.6.1 Memory management with bitmaps

One of the ways to keep track of memory usage is to use bitmaps. In this method, memory is divided into small units, with each unit corresponding to a bit in the bitmap. The bit is set to 0 if the unit is free and 1 if it is occupied.

The bitmap can be stored in memory itself, or in some other data structure that can be accessed quickly. The advantage of using a bitmap is that it is simple and easy to implement. The disadvantage is that it can be inefficient in terms of memory usage, especially if the allocation units are small.

For example, consider a system where memory is divided into 4KB pages, and each page is further divided into 4-byte units. In this case, the bitmap would consist of one bit for each 4-byte unit. If a process requests 1KB of memory, the system would need to find a contiguous block of 256 free units (1KB/4B=256). To do this, the system would search the bitmap for a block of 256 consecutive 0 bits, which indicates that the corresponding units are free.

To allocate the memory, the system would set the corresponding bits in the bitmap to 1, indicating that the units are now occupied. To deallocate the memory, the system would simply set the bits back to 0.

One potential issue with bitmap memory management is fragmentation. Over time, memory may become fragmented, with small free blocks scattered throughout the address space. This can make it difficult to find large contiguous blocks of free memory, even if the total amount of free memory is sufficient.

Overall, while bitmap memory management is simple and straightforward, it may not be the most efficient method for managing memory in all cases. Other methods, such as free lists, may be more suitable for some systems.

3.6.2 Memory management with linked lists

In addition to bitmap-based memory management, another popular technique for keeping track of memory usage is through linked lists. In this method, the operating system maintains a linked list of allocated and free memory segments.

Each memory segment can either contain a process or be an empty hole between two processes. The linked list maintains pointers to the start and end of each segment and allows the operating system to efficiently allocate and deallocate memory as needed.

When a new process needs memory, the operating system searches the linked list for a free segment of the required size. If a suitable segment is found, it is allocated to the new process, and the linked list is updated to reflect the change in memory usage. If no suitable segment is available, the operating system must either wait for a process to release memory or swap an idle process to disk to free up memory.

Deallocating memory is also straightforward with linked lists. When a process terminates, the operating system marks the corresponding memory segment as free and updates the linked list accordingly. If adjacent free segments are found, they can be combined into a larger hole to be used for future memory allocation.

One advantage of using linked lists for memory management is that they can be more memory-efficient than bitmaps. Linked lists only require one pointer per memory segment, whereas bitmaps require a bit for each allocation unit, which can quickly become prohibitively large for large memory systems.

However, linked lists also have some disadvantages. For example, they can be slower than bitmaps for large memory systems because searching for free segments requires traversing the entire linked list. Additionally, fragmentation can be a problem if memory segments are not combined properly, leading to wasted space and reduced memory efficiency.

Overall, linked lists provide a flexible and efficient method for managing memory in an operating system. However, as with any memory management technique, it is important to carefully consider the tradeoffs between efficiency, fragmentation, and complexity when choosing a specific method for a given system.

4 Memory management unit (MMU)

A relocation register is a hardware register that holds a base address for a process. When a process generates an address, the value in the relocation register is added to the address, resulting in a physical address. This scheme allows the operating system to map the logical or virtual addresses generated by a process to physical addresses at runtime.

One advantage of this approach is that the user program deals with logical addresses, and it never sees the real physical addresses. This is because the physical address is generated at runtime by adding the value in the relocation register to the logical address generated by the process. This approach is commonly used in systems with limited memory, such as embedded systems, where the cost of additional hardware to support more complex mapping schemes is prohibitive.

Execution-time binding occurs when a reference is made to a location in memory. When the reference is made, the logical address is bound to a physical address. This binding ensures that the process can access the data it needs in physical memory.

In the MS-DOS operating system running on Intel 80x86 processors, four relocation registers were used to support this simple mapping scheme. This approach is an example of a runtime address-binding method. Other runtime address-binding methods include segment-base addressing and paging.

In conclusion, the hardware device that maps virtual addresses to physical addresses is a crucial component of memory management in an operating system. The simple scheme of adding the value in the relocation register to every address generated by a user process at the time it is sent to memory is an effective approach for systems with limited memory. This approach allows the user program to deal with logical addresses and never see the real physical addresses, while

execution-time binding ensures that the logical address is bound to a physical address.

4.1 Dynamic loading

Dynamic loading is a useful technique in modern operating systems that allows routines or libraries to be loaded from disk only when they are actually needed. This can lead to better memory-space utilization, as unused routines are never loaded, freeing up memory for other processes.

All routines are kept on disk in relocatable load format, which allows them to be loaded into any available memory space. This technique is particularly useful when large amounts of code are needed to handle infrequently occurring cases, as the routines are only loaded when needed, reducing memory usage for the rest of the time.

The operating system does not need to provide any special support for dynamic loading, as it can be implemented through program design. However, the operating system can provide libraries to implement dynamic loading, making it easier for programmers to use this technique.

One important advantage of dynamic loading is that it allows a program to be smaller, as it does not have to include all of the code needed for every possible scenario. This can lead to faster program startup times, as only the necessary code is loaded into memory.

Dynamic loading can also improve system security, as it can prevent malicious code from being loaded into memory until it is actually needed. This can reduce the risk of system vulnerabilities being exploited by attackers.

Overall, dynamic loading is a useful technique for improving memory utilization and program performance in modern operating systems. It is

a powerful tool that can be used by programmers to optimize their programs, and can be supported by the operating system through the use of libraries and other tools.

4.2 Dynamic linking

In operating systems, there are two main methods for linking library code with the main program: static and dynamic linking. Static linking involves the loader combining system libraries and program code into a single binary image, while dynamic linking delays the linking process until the program is executed.

With dynamic linking, a small piece of code called a stub is used to locate the appropriate memory-resident library routine. The stub replaces itself with the address of the routine and executes it. The operating system checks if the routine is in the process's memory address. If it is not in the address space, it is added to the address space.

Dynamic linking is particularly useful for libraries because it allows multiple programs to share a single copy of a library in memory, which reduces the overall memory usage of the system. This technique is also known as shared libraries.

One of the advantages of dynamic linking is that it allows for patching system libraries. Versioning may be needed to ensure that applications continue to work properly after updates to system libraries. Another advantage of dynamic linking is that it can lead to better performance, as only the code that is actually used needs to be loaded into memory.

However, there are also some drawbacks to dynamic linking. One potential problem is that it can make the program slower to start up because it requires more time to locate and load the necessary library routines. Another problem is that it can make the program more vulnerable to security exploits, as malicious code could potentially be

injected into a shared library and executed by any program that uses that library.

Overall, the decision to use static or dynamic linking depends on the specific needs of the application and the operating system. Both methods have their advantages and disadvantages, and it is up to the programmer to decide which method is most appropriate for their particular project.

4.3 Swapping

One of the key features of modern operating systems is the ability to manage memory effectively, even when the total physical memory space of processes exceeds the available physical memory. One technique used to manage this situation is called swapping. In this chapter, we will discuss the concept of swapping and how it allows processes to be temporarily moved out of memory to a backing store, and then brought back into memory for continued execution.

The first step in implementing swapping is to create a fast disk called a backing store that is large enough to hold copies of all memory images for all users. The backing store must provide direct access to these memory images so they can be easily swapped in and out of physical memory. When a process needs to be swapped out of memory, the operating system copies its entire memory image to the backing store, freeing up physical memory for other processes.

When a swapped-out process needs to be brought back into memory for continued execution, the operating system copies its memory image from the backing store back into physical memory. This process is known as "roll in." The amount of time it takes to roll in a process is directly proportional to the size of its memory image.

In priority-based scheduling algorithms, a variant of swapping called "roll out, roll in" is used. When a higher-priority process becomes

available to run, the operating system temporarily moves the lower-priority process out of memory and onto the backing store, freeing up memory for the higher-priority process. The higher-priority process is then loaded into memory and executed. When the higher-priority process is finished, the operating system swaps it out and reloads the lower-priority process back into memory.

The major challenge in swapping is to minimize the total transfer time required to move a process in and out of memory. This transfer time includes both the time required to copy the process image to or from the backing store and the time required to perform any necessary address translation or other housekeeping tasks. The operating system must also maintain a ready queue of processes that are ready to run, but whose memory images are currently stored on the backing store.

In summary, swapping is a powerful memory management technique that allows processes to be temporarily moved out of memory to a backing store, freeing up physical memory for other processes. With the help of a fast and large enough backing store, the total physical memory space of processes can exceed the available physical memory. By maintaining a ready queue of processes that are ready to run, but whose memory images are currently stored on the backing store, the operating system can ensure that these processes are loaded back into memory quickly and efficiently when required.

4.4 Context Switch Time

Memory management is a critical aspect of modern operating systems, and swapping processes in and out of memory is a key strategy used to manage memory effectively. However, swapping can be a time-consuming process, particularly when the process to be executed is not currently in memory.

When a process needs to be executed, but it is not currently in memory, the system must swap out a process and swap in the target process. This operation can take a significant amount of time, especially if the process being swapped out is large. For example, if a 100MB process is swapped to a hard disk with a transfer rate of 50MB/sec, the swap-out time alone will take 2000 ms. This is in addition to the time required to swap in the target process, resulting in a total context switch swapping component time of 4000 ms, or 4 seconds.

To reduce the time required for swapping, modern operating systems use a modified version of standard swapping. One approach is to reduce the amount of memory that is swapped by knowing how much memory is really being used. System calls such as request_memory() and release_memory() can be used to inform the OS of the memory use, along with other constraints on swapping.

Another challenge with swapping is pending I/O. If a process has pending I/O, it cannot be swapped out as the I/O would occur to the wrong process. One approach to this issue is to transfer the I/O to kernel space first and then to the I/O device, a technique known as double buffering. However, this approach adds overhead.

In conclusion, while standard swapping is not commonly used in modern operating systems, swapping processes in and out of memory is still an essential technique for effective memory management. By using modified swapping techniques and managing memory usage efficiently, the system can reduce the time required for swapping and improve overall system performance.

5 Memory allocation strategies:

Memory allocation strategies are crucial to effective management of memory in operating systems. In this chapter, we will explore two of the most popular memory allocation strategies: paging and segmentation.

5.1 Contiguous Allocation

Main memory is a precious resource in any computer system, and operating systems must allocate it efficiently to ensure optimal performance. One early method of allocation is contiguous allocation, in which each process is contained in a single contiguous section of memory. However, this method is limited in its ability to support multiple processes efficiently.

Modern systems typically divide main memory into two partitions: one for the resident operating system and one for user processes. The operating system is usually held in low memory with the interrupt vector, while user processes are held in high memory. To protect user processes from each other and from changing operating-system code and data, relocation registers are used.

The base register contains the value of the smallest physical address, while the limit register contains the range of logical addresses. Each logical address must be less than the limit register. The memory management unit (MMU) maps logical addresses dynamically, allowing for actions such as kernel code being transient and the kernel changing size.

This partitioning of memory and use of relocation registers allows for more efficient allocation of memory resources. However, it is important to note that this approach is not without its limitations. For example, there is a limit to the amount of memory that can be allocated to a single process, and fragmentation can occur as processes are loaded and unloaded from memory.

Overall, the efficient allocation of main memory is critical to the performance of any operating system. By dividing memory into partitions and using relocation registers, modern systems are able to support both the operating system and multiple user processes in a way that maximizes the use of this valuable resource.

5.2 Multiple partition allocation

In a computer system, main memory is a limited resource and must be allocated efficiently to support both the operating system and user processes. One early method of memory allocation is contiguous allocation, where each process is contained in a single contiguous section of memory. However, this method has its limitations, as it restricts the degree of multiprogramming to the number of partitions available.

To address this issue, multiple-partition allocation is used, which allows for variable-partition sizes that are sized to fit a given process's needs. In this method, memory is divided into several partitions, and each process is allocated memory from a hole large enough to accommodate it. A hole is a block of available memory of various sizes that are scattered throughout memory.

When a process exits, it frees up its partition, which can then be used to accommodate new processes. To optimize memory allocation, adjacent free partitions are combined to create larger holes. The operating system keeps track of information about allocated and free partitions, which enables it to allocate memory to new processes and manage memory efficiently.

This method of memory allocation has several benefits, including increased flexibility, improved memory utilization, and the ability to accommodate processes with varying memory requirements. However, it also has some limitations, such as fragmentation, where the free partitions are too small to accommodate new processes. This can lead to inefficient memory utilization and reduced performance.

To address these issues, some modern operating systems use dynamic partitioning, where the size of the partitions is adjusted dynamically to accommodate new processes. This approach minimizes fragmentation

and optimizes memory utilization, resulting in improved system performance.

5.3 Paging

Paging is a memory allocation strategy that divides physical memory into fixed-size blocks called pages. In contrast to segmentation, paging does not divide memory based on the size of the program. Instead, programs are divided into pages of equal size, usually 4KB or 8KB.

Each program is assigned a page table that keeps track of the physical addresses of each page. The operating system maintains a page table for each process. When a program is executed, the virtual addresses generated by the program are translated to physical addresses by the page table.

One advantage of paging is that it enables the use of virtual memory, which allows programs to use more memory than is physically available. Paging also allows for more efficient use of physical memory, as programs can be loaded into memory only when needed.

With paging, physical memory is divided into fixed-sized blocks called frames, which are typically between 512 bytes and 16 Mbytes and are sized to be a power of 2. Similarly, logical memory is divided into blocks of the same size called pages. Pages are the basic unit of transfer between physical memory and the backing store.

When a program is run, the system finds N free frames to load the program into, where N is the number of pages required for the program. The system keeps track of all free frames, and the program's pages are loaded into the frames. A page table is then set up to translate logical to physical addresses, allowing the system to map each page of the program to a specific physical frame.

One advantage of paging is that it avoids external fragmentation because the physical address space of a process can be noncontiguous. This is because a process is allocated physical memory whenever it's available, so it doesn't need to wait for contiguous memory blocks to become available.

However, paging still suffers from internal fragmentation, where allocated memory may be slightly larger than requested memory. The size difference between a page and a segment that a program requires is internal to a frame but not being used.

To avoid this issue, the system may employ a technique called demand paging, where pages are brought into memory only when they are needed. This can help reduce the amount of memory that is wasted due to internal fragmentation.

Finally, it's worth noting that the backing store is also divided into pages, so it also suffers from the same fragmentation issues as physical memory. However, modern systems use sophisticated algorithms to manage the backing store, ensuring that pages are allocated efficiently and minimizing fragmentation as much as possible.

Example: Here's a sample pseudocode for the paging algorithm:

```
// Initialize variables

pageSize = 4KB

pageTable = []

numPages = totalMemory / pageSize

freeList = [0, 1, 2, ..., numPages-1]


// Allocate a page

function allocatePage():

    if freeList is empty:
```

```
        return null // no free page available
    else:
        pageFrame = freeList.pop(0) // get the first free page
frame
        pageTable[pageNumber] = pageFrame // map the page to the
frame
        return pageFrame


// Free a page
function freePage(pageNumber):
    pageFrame = pageTable[pageNumber]
    freeList.append(pageFrame) // add the frame to the free list
    pageTable[pageNumber] = null // unmap the page
```

This is just a basic example of how the paging algorithm can be implemented in pseudocode. Actual implementations may vary depending on the specific requirements and constraints of the system.

5.3.1 Address Translation Scheme

In virtual memory systems, the address generated by the CPU is divided into a page number and a page offset. The page number is used as an index into a page table that contains the base address of each page in physical memory. The page offset is combined with the base address to define the physical memory address that is sent to the memory unit.

The division of the address into page number and offset is important because it allows the operating system to manage memory more efficiently. Instead of requiring that all of a program's memory be loaded into physical memory at once, the operating system can load only the pages that are currently needed. This is possible because each page is a

fixed size, and the operating system can keep track of which pages are currently in use and which are not.

The size of the page is an important parameter in virtual memory systems, as it determines the granularity of memory management. In general, larger page sizes reduce the overhead of managing virtual memory, but they also increase the amount of internal fragmentation. This is because each page must be allocated in its entirety, even if the program only needs a portion of the page.

The relationship between the logical address space and the page size is also important, as it determines the number of pages that can be addressed. For a given logical address space of 2^m and a page size of 2^n, the number of pages that can be addressed is 2^(m-n). This means that larger logical address spaces or smaller page sizes will require larger page tables to be maintained by the operating system.

In summary, the division of the address generated by the CPU into page number and page offset allows for efficient management of memory in virtual memory systems. The page size and the relationship between the logical address space and the page size are important parameters that must be carefully chosen to balance efficiency and internal fragmentation.

5.3.2 The page table

The page table is a data structure used by the operating system to map virtual addresses to physical addresses. Each process has its own page table, which is kept in main memory. The page table is typically implemented as an array of page table entries (PTEs), with one entry for each page in the process's address space.

The page table base register (PTBR) is a hardware register that points to the base of the page table in memory. The page table length register (PTLR) indicates the size of the page table. When the CPU generates a virtual address, the page number is used as an index into the page table.

The PTE at that index contains the physical page number and the status of the page (whether it is in memory or on disk).

One major drawback of this scheme is that every memory access requires two memory accesses: one to retrieve the PTE from memory and one to retrieve the actual data or instruction. This can significantly slow down the system's performance.

To mitigate this issue, a special fast-lookup hardware cache called associative memory or translation look-aside buffers (TLBs) is used. The TLB is a small, high-speed cache that stores recently used page table entries. When the CPU generates a virtual address, it first checks the TLB. If the page table entry is found in the TLB, the corresponding physical address is immediately used. If the page table entry is not in the TLB, the page table must be accessed in memory.

5.3.3 Associative Memory and Translation Look-Aside Buffers (TLBs)

In simple terms, associative memory is a type of computer memory that enables quick and efficient data retrieval by searching for a specific piece of information based on its content rather than its address. Unlike the conventional computer memory that uses addresses to store and retrieve data, associative memory stores data as pairs of key-value, where the key represents the content of the data, and the value is the actual data.

The key-value pairs in associative memory are stored in a table, and when a search operation is performed, the memory looks for the key in the table and retrieves the corresponding value. Associative memory is commonly used in CPU caches, where it enables quick access to frequently used data without the need to search through the main memory.

Translation Look-Aside Buffers, or TLBs for short, are a type of cache memory that stores recently accessed virtual-to-physical address translations. TLBs are used in modern computer systems to speed up

the memory access process by reducing the number of memory accesses required to translate virtual memory addresses to physical memory addresses.

In simple terms, when a program running on a computer system accesses memory, it provides a virtual memory address, which needs to be translated into a physical memory address. This process can be time-consuming and resource-intensive, especially when the same memory locations are accessed multiple times.

To speed up this process, TLBs are used to store the most recently accessed virtual-to-physical address translations. When a program requests memory access, the TLB is checked first to see if the required translation is available. If it is, the TLB provides the physical memory address directly, which speeds up the memory access process. If the translation is not available in the TLB, the CPU must perform a complete virtual-to-physical address translation, which takes longer.

Associative memory and TLBs are both used extensively in modern computer systems to improve memory management efficiency. Associative memory is used in CPU caches to speed up data retrieval, while TLBs are used to speed up virtual-to-physical memory address translations.

By reducing the number of memory accesses required, these components help to reduce the overall memory access time and improve the performance of computer systems. Additionally, they also help to reduce the workload on the CPU and improve the overall system efficiency.

In conclusion, associative memory and translation look-aside buffers (TLBs) are critical components in modern computer memory management systems. They play a significant role in reducing memory access times, improving system efficiency, and reducing the workload on the CPU. As an operating system book author, it's crucial to

understand the workings of these components to design efficient and reliable memory management systems.

5.3.4 Effective Access Time

Associative lookup is a method used in computer memory management to search for information based on its content rather than its address. It is commonly used in translation look-aside buffers (TLBs) to speed up the memory access process. When a program running on a computer system requests memory access, the TLB checks its associative registers to see if the required translation is available. If it is, the TLB provides the physical memory address directly, which speeds up the memory access process. If the translation is not available in the TLB, the CPU must perform a complete virtual-to-physical address translation, which takes longer.

The time required for associative lookup is usually very short, typically in the range of ε time units, where ε represents the search time required by the TLB. This time is significantly shorter than the time required for a complete virtual-to-physical address translation, which can take up to 10% of memory access time.

The hit ratio is the percentage of times that a page number is found in the associative registers. The hit ratio is related to the number of associative registers available in the TLB. A higher hit ratio means that more page numbers are found in the associative registers, reducing the number of memory accesses required.

To calculate the impact of associative lookup on memory access time, we use the effective access time (EAT) formula. EAT is the average time required to access a memory location, taking into account both the hit ratio and the time required for memory access. The EAT formula is:

$$EAT = (1 + \varepsilon)\ \alpha + (2 + \varepsilon)(1 - \alpha)$$

Where α is the hit ratio and ε is the time required for associative lookup.

Let's consider an example to illustrate the impact of associative lookup on memory access time. Suppose we have a computer system with a hit ratio of 80%, a TLB search time of 20ns, and a memory access time of 100ns. Using the EAT formula, we can calculate the effective access time as:

EAT = (1 + 0.2) x 0.8 x 100 + (2 + 0.2) x (1 - 0.8) x 100 = 120ns

In this example, the effective access time is 120ns. However, if we increase the hit ratio to a more realistic value of 99%, the effective access time is reduced to:

EAT = (1 + 0.2) x 0.99 x 100 + (2 + 0.2) x (1 - 0.99) x 100 = 101ns

As you can see, the impact of associative lookup on memory access time is significant, especially when the hit ratio is high.

In conclusion, associative lookup is a powerful technique used in computer memory management to reduce memory access time and improve overall system efficiency. By using associative lookup in TLBs, computer systems can speed up the memory access process and reduce the workload on the CPU. The hit ratio and the time required for associative lookup are critical factors that impact the effective access time, and it is essential to consider these factors when designing memory management systems.

5.3.5 Memory Protection

One way memory protection is implemented is by associating a protection bit with each frame to indicate whether read-only or read-write access is allowed. This protection bit can also be used to indicate execute-only access or other forms of access control. By using these protection bits, the operating system can prevent programs from accessing memory in ways that could potentially cause harm or corruption.

Another way memory protection is implemented is by using valid-invalid bits attached to each entry in the page table. The valid bit indicates that the associated page is in the process' logical address space and is, therefore, a legal page. The invalid bit indicates that the page is not in the process' logical address space.

The page table is used by the CPU to translate virtual memory addresses into physical memory addresses. When a program attempts to access memory, the CPU uses the page table to determine if the memory location is valid and if the access is legal. If a program attempts to access memory that is not in its logical address space, or if it attempts to perform an illegal operation on memory, the CPU triggers an exception or trap, which is handled by the operating system.

Another approach to memory protection is to use a page-table length register (PTLR), which stores the maximum index of the page table. This register ensures that programs do not exceed the allocated space in the page table and helps prevent buffer overflow attacks.

When a memory violation occurs, such as an attempt to access an invalid memory location or perform an illegal operation on memory, the CPU triggers an exception or trap. The operating system handles the exception by terminating the offending process or by taking other appropriate actions, such as displaying an error message or initiating a recovery process.

In conclusion, memory protection is a critical aspect of computer systems that ensures the safety and integrity of programs and data. By using protection bits and valid-invalid bits, and by using a page table length register, the operating system can prevent programs from accessing memory in ways that could potentially cause harm or corruption. Any violations of memory protection result in an exception or trap, which is handled by the operating system. By implementing robust memory protection mechanisms, computer systems can ensure the safety and security of the programs and data they contain.

5.4 Segmentation

Segmentation is a memory allocation strategy that divides memory into variable-sized segments. Each segment represents a logical unit of the program, such as a function or data structure.

Like paging, segmentation requires the use of a segment table to translate virtual addresses to physical addresses. Each segment table entry contains the base address and length of the segment.

A segment is a logical unit that represents a specific part of the program, such as the main program, a procedure, a function, a method, an object, local variables, global variables, a common block, a stack, a symbol table, or an array.

Segments provide a flexible way of organizing program code and data. Instead of being limited to a single contiguous block of memory, a program can be composed of multiple segments, each with its own size and location in memory. This allows for more efficient use of memory and enables the system to better manage the allocation and deallocation of memory resources.

Segments are typically defined by the programmer and are managed by the operating system. The operating system is responsible for allocating memory for each segment and for loading the segments into memory when needed. The operating system also provides mechanisms for accessing the segments and for ensuring that the segments are protected from unauthorized access or modification.

One of the advantages of using segments is that they provide a way of separating program code and data into distinct logical units. This can make it easier for programmers to organize their code and can help to reduce errors and bugs. For example, by grouping related variables and functions into a single segment, programmers can more easily keep track of the data and code that are associated with each other.

Another advantage of using segments is that they provide a way of managing memory resources more efficiently. Because segments can be allocated and deallocated independently, the system can more easily manage memory usage and avoid problems such as fragmentation or overallocation.

However, segmentation is also more complex than paging, as it requires the management of variable-sized memory segments. This can lead to external fragmentation, which occurs when free memory is broken up into small chunks that cannot be used to satisfy larger memory requests.

Example: Here is an example of pseudocode for segmentation:

```
// Define the Segment Table data structure
struct SegmentTableEntry {
    uint32_t base_address;
    uint32_t limit;
    uint8_t protection;
};

// Define the Process data structure
struct Process {
    uint32_t pid;
    SegmentTableEntry segment_table[MAX_SEGMENTS];
    uint32_t num_segments;
};

// Allocate a new segment for a process
void allocate_segment(Process *process, uint32_t size, uint8_t protection) {
```

```
    // Find a free slot in the segment table
    uint32_t index = 0;
    while (process->segment_table[index].limit != 0 && index < MAX_SEGMENTS) {
        index++;
    }

    if (index == MAX_SEGMENTS) {
        // No free slots in the segment table
        return;
    }

    // Allocate memory for the new segment
    uint32_t base_address = allocate_memory(size);

    // Update the segment table entry
    process->segment_table[index].base_address = base_address;
    process->segment_table[index].limit = size;
    process->segment_table[index].protection = protection;

    // Increment the number of segments in the process
    process->num_segments++;
}

// Free a segment for a process
void free_segment(Process *process, uint32_t segment_index) {
```

```
    // Check if the segment index is valid
    if (segment_index >= process->num_segments) {
        return;
    }

    // Free the memory associated with the segment
    uint32_t base_address = process->segment_table[segment_index].base_address;
    uint32_t size = process->segment_table[segment_index].limit;
    free_memory(base_address, size);

    // Clear the segment table entry
    process->segment_table[segment_index].base_address = 0;
    process->segment_table[segment_index].limit = 0;
    process->segment_table[segment_index].protection = 0;

    // Decrement the number of segments in the process
    process->num_segments--;
}
```

Note that this is just a basic example and does not include error checking or other important details.

5.4.1 Address Translation Scheme

In some operating systems, the logical address of a process is divided into two parts: the segment number and the offset. The segment number is used to index into a segment table that maps two-dimensional physical addresses. The segment table contains

information about the segments used by the process, including their base addresses and lengths.

Each entry in the segment table has a base field that contains the starting physical address of the segment in memory, and a limit field that specifies the length of the segment. The segment table base register (STBR) points to the location of the segment table in memory, while the segment table length register (STLR) indicates the number of segments used by the program.

When a program generates a logical address, the segment number is used to index into the segment table to obtain the base address and length of the segment that contains the offset. The physical address is then computed by adding the base address and offset.

Using a segment table allows for non-contiguous allocation of physical memory to a process, which can help reduce external fragmentation. However, like with paging, there may still be internal fragmentation due to unused space within a segment.

It's important to note that not all operating systems use segment tables, and those that do may use different variations of this approach. Nonetheless, understanding how logical addresses are translated into physical addresses can help us better understand how processes interact with memory in an operating system.

5.5 Paging vs segmentation

Advances in memory technology have led to the development of different memory allocation strategies, such as paging and segmentation. Each strategy has its advantages and disadvantages, which affect the overall performance of the system. In this chapter, we will explore the pros and cons of each strategy in the context of memory management.

Paging is a memory allocation strategy that divides the memory into fixed-size pages, usually 4KB in size. The process's memory is also divided into fixed-size pages. The system maps each page of the process's memory to a corresponding page in physical memory, resulting in a virtual-to-physical address mapping. Paging has several advantages, such as:

- Easy management: Paging is easy to manage since the memory is divided into fixed-size pages. The system can allocate and deallocate pages quickly and efficiently.
- Efficient use of memory: Paging can use the physical memory efficiently, as the system only loads the necessary pages of a process into memory. Unused pages can be swapped out to disk, freeing up physical memory for other processes.
- Memory protection: Paging provides memory protection by mapping each process to its own memory space. This isolation ensures that a process cannot access the memory space of another process.

However, paging has some disadvantages, such as:

- Fragmentation: Paging can lead to fragmentation of the physical memory. If the system needs to allocate a contiguous block of physical memory larger than the available free memory, it must move pages around to create a contiguous block, resulting in fragmentation.
- Overhead: Paging can incur an overhead in terms of memory access time due to the extra level of indirection involved in accessing memory through the page table.

Segmentation is another memory allocation strategy that divides the memory into logical segments of varying sizes. Each segment

corresponds to a portion of the process's memory, such as the stack, heap, or code segment. Segmentation has several advantages, such as:

- Flexibility: Segmentation is flexible since it can allocate memory segments of different sizes. This flexibility makes it suitable for applications that require dynamic memory allocation.
- No fragmentation: Segmentation does not lead to fragmentation since each segment can be allocated independently. This makes it easier to allocate contiguous memory blocks.
- Sharing: Segmentation allows memory sharing between processes since different processes can share segments. This feature enables faster inter-process communication.

However, segmentation also has some disadvantages, such as:

- Overhead: Segmentation can incur an overhead in terms of memory access time due to the extra level of indirection involved in accessing memory through the segment table.
- Memory protection: Segmentation can be challenging to manage, as there is no built-in memory protection mechanism. This lack of protection can lead to memory leaks, buffer overflows, and other security vulnerabilities.

In conclusion, both paging and segmentation have their advantages and disadvantages. The choice of memory allocation strategy depends on the requirements of the application and the hardware constraints. While paging is easy to manage and provides memory protection, segmentation is flexible and can be used for sharing memory between processes.

5.6 Choosing a Memory Allocation Strategy

When choosing a memory allocation strategy, it is important to consider the requirements of the program and the system resources available. Paging is generally used in systems with limited physical memory and a large virtual address space, while segmentation is more commonly used in systems with more available physical memory and variable program memory requirements.

In addition to paging and segmentation, other memory allocation strategies include buddy memory allocation and slab allocation. Buddy memory allocation divides memory into fixed-size blocks and allocates blocks that are closest in size to the requested size. Slab allocation is a more specialized technique that is used in systems with a large number of similar objects, such as file system buffers.

Overall, choosing the most appropriate memory allocation strategy is critical to the effective management of memory in operating systems. By carefully considering the requirements of the system and the program, developers can choose the most efficient and effective memory allocation strategy.

5.7 Dynamic Storage Allocation

Memory management is a critical aspect of operating systems, particularly when it comes to managing free space. This chapter discusses different strategies that can be used to manage free space and minimize fragmentation.

When a program requests memory from the operating system, it may not always know exactly how much memory it will need. For this reason, many memory allocation systems allow for variable-sized requests. However, this can lead to fragmentation, where there are small pockets

of free space scattered throughout memory that cannot be used to satisfy larger requests.

One common approach to managing free space is called the buddy system. In this approach, the operating system maintains a list of free memory blocks, each of which is a power of two in size. When a request comes in, the system finds the smallest free block that can satisfy the request, and splits it in two if necessary to create two smaller free blocks. This continues recursively until the smallest block that can satisfy the request is found. When a block is freed, the system checks to see if its buddy block (the block with which it was originally split) is also free. If so, the two blocks are merged back into a larger block.

Another approach is called the slab allocation system. In this approach, the operating system maintains a set of pre-allocated memory chunks, each of which is of a fixed size. When a request comes in, the system finds the appropriate chunk and returns a pointer to the requested memory within that chunk. When the memory is freed, it is returned to the appropriate chunk rather than being released back to the general free space pool. This can reduce fragmentation because memory is always released to a specific chunk rather than being returned to the general pool, which can lead to small pockets of free space that cannot be used.

Different strategies have different trade-offs in terms of time and space overheads. For example, the buddy system can be more efficient in terms of space usage because it can split and merge blocks to exactly fit the requested size. However, it can be less efficient in terms of time overhead because it may need to search the free block list recursively to find a block that is the right size. The slab allocation system, on the other hand, can be more efficient in terms of time overhead because it always returns memory from a pre-allocated chunk, but it may be less efficient in terms of space usage because chunks may not be fully utilized.

In summary, managing free space is an important aspect of memory management in operating systems. Different strategies can be used to minimize fragmentation and balance time and space overheads. The buddy system and slab allocation system are two common approaches, each with their own trade-offs.

5.7.1 The best-fit strategy

In the management of free space, one strategy that can be used to minimize fragmentation is the best fit strategy. This strategy involves searching through the free list to find chunks of free memory that are as big or bigger than the requested size. Then, the strategy returns the smallest block from the group of candidates that meet the requested size, known as the best-fit chunk.

The best-fit strategy aims to reduce wasted space by returning a block that is as close as possible to what the user asks for. However, the strategy also comes with a performance cost. Naive implementations of the best-fit strategy may suffer a heavy performance penalty when performing an exhaustive search for the correct free block.

To implement the best-fit strategy, the allocator has to traverse the free list and compare each block's size to the requested size. Once the allocator finds a block that can fit the request, it has to determine which block is the smallest from the group of candidates. The allocator then returns this block to the requesting program.

One significant drawback of the best-fit strategy is that it can lead to external fragmentation. External fragmentation occurs when the allocator cannot find a single block of memory that is large enough to satisfy a request, even though the total free memory is sufficient. This issue arises when small blocks of free memory are scattered throughout the heap.

To address the issue of external fragmentation, some implementations of the best-fit strategy combine adjacent free blocks into a single larger

block. This approach can help reduce fragmentation by consolidating smaller free blocks into more significant ones.

Overall, the best-fit strategy is a useful approach to manage free space when allocating variable-sized requests. It aims to minimize wasted space and can be combined with other strategies to further reduce fragmentation. However, its performance cost must be carefully considered when implementing this strategy.

5.7.2 The worst-fit strategy

Worst fit is a memory allocation strategy that takes the opposite approach to best fit. Instead of finding the smallest available block that can satisfy a request, it searches for the largest block and allocates the requested amount from it, leaving the remaining space on the free list.

The rationale behind this approach is to leave large chunks of free memory that can be used for larger requests in the future, thereby reducing fragmentation. However, studies have shown that worst fit tends to perform poorly, leading to excessive fragmentation and high overheads.

One of the reasons for this poor performance is that worst fit requires a full search of the free list, just like best fit. This search can be expensive and time-consuming, especially in the presence of a large number of small free blocks.

Moreover, worst fit can lead to a phenomenon known as external fragmentation, where the available memory is fragmented into many small free blocks that cannot be used to satisfy larger requests, even if their total size is sufficient. This can happen if worst fit repeatedly breaks down larger free blocks into smaller ones to satisfy requests.

Overall, worst fit is not a recommended strategy for managing free memory. Other, more sophisticated approaches, such as buddy allocation and slab allocation, have been developed to address the

shortcomings of simple strategies like best fit and worst fit. These approaches aim to minimize fragmentation, reduce overheads, and improve performance by using more efficient data structures and algorithms.

5.7.3 The first-fit strategy

First fit is one of the most commonly used strategies for managing free space in a memory allocator. It's also one of the simplest. The basic idea is to search the free list from the beginning, looking for the first block that is large enough to satisfy the allocation request. Once a block is found, it is allocated to the user, and any remaining free space is added back to the free list.

One advantage of the first fit strategy is speed. Because the allocator only needs to search the free list until it finds a block that is large enough to satisfy the allocation request, it can be quite fast. This is especially true when the free list is relatively small or when the allocation request is relatively small.

However, one potential disadvantage of the first fit strategy is that it can lead to fragmentation of the free list. Specifically, if the allocator repeatedly allocates and deallocates small blocks of memory, the free list can become fragmented with many small, unusable gaps. This can make it more difficult to find large blocks of contiguous memory when a larger allocation request is made.

To address this issue, some memory allocators use address-based ordering of the free list. By keeping the list ordered by the address of the free space, it becomes easier to coalesce adjacent blocks of free space, which can help to reduce fragmentation and make it easier to find larger blocks of contiguous memory.

Overall, the first fit strategy is a useful and widely used approach for managing free space in memory allocators. However, it's important to

be aware of its potential drawbacks, particularly when dealing with small or frequent allocation requests.

5.7.4 The next-fit strategy

The next fit algorithm is an improvement on the first fit method, which simply finds the first block that is big enough and returns the requested amount to the user. The problem with first fit is that it can pollute the beginning of the free list with small objects, leading to fragmentation. Next fit aims to avoid this problem by keeping an extra pointer that indicates where the last search for free space ended.

The next fit algorithm works by starting the search for free space at the location where the last search ended. If there is no free space available at that location, the search continues from the beginning of the list. By doing this, next fit spreads the searches for free space throughout the list more uniformly, thus avoiding splintering of the beginning of the list.

One advantage of next fit is that it performs better than worst fit since it avoids the fragmentation that worst fit can create. However, it may still suffer from fragmentation since it does not attempt to compact free space. Moreover, it may not find the best fit for a request since it does not perform an exhaustive search of all the free spaces available.

In conclusion, next fit is a memory management strategy that tries to avoid the problems of first fit and worst fit. By keeping an extra pointer to the location where the last search ended, next fit can spread the searches for free space throughout the list more uniformly, thus avoiding splintering of the beginning of the list. However, it does not attempt to compact free space and may not find the best fit for a request. Therefore, choosing the right memory management strategy depends on the specific requirements of the system.

5.8 Fragmentation

Memory management is a critical component of any operating system, and understanding the different types of fragmentation is essential to efficient memory usage. External fragmentation occurs when there is enough total memory space to satisfy a request, but it is not contiguous, resulting in wasted space. Internal fragmentation, on the other hand, occurs when the allocated memory is slightly larger than the requested memory, leaving unused memory within a partition.

One way to address external fragmentation is through compaction, which involves shuffling memory contents to place all free memory together in one large block. However, this is only possible if relocation is dynamic and can be done at execution time. This technique can be effective in reducing external fragmentation and increasing the amount of available memory.

Another issue to consider is the impact of I/O operations on memory usage. To prevent data corruption during I/O, jobs must be latched in memory while involved in I/O, and I/O operations should only be conducted into OS buffers. This approach helps to minimize the risk of data loss and ensures that memory usage remains efficient.

It is also worth noting that fragmentation is not just limited to main memory but can also occur in the backing store. Thus, similar techniques, such as compaction, can be used to address fragmentation in the backing store and ensure that memory usage remains efficient and effective.

In conclusion, understanding the different types of fragmentation and the techniques to address them is essential for efficient memory usage in any operating system. By minimizing fragmentation and optimizing memory allocation, the overall performance of the system can be improved.

6 Memory API

In general, there are two types of memory allocation in a running program: stack memory and heap memory. Understanding the differences between these two types of memory is essential to writing programs that are fast and stable.

Stack memory is a type of automatic memory, which means that the compiler manages the allocations and deallocations of this memory for you, the programmer. This type of memory is used to hold local variables, function call frames, and other temporary data that is needed during the execution of a program. Stack memory is fast and efficient, and its usage is straightforward since the compiler takes care of all the details.

Heap memory, on the other hand, is not automatically managed by the compiler. Instead, the programmer is responsible for explicitly allocating and deallocating this memory. Heap memory is used to store data that needs to be accessed over a longer period than stack memory. Since heap memory is not automatically managed, it can be more challenging to use correctly, and mistakes can lead to serious bugs in a program.

In a typical C program, the stack and heap memory are used in conjunction with each other. The stack is used for small, short-lived variables and function call frames, while the heap is used for large, long-lived data structures that need to be dynamically allocated and deallocated. Efficiently managing the use of these two types of memory is critical to the performance and stability of a program.

The operating system provides a virtual memory abstraction that allows programs to access memory as if it were a contiguous block of physical memory. This abstraction is called the address space, and it is the running program's view of memory in the system. The address space of

a process contains all of the memory state of the running program, including the stack and heap memory.

In summary, understanding the differences between stack and heap memory is critical to writing efficient and reliable C programs. Stack memory is automatically managed by the compiler and is used for small, short-lived variables, while heap memory is explicitly managed by the programmer and is used for large, long-lived data structures. Both types of memory are critical to the performance and stability of a program, and efficient management of these resources is a crucial task for any C programmer.

6.1 The malloc() Call

When writing programs in C, one of the most common tasks is to allocate memory dynamically. This is often necessary when you don't know beforehand how much memory you will need, or when you need memory that persists beyond the lifetime of a particular function call.

In C, the malloc() function is used to dynamically allocate memory from the heap. The malloc() call is quite simple: you pass it a size asking for some room on the heap, and it either succeeds and gives you back a pointer to the newly-allocated space, or fails and returns NULL.

For example, if you wanted to allocate an array of integers with a size of 10, you could do so with the following code:

```
int* my_array = (int*)malloc(10 * sizeof(int));
```

This code allocates enough memory to hold 10 integers and returns a pointer to the first element of the array. Note that we cast the result of malloc() to an int pointer, since malloc() returns a void pointer by default.

Once you've finished using the memory you've allocated, it's important to free it to avoid memory leaks. To do so, simply call the free() function and pass it the pointer to the memory you want to free:

It's important to note that dynamic memory allocation can be a source of bugs and performance problems if not used carefully. Allocating too much memory or failing to free memory can lead to memory leaks, while allocating too little memory can result in buffer overflows and other errors. It's also important to consider the lifetime of the memory you allocate, as well as any potential race conditions that may arise when multiple threads are accessing the same memory.

6.2 The free() Call

Memory management is an essential aspect of programming, especially when dealing with heap memory. Allocating memory is an easy task, but knowing when and how to free memory is challenging. The free() call is used to free heap memory that is no longer in use.

When a program no longer needs a particular block of memory, it should free that memory to prevent memory leaks. Memory leaks occur when a program continues to allocate memory without freeing it, leading to a shortage of available memory.

To free memory using the free() call, the programmer needs to pass in the pointer to the allocated memory block. Once freed, the memory becomes available for future allocation. It is crucial to note that freeing a block of memory does not necessarily erase its contents; it only marks it as available for reuse.

It is also essential to be cautious when using the free() call. Attempting to free a block of memory that has already been freed or attempting to free an invalid pointer can result in unexpected behavior. For this reason, it is a good practice to assign the pointer to NULL after freeing it to avoid potential issues in the future.

In summary, the free() call is a simple yet essential function for managing heap memory in a program. It helps prevent memory leaks and ensures that memory is efficiently utilized by the program. However, care must be taken when using the free() call to avoid unexpected behavior.

Example: Here's an example program that demonstrates the use of the free() function in C:

```
#include <stdio.h>
#include <stdlib.h>


int main() {
    // Allocate memory for an integer on the heap
    int *num = (int*)malloc(sizeof(int));
    if (num == NULL) {
        printf("Error: failed to allocate memory.\n");
        return 1;
    }

    // Assign a value to the integer
    *num = 42;
    printf("The value of num is: %d\n", *num);

    // Free the memory
    free(num);

    // Attempt to access the memory after it has been freed
```

```
    printf("The value of num is now: %d\n", *num);


    return 0;
}
```

This program first allocates memory for an integer on the heap using the malloc() function. It then assigns a value to the integer and prints it out. After that, it frees the memory using the free() function. Finally, it attempts to access the memory again and print out the value of the integer, which should result in undefined behavior since the memory has been freed.

7 Paging and Page Replacement Algorithms

Welcome to the chapter on Paging and Page Replacement Algorithms. Memory management is a crucial aspect of operating systems that involves managing the memory resources of a system to ensure efficient allocation and usage. In this chapter, we will discuss the concept of paging and page tables, which is one of the most common strategies used for memory allocation in modern operating systems.

We will also explore the different page replacement algorithms, such as FIFO, LRU, Optimal, and Clock. These algorithms are used to determine which page to replace when a page fault occurs. Finally, we will evaluate the performance of these algorithms and compare their advantages and disadvantages.

Understanding paging and page replacement algorithms is essential for optimizing the use of memory resources in an operating system. So, let's dive into the details of these concepts and explore how they are used to manage memory in modern operating systems.

7.1 Overview of paging and page tables

In modern operating systems, the use of virtual memory is crucial for efficient memory management. One of the key components of virtual memory is paging. In this chapter, we will provide an overview of paging and page tables, which are used to manage virtual memory.

Paging is a memory management scheme that allows an operating system to use secondary memory, such as a hard disk, as an extension of primary memory, such as RAM. The idea behind paging is to divide the physical memory into small fixed-sized blocks called frames, and divide the virtual memory into the same-sized blocks called pages. These pages are then mapped to the frames using a page table.

A page table is a data structure used by the operating system to map virtual addresses to physical addresses. Each entry in the page table corresponds to a page in the virtual address space. The entry contains information about the physical address where the page is stored in memory, as well as some control bits that indicate whether the page is valid, dirty, or accessed.

The page table is stored in memory and is accessed by the hardware during memory accesses. The page table is typically organized as a tree or a hash table for efficient access.

When a process requests access to a virtual address, the hardware first checks the page table to see if the page is currently in physical memory. If the page is not in memory, a page fault occurs and the operating system must retrieve the page from secondary storage and load it into physical memory.

Paging has several advantages over traditional memory management schemes:

- Flexibility: Paging allows the operating system to manage physical memory in a more flexible manner. With paging, the operating

system can allocate and deallocate memory on demand, and can use secondary storage as an extension of physical memory.

- Protection: Paging provides protection against unauthorized access to memory. Each process has its own page table, which ensures that it can only access its own memory and not the memory of other processes.
- Sharing: Paging allows multiple processes to share the same physical memory. This is useful for programs that need to share large data structures, such as databases.

Paging also has some disadvantages:

- Overhead: Paging introduces some overhead, both in terms of CPU time and memory usage. The page table must be maintained by the operating system, and each memory access requires an additional lookup in the page table.
- Fragmentation: Paging can lead to fragmentation of physical memory. When a page is swapped out to secondary storage, the physical memory it occupied becomes available for other pages. However, this memory may not be contiguous, which can lead to fragmentation.

Paging is a key component of virtual memory and is used by most modern operating systems. It provides flexibility, protection, and sharing, but also introduces overhead and can lead to fragmentation. The use of page tables allows the operating system to efficiently map virtual addresses to physical addresses and manage memory in a more flexible manner.

Example: Here's a pseudocode for a basic page table in a virtual memory system:

```
// Define the page table structure
```

```
struct PageTableEntry {
    int present;   // Indicates whether the page is in physical memory (1) or on disk (0)
    int frame;    // The frame number in physical memory where the page is stored
};

// Create an array to hold the page table entries
PageTableEntry page_table[num_pages];

// Initialize the page table entries
for (int i = 0; i < num_pages; i++) {
    page_table[i].present = 0;
    page_table[i].frame = -1;
}

// Function to translate a virtual address to a physical address
int translate_address(int virtual_address) {
    int page_number = virtual_address / page_size;
    int offset = virtual_address % page_size;

    if (page_table[page_number].present == 0) {
        // Page fault - load the page into physical memory from disk
        int frame_number = get_free_frame();
        load_page_from_disk(page_number, frame_number);
        page_table[page_number].present = 1;
```

```
        page_table[page_number].frame = frame_number;
    }

    int physical_address = page_table[page_number].frame * page_size + offset;
    return physical_address;
}
```

This pseudocode defines a page table as an array of PageTableEntry structs. Each entry in the page table contains a present flag that indicates whether the page is currently in physical memory or on disk, and a frame number that specifies the physical frame number where the page is stored.

The translate_address function takes a virtual address as input and returns the corresponding physical address. It first computes the page number and offset of the virtual address, and then checks whether the corresponding page is currently in physical memory. If the page is not present, a page fault occurs and the page is loaded into a free physical frame from disk. The get_free_frame function and load_page_from_disk function are not shown here, but they would be responsible for allocating a free physical frame and reading the page data from disk, respectively.

Finally, the physical address is computed by multiplying the frame number by the page size and adding the offset, and then returned by the function.

7.2 Page replacement algorithms

In virtual memory systems, page replacement algorithms are used to select which pages to remove from physical memory when there is no more free space available. The goal of these algorithms is to minimize

the number of page faults that occur and ensure that the most important pages remain in memory. In this chapter, we will review some of the most common page replacement algorithms and explore their strengths and weaknesses.

7.2.1 First-In-First-Out (FIFO) Algorithm

The FIFO algorithm selects the page that was loaded into memory first for eviction. This algorithm is easy to implement and requires only a simple data structure to maintain the order in which pages were loaded. However, the FIFO algorithm suffers from the "Belady's Anomaly" problem, which is when increasing the number of frames in memory can actually increase the number of page faults.

7.2.2 Least Recently Used (LRU) Algorithm

The LRU algorithm selects the page that has not been accessed for the longest period of time for eviction. This algorithm is based on the principle of locality, which states that recently accessed pages are more likely to be accessed again in the near future. The LRU algorithm is effective at minimizing the number of page faults and has been shown to perform well in practice. However, the LRU algorithm can be difficult to implement efficiently, as it requires tracking the access history of every page.

7.2.3 Clock Algorithm

The Clock algorithm is a modification of the FIFO algorithm that is designed to reduce the likelihood of Belady's Anomaly. This algorithm maintains a circular list of pages in memory and uses a clock hand to traverse the list. When a page is accessed, its reference bit is set to 1. When the clock hand reaches a page with a reference bit of 0, that page is selected for eviction. The Clock algorithm is simple to implement and has been shown to perform well in practice.

7.2.4 Least-Frequently Used (LFU) Algorithm

The LFU algorithm selects the page that has been accessed the fewest number of times for eviction. This algorithm is based on the principle that pages that are accessed less frequently are less likely to be accessed in the future. The LFU algorithm can be effective in reducing the number of page faults in workloads with predictable access patterns. However, the LFU algorithm can perform poorly in workloads with irregular access patterns.

7.2.5 Random Algorithm

The Random algorithm selects a page for eviction at random. This algorithm is simple to implement and requires no tracking of page history or usage. However, the Random algorithm can perform poorly in practice, as it may select important pages for eviction.

In this chapter, we have reviewed several common page replacement algorithms and explored their strengths and weaknesses. The choice of page replacement algorithm depends on the specific requirements of the system and the characteristics of the workload. The LRU algorithm is generally considered to be a good default choice, as it performs well in a wide range of workloads. However, other algorithms such as the Clock algorithm and the LFU algorithm can be effective in specific circumstances. The key to effective page replacement is to select an algorithm that balances the competing goals of minimizing the number of page faults and ensuring that the most important pages remain in memory.

8 Segmentation and Compaction

As the complexity of computer systems has increased, so has the need for efficient and effective memory management. Memory is a precious resource, and it is essential to ensure that it is used efficiently to optimize system performance. Segmentation is one approach to memory management that offers some benefits over other methods such as paging. However, segmentation also has its drawbacks, which we will explore in this chapter.

We will then examine the issue of fragmentation, which occurs when the memory is divided into small pieces that cannot be effectively utilized. We will see how external fragmentation arises due to the allocation and deallocation of memory blocks, while internal fragmentation occurs when memory allocated to a process is not fully utilized. We will also look at how compaction can be used to resolve these issues.

Finally, we will dive into garbage collection, which is the process of automatically freeing up memory that is no longer in use. We will discuss two methods of garbage collection, mark-and-sweep and reference counting, and compare their advantages and disadvantages.

8.1 Segmentation

Segmentation is a memory management technique that allows a process to be divided into logical segments, where each segment represents a different part of the program such as code, data, and stack. The segments can be of variable length, allowing for more flexibility in memory allocation than the fixed-size pages used in paging. In this chapter, we will discuss the advantages and disadvantages of segmentation.

Advantages of Segmentation:

- Flexibility: Segmentation provides more flexibility in memory allocation than paging, as the segments can be of variable size. This allows for more efficient use of memory, as segments can be allocated according to the size requirements of the program.
- Protection: Segmentation provides protection for the program's code, data, and stack by dividing them into separate segments. This helps prevent one part of the program from overwriting another part, resulting in a more reliable and stable system.
- Sharing: Segments can be shared between processes, allowing multiple processes to access the same data without the need for copying. This can improve the overall efficiency of the system.
- Simplified Memory Management: Segmentation simplifies memory management by dividing memory into logical segments that can be easily managed by the operating system.

Disadvantages of Segmentation:

- Fragmentation: Segmentation can lead to fragmentation of memory, where the available memory becomes divided into small, unusable chunks. This can result in wasted memory and reduced efficiency.
- Overhead: Segmentation requires additional overhead compared to paging, as the operating system needs to manage the segment table to keep track of the segments.
- External Fragmentation: Segmentation can lead to external fragmentation, where there are enough free memory blocks available, but they are not contiguous. This can cause memory allocation to fail even if there is enough free memory.
- Complexity: Segmentation is more complex than paging, as the segments can be of variable size and need to be managed separately.

In conclusion, segmentation offers several advantages over paging, including flexibility, protection, sharing, and simplified memory management. However, it also has some drawbacks, such as fragmentation, overhead, external fragmentation, and complexity. As with any memory management technique, it is important to weigh the advantages and disadvantages to determine which approach is best for a particular system.

8.2 Fragmentation and compaction

Fragmentation and compaction are critical aspects of memory management that operating systems must address. Fragmentation occurs when memory becomes divided into many small, unusable sections, while compaction is the process of merging these sections to form larger, usable ones.

There are two types of fragmentation: internal fragmentation and external fragmentation. Internal fragmentation occurs when a process is allocated more memory than it needs, resulting in the unused portion of memory remaining unusable. This can occur when a process requests a fixed-sized block of memory but doesn't use it entirely. External fragmentation occurs when there is free memory available but is not contiguous, making it unusable for allocation to processes.

To address fragmentation, an operating system may use compaction. Compaction is a process of moving memory contents around to create larger contiguous blocks of free memory. This process can be time-consuming and is generally used in situations where fragmentation has become a severe issue.

One of the primary advantages of segmentation is that it allows for more efficient memory management. This is because each process is allocated only the amount of memory it requires, eliminating internal

fragmentation. Additionally, segmentation provides better support for dynamic memory allocation, as the allocation size can vary for each segment. This makes it easier to manage memory for complex programs that require different memory sizes for different components.

However, segmentation can also lead to external fragmentation, as memory segments may not be contiguous. This can limit the available memory for new processes, leading to poor system performance. Additionally, managing memory in a segmented environment can be more complex than managing memory in a paged environment.

In summary, fragmentation and compaction are important concepts in memory management. External and internal fragmentation can lead to inefficient use of memory, while compaction can be used to address fragmentation issues. Segmentation is a useful approach to managing memory, but it can also lead to external fragmentation, which can be challenging to manage.

Example: Here's a pseudocode for a basic compaction algorithm for a segmented memory management scheme:

```
Function compact():
    sorted_segments = sort_segments_by_address()
    current_address = 0

    for segment in sorted_segments:
        if segment.base_address != current_address:
            move_segment(segment, current_address)
        current_address += segment.size

    update_segment_table()
```

In this algorithm, we first sort the segments in the memory according to their base address. Then, we start iterating over them in order and check if the current segment's base address is the same as the current address we are tracking. If it's not, we move the segment to the current address. After moving the segment, we update the current address to reflect the new end of the segment. Finally, we update the segment table to reflect the new base addresses of the moved segments.

This basic algorithm assumes that we have access to the segment table and can move segments around in memory. It also assumes that we are working with a system that uses base and limit registers to define segments.

8.3 Garbage collection

Memory management in modern operating systems is a complex and challenging task. Among the various techniques employed to efficiently manage the memory, garbage collection is one of the most important. It is a technique that automatically deallocates memory that is no longer being used by the program. There are two commonly used garbage collection algorithms: mark-and-sweep and reference counting.

Mark-and-sweep algorithm is a garbage collection technique that involves a two-phase process: marking and sweeping. During the marking phase, the garbage collector traverses the object graph starting from the roots and marks all objects that are still in use. During the sweeping phase, the garbage collector deallocates all objects that are not marked.

Example: Here's an example pseudocode for the mark-and-sweep algorithm:

```
function mark_and_sweep() {

   mark();
```

```
   sweep();
}

function mark() {
   for each object in heap {
      if (object.is_marked() == false) {
         object.mark();
         mark_referenced_objects(object);
      }
   }
}

function mark_referenced_objects(object) {
   for each reference in object.references {
      if (reference.is_marked() == false) {
         reference.mark();
         mark_referenced_objects(reference);
      }
   }
}

function sweep() {
   for each object in heap {
      if (object.is_marked() == false) {
         heap.deallocate(object);
```

```
      } else {
         object.unmark();
      }
   }
}
```

Reference counting is another garbage collection algorithm that maintains a count of the number of references to each object. When an object's reference count drops to zero, it is deallocated. The advantage of reference counting is that it can immediately reclaim memory when an object is no longer needed. However, reference counting can be inefficient in the presence of cycles.

Example: Here's an example pseudocode for the reference counting algorithm:

```
function increment_reference_count(object) {
   object.reference_count++;
}


function decrement_reference_count(object) {
   object.reference_count--;
   if (object.reference_count == 0) {
      deallocate(object);
   }
}
```

In conclusion, garbage collection algorithms play a critical role in modern memory management. Mark-and-sweep and reference counting are two commonly used garbage collection algorithms, each with its own advantages and disadvantages. Understanding the benefits

and limitations of these algorithms is important for designing efficient and reliable memory management systems.

9 Memory Protection and Sharing

In this chapter, we will discuss the important topics of memory protection and sharing mechanisms in operating systems. Memory protection is a crucial feature in modern operating systems that ensures the security and integrity of the system. We will look at various protection mechanisms such as access control lists and capabilities.

In addition, we will discuss the concept of memory sharing, which allows multiple processes to access the same memory space. This can greatly improve system performance and efficiency. We will explore different sharing mechanisms such as copy-on-write, memory-mapped files, and shared memory.

By the end of this chapter, you will have a better understanding of how memory protection and sharing work in operating systems and the different techniques used to implement them. So, let's dive into the world of memory protection and sharing!

9.1 Protection mechanisms

In an operating system, it is essential to ensure that processes and users have access only to the resources they are authorized to use. Protection mechanisms are used to control access to these resources. Two common types of protection mechanisms used in operating systems are access control lists (ACLs) and capabilities. In this chapter, we will discuss the basics of ACLs and capabilities and their advantages and disadvantages.

9.1.1 Access Control Lists (ACLs)

An Access Control List is a list of permissions attached to an object. An object can be a file, folder, device, or any other resource that a user or process may need to access. The ACL contains a list of users or groups and their corresponding permissions for the object. For example, a file may have an ACL that allows read and write permissions for the owner, read-only permissions for members of the "developers" group, and no access for others.

An ACL-based access control system can be either discretionary or mandatory. In discretionary access control (DAC), the owner of the object has full control over the access permissions for that object. The owner can modify the ACL to grant or revoke access rights as needed. In mandatory access control (MAC), the operating system enforces access control policies set by an administrator or security policy.

The advantages of using ACLs include the ability to control access to individual resources on a per-user or per-group basis. ACLs also enable delegation of permissions to other users or groups. However, managing ACLs can become complex, especially when dealing with large numbers of users and resources.

9.1.2 Capabilities

Capabilities are a type of access control mechanism that grants permissions to a process rather than a user or group. In this approach, the operating system assigns a set of capabilities to each process at the time of its creation. These capabilities determine what resources the process can access.

Capabilities-based access control is often used in microkernel-based operating systems, where system services are provided by separate processes with defined capabilities. This approach helps to enforce the principle of least privilege, where each process has the minimum permissions needed to perform its task.

The advantages of using capabilities include improved security and the ability to limit access to resources based on the specific requirements of a process. However, capabilities can be challenging to manage when dealing with complex access control scenarios.

9.1.3 Comparison of ACLs and Capabilities

ACLs and capabilities have different approaches to access control. ACLs are generally easier to manage and are used in most operating systems. However, they may be less secure than capabilities-based systems since they grant permissions based on user or group membership. On the other hand, capabilities are more secure and granular, but they can be challenging to manage and are not widely used.

The choice between ACLs and capabilities depends on the specific requirements of the system. In general, ACLs are suitable for systems that require simple access control policies, while capabilities are more suitable for systems that require a high level of security and fine-grained access control.

Access control mechanisms such as ACLs and capabilities are essential for ensuring the security of an operating system. They enable administrators to control access to resources based on specific requirements and help to enforce the principle of least privilege. However, choosing the right access control mechanism requires a careful assessment of the system's requirements, including security, manageability, and complexity. By understanding the advantages and disadvantages of ACLs and capabilities, administrators can choose the right mechanism for their system.

Access Control List (ACL) is a data structure that is used to store permissions associated with an object in an operating system.

Example: Here's a sample pseudocode for implementing an ACL:

```
// Structure of an Access Control Entry (ACE)
```

```
struct ACE {
    int uid;              // User ID
    int gid;              // Group ID
    int permissions;      // Permissions granted to the user/group
};

// Structure of an Access Control List (ACL)
struct ACL {
    int owner_uid;        // User ID of the owner of the object
    int owner_gid;        // Group ID of the owner of the object
    int num_entries;     // Number of Access Control Entries (ACEs)
    ACE entries[MAX_ENTRIES];   // Array of ACEs
};

// Function to check if a given user has permission to perform a certain action on the object
bool check_permission(int user_id, int group_id, int action, ACL acl) {
    // Check if the user is the owner of the object
    if (user_id == acl.owner_uid) {
        if (action & acl.permissions) {
            return true;
        }
    }

    // Check if the user belongs to the group that owns the object
```

```
    if (group_id == acl.owner_gid) {
        if (action & (acl.permissions >> 3)) {
            return true;
        }
    }

    // Check if the user has explicit permission in the ACL
    for (int i = 0; i < acl.num_entries; i++) {
        if (user_id == acl.entries[i].uid || group_id == acl.entries[i].gid) {
            if (action & acl.entries[i].permissions) {
                return true;
            }
        }
    }

    // If none of the above conditions are satisfied, the user does not have permission
    return false;
}
```

In this pseudocode, the ACL struct contains information about the owner of the object, the number of ACEs, and an array of ACE structures. Each ACE structure contains the user/group ID and the permissions granted to that user/group. The check_permission function takes as input the user ID, group ID, and the action to be performed on the object. It then checks whether the user has permission to perform that action based on the information in the ACL. The function returns true if the user has permission and false otherwise.

9.2 Sharing mechanisms

In modern operating systems, processes often need to share information and data with each other. Sharing mechanisms provide a way for processes to communicate and share resources with each other. In this chapter, we will discuss three popular sharing mechanisms: copy-on-write, memory-mapped files, and shared memory.

9.2.1 Copy-On-Write (COW)

Copy-on-write is a technique used by many operating systems to efficiently share memory between processes. In this technique, when a process wants to create a copy of a memory page that is shared with another process, it does not create a new copy of the page immediately. Instead, the operating system creates a new page and marks it as a copy-on-write page. When a process writes to this page, the operating system intercepts the write and makes a new copy of the page before writing to it. This way, both processes have their own separate copy of the page.

One of the advantages of COW is that it is very efficient in terms of memory usage. It allows multiple processes to share the same memory pages without actually duplicating the pages until necessary. This means that COW can save a lot of memory and reduce the overhead of creating and managing memory copies.

Example: Here's a pseudocode for copy-on-write:

```
1. When a process attempts to write to a shared memory page:

2.   If the page is not already marked as copy-on-write:

3.      Create a new page frame in the process's private memory space.

4.     Copy the contents of the shared memory page to the new page frame.
```

```
5.     Mark the new page frame as copy-on-write and set the shared
page to read-only.

6.     Update the page table entry to point to the new page frame.

7.   Otherwise, if the page is already marked as copy-on-write:

8.     Copy the shared memory page to a new page frame, as in steps
3-4.

9.     Update the page table entry to point to the new page frame.

10. Allow the process to write to the new page frame.
```

In summary, when a process tries to write to a shared memory page, the operating system checks whether the page is already marked as copy-on-write. If it isn't, a new page frame is created in the process's private memory space, the contents of the shared memory page are copied to the new frame, and the page table entry is updated to point to the new frame. The shared memory page is then marked as read-only and copy-on-write. If the page is already marked as copy-on-write, a new page frame is created and the shared memory page is copied to it. The page table entry is then updated to point to the new page frame, and the process is allowed to write to the new frame.

9.2.2 Memory-Mapped Files

Memory-mapped files allow processes to share data by mapping the same file into their address spaces. This technique is often used to share large amounts of data between processes. In memory-mapped files, the operating system maps a file into a process's address space, creating a virtual memory page for each block of data in the file. When a process reads or writes to this memory, the operating system reads or writes to the file on the disk.

Memory-mapped files have several advantages. They provide a simple and efficient way to share data between processes. They also allow processes to treat files as if they were part of their address space, which can simplify programming. Memory-mapped files are particularly useful

for sharing large amounts of data, as they can be mapped into memory on demand, rather than being loaded into memory all at once.

Example: Here's an example pseudocode for memory-mapped files:

```
// Open file for reading/writing and memory-map it

file_descriptor = open("filename.txt", O_RDWR);

file_size = get_file_size("filename.txt");

file_map = mmap(NULL, file_size, PROT_READ | PROT_WRITE, MAP_SHARED,
file_descriptor, 0);


// Use the memory-mapped file as a buffer

memcpy(file_map + offset, buffer, length);


// Unmap the memory and close the file

munmap(file_map, file_size);

close(file_descriptor);
```

Note that this is just an example and may need to be adapted to the specific operating system and programming language being used. The get_file_size function is not part of the standard C library, but can be implemented using system calls such as stat or fstat. Additionally, the memcpy function is used to demonstrate how the memory-mapped file can be used as a buffer for reading/writing data.

9.2.3 Shared Memory

Shared memory is another popular technique for sharing data between processes. In shared memory, a region of memory is created that can be accessed by multiple processes. Each process can read and write to this memory as if it were its own private memory. The operating system is

responsible for managing the shared memory region and ensuring that all processes have the correct permissions to access it.

One of the advantages of shared memory is that it is very fast, as there is no need to copy data between processes. This can be particularly useful when processes need to share large amounts of data. Shared memory is also very flexible, as it can be used for a wide range of data-sharing scenarios.

Example: Here is an example pseudocode for shared memory:

```
// Server process creates a shared memory segment
int shm_id = shmget(key, size, IPC_CREAT | 0666);


// Attach the shared memory segment to the address space of the process
char* shared_memory = shmat(shm_id, NULL, 0);


// Write to the shared memory segment
sprintf(shared_memory, "Hello, world!");


// Detach the shared memory segment from the address space of the process
shmdt(shared_memory);


// Client process attaches to the shared memory segment
int shm_id = shmget(key, size, 0666);


// Attach the shared memory segment to the address space of the process
```

```
char* shared_memory = shmat(shm_id, NULL, 0);


// Read from the shared memory segment

printf("Message from shared memory: %s\n", shared_memory);


// Detach the shared memory segment from the address space of the
process

shmdt(shared_memory);


// Server process removes the shared memory segment

shmctl(shm_id, IPC_RMID, NULL);
```

In this example, a server process creates a shared memory segment using shmget() and attaches it to its address space using shmat(). It then writes a message to the shared memory segment.

A client process attaches to the same shared memory segment using shmget() and shmat(), and reads the message from the shared memory segment.

Finally, the server process removes the shared memory segment using shmctl().

Sharing mechanisms such as copy-on-write, memory-mapped files, and shared memory are essential components of modern operating systems. They allow processes to communicate and share data efficiently and securely. Each mechanism has its own advantages and disadvantages, and choosing the right one depends on the specific requirements of the application.

10 Case Study: Memory Management in Linux

In this chapter, we will begin by providing an overview of Linux's memory management approach, including its memory hierarchy and virtual memory. We will then compare Linux's approach to memory management with other operating systems. Finally, we will discuss the impact of Linux's memory management on its performance and reliability.

10.1 Overview of Linux's approach

Linux's approach to memory management is one of the key reasons for its popularity and success. The memory management subsystem in Linux is responsible for managing the allocation and deallocation of memory resources to various processes in a fair and efficient manner. In this chapter, we will provide an overview of Linux's approach to memory management.

The Linux memory management system is based on the virtual memory concept, which allows the system to manage more memory than is physically available. The virtual memory subsystem in Linux maps the physical memory to a process's virtual address space. This mapping is done using a page table, which is a data structure that maps virtual addresses to physical addresses.

Linux uses a demand-paging approach, which means that pages are loaded into memory only when they are accessed. When a process accesses a page that is not currently in memory, a page fault is triggered, and the page is loaded from the disk into memory. This approach minimizes the amount of memory required by a process, as only the pages that are needed are loaded into memory.

Linux provides various memory allocation algorithms to manage memory resources efficiently. One such algorithm is the Slab Allocator,

which is a fast and efficient memory allocator that is optimized for allocating small objects.

The Linux memory management system also includes a number of page replacement algorithms, which are used to decide which pages to evict from memory when the system is low on memory. Some of the page replacement algorithms used in Linux include the Least Recently Used (LRU) algorithm, the Random algorithm, and the Clock algorithm.

Linux also provides various tools and commands that can be used to monitor and manage memory usage. One such tool is the top command, which provides real-time information about the memory usage of processes running on the system.

In addition to these features, Linux also supports various advanced memory management techniques such as memory compression, memory deduplication, and transparent huge pages. These techniques are designed to optimize memory usage and improve system performance.

Overall, Linux's approach to memory management is robust, efficient, and highly configurable. The use of virtual memory, demand-paging, and advanced memory management techniques ensures that Linux can efficiently manage memory resources, even on systems with limited physical memory.

10.2Comparison with other operating systems

Memory management is a critical component of an operating system, as it determines how the system manages its memory resources. Different operating systems use different approaches to manage their memory resources, and it is essential to understand the pros and cons of each approach.

In this chapter, we will compare the memory management approaches of different operating systems and analyze their strengths and weaknesses.

First, let's look at Windows. Windows uses a demand paging system, where pages are loaded into memory when they are needed. Windows also uses a page file to swap out pages to disk when there is not enough physical memory available. One of the strengths of Windows' memory management is its ability to handle large amounts of memory effectively. However, Windows can suffer from memory fragmentation, which can lead to a slowdown in performance.

Next, let's consider macOS. Like Windows, macOS uses a demand paging system, but it also has a feature called memory compression, which compresses memory pages to save space. macOS also uses a page file, similar to Windows. One of the strengths of macOS's memory management is its ability to handle low memory situations effectively. However, macOS can also suffer from memory fragmentation, which can cause performance issues.

Linux, on the other hand, uses a different approach to memory management. Linux uses a combined demand paging and pre-fetching system, where pages are loaded into memory before they are needed. Linux also uses a swap space to swap out pages when there is not enough physical memory available. One of the strengths of Linux's memory management is its ability to handle a large number of processes effectively. Linux also has better memory utilization compared to other operating systems. However, Linux can also suffer from memory fragmentation, which can cause performance issues.

Finally, let's consider FreeBSD. FreeBSD uses a demand paging system, similar to Windows and macOS, but it also has a feature called UMA (Unified Memory Architecture), which is a memory allocator that manages both kernel and user-space memory. FreeBSD also uses a swap space to swap out pages when there is not enough physical memory available. One of the strengths of FreeBSD's memory management is its

ability to handle a large number of processes effectively. FreeBSD also has better memory utilization compared to other operating systems. However, like other operating systems, FreeBSD can also suffer from memory fragmentation.

In conclusion, each operating system uses a different approach to manage its memory resources, and each approach has its strengths and weaknesses. Windows and macOS use a demand paging system, while Linux and FreeBSD use a combination of demand paging and pre-fetching. Windows and macOS both use a page file, while Linux and FreeBSD use a swap space. Linux and FreeBSD have better memory utilization and can handle a larger number of processes effectively, but all operating systems can suffer from memory fragmentation, which can cause performance issues.

11 Conclusion

In conclusion, memory management is a critical component of any operating system. It enables processes to access the resources they need while ensuring the system's overall stability and performance.

This chapter provided an overview of the different aspects of memory management, including address spaces and memory allocation, paging and page replacement algorithms, segmentation and compaction, and memory protection and sharing. We also looked at Linux's approach to memory management and compared it with other operating systems.

It's important for system administrators and developers to have a good understanding of memory management concepts and techniques to ensure optimal system performance and stability. By implementing efficient memory management strategies, it's possible to achieve a balance between resource utilization and system responsiveness, ultimately leading to a better user experience.