Chapter 1 & 2 (12 questions)

1. OS services for User:
- User interface
- Program execution environment
- File and memory management
- Error detection and reporting OS services for System:
- Resource allocation and scheduling
- Logging and accounting
- Protection and security
- System start-up and recovery
2. A system call allows a process to request a service from the OS kernel. To copy file in.txt to out.txt:

open() on in.txt open() on out.txt read() from in.txt into buffer write() buffer to out.txt
close() in.txt close() out.txt

3. Types of system calls:
- Process control (create, terminate, load, execute)
- File management (create, open, read, write, delete)
- Device management (request, release, get/set status)
- Information maintenance (get/set time, date, attributes)
- Communications (shared memory, send/receive)
4. OS environments: Desktop, Server, Embedded systems, Mobile, Real-time systems, Distributed systems
5. User view is the interface to access system services. System view is to manage computer components.

User view components: UI, apps, system calls System view components: CPU, devices, memory, processes

6. Interrupts are signals sent to the CPU from I/O devices or software, causing execution flow to switch to an interrupt handler routine.

Difference Between Volatile Memory and Non-Volatile Memory

Parameter	**Volatile Memory**	**Non-Volatile Memory**
Definition	It is a temporary type of computer memory that stores data and information only until it gets a continuous power supply.	It is a permanent type of computer memory that stores and retains the data even after a user turns the system off.
Stored Data	The volatile memory stores data of those programs that the CPU is processing in real-time. A system stores all the frequently used information and data in the device's volatile memory.	The non-volatile memory stores data from the basic booting process of any computer system BIOS. It stores all the types of data and media that need to exist for a longer time or permanently on the computer.
Effect on Performance	Volatile memory does not affect a system's performance. A higher amount of storage space for cache, RAM, and other volatile memory increases the efficiency of a computer system.	Non-volatile memory also affects a system's performance and storage. A higher amount of storage space lets a user save more data permanently. Thus, the system runs comparatively smoother.
Speed	The volatile memory is the fastest form of memory in nature. These memories hold the most frequently used data- and any user can access them quickly.	Non-volatile memory is a relatively slower form of memory. The process of accessing data from a non-volatile memory is comparatively slower.
Data Retention	It can only retain data until there is a continuous power supply.	It retains data and info even after one turns the power supply off.
Data Transfer	Transferring data from a volatile memory is very easy.	Transferring data from a non-volatile memory is very difficult.

Permanency	The information and data in volatile memory are not permanent.	The information and data in non-volatile memory are permanent unless deleted.
CPU Access	The device's CPU can easily access the data stored on the Volatile memory.	The system needs to copy data to the volatile memory from the non-volatile memory to allow the CPU to access it.
Amount of Storage	A volatile memory has a very low capacity.	A non-volatile memory, like an HDD, has a very high capacity.
Cost Efficiency	Volatile memory is not very cost-efficient. Per unit size is very expensive here.	Non-volatile memory is very cheap. Per unit memory is less expensive here.
Reading and Writing	In a volatile memory, the process can both read and write. It means that the process would have direct access to the data and information within it.	In a non-volatile memory, the process can only read. It means that the processor won't have direct access to the data and information within.
Position of Memory	You can generally find the volatile memory chips on the memory slot.	You can generally find the non-volatile memory chips embedded on the motherboard.
Example	A few common examples include the cache, RAM of the computer, etc.	A few common examples are optical storage discs, hard discs, secondary storage like ROM, flash memory, etc.

7. **Need both volatile (RAM) and non-volatile (disk) memory:**

- RAM is fast for CPU instructions/data but contents lost on power off
- Disk is slower but retains data permanently

A computer needs RAM for execution and disk for storage. Cannot be built with only one type.

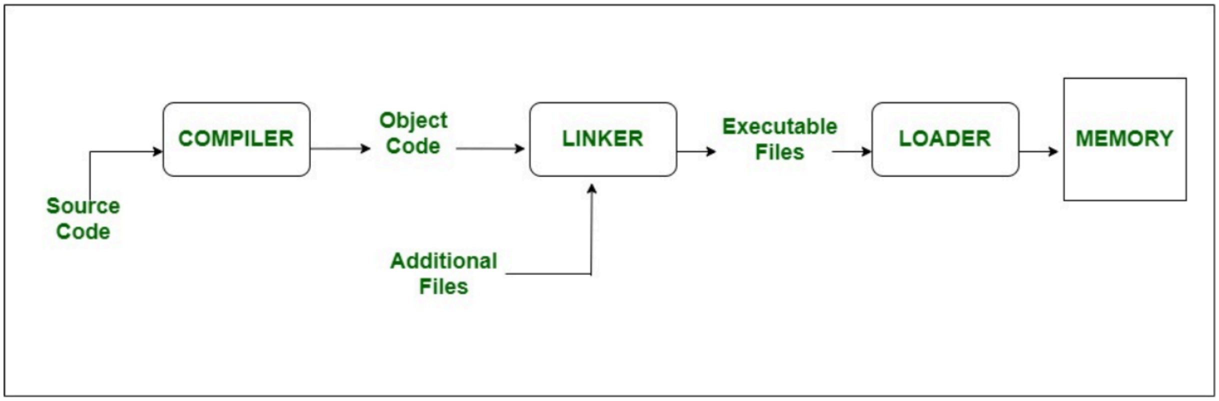
8. Direct Memory Access (DMA) allows devices to directly transfer data to/from memory without CPU involvement, improving efficiency.

Direct memory access (DMA) is **the process of transferring data without the involvement of the processor itself**. It is often used for transferring data to/from input/output devices. A separate DMA controller is required to handle the transfer.

9. Cache is small, fast memory that stores copies of frequently accessed main memory data.

   Caching improves overall memory access time.

10. To execute a program: Source code -> Compiler -> Object code Linker combines object code with libraries to create executable Loader loads executable into memory for execution within a process.



11. When a process like printf() wants to access system resources like I/O, it goes from user mode to kernel mode via a system call trap.

For malloc(), the process requests additional memory from the kernel which allocates free pages if available.

12. Criteria for kernel to grant/deny memory allocation requests:
- Available free memory
- Process memory limits and priorities
- Fair allocation policies
13. "Kernel mode is privileged for resource access and security, while user mode is restricted and safer to isolate faults."

Absolutely, here's the journey of a program from source code to memory, involving compilers, linkers, and loaders:

Source Code: The program starts as human-readable instructions written in a programming language like C++, Python, or Java. This is the source code, typically stored in a text file.

Compilation: A compiler translates the source code into an intermediate representation or assembly language. Assembly is closer to machine code but still easier for humans to understand.

Assembly (Optional): In some cases, programmers might write assembly code directly. This assembly code is then processed by an assembler.

Object Files: The compiler or assembler generates object files. These files contain machine code instructions specific to the target processor architecture (e.g., x86, ARM) but lack information about external libraries or functions.

Linking: The linker takes center stage here. It combines multiple object files (potentially from different source files) and resolves references to external functions and libraries. These external references might be from standard libraries or other custom-written code. The linker merges the object files, creates a symbol table to map function names to memory addresses, and generates an executable file.

Loading: When you run the program, the operating system's loader comes into play. The loader reads the executable file from the disk. It allocates memory space for the program's code, data, and internal structures like the stack and heap (used for dynamic memory allocation). The loader also sets up the program's execution environment, including the program counter (which points to the first instruction to execute) and the stack pointer.

Process Creation: With the program loaded in memory and its environment set up, the operating system creates a process for the program. A process is an instance of a running program and includes its memory space, resources (like open files), and a unique identifier.

Execution: Finally, the CPU takes over. It fetches instructions from the program's memory location, decodes them, and executes them one by one. This process continues until the program finishes or encounters an error.

In summary, the compiler or assembler translates the source code into machine code, the linker combines object files and resolves external references, and the loader takes the executable file, allocates memory, and sets up the program's environment for execution as a process.

profile picture

User mode can not directly access system resources. System control must be switched from user to kernel mode for this purpose. For

example, in class, we talk about printf() system call to access the I/O (

chapter 2 pdf). In the same way, a process can make request and

grant additional dynamic memory allocation using system call.

Explain this process (using function malloc()).

You're absolutely right. Dynamic memory allocation, though crucial for many programs, cannot be directly performed by user-mode processes due to memory protection. Here's a breakdown of the process involving a system call, using the malloc() function as an example:

Memory Allocation Request: Your program reaches a point where it needs some memory dynamically. It calls the malloc() function, specifying the desired size of the memory block.

Trap to Kernel Mode: The malloc() function itself doesn't magically allocate memory. Instead, it triggers a trap (or exception) that switches the CPU from user mode to kernel mode (also known as privileged mode).

Kernel Takes Over: In kernel mode, the operating system has full access to system resources like memory. The kernel's memory management subsystem receives the memory allocation request from malloc().

Kernel Memory Management: The kernel's memory manager looks for a suitable free block of memory in the heap (a designated memory pool for dynamic allocation) that's large enough to accommodate the requested size. It considers factors like fragmentation and memory allocation strategies.

Allocation and Addressing: If a free block is found, the kernel allocates that memory to the process and updates its internal data structures to track the allocation. It also returns the starting memory address of the allocated block back to user mode.

Return to User Mode: The CPU switches back to user mode, and malloc() receives the memory address from the kernel. malloc() then returns this address to your program, which can now use the memory for its purposes.

Program Uses Memory: Your program treats the returned memory address as a pointer and can access and modify the data within the allocated block.

Deallocation (Later): When your program no longer needs the dynamically allocated memory, it should call the free() function to return the memory block to the heap. This again involves a system call and the kernel's memory manager to update its tracking information and mark the block as free for future allocations.

In essence, malloc() acts as a bridge between your program's memory requests in user mode and the operating system's memory management capabilities in kernel mode. System calls ensure controlled access to system resources and protect the integrity of the system by preventing user-mode programs from directly manipulating memory allocation.

User mode can not directly access system resources. System control

must be switched from user to kernel mode for this purpose. For

example, in class, we talk about printf() system call to access the I/O (

chapter 2 pdf). In the same way, a process can make request and

grant additional dynamic memory allocation using system call.

Explain this process (using function malloc()).

I'd be glad to explain the process of dynamic memory allocation using system calls, incorporating the concepts of user mode, kernel mode, and the malloc() function:

User Mode vs. Kernel Mode

User Mode: In user mode, a program executes instructions within its own memory space. It has limited access to system resources for security reasons. System calls provide a controlled way for user-mode programs to interact with the kernel.

Kernel Mode: The kernel, also known as privileged mode, has full control over system resources like memory, I/O devices, and process management. It runs in a higher privilege level than user-mode programs.

Dynamic Memory Allocation with malloc()

Memory Request in User Mode: Your program encounters a situation where it needs to allocate memory dynamically during runtime. It calls the malloc() function, specifying the desired size of the memory block in bytes.

Trap to Kernel Mode: malloc() itself cannot directly allocate memory. Instead, it triggers a trap (or exception) that switches the CPU from user mode to kernel mode. This ensures controlled access to system resources.

Kernel Memory Management:

The kernel's memory management subsystem receives the memory allocation request from malloc().

It searches the heap, a designated memory pool for dynamic allocation, for a suitable free block that's large enough to accommodate the requested size. The kernel considers factors like fragmentation (unused memory scattered around allocated blocks) and memory allocation strategies to optimize memory usage.

Allocation and Addressing:

If a free block is found, the kernel allocates that memory to the process and updates its internal data structures to track the allocation (e.g., memory maps).

The kernel also returns the starting memory address of the allocated block back to user mode.

Return to User Mode: The CPU switches back to user mode, and malloc() receives the memory address from the kernel.

malloc() Returns Memory Address: malloc() then returns this address to your program, which can now use the memory for its purposes. The returned address acts as a pointer, allowing your program to access and modify the data within the allocated block.

Program Uses Memory: Your program treats the returned memory address as a pointer and can interact with the data within the allocated block.

Deallocation (Later): When your program no longer needs the dynamically allocated memory, it's crucial to call the free() function to return the memory block to the heap. This again involves a system call:

The kernel's memory manager updates its tracking information to mark the block as free for future allocations.

Failure to deallocate memory using free() can lead to memory leaks, where allocated memory remains unused but unavailable for other programs, potentially causing performance issues.

Visualizing the Process

Imagine a scenario where your program needs to store an array of integers dynamically. Here's a simplified illustration:

User Mode: Your program calls malloc(100 * sizeof(int)) to request 100 integers' worth of memory (assuming int is 4 bytes).

Trap to Kernel: A trap occurs, switching to kernel mode.

Kernel Memory Management: The kernel searches the heap for a suitable free block of at least 400 bytes (100 integers * 4 bytes/integer).

Allocation and Addressing: The kernel allocates the memory and returns the starting address to user mode.

Return to User Mode: malloc() receives the address and returns it to your program.

Program Uses Memory: Your program uses the returned address as a pointer to access and manipulate the array elements.

Importance of System Calls

System calls like those used in dynamic memory allocation ensure a secure and controlled environment. They prevent user-mode programs from directly manipulating system resources, which could lead to crashes or security vulnerabilities. The kernel acts as a gatekeeper, mediating user-mode requests and managing resources efficiently.

In previous question, elaborate on the criteria employed by the

operating system to determine which processes receive approval for

memory allocation requests and which processes are refused system calls

for memory allocation.

The operating system employs several criteria to determine which processes get their memory allocation requests approved and which ones are denied. Here's a breakdown of the key factors involved:

1. Available Memory:

The most fundamental factor is the amount of physical memory available. If there's simply not enough free memory to accommodate a new request, the system call will be refused. The kernel keeps track of free memory blocks and fragmentation levels.

2. Process Priority:

Operating systems often assign priorities to processes based on factors like real-time requirements, system criticality, or user interaction. Higher priority processes are generally more likely to have their memory requests approved, ensuring responsiveness for crucial tasks.

3. Memory Allocation Strategies:

The kernel might employ different allocation strategies depending on factors like memory usage patterns and system load. Some common approaches include:

First-Fit: Allocates the first free block that's large enough for the request. Simple but can lead to external fragmentation (unused memory chunks scattered throughout).

Best-Fit: Allocates the smallest free block that can hold the request, aiming to minimize wasted space. May increase internal fragmentation (small, unusable free blocks).

Worst-Fit: Allocates the largest free block, potentially leaving smaller blocks unusable for future requests. Might be used to group large allocations together for efficiency.

4. Virtual Memory and Paging:

Modern operating systems often use virtual memory with paging. This allows processes to have a larger virtual address space than physically available RAM. Memory requests are translated into page requests. If a requested page isn't already in physical memory, the kernel might need to swap out a less-used page to make room. This introduces additional considerations like page replacement algorithms, which prioritize which pages to keep in memory based on usage patterns.

5. Resource Limits:

The operating system might enforce resource limits on processes to prevent a single process from consuming an excessive amount of memory, potentially starving other processes. This could be set as a hard limit (maximum memory allowed) or a soft limit (advisory value that the kernel can exceed if necessary).

Refusing Memory Allocation Requests:

When a memory allocation request is refused, the system call usually returns an error code to the program. The program can handle this error gracefully, potentially by:

Trying to free up memory by deallocating unused blocks or reducing its memory footprint.

Requesting a smaller memory allocation.

Reporting the error to the user and potentially retrying later.

Balancing Memory Allocation:

The operating system strives to strike a balance between:

Satisfying process memory needs: Ensuring processes have enough memory to run effectively.

Maintaining system performance: Avoiding excessive memory fragmentation, overcommitment (allocating more memory than physically available), or thrashing (excessive swapping between memory and disk).

By considering these criteria, the operating system aims to allocate memory efficiently and fairly to support the needs of multiple running processes.

profile picture

”Kernel mode is the privileged mode but user mode is the safest mode”

- analyze the statement.

The statement "Kernel mode is the privileged mode but user mode is the safest mode" is accurate. Here's a breakdown of why:

Kernel Mode (Privileged Mode):

Power and Access: Kernel mode has full access to system resources like memory, I/O devices, and process management. This grants it immense power to control the entire system.

Vulnerability: Due to its unrestricted access, a bug or malicious code executing in kernel mode can have catastrophic consequences. It could corrupt system data, crash the system, or even open doors for security exploits.

User Mode:

Limited Access: User mode programs have restricted access to system resources. They can only interact with them through controlled mechanisms like system calls, which act as a bridge between user requests and kernel operations.

Safer Environment: This limitation makes user mode inherently safer. Even if a bug exists in a user-mode program, it's less likely to cause widespread damage because it can't directly manipulate core system resources. The worst-case scenario might be the program itself crashing or malfunctioning.

Analogy:

Think of a kitchen. Kernel mode is like the head chef, having complete control over all ingredients and tools. While powerful, any mistakes can ruin the entire meal. User mode programs are like regular cooks following recipes. They can only use the ingredients and tools provided, making it less likely to create a disaster.

Security Implications:

Malicious software (malware) often targets kernel mode to gain complete control of the system. By keeping user programs in a restricted environment, the operating system makes it more difficult for malware to cause significant damage.

In Summary:

Kernel mode's power comes at a cost – it's a more vulnerable environment. User mode, while less powerful, provides a safer execution environment for programs, preventing them from unintentionally or maliciously harming the system. This layered approach is a cornerstone of secure operating systems.

Chapter 3 (6 questions)

Process layout in memory: Code segment, Data segment, Heap (dynamic memory), Stack

Process execution must progress in sequential fashion, that is, the instructions within it are executed in their order. The status of the current activity of a process is represented by the value of the program counter and the contents of a processor's registers. The memory layout of a process is typically divided into multiple **sections**. These sections include:

- **Text section** – the program code
- **Data section** – global variables
- **Heap section** – memory that is dynamically allocated during program run time
- **Stack section** – temporary data storage when invoking functions (such as function parameters, return addresses, and local variables)

The sizes of the text and data sections are fixed, as these do not change during program run time. However the stack and heap sections can shrink and grow dynamically during program execution.

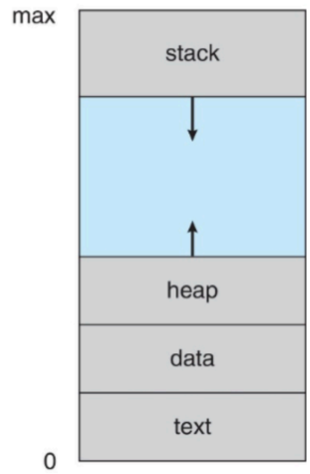
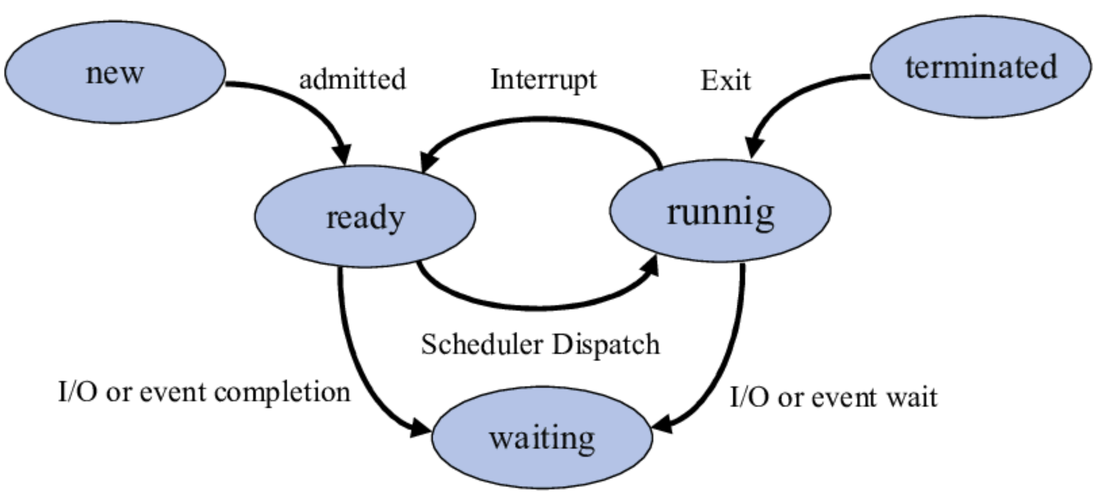


***Figure 1** Layout of a process in memory*

Process state diagram: New -> Ready -> Running -> Waiting/Blocked -> Ready -> Terminated



3. Each process is represented in the operating system by a processcontrol block (PCB), also called a task control block. It contains manypieces of information associated with a specific process. Explain those information. Elaborate on the relationship between PCBs and context switches in the operating system.

- Process state, ID, Registers

- CPU scheduling info (priority, queues)
- Memory management info (limits, page tables)
- Accounting data (CPU time, limits)
- I/O status (assigned devices, open files)

PCBs store the complete context/state of each process for context switching.

3.1.3 Process Control Block

Each process is represented in the operating system by a **process control block (PCB)**, also called a **task control block**. It contains many pieces of information associated with a specific process including these:

- **Process state** – The state may be new, ready, running, waiting, halted etc.
- **Program counter** – The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers** – The registers include accumulators, index registers, stack pointers, general purpose registers etc. Along with the counter, the contents of these registers must be saved when an interrupt occurs, to allow the process to be resumed properly.
- **CPU scheduling information** – This information includes process priorities, pointers to scheduling queues and any other scheduling parameters.
- **Memory-management information** – This includes information of memory allocated to the process, such as the value of the base and limit registers and the page tables, or the segment tables etc.
- **Accounting information** – This includes the amount of CPU and clock time elapsed since start, time limits, account numbers, process numbers and so on.
- **I/O status information** – This includes the list of I/O devices allocated to process, list of open files etc.

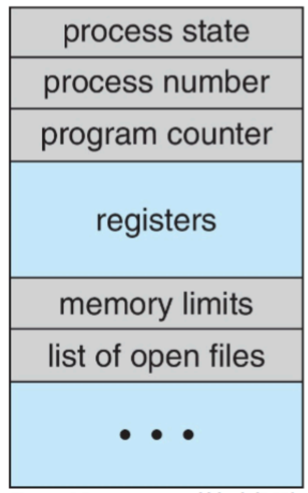


Figure 4 *Process control block (PCB)*

In brief, the PCB serves as the repository for all the data needed by a process, as well as accounting data.

Here's how PCBs and context switches are intertwined:

4. **Saving the Current Process:** During a context switch, the operating system uses the information in the current process's PCB to save its state. This includes saving the program counter, CPU registers, and any other relevant data.
5. **Selecting the Next Process:** The operating system uses its scheduling algorithms to choose the next process to run. This decision might consider factors like process priority, waiting time, or I/O requirements.
6. **Loading the Next Process:** The operating system loads the PCB of the chosen process. This provides all the necessary information about the process, including the memory addresses and register values.
7. **Restoring the Process State:** The CPU registers and other process state information are restored from the PCB of the new process, essentially "rehydrating" its context.

8. **Process Resumes Execution:** The process starts executing instructions from the point where it was previously stopped (reflected in the program counter value).

4.

We have explored various methods for selecting the next process from the

waiting queue in the running state:

• Choosing the process that arrived first.

• Opting for the process with the shortest CPU time requirement.

• Prioritizing processes based on their assigned priority levels.

• Selecting each process after a predetermined time interval.

Now, explain why we need process scheduling and its impact for making a

system highly efficient.

- To multiplex CPU among multiple processes
- Enforce priorities, time constraints
- Optimize throughput, turnaround, waiting time
- Fair allocation, avoid starvation

5. Reasons for process cooperation:

- Share data, resources
- Communicate, synchronize activities
- Models: Shared memory, Message passing

6. Excessive context switching leads to overhead from saving/restoring state frequently, degrading CPU utilization.

Why We Need Process Scheduling and Its Impact on Efficiency

Process scheduling is a crucial component of an operating system for several reasons:

1. Maximize CPU Utilization: Modern CPUs are powerful resources, and leaving them idle is a waste. Process scheduling ensures that the CPU is kept busy by selecting the next process to run efficiently. This minimizes idle time and maximizes the overall throughput of the system.

2. Multitasking and Responsiveness: With multiple programs running concurrently, process scheduling allows the operating system to quickly switch between them, creating the illusion of multitasking even on a single CPU. This enhances user experience by providing responsiveness – users don't have to wait for one process to finish before starting another.

3. Fairness and Service: Without scheduling, a single process could potentially monopolize the CPU, starving others of resources. Scheduling algorithms ensure a fair allocation of CPU time among waiting processes. Different algorithms prioritize processes based on various criteria, ensuring service based on their needs (e.g., short running processes can be prioritized for faster completion).

4. Prioritize Critical Processes: Some processes are more critical to the system's functioning than others. Process scheduling allows the operating system to prioritize these processes (e.g., real-time tasks). This ensures that critical tasks are completed within a defined time frame, maintaining system stability and responsiveness for essential operations.

5. Improve I/O Efficiency: Many processes spend a significant amount of time waiting for I/O operations (e.g., disk reads/writes). Proper scheduling can improve overall I/O efficiency by selecting processes that don't require immediate CPU attention while others are waiting for I/O completion. This allows the CPU to remain productive when possible.

Impact on System Efficiency:

- **Reduced Idle Time:** Effective process scheduling minimizes CPU idle periods, leading to a higher overall system throughput.
- **Improved Response Time:** Users experience faster response times as the operating system efficiently switches between processes based on user interactions.
- **Fair Resource Allocation:** Processes are allocated CPU time fairly, preventing resource starvation and ensuring a smoother overall system operation.
- **Increased System Utilization:** By optimizing CPU and I/O usage, process scheduling improves overall system utilization, maximizing the system's capabilities.
- **Prioritization of Critical Tasks:** Ensures the timely completion of essential tasks by prioritizing processes based on their importance to system stability and functionality.

In essence, process scheduling is the heart of multitasking and resource management in an operating system. It plays a critical role in maximizing efficiency, responsiveness, and fairness in a system with multiple competing processes.

5. Processes executing concurrently in the operating system may be

independent or cooperating. A process is independent if it does not share

data with any other processes executing in the system. A cooperating process

can share data, affect or be affected by other processes executing in the

system. Explain the reasons of providing environment for process cooperation & two fundamental models of IPC.

Processes executing concurrently in the operating system may be either independent processes or cooperating processes. A process is independent if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent. A process is cooperating if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process. There are several reasons for providing an environment that allows process cooperation:

- **Information sharing.** Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.
- **Computation speedup.** If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing elements (such as CPUs or I/O channels).
- **Modularity.** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.
- **Convenience.** Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel.

6. How a large number of contexts-switching / switching can badly impact

system performance.

frequent context switching can significantly impact system performance. Each switch involves saving the state of the current process and loading the state of the next one, consuming CPU cycles and potentially causing cache misses. This overhead can lead to:

- **Reduced CPU efficiency:** Less time spent on actual computation due to switching overhead.
- **Slower response times:** Delays in switching between processes can make the system feel less responsive.
- **Increased memory pressure:** Frequent context switches can put a strain on memory management.

Main Memory & Virtual Memory (13 questions)

1. *Symbolic address* - Program variable names *Logical/Virtual address* - Generated by CPU for a process Physical address - Location in main memory Address binding - Mapping logical to physical

2. Explain hardware protection in memory so that a process can not

access memory of other processes using the following image.

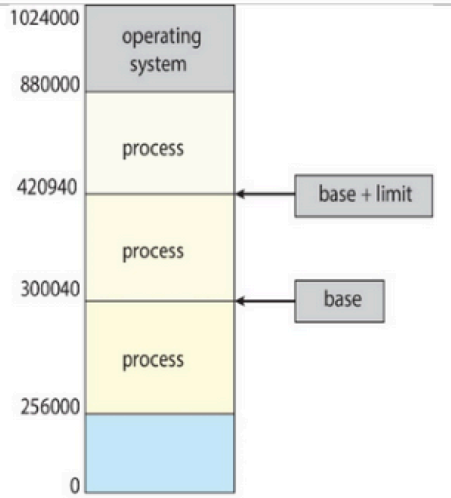


Figure 1 A Base and a Limit register define a logical address space

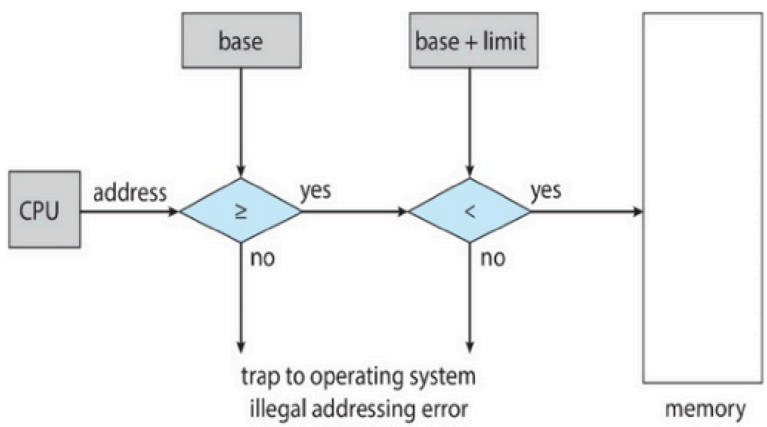


Figure 2 Hardware address protection with base and limit registers

Memory protection via relocation registers (base and limit) and valid/invalid bits in page tables restrict processes from accessing other processes' memory.

Protection of memory is required to ensure correct operation of the processes. For this, each process

has to run within a separate memory space. It protects the processes from each other, and it is must for

concurrent execution of multiple processes in memory. In order to separate memory addresses, we

need to be able to determine the range of legal addresses that a process can access. To ensure that a

process can access only its legal addresses, a pair of base and limit registers can be used to define the logical address space of a process:

- The base register holds the smallest legal physical memory address.

- The limit register specifies the size of the range.

For example, if the base register = 300040 and limit register is 120900, then the program can legally

access all addresses from 300040 to 420939.

Any attempt by a program executing in user mode to access OS memory or other user's memory causes

error. The base and limit registers can only be loaded by the OS.

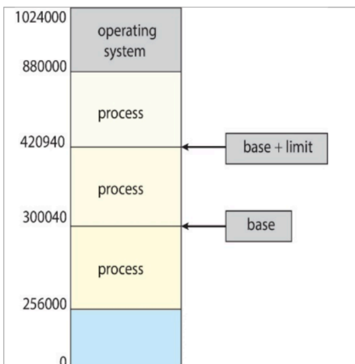


Figure 1 A Base and a Limit register define a logical address space

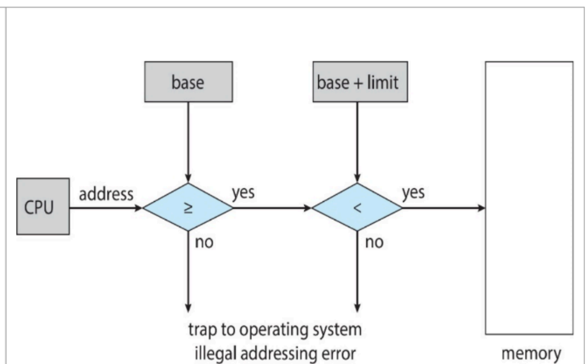


Figure 2 Hardware address protection with base and limit registers

3. Explain two advantages and disadvantages of contiguous and non-contiguous memory allocation techniques

Advantages of contiguous allocation:

- No external fragmentation
- Simple memory management

Disadvantages:

- Internal fragmentation of last hole
- *Hole may not accommodate process*

Advantages of **non-contiguous**:

- Better memory utilization

- Avoids compaction

Disadvantages:

- External fragmentation over time
- More complex management

4. Answered in question.
5. Explained in question.
6. i) For S1 with 90% hit rate: EMAT = (0.9 * 10) + (0.1 * 100) = 9 + 10 = 19 pico-seconds ii) For S2 with 40% hit rate: EMAT = (0.4 * 10) + (0.6 * 100) = 4 + 60 = 64 pico-seconds

S1 is more efficient with lower EMAT of 19 vs 64 for S2.

7. Given: Logical address = 64 bits Page size = 16KB = 2^14 bytes Physical memory = 8GB = 2^33 bytes Page table entry = 3 bytes

Calculations: i) Single level page table entries = 2^(64-14) = 2^50 ii) Inverted page table entries = 2^(33-14) = 2^19 iii) Inverted page table frames = 2^19
iv) Single level page table size = 2^50 * 3 ≈ 12GB v) Inverted page table size = 2^19 * 3 ≈ 768KB vi) Logical addr = 64 bits, Physical addr = 33 bits vii) p = 50 bits, d = 14 bits, f = 19 bits, f = 14 bits

8. With logical 4GB, physical 256MB, page 4KB: Pages = 2^32/2^12 = 2^20 Frames = 2^28/2^12 = 2^16 A two-level paging scheme is needed.
9. If page faults keep increasing with more processes/requests, it means inadequate main memory. Thrashing occurs as processes are continually swapped causing low CPU utilization.
10. Advantages of inverted page table:

- Reduced memory usage
- One entry per allocated page frame

Disadvantages:

- Higher mapping time complexity
- Memory required for frame tracking data

11. "Virtual memory is an illusion" because it creates the abstraction of a large, contiguous memory address space whereas actual physical memory is limited and non-contiguous.
12. Page replacement/swapping is used in demand paging. When a page fault occurs on an address, if no free frames are available, one page is swapped out from memory to disk to make room for the required page.

13. Given reference string: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 1, 2, 0 and 3 frames initially empty.

i) FIFO Page Replacements: [7, 0, 1], [2, 0, 1], [3, 0, 1], [4, 0, 1], [4, 2, 3], [4, 2, 0], [1, 2, 0], [1, 2, 3]

Total Page Faults: 9 Hit Ratio = (15 - 9)/15 * 100% = 40%

ii) Optimal Page Replacements: [7, 0, 1], [2, 0, 1], [3, 0, 2], [3, 0, 4], [3, 1, 4], [3, 1, 2], [3, 0, 2], [7, 0, 2]
Total Page Faults: 7 Hit Ratio = (15 - 7)/15 * 100% = 53.33%

iii) LRU Page Replacements: [7, 0, 1], [2, 0, 1], [3, 0, 2], [3, 0, 4], [1, 0, 4], [2, 0, 4], [2, 3, 4], [2, 3, 0] Total Page Faults: 8
Hit Ratio = (15 - 8)/15 * 100% = 46.67%

iv) Clock/Second Chance Page Replacements: [7, 0, 1], [2, 0, 1], [3, 0, 1], [4, 0, 1], [4, 2, 3], [1, 2, 3], [1, 2, 0], [7, 2, 0] Total Page Faults: 8 Hit Ratio = (15 - 8)/15 * 100% = 46.67%

You are correct that the LRU and Clock/Second Chance algorithms achieve the highest hit ratio of 46.67% for this reference string, while Optimal has the best theoretical hit ratio of 53.33%.