MSE – II Operating System-2

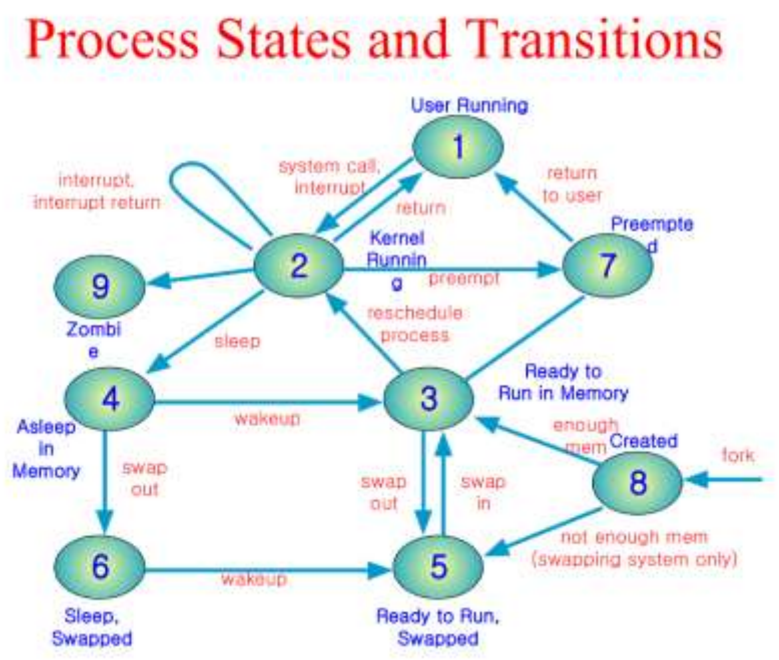
A. Question – Explain in detail the process state transition in an Unix OS, and draw a neat diagram illustrating the various states a process can go through during its execution..

Answer –

Process: In simple terms, a process is like a running instance of a program. Imagine you have a game installed on your computer. When you double-click to play it, a process starts running, allowing you to interact with and play the game. That's a process—a program in action.

State: Think of the state as the condition or situation a process is in at any given time. Like in real life, a process can be doing different things at different times—sometimes active, sometimes waiting, and so on. The states describe what the process is up to.

State Transition: When a process changes from one state to another, it's called a state transition. Just like when you switch from playing a game to checking your email, the process (the game) changes its state from active to paused or suspended.



1. **Newly Created (8)**: This is when a process is born but hasn't started running yet. It's like a newborn baby waiting for its turn to enter the world.

2. **Ready to Run in Memory (3)**: Once the system is ready to give it some attention, the process gets loaded into memory and waits for its turn to run. It's like being in a waiting room, ready to jump into action when called.

3. **User Running (1)**: Now, the process gets the spotlight. It's actively doing its thing, executing instructions in user mode. Imagine the game running on your screen, responding to your commands.
4. **Kernel Running (2)**: Sometimes, the process needs extra help from the system itself. That's when it switches to kernel mode to execute some system-level instructions, like asking for access to a file. It's like the game asking the computer for more memory to load a new level.
5. **Asleep in Memory (4)**: If the process needs to wait for something, like data from a slow hard drive, it goes into a sleep state while still sitting in memory. It's like putting the game on pause while waiting for a friend to join.
6. **Ready to Run, Swapped (5)**: When the system is low on memory, it might swap out less active processes to disk to make room for more important ones. So, the process is ready to run but chilling out on the disk for now.
7. **Sleep, Swapped (6)**: Similar to being asleep in memory, but now the process is swapped out to disk. It's like the game taking a nap in a file on your computer until you're ready to play again.
8. **Preempted (7)**: Sometimes, the system needs to interrupt a running process to let another one have a turn. It's like pausing the game to check your messages before resuming.
9. **Zombie (9)**: Finally, when a process completes its job, it becomes a zombie—it's technically dead but still hanging around in the process table until its parent process (like the game launcher) reads its exit status. It's like finishing a game level but still seeing your score on the screen.

Each of these states and transitions happens based on events like starting a process, completing a task, or needing system resources. It's like a dance of processes, each waiting for its turn on the stage of your computer's memory and CPU.

B. Question - Evaluate the significance of the process address space manipulation in Unix, explaining how it enables memory allocation, deallocation, and sharing among processes.

Answer –

Process: In computing, a process is essentially a running instance of a program. It's like when you open a web browser on your computer—the browser program starts running as a process, allowing you to browse the internet. Each process has its own virtual address space, which is a range of memory addresses that it can use to store data and instructions.

Process Address Space: This is the memory space allocated by the operating system for a process. It includes different sections like text (for program instructions), data (for variables and other runtime data), stack (for function calls and local variables), and more. Each process has its own address space, isolated from other processes for security and stability.

Memory Manipulation: This refers to the actions taken by the operating system to manage the memory allocated to processes. This includes tasks like allocating memory when a process is created, deallocating memory when a process terminates, and allowing processes to share memory when needed.

The process address space manipulation in Unix is significant because it enables memory allocation, deallocation, and sharing among processes. When a process is created, the kernel allocates a region of memory for the process, which includes the text, data, stack, and other necessary components. This region is managed by the per process region table, which contains information about the region's size, location in physical memory, and state.

When a process needs to allocate more memory, such as when a shared memory region is created or a process is forked, the kernel can allocate a new region and attach it to the process's address space. Similarly, when a process is terminated or a shared memory region is detached, the kernel can free the associated region.

The process address space manipulation also allows for memory sharing among processes. When a process forks, the child process inherits the parent's memory regions, and the reference count of the regions is incremented to reflect the additional process referencing them. If the region is not shared, the kernel must physically copy it for the child process.

Memory Allocation : Each process in Unix is given a unique virtual address space, which is mapped to physical memory by the operating system. This separation ensures that processes do not interfere with each other's memory, providing stability and security. When a process is created, Unix allocates memory for it by reserving a range of addresses in the process's address space.

Memory Deallocation : When a process terminates, Unix reclaims the memory it occupied. The memory descriptor (`mm_struct`) associated with the process is updated to reflect the deallocation, making the memory available for other processes. This prevents memory leaks and ensures efficient use of system resources.

Memory Sharing: Despite the isolation, Unix allows processes to share specific memory segments when necessary. This is done through controlled mechanisms like memory-mapped files or shared memory segments, which enable processes to communicate and share data securely without compromising the integrity of each process's address space.

The manipulation of the process address space is facilitated by system calls like `**sbrk**` for allocating more data space and by automatic expansion of the stack segment when needed. Additionally, the kernel may dynamically modify a process's address space by adding or removing intervals of linear addresses, which is essential for managing shared libraries, memory-mapped files, and other shared resources.

C. Question - Demonstrate the steps involved in saving the context of a process in a Unix operating system .

Answer –

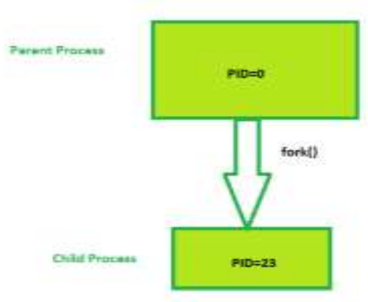
Saving the context of a process in a Unix operating system involves preserving various aspects of the process's state so that it can be restored later. Here are the steps involved:

1. **Save the user-level context**: This step focuses on preserving the user-level context of the process. This includes critical information such as the program counter (PC), processor status register (PS), and stack pointer. Additionally, it encompasses saving data related to the process's text, data, user stack, and any shared memory it might be using. This information is typically stored in the user structure, which can be swapped or paged out when the process is not actively running in memory.
2. **Save the system-level context**: Here, the focus shifts to preserving the system-level context of the process. This includes saving the kernel stack and the kernel mode program counter. These pieces of information are crucial for the kernel to understand the state of the process when it is executing in kernel mode. The system-level context is stored in the process table, which is a data structure resident in memory at all times. The process table contains essential information for all processes, even those that are not currently present in memory.
3. **Update the process table**: When a process is forked, the kernel creates a new entry in the process table for the child process. The parent process's entry in the process table is duplicated into the child's entry, ensuring that the child inherits necessary information from the parent. This includes copying the user structure, which holds the user-level context, along with the stack.
4. **Allocate memory for the child process**: Once the child process's entry is created in the process table, the kernel allocates memory for the child's data and stack segments. This involves making exact copies of the parent's data and stack segments for the child process to use. Depending on whether the text segment is shared or copied, the kernel either shares the text segment (if it's read-only) or duplicates it for the child.
5. **Create a new process**: With all necessary context saved and memory allocated, the child process is now ready to run. When a user initiates a command, such as running a program from the shell, the shell creates a new process by forking a clone of itself. The new process then uses the **exec** system call to replace its memory contents with the executable file's contents, effectively launching the desired program.

D. Question - Analyze the significance of the fork system call in operating systems. Elaborate on the intricate sequence of operations carried out by the kernel when initiating the fork system call.

Answer –

The **fork()** system call in operating systems is a fundamental operation that plays a significant role in process management. It allows a process to create a copy of itself, known as a child process, which can then execute independently of the parent process. This system call is essential for various aspects of operating systems, enabling features like parallel processing, multitasking, and the creation of complex process hierarchies.



Let's dive into the intricate sequence of operations carried out by the kernel when initiating the **fork()** system call:

1. **Allocating a Slot in the Process Table**: When a process calls **fork()**, the kernel first allocates a slot in the process table for the new child process. This table keeps track of all active processes in the system and assigns each process a unique identifier (PID).

2. **Logical Copy of the Parent's Context**: The kernel then makes a logical copy of the parent process's context. This includes duplicating critical aspects of the parent process, such as the text region, data region, user stack, and other memory segments. Initially, both the parent and child processes share the same physical memory pages, thanks to a technique called copy-on-write.

3. **Incrementing File and Inode Table Counters**: If the parent process has any open files or network connections, the kernel increments the corresponding file and inode table counters for the child process. This ensures that the child process has access to the same resources as the parent process.

4. **Returning Process ID and Status**: After the fork operation is complete, the kernel returns the process ID (PID) of the child process to the parent process. The child process receives a return value of 0, indicating that it is the child process. This allows

both the parent and child processes to determine their respective roles and execute different code paths accordingly.

5. **Execution of Parent and Child Processes**: Following the **fork()** system call, both the parent and child processes continue execution independently. The child process starts executing from the point where the **fork()** call was made, while the parent process resumes execution immediately after the **fork()** call. They each have their own execution environments, including separate memory spaces and stacks.

6. **Handling Communication and Coordination**: If the parent and child processes need to communicate or coordinate with each other, they can do so using interprocess communication mechanisms such as pipes, shared memory, or signals. These mechanisms allow data exchange and synchronization between processes, enabling collaboration and coordination in complex systems.

E. Question - Describe the process of system boot and explain the role of the Init process in initializing an operating system

Answer –

The system boot process is a crucial sequence of events that initializes an operating system and prepares it for user interaction. It involves several stages, starting from the execution of firmware instructions stored in the computer's nonvolatile memory to the initialization of essential system processes.

1. **Firmware Execution**: The boot process begins with the execution of firmware instructions stored in the computer's nonvolatile memory, such as BIOS or ROM. These instructions are automatically executed when the power is turned on or the system is reset.
2. **Boot Program Execution**: The firmware instructions locate and execute the system's boot program, typically stored in a standard location on a bootable device, such as block 0 of the root disk or a special partition. The boot program is responsible for loading the Unix kernel into memory and passing control of the system to it.
3. **Kernel Initialization**: Once the kernel is loaded into memory, it initializes various system components, including device drivers, file systems, and process management. At this stage, the kernel sets up essential data structures and prepares the system for user interaction.
4. **Init Process Start**: The kernel starts the Init process, also known as initialization. Init is the first user-space process started during the booting process. It continues running until the system is shut down and is responsible for managing other processes.

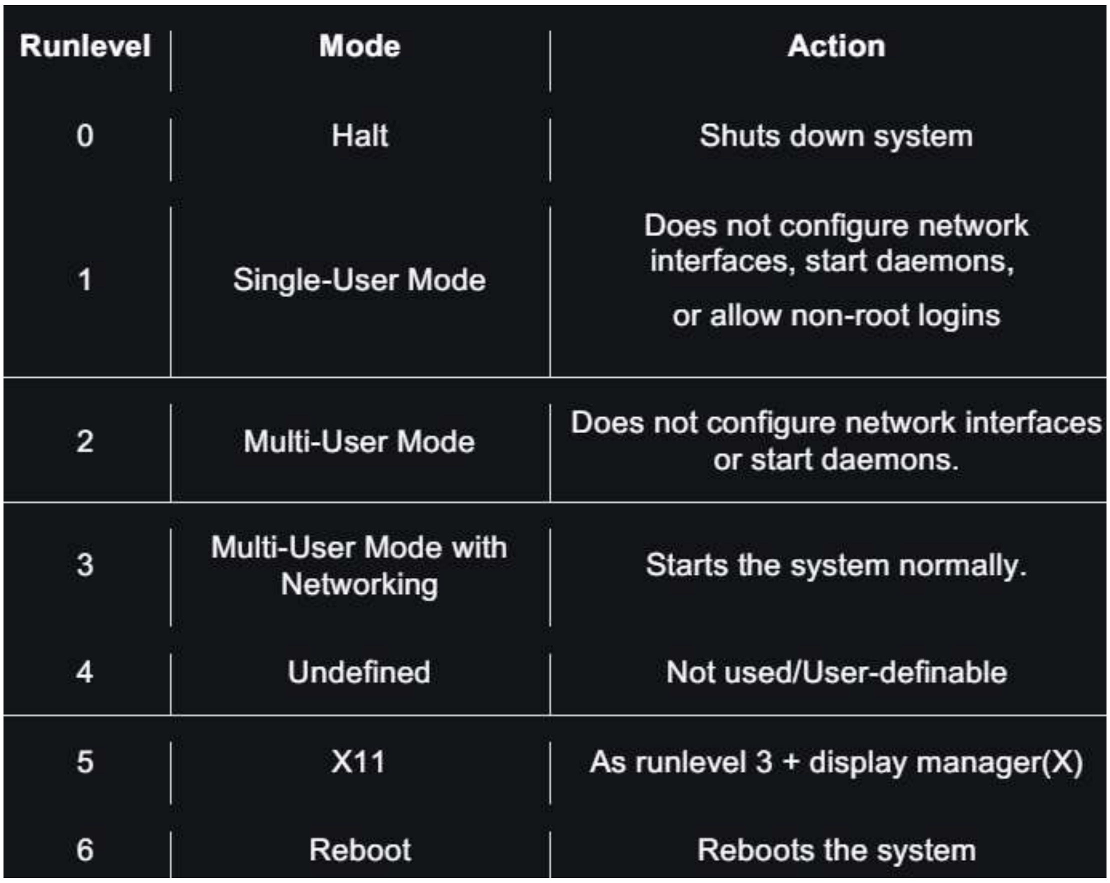
The **init** process plays a critical role in the initialization of an operating system. Here's a closer look at its responsibilities:

- **Parent of All Processes**: Init is the parent of all processes executed by the kernel during the booting of a system. Its primary role is to create processes from a script stored in the file **/etc/inittab**.
- **Daemon Process**: Init is a daemon process that continues running until the system is shut down. It is the direct or indirect ancestor of all other processes and automatically adopts all orphaned processes.
- **Kernel Panic Handling**: If the kernel is unable to start the init process, or if it should die for any reason, a kernel panic will occur. Init is typically assigned process identifier 1, making it a crucial process for system stability.

- **Setting Runlevel**: The init process also sets the runlevel, which defines the system's mode of operation (e.g., single-user mode, multi-user mode, or graphical mode). This helps determine which services and processes should be started during boot-up.

Runlevels:

- The runlevel determines which services are started during boot.
- Common runlevels include:
- The init process transitions the system to the appropriate runlevel.



Runlevel	Mode	Action
0	Halt	Shuts down system
1	Single-User Mode	Does not configure network interfaces, start daemons, or allow non-root logins
2	Multi-User Mode	Does not configure network interfaces or start daemons.
3	Multi-User Mode with Networking	Starts the system normally.
4	Undefined	Not used/User-definable
5	X11	As runlevel 3 + display manager(X)
6	Reboot	Reboots the system

F. Question - Explain the swapping of a process between swap space and main memory?

Answer –

Swapping a process between swap space and main memory is a crucial technique employed by operating systems to efficiently manage memory resources. Here's a detailed explanation of the swapping process:

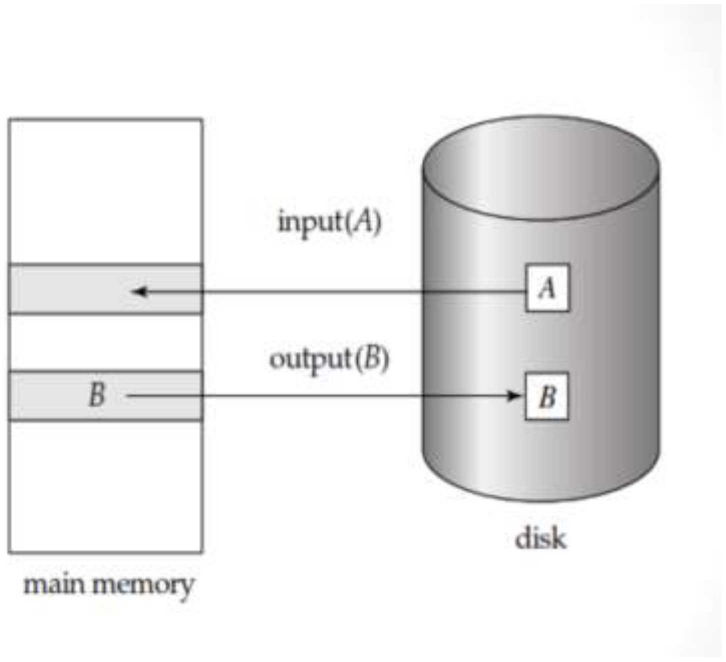
1. **Swapping Out**: When a process is selected for swapping, its entire contents, including code, data, and stack, are copied from the main memory (RAM) to a reserved area on the hard disk known as swap space. This frees up valuable memory in RAM for other processes that need to be executed. Swapping out occurs when the system detects a shortage of available memory or to prevent memory congestion.

2. **Swapping In**: If the swapped-out process needs to be executed again, it is brought back from the swap space into the main memory. This operation is known as swapping in. The operating system retrieves the necessary data and instructions from the swap space and loads them into the appropriate memory locations in RAM. Swapping in occurs when the system determines that the swapped-out process is needed for execution.

3. **Page Replacement**: The operating system employs a page replacement algorithm to decide which pages of a process to swap out when memory becomes scarce. These algorithms determine which pages are least likely to be used in the near future and select them for swapping out. Common page replacement algorithms include Least Recently Used (LRU), First-In-First-Out (FIFO), and Optimal Page Replacement.

4. **Performance Impact**: Swapping can have a significant impact on system performance because accessing data from the hard disk is much slower compared to accessing it from RAM. Excessive swapping, known as thrashing, occurs when the system spends more time swapping processes in and out of memory than executing actual tasks, leading to a severe degradation in performance.

5. **Optimizations**: To mitigate the performance impact of swapping, modern operating systems employ various optimizations. One such optimization is demand paging, where only the necessary parts of a process are loaded into memory when they are needed, rather than loading the entire process into memory at once. Additionally, pre-fetching techniques are used to anticipate which pages will be needed next and load them into memory in advance, reducing the latency associated with swapping.


input(A)
A
output(B)
B
B
main memory
disk

G. Question - Write short note - Demand paging

Answer –

What is Demand Paging?

Demand paging can be described as a memory management technique that is used in operating systems to improve memory usage and system performance. Demand paging is a technique used in virtual memory systems where pages enter main memory only when requested or needed by the CPU.

In demand paging, the operating system loads only the necessary pages of a program into memory at runtime, instead of loading the entire program into memory at the start. A page fault occurred when the program needed to access a page that is not currently in memory. The operating system then loads the required pages from the disk into memory and updates the page tables accordingly. This process is transparent to the running program and it continues to run as if the page had always been in memory.

What is Page Fault?

The term "page miss" or "page fault" refers to a situation where a referenced page is not found in the main memory.

When a program tries to access a page, or fixed-size block of memory, that isn't currently loaded in physical memory (RAM), an exception known as a page fault happens. Before enabling the program to access a page that is required, the operating system must bring it into memory from secondary storage (such a hard drive) in order to handle a page fault.

In modern operating systems, page faults are a common component of virtual memory management. By enabling programs to operate with more data than can fit in physical memory at once, they enable the efficient use of physical memory. The operating system is responsible for coordinating the transfer of data between physical memory and secondary storage as needed.

What is Thrashing?

Thrashing is the term used to describe a state in which excessive paging activity takes place in computer systems, especially in operating systems that use virtual memory, severely impairing system performance. Thrashing occurs when a system's high memory demand and low physical memory capacity cause it to spend a large amount of time rotating pages between main memory (RAM) and secondary storage, which is typically a hard disc.

It is caused due to insufficient physical memory, overloading and poor memory management. The operating system may use a variety of techniques to lessen thrashing, including lowering the number of running processes, adjusting paging parameters, and improving memory allocation algorithms. Increasing the system's physical memory (RAM) capacity can also lessen thrashing by lowering the frequency of page swaps between RAM and the disc.

Pure Demand Paging

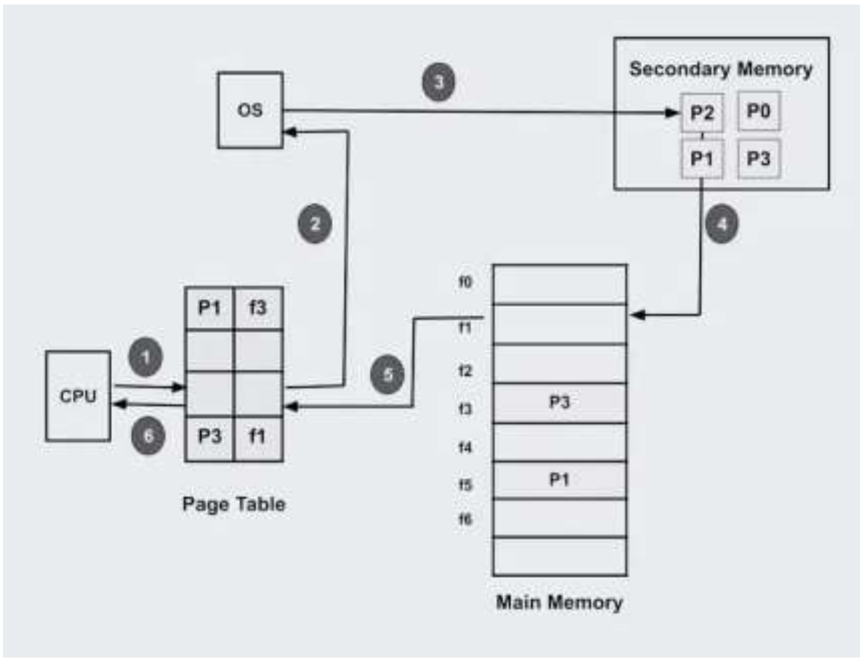
Pure demand paging is a specific implementation of demand paging. The operating system only loads pages into memory when the program needs them. In on-demand paging only, no pages are initially loaded into memory when the program starts, and all pages are initially marked as being on disk.

Operating systems that use pure demand paging as a memory management strategy do so without preloading any pages into physical memory prior to the commencement of a task. Demand paging loads a process's whole address space into memory one step at a time, bringing just the parts of the process that are actively being used into memory from disc as needed.

It is useful for executing huge programs that might not fit totally in memory or for computers with limited physical memory. If the program accesses a lot of pages that are not in memory right now, it could also result in a rise in page faults and possible performance overhead. Operating systems frequently use caching techniques and improve page replacement algorithms to lessen the negative effects of page faults on system performance as a whole.

Working Process of Demand Paging

So, let us understand this with the help of an example. Suppose we want to run a process P which have four pages P0, P1, P2, and P3. Currently, in the page table, we have pages P1 and P3.



Therefore, the operating system's demand paging mechanism follows a few steps in its operation.

- **Program Execution:** Upon launching a program, the operating system allocates a certain amount of memory to the program and establishes a process for it.
- **Creating page tables:** To keep track of which program pages are currently in memory and which are on disk, the operating system makes page tables for each process.
- **Handling Page Fault:** When a program tries to access a page that isn't in memory at the moment, a page fault happens. In order to determine whether the necessary page is on disk, the operating system pauses the application and consults the page tables.
- **Page Fetch:** The operating system loads the necessary page into memory by retrieving it from the disk if it is there.
- The page's new location in memory is then reflected in the page table.
- **Resuming the program:** The operating system picks up where it left off when the necessary pages are loaded into memory.
- **Page replacement:** If there is not enough free memory to hold all the pages a program needs, the operating system may need to replace one or more pages currently in memory with pages currently in memory. on the disk. The page replacement algorithm used by the operating system determines which pages are selected for replacement.
- **Page cleanup:** When a process terminates, the operating system frees the memory allocated to the process and cleans up the corresponding entries in the page tables.

Advantages of Demand Paging

So in the Demand Paging technique, there are some benefits that provide efficiency of the operating system.

- **Efficient use of physical memory**: Query paging allows for more efficient use because only the necessary pages are loaded into memory at any given time.
- **Support for larger programs:** Programs can be larger than the physical memory available on the system because only the necessary pages will be loaded into memory.
- **Faster program start:** Because only part of a program is initially loaded into memory, programs can start faster than if the entire program were loaded at once.
- **Reduce memory usage:** Query paging can help reduce the amount of memory a program needs, which can improve system performance by reducing the amount of disk I/O required.

Disadvantages of Demand Paging

- **Page Fault Overload:** The process of swapping pages between memory and disk can cause a performance overhead, especially if the program frequently accesses pages that are not currently in memory.
- **Degraded performance:** If a program frequently accesses pages that are not currently in memory, the system spends a lot of time swapping out pages, which degrades performance.
- **Fragmentation:** Query paging can cause physical memory fragmentation, degrading system performance over time.
- **Complexity:** Implementing query paging in an operating system can be complex, requiring complex algorithms and data structures to manage page tables and swap space.

H. Question - Write short note - Demand paging

Answer –

In Unix, signals are a mechanism for informing processes of the occurrence of synchronous events. Signals can be sent by the kernel or by other processes, and they can indicate a variety of events, such as termination of a process, process-induced exceptions, unrecoverable conditions during a system call, unexpected error conditions during a system call, errors in user mode, and terminal interaction .

Here's a breakdown of the types of signals in Unix:

1. **Termination Signals**: These signals indicate the termination of a process. Examples include SIGTERM (termination signal) and SIGKILL (kill signal).
2. **Exception Signals**: Signals related to process-induced exceptions, like SIGFPE (floating point exception) and SIGSEGV (segmentation fault).
3. **System Call Signals**: Signals indicating unrecoverable conditions during a system call, such as SIGPIPE (broken pipe) and SIGIO (I/O error).
4. **Error Signals**: Signals caused by unexpected error conditions during a system call, like SIGEMT (emulation trap) and SIGSYS (bad system call).
5. **User Mode Signals**: Signals originating from a process in user mode, such as SIGINT (interrupt) and SIGQUIT (quit).
6. **Terminal Interaction Signals**: Signals related to terminal interaction, like SIGTSTP (terminal stop) and SIGTTIN (background terminal read).
7. **Tracing Signals**: Signals for tracing execution of a process, such as SIGTRAP (trace trap) and SIGSTEP (step).

Processes can handle signals in three ways: ignoring the signal, catching the signal and executing a specified function, or terminating the process. The signal system call, with syntax **signal(signum, function)**, is used to define the action to be taken when a specific signal is caught.

Signals can be sent by the kernel, other processes, or by the process itself using the **kill** system call. They play a crucial role in inter-process communication and controlling process behavior in Unix and Unix-like operating systems. Understanding signals and their handling mechanisms is essential for effective process management and system stability.

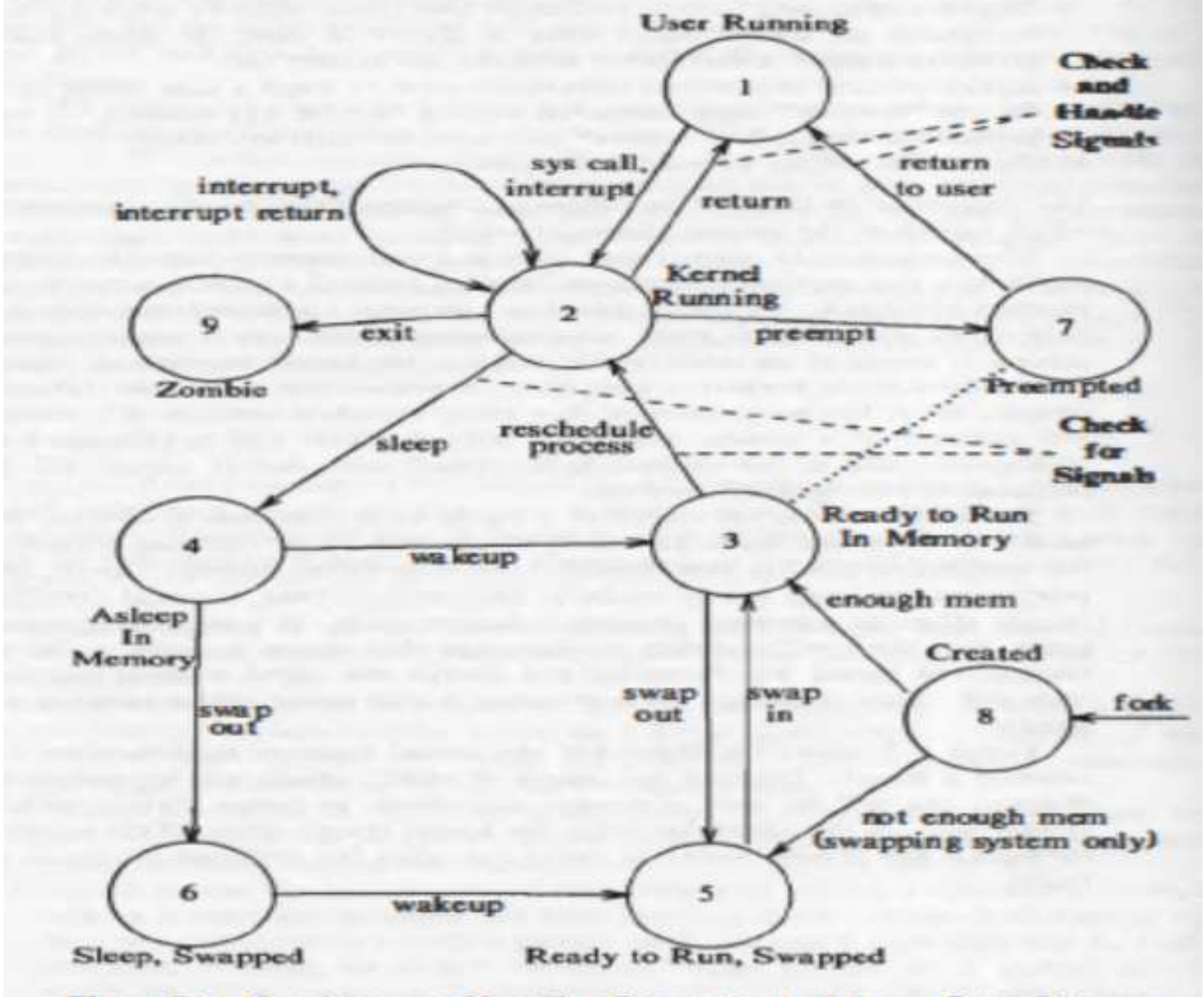


Figure 7.6. Checking and Handling Signals in the Process State Diagram