1. With a neat diagram, explain the concept of the virtual machine.

Virtual Machine: Concept and Example

Definition:
A virtual machine (VM) abstracts the hardware of a computer (CPU, memory, disk, etc.) into multiple independent execution environments, making each environment appear like a private computer to the user.

Key Points:

- Each process has the illusion of its own processor and memory.
- The **host OS** is the main operating system installed on the system, while the additional operating systems installed are called **guest OS**.
- VMs were first implemented in the **VM Operating System** for IBM mainframes in 1972.

Implementation:

- Virtual machine software runs in **kernel mode** (like an OS), while the virtual machine itself executes in **user mode**.
- Implementation is complex as it requires duplicating the underlying hardware environment.

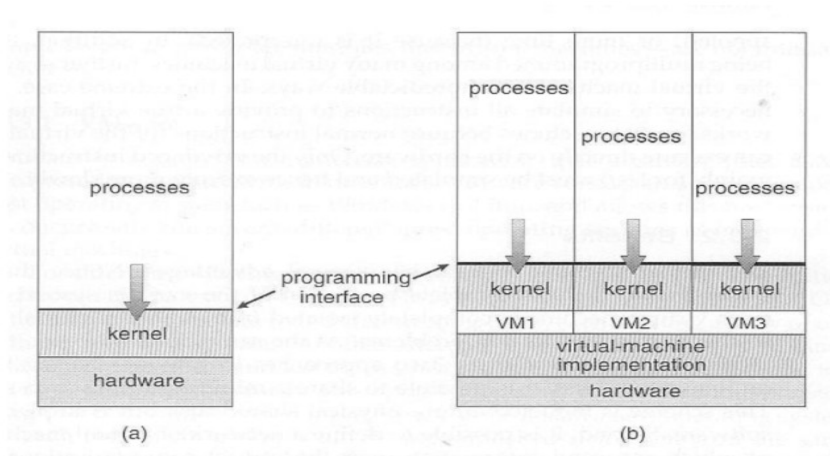


Figure: System modes. (A) Non-virtual machine (b) Virtual machine

Benefits of Virtual Machines:

1. **Resource Sharing:**
   - Multiple operating systems can share the same hardware.
2. **Isolation:**
   - Guest OS are isolated from one another and the host OS. For example, a virus in one guest OS cannot affect others or the host system.
3. **Software Sharing:**
   - Resources can be shared via shared file systems or virtual communication networks.
4. **Development and Testing:**
   - Developers can run multiple OS environments simultaneously, allowing rapid code testing and system development.
5. **System Consolidation:**

- Multiple systems can be combined into one physical system, reducing hardware requirements.

Example:
Running **Windows** as the host OS and installing **Ubuntu** and **macOS** as guest OS in a virtualization software like **VMware** or **VirtualBox**. Each OS can run independently, sharing the same physical resources like CPU and memory.

---

2. What are system calls? Briefly point out its types with illustrations.

System Calls and Their Types

Definition:
System calls provide an interface between a program and the operating system, enabling the program to request services from the OS. They are often written in C or C++ and sometimes in assembly for performance optimization.

How System Calls Work:

- System calls are used in tasks like copying data from one file to another.
  - The program may use system calls to open files, read from the input file, write to the output file, handle errors (e.g., missing files), and terminate the program.
- Most programmers use APIs to access system calls indirectly, ensuring portability across systems.
- System call numbers in a **system call table** determine the appropriate service routine to execute.

Parameter Passing Methods:

1. **In registers**
2. **Passing the address of a parameter block** (used in Linux & Solaris)
3. **Using the stack** (parameters pushed by the program and popped by the OS).

Types of System Calls:

1. **Process Control**
   - Examples: Create process, terminate process, load/execute program.
2. **File Management**
   - Examples: Open, read, write, close files.
3. **Device Management**
   - Examples: Request/release devices, read/write to devices.
4. **Information Management**
   - Examples: Get/set system data, retrieve file attributes.
5. **Communications**
   - Examples: Send/receive messages, establish/terminate connections.
6. **Protection**
   - Examples: Control access permissions, manage user credentials.

3. Explain the states of a process with a transition diagram and process control block.

Process States and Process Control Block (PCB)

Process States:

A process can exist in the following five states:

1. **New:**
   - The process is being created.
2. **Ready:**
   - The process has all required resources and is waiting to be assigned to the CPU.
3. **Running:**
   - The process is currently executing its instructions.
4. **Waiting:**
   - The process is waiting for an event (e.g., I/O operations, user input) to occur.
5. **Terminated:**
   - The process has completed execution and is removed from the system.

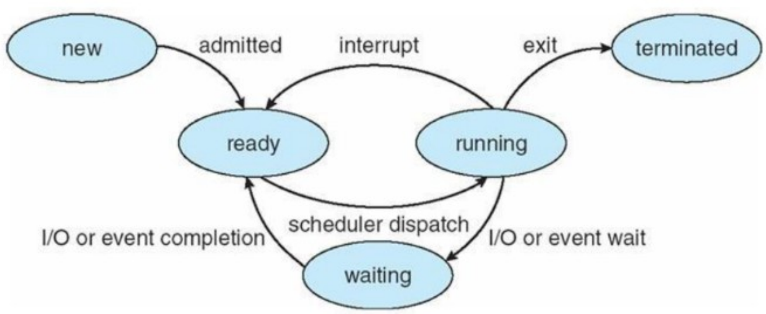


Figure: Diagram of process state

Process Control Block (PCB):

The **PCB** is a data structure that stores all information related to a specific process. It contains:

1. **Process State:**
   - The current state (new, ready, running, waiting, or terminated).
2. **Program Counter:**
   - Address of the next instruction to be executed by the process.
3. **CPU Registers:**
   - Information like accumulators, stack pointers, and index registers, saved for process resumption.
4. **CPU Scheduling Information:**

- Includes process priority, scheduling parameters, and pointers to scheduling queues.

5. **Memory Management Information:**
   - Includes base and limit registers, page tables, or segment tables.
6. **Accounting Information:**
   - Tracks CPU usage, time limits, process ID, and other details.
7. **I/O Status Information:**
   - List of I/O devices and open files associated with the process.

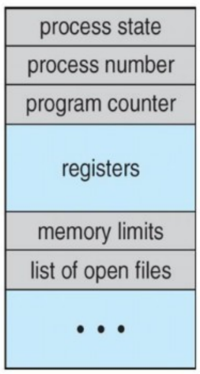


Figure: Process control block (PCB)

---

4. Calculate average waiting and turnaround times by drawing the Gantt chart using FCFS and RR(q=2ms).

Processes	Arrival Time	Burst Time
P1	0	9
P2	1	4
P3	2	9
P4	3	5

---

5. Explain five different scheduling criteria used in the computing scheduling mechanism.

Five Scheduling Criteria in CPU Scheduling

1. **CPU Utilization:**
   - Measures how effectively the CPU is being used.
   - Goal: Keep CPU busy as much as possible (ranges from 40% in light systems to 90% in heavy systems).
2. **Throughput:**
   - Number of processes completed per time unit.

- Example: Long processes may complete 1 process/hour, while short processes may complete 10 processes/second.

3. **Turnaround Time:**
    - Total time taken from process submission to completion.
    - Includes waiting time in the ready queue, execution time, and I/O time.
    - Lower turnaround time is preferred.
4. **Waiting Time:**
    - Time a process spends in the ready queue waiting for CPU allocation.
    - Shorter waiting times improve overall system performance.
5. **Response Time:**
    - Time from the submission of a request until the first response is produced.
    - Important in interactive systems where quick feedback is required.
    - Does not measure full output time, just the time to start responding.

Summary:

These criteria are used to evaluate and compare CPU scheduling algorithms to select the most efficient one for a given system.

---

6. Define deadlock. What are the necessary conditions for deadlock to occur?

Deadlock and Necessary Conditions

Definition of Deadlock:
Deadlock is a situation where processes are stuck in a waiting state indefinitely because the resources they need are held by other waiting processes.

Necessary Conditions for Deadlock:
Deadlock can occur only if all the following four conditions hold simultaneously:

1. **Mutual Exclusion:**
    - At least one resource is non-sharable, meaning only one process can use it at a time.
    - If another process requests the resource, it must wait until the resource is released.
2. **Hold and Wait:**
    - A process is holding at least one resource and waiting for additional resources that are held by other processes.
3. **No Preemption:**
    - Resources cannot be forcibly taken from a process; they can only be released voluntarily after the process completes its task.
4. **Circular Wait:**
    - A circular chain of processes exists where each process is waiting for a resource held by the next process in the chain.

- For example, P_0 is waiting for P_1's resource, P_1 is waiting for P_2's resource, and P_n is waiting for P_0's resource.

Summary:

Deadlock occurs when these four conditions are present simultaneously. To prevent or resolve deadlocks, one or more of these conditions must be eliminated.

---

7. Illustrate Peterson's solution for the critical section problem.

Peterson's Solution for the Critical Section Problem

Definition:
Peterson's solution is a software-based approach to ensure mutual exclusion, progress, and bounded waiting for two processes that alternate between their **critical sections** and **remainder sections**.

Key Points:

- Works for **two processes only** (labeled P0P_0 and P1P_1).
- Requires two shared variables:

1. **int turn:**
   - Indicates whose turn it is to enter the critical section.
   - Example: If turn == i, process PiP_i is allowed to enter.
2. **boolean flag[2]:**
   - Indicates whether a process is ready to enter its critical section.
   - Example: If flag[i] == true, process PiP_i is ready.

How It Works:

1. **Entering the Critical Section:**
   - Process PiP_i:
     - Sets flag[i] = true (shows it wants to enter).
     - Sets turn = j (allows PjP_j to enter if it also wants).
     - Waits until flag[j] == false or its own turn (turn == i).
2. **Handling Simultaneous Requests:**
   - If both processes try to enter at the same time, the turn variable determines which process enters the critical section first.
   - The process whose turn is set last will wait.
3. **Exiting the Critical Section:**
   - After finishing, PiP_i sets flag[i] = false to allow PjP_j to enter.

```
do {
        flag[i] = TRUE;
        turn = j;
        while (flag[j] && turn == j)
                ; // do nothing
         critical section
        flag[i] = FALSE;

                remainder section

    } while (TRUE);
```

Figure: The structure of process P_i in Peterson's solution

Summary of Benefits:

- Ensures **mutual exclusion** (only one process in the critical section at a time).
- Maintains **progress** (no process is unnecessarily delayed).
- Satisfies **bounded waiting** (no process waits forever to enter).

---

8. Explain different methods to recover from deadlocks.

Methods to Recover from Deadlocks

Deadlocks can be managed in the following ways:

1. **Prevent or Avoid Deadlocks:**
    - The system uses strategies to ensure that deadlocks never occur.
    - Two approaches:
        - **Deadlock Prevention:** Prevents at least one of the four necessary conditions (Mutual Exclusion, Hold and Wait, No Preemption, Circular Wait) from happening.
        - **Deadlock Avoidance:** The system is given additional information about resource usage in advance and makes decisions to avoid entering a deadlock state.
2. **Detect and Recover from Deadlocks:**
    - If a deadlock occurs, the system identifies it using algorithms and then takes steps to recover.
    - Recovery can involve:
        - Terminating one or more processes to break the deadlock.
        - Preempting resources from some processes and allocating them to others.
3. **Ignore Deadlocks:**
    - The system assumes that deadlocks are rare and does nothing to prevent, detect, or recover from them.
    - This approach is used by systems like UNIX, where manual intervention (e.g., restarting the system) is required if a deadlock occurs.

9. What is a semaphore? State a Dining Philosopher problem gives a solution using semaphore.

10. What is demand paging? Explain the steps in handling page faults using the appropriate diagram.

Demand Paging

Demand paging is a memory management technique where a page is loaded into memory **only when it is needed** during program execution.

Key Features of Demand Paging:

- Unlike swapping entire processes, **only required pages** are swapped into memory.
- **Less I/O and memory usage:** Reduces swap time and physical memory usage.
- **Faster response time:** Unused pages are not loaded.
- **More users can be supported** as memory is used efficiently.
- Operates using a **pager** that swaps pages on demand.

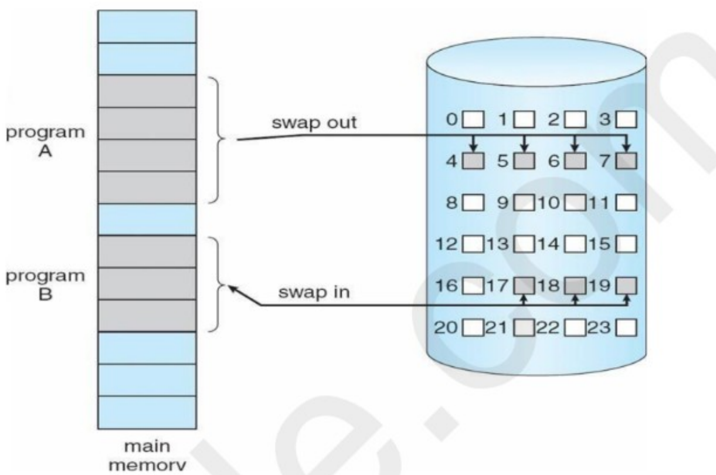


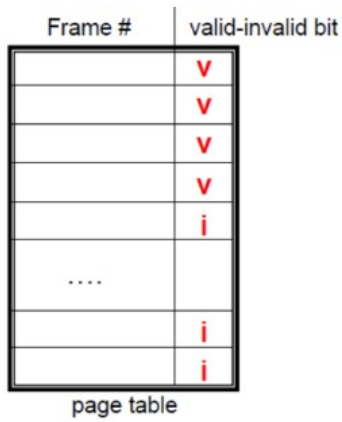
Fig: Transfer of a paged memory into continuous disk space

Handling Page Faults

A **page fault** occurs when a process tries to access a page that is not in memory. The steps to handle a page fault are:

1. **Access the Page Table:**
    - The **valid-invalid bit** in the page table is checked.
    - If the bit is valid (v), the page is in memory.
    - If the bit is invalid (i), the page is either on the disk or invalid, causing a page fault.
2. **Trigger a Page Fault Trap:**
    - The operating system (OS) detects the page fault.
3. **Determine the Page Status:**
    - Check if the page is valid but not in memory.
    - If invalid, terminate the process.

4. **Load the Page:**
    - Locate the page on the disk.
    - Bring the page into memory.
5. **Update the Page Table:**
    - Mark the page as valid (v) in the page table.
6. **Resume Execution:**
    - Restart the process from where the page fault occurred.



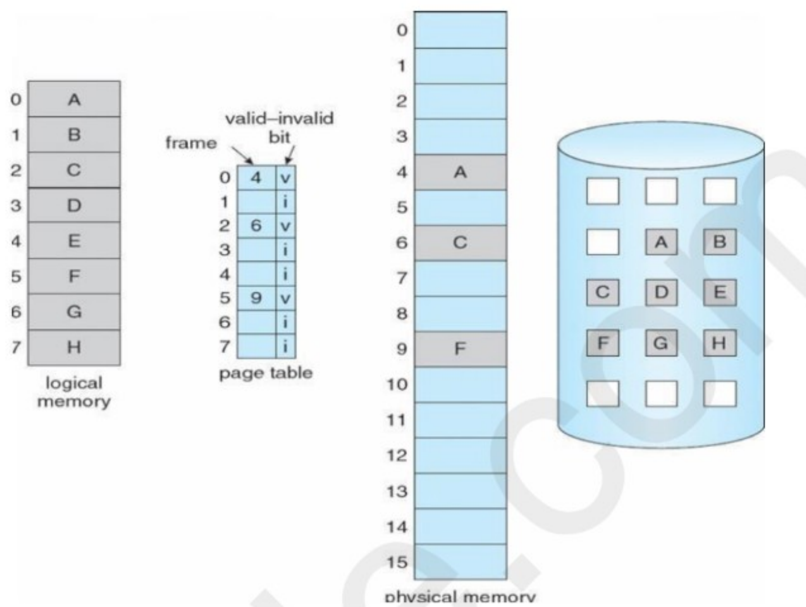


Fig: Page Table when some pages are not in main memory

Advantages of Demand Paging:

- Optimizes memory usage by loading only the required pages.
- Reduces unnecessary I/O operations.
- Supports multitasking by freeing up memory for multiple processes.

11. Discuss the structure of the page table with a suitable diagram.

Structure of the Page Table

A **page table** maps the virtual addresses used by a program to the physical addresses in memory. There are several ways to organize the page table to improve performance and manage large address spaces.

1. Hierarchical Paging

Hierarchical paging is used to handle large address spaces (32-bit or more) where the page table itself can become too large. This method divides the page table into smaller parts to reduce its size.

Two-Level Paging Example:

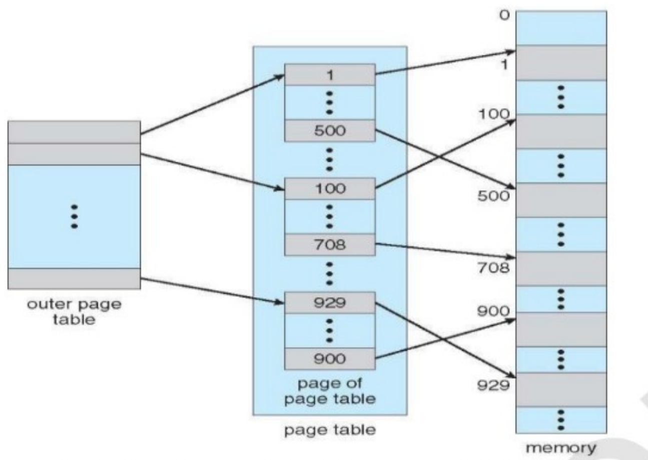


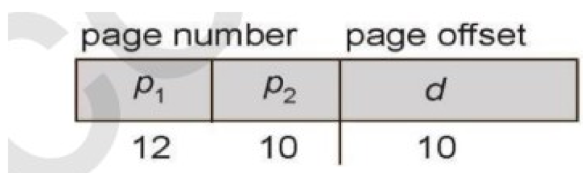
Figure: A two-level page-table scheme

Consider a system with a 32-bit logical address space and a 4 KB page size. The logical address is split into:

- **20 bits** for the page number
- **12 bits** for the page offset

Since the page table itself is large, the page number is further divided into two parts:

- **10 bits** for the first-level page table index
- **10 bits** for the second-level page table index



In this scheme, the logical address is divided into:

- p1: Index into the outer page table
- p2: Index into the inner page table

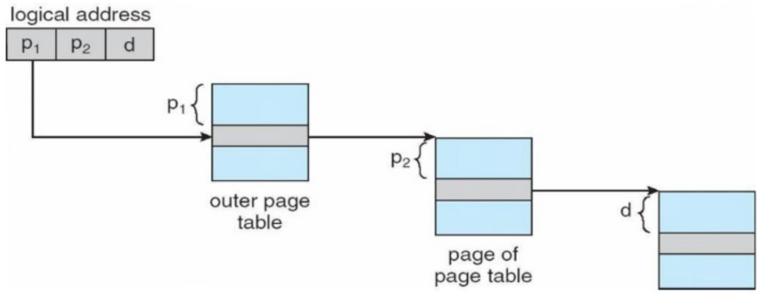


Figure: Address translation for a two-level 32-bit paging architecture

Address translation happens from the outer table inward, making this a **forward-mapped page table**.

2. Hashed Page Tables

Hashed page tables are used for larger address spaces (more than 32 bits). This method uses a hash function to map virtual page numbers to entries in a hash table.

Each hash table entry contains a **linked list** of elements that hash to the same location to handle **collisions**. Each element in the list has:

1. **Virtual page number**
2. **Mapped physical page frame**
3. **Pointer to the next element**

The process works as follows:

- The virtual page number is hashed into the table.
- The system checks the list for a match with the virtual page number.
- If a match is found, the corresponding physical address is used; if not, the system searches further down the list.

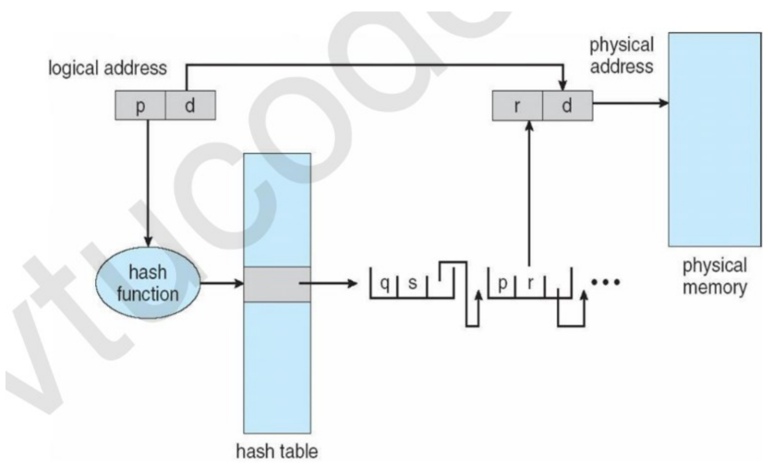


Figure: Hashed page-table

3. Inverted Page Tables

In inverted page tables, there is only one entry for each physical page in memory. Each entry contains:

- The **virtual address** of the page in that memory location
- Information about the **process** that owns the page

Each virtual address is represented as a triplet: <process-id, page-number, offset>. When a memory reference occurs:

1. The virtual address is checked against the inverted page table.
2. If a match is found, the physical address is generated.
3. If no match is found, it indicates an illegal memory access.

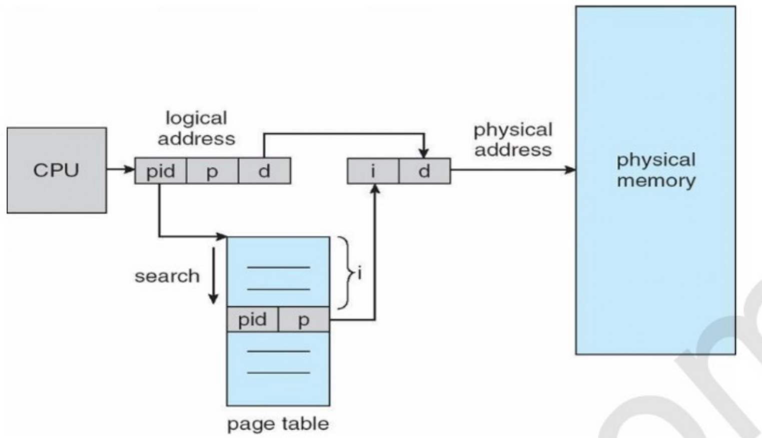


Figure: Inverted page-table

Advantages and Disadvantages:

- **Advantage:** It reduces the memory needed to store the page table.
- **Disadvantages:** Searching the table is slower, and it can be challenging to implement for **shared memory** scenarios.

---

12. Explain in detail about various file operations in a file system.

File Operations in a File System

A file system allows various operations on files. These operations are essential for managing files effectively and include the following:

1. Creating a File

To create a file, two steps are required:

- **Space Allocation:** The system must find space for the new file in the file system.
- **Directory Entry:** An entry for the file must be added to the directory to keep track of the file's location.

2. Writing to a File

To write to a file:

- A system call is made with the file name and data to be written.
- The system searches the directory to find the file's location.
- A **write pointer** keeps track of where in the file the next data should be written.
- The write pointer is updated after each write operation.

3. Reading from a File

To read from a file:

- A system call is made with the file name and a location where the next block should be placed.
- The system searches the directory for the file's entry.
- A **read pointer** keeps track of where the next read will take place, and it is updated after each read.

4. Repositioning within a File

This operation involves moving the file's **current file position** to a new location without performing actual I/O. This is known as **file seek**, allowing you to jump to a specific point in the file.

5. Deleting a File

To delete a file:

- The system searches the directory for the file.
- Once the file is found, the file space is released, and the directory entry is erased.

6. Truncating a File

Truncating a file removes its content but keeps its attributes (like name and permissions) intact. This operation resets the file's length to zero and releases its space for other uses.

Additional Operations:

- **Appending:** Adding new data to the end of an existing file.
- **Renaming:** Changing the name of an existing file.