black N White

BLACK N WHITE

Learn Today Lead Tomorrow

jag...

| | |
|---|---|
| NAME | |
| ROLL NUMBER | |
| SEMESTER | 4th |
| COURSE CODE | DCA2203 |
| COURSE NAME | SYSTEM SOFTWARE |

## SET-I

**Q.1) Describe the function and significance of the segment and pointer registers in the 8086 microprocessor. How do these registers support memory segmentation and data handling?**

### Answer .:-

The 8086 microprocessor uses a system of segment and pointer registers to manage its memory and data handling efficiently. These registers play a crucial role in allowing the processor to access a larger memory space while maintaining speed and organization.

The segment registers in 8086 are four 16-bit registers: Code Segment **(CS)**, Data Segment **(DS)**, Stack Segment **(SS)**, and Extra Segment **(ES)**. Each segment register holds the base address of a memory segment, which is a 64KB block of memory.

The Code Segment **(CS)** register points to the segment containing the currently executing program instructions.

The Data Segment **(DS)** is used to store program variables and data.

The Stack Segment **(SS)** manages the stack memory, which is used for function calls, returns, and temporary data storage.

The Extra Segment **(ES)** acts as an additional data segment that can be used for operations involving memory blocks.

The pointer registers include the Instruction Pointer **(IP)**, Stack Pointer **(SP),** and Base Pointer **(BP)**.

The Instruction Pointer **(IP)** holds the offset address of the next instruction to be executed within the Code Segment.

The Stack Pointer **(SP)** keeps track of the top of the stack in the Stack Segment.

The Base Pointer **(BP)** is mainly used to access parameters and local variables in the stack during subroutine execution.

The significance of these registers lies in their ability to enable memory segmentation, a technique where the total 1MB memory space (20-bit address) is divided into manageable segments. Each memory address is calculated by combining a segment base address (from segment registers) and an offset (from pointer registers).
For example, a physical address is computed as:
Physical Address = (Segment Register × 16) + Offset

This segmentation provides several benefits. It allows programs to be modular by separating code, data, and stack segments. It also makes memory management more flexible and efficient, enabling multiple programs or data blocks to coexist without interference.

In terms of data handling, segment and pointer registers simplify data movement and access. When dealing with large data structures, the Extra Segment (ES) can be used along with the Data Segment (DS) to efficiently move blocks of memory. Stack management becomes easier with the Stack Segment (SS) and Stack Pointer (SP), ensuring safe execution of function calls and returns.

The segment and pointer registers in the 8086 microprocessor work together to effectively manage memory through segmentation and support efficient data handling. Their design allows the processor to maximize the use of available memory, organize programs logically, and ensure smooth execution of complex operations.

## Q.2) Differentiate between the roles of the Symbol Table and the Literal Table in an assembler. Describe how each is constructed during Pass 1 and how they are used during Pass 2.

### Answer .:-

In an assembler, both the Symbol Table and the Literal Table are essential for handling addresses and data during program translation. However, they serve different purposes and are constructed differently across the two passes of the assembler.

**The Symbol Table (SYMTAB)** records the names and addresses of labels used in the program. A label usually marks a particular memory location or instruction, allowing future references to it. During **Pass 1**, whenever the assembler encounters a label, it adds the label name along with the current location counter (LC) value into the Symbol Table. This way, every label is linked to a specific memory address.
In **Pass 2**, when the assembler sees a label in an instruction (for example, in jump operations), it consults the Symbol Table to fetch the correct address to replace the label with the actual numeric memory address in the final machine code.

**The Literal Table (LITTAB)**, on the other hand, stores literal constants — fixed values written directly into the code, often indicated with an '=' symbol (e.g., =5). During **Pass 1**, whenever a literal is detected, it is added to the Literal Table. However, its address is typically assigned later, when a LTORG directive or the end of the program is reached. At that point, the assembler allocates memory addresses for all collected literals and updates the Literal Table accordingly.
During **Pass 2**, when an instruction needs to load a literal into a register or memory, the assembler uses the Literal Table to find the assigned address of the literal and places it into the machine code.

**Main Differences**:

- **Symbol Table** maps labels (symbols) to addresses; **Literal Table** maps constant values to addresses.
- Symbols are created when defining parts of the program; literals are constants needing storage.
- In **Pass 1**, Symbol Table entries are added immediately on encountering labels, while literals are collected and assigned addresses when necessary.
- In **Pass 2**, the Symbol Table is used to resolve symbolic references, and the Literal Table is used to insert literal values at the correct locations.

The Symbol Table supports the identification and linking of program labels, whereas the Literal Table manages constant data within the program. Both ensure that the assembler accurately converts symbolic source code into machine-executable form.

## Q.3) Compare and contrast Absolute Loaders and Relocating Loaders. Why are relocating loaders preferred for advanced systems? Support your answer with examples and scenarios.

### Answer .:-

In the world of system software, loaders play an essential role in placing programs into memory for execution. Two common types of loaders are **Absolute Loaders** and **Relocating Loaders**, each serving distinct purposes based on system complexity.

An **Absolute Loader** loads the compiled or assembled code into a specific, fixed memory location. The program must be built with the exact memory addresses where it will reside. There is no flexibility—if the memory is not free or if multiple programs need to be loaded, conflicts may occur. For example, early simple computers or small embedded systems often used absolute loaders because the programs were tiny and hardware resources were limited.

In contrast, a **Relocating Loader** provides much greater flexibility. It allows programs to be loaded at different memory locations dynamically. When the program is prepared for execution, the relocating loader adjusts address-dependent parts of the code according to the current memory availability. This dynamic relocation enables multiple programs to coexist in memory without interfering with each other. Systems like modern operating systems (Windows, Linux) extensively use relocating loaders to manage multitasking environments where multiple processes are loaded and moved around in memory.

**Key Differences**:

- **Address Flexibility**: Absolute loaders require fixed addresses, while relocating loaders allow dynamic address adjustment.
- **Memory Management**: Absolute loaders have no memory optimization; relocating loaders optimize memory usage efficiently.
- **Error Handling**: Absolute loaders are prone to memory conflicts, while relocating loaders prevent such issues through address relocation.
- **Complexity**: Absolute loaders are simple and fast; relocating loaders are more complex but versatile.

**Why relocating loaders are preferred**:

In advanced systems, flexibility, multitasking, and efficient memory utilization are critical. A relocating loader supports all these needs. Imagine a server running hundreds of applications simultaneously. Assigning fixed memory addresses to each would be impractical. Instead, relocating loaders assign memory at runtime based on availability, preventing crashes and optimizing performance.

For example, in operating systems like Linux, when a user opens multiple applications, each program is loaded into different available memory areas, even though none of them were specifically coded for those exact locations. This is only possible because the relocating loader adjusts their internal memory references dynamically.

**A fresh perspective**:

Looking deeper, relocating loaders contribute significantly to the stability, flexibility, and performance of modern computing environments. Their ability to adapt to dynamic memory conditions makes them a backbone technology for any advanced multitasking or multiprocessing system today.

## SET-II

**Q.4) Compare and contrast device driver management in UNIX/Linux, MS-DOS, and Windows operating systems. Discuss the key differences in installation, configuration, communication, and architecture.**

### Answer .:-

Managing device drivers varies significantly across UNIX/Linux, MS-DOS, and Windows systems, each reflecting the evolution of operating system design over decades. Understanding these differences offers a clear view into how modern systems prioritize stability, flexibility, and user experience.

**UNIX/Linux systems** handle device drivers through a modular kernel approach. Most drivers are developed as separate modules, allowing them to be loaded and unloaded dynamically without rebooting the system. Installation typically involves compiling the driver source code or using package managers. Configuration is file-based, often requiring edits to files like /etc/modules or using commands such as modprobe. Communication between the kernel and device drivers occurs through well-defined interfaces, ensuring hardware abstraction. The architecture is highly modular, enabling flexibility and customization for various hardware platforms.

**MS-DOS**, on the other hand, follows a much simpler and rigid approach. Device drivers in MS-DOS are loaded during the boot process through system files like CONFIG.SYS. Installation usually involves manually placing driver files and editing configuration files. There is minimal dynamic management—drivers must be loaded at startup, and changes require a reboot. Communication is direct and primitive, often bypassing complex abstraction layers. Architecturally, MS-DOS treats device drivers as essential system extensions that remain static throughout the session.

**Windows operating systems** introduce a hybrid model combining dynamic management with strong system integration. Modern Windows versions allow plug-and-play installation of drivers, automatically detecting hardware and installing appropriate drivers with minimal user intervention. Configuration is often handled through graphical user interfaces like the Device Manager. Communication between the operating system and drivers follows a layered model, with frameworks like the Windows Driver Framework (WDF) providing structured interfaces. Architecturally, Windows drivers are tightly coupled with the system's security and stability layers, including digital signature enforcement to prevent malicious drivers from being installed.

**Key differences summarized**:

- **Installation**: Linux relies on package managers or manual compilation; MS-DOS needs manual editing; Windows automates most of the process.
- **Configuration**: UNIX/Linux uses text files and command-line tools; MS-DOS uses startup files; Windows offers GUI-based management.
- **Communication**: Linux uses kernel APIs, MS-DOS uses direct memory addressing, and Windows uses structured frameworks.

- **Architecture**: Linux and Windows favor modular, dynamic systems; MS-DOS maintains a static, manual setup.

**Taking a closer look**, device driver management reflects each system's priorities: simplicity for MS-DOS, flexibility for Linux, and usability combined with security for Windows. The differences highlight how operating systems have adapted to growing hardware complexity and user expectations over time.

**Q.5) Explain the process of IP Address allocation in UPnP devices using both DHCP and Auto-IP mechanisms. Discuss the importance of address management in UPnP and how it ensures seamless device communication on the network.**

## Answer .:-

When setting up Universal Plug and Play (UPnP) devices on a network, effective IP address allocation plays a crucial role in enabling smooth communication between devices. UPnP is designed to promote zero-configuration networking, allowing devices to discover each other automatically without manual setup.

In most cases, UPnP devices first attempt to obtain an IP address using **DHCP (Dynamic Host Configuration Protocol)**. DHCP is a network management protocol that dynamically assigns an IP address and other network configuration parameters to devices, allowing them to communicate on an IP network. When a UPnP device connects, it sends a DHCP request to the network's DHCP server (typically a router). If the DHCP server responds, the device is assigned an IP address, subnet mask, gateway, and DNS information. This method ensures that devices are correctly integrated into the broader network infrastructure and can seamlessly interact with other networked resources.

However, if the DHCP server is unavailable or fails to respond, UPnP devices fall back to an alternative mechanism known as **Auto-IP**. Auto-IP, based on the Zero-Configuration Networking (Zeroconf) concept, allows devices to self-assign an IP address from a predefined range—typically between 169.254.0.1 and 169.254.255.254. Before finalizing an address, the device uses ARP (Address Resolution Protocol) to check if the selected address is already in use by another device. If it is free, the device claims it and announces its presence on the network. This fallback ensures that even without a DHCP server, basic peer-to-peer communication between devices remains possible.

**Address management is vital** in UPnP environments because it guarantees that devices can reliably discover, connect, and communicate with each other without user intervention. Without proper IP assignment, devices might fail to appear in the network, resulting in poor user experience and disrupted service. Whether through DHCP or Auto-IP, consistent address management allows devices such as smart TVs, printers, and gaming consoles to connect effortlessly, supporting activities like media sharing, online gaming, and remote management.

**Looking beyond the basics**, the seamless address allocation processes in UPnP reflect a broader move toward creating intelligent, self-managing networks. As homes and businesses continue to adopt a growing range of IoT devices, reliable address management ensures scalability, minimizes downtime, and simplifies network maintenance—all without overwhelming users with technical complexities.

## Q.6) Explain the hierarchy process in Android memory management. How does Android decide which process to terminate when memory is low?

### Answer .:-

Managing memory efficiently is critical in Android devices, where resources are inherently limited compared to traditional computers. Android's memory management system is designed to prioritize user experience by dynamically balancing system performance and application availability.

Android categorizes running processes into a **hierarchical structure** based on their importance to the user and the system. This hierarchy helps the system make intelligent decisions about which processes to retain and which to terminate when memory becomes scarce.

At the top of the hierarchy are **Foreground Processes**. These are applications that are currently visible to the user, such as an app with which the user is directly interacting, or components like services bound to the visible app. Foreground processes are the most critical and are the last to be killed when memory is low.

Next are **Visible Processes**. These processes are not in the foreground but still affect what the user sees. For instance, an app running an activity that is partially visible (due to a dialog from another app) falls into this category. While they are less critical than foreground processes, they are still given high priority.

Following that are **Service Processes**. These background services perform ongoing operations that the user may care about, such as playing music or downloading a file. Although not immediately visible, their relevance to the user keeps them safe longer compared to other background activities.

**Background Processes** come next. These are apps that are no longer visible or actively interacting with the user but whose data and state are preserved for quick restoration. Android is more willing to terminate these processes if memory is needed because they can be restarted later without major disruption.

At the bottom are **Empty Processes**. These processes hold no active components but are kept in memory to improve app launch speed in case the user returns to them. They are the first candidates for termination when memory pressure rises.

**Decision-making during low memory situations** is handled through a system component called the **Low Memory Killer (LMK)**, and in modern Android versions, it is replaced or supplemented by **OOM (Out-Of-Memory) adjustments**. Each process is assigned an "adjustment value" based on its category. Lower adjustment values indicate higher priority. When memory becomes insufficient, Android starts killing processes with the highest adjustment values first, ensuring that essential user interactions are not interrupted.

**Stepping back**, Android's memory management isn't just about freeing up RAM; it’s a calculated system to ensure that user experiences remain smooth, apps feel responsive, and background operations do not unnecessarily burden the system. This layered and dynamic management strategy enables Android to run efficiently across a vast range of devices with different hardware capabilities.