Task 1: two basic ways of computer architecture, and which one is the best

1. Von Neumann Architecture:

In the von Neumann architecture, named after computer scientist John von Neumann, the memory used for storing both instructions and data is combined into a single memory space. The central processing unit (CPU) fetches instructions from memory, decodes them, executes them, and stores the results back in memory. This architecture is the basis for most general-purpose computers.

Advantages:

- Simplicity and uniformity in instruction fetching and execution.

- Efficient use of memory, as it can be shared for both instructions and data.

Disadvantages:

- Limited parallelism due to shared memory space for instructions and data.

- Bottleneck in instruction fetching when executing large programs.

2. Harvard Architecture:

The Harvard architecture, named after the Harvard Mark I computer, uses separate memory spaces for instructions and data. This allows simultaneous access to both instruction memory and data memory, potentially enabling better parallelism and performance.

Advantages:

- Reduced bottleneck in instruction fetching, allowing faster execution.

- Improved potential for parallelism due to separate memory spaces.

Disadvantages:

- More complex to design and implement due to separate memory spaces.

- Typically used in specialized systems and not as common as the von Neumann architecture.

Which Architecture is Best:

The choice between von Neumann and Harvard architecture depends on the specific use case and design goals. There is no universally "best" architecture, as each has its own strengths and weaknesses.

- Von Neumann: This architecture is widely used in general-purpose computers due to its simplicity and ease of implementation. It's suitable for a wide range of applications and is well-suited for modern computing environments where memory is more accessible.

- Harvard: This architecture is often used in specialized systems where parallelism and performance are critical, such as in embedded systems and digital signal processors. It can provide better performance in certain scenarios but may be more complex to design and implement.

Task 2: Languages support auto garbage collection

1. Java

2. C#

3. Python

4. Ruby

5. JavaScript

6. C++

7. Swift

8. Kotlin

Task 3: What is BIOS? Why do we use it?

BIOS stands for "Basic Input/Output System." It is a firmware program that resides on a computer's motherboard and is responsible for initializing and controlling hardware components during the boot-up process. BIOS provides the fundamental interface between the computer's hardware and the operating system, allowing the OS to communicate with and control hardware devices.

The main functions of BIOS include:

1. Power-On Self Test (POST): When a computer is powered on, BIOS performs a series of tests to check the hardware components such as memory, CPU, storage devices, and peripherals. It ensures that the essential components are functional before loading the operating system.

2. Bootstrap Loader: BIOS locates and loads the initial software required to start the operating system. This initial software is often referred to as the "bootloader."

3. Hardware Initialization: BIOS initializes hardware devices, sets their configuration parameters, and prepares them for operation. This includes setting up system clocks, configuring input/output devices, and determining boot device priorities.

4. CMOS Setup Utility: BIOS provides a user interface known as the CMOS setup utility that allows users to configure hardware settings, such as boot order, system time, and device settings.

5. Interrupt Handling: BIOS handles hardware interrupts and manages the system's interrupt request lines (IRQs), allowing devices to communicate with the CPU.

6. System Services: BIOS provides a set of low-level system services that can be used by applications and the operating system. These services include disk access and keyboard input.

7. Compatibility Support: Some modern BIOS implementations include legacy support for older hardware and software that rely on BIOS services.

Why Do We Use BIOS:

1. Boot Process: BIOS is essential for starting the computer. It initializes hardware, checks for errors, and loads the bootloader, which in turn loads the operating system.

2. Hardware Configuration: BIOS provides a way to configure and customize hardware settings. Users can access the BIOS setup utility to adjust system settings like boot order, system time, and device configurations.

3. Low-Level Services: BIOS offers low-level services that operating systems and applications can use to interact with hardware, such as reading and writing to storage devices and managing hardware interrupts.

4. System Maintenance: BIOS can help diagnose hardware issues during the POST process by reporting errors and identifying faulty components.

5. Legacy Compatibility: While modern systems are transitioning to newer firmware standards like UEFI (Unified Extensible Firmware Interface), many systems still use BIOS for backward compatibility with older hardware and software.

Task 4: Linux Vs Unix with example

Linux:

1. Origin: Linux is an open-source operating system kernel developed by Linus Torvalds in the early 1990s. It is often used in conjunction with various user-space tools and software to create a complete operating system distribution.

2. Variety: There are many different Linux distributions (distros) available, each with its own package management system, default desktop environment, and configurations. Examples include Ubuntu, CentOS, Fedora, and Debian.

3. Licensing: Linux is typically distributed under the GNU General Public License (GPL) and various other open-source licenses.

4. Community-Driven: The development of Linux is driven by a large and diverse community of developers and contributors.

Unix:

1. Origin: Unix is an operating system family developed in the 1960s at AT&T's Bell Labs. It has evolved into various commercial Unix variants and open-source implementations like FreeBSD, OpenBSD, and NetBSD.

2. Variety: Commercial Unix variants include AIX (IBM), HP-UX (Hewlett-Packard), and Solaris (Oracle). These variants have their own unique features and target specific markets.

3. Licensing: Unix has both commercial and open-source variants. Commercial Unix versions often require licenses, while open-source Unix variants use different open-source licenses.

4. Standardization: Unix systems generally adhere to a set of standards defined by organizations like the Single UNIX Specification. This ensures a level of compatibility and consistency across different Unix systems.

Examples:

1. File Path: In Linux, the file path separator is a forward slash (`/`), like `/home/user/documents/file.txt`. In Unix, it is also a forward slash.

2. Shell: Both Linux and Unix systems use shells for command-line interaction. Common shells include Bash (Bourne-Again Shell) and Zsh.

3. File Permissions: In both Linux and Unix, file permissions are managed using the `chmod` command. For example, to make a file executable, you might use `chmod +x filename`.

4. Package Management: Linux distributions use different package managers. For example, Debian-based distributions use `apt`, while Red Hat-based distributions use `yum` or `dnf`. Unix systems often rely on their own package management methods.

5. Networking: Networking commands are similar between Linux and Unix. For instance, the `ping` command to check network connectivity is the same in both.

6. Text Editors: Text editors like `vi` (or `vim`) and `nano` are commonly available on both Linux and Unix systems for editing files from the command line.

7. Process Management: Both Linux and Unix provide commands like `ps` for displaying information about running processes and `kill` for terminating processes.

Task 5: What is fragmentation?

Fragmentation refers to the division of memory or storage into smaller, non-contiguous blocks that can lead to inefficient use of resources and reduced system performance. It occurs in both computer memory (RAM) and storage systems (hard drives, solid-state drives), and it can be classified into two main types: memory fragmentation and disk fragmentation.

Memory Fragmentation:

Memory fragmentation occurs when the available memory is divided into smaller chunks, making it challenging to allocate contiguous blocks of memory for new processes or data. There are two main types of memory fragmentation:

1. Internal Fragmentation: Internal fragmentation occurs when memory blocks allocated to processes are larger than the actual data they hold. This leads to wasted memory space within allocated blocks. It is more common in systems that use fixed-size memory allocation.

2. External Fragmentation: External fragmentation occurs when free memory blocks are scattered throughout the available memory, making it difficult to allocate contiguous memory for new processes. Although there might be enough total free memory, it may not be available in a single, contiguous block.

Disk Fragmentation:

Disk fragmentation occurs in storage systems when files are stored in non-contiguous clusters on a storage device. This happens over time as files are created, modified, and deleted. Disk fragmentation can impact system performance, as the system needs to spend extra time seeking and reading data from different parts of the disk.

Task 6: Compare among all scheduling algorithms [Round robin - Priority - First come first serve]

1. Round Robin (RR) Scheduling:

-Principle: Each process is assigned a fixed time slice (quantum) to execute on the CPU, and after that time slice expires, the process is moved to the back of the queue.

- Advantages:

- Fairness: All processes get an equal share of the CPU time.
- Suitable for time-sharing systems, where multiple users interact with the system concurrently.

- Disadvantages:

- Overhead: Frequent context switches can lead to overhead due to saving and restoring process state.
- Not ideal for long-running tasks, as processes with longer burst times may need to wait for their turn even if the CPU is available.

2. Priority Scheduling:

- Principle: Each process is assigned a priority value, and the CPU is allocated to the process with the highest priority. Processes with equal

priority might follow FCFS within their priority level.

- Advantages:

- High-priority processes can be given preference for critical tasks.
- Suitable for real-time systems where certain tasks require immediate attention.

- Disadvantages:

- "Starvation": Low-priority processes might never get CPU time if higher-priority processes keep arriving.
- Process priority must be carefully managed to avoid resource monopolization.

3. First Come First Serve (FCFS) Scheduling:

- Principle: Processes are executed in the order they arrive in the ready queue. The first process to arrive gets the CPU first.

- Advantages:

- Simple and easy to implement.
- Fairness in terms of order of execution.

- Disadvantages:

- "Convoy Effect": If a long CPU-bound process arrives first, it can cause shorter processes to wait, leading to poor resource utilization.
- Not suitable for interactive systems or real-time tasks, as a long-running process can block others.

Comparison:

- Fairness:

- RR: Fair, as each process gets an equal time slice.
- Priority: Can be unfair if priorities are not managed well.
- FCFS: Fair, but the "convoy effect" can impact fairness.

- Throughput:

- RR: Moderate throughput due to context switch overhead.
- Priority: High-priority processes might dominate, leading to skewed throughput.
- FCFS: Moderate throughput, impacted by process arrival order.

- Response Time:

- RR: Good for interactive tasks due to regular context switches.
- Priority: High-priority tasks have low response times; others might suffer.
- FCFS: Poor for interactive tasks, as short tasks wait behind long ones.

- Starvation:

- RR: Minimal, as all processes get a fair share.
- Priority: Possible if lower-priority processes are constantly preempted.
- FCFS: Minimal if all processes eventually get a turn.

Task 7: Parallel processing Vs Threads

Parallel Processing:

Parallel processing refers to the simultaneous execution of multiple tasks or processes to achieve faster execution and better resource utilization. It involves dividing a larger task into smaller subtasks that can be executed concurrently on multiple processing units (such as CPU cores or even multiple processors). Parallel processing can be achieved through various mechanisms, including multiple processors, multi-core CPUs, or distributed computing.

Threads:

Threads are smaller units of a process that can execute concurrently within the same process context. Threads within a process share the same memory space and resources, which allows for efficient communication and data sharing between threads. Threads enable multitasking within a single process and are commonly used to achieve better responsiveness in applications.

Comparison:

- Parallel processing operates at a higher level of concurrency, involving multiple independent tasks or processes executed simultaneously.

- Threads operate within the context of a single process, allowing for concurrent execution of different tasks within that process.

- Parallel processing is often used for computationally intensive tasks that can be divided into independent units of work.

- Threads are commonly used for tasks that require multitasking and responsiveness within an application, such as GUI responsiveness or server handling multiple client requests.

Task 8: Languages support multithreading

1. Java

2. C#

3. Python

4. C++

5. C

6. Ruby

7. Go

8. Swift

9. Perl

10. Rust

Task 9: Clean Code principles

Clean Code principles are a set of guidelines and best practices that focus on writing code that is easy to read, understand, and maintain.

1. Meaningful Names:

- Choose descriptive and meaningful names for variables, functions, classes, and other code entities.
- Names should convey the purpose and intent of the entity, making the code self-documenting.

2. Functions and Methods:

- Keep functions and methods small and focused on a single task (Single Responsibility Principle).
- Use meaningful names for functions that reflect what they do.
- Avoid long parameter lists; aim for fewer than three to four parameters.
- Functions should have a clear input and output relationship.

3. Comments and Documentation:

- Aim to write code that is self-explanatory, reducing the need for excessive comments.
- When comments are necessary, explain why something is done rather than how.
- Update comments when code changes to keep them accurate.

4. Formatting and Indentation:

- Use consistent and clear formatting to make code visually appealing.
- Indentation should reflect the code's structure and hierarchy.
- Use consistent line lengths and avoid overly long lines.

5. Single Responsibility Principle (SRP):

- Each class or module should have a single, well-defined responsibility.
- Avoid creating classes or functions that do too many different things.

6. Don't Repeat Yourself (DRY):

- Avoid duplicating code; use abstraction and reusability to eliminate redundancy.
- Encapsulate common logic in functions, methods, or classes.

7. Open/Closed Principle (OCP):

- Software entities (classes, modules, etc.) should be open for extension but closed for modification.
- Use inheritance, interfaces, and polymorphism to achieve extensibility.

8. Avoid Magic Numbers and Strings:

- Replace arbitrary numbers and strings with named constants or enums to improve code readability and maintainability.

9. Testing:

- Write unit tests to validate the correctness of your code.
- Keep test cases separate from the code being tested.
- Test edge cases and boundary conditions.

10. Minimal Dependencies:

- Keep dependencies to a minimum to reduce coupling and make the codebase more modular.
- Use dependency injection to manage external dependencies.

11. Error Handling:

- Use meaningful exception names and handle exceptions at the appropriate level of abstraction.
- Avoid returning special error values; prefer throwing exceptions.

12. Consistent Style:

- Follow consistent naming, formatting, and coding conventions throughout the codebase.
- Use a style guide if available for your programming language.