**SundarBan Payment Gateway**

**Program**

**Program Execution**

Executing a computer program involves multiple stages, beginning with writing the code and ending with its execution. The key steps in this process are as follows:

1. **Writing the Code**: Initially, a programmer writes the source code in a high-level language such as C, C++, or Python and saves it with an appropriate file extension (e.g., .c for C programs).
2. **Preprocessing**: The preprocessor processes the source code, handling macros, including necessary header files, and processing directives such as #include and #define.
3. **Compilation**: The compiler converts the preprocessed code into an assembly language, which provides a low-level representation of the program.
4. **Assembly**: The assembly code is transformed into object code, which consists of machine-readable binary instructions. Modern compilers often integrate this step with compilation.
5. **Linking**: The object code is linked with external library functions to create an executable file. This step resolves references and merges necessary dependencies.
6. **Loading**: The operating system loads the executable file into memory and initializes the program counter, setting up the runtime environment before execution begins.

Below is an illustrative diagram (Figure 3.1) to help in understanding this process.

**Executable File**

An executable file is a type of file that contains compiled code ready to be executed by the operating system. Upon launching an executable file, the OS loads it into memory and initiates execution. Common executable formats include .exe for Windows and ELF (Executable and Linkable Format) or Mach-O for Linux and macOS, respectively.

**Main Components of an Executable File**

1. **Header (Metadata)**: This section contains essential metadata about the executable file, such as its format, entry point, memory layout, and dependencies. This information guides the operating system in correctly loading and executing the program.
2. **Program Counter (PC)**: A CPU register that holds the address of the next instruction to be executed. During execution, the PC is updated sequentially to keep track of program flow.

**Load Testing**

Load testing is a performance evaluation method that assesses a system's behavior under varying load conditions, ensuring stability and efficiency. It helps identify potential failures due to high traffic, ensuring seamless user experiences and preventing system crashes.

**Types of Load Testing**

1. **Smoke Testing**: A preliminary test to check if the software is stable enough for further testing.
2. **Average-Load Testing**: Evaluates system performance under normal operating conditions.
3. **Stress Testing**: Tests the system under extreme conditions to determine its breaking point.
4. **Soak Testing**: Assesses the system's stability over an extended period under normal load conditions.
5. **Spike Testing**: Measures system performance when exposed to sudden increases in load.
6. **Breakout Point Testing**: Determines the point where system performance significantly degrades.

**Load Testing Tools**

Various tools are available for load testing, such as LoadView, which provides functionalities like test recording, execution, distributed testing, and detailed performance reports.

**Load Testing vs. Stress Testing**

While both load and stress testing assess system performance, load testing focuses on evaluating performance under normal or peak conditions, whereas stress testing examines how the system behaves under extreme conditions and its ability to recover afterward.

**Benefits of Load Testing**

* Prevents system crashes and performance degradation.
* Improves user experience by ensuring stability and responsiveness.
* Helps detect performance issues early, reducing long-term costs.
* Supports business growth by ensuring applications can handle increased user demands.

**Scaling in Software Architecture**

Scaling in software architecture refers to a system's ability to accommodate growing workloads, ensuring reliability, availability, and performance. Effective scaling is crucial for handling increased user demand and optimizing resource utilization.

**Types of Scaling**

1. **Vertical Scaling (Scaling Up)**: Enhances the capacity of a single machine by upgrading its hardware components, such as CPU, RAM, and storage. Suitable for small to medium applications and monolithic architectures.
2. **Horizontal Scaling (Scaling Out)**: Expands capacity by adding more instances or servers, distributing the workload across multiple resources. Commonly used in large-scale web applications, cloud-native environments, and microservices architectures.
3. **Diagonal Scaling**: Combines both vertical and horizontal scaling, enabling dynamic resource allocation based on demand. Often utilized in cloud-based infrastructures.

**Function Calls**

Function calls can be categorized as either synchronous or asynchronous, impacting how programs handle concurrent tasks and operations such as networking, I/O, or real-time interactions.

**Synchronous Calls**

A synchronous function call requires the system to complete the current task before proceeding to the next. The calling process remains blocked until execution finishes, ensuring sequential task execution.

**Asynchronous Calls**

In contrast, asynchronous function calls enable concurrent execution, allowing other tasks to proceed without waiting for a previous task to complete. This approach is essential for handling time-intensive operations such as network requests or large-scale data processing.