Chapter 3

SundarBan Payment Gateway

08 January 2025

3.1 Program

3.1.1 Program Execution

The execution of a computer program involves several steps, from the initial writing of the code to its final execution. Here are the key steps in the process.

1. **Writing The code:** The first step is to write the program in a high-level language like C, C++, Python, etc. This code is saved with an appropriate file extension (e.g., .c for C programs)
2. **Preprocessing:** The preprocessor preprocesses the source code. This step involves expanding macros, including header files and processing preprocessor directives like `#include` and `#define`.
3. **Compilation:** The preprocessed code is then compiled by a compiler. The compiler translates the source code into assembly language, which is a low-level representation of the program.
4. **Assemly:** The assembly code is assembled into object code, which is a binary representation of the program. This step is often combined with the compilation steps in many modern compilers.
5. **Linking:** The object code is linked with library functions to create and executable file. This step resolves external references and combines the object code with the necessary libraries to form a complete executable.
6. **Loading:** The executable file is loaded into memory by the loader. The loader sets up the program's environment and initializes the program counter to the starting address of the program.

Below is a simple illustration for better memoization (Figure 3.1):

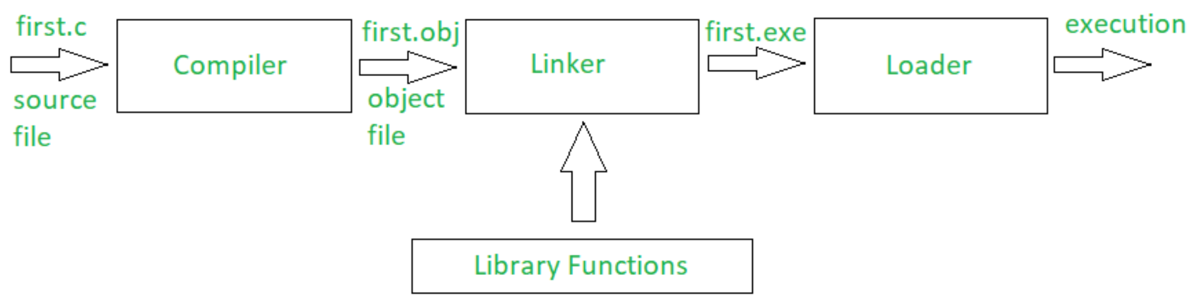

Figure 3.1: Program Execution

3.1.2 Exucatable File

An executable file is a type of file that contains a program that is capable of being executed or run by a computer. When one launches an executable file, the OS loads the file into memory and begin process of running the program that it contains. An executable file typically has a specific structure and format that makes it ready to be executed by a system's CPU. Common formats for executable files include `.exe` on Windows, and ELF (Executable and Linkable Format) or Mach-O on Linux and macOS, respectively. Main components of the executable file is shown below

Header (metadata)
Program Counter (PC)

1. **Header (Metadata):** The header of an executable file contains metadata about the file itself and how the operating system should execute it. This includes file type, entry point, memory layout, and dependencies.

   The metadata in the header helps the operating system (OS) know how to load, map, and execute the program.

2. **Program Counter (PC):** PC is a register within the CPU that keeps track of the address of the next instruction to be executed. When a program is running, the pc is incremented to the next instructions after the current one is executed.

3.2 Load Test

Load testing is a type of performance testing that evaluates how a system behaves under normal and peak load conditions. It helps to ensure that software or websites can handle the demands from the users and prevent crashes, shutdowns, and user frustration.

Load testing simulates real-world user traffic to determine if a system, software, or computing device can handle high loads given a high demand of end users. It is crucial to ensure that applications perform well under expected usage conditions.

Types of Load Testing

Different types of load testing include:

1. smoke testing
2. average-load testing
3. stress testing
4. soak testing
5. spike testing and
6. breakout point testing

Each type focuses on specific aspects of system performance under various conditions.

Load Testing Tools

Tools like **LoadView** offer features such as recording and replaying tests, setting up and running load tests, geo-distributed network capabilities, detailed performance reports, and 24/7 support. This tool help in accurately simulating real-world scenarios.

Load Testing vs Stress Testing

While both load testing and stress testing are subsets of performance testing, they differ in their focus. Load testing checks how systems behave under normal or peak load conditions, whereas stress testing checks how the system behaves beyond normal or peak load conditions and how it responds when returning to normal loads

Benefits of Load Testing

Load testing helps prevent crashes, slowdowns, and frustrated users, and can catch performance issues early to avoid costly problems. It is particularly important as more businesses go digital, and the cost of overlooking performance issues can be significant.

3.3 Scaling in Software Architecture

Scaling refers the to the ability of a software system to handle an increasing number of users, transactions, or workloads while maintaining performance, reliability, and availability. Scaling is a critical aspect of designing robust and resilient systems, especially as they grow in terms of traffic, data volume, or complexity.

In software architecture, scaling is typically considered from two primary perspectives: **scalability of the system** (how it can grow) and **how resources are managed** (how the system adapts to increased load).

Types of Scaling SA

The main two types of scaling in software architecture are:

1. **Vertical Scaling** (Scaling Up)
2. **Horizontal Scaling** (Scaling Out)

Additionally, **diagonal scaling** is also considered in some scenarios, especially in cloud-native and microservice architectures.

1. **vertical Scaling**
   Vertical scaling also known as scaling up, refers to increasing the power of a single server or machine. This typically involves upgrading the hardware (CPU, RAM, Storage) or a server or increasing its capacity to handle more workloads.

   Use cases:

   - small to medium application
   - monolithic architecture

2. **Horizontal Scaling**
   Horizontal scaling, also known as scaling out, refers to adding more instances or nodes (servers) to a system, such as adding additional servers or containers, to distribute the load across multiple resources. It involves running multiple instances of the same service or application and load-balancing requests between them. **Use cases:**

   - large-scale web application
   - cloud-native application
   - microservice architecture

3. **Diagonal Scaling**
   Diagonal scaling is a hybrid approach that combines vertical and horizontal scaling.

   Use cases:

   - cloud environments

3.4 Function Calls

In software development, the terms synchronous and asynchronous describe how different tasks or function calls are executed in relation to each other, particularly when handling operations like I/O, networking, or user interactions. These concepts are crucial for designing efficient systems, especially in applications that require concurrency, parallelism, or real-time responsiveness.

Synchronous Call: A synchronous call is one where the program waits for a task or function to complete before proceeding to the next task. In other words, the calling thread (or process) is blocked until the function or task finishes its execution.

Asynchronous Call: An asynchronous call refers to the ability to perform tasks concurrently without blocking the execution of other tasks. This is particularly useful when dealing with operations that may take a long time to complete, such as file I/O, network requests, or waiting for external resources.