K. J. Somaiya College of Engineering, Mumbai-77

Batch:	Roll No.:
Experiment / assignment / tutorial No	

TITLE: Architecture of any operating system, draw diagram using ADL

AIM: Study of architecture of operating system .

Expected OUTCOME of Experiment:

Using ADL represent operating system architecture

CO 1. Design the architecture of software systems in various architectural styles.

Books/ Journals/ Websites referred:

- https://www.xcubelabs.com/blog/software-architecture-understanding-styles-and-patterns-for-effective-system-design/#:~:text=Architectural Styles are high-level strategies that provide an,or homes. Examples include Layered Event-Driven and Microservices. Last visited on 24 July 2024
-

Background Theory:

A software system's high-level structure, or software architecture, is made up of the rules, patterns, and guidelines that determine how the system is organized, interacts, and is related to its components.

The design decisions will help in providing:

Modularity: allowing the system into interchangeable components that can be developed, tested, and maintained independently.

Encapsulation: Hiding the details of the components, exposing only the necessary information, thus reducing the complexity of the system.

Security: Incorporating the measures to protect the system against the unauthorized access.

Documentation: Provides clear documentation of the system architecture, thus facilitating communication and better understanding of the system.

Performance: Ensuring that the system meets the required performance metrics such as resource utilization, throughput, etc.

There are various types of Software Architectural Styles.

- Data-centered architecture
- Data-flow architecture
- Call and return architectures
- Object-oriented architectures
- Layered architectures

Different architectural patterns will be used for implementing software systems based on:

1. System Requirements

Functional Requirements: What the system needs to do, the features and capabilities it must provide.

Non-Functional Requirements: Attributes such as performance, scalability, security, maintainability, and usability.

2. Design Principles

Separation of Concerns: Dividing the system into distinct sections, each addressing a separate concern.

Modularity: Designing the system in a way that components can be developed, tested, and understood in isolation.

Abstraction: Hiding the complexity of implementation details, providing a simplified interface.

Encapsulation: Bundling data and methods that operate on the data within one unit, restricting access to some of the object's components.

3. Scalability and Performance

The ability to handle increased load, either by scaling up (adding more resources) or scaling out (adding more nodes).

Patterns like microservices and event-driven architecture are designed to improve scalability and performance.

4. Maintainability and Flexibility

Ease of modifying the system to fix bugs, add features, or adapt to changing requirements.

Patterns like layered architecture and component-based architecture promote maintainability and flexibility.

5. Security

Protecting the system from unauthorized access and ensuring data integrity and confidentiality.

Patterns like service-oriented architecture (SOA) and microservices can help isolate and secure individual components.

6. Development Team Skills and Experience

The expertise of the development team with certain technologies and architectural patterns.

Patterns should align with the team's capabilities to ensure effective implementation and maintenance.

7. Technology Stack

The programming languages, frameworks, and tools that will be used in the system.

Patterns should be chosen to leverage the strengths of the chosen technology stack.

8. Integration Requirements

How the system will interact with other systems, services, and databases.

Patterns like microservices and SOA facilitate integration with external systems and services.

9. Deployment Environment

Whether the system will be deployed on-premises, in the cloud, or in a hybrid environment.

Patterns like cloud-native architecture are optimized for cloud deployment.

10. Business Goals

Alignment with the strategic objectives of the organization, including time-to-market, cost constraints, and long-term vision.

Patterns should support achieving the business goals effectively.

11. User Experience

The expectations of end-users in terms of interface design and interaction.

Patterns like model-view-controller (MVC) help create a better user experience by separating concerns.

Software architecture patterns offer reusable designs for various situations, to offer advantages such as improved efficiency, productivity, speed, cost optimization, and better planning.

Laboratory Work:

The previous experiment introduces ArchiMate a tool used for drawing Software architecture design. The students will be required to study architecture of any of the Operating System from the following list and draw diagram

- Ubuntu
- Windows
- macOS

1. System Libraries

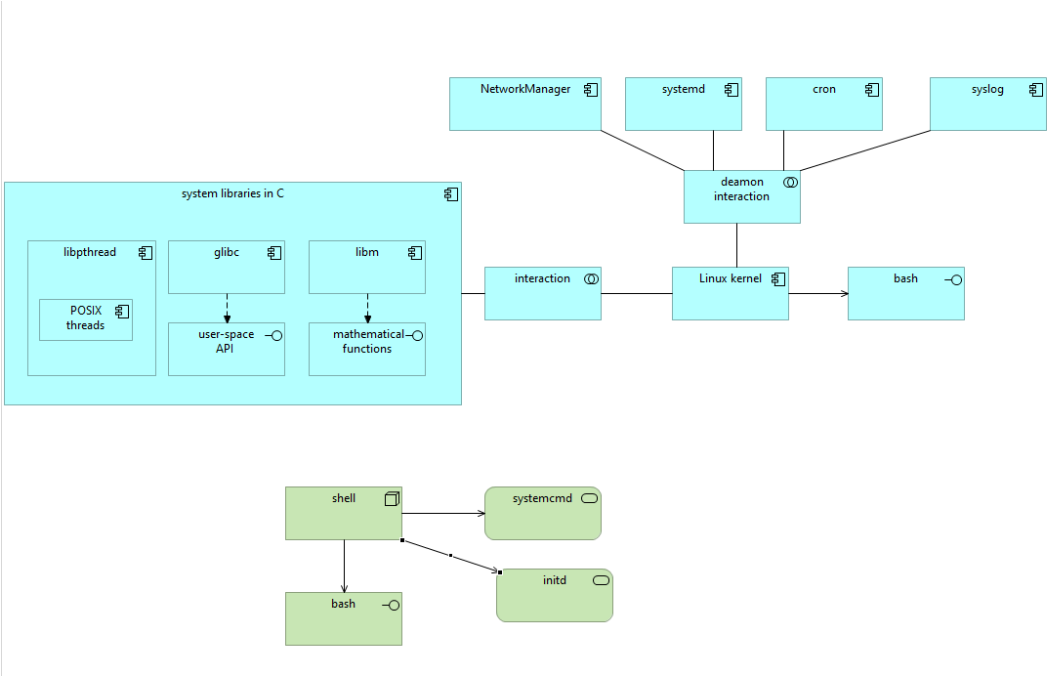
System libraries are critical components of the operating system that provide essential functions and services to applications and other system components. They sit between the kernel and user applications, facilitating interactions and providing standard functionalities.

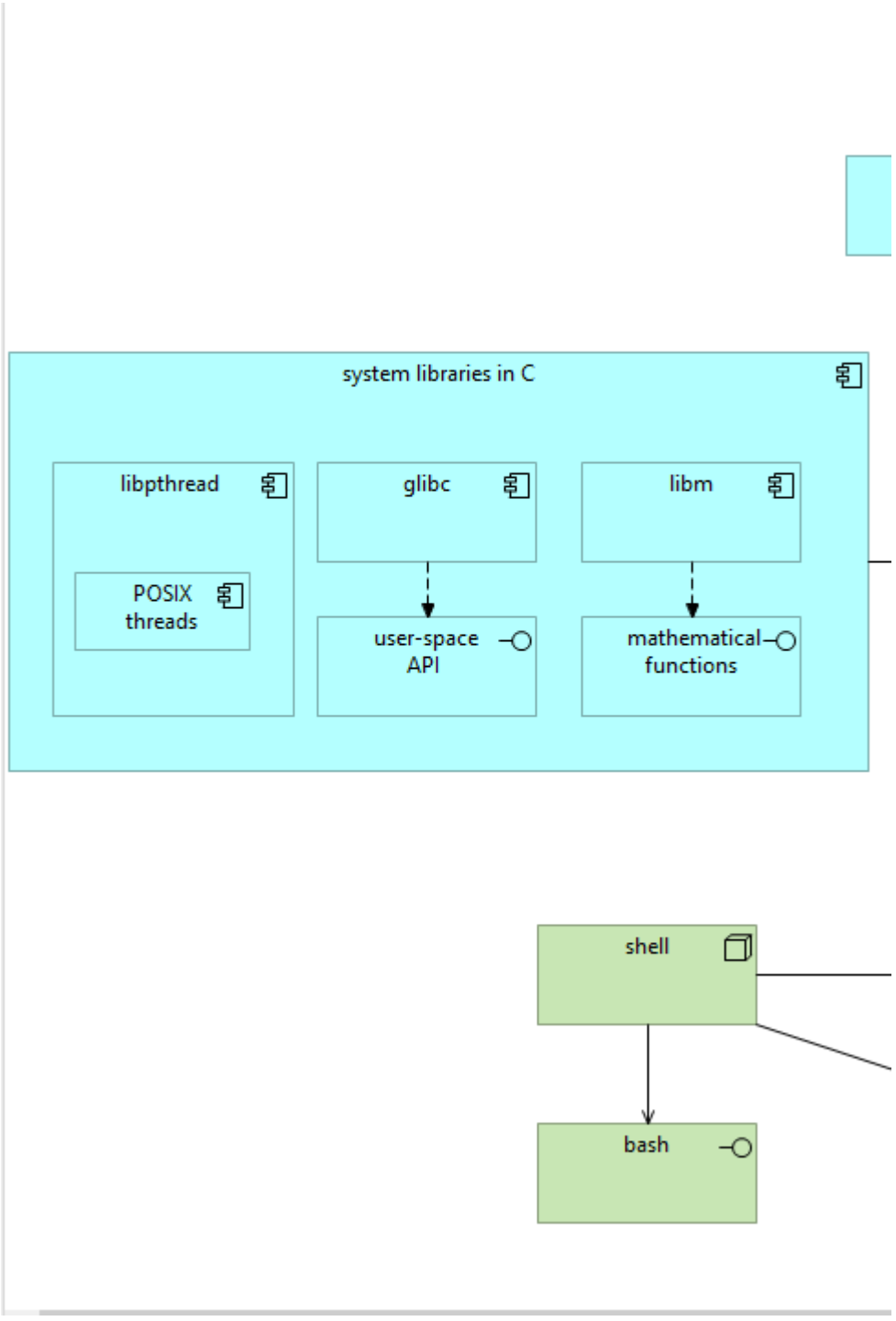
- **C Library (glibc)**:
  - **Role**: The GNU C Library (glibc) is the primary library that provides the system call interface to user-space programs. It includes implementations of standard C library functions such as I/O operations, memory management, and string manipulation.
  - **Interaction with Kernel**: When an application needs to perform a task that requires kernel intervention (like reading a file or allocating memory), it makes a system call to the kernel. The C library provides the user-space API to these system calls. For instance, a read() function in a program calls read() in glibc, which then performs the actual system call to the kernel.
- **Other Libraries**:
  - **libm**: Provides mathematical functions (e.g., sin(), cos(), sqrt()) that are used by applications and are implemented in user space but rely on kernel services for certain functionalities.
  - **libpthread**: Implements POSIX threads for multi-threading support. It interacts with the kernel to manage thread creation, synchronization, and scheduling.

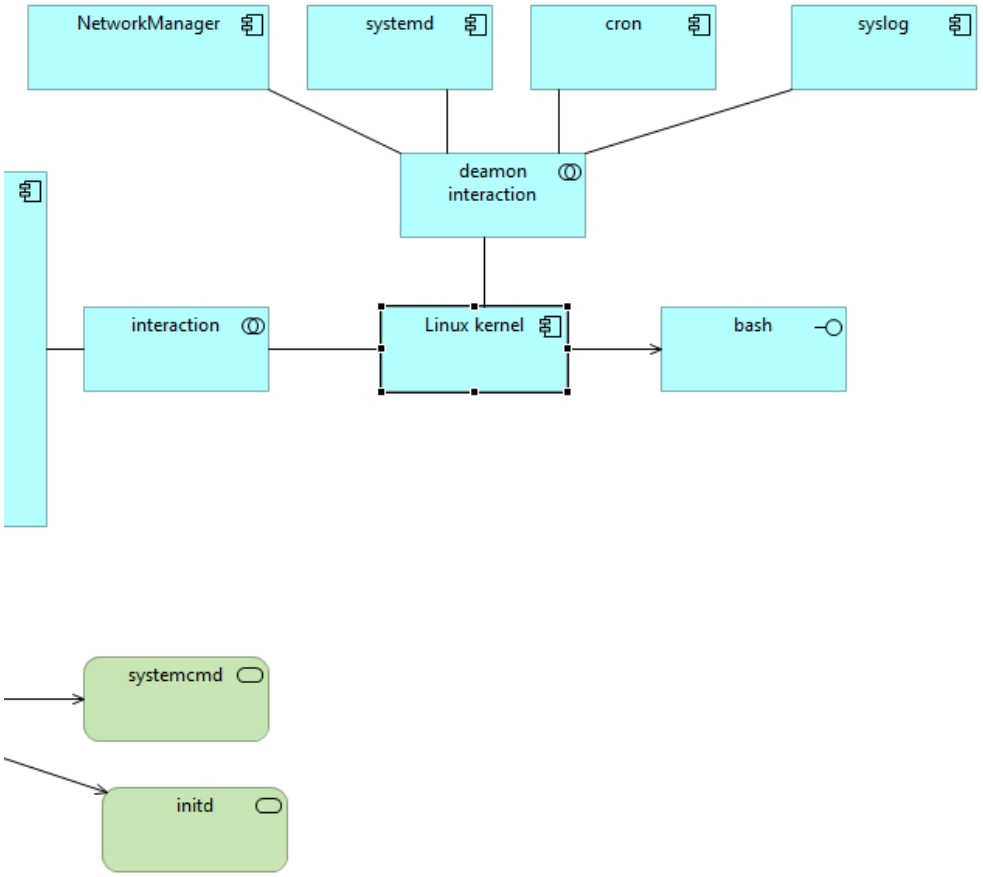
2. System Daemons

System daemons are background processes that manage various aspects of the system and services. They start at boot time and run continuously to handle specific tasks or provide services.

- **Init System (systemd)**:
  - **Role**: systemd is the init system used by Ubuntu for system initialization and service management. It is responsible for booting the system, managing services, and handling dependencies between services.
  - **Interaction with Kernel**: systemd interacts with the kernel to manage system resources, control service execution, and handle system states. For example, it uses the kernel's features to control processes, manage memory, and handle system shutdowns and restarts.
- **Various Daemons**:
  - **NetworkManager**: Manages network connections and interfaces. It communicates with the kernel to configure network interfaces and handle networking events.
  - **cron**: Manages scheduled tasks. It relies on the kernel to handle process scheduling and execution.
  - **syslog**: Handles logging of system and application messages. It uses kernel facilities for logging system events.


NetworkManager
systemd
cron
syslog
deamon interaction
system libraries in C
libpthread
POSIX threads
glibc
libm
user-space API
mathematical functions
interaction
Linux kernel
bash
shell
systemcmd
initd
bash


system libraries in C
libpthread
POSIX threads
glibc
user-space API
libm
mathematical functions
shell
bash



Post Laboratory questions:

- Compare Software Architectural patterns and Design Pattern

Software Architectural Patterns and Design Patterns both provide solutions to common problems in software design but operate at different levels of abstraction:

Software Architectural Patterns:

- Level of Abstraction: High-level
- Scope: Define the overall structure and organization of a system.
- Purpose: Address large-scale concerns such as system organization, how different components interact, and how to achieve scalability, maintainability, and robustness.
- Examples: Microservices, Layered Architecture, Event-Driven Architecture.

Design Patterns:

- Level of Abstraction: Low-level
- Scope: Focus on solving specific problems in a particular context within a given component or subsystem.
- Purpose: Offer solutions to common issues in object creation, composition, and interaction within the context of a specific class or method.
- Examples: Singleton, Observer, Factory Method.

- Explain Various type of Software Architecture Patterns giving example of each

Here's a brief overview of some common software architecture patterns with examples:

- **Layered (n-tier) Architecture**:
  - **Description**: Divides the system into layers where each layer has specific responsibilities. Typically includes presentation, business logic, and data access layers.
  - **Example**: A traditional enterprise application where the user interface, business logic, and database access are separated into different layers.
- **Client-Server Architecture**:
  - **Description**: Divides the system into two parts: clients and servers. Clients request services, and servers provide them.
  - **Example**: Web applications where the web browser (client) requests data from a web server (server).
- **Microservices Architecture**:
  - **Description**: Breaks down an application into small, loosely coupled services, each responsible for a specific function. Each service can be developed, deployed, and scaled independently.
  - **Example**: E-commerce platforms where services like user management, order processing, and payment are handled by different microservices.
- **Event-Driven Architecture**:
  - **Description**: Focuses on the production, detection, and reaction to events. Components communicate by producing and consuming events.
  - **Example**: A real-time chat application where messages are published as events and consumed by clients.

3. Explain in brief the architecture of Android Operating system.

The Android Operating System architecture is designed to be modular and flexible, making it suitable for a wide range of devices. Here's a brief overview:

1. **Linux Kernel**: At the base of the architecture is the Linux kernel, which handles low-level hardware interactions, such as memory management, process management, and hardware abstraction. It provides a foundation for higher-level functionality and security.

2. **Hardware Abstraction Layer (HAL)**: The HAL provides a standard interface for the hardware components, allowing the Android system to interact with different hardware without needing to modify the Android framework. It consists of various modules for different hardware types (e.g., camera, GPS).
3. **Android Runtime (ART)**: ART is the runtime environment for Android applications. It includes a Just-In-Time (JIT) compiler that translates app code into machine code during runtime, and an Ahead-Of-Time (AOT) compiler that compiles the app code before it runs to improve performance.
4. **Libraries**: The libraries layer contains various C/C++ libraries that Android applications can use. This includes core libraries (like libc) and specialized libraries (like SQLite for databases or WebKit for web rendering).
5. **Application Framework**: This layer provides the APIs and services needed for app development. It includes components like Activity Manager, Window Manager, and Package Manager, which help manage application life cycles, user interfaces, and app installations.
6. **Applications**: At the top of the stack are the user-facing applications. These include built-in apps (such as the Phone, Contacts, and Browser apps) and third-party applications installed by the user. They interact with the system through the Application Framework and utilize the underlying Android Runtime and libraries.