A SYNOPSIS ON

LINUX OS from Scratch

Submitted in partial fulfilment of the requirement for the award of the degree of

BACHELOR OF TECHNOLOGY

In

Computer Science & Engineering

Submitted by:

Shailesh Jukaria	**2261519**
Shaurya Singh Panwar	**2261521**
Saurabh Joshi	**2261517**
Khilendra Gauniya	**2261320**

Under the Guidance of

Mr. Prince Kumar

Assistant Professor

Project Team ID: 33

Department of Computer Science & Engineering

Graphic Era Hill University, Bhimtal, Uttarakhand

March-2025

Graphic Era Hill University
BHIMTAL CAMPUS

CANDIDATE'S DECLARATION

We hereby certify that the work which is being presented in the Synopsis entitled **"LINUX OS from Scratch "** in partial fulfilment of the requirements for the award of the Degree of Bachelor of Technology in Computer Science & Engineering of the Graphic Era Hill University, Bhimtal campus and shall be carried out by the undersigned under the supervision of ***Mr. Prince Kumar (Assistant Professor)***, Department of Computer Science & Engineering, Graphic Era Hill University, Bhimtal.

Shailesh Jukaria	2261519
Shaurya Singh Panwar	2261521
Saurabh Joshi	2261517
Khilendra Gauniya	2261320

The above mentioned students shall be working under the supervision of the undersigned on the **"LINUX OS from Scratch"**

Signature — Signature

Supervisor — **Head of the Department**

Internal Evaluation (By DPRC Committee)

Status of the Synopsis: Accepted / Rejected

Any Comments:

Name of the Committee Members: **Signature with Date**

1.

2.

Table of Contents

Chapter 1

Introduction and Problem Statement

1.1) Introduction

Here's a detailed breakdown of the process of building a Linux system from scratch:

1. Setting Up the Environment

Before you begin, set up a development environment with the necessary tools. You will need:

- A host Linux system with essential development tools (gcc, make, binutils, bison, flex, etc.).
- A separate partition or a virtual machine to install the custom Linux system.
- A working internet connection to download source code.

Steps:

- Install dependencies.
- Create a working directory.

2. Downloading Source Code

You need to download the source code for the Linux kernel, GNU utilities, and essential libraries.

Steps:

- Download the Linux kernel:
- Download GNU utilities and libraries from GNU_FTP or use wget.
- Organize all source code in a sources/ directory.

3. Compiling the Toolchain

A toolchain consists of essential tools like gcc, binutils, and glibc, which are required to compile the system.

4. Kernel Configuration and Compilation

The Linux kernel is the core of the operating system.

5. Building Essential Components

Now, build essential system utilities like coreutils, bash, and systemd.

6. Bootloader Setup

The bootloader (GRUB) loads the kernel into memory during startup.

7. Building the Filesystem

A filesystem is necessary for organizing data in Linux.

8. System Configuration

Configure system services and networking.

9. Optimization

You can optimize performance by configuring:

- **Kernel parameters** (sysctl.conf).
- **System services** to minimize unnecessary background processes.

10. Testing and Debugging

Now, test if the system boots properly.

Steps:

- Reboot the system: • sudo reboot
- If the system fails to boot:
  - Check GRUB settings (/boot/grub/grub.cfg).
  - Verify kernel logs (dmesg). o Boot into a rescue mode (init=/bin/sh in GRUB.

1.2) Problem Statement

Most mainstream Linux distributions are designed to serve a wide variety of users and use cases, which often results in **pre-installed software, bloated configurations**, and **excess services** that may not be necessary for every environment. While this approach increases usability and convenience, it introduces inefficiencies in system performance and creates potential **security vulnerabilities** due to the presence of unneeded packages and daemons.

This project addresses those issues by focusing on the development of a **minimalistic, efficient, and customizable Linux system**, specifically tailored to meet unique performance and security requirements. The goal is to build a lean Linux environment from the ground up, ensuring that only essential components are included, and each part of the system is configured with purpose and precision.
Key Objectives and Benefits:

- **Minimalism**: By avoiding unnecessary packages and services, the system becomes lightweight and consumes fewer resources, which is ideal for older hardware, embedded devices, or virtual machines.
- **Customization**: The system can be tailored to a specific use case—whether it be a dedicated firewall, a secure development machine, or a highperformance computing environment.
- **Security**: Reducing the system footprint minimizes the attack surface. Users can control every installed package and apply only the necessary permissions and configurations.
- **Transparency and Control**: Every step in building the system—from compiling the kernel to setting up the user environment—is transparent and fully under the user's control, making it ideal for learning and debugging.

Key Challenges:

- **Understanding Dependencies and Compiling from Source**:
  Most applications rely on a web of dependencies, which must be properly understood and managed. Compiling from source requires deep knowledge of build tools, compiler flags, and dependency resolution.
- **Kernel Optimization**:
  Configuring and compiling a custom kernel is central to system performance. This involves selecting only the necessary drivers, enabling CPU-specific optimizations, and excluding redundant features to reduce boot time and runtime overhead.
- **Bootloader Configuration and System Stability**:
  A misconfigured bootloader can render the system unbootable. Ensuring that GRUB or any other bootloader is properly installed, configured, and maintained is essential for stability. Attention must also be given to init systems and systemd alternatives (e.g., SysVinit, OpenRC) to ensure smooth boot and service management.

Chapter 2

Background/ Literature Survey

2.1 Existing Linux Distributions

Linux has a wide range of distributions (distros) catering to different user needs, from ease of use to high performance and deep system control.

Popular distributions like **Ubuntu**, **Fedora**, and **Arch Linux** offer pre-compiled binaries and package managers that make software installation and system maintenance convenient. These distributions come with pre-set configurations and bundled software, targeting general-purpose usage and rapid deployment. While they are user-friendly and great for beginners or production environments, they often sacrifice fine-grained control in favor of ease of use.

On the other hand, advanced projects like **Gentoo** and **Linux From Scratch (LFS)** shift the control back to the user.

- **Gentoo** uses the Portage system, allowing users to compile packages with specific options (USE flags) tailored to their hardware and needs.
- **Linux From Scratch (LFS)** provides a step-by-step guide to building an entire Linux system from source code, offering complete transparency and control over every installed component.

These customizable approaches cater to users who want to deeply understand Linux internals, optimize performance, and build highly specific systems.

2.2 Importance of Building a Custom Linux System

Creating a custom Linux system is not just an academic exercise—it offers several practical benefits, especially in specialized environments such as embedded systems, servers, or security-sensitive applications.

- **Customization:**
  Building a system from scratch allows users to include only the software and libraries they need. This leads to a leaner, more efficient OS, free from bloatware. It's particularly useful in embedded devices, minimal containers, or systems with limited resources.

- **Security:**
  By minimizing the number of packages and services running on the system, users significantly reduce potential attack vectors. A custom system can be hardened from the ground up by applying security patches manually, disabling unused ports, and avoiding unverified software.

- **Performance:**
  Compiling applications and the kernel with architecture-specific optimizations can lead to improved execution speed and reduced memory usage. Users can strip debug symbols, remove unnecessary kernel modules, and fine-tune system daemons to maximize efficiency.

Chapter 3

Objectives

Most mainstream Linux distributions are designed with general-purpose functionality in mind, catering to a wide variety of users, including home users, developers, and enterprise environments. As a result, these distributions often come **pre-packaged with a large set of software**, graphical environments, device drivers, and background services. While this broad inclusion ensures compatibility and convenience, it also leads to **system bloat**, with many packages and daemons running that may not be necessary for a specific use case.

This excessive inclusion not only consumes valuable system resources but also increases **attack surfaces**, introduces **redundant dependencies**, and often limits the user's ability to **fully understand and control** the underlying system. In performance-critical, resource-constrained, or security-sensitive environments, such generalized configurations are inefficient and, at times, even impractical. **Project Focus**

This project aims to **design and build a custom Linux system from the ground up**, tailored specifically for **minimalism, performance, security**, and **purpose-specific functionality**. By carefully selecting, configuring, and compiling every component— from the kernel to user-space utilities—the system eliminates unnecessary overhead and achieves optimal control over the environment. **Key Objectives and Benefits**

- **Minimalism**:
  Stripping the system of all unnecessary services and applications significantly reduces memory usage, startup time, and disk space. The result is a faster, cleaner operating system that performs optimally, especially in low-power or legacy hardware environments.

- **Customization**:
  Users gain the ability to build a system that matches their exact requirements— whether for networking, development, automation, embedded systems, or system recovery. Every component, from the shell environment to system utilities, can be tailored to fit the purpose.

- **Security**:
  A smaller attack surface means better security. Removing unused daemons, libraries, and user permissions prevents potential vulnerabilities. Users can also compile all software with security-focused compiler flags such as **Stack Smashing Protection (SSP)**, **Position Independent Executables (PIE)**, and **Address Space Layout Randomization (ASLR)** compatibility.

- **Performance Tuning**:
  Systems built from source can be fine-tuned for the specific architecture (e.g., using march=native or -O2/-O3 flags in GCC). Services can be optimized, boot times reduced, and unnecessary I/O operations eliminated.

- **Transparency and Learning**:
  Building everything manually from the Linux kernel to the bootloader provides invaluable experience. It gives developers, system administrators, and cybersecurity professionals a detailed understanding of the OS's internal workings.

Key Challenges and Considerations

- **Understanding Dependencies and Compiling from Source**:
  Modern software is complex and often depends on multiple libraries.
  Identifying, obtaining, and compiling each of these dependencies in the correct order requires a comprehensive understanding of build systems (e.g., make, autotools, cmake) and environment configuration (e.g., PKG_CONFIG_PATH, LD_LIBRARY_PATH).

- **Kernel Optimization**:
  Choosing the right set of kernel modules, enabling or disabling specific features, and applying hardware-specific flags all contribute to building an optimized kernel. Mistakes during this stage can result in hardware incompatibility, system instability, or reduced performance.

- **Bootloader Configuration and System Stability**:
  The bootloader (commonly GRUB or Syslinux) plays a critical role in initializing the system. A misstep here can prevent the system from booting entirely. Understanding partitioning schemes (MBR vs. GPT), filesystem types, and kernel/initrd loading is essential for a stable build.

- **File System Hierarchy and Package Management**:
  Unlike traditional distributions, where a package manager maintains file locations and versions, custom-built systems require the user to **manually maintain the FHS (Filesystem Hierarchy Standard)**. This includes managing directory structures, user permissions, and environment paths. Without automation, consistency must be rigorously maintained through manual scripts and careful logging.

- **Init System Integration**:
  Choosing and configuring the right init system (e.g., **SysVinit**, **OpenRC**, or **runit**) is key to how services are started, stopped, and monitored. Each init system has its syntax, dependencies, and runtime behavior, which must align with the system's intended use.

- **Networking and System Services**:
  Networking must often be configured manually, including static IP assignments, hostname resolution, and service daemon setups (like SSH or NTP). Proper firewall configuration using tools like iptables or nftables is essential to maintaining security.

- **Logging and Monitoring**:
  Without built-in logging utilities, system logs must be manually implemented using tools like syslog-ng, rsyslog, or even custom scripts. System stability relies heavily on observing logs for kernel panics, crashes, or hardware issues.

- **Documentation and Troubleshooting**:
  Since this is a unique build, it diverges from standard documentation. Thorough notes, shell scripts, and version tracking (possibly using tools like Git) are vital for reproducibility and future debugging. Troubleshooting can be timeconsuming due to the lack of standard support channels.

Chapter 4

Hardware and Software Requirements

4.1 Hardware Requirements

S. No.	Name	Specification
1	Processor	Intel i5/i7 or equivalent
2	RAM	8GB or more
3	Storage	250GB SSD or more
4	Network	High-speed internet connection

4.2 Software Requirements

S. No.	Component	Specification
1	Base OS	Any existing Linux distribution (Ubuntu, Fedora, etc.)
2	Compiler	GCC, Clang
3	BootLoader	GRUB
4	Kernel	Latest Linux Kernel Source

Chapter 5

Possible Approach/Algorithms

The process of setting up and optimizing a custom Linux environment was carried out through a structured series of steps, detailed as follows:

1. Environment Initialization

To begin, the system environment was prepared for development:

- Logical disk partitions were created and appropriate file systems were configured to facilitate system organization and performance.
- A chroot (change root) environment was established to isolate the build environment from the host system, ensuring greater control and consistency during the compilation process.

2. Linux Kernel Compilation

This phase focused on obtaining and preparing the Linux kernel:

- The latest stable version of the Linux kernel was downloaded from the official source.
- Kernel configuration was performed using tools like make menuconfig to enable/disable specific features.
- The kernel was compiled and installed, including the creation of necessary initramfs/initrd images.

3. Bootloader Configuration

Once the kernel was ready, bootloader configuration ensured proper system startup:

- GRUB (GRand Unified Bootloader) was installed as the system boot manager.
- GRUB configuration files were modified to include the newly compiled kernel.
- The system was tested to confirm a successful boot into the custom kernel environment.

4. System Services and Cron Job Setup

With the base system operational, automation and essential services were configured:

- Core libraries and utilities necessary for system functionality were installed.
- Network configuration was completed, ensuring reliable connectivity..

References

1. G. Negus, Linux Bible, 9th ed. Indianapolis, IN, USA: Wiley, 2015.

2. G. Beekmans, Linux From Scratch, 11.3 ed., The Linux From Scratch Project, Dec. 2023.

3. R. Love, Linux Kernel Development, 3rd ed. Boston, MA, USA: Addison-Wesley, 2010.

4. J. Hall, Linux Command Line and Shell Scripting Bible, 3rd ed. Indianapolis, IN, USA: Wiley,
5. 2020.

6. The GNU Project, “GNU Operating System,” Free Software Foundation.

7. D. Bovet and M. Cesati, Understanding the Linux Kernel, 3rd ed. Sebastopol, CA, USA: O’Reilly Media, 2005.

8. S. Das, Your UNIX/Linux: The Ultimate Guide, 3rd ed. New York, NY, USA: McGraw-Hill Education, 2012.