SCHOOL:KARENGE ADVANTIST SECONDARY SCHOOL
TRAINER:NDIKUMANA Eric

## Learning outcome 1: Prepare working environment

## Chapter 1: Selection of tools and equipment

### 1.0.Hobby kernel description

#### 1.0.1.What is Kernel?

A kernel is the core part of an operating system. It acts as a bridge between software applications and the hardware of a computer. The kernel manages system resources, such as the CPU, memory, and devices, ensuring everything works together smoothly and efficiently. It handles tasks like running programs, accessing files, and connecting to devices like printers and keyboards.

#### 1.0.2.Kernel in Operating System

A kernel is a central component of an operating system that manages the operations of computers and hardware. It basically manages operations of memory and CPU time. It is a core component of an operating system. Kernel acts as a bridge between applications and data processing performed at the hardware level using inter-process communication and system calls.

#### 1.0.3.Working of Kernel

- A kernel loads first into memory when an operating system is loaded and remains in memory until the operating system is shut down again. It is responsible for various tasks such as disk management, task management, and memory management.
- The kernel has a process table that keeps track of all active processes
- The process table contains a per-process region table whose entry points to entries in the region table.
- The kernel loads an executable file into memory during the ‘exec’ system call’.
- It decides which process should be allocated to the processor to execute and which process should be kept in the main memory to execute. It basically acts as an interface between user applications and hardware. The major aim of the kernel is to manage communication between software i.e. user-level applications and hardware i.e., CPU and disk memory.

#### 1.0.4.Objectives of Kernel

- To establish communication between user-level applications and hardware.
- To decide the state of incoming processes.
- To control disk management.
- To control memory management.
- To control task management.

### 1.1.1.Definition

A hobby kernel is a simplified, usually non-commercial operating system kernel created for educational or personal enjoyment purposes by enthusiasts. It serves as a playground for learning about low-level system programming, operating system design, and kernel development.

### 1.1.2.Role of Kernel in Operating System

The kernel is the core component of an operating system that manages system resources (CPU, memory, devices) and provides a foundation for higher-level software to run. It handles tasks such as process scheduling, memory management, file system access, and device driver interfaces.

### 1.1.3.Benefits of Hobby Kernel Development

1. **Learning Experience:** Developing a hobby kernel provides hands-on experience in system-level programming, understanding hardware interaction, and debugging complex software.
2. **Customization:** Hobby kernels can be tailored to specific needs or preferences, allowing developers to experiment with unique features or optimizations.
3. **Understanding Operating Systems:** It deepens understanding of how operating systems work internally, including the interaction between hardware and software layers.
4. **Community Engagement:** Engaging in hobby kernel projects often involves participating in online communities, sharing knowledge, and collaborating with other enthusiasts.
5. **Career Development:** Skills gained from hobby kernel development can be valuable for careers in system programming, kernel development, and low-level software engineering.

## 2.1.Requirements for hobby kernel

To develop a hobby kernel, there are several key requirements and tools needed to ensure the process runs smoothly. These are categorized into three main aspects: target architecture, virtualization tools, and development tools.

### 2.1.1.Target Architecture

A hobby kernel must be developed for a specific hardware architecture, which dictates how the kernel interacts with the Central Processing Unit (CPU), memory, and devices. Popular target architectures include:

- **x86 / x86_64 (Intel/AMD)**: This is the most widely used architecture due to its extensive support, documentation, and resources. It is ideal for beginners because of the large community and available tutorials.
- **ARM**: Commonly used in mobile devices and embedded systems, the Advanced RISC Machine (ARM) architecture is known for its low power consumption and

versatility. ARM-based development is particularly useful for Internet of Things (IoT) or mobile operating system kernel projects.
- **RISC-V**: A modern, open-source hardware instruction set that is growing in popularity. Reduced Instruction Set Computer (RISC) V's open nature allows for extensive customization, making it a great choice for those wanting full control over the hardware-software interaction.

Choosing an architecture early on is crucial as it affects everything from assembly code to how the kernel will manage hardware resources.

---

### 2.1.2. Virtualization Tools

Virtualization tools allow developers to test their hobby kernel in a controlled, virtual environment. These tools simulate real hardware, providing an ideal platform for safe testing without the risk of crashing or damaging an actual machine. Common virtualization tools include:

- **Quick Emulator (QEMU)**: An open-source emulator that supports a wide range of hardware architectures, including x86, ARM, and RISC-V. It is extremely versatile and can run both user-mode applications and entire operating systems. QEMU is commonly used for kernel development because of its debugging support.
- **VirtualBox**: This is a general-purpose virtualizer that supports a variety of guest operating systems. It's user-friendly and widely used, though it's slightly less flexible for kernel debugging compared to QEMU.
- **Bochs**: A highly detailed x86 emulator. Bochs is slower than QEMU but is great for low-level debugging, providing an in-depth view of Central Processing Unit (CPU) operations during kernel execution.

Using virtualization tools helps developers avoid the risks associated with testing kernels on real hardware. These tools allow fast recovery from crashes, offer snapshot support, and provide detailed debugging.

---

### 2.1.3. Development Tools

A variety of tools are necessary to write, compile, and test a hobby kernel. Each tool plays a critical role in the development pipeline, from writing code to compiling and debugging the kernel.

#### *1.Cross-Compiler*

A cross-compiler is needed to build a kernel for the target architecture on your development machine. For example, if you are developing an x86-based kernel on a Linux system, you will use a compiler like **x86_64 Executable and Linkable Format GNU Compiler Collection**

**(x86_64-elf-gcc)**. The cross-compiler translates your code into machine instructions compatible with the target architecture.

### *2.Linker (LD)*

The linker is responsible for combining object files generated by the compiler into a single binary that can be loaded into memory and executed. A custom linker script is often required for kernel development to manage memory layout, entry points, and segment alignment.

### *3.Assembler*

Low-level system code often requires the use of assembly language, which directly interacts with the Central Processing Unit (CPU). An assembler like **Netwide Assembler (NASM)** is used to write and compile assembly code. Assembly is commonly used to set up the CPU environment, handle interrupts, and manage low-level bootstrapping processes.

### *4.Bootloader (Grand Unified Bootloader (GRUB) or Limine)*

A bootloader is required to load the kernel into memory during the booting process. **Grand Unified Bootloader (GRUB)** is a popular, multi-boot compliant bootloader. It supports multiple architectures and makes it easier to load kernels developed for hobby purposes. Alternatively, **Limine** is a modern bootloader often used in hobbyist circles, especially for newer systems.

### *5.Debugger (GNU Debugger (GDB))*

Debugging a kernel is essential for tracking down bugs in memory management, scheduling, or hardware interactions. **GNU Debugger (GDB)** is a popular debugging tool that can be used in conjunction with QEMU or Bochs to step through kernel code, view CPU registers, and analyze memory during execution. GDB offers remote debugging capabilities, which is ideal for low-level system development.

### *6.Text Editor or Integrated Development Environment (IDE)*

A suitable development environment is key for efficient kernel development. Lightweight text editors like **Vim** or **Emacs** are popular among hobbyists due to their customization and flexibility. For a more user-friendly experience, Integrated Development Environments (IDEs) like **Visual Studio Code** or **CLion** can be configured for kernel development, providing advanced features such as syntax highlighting, code completion, and integrated debugging.



---

## Example Development Workflow

1. **Choose a Target Architecture**: Select an architecture such as x86 or ARM for which to design the kernel.

2. **Set Up Cross-Compiler**: Install and configure a cross-compiler that can generate machine code for the selected architecture.
3. **Write Bootloader**: Use Grand Unified Bootloader (GRUB) or a similar bootloader to load your kernel into memory during boot.
4. **Develop Kernel Components**: Start coding your kernel in a low-level language (like C or assembly), focusing on features like memory management, task scheduling, and hardware abstraction.
5. **Test Using Virtualization Tools**: Use Quick Emulator (QEMU), Bochs, or VirtualBox to test your kernel in a safe, virtualized environment.
6. **Debug**: Use GNU Debugger (GDB) to find and fix bugs in your kernel code, paying special attention to memory access violations, crashes, and CPU behavior.

### Summary

- Target Architecture**: Choose x86, ARM, or Reduced Instruction Set Computer (RISC)-V based on your kernel's goals.**
- Virtualization Tools**: Use Quick Emulator (QEMU), VirtualBox, or Bochs for safe and flexible testing.**
- Development Tools**: Utilize cross-compilers, linkers, bootloaders (Grand Unified Bootloader (GRUB)), assemblers, and debuggers like GNU Debugger (GDB) for effective kernel creation and testing.**

By following these requirements and using the right tools, developers can build, test, and refine their hobby kernel projects while gaining valuable insights into operating system design and low-level programming.

## 1.3. Identification of the right tools and equipment to use in development.

Choosing the right tools and equipment is critical for building a hobby kernel because the kernel interacts closely with hardware, and its development involves low-level programming. Here's an overview of the key tools you need and their roles:

---

### 1. Cross-Compiler

- Purpose**: A cross-compiler is a tool that compiles code written for one system (your development machine) into a binary that can run on another system (the target architecture, such as x86 or ARM).**
- Why It's Important**: You cannot use the same compiler for regular applications and kernels. A cross-compiler like the** GNU Compiler Collection (GCC) **is configured specifically for the target platform (e.g., x86, ARM), ensuring the compiled kernel is compatible with the hardware you're developing for.**

---

## 2. Assembler

- Purpose**: An assembler converts low-level assembly language code into machine code that the Central Processing Unit (CPU) can execute directly.**
- Why It's Important**: Kernel development often requires direct manipulation of hardware, which is done through assembly language. Tools like** Netwide Assembler (NASM) **or** GNU Assembler (GAS) **are crucial for tasks like initializing the CPU, handling interrupts, and interacting with hardware components.**

---

## 3. Linker

- Purpose**: The linker combines multiple object files (small pieces of compiled code) into a single executable file that can be loaded into memory and executed.**

- Why It's Important**: The kernel needs a custom memory layout. The** GNU Linker (LD) **allows developers to control the placement of various sections of the kernel code, ensuring that the kernel is loaded into the right memory areas when the system boots.**

---

## 4. Bootloader

- Purpose**: The bootloader is the first software that runs when a computer starts. It loads the kernel into memory and starts its execution.**
- Why It's Important**: Without a bootloader, your kernel will not start.** GRUB (Grand Unified Bootloader) **and** Limine **are commonly used bootloaders that help load the kernel from disk and pass control to it.**

---

## 5. Virtualization/Emulation Tools

- Purpose**: Virtualization and emulation tools simulate hardware environments, allowing developers to test their kernel without the need for physical hardware.**
- Why It's Important**: Testing kernels on real hardware is risky (a kernel bug can crash or damage the system). Tools like** Quick Emulator (QEMU), VirtualBox**, and** Bochs **create safe, virtualized environments to test and debug the kernel, ensuring it works properly before being deployed on actual hardware.**

---

## 6. Debugger

- Purpose**: A debugger helps you find and fix bugs in your kernel code by allowing you to pause execution, inspect CPU registers, memory, and see how the code is being executed step by step.**
- Why It's Important**: Kernel bugs can be hard to track because they often involve low-level issues like memory access violations or CPU crashes.** GNU Debugger

(GDB), **when used with QEMU, can provide remote debugging capabilities to inspect kernel behavior and fix critical bugs.**

---

### 7. Text Editor or Integrated Development Environment (IDE)

- Purpose**: A text editor or IDE is where you write and manage the kernel code.**
- Why It's Important**: Efficient coding requires tools that offer syntax highlighting, code completion, and debugging integration. Editors like** Vim **or** Emacs **are lightweight and customizable, while IDEs like** Visual Studio Code **or** CLion **provide advanced features for productivity and debugging support.**

---

### 8. Version Control System

- Purpose**: Version control systems manage changes to your codebase, allowing you to track progress, collaborate with others, and roll back to previous versions if needed.**
- Why It's Important**: Kernel development involves complex changes, and a tool like** Git **helps keep track of every change, prevents accidental data loss, and facilitates collaboration in open-source projects.**

---

### 9. Build Automation Tool

- Purpose**: Build automation tools simplify the process of compiling, linking, and testing your kernel by automating repetitive tasks.**
- Why It's Important**: Kernels involve multiple files and components. Tools like** Make**, along with a Makefile, ensure that all files are compiled and linked in the correct order, speeding up the development cycle and reducing human error.**

### Summary of Key Tools and Their Purpose

| Tool | Purpose | Recommended Options |
|---|---|---|
| Cross-Compiler | Compile code for the target architecture | GCC (x86_64-elf-gcc, arm-none-eabigcc) |
| Assembler | Compile assembly code | NASM, GAS |

| | | |
|---|---|---|
| Linker | Combine object files into a single binary | GNU Linker (LD) |
| Bootloader | Load the kernel into memory | GRUB, Limine |
| Virtualization Tool | Simulate hardware for testing | QEMU, VirtualBox, Bochs |
| Debugger | Debug and inspect kernel execution | GNU Debugger (GDB) |
| Text Editor/IDE | Write and manage kernel code | Vim, Emacs, Visual Studio Code, CLion |
| Version Control | Track changes and manage versions | Git |
| Build Automation | Automate the building and testing process | Make |

## Chapter 2: Installation of Software tools

The **installation of software tools** refers to the process of setting up various programs, applications, and utilities on a computer that are essential for performing specific tasks or

projects. In the context of hobby kernel development, these tools include compilers, emulators, editors, and debuggers. Each tool serves a distinct purpose in the workflow, such as writing code, compiling it into machine-readable format, or testing the kernel in a virtual environment.

Proper installation and configuration of these tools ensure a smooth and efficient development process. For hobby kernel development, the following software tools are typically required:

- **QEMU**: For virtualizing and testing the kernel.
- **QtEMU**: A graphical frontend for QEMU.
- **HEX Editor**: To view and edit binary files.
- **Text Editor (e.g., VSCode)**: For writing and editing code.
- **NASM**: An assembler for assembly programming.
- **SASM**: An IDE for working with assembly languages.
- **MinGW**: A lightweight compiler for compiling C programs.

Here's a guide on how to install the essential software tools for hobby kernel development.

### 1. Installing QEMU (Quick Emulator)

**QEMU** is a powerful emulator that allows you to test your hobby kernel in a virtual environment.

#### *For Windows:*

1. **Download QEMU from the official website: QEMU for Windows.**
2. **Install QEMU by running the installer and following the on-screen instructions.**
3. **After installation, add the qemu executable directory to your system's PATH environment variable for easy access via the command line.**

#### *For Linux:*

1. **Open a terminal.**
2. **Run the following command to install QEMU:**

   sudo apt install qemu-system-x86

3. **Verify the installation by running:**

   qemu-system-x86_64 --version

## 2. Installing QtEMU (Graphical Frontend for QEMU) QtEMU

provides a user-friendly graphical interface for QEMU.

### *For Windows:*

1. **Download the latest version of QtEMU from SourceForge.**
2. **Extract the downloaded file and run the installer.**
3. **During installation, ensure that QEMU is installed beforehand, as QtEMU will require the QEMU path to be configured.**

### *For Linux:*

1. **Open a terminal.**
2. **Run the following commands:**

   sudo apt install qtemu

3. **Once installed, launch QtEMU by searching for it in your application menu or typing qtemu in the terminal.**

---

## 3. Installing a HEX Editor

A **HEX Editor** is useful for viewing and editing binary files at the byte level.

### *For Windows:*

1. **Download and install** HxD **from the official website: HxD Download.**
2. **Follow the installer prompts to complete the installation.**

### *For Linux:*

1. **Open a terminal and install** HexEdit**:**

   sudo apt install hexedit

2. **Run the HexEdit tool by typing hexedit in the terminal.**

---

## 4. Installing Visual Studio Code (Text Editor)

**Visual Studio Code (VSCode)** is a versatile text editor with support for many programming languages, including C, C++, and Assembly.

### *For Windows:*

1. **Download Visual Studio Code from the official website: VSCode Download.**
2. **Run the installer and follow the prompts to complete the installation.**

*For Linux:*

1. **Open a terminal and run:**

   sudo apt install code

2. **After installation, launch Visual Studio Code by typing code in the terminal.**

---

## 5. Installing NASM (Netwide Assembler)

**NASM** is a popular assembler used for x86 architecture programming.

*For Windows:*

1. **Download the NASM installer from the official website: NASM Download.**
2. **Run the installer and add NASM to your PATH for command line access.**

*For Linux:*

1. **Open a terminal and run:**

   sudo apt install nasm

2. **Verify the installation by typing:**

   nasm -v

---

## 6. Installing SASM (Simple Assembler)

**SASM** is an IDE for assembly language programming that supports NASM and other assemblers.

*For Windows:*

1. **Download SASM from the official GitHub repository: SASM GitHub.**
2. **Extract the downloaded file and run the installer.**

*For Linux:*

1. **Open a terminal and run:**

sudo apt install sasm

2. **Once installed, you can run SASM by typing sasm in the terminal.**

---

## 7. Installing MinGW (Minimalist GNU for Windows)

**MinGW** provides a lightweight development environment for compiling C and C++ programs.

### *For Windows:*

1. **Download the** MinGW installer **from the official website: MinGW Download.**
2. **Run the installer and select the gcc and g++ packages for installation.**
3. **Add MinGW to your system's PATH environment variable to use the compiler from the command line.**

---

## 8. Configuring MinGW for Compiling C Programs

After installing MinGW, it must be properly configured to compile C programs.

1. **Set the PATH Variable**:
   - **Go to** Control Panel > System > Advanced System Settings > Environment Variables**.**
   - **Under "System variables," find** Path **and click** Edit. ○ **Add the path to MinGW's bin directory (e.g., C:\MinGW\bin).**
   - **Click** OK **to save changes.**
2. **Verify the Configuration**:
   - **Open** Command Prompt **and run:**

     gcc --version

   - **If the configuration is correct, you should see the version of GCC installed.**



## Chapter 3: Configuring virtual environment

Configuring a virtual environment for hobby kernel development is a crucial step to ensure that all necessary tools are properly installed, compatible with the hardware, and function together seamlessly. This process involves several key tasks including identifying hardware compatibility, setting system paths for software, configuring virtualization tools, and testing the entire setup. Below is a full description of each step:

SCHOOL:KARENGE ADVANTIST SECONDARY SCHOOL
TRAINER:NDIKUMANA Eric

## *1. Identifying Hardware Compatibility*

Before beginning kernel development, it's important to ensure that the computer hardware is capable of supporting virtualization and development tasks. Virtualization tools such as QEMU require certain hardware features to function effectively.

- **Processor Compatibility**: Ensure that the CPU supports hardware virtualization technologies like Intel VT-x or AMD-V. These features allow efficient operation of virtual machines by utilizing the processor's native capabilities. ○ How to Check**:**
    - **On** Windows**:**
        1. **Open** Task Manager **by pressing Ctrl + Shift + Esc.**
        2. **Go to the** Performance **tab.**
        3. **Under the** CPU **section, check if "Virtualization" is enabled.**
    - **On** Linux**: Open a terminal and type:**

      egrep -o '(vmx|svm)' /proc/cpuinfo

      **If the output shows "vmx" (Intel) or "svm" (AMD), the processor supports virtualization.**

- **Memory and Storage**: Ensure the computer has sufficient RAM and disk space. Virtual machines (VMs) typically need at least 512MB to 2GB of memory for smooth operation. Disk space is also needed for VM images and kernel files.

---

## *2. Configuring Installed Software to Environment Path*

After installing necessary software tools such as compilers (MinGW), assemblers (NASM), and virtualization tools (QEMU), it's important to add these tools to the system's PATH. The PATH variable allows you to access the software from any directory in the terminal or command prompt without needing to navigate to the installation folder.

- **Configuring PATH on Windows**:
    1. **Open** Control Panel **and go to** System **>** Advanced System Settings**.**
    2. **Click** Environment Variables**.**
    3. **Under** System Variables**, find the** Path **variable, select it, and click** Edit**.**
    4. **Add the installation paths of the tools (e.g., C:\MinGW\bin, C:\NASM\, C:\QEMU\bin) one by one.**
    5. **Click** OK **to save the changes.**
- **Configuring PATH on Linux**:
    1. **Open a terminal and edit the** .bashrc **or** .zshrc **file, depending on the shell you use:**

       nano ~/.bashrc

    2. **Add the installation path of each tool to the PATH. For example:**

```
export PATH=$PATH:/path_to_mingw/bin
export PATH=$PATH:/path_to_nasm export
PATH=$PATH:/path_to_qemu/bin
```

3. **Save the file and run:**

   source ~/.bashrc

- This ensures that the system recognizes the installed tools when you run commands.

---

### *3. Configuring Installed Virtualization Tools*

Once the virtualization tools are installed (such as QEMU and QtEMU), you need to configure them for kernel development. QEMU is commonly used for testing kernels in a virtual machine, and QtEMU provides a graphical user interface (GUI) for easier management.

- QEMU Configuration**:**
  1. Verify Installation**: After installing QEMU, verify that it's properly installed by opening a terminal or command prompt and running:**

     qemu-system-x86_64 --version

  2. Basic QEMU Command**: To run your kernel in QEMU, use the following command structure:**

     qemu-system-x86_64 -kernel path_to_your_kernel.bin -m 512 -drive format=raw,file=disk_image.img

     - **-kernel: Specifies the location of the kernel binary.**
     - **-m 512: Allocates 512MB of RAM to the virtual machine.**
     - **-drive: Defines the virtual disk file and format.**
- QtEMU Configuration**:**
  1. **Download and install QtEMU from SourceForge.**
  2. **In QtEMU settings, link the installation path to QEMU, so the graphical interface can interact with the QEMU binary.**
  3. **Configure the virtual machines by specifying the location of the kernel binary and disk images.**

---

### *4. Running, Checking, and Testing Tools and Environment*

After setting up all tools, it's time to test them to ensure the development environment is functioning correctly.

- **Test Programs**:
  - Create a simple "Hello World" program in C or Assembly to verify your compiler setup.
  - **For C**:
    1. **Write a simple C program (test_program.c):**

```
#include <stdio.h> int
main() {    printf("Hello
World\n");
  return 0;
}
```

    2. **Compile it using MinGW:**

```
gcc -o test_program test_program.c
```

    3. **Run the program to ensure the compiler is working.**
  - **For Assembly (using NASM)**:
    1. **Write a basic assembly program (test_program.asm):**

```
section .data
msg db 'Hello, World!',0

section .text
global _start

_start:
   ; write(1, msg, 13)
mov eax, 4

   mov ebx, 1
mov ecx, msg
mov edx, 13
   int 0x80

   ; exit(0)
mov eax, 1
xor ebx, ebx
int 0x80
```

    2. **Compile it with NASM:**

```
nasm -f elf test_program.asm
ld -m elf_i386 -s -o test_program test_program.o
```

3. **Run the program to verify your NASM setup.**

- **Testing in QEMU**: Once your kernel or test program is ready, run it in QEMU to check for errors and to see if it behaves as expected. Use a command like this to run your program:

  bash Copy
  code
  qemu-system-x86_64 -kernel path_to_test_program -m 512

- **Debugging**: If you encounter errors or the environment doesn't work as expected, check the paths, environment variable settings, and hardware compatibility. Make adjustments as needed to correct any issues.

---

## Summary

Configuring a virtual environment for hobby kernel development ensures that you have all the necessary tools and systems in place to begin writing, compiling, and testing your kernel. From ensuring hardware compatibility to configuring the PATH variables and setting up virtualization tools like QEMU, each step is vital in building an efficient development environment. Once configured, test the environment using simple programs and debug any issues to ensure smooth kernel development.