**Report: Understanding the Bare Bones and Meaty Skeleton Kernels**

**Introduction:**

The journey of building a **barebones kernel** and a **meaty skeleton kernel** was both challenging and rewarding. Initially, our goal was simple: create a kernel that could boot using **GRUB** (Grand Unified Bootloader) and display a message to the screen. What seemed like a straightforward task soon turned into a multi-step process filled with learning and problem-solving. We started with the idea of building one simple kernel, but along the way, we realized that we needed to create two separate kernels: **Bare Bones** and **Meaty Skeleton(**we weren’t able to consolidate it into one kernel!).

In this report, we will explain the entire process, starting with what each kernel does, how we built them, and the challenges we faced along the way. The journey of building these two kernels was like assembling a **basic machine**, with the first kernel being just a **simple frame** and the second one adding the necessary parts to make it functional. This project gave us hands-on experience in understanding the foundational components of an **operating system**, and the importance of the kernel in managing computer resources.

**What Happens in the Bare Bones Kernel?**

The **Bare Bones Kernel** is the simplest form of a kernel that you can imagine. It is like starting from scratch — **a foundation** with no extra features added. This kernel serves as the **most basic example** of what a kernel can do. Here's what happens in the Bare Bones kernel:

1. **The Core Concept:**
   * The **Bare Bones Kernel** is minimalistic. Think of it as a **skeleton** that provides the basic structure without any extra functionality.
   * The only job of this kernel is to **display a message**, specifically: **"Hello, Bare Bones Kernel!"** on the screen. This step is like learning to say “Hi!” before learning to have a full conversation.
2. **Why Is It Called Bare Bones?**
   * "Bare Bones" refers to something that is stripped down to its most basic form. It’s like building a **framework** for something larger, but with no features yet. It's **empty** but structured. The kernel, in this case, just displays a message and doesn't interact with other hardware or manage tasks.
3. **How Does It Work?**
   * **GRUB** (the bootloader) is the first thing that runs when the system starts. Think of GRUB as the **key holder** — it unlocks the kernel and loads it into memory.
   * Once the kernel is loaded, it writes the "Hello, Bare Bones Kernel!" message to the screen. This is done by accessing the **video memory**, which is like **opening a blank notebook** and writing in it.
   * However, after printing the message, the kernel doesn’t do anything else. It's like someone saying "Hi" and then standing still. The kernel doesn't respond to user input or manage anything else on the computer.
4. **Limitations and Why It’s Useful:**
   * The **Bare Bones Kernel** doesn’t handle **user input**, **file systems**, or any other processes. Its sole purpose is to demonstrate the very **basics of kernel creation**.
   * Despite its simplicity, this kernel is very important for understanding **how operating systems boot** and how the kernel works to **interact with the hardware**.

**What Happens in the Meaty Skeleton Kernel?**

The **Meaty Skeleton Kernel** is an extension of the Bare Bones Kernel, but with more functionality added to it. It's like taking the **Bare Bones framework** and adding the **necessary parts** to make it work. Here's a breakdown:

1. **The Start of Something Bigger:**
   * Unlike the Bare Bones kernel, the **Meaty Skeleton Kernel** does more than just print a message. Before printing, it performs critical tasks like setting up the **Global Descriptor Table (GDT)** and handling **interrupts**.
   * These additional setup tasks are necessary to make the kernel more **interactive** and functional. This kernel has the potential to do much more than just display a message; it can **handle input**, **manage memory**, and **respond to events**.
2. **GDT (Global Descriptor Table):**
   * The **GDT** is like a **map** for the kernel. It tells the kernel where it can safely access memory and how to interact with other parts of the system.
   * Without the GDT, the kernel wouldn't know where to look for resources. It’s like driving without a **GPS system** — you might get lost.
   * Setting up the **GDT** in the Meaty Skeleton Kernel was one of the critical steps to make sure the kernel could work efficiently and without crashing.
3. **Interrupts and Event Handling:**
   * Interrupts are like **signals** or **notifications** to the kernel when something happens. For example, when you press a key on your keyboard or move your mouse, the kernel needs to know about it. That’s what interrupts help with.
   * Setting up **interrupt handling** in this kernel makes it more **interactive** and ready to handle real-world inputs. It’s like setting up a **notification system** so the kernel can react to events.
4. **How It Works:**
   * The Meaty Skeleton Kernel begins by setting up the **GDT** and **interrupts**.
   * Once these are set, the kernel can safely handle tasks and respond to events. Finally, the kernel prints **“Hello, Meaty Skeleton Kernel!”** to the screen, showing that it’s ready to do more than just greet the user.
5. **Comparison:**
   * While the Bare Bones kernel simply prints a message, the Meaty Skeleton kernel **prepares the system** before showing the message. It’s like organizing a desk before you start working. The Meaty Skeleton Kernel isn’t just functional — it’s organized and ready to take on more tasks. (Meaty skeleton is sooo not us students! 😊lol)

**Approach to Development**

The journey of creating these two kernels involved several steps, each teaching us something new about how operating systems work. The process was both frustrating at times and exciting, especially when things finally clicked into place. Here's a breakdown of the approach:

1. **Setting Up the Environment:**
   * The first step was making sure we had the right tools to **build the kernel**. This included installing **GCC** (the C compiler), **NASM** (the assembler), and **GRUB** (the bootloader).
   * Setting up the tools felt like preparing for a big project (more like a WAR) — you can’t build a house without the right tools, and you can’t build a kernel without the right software.
2. **Writing the Code:**
   * We wrote the kernel code using **C** for the higher-level logic and **Assembly** for low-level operations like setting up the GDT.
   * Writing this code was similar to learning a new language. Each function we wrote had to be carefully crafted so that it would run properly on the machine.
3. **Creating the Multiboot Header:**
   * The **multiboot header** is a key component that tells the **GRUB bootloader** how to load the kernel. It’s like the **invitation** that GRUB needs to get the kernel into memory and ready to run.
   * This step was confusing at first, but once we understood how it worked, it was like unlocking a door that led to the rest of the process. (after like 1000 hours)
4. **Compiling and Linking:**
   * The next step involved **compiling** the code using GCC and **assembling** it with NASM. The final output had to be linked together using **LD** (the linker) to create the final **kernel binary**.
   * Compiling and linking were like **mixing ingredients** and then baking them into something solid. It’s where the magic happens. (we were cooked btw)
5. **Creating the Bootable ISO:**
   * After everything was compiled and linked, we used **GRUB’s grub-mkrescue** tool to create the **bootable ISO**.
   * This step was a bit like burning a **bootable CD**, except it was all in a digital ISO file format. The ISO file contained everything necessary to load the kernel into memory and run it on a system.

**Challenges Faced**

1. **The OS Hopping Saga:**

* Initially, we tried this project on **MacBook**, thinking we could get away with it. Spoiler alert: we couldn’t. Then we shifted to **Windows with WSL**, but that wasn’t smooth either. Finally, we went all-in and installed a **dual-boot Linux OS** on the Windows laptop. Just when we thought we were unstoppable, we managed to **crash the Linux OS within two hours**. It was chaotic, funny, and exhilarating at the same time. Picture a team of engineers crashing their car right after assembling it — yep, that was us with Linux!

1. **The Infinite Loop of Broken Packages:**

* Ah**, broken packages** — the bane of our existence! We were stuck in a seemingly endless cycle of “held broken packages”, trying every fix the internet could throw at us. And let’s not even get started on the **make** and **make clean** commands. It felt like we were stuck in a time loop, running those commands over and over, hoping something would magically work. By the end, “make clean” became our mantra — the answer to all our problems and frustrations.

1. **Multiboot Header Issues:**
   * Understanding how the **multiboot header** worked was one of the toughest parts. At first, we didn’t realize how important it was, but without it, the kernel wouldn’t even load. It was like trying to **make tea and then halfway through realizing u don’t have tea leaves**.
2. **Configuring GRUB:**
   * Setting up **GRUB** was tricky. We had to make sure it was configured properly to point to the correct kernel file.
3. **Kernel Not Booting:**
   * There were several times when the **kernel wouldn’t boot** correctly. We spent a lot of time troubleshooting to figure out why. Debugging this felt like trying to **fix a broken engine**. Sometimes, the issue was small, like a typo in the code, and other times, it was more complex.

**Learnings and Observations**

1. **Operating Systems at a Low Level:**

* One of the biggest lessons was understanding how **kernels** work at a **low level**. The kernel interacts directly with the hardware to manage resources like memory, CPU, and devices.

1. **GRUB and Bootloaders:**
   * We learned how essential a **bootloader** is for loading the kernel. Without GRUB, the kernel wouldn’t be able to boot properly. This showed us how **critical** the first part of the boot process is.
2. **Interrupts and Memory Management:**
   * Setting up **interrupts** and the **Global Descriptor Table (GDT)** was crucial for the Meaty Skeleton Kernel to function correctly. These concepts allowed the kernel to manage memory and respond to hardware events.

**Conclusion**

This project was a great learning experience. While we started with the goal of creating one kernel, we ended up creating two: **Bare Bones** and **Meaty Skeleton**. The **Bare Bones Kernel** was simple and acted as a basic starting point, while the **Meaty Skeleton Kernel** added more complexity with interrupt handling and memory management. Through this process, we learned a lot about how **kernels** work, how **bootloaders** function, and how **low-level programming** interacts with hardware.

It wasn’t easy, but by the end of this project, we feel much more confident in our understanding of **operating systems** and **kernel development**. This project laid the foundation for us to explore more advanced concepts like handling user input, building file systems, and managing processes.

**Contributors:**

1. **Neha Reddy Putta**, Roll No: **SE22UCAM006**
2. **Prakriti Gupta**, Roll No: **SE22UCAM008**
3. **Sagarla Priyanka**, Roll No: **SE22UCAM009**
4. **Shriya Sengupta**, Roll No: **SE22UCAM010**
5. **Chirala Hema**, Roll No: **SE22UCAM013**