

**PROJECT REVIEW**

**OPERATING SYSTEM**

**TITLE: BUILDING A BASIC KERNEL**

**Slot:** F2

**Submitted By:**

Aaron Niju John (19BCE0616)

Anamaya Vyas (19BCE0568)

Avnish Tiwari (19BCE0634)

Vibhu Kumar Singh (19BCE0215)

**Submitted to:** Dr. Manikandan K

**TABLE OF CONTENTS:**

|  |  |  |
| --- | --- | --- |
| **Serial No** | **Topic** | **Page No** |
| **1.** | **Acknowledgement** |  |
| **2.** | **Abstract** |  |
| **3.** | **Introduction** |  |
| **4.** | **Literature Review and Related Works** |  |
| **5.** | **Existing Approach** |  |
| **6.** | **Proposed Approach** |  |
| **7.** | **Conclusion** |  |
| **8.** | **Future Works** |  |
| **9.** | **References** |  |

**ACKNOWLEDGEMENT:**

The euphoria and satisfaction of the completion of the project will be incomplete without thanking the people responsible for this venture. We would like to acknowledge our SCOPE school Dean for providing us an opportunity to create a mini-project in the 3rd semester. We would like to express our immense gratitude towards our project guide, Dr. Manikandan K, School of Computer Science and Engineering, without whom this project wouldn’t have been possible, for his constant encouragement, timely assistance, guidance and valuable feedback extended in the completion of this project.

**ABSTRACT:**

Operating Systems are large, highly complex pieces of software that are difficult to design and maintain in terms of functionality and security. An operating system acts as an intermediary between the user of a computer and the computer hardware. The purpose of an operating system is to provide an environment in which a user can execute programs in a convenient and efficient manner. The k[ernel](https://www.geeksforgeeks.org/kernel-i-o-subsystem-in-operating-system/) is central component of an operating system that manages operations of computer and hardware. It basically manages operations of memory and CPU time. It is core component of an operating system. Kernel acts as a bridge between applications and data processing performed at hardware level using inter-process communication and system calls. Even though there are many well-built and complex kernels in use today, our team wanted to develop and implement a new kernel so that our understanding on the complex development process and of kernels in general will increase. The idea behind this project is to build a very basic kernel with limited operations that mainly serve as a technology demonstrator rather than serve any practical use.

**KEYWORDS**: Kernel, core component, technology demonstrator

**INTRODUCTION:**

**What is a kernel?**

The kernel is a [computer program](https://en.wikipedia.org/wiki/Computer_program) at the core of a computer's [operating system](https://en.wikipedia.org/wiki/Operating_system) with complete control over everything in the system. It is an integral part of any operating system and serves a role similar to heart in a body. It is the "portion of the operating system code that is always resident in memory". It facilitates interactions between hardware and software components. On most systems, it is one of the first programs loaded on [startup](https://en.wikipedia.org/wiki/Booting) (after the [bootloader](https://en.wikipedia.org/wiki/Bootloader)). It handles the rest of startup as well as [input/output](https://en.wikipedia.org/wiki/Input/output) (I/O) requests from [software](https://en.wikipedia.org/wiki/Software), translating them into [data-processing](https://en.wikipedia.org/wiki/Data_processing) instructions for the [central processing unit](https://en.wikipedia.org/wiki/Central_processing_unit). It handles memory and [peripherals](https://en.wikipedia.org/wiki/Peripheral) like keyboards, monitors, printers, and speakers. The critical code of the kernel is usually loaded into a separate area of memory, which is protected from access by [application programs](https://en.wikipedia.org/wiki/Application_software) or other, less critical parts of the operating system. The kernel performs its tasks, such as running processes, managing hardware devices such as the [hard disk](https://en.wikipedia.org/wiki/Hard_disk), and handling interrupts, in this protected [kernel space](https://en.wikipedia.org/wiki/Kernel_space). In contrast, [application programs](https://en.wikipedia.org/wiki/Application_program) like browsers, word processors, or audio or video players use a separate area of memory, [user space](https://en.wikipedia.org/wiki/User_space). This separation prevents user data and kernel data from interfering with each other and causing instability and slowness, as well as preventing malfunctioning application programs from crashing the entire operating system. This central component of a computer system is responsible for 'running' or 'executing' programs. The kernel takes responsibility for deciding at any time which of the many running programs should be allocated to the processor or processors.

**What are the different kernel architecture designs present?**

There are different kernel architecture designs:

1. [Monolithic Kernel](https://www.geeksforgeeks.org/monolithic-kernel-and-key-differences-from-microkernel/) –  
It is one of types of kernel where all operating system services operate in kernel space. It has dependencies between systems components. It has huge lines of code which is complex. [Monolithic kernels](https://en.wikipedia.org/wiki/Monolithic_kernel) run entirely in a single [address space](https://en.wikipedia.org/wiki/Address_space) with the CPU executing in [supervisor mode](https://en.wikipedia.org/wiki/Protection_ring), mainly for speed.

Example :

Unix, Linux, Open VMS, XTS-400 etc.

2. [Micro Kernel](https://www.geeksforgeeks.org/microkernel-in-operating-systems/) –  
It is kernel types which has minimalist approach. It has virtual memory and thread scheduling. It is more stable with less services in kernel space. It puts rest in user space. [Microkernels](https://en.wikipedia.org/wiki/Microkernel) run most but not all of their services in user space, like user processes do, mainly for resilience and [modularity](https://en.wikipedia.org/wiki/Modular_programming).

3. Hybrid Kernel –  
It is the combination of both monolithic kernel and mircrokernel. It has speed and design of monolithic kernel and modularity and stability of microkernel.

4. Exo Kernel –  
It is the type of kernel which follows end-to-end principle. It has fewest hardware abstractions as possible. It allocates physical resources to applications.

5. Nano Kernel –  
It is the type of kernel that offers hardware abstraction but without system services. Micro Kernel also does not have system services therefore the Micro Kernel and Nano Kernel have become analogous

**PROBLEM DEFINITION:**

Kernels are the core component of an operating system. A good, efficient and functioning kernel satisfies the following objectives:

* To establish communication between user level application and hardware.
* To decide state of incoming processes.
* To control disk management.
* To control memory management.
* To control task management.

Kernel development is a complex process. Building an advanced kernel usually takes hundreds of thousands to millions of lines of code to develop to be suitable for production environments. For example, the linux kernel’s source code when deemed to be suitable for production environments, had around 176 thousand lines of source code and currently contains about 27.8 million lines of code. While we are unable to build a kernel at such a scale with our limited knowledge on the development of kernels, we wanted to try to build a basic kernel with basic functions to understand the complex and tedious process behind kernel development.

The aim for our project is to build a kernel which can be loaded using a GRUB bootloader on an x86 system and also include a keyboard driver that can print the characters a-z and numbers 0-9 on the screen.

**NEED:**

While many kernels exist, such as the Linux kernel, the Windows kernel, etc, our team wanted to develop and implement a basic kernel with limited functions to understand the difficulty in developing a kernel with complex operations to be used on a large scale such as those mentioned above. Rather than build upon existing kernel infrastructure, we wanted to build a kernel from scratch. We hope that by taking on this project, we get a glimpse of the vast, incredible and complex field of kernel development and help us to expand our knowledge on the topic.

**USE:**

The basic kernel our team has developed is mainly developed with the intention to serve as a technology demonstrator and doesn’t really serve any practical use. This kernel that has been built gives us insight into the complex process of kernel development. We hope that with more time and dedication, we can develop a more advanced kernel which performs more complex operations.

**LITERATURE REVIEW AND RELATED WORKS:**

In this modern day and age, the world has undergone a digital transformation and computer science is one of the main reasons behind it by providing a foundation for the emerging technologies. Operating Systems are one of the essential and fundamental blocks of Computer Science. A sound and extensive knowledge of this topic provides the base to be a good computer scientist and can be applied to solve real-world problems by creating projects that use them efficiently. As computer science students, we wanted to create a project which focuses on how a basic kernel is built to understand the intricate details behind kernel development.

**1) Linux Kernel Development – Ian McDonnald – Fifth New Zealand Computer Science Research Student Conference, 2007**

Problem Definition:

This research paper gives us insight into the world of Linux kernel development and gives us an overall glimpse of the process behind Linux kernel development from the social aspects to building the kernel to repeated tested to finally releasing the code. From the social aspect, the author describes the difficulty in getting proposed changes accepted and how parties reviewing the proposed changes can be a bit hostile but will accept the changes if the idea is correct. The author then goes on to describe the his approach to the process of kernel development.

Proposed Solution:

The first step is maintaining a source code tree, keeping in sync with the Linux kernel source code tree which is stored as a git tree if it is intended to have the code merged into the tree or released as an ongoing codebase. The author further describes various commands and tools that help create a clone of the Linux source code tree to which the proposed changes can be applied. The author discusses some caveats that are extremely useful to new kernel developers as well as useful resources that can help give a deeper introduction to Linux kernel development.

The author also highlights importance of ensuring code is well tested before it is released and discusses a number of virtualisation tools available to assist with testing software such as qemu, UML, VMWare, etc and how these tools can make testing easier if computers with sufficient processing power are used as this can uncover separate bugs.

For the last step, the author highlights the importance of releasing code early and often and how it can be productive and useful as it gives other developers an opportunity to view your code and suggest possible enhancements which is the foundational idea behind the open source community.

Advantage:

This research paper gives a complete overview on the various steps involved in the Linux kernel development and gives new kernel developers tips, tricks and caveats as well as commands and tools that can prove to be extremely useful in the development process.

Limitations:

While this research paper gives a complete overview on the various steps of Linux kernel development, it doesn’t give an in-depth analysis and explanation of each step.

## **2)Research of an architecture of operating system kernel based on modularity concept - Hongbiao Jiang, Wanlin Gao, Manwei Wang, Simon See, Ying Yang, Wei Liu, Jin Wang – ELSEVIER, Mathematical and Computer Modelling,** [**Volume 51, Issues 11–12**](https://www.sciencedirect.com/science/journal/08957177/51/11)**, June 2010, Pages 1421-1427**

**Problem Definition:**

This research paper firstly highlights the difference between the Linux kernel, which focus on executing efficiency and is a monolithic kernel operating system, and Windows family which pays more attention to convenient use and is a microkernel operating system. The paper further highlights the existing problems of the classical kernel architecture such as the kernel monopolizing the rights of assigning all the system resources as well as holding the actual executing rights and safety monitoring due to only one kernel being present in the operating system and also states that if an error occurs in the kernel level, the whole system would crash as well as the possibility of some applications making use of some mechanisms in the operating system to capture the CPU time to execute all kinds of malicious or destructive operations, such as tampering data, monopolizing resources, even causing the system crash or user data missing.

Proposed Solution:

According to the authors, to fully utilize the system resource, the operating system kernel’s monopolization of the system resources must be changed, and the kernel error or bugs must be decreased and not causing the system crash. The author presents a new architecture of operating system kernel which is based on modularity concept and takes into consideration the shortcomings of a traditional kernel. The kernel is divided into three independent modules according to their functions — executing module, policy module and monitoring module. The policy module determines the policy of process scheduling separated from traditional kernel. Monitoring module is responsible for monitoring processes, handling system error and detecting of important data. These three modules work independently and communicate with each other by interrupts to ensure the safety of CPU time, important system data structures and user data. The authors further explain each module in detail and define data structures that can be implemented to execute each module to complete the development of this proposed architecture.

**Advantage:**

The three proposed independent modules help improve the expandability and simplify maintenance of the operating system kernel and enhance the robustness of the system This architecture has modularity to confirm the concept of modularity programming which is convenient for design, extension and implement of system. In addition, the communication interface between modules is simple and unified because it is implemented by timer interrupt and addresses structure. Moreover, this architecture intimately concerns with safety and protection of user data.

**Limitations:**

The main potential problem of the proposed architecture is the executing efficiency of the system because of the extra communication overhead of three kernel modules.

There is also scope for possible improvements which include adding debugging to error handling in the monitoring module which enable the programs to be debugged besides being terminated or restarted.

To conclude, these scholarly articles give us an idea and provide insight on the structure and design of an operating system kernel, the various requirements, technologies used as well as various ways of implementing it which can be applied and used in our project

**EXISTING APPROACH:**

**PROPOSED APPROACH:**

**DESIGN:**

**The first part is seeing how the keyboard will communicate with the kernel.**

For this we will use the I/O ports. These ports are addresses on the I/O bus.

We will use the read\_port and the write\_port for communicating.

Write\_port places a message at the tail of the port's message queue and gets blocked when the queue gets full and it returns back when room is made in the queue by read\_port.

Read\_port removes the message at the head of the queue and returns it to the caller and it also gets blocked when the queue becomes empty and returns back when write\_port is called.

**The second part is how to tell the processor that the device has performed the event.**

For this there are 2 methods available-polling and interrupt.

Polling keeps checking the status of the device forever while interrupt sends a signal to the processor from the hardware indicating an event.

Using interrupt is a more practical approach.

**The third part is to write the device driver for the keyboard**

A device or a chip called Programmable Interrupt Controller (PIC) is responsible for x86 being an interrupt driven architecture. It manages hardware interrupts and sends them to the appropriate system interrupt.

When the actions are performed on the keyboard it sends a pulse called Interrupt Request (IRQ) along its specific interrupt line to the PIC chip.For the keyboard the line is IRQ1.

The PIC then translates the received IRQ into a system interrupt, and sends a message to interrupt the CPU from whatever it is doing. It is then the kernel’s job to handle these interrupts.

For the Keyboard, it works through the I/O ports 0x60 and 0x64. Port 0x60 gives the data (pressed key) and port 0x64 gives the status. However, you have to know exactly when to read these ports.

When a key is pressed, the keyboard gives a signal to the PIC along its interrupt line IRQ1. The processor looks up a certain data structure called the Interrupt Descriptor Table (IDT) to give the interrupt handler address corresponding to the interrupt number.

After this we can map keyboard interrupts with keyboard handler via IDT

**The Fourth Part is building the kernel-**

For this we need-

* Linux
* NASM assembler
* gcc
* ld (GNU Linker)
* grub

We will write a few lines of assembly code that serves as a starting point for the kernel.

Our assembly file will invoke an external function which we will write in C, and then halt the program flow.

ld is the linker file which will link the kernel.c file with the assembly code.

Now the only part left is writing the kernel file, which will be written in C language.

The **last** thing is to boot the kernel which will be done using a boot loader called Multiboot Specification.

Now just build the kernel, configure and run the kernel.

**Implementation/Code:**

**kernel.c file**

#include "keyboard\_map.h"

#define LINES 25

#define COLUMNS\_IN\_LINE 80

#define BYTES\_FOR\_EACH\_ELEMENT 2

#define SCREENSIZE BYTES\_FOR\_EACH\_ELEMENT \* COLUMNS\_IN\_LINE \* LINES

#define KEYBOARD\_DATA\_PORT 0x60

#define KEYBOARD\_STATUS\_PORT 0x64

#define IDT\_SIZE 256

#define INTERRUPT\_GATE 0x8e

#define KERNEL\_CODE\_SEGMENT\_OFFSET 0x08

#define ENTER\_KEY\_CODE 0x1C

extern unsigned char keyboard\_map[128];

extern void keyboard\_handler(void);

extern char read\_port(unsigned short port);

extern void write\_port(unsigned short port, unsigned char data);

extern void load\_idt(unsigned long \*idt\_ptr);

unsigned int current\_loc = 0;

char \*vidptr = (char\*)0xb8000;

struct IDT\_entry {

unsigned short int offset\_lowerbits;

unsigned short int selector;

unsigned char zero;

unsigned char type\_attr;

unsigned short int offset\_higherbits;

};

struct IDT\_entry IDT[IDT\_SIZE];

void idt\_init(void)

{

unsigned long keyboard\_address;

unsigned long idt\_address;

unsigned long idt\_ptr[2];

keyboard\_address = (unsigned long)keyboard\_handler;

IDT[0x21].offset\_lowerbits = keyboard\_address & 0xffff;

IDT[0x21].selector = KERNEL\_CODE\_SEGMENT\_OFFSET;

IDT[0x21].zero = 0;

IDT[0x21].type\_attr = INTERRUPT\_GATE;

IDT[0x21].offset\_higherbits = (keyboard\_address & 0xffff0000) >> 16;

idt\_address = (unsigned long)IDT ;

idt\_ptr[0] = (sizeof (struct IDT\_entry) \* IDT\_SIZE) + ((idt\_address & 0xffff) << 16);

idt\_ptr[1] = idt\_address >> 16 ;

load\_idt(idt\_ptr);

}

void kb\_init(void)

{

write\_port(0x21 , 0xFD);

}

void kprint(const char \*str)

{

unsigned int i = 0;

while (str[i] != '\0') {

vidptr[current\_loc++] = str[i++];

vidptr[current\_loc++] = 0x07;

}

}

void kprint\_newline(void)

{

unsigned int line\_size = BYTES\_FOR\_EACH\_ELEMENT \* COLUMNS\_IN\_LINE;

current\_loc = current\_loc + (line\_size - current\_loc % (line\_size));

}

void clear\_screen(void)

{

unsigned int i = 0;

while (i < SCREENSIZE) {

vidptr[i++] = ' ';

vidptr[i++] = 0x07;

}

}

void keyboard\_handler\_main(void)

{

unsigned char status;

char keycode;

write\_port(0x20, 0x20);

status = read\_port(KEYBOARD\_STATUS\_PORT);

if (status & 0x01) {

keycode = read\_port(KEYBOARD\_DATA\_PORT);

if(keycode < 0)

return;

if(keycode == ENTER\_KEY\_CODE) {

kprint\_newline();

return;

}

vidptr[current\_loc++] = keyboard\_map[(unsigned char) keycode];

vidptr[current\_loc++] = 0x07;

}

}

void kmain(void)

{

const char \*str = "Our first kernel";

clear\_screen();

kprint(str);

kprint\_newline();

kprint\_newline();

idt\_init();

kb\_init();

while(1);

}

**Kernel.asm File**

bits 32

section .text

;multiboot spec

align 4

dd 0x1BADB002 ;magic

dd 0x00 ;flags

dd - (0x1BADB002 + 0x00) ;checksum. m+f+c should be zero

global start

global keyboard\_handler

global read\_port

global write\_port

global load\_idt

extern kmain ;this is defined in the c file

extern keyboard\_handler\_main

read\_port:

mov edx, [esp + 4]

;al is the lower 8 bits of eax

in al, dx ;dx is the lower 16 bits of edx

ret

write\_port:

mov edx, [esp + 4]

mov al, [esp + 4 + 4]

out dx, al

ret

load\_idt:

mov edx, [esp + 4]

lidt [edx]

sti ;turn on interrupts

ret

keyboard\_handler:

call keyboard\_handler\_main

iretd

start:

cli ;block interrupts

mov esp, stack\_space

call kmain

hlt ;halt the CPU

section .bss

resb 8192; 8KB for stack

stack\_space:

**link.ld File**

OUTPUT\_FORMAT(elf32-i386)

ENTRY(start)

SECTIONS

{

. = 0x100000;

.text : { \*(.text) }

.data : { \*(.data) }

.bss : { \*(.bss) }

}

**keyboard\_map.h File**

unsigned char keyboard\_map[128] =

{

0, 27, '1', '2', '3', '4', '5', '6', '7', '8', /\* 9 \*/

'9', '0', '-', '=', '\b', /\* Backspace \*/

'\t', /\* Tab \*/

'q', 'w', 'e', 'r', /\* 19 \*/

't', 'y', 'u', 'i', 'o', 'p', '[', ']', '\n', /\* Enter key \*/

0, /\* 29 - Control \*/

'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l', ';', /\* 39 \*/

'\'', '`', 0, /\* Left shift \*/

'[\\](about:blank)', 'z', 'x', 'c', 'v', 'b', 'n', /\* 49 \*/

'm', ',', '.', '/', 0, /\* Right shift \*/

'\*',

0, /\* Alt \*/

' ', /\* Space bar \*/

0, /\* Caps lock \*/

0, /\* 59 - F1 key ... > \*/

0, 0, 0, 0, 0, 0, 0, 0,

0, /\* < ... F10 \*/

0, /\* 69 - Num lock\*/

0, /\* Scroll Lock \*/

0, /\* Home key \*/

0, /\* Up Arrow \*/

0, /\* Page Up \*/

'-',

0, /\* Left Arrow \*/

0,

0, /\* Right Arrow \*/

'+',

0, /\* 79 - End key\*/

0, /\* Down Arrow \*/

0, /\* Page Down \*/

0, /\* Insert Key \*/

0, /\* Delete Key \*/

0, 0, 0,

0, /\* F11 Key \*/

0, /\* F12 Key \*/

0, /\* All other keys are undefined \*/

};

**RESULT ANALYSIS:**

We first build our kernel using the commands in the terminal

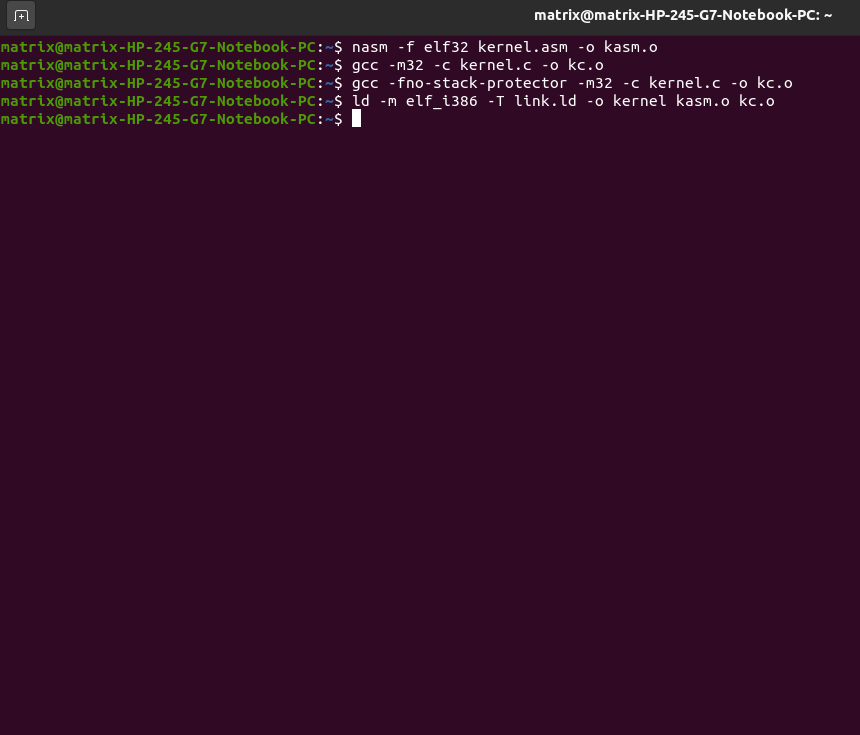
The commands are as follows-

**nasm -f elf32 kernel.asm -o kasm.o**

**gcc -m32 -c kernel.c -o kc.o**

**gcc -fno-stack-protector -m32 -c kernel.c -o kc.o**

**ld -m elf\_i386 -T link.ld -o kernel kasm.o kc.o**

****

After building the kernel we run it on the QEMU emulator with the following command

**qemu-system-i386 -kernel kernel**

****

**CONCLUSION:**

With this project, we have successfully completed the development and implementation of a basic kernelwhich can be loaded using a GRUB bootloader on an x86 system and also include a keyboard driver that can print the characters a-z and numbers 0-9 on the screen.

**FUTURE WORKS:**

The kernel we have built is a very basic one with its operations limited to printing alphabets and numbers from a keyboard. While this kernel may not have much practical use or solve any real-world problem and was mainly designed to serve as a technology demonstrator, this kernel can serve as the foundation for building a more advanced kernel with more complex operations and the experience and knowledge gained during this process will help us in building a more complex kernel.

**REFERENCES:**

**1)**

**2)**

**3)**

**4)**

**5)**

**6)**

**7)**

**8)**

**9)**

**10)**

13. Love, R.: Linux Kernel Development. Second edn. Novell Press (2005)

14. Web: Lxr. http://lxr.linux.no/ (Accessed 2006)