Amity Journal of Computational Sciences Volume 6 Issue 2, 2022

ISSN 2456-6616 (Online)

DEVELOPMENT OF KERNEL FOR NEW OPERATING SYSTEM

Ananya Vishnoi
Computer Science and Engineering, ASET
Amity University, India
ananyavishnoi03@gmail.com

Gitansh Agrawal
Computer Science and Engineering, ASET
Amity University, India
agitansh123@gmail.com

Muteen Nabi Kundangar
Computer Science and Engineering, ASET
Amity University, India
muteen.kundangar@s.amity.edu

Omnath Dixit
Computer Science and Engineering, ASET
Amity University, India
omnathdixit7033@gmail.com

Abstract- This project aims to explore the fundamentals of kernel development for an operating system. By embarking on this project, we gained a deeper understanding of operating systems concepts, low-level programming, and computer architecture. The project focuses on implementing key kernel functionalities. This involves designing data structures, algorithms, and system call interfaces. Proper documentation of the project's design decisions, implementation details, and lessons learned is crucial. We have created comprehensive documentation and prepare a presentation to share our insights and experiences with others. At the end of this project, we have developed a basic kernel for an operating system with essential functionalities. The project's documentation and presentation serve as valuable resources for future reference and sharing knowledge with others interested in kernel development.

Keywords: Operating System, Kernel, Bootloader,

I INTRODUCTION

The kernel is the core component responsible for managing hardware resources, providing system services, and facilitating communication between software and hardware.

We delve into the principles and concepts underlying operating systems, such as process management, memory management, file systems, and device drivers. Kernel development requires expertise in low-level programming languages such as C and assembly. To develop a kernel, we need to comprehend the underlying computer architecture. This project will provide an opportunity to explore processor-specific features, memory organization, interrupt handling, and other architectural aspects crucial to kernel development.

Testing and debugging are critical aspects of kernel development. We have employed various techniques and tools to ensure the stability, performance, and correctness of the kernel code. This may involve using debuggers, emulators, and writing test cases to validate the implemented functionality.

II OPERATING SYSTEM

Operating system is a low-level and most important system software that acts as an interface between the user and the computer hardware. It is a program that is initially loaded into the computer by a boot program and then it controls the execution and management of all other system programs and resources respectively. The system programs make use of operating system by making requests for services through a defined Application Program Interface (API). The users interact with the operating system through a user interface such as Command-line Interface (CLI) or a Graphical User Interface (GUI).[5]

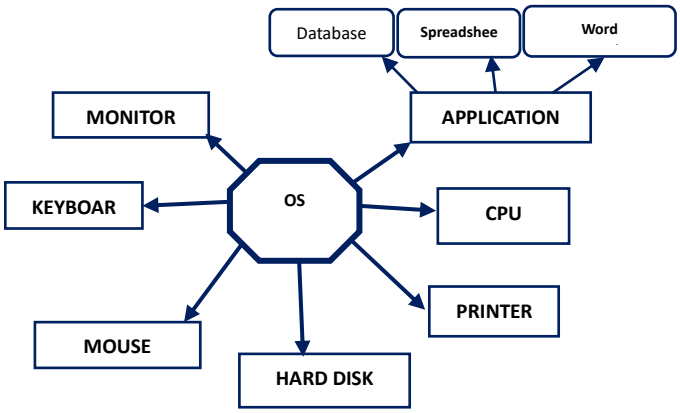


Fig-1 Operating System

III ARCHITECTURE OF OPERATING SYSTEM

An operating system comprises of four fundamental elements which are discussed in detail below:[6,7]

Application

The term "application" refers to the programme that a user makes use of to operate their operating system. a few examples of applications include Slack, the Sublime Text editor, and others.

ISSN 2456-6616 (Online)

Shell
It is an interface that helps the user to communicate with the operating system by typing commands or executing scripts. The operating system provides users process management, memory management, and file systems, among others.

Kernel
The operating system's kernel is the most important and core component which handles the resource management by providing application processes the required I/O, CPU, and memory via system calls and inter-process communication techniques. It is also responsible for ensuring that the system is stable and secure.

Hardware
The hardware refers to the physical parts of a digital computer. The components encompassed within it consist of memory, CPU, Arithmetic Logic Unit (ALU) etc.

IV MODES IN OPERATING SYSTEM

In an operating system, user mode and kernel mode are two distinct execution states that determine the level of privileges and access rights available to a program or process.

User Mode
User mode is the execution state in which most applications or user-level processes run. Programs running in user mode have restricted privileges and access to system resources. They cannot directly access hardware or perform privileged operations. In user mode, processes can execute instructions, access their own memory space, and perform basic operations like reading from or writing to files. This separation ensures that a malfunctioning or malicious program cannot disrupt the stability or security of the entire system.

Kernel Mode
Kernel mode is the execution state in which the operating system's kernel operates. Kernel mode allows the operating system to perform critical operations, manage hardware resources, control processes, and enforce security policies. The transition from user mode to kernel mode is a controlled mechanism, and only trusted and privileged code can execute in kernel mode. This ensures that the kernel remains protected from arbitrary or unauthorized access. It helps maintain system integrity, protects against unauthorized access, and facilitates controlled access to system resources through system calls.

V SYSTEM CALL

A system call is a mechanism provided by the operating system that allows user-level processes or programs to request services from the kernel, which is the core component of the operating system. It acts as an interface between the user-level applications and the operating system and is the only way to enter in kernel mode.

VI KERNEL ARCHITECTURE

There are three types of kernel architecture which are discussed as the following.

Monolithic Kernel
In a monolithic kernel, kernel services and user services are carried out in the same memory region. However, because identical address locations are utilised by the same address processes, the monolithic kernel has a propensity to produce more defects and failures. Identical address increases the size of the kernel which further increases the size of the operating system. Also, monolithic features must be completely rewritten, making the process of adding or removing them more laborious.[3]

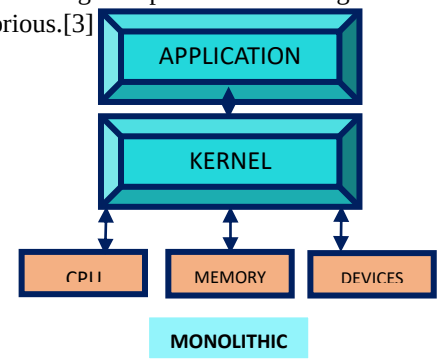
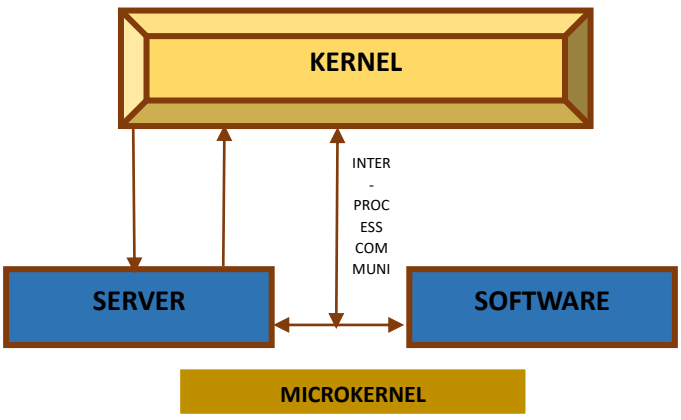


Fig-2 Architecture of Monolithic kernel

Microkernel
A microkernel is distinct from a conventional kernel or Monolithic Kernel. This uses separate address spaces for user space and kernel space to implement user services and kernel services. As it utilises separate locations for the two services and performs minimal operations, the size of the microkernel is less, which also results in a smaller size of OS. The compact nature of the kernel contributes to enhancing the stability and security of the operating system.

In comparison to monolithic kernels, microkernels are simpler to administer and maintain. However, if there are more system calls and context switches, the system's performance may suffer due to its slowness.[8]



ISSN 2456-6616 (Online)

Fig-3 Architecture of Microkernel

Hybrid Kernel

Monolithic and Microkernels are combined to create hybrid kernels, which are also referred to as modular kernels. It makes use of the modular structure of microkernels and the speed of monolithic kernels. Hybrid kernels are widely used in commercial operating systems. This type of kernel in addition to processing processes similar to microkernel also operates the device drivers and the application IPC in kernel mode.[3,8]

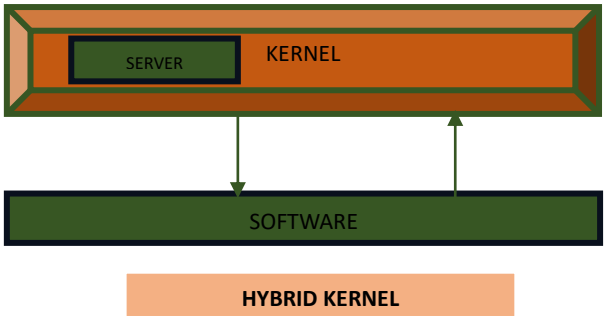


Fig-4 Architecture of Hybrid kernel

VII BOOT LOADER

The operating system of a digital system is loaded into the main memory with the aid of a program which is known as a boot loader. It is also referred as boot manager. When a digital system is started or restarted some tasks are initially performed by the Basic Input/Output System (BIOS) then it directs the administration to the Master Boot Record (MBR). The boot loader is present in the MBR. The majority of brand-new computers come pre-loaded with boot loaders for either a Mac OS or a version of Microsoft Windows. An exclusive boot loader must be installed on a computer before it can be used with Linux.

Working of a Boot Loader

When we push the system's power button, the hardware is initialised, and we may view several bits of information about the hardware. The system's primary memory won't contain any data since it is volatile. The first set of instructions, known as the BIOS, are retrieved from ROM (Read Only Memory), a non-volatile memory, to begin the execution.

A little programme called BIOS searches the system for bootable media including CDs, floppy discs, and hard drives. After locating the bootable device, the bootloader's Master Boot Record (MBR) is sought after by the BIOS. If the MBR is discovered, BIOS will launch the bootloader that is stored there. The bootloader now takes over the CPU and loads the operating system into our machine's main memory. After being loaded into main memory, the operating system takes control of the whole system.[6]

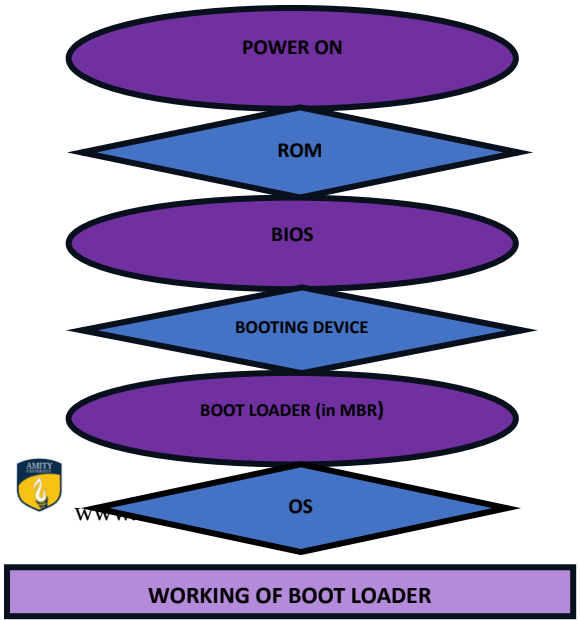


Fig-5 Working of a bootloader

VIII WRITING THE BOOT LOADER

IX WRITING THE BOOTLOADER

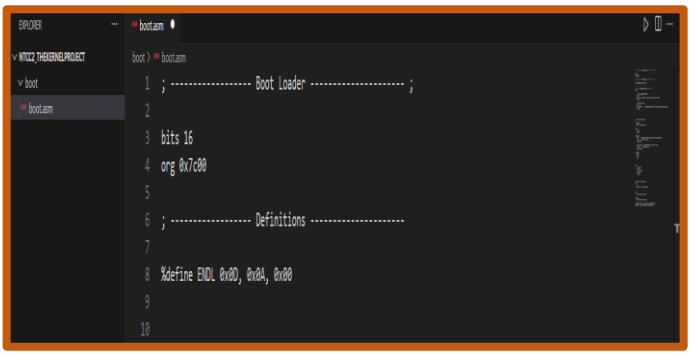
The Master Boot Record (MBR) is defined in the boot loader’s main assembly file. This file also contains the statements which are required for all other required supporting modules.[9]

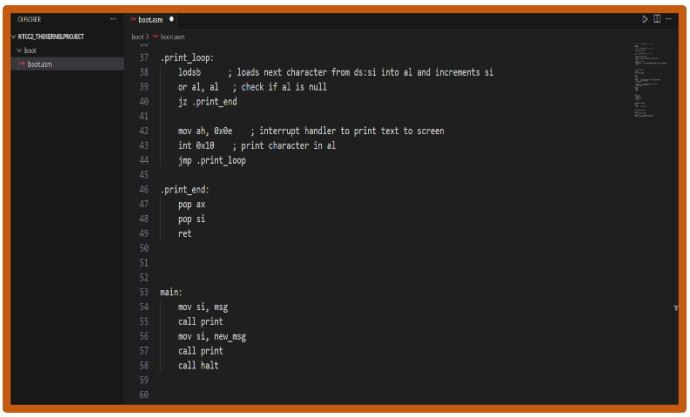
boot.asm file

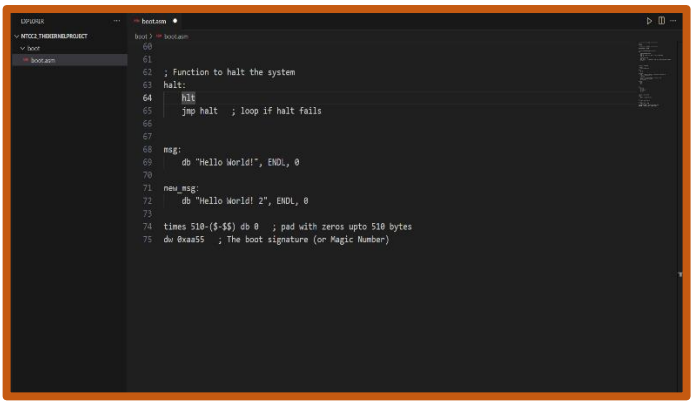
This is the main assembly file for the boot loader which contains the definition of the master boot record and also all the include statements for all the relevant helper modules.

The snippet of the code in the **boot.asm** file is shown below.

ISSN 2456-6616 (Online)







Boot loader works in 16-bit real mode. Due to this fact, it is required for the assembler to comprehend that it must generate instructions for 16-bit mode. This can be done using the **bits 16** command.

X SWITCHING BETWEEN 16-BIT AND 32-BIT

The transition between 16-bit and 32-bit modes while developing a boot loader in assembly language refers to altering the processor's operating mode throughout the boot process. This process is mentioned as below:

16-bit mode

Real mode is a compatibility mode designed to support older software and operating systems. When the booting process starts, the processor is in 16-bit real mode. The processor can only access a maximum of 1 MB of memory directly in the real mode.

The code written for the boot loader functions in a modular memory model in the real mode. We have to work with limited computational power and memory addressing capabilities because registers and instructions are limited to 16 bits in size.

Switching to 32-bit mode

32-bit mode is the protected mode which offers many advantages over the real mode such as more memory access, enhanced memory protection, and support for modern operating systems. When the boot loader completes the initial task it becomes ready to load the operating system. To switch to 32-bit protected mode, the boot loader performs the following steps:

Disable interrupts: Before transitioning to protected mode, interrupts must be disabled to avoid any unexpected interruptions during the mode switch. Setting up a Global Descriptor Table (GDT): The boot loader must initialize the GDT with appropriate segment descriptors as the GDT is a data structure that defines memory segments and access rights in protected mode.

Loading the address of the GDT: As the processor needs to know the location of the GDT, the address of the GDT is loaded into the GDTR (Global Descriptor Table Register) by the boot loader using the LGDT (Load Global Descriptor Table) instruction.

Enable protected mode: Finally, using the MOV and OR instructions the boot loader switches the processor into protected mode by setting the appropriate bit in the control register CR0.

Operating in 32-bit Mode

When the boot loader is in 32-bit protected mode, it can take advantage of the extended capabilities of the processor. It can access larger amounts of memory, use 32-bit instructions and registers, and utilize protected mode features like paging and virtual memory. The boot loader can then proceed to load the operating system into memory and transfer control to the operating system's entry point by performing a call instruction.

Global Descriptor Table (GDT)

Memory segmentation operates somewhat differently once 16-bit real mode is exited. Segment descriptors, which are a component of the GDT, are used to describe memory segments in protected mode.

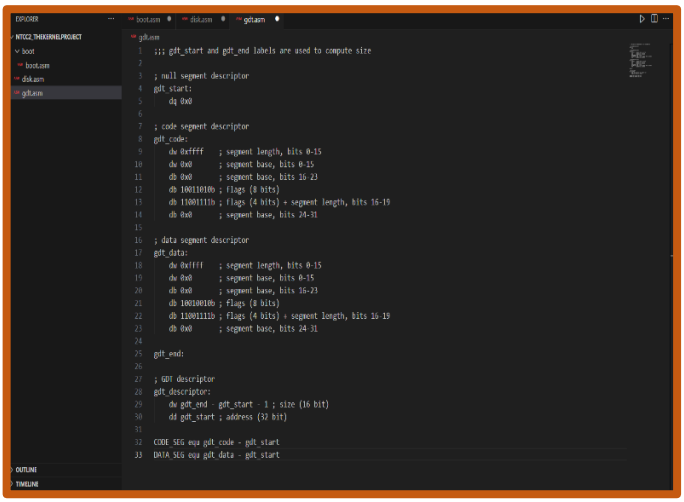
Following is a breakdown of our GDT:

ISSN 2456-6616 (Online)

- An eight 0-byte null segment descriptor. This is necessary as a safety measure to notice mistakes where our code forgets to choose a memory segment, leading to the default selection of an erroneous segment.
- The descriptor for the 4 GB code segment.
- The descriptor for the 4 GB data section.

A data structure i.e., segment descriptor includes the Base address, Segment limit, G (granularity), VL (available), DPL (descriptor privilege level), R (readable), W (writable), A (accessed). A GDT descriptor must be put up in addition to the GDT itself. The GDT location (memory address) and size are both contained in the descriptor.[9]

Below is the code snippet of the file **gdt.asm**:

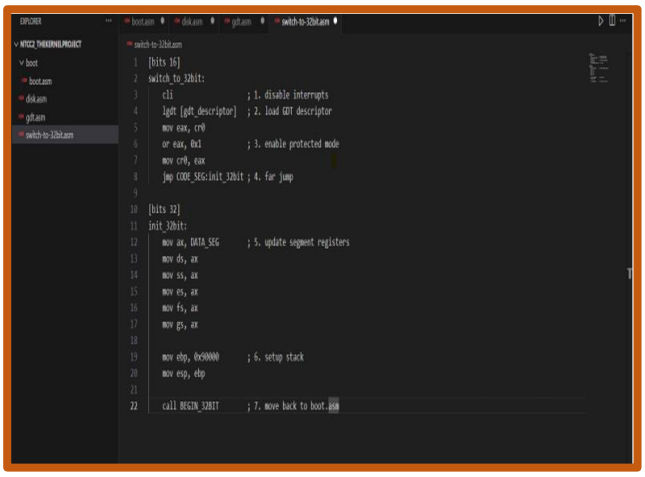


```
;;; gdt_start and gdt_end labels are used to compute size

; null segment descriptor
gdt_start:
    dq 0x0

; code segment descriptor
gdt_code:
    dw 0xffff    ; segment length, bits 0-15
    dw 0x0       ; segment base, bits 0-15
    db 0x0       ; segment base, bits 16-23
    db 10011010b ; flags (8 bits)
    db 11001111b ; flags (4 bits) + segment length, bits 16-19
    db 0x0       ; segment base, bits 24-31

; data segment descriptor
gdt_data:
    dw 0xffff    ; segment length, bits 0-15
    dw 0x0       ; segment base, bits 0-15
    db 0x0       ; segment base, bits 16-23
    db 10010010b ; flags (8 bits)
    db 11001111b ; flags (4 bits) + segment length, bits 16-19
    db 0x0       ; segment base, bits 24-31

gdt_end:

; GDT descriptor
gdt_descriptor:
    dw gdt_end - gdt_start - 1 ; size (16 bit)
    dd gdt_start ; address (32 bit)

CODE_SEG equ gdt_code - gdt_start
DATA_SEG equ gdt_data - gdt_start
```

The following actions must be taken in order to enter 32-bit protected mode so that our 32-bit kernel may take over:

- Use the cli instruction to turn off interruptions.
- Use the ‘lgdt’ instruction to load the GDT descriptor into the GDT register.
- In the control register cr0, enable protected mode.
- Using ‘jmp’, far jump into our section of code. This requires a long jump to clear the CPU pipeline and remove any remaining prefetched 16-bit instructions.
- Set up our single 4 GB data segment as the destination for all segment registers (ds, ss, es, fs, and gs).
- Set the 32-bit stack pointer (esp) and bottom pointer (ebp) to create a new stack.
- Return to mbr.asm and call our 32-bit kernel entry procedure to give the kernel command.

The code snippet for the file named **switch-to-32bit.asm** is displayed below.



```
[bits 16]
switch_to_32bit:
    cli                     ; 1. disable interrupts
    lgdt [gdt_descriptor]   ; 2. load GDT descriptor
    mov eax, cr0
    or eax, 0x1             ; 3. enable protected mode
    mov cr0, eax
    jmp CODE_SEG:init_32bit ; 4. far jump

[bits 32]
init_32bit:
    mov ax, DATA_SEG        ; 5. update segment registers
    mov ds, ax
    mov ss, ax
    mov es, ax
    mov fs, ax
    mov gs, ax

    mov ebp, 0x90000        ; 6. setup stack
    mov esp, ebp

    call BEGIN_32BIT        ; 7. move back to boot.asm
```

After changing the mode, we are prepared to give/pass on to our kernel full control.

XI PUTTING EVERYTHING TOGETHER

Some tooling is required in order to build our operating system image. To process our assembly files, we require **NASM**. To compile our C code, we require **gcc**. To create a bootable binary image, we need to merge our compiled kernel entry and object files. This can be achieved by using the ld linker. Furthermore, we will use the cat command to combine the kernel binary and master boot record into a single binary image that can be booted.

We can connect all the above things using a different tool which is **make** and the contents in a file named makefile. It is crucial to keep in mind that in order to compile and link into free standing x86 machine code, you might need to cross compile ld and gcc. Let's now load our image into qemu, build, assemble, link, and admire the stunning X in the upper left corner of the screen. [9]

QEMU

```
XeaBIOS (version rel-1.13.0-48-gd9c812dda519-prebuilt.qemu.org)

iPXE (http://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+07F8F390+07EEF390 CA00
```

VGA Driver

By directly altering a certain memory area known as the video memory, we may create screen output with VGA.

XII WRITING THE SHELL

High level capability for interacting with computer hardware is provided by operating systems. The user must be given access to this capability in some way, such as through a layer that surrounds the kernel and exposes basic instructions. Typically, this outer layer is referred to as a "shell". We will

ISSN 2456-6616 (Online)

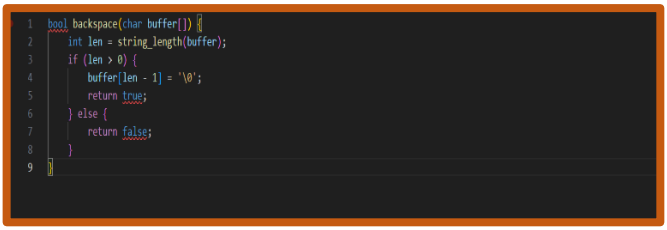
create a command-line shell because the VGA driver we have is merely a very basic text-based driver.

We will add basic functionalities in our shell:

Backspace

Typos should be able to be fixed by pressing backspace, which removes the final character from the screen and the buffer. By flipping the append function, the buffer modification can be implemented. We merely set the buffer's final non-zero byte to 0. If we were successful in removing an element from the buffer, the procedure would return true; otherwise, it would return false. Be aware that you must use the **#include <stdbool.h>** command to import the bool type definition. [9]

Below is the code snippet for **backspace** command.

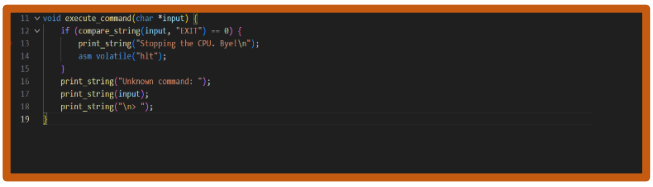


```
bool backspace(char buffer[]) {
    int len = string_length(buffer);
    if (len > 0) {
        buffer[len - 1] = '\0';
        return true;
    } else {
        return false;
    }
}
```

Parsing and Executing Commands

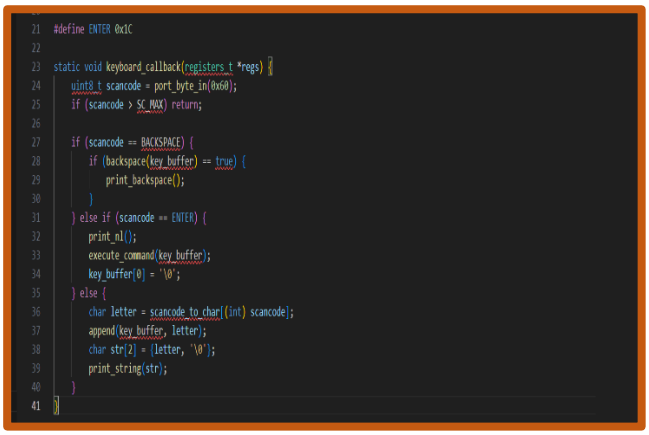
We want to run the specified command each time the user presses the enter key. Usually, this entails first processing the command, sometimes dividing it into several subcommands, parsing arguments, or invoking other functions. [9]

The code snippet for the **execute_command** is shown below:



```
void execute_command(char *input) {
    if (compare_string(input, "EXIT") == 0) {
        print_string("Stopping the CPU. Bye!\n");
        asm volatile("hlt");
    }
    print_string("Unknown command: ");
    print_string(input);
    print_string("\n> ");
}
```

Finally, the keyboard callback is modified such that pressing the enter key will cause the cursor to jump to the next line, execute_command to be called, and the key buffer to be reset.



```
#define ENTER 0x1C

static void keyboard_callback(registers_t *regs) {
    uint8_t scancode = port_byte_in(0x60);
    if (scancode > SC_MAX) return;

    if (scancode == BACKSPACE) {
        if (backspace(key_buffer) == true) {
            print_backspace();
        }
    } else if (scancode == ENTER) {
        print_nl();
        execute_command(key_buffer);
        key_buffer[0] = '\0';
    } else {
        char letter = scancode_to_char((int) scancode);
        append(key_buffer, letter);
        char str[2] = {letter, '\0'};
        print_string(str);
    }
}
```

XIII REFERENCES

[1] Hongbiao Jiang a , Wanlin Gaoa,∗ , Manwei Wang b , Simon See c,d , Ying Yang a , Wei Liue , Jin Wang a Mathematical and Computer Modelling 51 (2010) 1421–1427

[2] Shubhi R. M. Zeebaree, Rizgar R Zeebari, Mohammed A. M. Sadeeq, Zainab Salih Ageed , A Comprehensive Study of Kernel (Issues and Concepts) in Different Operating Systems, 2021

[3] Muthu Dayalan Kernel Design in Operating System 2019 JETIR May 2019, Volume 6, Issue 5

[4] Yuki Kinebuchi, Kazuo Makijima, Takushi Morita, Alexandre Courbot, and Tatsuo Nakajima Composition Kernel: A Software Solution for Constructing a Multi-OS Embedded System , Hindawi Publishing Corporation EURASIP Journal on Embedded Systems Volume 2010

[5] Stephen J. Bigelow, Senior Technology Editor, Techt@rget

[6] Ramandeep Singh, Architecture of Operating System, Scaler Topics

[7] Vikash kumar, Architecture of Operating System, Codingninjas/studio

[8] Javatpoint, what is kernel, Javatpoint.com

[9] Frank Rosner, Writing my own Boot loader, Dev Community, 2021

[10] Ahlawat, A., Rana, A., Goyal, N. et al. Potential role of nitric oxide synthase isoforms in pathophysiology of neuropathic pain. Inflammopharmacol 22, 269–278 (2014). https://doi.org/10.1007/s10787-014-0213-0

[11] A. R. Yeruva, P. Choudhari, A. Shrivastava, D. Verma, S. Shaw and A. Rana, "Covid-19 Disease Detection using Chest X-Ray Images by Means of CNN," 2022 2nd International Conference on Technological Advancements in Computational Sciences (ICTACS), Tashkent, Uzbekistan, 2022, pp. 625-631, doi: 10.1109/ICTACS56270.2022.9988148.

[12] Ghosh, S., Rana, A., & Kansal, V. (2020). A benchmarking framework using nonlinear manifold detection techniques for software defect prediction. International Journal of Computational Science and Engineering, 21(4), 593-614.

[13] Raghavendra, M. S., Chawla, P., & Rana, A. (2020, June). A survey of optimization algorithms for fog computing service placement. In 2020 8th international conference on reliability, infocom technologies and optimization (trends and future directions)(ICRITO) (pp. 259-262). IEEE.

[14] Gupta, S., Rana, A., & Kansal, V. (2020). Optimization in wireless sensor network using soft computing. In Proceedings of the Third International Conference on Computational Intelligence and Informatics: ICCII 2018 (pp. 801-810). Springer Singapore.

[15] Kunwar, V., Agarwal, N., Rana, A., & Pandey, J. P. (2018). Load balancing in cloud—a systematic review. Big Data Analytics: Proceedings of CSI 2015, 583-593.

[16] Chawla, P., Chana, I., & Rana, A. (2015). A novel strategy for automatic test data generation using soft computing technique. Frontiers of Computer Science, 9, 346-363.

[17] Walia, H., Rana, A., & Kansal, V. (2017, September). A Naïve Bayes Approach for working on Gurmukhi Word Sense Disambiguation. In 2017 6th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO) (pp. 432-435). IEEE.

[18] Dash, Y., Dubey, S. K., & Rana, A. (2012). Maintainability prediction of object oriented software system by using artificial neural network approach. International Journal of Soft Computing and Engineering (IJSCE), 2(2), 420-423.

[19] Dubey, S. K., & Rana, A. (2010). A comprehensive assessment of object-oriented software systems using metrics approach. International Journal on Computer Science and Engineering, 2(8), 2726-2730.

[20] S. Gupta, A. Rana and V. Kansal, "Comparison of Heuristic techniques:A case of TSP," 2020 10th International Conference on Cloud Computing, Data Science & Engineering (Confluence), Noida, India, 2020, pp. 172-177, doi: 10.1109/Confluence47617.2020.9058211.

[21] Ghosh, S., Rana, A., & Kansal, V. (2018). A nonlinear manifold detection based model for software defect prediction. Procedia computer science, 132, 581-594.

[22] Chawla, P., Chana, I., & Rana, A. (2016). Cloud-based automatic test data generation framework. Journal of Computer and System Sciences, 82(5), 712-738.

[23] Bhardwaj, M., & Rana, A. (2016). Key Software Metrics and its Impact on each other for Software Development Projects. International Journal of Electrical & Computer Engineering (2088-8708), 6(1).

[24] Rana, A., & Sharma, S. (2016). Mechanism of sphingosine-1-phosphate induced cardioprotection against I/R injury in diabetic rat heart: Possible involvement of glycogen synthase kinase 3β and mitochondrial permeability transition pore. Clinical and Experimental Pharmacology and Physiology, 43(2), 166-173.

[25] G. Dubey, A. Rana and N. K. Shukla, "User reviews data analysis using opinion mining on web," 2015 International Conference on Futuristic Trends on Computational Analysis and Knowledge Management (ABLAZE), Greater Noida, India, 2015, pp. 603-612, doi: 10.1109/ABLAZE.2015.7154934.

[26] Ghosh, S., Rana, A., Kansal, V. (2017). Predicting Defect of Software System. In: Satapathy, S., Bhateja, V., Udgata, S., Pattnaik, P. (eds) Proceedings of the 5th International Conference on Frontiers in Intelligent Computing: Theory and Applications . Advances in Intelligent Systems and Computing, vol 516. Springer, Singapore. https://doi.org/10.1007/978-981-10-3156-4_6

[27] Sanjay Kumar Dubey, Ajay Rana, and Yajnaseni Dash. 2012. Maintainability prediction of object-oriented software system by multilayer perceptron model. SIGSOFT Softw. Eng. Notes 37, 5 (September 2012), 1–4. https://doi.org/10.1145/2347696.2347703

[28] S. Chawla, G. Dubey and A. Rana, "Product opinion mining using sentiment analysis on smartphone reviews," 2017 6th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO), Noida, India, 2017, pp. 377-383, doi: 10.1109/ICRITO.2017.8342455.

[29] Dubey, S. K., Rana, A., & Sharma, A. (2012). Usability evaluation of object oriented software system using fuzzy logic approach. International Journal of Computer Applications, 43(19), 1-6.

[30] Saini, Rimmi, Sanjay Kumar Dubey, and Ajay Rana. "Analytical study of maintainability models for quality evaluation." Indian Journal of Computer Science and Engineering 2.3 (2011): 449-454.

[31] Ghosh, Soumi, Ajay Rana, and Vineet Kansal. "A statistical comparison for evaluating the effectiveness of linear and nonlinear manifold detection techniques for software defect prediction." International Journal of Advanced Intelligence Paradigms 12.3-4 (2019): 370-391.

[32] A. Singh, M. Chaudhary, A. Rana and G. Dubey, "Online Mining of data to generate association rule mining in large databases," 2011 International Conference on Recent Trends in Information Systems, Kolkata, India, 2011, pp. 126-131, doi: 10.1109/ReTIS.2011.6146853.

[33] N. Tyagi, A. Rana and V. Kansal, "Creating Elasticity with Enhanced Weighted Optimization Load Balancing Algorithm in Cloud Computing," 2019 Amity International Conference on Artificial Intelligence (AICAI), Dubai, United Arab Emirates, 2019, pp. 600-604, doi: 10.1109/AICAI.2019.8701375.

[34] Dubey, Sanjay Kumar, and Ajay Rana. "A fuzzy approach for evaluation of maintainability of object oriented software system." International Journal of Computer Applications 49.21 (2012).

[35] P. K. Kushwaha and M. Kumaresan, "Machine learning algorithm in healthcare system: A Review," 2021 International Conference on Technological Advancements and Innovations (ICTAI), Tashkent, Uzbekistan, 2021, pp. 478-481, doi: 10.1109/ICTAI53825.2021.9673220.

[36] P. K. Kushwaha, V. Bibhu, B. P. Lohani and D. Singh, "Review on information security, laws and ethical issues with online financial system," 2016 International Conference on Innovation and Challenges in Cyber Security (ICICCS-INBUSH), Greater Noida, India, 2016, pp. 49-53, doi: 10.1109/ICICCS.2016.7542350.

[37] G. Gulati, B. P. Lohani and P. K. Kushwaha, "A Novel Application Of IoT In Empowering Women Safety Using GPS Tracking Module," 2020 Research, Innovation, Knowledge Management and Technology Application for Business Sustainability (INBUSH), Greater Noida, India, 2020, pp. 131-137, doi: 10.1109/INBUSH46973.2020.9392193.

[38] D. Pareta, I. N. Verma, B. P. Lohani, P. K. Kushwaha and V. Bibhu, "IoT Enabled Smart and Efficient Musical Water Fountain," 2022 2nd International Conference on Innovative Practices in Technology and Management (ICIPTM), Gautam Buddha Nagar, India, 2022, pp. 369-373, doi: 10.1109/ICIPTM54933.2022.9754129.

[39] B. P. Lohani, M. Trivedi, R. J. Singh, V. Bibhu, S. Ranjan and P. K. Kushwaha, "Machine Learning Based Model for Prediction of Loan Approval," 2022 3rd International Conference on Intelligent Engineering and Management (ICIEM), London, United Kingdom, 2022, pp. 465-470, doi: 10.1109/ICIEM54221.2022.9853160.

[40] V. Bibhu, A. Kumar, B. P. Lohani and P. K. Kushwaha, "Robust Secured Framework for Online Business Transactions over Public Network," 2021 2nd International Conference on Intelligent Engineering and Management (ICIEM), London, United Kingdom, 2021, pp. 555-560, doi: 10.1109/ICIEM51511.2021.9445380.

[41] V. Bibhu, P. K. Kushwaha and B. P. Lohani, "A review of security of the cloud computing over business with implementation," 2016 International Conference on Innovation and Challenges in Cyber Security (ICICCS-INBUSH), Greater Noida, India, 2016, pp. 192-198, doi: 10.1109/ICICCS.2016.7542342.

[42] Amardeep Gupta and Ranjeet Kumar Rout, "ROTEE: Remora Optimization and Tunicate swarm algorithm-based Energy-Efficient cluster-based routing for EHenabled heterogeneous WSNs," International Journal of Communication System (Q1), vol. 33, no. 6, pp. 1-23, 2022. DOI:https://doi.org/10.1002/dac.5372. (IF: 2.0)

[43] A. D. Gupta and R. K. Rout, "An Effective Optimization Method for Energy Efficient Clustering in EH Wireless Sensor Networks," 2021 International Conference on Technological Advancements and Innovations (ICTAI), Tashkent, Uzbekistan, 2021, pp. 699-702, doi: 10.1109/ICTAI53825.2021.9673312.

[44] S. S. N. Challapalli, P. Kaushik, S. Suman, B. D. Shivahare, V. Bibhu and A. D. Gupta, "Web Development and performance comparison of Web Development Technologies in Node.js and Python," 2021 International Conference on Technological Advancements and Innovations (ICTAI), Tashkent, Uzbekistan, 2021, pp. 303-307, doi: 10.1109/ICTAI53825.2021.9673464.

[45] A. D. Gupta, S. Suman, S. S. N. Challapalli, P. Kaushik and V. Bibhu, "Survey Paper: Comparative Study of Machine Learning Techniques and its Recent Applications," 2022 2nd International Conference on Innovative Practices in Technology and

ISSN 2456-6616 (Online)

Management (ICIPTM), Gautam Buddha Nagar, India, 2022, pp. 449-454, doi: 10.1109/ICIPTM54933.2022.9754206.