® RV Educational Institutions ®
RV College of Engineering ®

Autonomous Institution
Affiliated to Visvesvaraya
Technological University,
Belagavi

Approved by AICTE,
New Delhi

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

OPERATING SYSTEMS - CS235AI

REPORT

CREATING A KERNEL FROM SCRATCH

Submitted by

Suhas Peri	**<1RV22CS208 >**
Shaik Khadar Vali	**<1RV22CS181>**

Computer Science and Engineering
2023-2024

TABLE OF CONTENTS

Introduction:

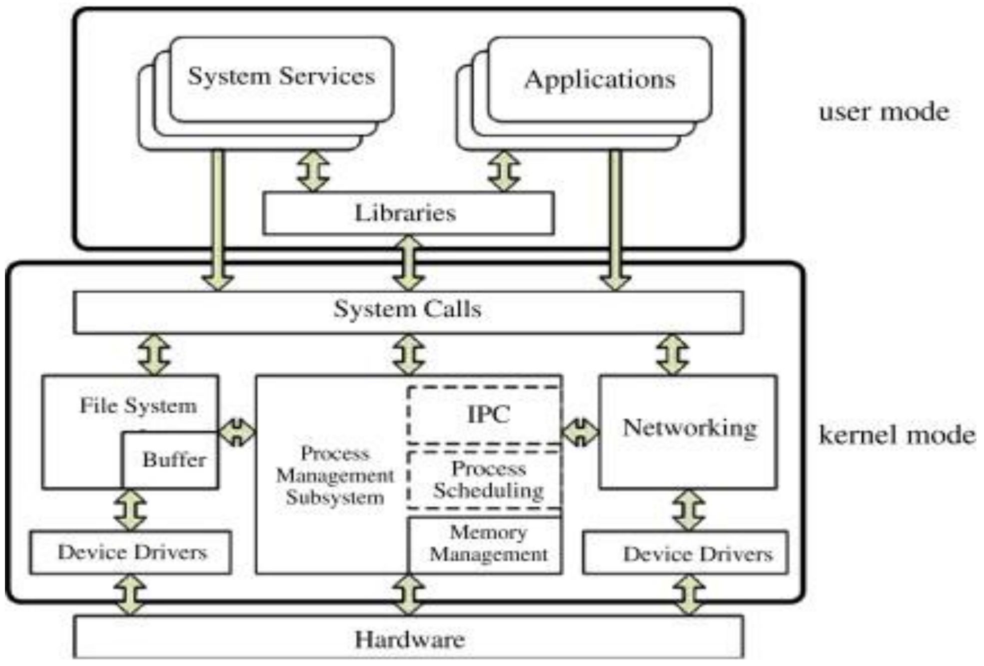
Embarking on the creation of a basic kernel from scratch presents an engaging project within the field of operating system exploration. The kernel, as the fundamental layer of an operating system, holds key responsibilities in managing hardware resources, facilitating software-to-hardware interactions, and providing vital services to user applications. This project offers budding developers a valuable opportunity to delve into the intricate workings of computer systems, enhancing their understanding of low-level programming concepts and system architecture.

A basic kernel project involves designing and implementing core functionalities such as process management, memory handling, interrupt processing, and device driver creation. Through careful planning and coding, developers build a robust foundation on which higher-level software can operate smoothly. Additionally, tailoring the kernel to specific needs allows for experimentation with different approaches and features, fostering creativity in operating system design.

While creating a basic kernel requires effort and persistence, it provides unmatched learning opportunities and personal growth. Developers gain hands-on experience with operating system internals, tackling complex algorithms and exploring hardware-level interactions. Overcoming challenges in this project not only improves problem-solving skills but also fosters resilience and deepens appreciation for the intricacies of system software development. Ultimately, crafting a basic kernel demonstrates the dedication and resourcefulness of its creators, paving the way for future innovations in computing.

System Architecture:

The system architecture of a basic kernel typically comprises several essential components that collectively manage the operation of the operating system. These components interact closely to facilitate the execution of user programs, manage system resources, and provide a seamless user experience. The following is an overview of the key components commonly found in the architecture of a basic kernel:



1. **Kernel Core**: At the heart of the system architecture lies the kernel core, which is responsible for the most fundamental operations of the operating system. This includes process management, memory management, interrupt handling, and scheduling. The kernel core interacts directly with hardware components and coordinates the execution of system tasks.
2. **Device Drivers**: Device drivers are modules within the kernel that facilitate communication between the operating system and hardware devices such as disk drives, network adapters, and input/output devices.

These drivers provide an abstraction layer that allows applications to interact with hardware devices without needing to understand the intricacies of their operation.

3. **Interrupt Handling**: Interrupt handling mechanisms are crucial for responding to events that require immediate attention, such as hardware interrupts from devices or system exceptions. The interrupt handling subsystem of the kernel prioritizes and processes these events, ensuring that the operating system responds promptly and efficiently to external stimuli.
4. **Memory Management**: The memory management component of the kernel is responsible for managing system memory, including allocation, deallocation, and protection. It ensures that each process has access to the memory it requires while preventing unauthorized access and memory leaks.
5. **Process Management**: Process management functionality oversees the creation, execution, and termination of processes within the operating system. This includes managing process states, scheduling tasks for execution, and providing inter-process communication mechanisms.
6. **System Calls**: System calls are interfaces provided by the kernel that allow user programs to request services from the operating system, such as file operations, network communication, and process management. These calls provide a standardized way for user applications to interact with the kernel.
7. **File System**: The file system component of the kernel manages storage devices and organizes data into files and directories. It provides an abstraction layer that allows applications to read from and write to storage devices without needing to understand the specific characteristics of the underlying hardware.

Methodology

1.Bootloader:

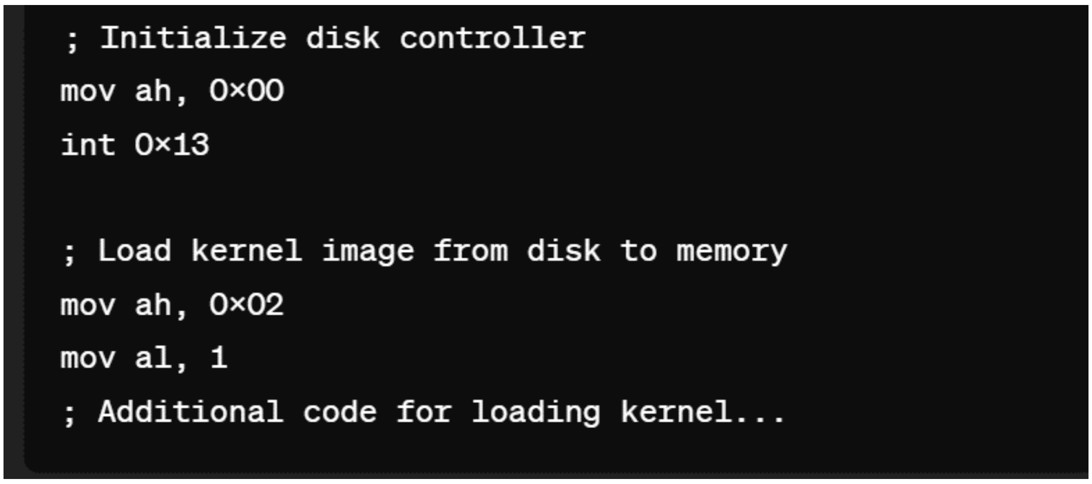
- Identify hardware initialization tasks and boot process requirements.

Bootloader Initialization:

- Initialize hardware components (e.g., CPU, memory, disk controller).

Kernel Loading:

- Read kernel image from storage device.
- Load kernel into predefined memory location.

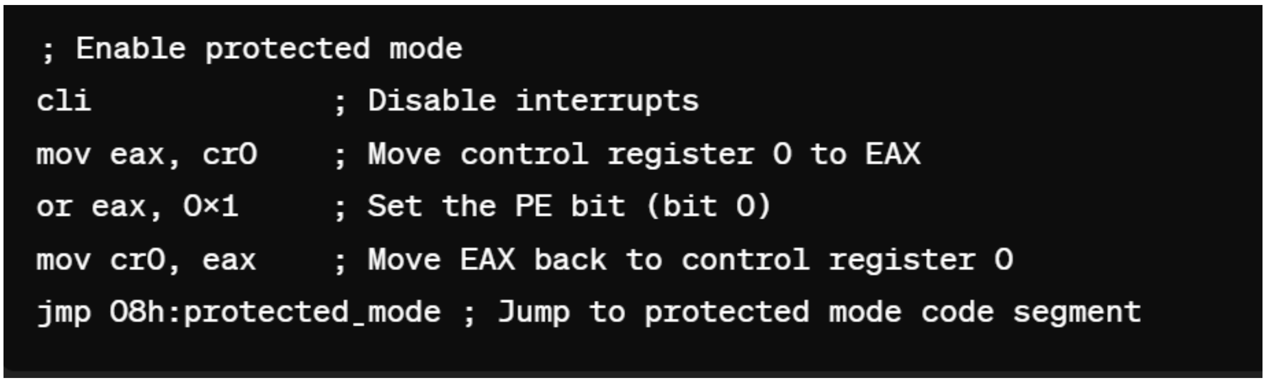


```
; Initialize disk controller
mov ah, 0x00
int 0x13

; Load kernel image from disk to memory
mov ah, 0x02
mov al, 1
; Additional code for loading kernel...
```

2.Central Processing Unit:

1. Understanding Requirements:
   - Identify CPU initialization tasks required for bootstrapping the system.
   - Determine the necessary steps for configuring the CPU to execute code correctly.

2. Processor Mode Setup:
   - Initialize the CPU in the appropriate mode (e.g., real mode, protected mode, long mode).
   - Configure segment registers and descriptor tables as necessary for the selected mode.



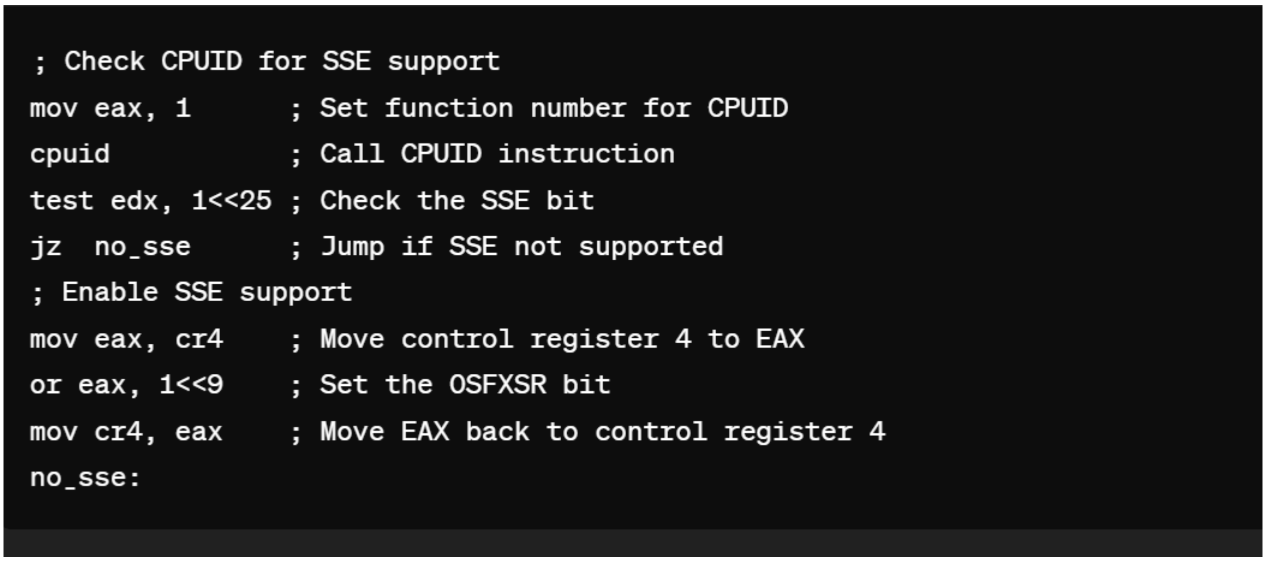
```
; Enable protected mode
cli              ; Disable interrupts
mov eax, cr0     ; Move control register 0 to EAX
or eax, 0x1      ; Set the PE bit (bit 0)
mov cr0, eax     ; Move EAX back to control register 0
jmp 08h:protected_mode ; Jump to protected mode code segment
```

Interrupt Initialization:

- Configure interrupt handling mechanisms, such as interrupt descriptor table (IDT) setup.
- Define and install interrupt service routines (ISRs) to handle hardware interrupts.
- Example code for setting up the IDT entry for keyboard interrupts:

CPU Feature Detection:

- Detect and enable CPU features (e.g., SSE, AVX) as required by the system.
- Check CPUID flags to determine supported features and configure them accordingly.
- Example code for detecting and enabling SSE support:



```
; Check CPUID for SSE support
mov eax, 1      ; Set function number for CPUID
cpuid           ; Call CPUID instruction
test edx, 1<<25 ; Check the SSE bit
jz  no_sse      ; Jump if SSE not supported
; Enable SSE support
mov eax, cr4    ; Move control register 4 to EAX
or eax, 1<<9    ; Set the OSFXSR bit
mov cr4, eax    ; Move EAX back to control register 4
no_sse:
```

3.Graphical User Interface

- Requirements Analysis:
  - Define GUI elements and functionality.
  - Choose target platform and GUI framework.
- Framework Selection:
  - Choose suitable GUI framework/toolkit.
- UI Design:
  - Design GUI layout and appearance.
- Widget Implementation:
  - Implement GUI components.
- Event Handling:
  - Implement user interaction responses.
- Graphics Rendering:
  - Draw GUI elements on screen.
- User Feedback:
  - Provide visual feedback to users.
- Testing and Debugging:
  - Test across platforms and resolutions.
  - Debug layout and rendering issues.
- Performance Optimization:
  - Optimize rendering and event handling.
- Documentation and Support:
  - Document GUI design and usage instructions.
  - Provide user support as needed.

4.KERNEL

1. Understanding Requirements**:**
   - Define kernel functionalities and system requirements.
   - Identify supported hardware architectures and devices.

2. Architecture Selection:
   - Choose appropriate kernel architecture (e.g., monolithic, microkernel).
   - Determine memory management and process scheduling strategies.

3. Kernel Initialization:
   - Implement initialization routines for essential kernel components.
   - Initialize memory management, interrupt handling, and hardware drivers.

4. Memory Management:
   - Implement memory allocation and deallocation mechanisms.
   - Manage physical and virtual memory addressing.

5. Process Management:
   - Implement process creation, scheduling, and termination routines.
   - Manage process states, context switching, and synchronization primitives.

6. Interrupt Handling:
   - Implement interrupt service routines (ISRs) for handling hardware interrupts.

- Manage interrupt priorities, masking, and vectoring.

7. Device Drivers:
    - Develop drivers for essential hardware components (e.g., disk, keyboard, display).
    - Implement device initialization, data transfer, and error handling.

8. File System:
    - Design and implement file system structures and operations (e.g., file creation, reading, writing).
    - Support file access permissions, directories, and file metadata.

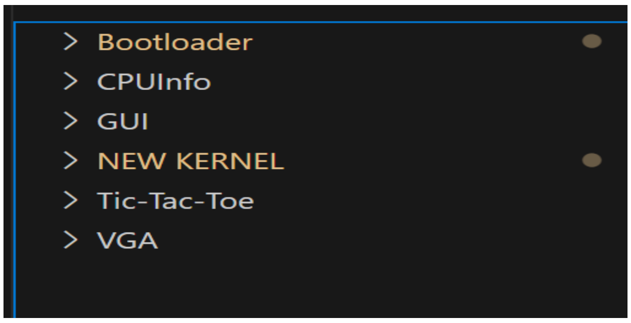
9. System Calls:
    - Define and implement system call interface for user-space interaction.
    - Provide system calls for process management, file operations, and inter-process communication.

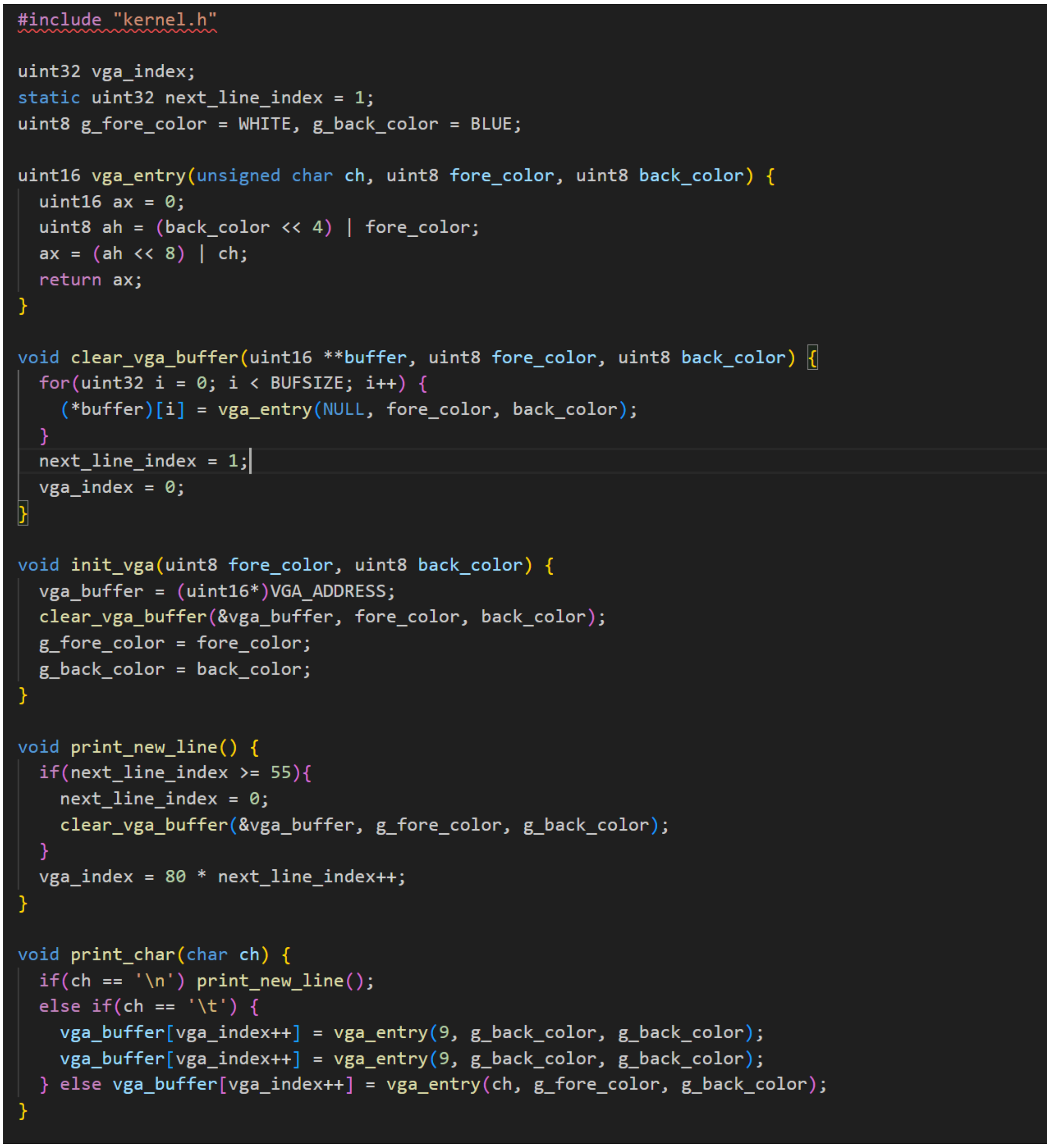
10. Documentation and Support:
    - Document kernel design, APIs, and usage guidelines for developers and users.
    - Provide user support and address questions or issues related to kernel usage.

5.VIDEO GRAPHIC ARRAY

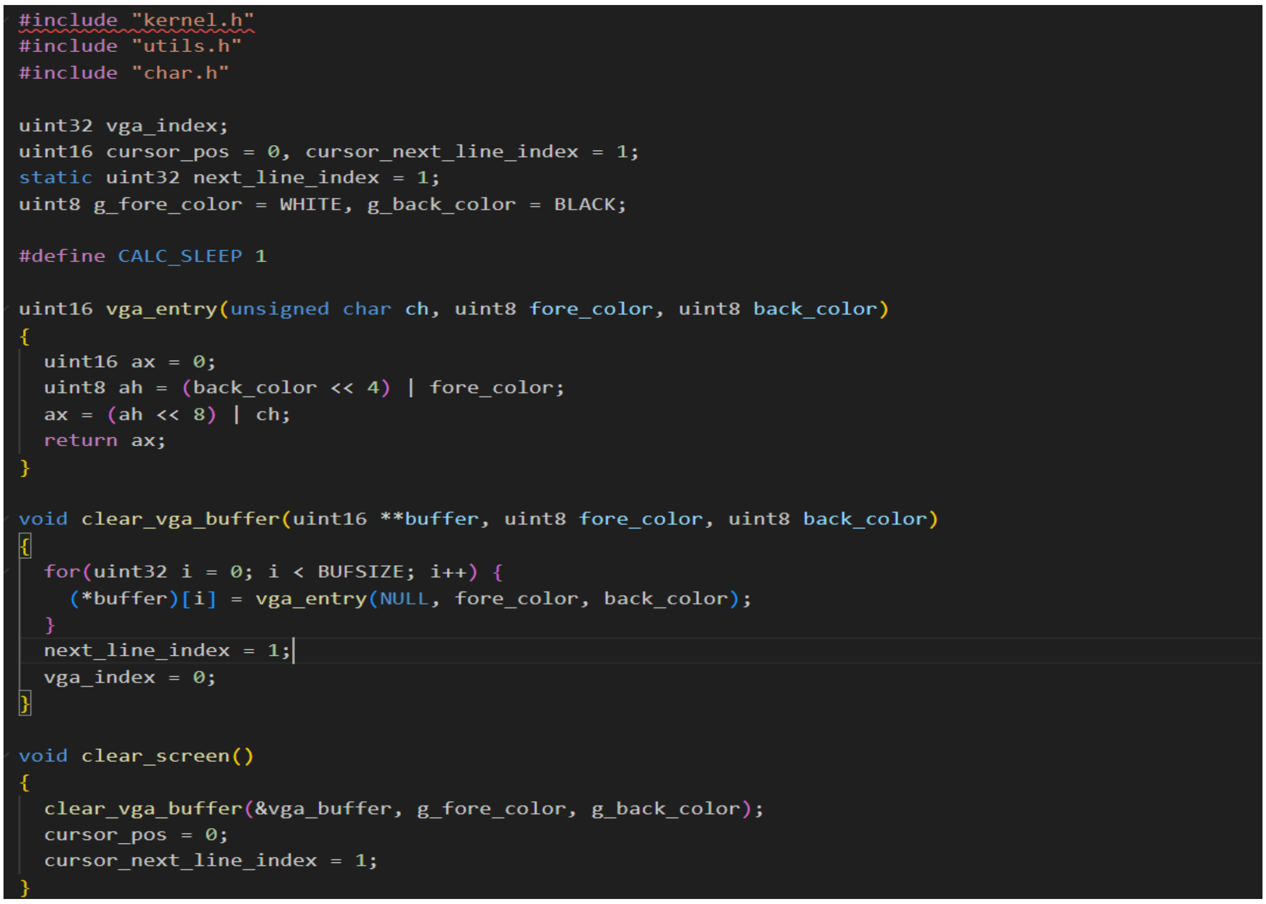
1. Understanding VGA:
    - Gain familiarity with the VGA standard, which defines the hardware interface for displaying graphics on computer monitors.
    - Learn about VGA's capabilities, including screen resolution, color depth, and supported display modes.
2. VGA Controller Initialization:
    - Initialize the VGA controller to prepare it for displaying graphics.
    - Configure VGA registers to set display mode, screen resolution, and colour depth.
3. Pixel Manipulation:
    - Use memory-mapped I/O or VGA-specific instructions to write pixel data directly to the VGA frame buffer.
4. Drawing Primitives:
    - Create functions for drawing basic geometric shapes and text on the screen.
    - Implement algorithms for drawing lines, circles, rectangles, and other shapes efficiently.
5. Double Buffering:
    - Implement double buffering to prevent screen flickering and improve rendering performance.
    - Maintain two separate frame buffers and alternate between them to display updated graphics.
6. Input Handling:
    - Handle user input from input devices such as keyboards and mice to interact with the VGA display.

CODE :

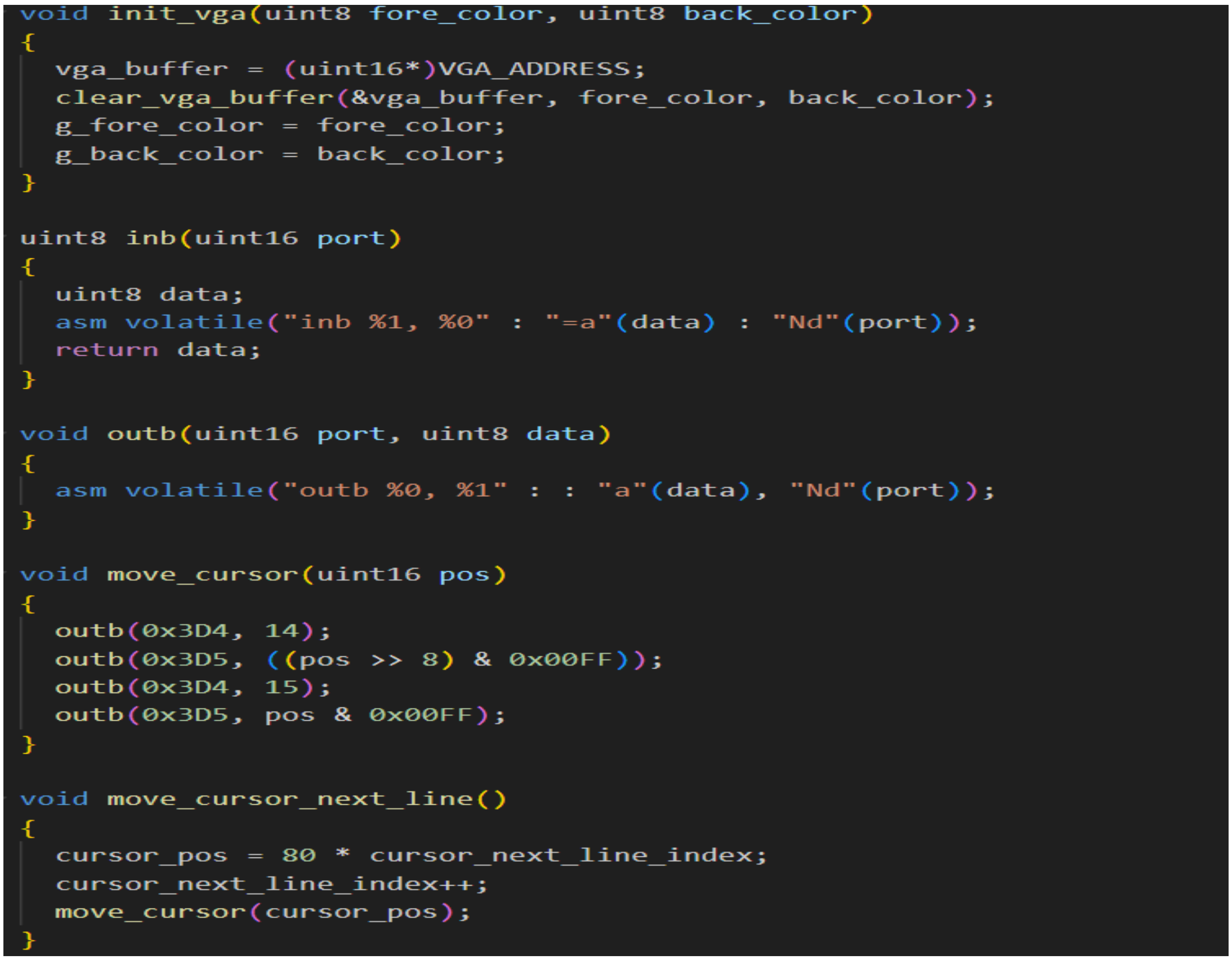




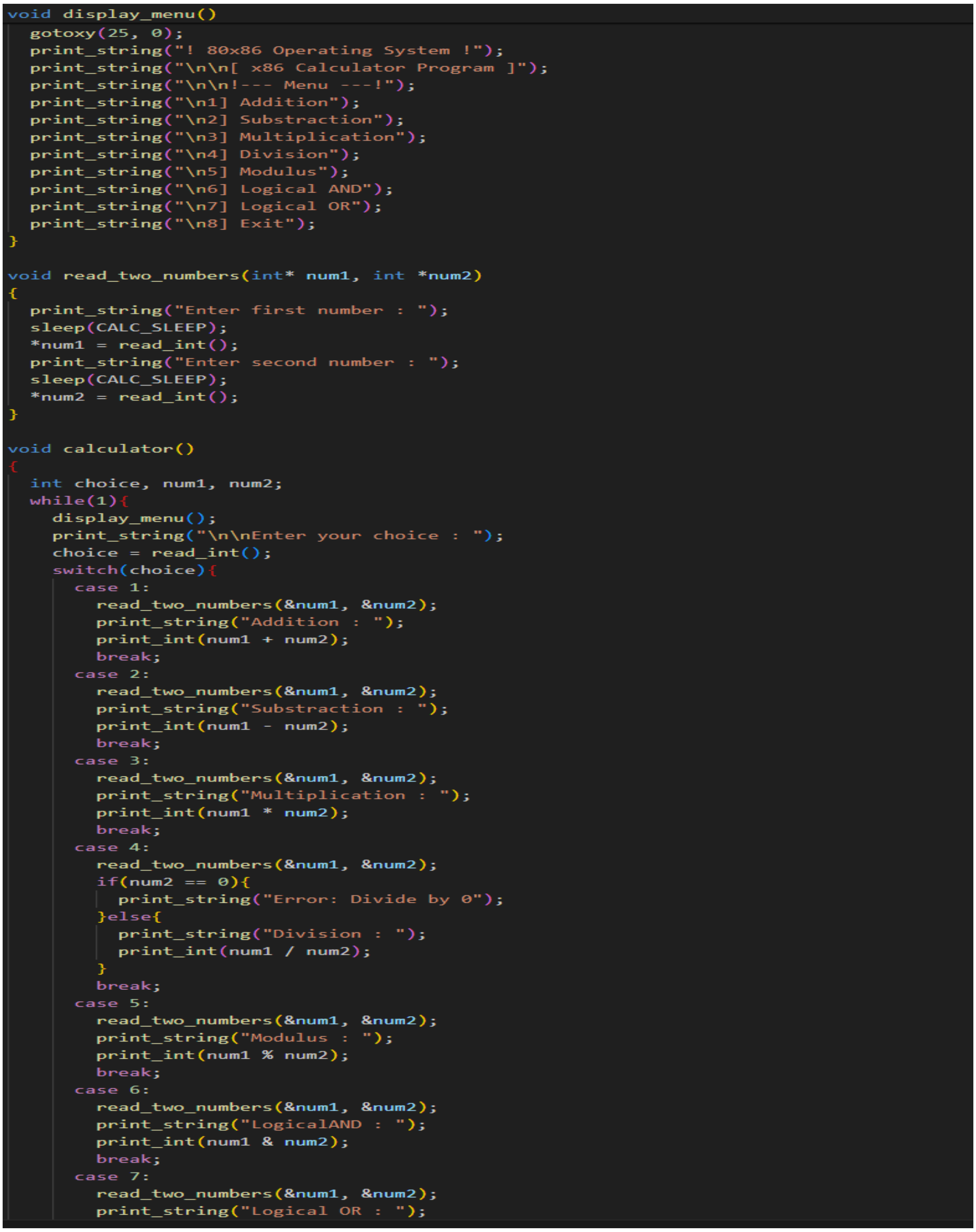
```
#include "kernel.h"

uint32 vga_index;
static uint32 next_line_index = 1;
uint8 g_fore_color = WHITE, g_back_color = BLUE;

uint16 vga_entry(unsigned char ch, uint8 fore_color, uint8 back_color) {
  uint16 ax = 0;
  uint8 ah = (back_color << 4) | fore_color;
  ax = (ah << 8) | ch;
  return ax;
}

void clear_vga_buffer(uint16 **buffer, uint8 fore_color, uint8 back_color) {
  for(uint32 i = 0; i < BUFSIZE; i++) {
    (*buffer)[i] = vga_entry(NULL, fore_color, back_color);
  }
  next_line_index = 1;
  vga_index = 0;
}

void init_vga(uint8 fore_color, uint8 back_color) {
  vga_buffer = (uint16*)VGA_ADDRESS;
  clear_vga_buffer(&vga_buffer, fore_color, back_color);
  g_fore_color = fore_color;
  g_back_color = back_color;
}

void print_new_line() {
  if(next_line_index >= 55){
    next_line_index = 0;
    clear_vga_buffer(&vga_buffer, g_fore_color, g_back_color);
  }
  vga_index = 80 * next_line_index++;
}

void print_char(char ch) {
  if(ch == '\n') print_new_line();
  else if(ch == '\t') {
    vga_buffer[vga_index++] = vga_entry(9, g_back_color, g_back_color);
    vga_buffer[vga_index++] = vga_entry(9, g_back_color, g_back_color);
  } else vga_buffer[vga_index++] = vga_entry(ch, g_fore_color, g_back_color);
}
```



```c
#include "kernel.h"
#include "utils.h"
#include "char.h"

uint32 vga_index;
uint16 cursor_pos = 0, cursor_next_line_index = 1;
static uint32 next_line_index = 1;
uint8 g_fore_color = WHITE, g_back_color = BLACK;

#define CALC_SLEEP 1

uint16 vga_entry(unsigned char ch, uint8 fore_color, uint8 back_color)
{
  uint16 ax = 0;
  uint8 ah = (back_color << 4) | fore_color;
  ax = (ah << 8) | ch;
  return ax;
}

void clear_vga_buffer(uint16 **buffer, uint8 fore_color, uint8 back_color)
{
  for(uint32 i = 0; i < BUFSIZE; i++) {
    (*buffer)[i] = vga_entry(NULL, fore_color, back_color);
  }
  next_line_index = 1;
  vga_index = 0;
}

void clear_screen()
{
  clear_vga_buffer(&vga_buffer, g_fore_color, g_back_color);
  cursor_pos = 0;
  cursor_next_line_index = 1;
}
```



```c
void init_vga(uint8 fore_color, uint8 back_color)
{
  vga_buffer = (uint16*)VGA_ADDRESS;
  clear_vga_buffer(&vga_buffer, fore_color, back_color);
  g_fore_color = fore_color;
  g_back_color = back_color;
}

uint8 inb(uint16 port)
{
  uint8 data;
  asm volatile("inb %1, %0" : "=a"(data) : "Nd"(port));
  return data;
}

void outb(uint16 port, uint8 data)
{
  asm volatile("outb %0, %1" : : "a"(data), "Nd"(port));
}

void move_cursor(uint16 pos)
{
  outb(0x3D4, 14);
  outb(0x3D5, ((pos >> 8) & 0x00FF));
  outb(0x3D4, 15);
  outb(0x3D5, pos & 0x00FF);
}

void move_cursor_next_line()
{
  cursor_pos = 80 * cursor_next_line_index;
  cursor_next_line_index++;
  move_cursor(cursor_pos);
}
```



```c
void display_menu()
  gotoxy(25, 0);
  print_string("! 80x86 Operating System !");
  print_string("\n\n[ x86 Calculator Program ]");
  print_string("\n\n!--- Menu ---!");
  print_string("\n1] Addition");
  print_string("\n2] Substraction");
  print_string("\n3] Multiplication");
  print_string("\n4] Division");
  print_string("\n5] Modulus");
  print_string("\n6] Logical AND");
  print_string("\n7] Logical OR");
  print_string("\n8] Exit");
}

void read_two_numbers(int* num1, int *num2)
{
  print_string("Enter first number : ");
  sleep(CALC_SLEEP);
  *num1 = read_int();
  print_string("Enter second number : ");
  sleep(CALC_SLEEP);
  *num2 = read_int();
}

void calculator()
{
  int choice, num1, num2;
  while(1){
    display_menu();
    print_string("\n\nEnter your choice : ");
    choice = read_int();
    switch(choice){
      case 1:
        read_two_numbers(&num1, &num2);
        print_string("Addition : ");
        print_int(num1 + num2);
        break;
      case 2:
        read_two_numbers(&num1, &num2);
        print_string("Substraction : ");
        print_int(num1 - num2);
        break;
      case 3:
        read_two_numbers(&num1, &num2);
        print_string("Multiplication : ");
        print_int(num1 * num2);
        break;
      case 4:
        read_two_numbers(&num1, &num2);
        if(num2 == 0){
          print_string("Error: Divide by 0");
        }else{
          print_string("Division : ");
          print_int(num1 / num2);
        }
        break;
      case 5:
        read_two_numbers(&num1, &num2);
        print_string("Modulus : ");
        print_int(num1 % num2);
        break;
      case 6:
        read_two_numbers(&num1, &num2);
        print_string("LogicalAND : ");
        print_int(num1 & num2);
        break;
      case 7:
        read_two_numbers(&num1, &num2);
        print_string("Logical OR : ");
```

SYSTEM CALLS USED

In the provided kernel code, there are several low-level operations and functions that can be considered as system calls. However, it's important to note that these functions are more akin to direct hardware interactions or utility functions rather than traditional system calls as seen in higher-level operating systems. Below is an explanation of some key system call-like functions used in the provided kernel:

1. inb:
    - This function reads a byte from an I/O port.
    - It takes a 16-bit port number as input and returns the byte read from that port.
    - This function is used for interacting with hardware devices by reading data from specific I/O ports.

2. outb:
    - This function writes a byte to an I/O port.
    - It takes a 16-bit port number and an 8-bit data byte as input and writes the byte to the specified port.
    - Similar to inb, outb is used for communicating with hardware devices by sending data to specific I/O ports.

3. move_cursor:
    - This function moves the cursor position on the screen in text mode.
    - It takes a 16-bit position value as input, where the high byte represents the row and the low byte represents the column.
    - Internally, it interacts with the VGA hardware to update the cursor position.

4. get_input_keycode:
    - This function retrieves a keycode from the keyboard.
    - It continuously polls the keyboard port until a keycode is received.
    - Keycodes represent the keys pressed on the keyboard and are used to interpret user input.

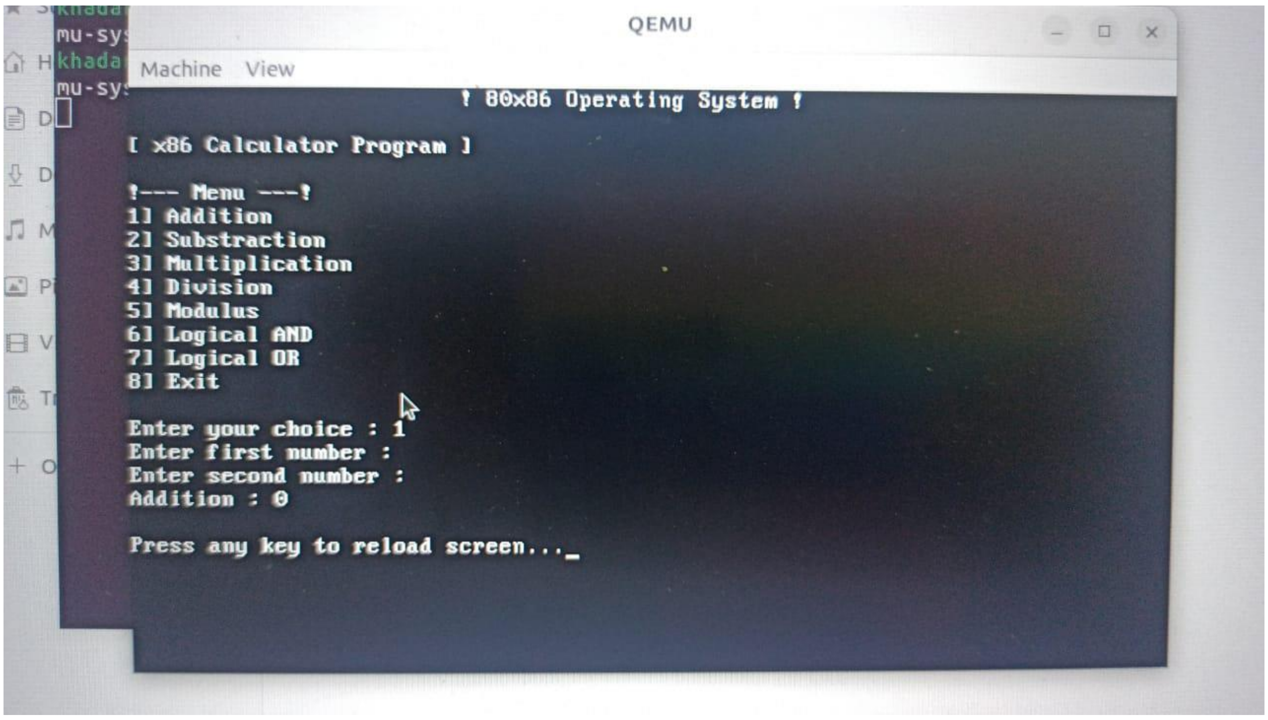
5. sleep:
    - This function introduces a delay by executing no-operation instructions for a specified period.
    - It takes a timer count as input and uses it to determine the duration of the delay.
    - Although not a traditional system call, sleep provides a simple way to pause execution for a certain amount of time.

6. print_new_line, print_char, print_string, print_int:
    - These functions are responsible for printing characters and strings to the VGA text mode screen.
    - They interact with the VGA hardware by writing data to the video memory.
    - While not system calls in the traditional sense, they serve similar purposes in providing input/output functionality to the user.

7. clear_vga_buffer:
    - This function clears the video buffer by filling it with a specified background color.
    - It takes a pointer to the video buffer, foreground color, and background color as input.
    - Internally, it iterates over the video buffer and sets each element to the specified background color.
    - This function is essential for clearing the screen or refreshing the display.

OUTPUT

The output of the basic calculator kernel primarily consists of the results of mathematical operations performed by the calculator.
Each result corresponds to an arithmetic operation such as addition, subtraction, multiplication, or division.
Additionally, upon completion of calculations or when exiting the calculator, appropriate messages are displayed to inform the user.
The output documentation aims to provide a clear understanding of the information presented in the terminal during the usage of the calculator kernel, facilitating user interaction and comprehension.



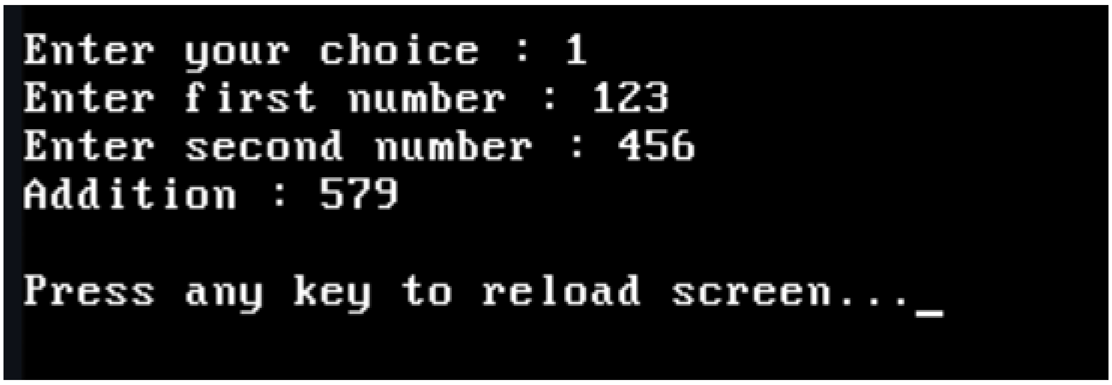
```
                    ! 80x86 Operating System !

[ x86 Calculator Program ]

!--- Menu ---!
1] Addition
2] Substraction
3] Multiplication
4] Division
5] Modulus
6] Logical AND
7] Logical OR
8] Exit

Enter your choice : 1
Enter first number :
Enter second number :
Addition : 0

Press any key to reload screen..._
```

The outputs generated by the basic calculator kernel represent the tangible results of constructing a rudimentary operating system.
These outputs signify the successful implementation of fundamental functionalities, such as arithmetic computations, within the operating system's kernel.

Each displayed result symbolizes a milestone achieved in the development process, showcasing the system's capability to execute basic operations.

Furthermore, the termination message serves as a marker of completion, indicating the conclusion of the project's initial phase.

Overall, these outputs demonstrate the tangible progress made in crafting a foundational operating system framework



```
Enter your choice : 1
Enter first number : 123
Enter second number : 456
Addition : 579

Press any key to reload screen..._
```

CONCLUSION

The outputs from the entire kernel development project can be likened to the tangible results you see when creating a basic operating system from scratch. They demonstrate the functionalities and capabilities of the system, such as managing processes, memory, and input/output operations. Each output, whether it's displaying text on the screen or executing a command, represents a step forward in building the operating system. When you reach the end of the project and see the final message indicating the termination of the kernel, it signifies the completion of your initial development phase. In essence, these outputs reflect the progress and achievements in crafting your own operating system.