RV College of Engineering®
Mysore Road, RV Vidyaniketan Post,
Bengaluru - 560059, Karnataka, India

Go, change the world®

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

OPERATING SYSTEMS - CS235AI

EXPERIENTIAL LEARNING REPORT

Kernel Development in C

1. SRAVANI H - 1RV22CS201
2. SYED FARHAN ASHRAF - 1RV22CS214

Submitted to

Dr. Jyoti Shetty

Assistant Professor

Department of Computer Science and Engineering

CONTENTS:

PROBLEM STATEMENT:

1. Create a basic operating system kernel with bootloader initialization, display output, and keyboard input handling.

INTRODUCTION:

Operating system (OS) development is a complex yet foundational aspect of computer science. At its core lies the kernel, the engine driving hardware interaction and user experience. This report delves into the process of developing a basic kernel, starting with bootloader creation in 16-bit assembly and extending to keyboard input handling and display functionality.

Beginning with bootloader development, we explore the critical role of initializing the system before OS execution, spotlighting bootloaders like GNU GRUB.

The report further examines keyboard input handling, showcasing port I/O operations to capture user keystrokes.

In summary, it provides a concise overview of kernel development, highlighting its pivotal role in system operation and user interaction.

RELEVANT OPERATING SYSTEM CONCEPTS:

1.**The GNU Assembler**, often abbreviated as GAS, is the assembler provided as part of the GNU Compiler Collection (GCC). It is a component of the GNU toolchain used for compiling programs written in languages like C, C++, and Fortran, into machine code that can be executed by a computer's processor. The GNU Assembler translates assembly language code, which is a low-level programming language that closely corresponds to the machine code instructions understood by the CPU, into binary machine code that the computer can execute directly. Assembly language provides a human-readable representation of machine instructions, allowing programmers to write code that interacts directly with the hardware of a computer system.

2.**GNU/Linux** is used to refer to the combination of the Linux kernel with the GNU operating system, developed by the Free Software Foundation (FSF) and its founder, Richard Stallman. GNU/Linux distributions come in various flavours, such as Ubuntu, Fedora, Debian, and CentOS, each of which may include different software packages and configurations, but all are based on the Linux kernel and the GNU userland tools. The combination of the Linux kernel with the GNU operating system and various other components has resulted in a powerful, versatile, and widely used operating system that powers everything from servers and desktop computers to embedded systems and mobile devices. Additionally, GNU/Linux is renowned for its stability, security, and the extensive range of free and open-source software available for it.

3. **grub-mkrescue** is a command-line utility used in the GNU GRUB (Grand Unified Bootloader) bootloader system. Its primary function is to create a bootable ISO image that contains the GRUB bootloader along with specified configuration files and bootable kernels. This ISO image can be burned onto a CD/DVD or written to a USB drive to create a bootable media. grub-mkrescue allows users to create rescue discs or installation media for their operating

systems, providing a convenient way to boot into a system or perform system recovery tasks. Additionally, it's often used in the process of creating custom Linux distributions or live CDs/DVDs. Overall, grub-mkrescue is a powerful tool for creating bootable media with GRUB bootloader functionality.

4. OS Core Functionality:

- Simulates essential CPU scheduling algorithms, integral to OS operation.

5. Process & Resource Management:

- Models processes and allocates resources akin to OS process management.

6. Algorithm Variety:

- Implements common scheduling strategies (FCFS, SJF, Round Robin, Priority, Real-Time) found in OS.

7. Real-Time Simulation:

- Includes Real-Time Scheduling, crucial for applications with stringent timing requirements.

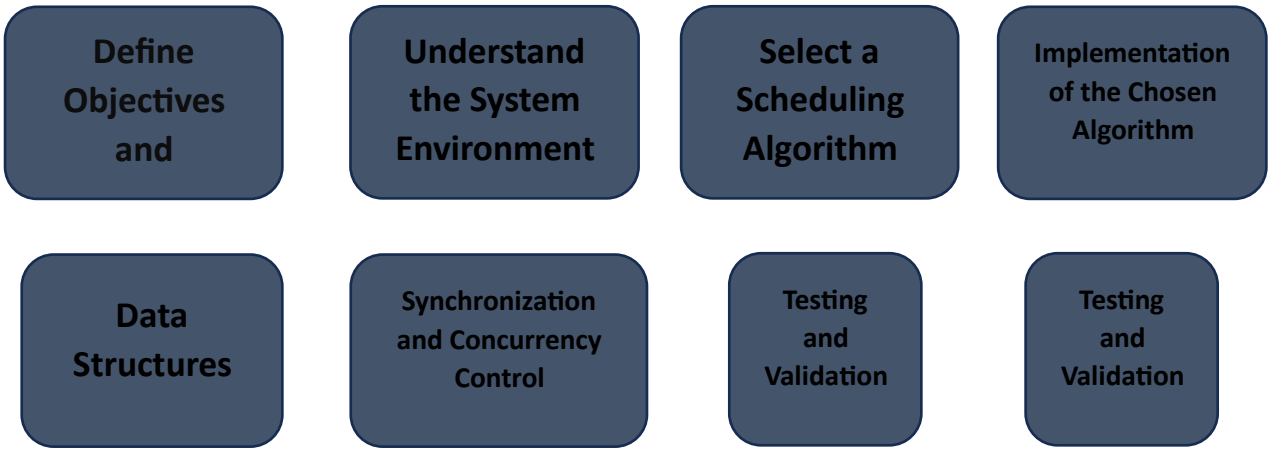
8. Decision-Making Illustration:

- Demonstrates the OS-like decision-making process in selecting processes for execution.

9. Educational Tool:

Provides a practical simulation for understanding OS CPU scheduling principles.

METHODOLOGY



DESIGN OF OPERATING SYSTEM KERNEL DEVELOPMENT:

1. Bootloader Development:

- Objective: Initialize the system and load the kernel into memory.
- Implementation:
- Define necessary information for multiboot compliance: magic number, flags, and checksum.
- Set up the stack and call the kernel entry point.
- Explanation: Bootloader initializes essential system components and prepares the environment for kernel execution. It sets up parameters required by the multiboot specification and transfers control to the kernel.

2. Kernel Initialization:

- Objective: Initialize the kernel environment and set up display output.
- Implementation:
- Initialize VGA display buffer and set colors for text output.
- Define functions for interacting with VGA buffer: print characters, strings, and integers.
- Implement kernel entry point to initialize VGA display and print initial message.
- Explanation: Kernel initialization is crucial for setting up the operating environment. It prepares the display buffer for output and initializes necessary data structures for further execution.

3. Keyboard Input Handling:

- Objective: Capture user input from the keyboard and process it.
- Implementation:
- Define constants for keyboard scan codes representing various keys.
- Implement functions to read input from the keyboard port, convert scan codes to ASCII characters, and handle keyboard interrupts.
- Explanation: Keyboard input handling enables interaction with the user. It allows the kernel to respond to user commands and input in real-time, enhancing the user experience.

4. Integration and Testing:

- Objective: Verify kernel functionality and compatibility using emulation tools.
- Implementation:

- Compile bootloader and kernel source files into object files.
- Link object files to create a multiboot-compliant kernel binary.
- Test kernel using emulation tools like QEMU to ensure proper functionality.
- Explanation: Integration and testing are critical phases to ensure the correctness and robustness of the kernel. Emulation tools like QEMU simulate system environments, allowing developers to assess kernel behaviour without relying on physical hardware.

SOURCE CODE: (Kernel)

1. Boot.S

```
# set magic number to 0x1BADB002 to identified by bootloader
.set MAGIC,    0x1BADB002
# set flags to 0
.set FLAGS,    0
# set the checksum
.set CHECKSUM, -(MAGIC + FLAGS)
# set multiboot enabled
.section .multiboot
# define type to long for each data defined as above
.long MAGIC
.long FLAGS
.long CHECKSUM
# set the stack bottom
stackBottom:
# define the maximum size of stack to 512 bytes
.skip 1024
# set the stack top which grows from higher to lower
stackTop:
.section .text
.global _start
.type _start, @function
_start:
```

```
 # assign current stack pointer location to stackTop
mov $stackTop, %esp
# call the kernel main source
call kernel_entry
cli
# put system in infinite loop
hltLoop:
hlt
jmp hltLoop
.size _start, . - _start
```

2. Kernel.c

```
#include "kernel.h"
//index for video buffer array
uint32 vga_index;
//counter to store new lines
static uint32 next_line_index = 1;
//fore & back color values
uint8 g_fore_color = WHITE, g_back_color = BLUE;
//digit ascii code for printing integers
int digit_ascii_codes[10] = {0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
0x38, 0x39};
/*
16 bit video buffer elements(register ax)
8 bits(ah) higher :
  lower 4 bits - forec olor
  higher 4 bits - back color
8 bits(al) lower :
  8 bits : ASCII character to print
*/
uint16 vga_entry(unsigned char ch, uint8 fore_color, uint8 back_color)
{
  uint16 ax = 0;
  uint8 ah = 0, al = 0;
 ah = back_color;
  ah <<= 4;
  ah |= fore_color;
  ax = ah;
  ax <<= 8;
```

```
  al = ch;
  ax |= al;

  return ax;
}

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

//initialize vga buffer
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;
}

/*increase vga_index by width of row(80)*/
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;
  next_line_index++;
}

//assign ascii character to video buffer
void print_char(char ch)
{
```

```
  vga_buffer[vga_index] = vga_entry(ch, g_fore_color, g_back_color);
  vga_index++;
}


uint32 strlen(const char* str)
{
  uint32 length = 0;
  while(str[length])
    length++;
  return length;
}

uint32 digit_count(int num)
{
  uint32 count = 0;
  if(num == 0)
    return 1;
  while(num > 0){
    count++;
    num = num/10;
  }
  return count;
}

void itoa(int num, char *number)
{
  int dgcount = digit_count(num);
  int index = dgcount - 1;
  char x;
  if(num == 0 && dgcount == 1){
    number[0] = '0';
    number[1] = '\0';
  }else{
    while(num != 0){
      x = num % 10;
      number[index] = x + '0';
      index--;
      num = num / 10;
    }
```

```
    number[dgcount] = '\0';
  }
}

//print string by calling print_char
void print_string(char *str)
{
  uint32 index = 0;
  while(str[index]){
    print_char(str[index]);
    index++;
  }
}
//print int by converting it into string
//& then printing string
void print_int(int num)
{
  char str_num[digit_count(num)+1];
  itoa(num, str_num);
  print_string(str_num);
}
void kernel_entry()
{
  //first init vga with fore & back colors
  init_vga(WHITE, BLACK);

  /*call above function to print something
    here to change the fore & back color
    assign g_fore_color & g_back_color to color values
    g_fore_color = BRIGHT_RED;
  */
  print_string("Hello World!");
  print_new_line();
  print_int(123456789);
  print_new_line();
  print_string("Goodbye World!");

}
```

3. run.sh

#assemble boot.s file as

--32 boot.s -o boot.o

#compile kernel.c file

gcc -m32 -c kernel.c -o kernel.o -std=gnu99 -ffreestanding -O2 -Wall -Wextra

#linking the kernel with kernel.o and boot.o files

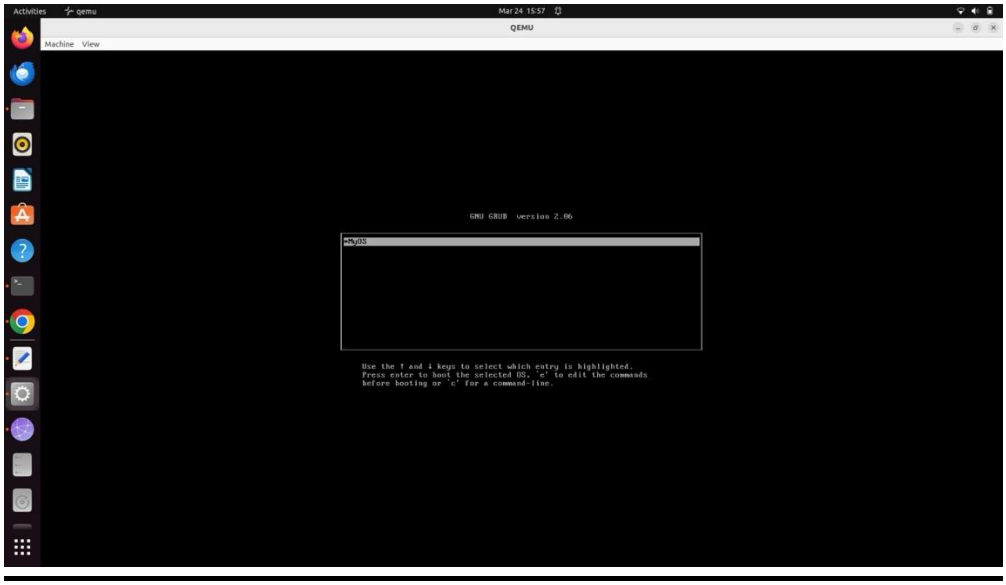
ld -m elf_i386 -T linker.ld kernel.o boot.o -o MyOS.bin -nostdlib

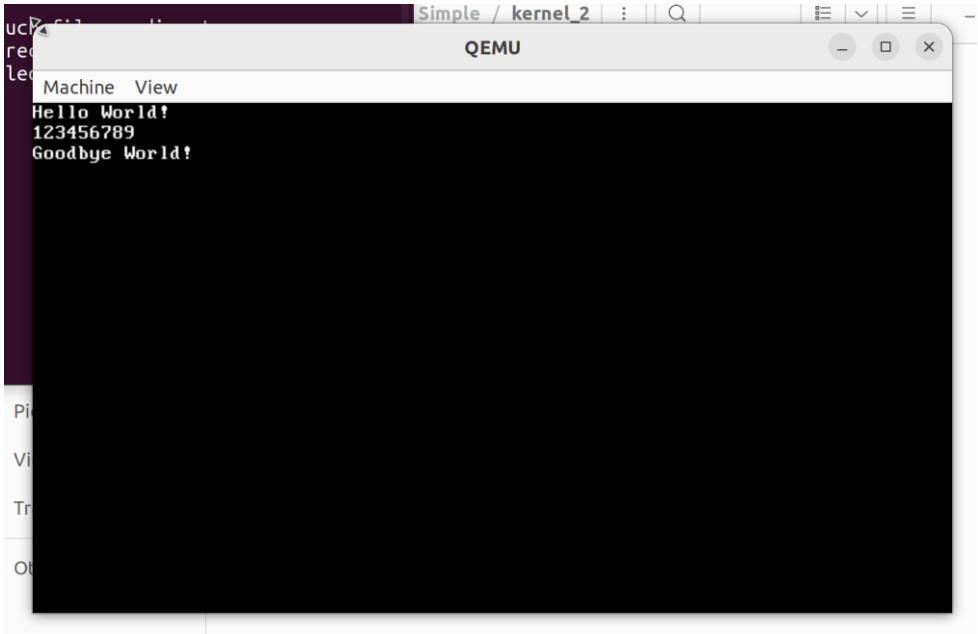
#check MyOS.bin file is x86 multiboot file or not grub-file --is-x86-multiboot MyOS.bin

#building the iso file mkdir -p isodir/boot/grub cp MyOS.bin isodir/boot/MyOS.bin cp grub.cfg isodir/boot/grub/grub.cfg grub-mkrescue -o MyOS.iso isodir

#run it in qemu qemu-system-x86_64 -cdrom MyOS.iso

OUTPUT





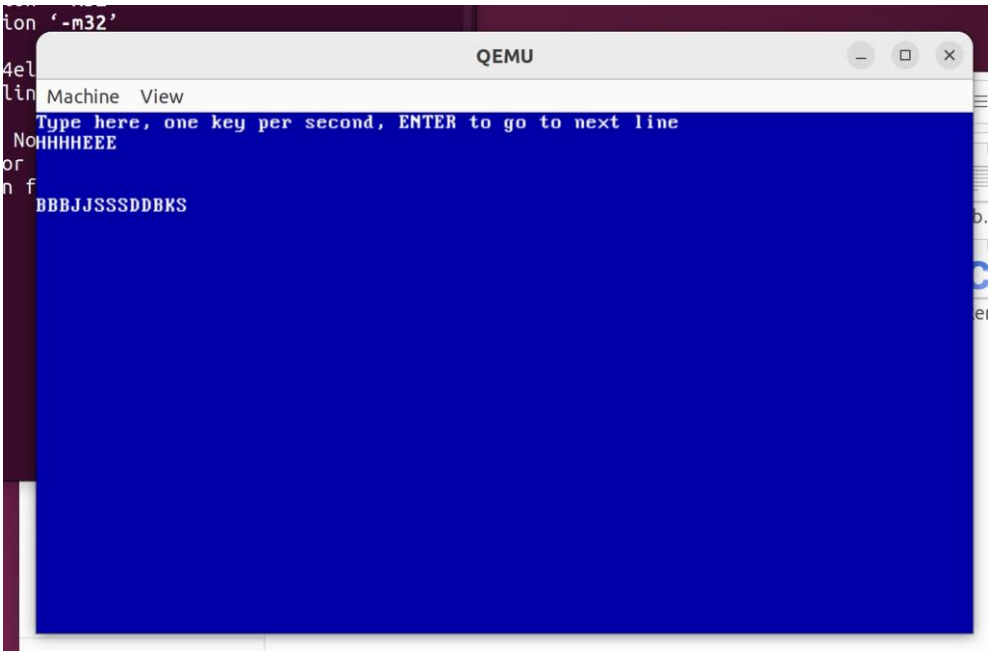
4.Keyboard.c

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

uint32 vga_index;
static uint32 next_line_index = 1;
uint8 g_fore_color = WHITE, g_back_color = BLUE;
int digit_ascii_codes[10] = {0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39};

/*
this is same as we did in our assembly code for vga_print_char

vga_print_char:
  mov di, word[VGA_INDEX]
  mov al, byte[VGA_CHAR]

  mov ah, byte[VGA_BACK_COLOR]
  sal ah, 4
  or ah, byte[VGA_FORE_COLOR]

  mov [es:di], ax

  ret

*/
uint16 vga_entry(unsigned char ch, uint8 fore_color, uint8 back_color)
{
  uint16 ax = 0;
  uint8 ah = 0, al = 0;
```

```
  ah = back_color;
  ah <<= 4;
  ah |= fore_color;
  ax = ah;
  ax <<= 8;
  al = ch;
  ax |= al;

  return ax;
}

void clear_vga_buffer(uint16 **buffer, uint8 fore_color, uint8 back_color)
{
  uint32 i;
  for(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;
  next_line_index++;
}

void print_char(char ch)
{
  vga_buffer[vga_index] = vga_entry(ch, g_fore_color, g_back_color);
  vga_index++;
}

void print_string(char *str)
{
  uint32 index = 0;
  while(str[index]){
```

```
    print_char(str[index]);
    index++;
  }
}

void print_int(int num)
{
  char str_num[digit_count(num)+1];
  itoa(num, str_num);
  print_string(str_num);
}

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

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

char get_input_keycode()
{
  char ch = 0;
  while((ch = inb(KEYBOARD_PORT)) != 0){
    if(ch > 0)
      return ch;
  }
  return ch;
}

/*
keep the cpu busy for doing nothing(nop)
so that io port will not be processed by cpu
here timer can also be used, but lets do this in looping counter
*/
void wait_for_io(uint32 timer_count)
{
  while(1){
    asm volatile("nop");
    timer_count--;
    if(timer_count <= 0)
      break;
    }
}

void sleep(uint32 timer_count)
```

```
{
  wait_for_io(timer_count);
}

void test_input()
{
  char ch = 0;
  char keycode = 0;
  do{
    keycode = get_input_keycode();
    if(keycode == KEY_ENTER){
      print_new_line();
    }else{
      ch = get_ascii_char(keycode);
      print_char(ch);
    }
    sleep(0x02FFFFFF);
  }while(ch > 0);
}

void kernel_entry()
{
  init_vga(WHITE, BLUE);
  print_string("Type here, one key per second, ENTER to go to next line");
  print_new_line();
  test_input();

}
```



Conclusion:

Creating your own kernel and integrating functionalities like printing "Hello, World!" and handling keyboard input is a significant milestone in operating system development. Here's a conclusion on this topic:

Embarking on the journey of creating a custom kernel and implementing basic functionalities like printing messages and handling keyboard input provides invaluable insights into the intricate workings of computer systems. Through this endeavor, one gains a deeper understanding of low-level programming, memory management, interrupt handling, and device interaction.

Printing "Hello, World!" may seem trivial, but it symbolizes the initiation of communication between the kernel and the user. It establishes a foundational link between the underlying system and the outside world, showcasing the kernel's ability to convey information to the user space.

Integrating keyboard input handling elevates the kernel's functionality, enabling interaction beyond passive output. The ability to capture and process keyboard events lays the groundwork for building more complex user interfaces and command-line interfaces, essential for user interaction in any operating system.

Moreover, the process of creating a kernel fosters problem-solving skills, as developers encounter numerous challenges along the way. From memory allocation issues to synchronization problems, each hurdle presents an opportunity for learning and improvement.

In conclusion, creating a custom kernel and implementing basic functionalities like printing "Hello, World!" and handling keyboard input is not just an academic exercise; it's a journey of exploration and discovery. It equips developers with invaluable knowledge and skills, paving the way for deeper insights into operating system internals and laying the foundation for more ambitious projects in system software development