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

Go, change the world®

Experiential Learning

Kernel Development in C

Submitted By

Samarth D Gothe – 1RV22CS173

Sathwik Chandra-1RV22CS179

Submitted in

partial fulfilment for the
award of degreeof

BACHELOR OF ENGINEERING
in
Computer Science and Engineering,

Course: Operating Systems – CS235AI

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

2023-24

CONTENTS:

PROBLEM STATEMENT:

Create a basic operating system kernel with bootloader initialization, display output, and keyboard input handling. Test and validate the kernel using emulation tools like QEMU.

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. Next, we delve into kernel development, emphasizing hardware interaction, memory management, and display output via the Visual Graphics Array (VGA).

The report further examines keyboard input handling, showcasing port I/O operations to capture user keystrokes. Testing and integration methods, including emulation with QEMU, are discussed to validate kernel functionality.

In summary, this report 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.**QEMU**

(Quick Emulator [3]) is a free and open-source emulator. It emulates a computer's processor through dynamic binary translation and provides a set of different hardware and device models for the machine, enabling it to run a variety of guest operating systems. It can interoperate with Kernel-based Virtual Machine (KVM) to run virtual machines at near-native speed. QEMU can also do emulation for user-level processes, allowing applications compiled for one architecture to run on another.

QEMU supports the emulation of various architectures, including x86, ARM, PowerPC, RISC-V, and others. QEMU can save and restore the state of the virtual machine with all programs running. Guest operating systems do not need patching in order to run inside QEMU.

5. **BOOTLOADER**

A bootloader is a small program or piece of code that is executed when a computer system is powered on or restarted. Its primary purpose is to initialize the hardware components of the computer and load the operating system (OS) or other essential programs into the computer's memory (RAM) so that the system can start up and become operational. Essentially, the bootloader acts as the bridge between the hardware and software layers of a computer system during the boot process.

METHODOLOGY:

1.Creating your own kernel involves several steps and requires a **good understanding of computer architecture, operating systems principles, and programming languages like C and assembly**.

2.**Set Objectives and Requirements**: Define the goals and requirements for your kernel. Consider factors like supported hardware platforms, desired features, performance goals, and target audience.

3.**Study Operating System Concepts**: Familiarize yourself with operating system concepts such as process management, memory management, file systems, device drivers, and system calls. Understanding these concepts is essential for designing and implementing a kernel.

4.**Choose Development Environment**: Decide on the development environment and tools you'll use. Common choices include GCC for compiling C code, GDB for debugging, and tools like QEMU or Bochs for testing your kernel.

5.**Start Small**: Begin by creating a minimalistic kernel that performs basic tasks such as printing text to the screen and handling interrupts. This allows you to understand the boot process, memory layout, and basic kernel architecture.

6.**Implement Bootloader Integration**: Write or integrate a bootloader (e.g., GRUB) that loads your kernel into memory and transfers control to it. Learn about the bootloader's requirements and ensure that your kernel complies with them.

7.**Testing and Debugging**: Test your kernel thoroughly using emulators like QEMU or virtual machines. Use debugging tools to identify and fix issues in your kernel code.

8.**Documentation and Community**: Document your kernel's design, architecture, and implementation details. Share your work with the community, seek feedback, and collaborate with other developers working on similar projects.

9.**Iterate and Improve**: Continuously iterate on your kernel, adding new features, optimizing performance, and addressing bugs and security vulnerabilities. Learn from your experiences and improve your design and implementation skills over time.

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.

SYSTEM ARCHITECHTURE:

x86 architecture refers to a family of instruction set architectures (ISAs) developed by Intel and later adopted by other processor manufacturers such as AMD. It has been the dominant architecture for personal computers and servers since the 1980s. Here's a brief overview:

History: The x86 architecture traces its roots back to Intel's 8086 microprocessor, released in 1978. It was a 16-bit processor designed as an improvement over Intel's earlier 8080 and 8085 processors Subsequent iterations, such as the 80286, 80386, and 80486, introduced various enhancements, including wider data paths and support for virtual memory.

Key Features:

Complex Instruction Set Computer (CISC): x86 architecture is known for its complex instruction set, which includes a wide range of instructions for performing various tasks. This richness in instructions allows for more operations to be performed directly by the hardware, reducing the need for software emulation.

Segmented Memory Model: In the original x86 architecture, memory addressing was based on a segmented memory model, dividing memory into segments of fixed size. This model was later extended to support flat memory addressing in protected mode.

Protected Mode and Virtual Memory: x86 processors support protected mode, which provides features like memory protection, multitasking, and virtual memory. These features enable modern operating systems to provide robust memory management and process isolation.

SMP and Multicore Support: x86 architecture supports symmetric multiprocessing (SMP) and multicore processors, allowing multiple processor cores to execute instructions concurrently. This capability enhances system performance and scalability.

Continued Evolution: The x86 architecture continues to evolve with advancements in processor technology. Modern x86 processors feature multiple cores, advanced vector processing units, hardware-level security features, and power-efficient designs.

Overall, the x86 architecture has played a significant role in the evolution of computing, powering a wide range of devices from desktops and laptops to servers and data centers. Its compatibility, performance, and versatility have made it a dominant force in the computing industry for decades.

SOURCE CODE:

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"

 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);
  }
}
 //initialize vga buffer
void init_vga(uint8 fore_color, uint8 back_color)
{
  vga_buffer = (uint16*)VGA_ADDRESS;  //point vga_buffer pointer to
VGA_ADDRESS
  clear_vga_buffer(&vga_buffer, fore_color, back_color);  //clear buffer
}

void kernel_entry()
{
  //first init vga with fore & back colors
  init_vga(WHITE, BLACK);

  //assign each ASCII character to video buffer
  //you can change colors here
  vga_buffer[0] = vga_entry('H', WHITE, BLACK);
  vga_buffer[1] = vga_entry('e', WHITE, BLACK);
```

```
  vga_buffer[2] = vga_entry('l', WHITE, BLACK);
  vga_buffer[3] = vga_entry('l', WHITE, BLACK);
  vga_buffer[4] = vga_entry('o', WHITE, BLACK);
  vga_buffer[5] = vga_entry(' ', WHITE, BLACK);
  vga_buffer[6] = vga_entry('W', WHITE, BLACK);
  vga_buffer[7] = vga_entry('o', WHITE, BLACK);
  vga_buffer[8] = vga_entry('r', WHITE, BLACK);
  vga_buffer[9] = vga_entry('l', WHITE, BLACK);
  vga_buffer[10] = vga_entry('d', WHITE, BLACK);
}
```

3. Tic Tac Toe.c

```
#include "kernel.h"
#include "keyboard.h"
#include "utils.h"
#include "types.h"
#include "box.h"
#include "tic_tac_toe.h"

#define PLAYER_1 1
#define PLAYER_2 2

uint8 grid[3][3];
uint8 row = 0, col = 0;
uint8 turn = PLAYER_1;

uint16 player_1_moves = 0;
```

```
uint16 player_2_moves = 0;

uint16 grid_inner_box_x = 30;
uint16 grid_inner_box_y = 2;
uint8 player_1_cell_color = BRIGHT_RED;
uint8 player_2_cell_color = BRIGHT_BLUE;

bool error = FALSE;

void update_cells()
{
  if(grid[0][0] == PLAYER_1){
    fill_box(NULL, 30, 2, 10, 5, player_1_cell_color);
  }else if(grid[0][0] == PLAYER_2){
    fill_box(NULL, 30, 2, 10, 5, player_2_cell_color);
  }
  if(grid[0][1] == PLAYER_1){
    fill_box(NULL, 43, 2, 10, 5, player_1_cell_color);
  }else if(grid[0][1] == PLAYER_2){
    fill_box(NULL, 43, 2, 10, 5, player_2_cell_color);
  }
  if(grid[0][2] == PLAYER_1){
    fill_box(NULL, 56, 2, 10, 5, player_1_cell_color);
  }else if(grid[0][2] == PLAYER_2){
    fill_box(NULL, 56, 2, 10, 5, player_2_cell_color);
  }
```

```
if(grid[1][0] == PLAYER_1){
  fill_box(NULL, 30, 9, 10, 5, player_1_cell_color);
}else if(grid[1][0] == PLAYER_2){
  fill_box(NULL, 30, 9, 10, 5, player_2_cell_color);
}
if(grid[1][1] == PLAYER_1){
  fill_box(NULL, 43, 9, 10, 5, player_1_cell_color);
}else if(grid[1][1] == PLAYER_2){
  fill_box(NULL, 43, 9, 10, 5, player_2_cell_color);
}
if(grid[1][2] == PLAYER_1){
  fill_box(NULL, 56, 9, 10, 5, player_1_cell_color);
}else if(grid[1][2] == PLAYER_2){
  fill_box(NULL, 56, 9, 10, 5, player_2_cell_color);
}
if(grid[2][0] == PLAYER_1){
  fill_box(NULL, 30, 16, 10, 5, player_1_cell_color);
}else if(grid[2][0] == PLAYER_2){
  fill_box(NULL, 30, 16, 10, 5, player_2_cell_color);
}
if(grid[2][1] == PLAYER_1){
  fill_box(NULL, 43, 16, 10, 5, player_1_cell_color);
}else if(grid[2][1] == PLAYER_2){
  fill_box(NULL, 43, 16, 10, 5, player_2_cell_color);
}
if(grid[2][2] == PLAYER_1){
```

```
   fill_box(NULL, 56, 16, 10, 5, player_1_cell_color);
  }else if(grid[2][2] == PLAYER_2){
   fill_box(NULL, 56, 16, 10, 5, player_2_cell_color);
  }
}


void draw_game_board()
{

  draw_box(BOX_SINGLELINE, 28, 1, 38, 20, WHITE, BLACK);

  draw_box(BOX_SINGLELINE, 28, 1, 12, 6, WHITE, BLACK);
  draw_box(BOX_SINGLELINE, 41, 1, 12, 6, WHITE, BLACK);
  draw_box(BOX_SINGLELINE, 54, 1, 12, 6, WHITE, BLACK);

  draw_box(BOX_SINGLELINE, 28, 8, 12, 6, WHITE, BLACK);
  draw_box(BOX_SINGLELINE, 41, 8, 12, 6, WHITE, BLACK);
  draw_box(BOX_SINGLELINE, 54, 8, 12, 6, WHITE, BLACK);

  draw_box(BOX_SINGLELINE, 28, 15, 12, 6, WHITE, BLACK);
  draw_box(BOX_SINGLELINE, 41, 15, 12, 6, WHITE, BLACK);
  draw_box(BOX_SINGLELINE, 54, 15, 12, 6, WHITE, BLACK);

  update_cells();
```

```
fill_box(NULL, grid_inner_box_x, grid_inner_box_y, 10, 5, WHITE);

draw_box(BOX_SINGLELINE, 0, 2, 18, 3, GREY, BLACK);

gotoxy(0, 0);
print_color_string("Tic-Tac-Toe", YELLOW, BLACK);

gotoxy(1, 3);
print_color_string("Player 1 Moves: ", BRIGHT_RED, BLACK);
print_int(player_1_moves);

gotoxy(1, 5);
print_color_string("Player 2 Moves: ", BRIGHT_BLUE, BLACK);
print_int(player_2_moves);

gotoxy(1, 7);
print_color_string("Turn: ", CYAN, BLACK);
gotoxy(8, 7);
if(turn == PLAYER_1){
  print_color_string("Player 1", BRIGHT_CYAN, BLACK);
}else{
  print_color_string("Player 2", BRIGHT_CYAN, BLACK);
}

draw_box(BOX_SINGLELINE, 0, 9, 18, 8, GREY, BLACK);
```

```
gotoxy(1, 9);
print_color_string("Keys", WHITE, BLACK);

gotoxy(1, 11);
print_color_string("Arrows", WHITE, BLACK);

gotoxy(12, 10);
print_char(30);

gotoxy(10, 11);
print_char(17);

gotoxy(14, 11);
print_char(16);

gotoxy(12, 12);
print_char(31);

gotoxy(1, 14);
print_color_string("Spacebar to Select", WHITE, BLACK);
gotoxy(1, 16);
print_color_string("Mov White Box", GREY, BLACK);
gotoxy(1, 17);
print_color_string(" to select cell", GREY, BLACK);

if(error == TRUE){
```

```
    gotoxy(1, 20);
    print_color_string("Cell is already selected", RED, BLACK);
    error = FALSE;
  }
}

int get_winner()
{
  int winner = 0;
  int i;
  //each row
  for(i = 0; i < 3; i++){
    if((grid[i][0] & grid[i][1] & grid[i][2]) == PLAYER_1){
      winner = PLAYER_1;
      break;
    }else if((grid[i][0] & grid[i][1] & grid[i][2]) == PLAYER_2){
      winner = PLAYER_2;
      break;
    }
  }
  //each column
  if(winner == 0){
    for(i = 0; i < 3; i++){
      if((grid[0][i] & grid[1][i] & grid[2][i]) == PLAYER_1){
        winner = PLAYER_1;
        break;
```

```
    }else if((grid[0][i] & grid[1][i] & grid[2][i]) == PLAYER_2){
      winner = PLAYER_2;
      break;
    }
   }
  }

  if(winner == 0){
   if((grid[0][0] & grid[1][1] & grid[2][2]) == PLAYER_1)
     winner = PLAYER_1;
   else if((grid[0][0] & grid[1][1] & grid[2][2]) == PLAYER_2)
     winner = PLAYER_2;
   if((grid[2][0] & grid[1][1] & grid[0][2]) == PLAYER_1)
     winner = PLAYER_1;
   else if((grid[2][0] & grid[1][1] & grid[0][2]) == PLAYER_2)
     winner = PLAYER_2;
  }

  return winner;
}

void restore_game_data_to_default()
{
  uint8 i,j;
  for(i = 0; i < 3; i++){
   for(j = 0; j < 3; j++){
```

```
      grid[i][j] = 0;
    }
  }
  row = 0;
  col = 0;
  turn = PLAYER_1;

  player_1_moves = 0;
  player_2_moves = 0;

  grid_inner_box_x = 30;
  grid_inner_box_y = 2;
}

void launch_game()
{
  byte keycode = 0;
  restore_game_data_to_default();

  draw_game_board();

  do{
    keycode = get_input_keycode();
    switch(keycode){

      case KEY_RIGHT :
```

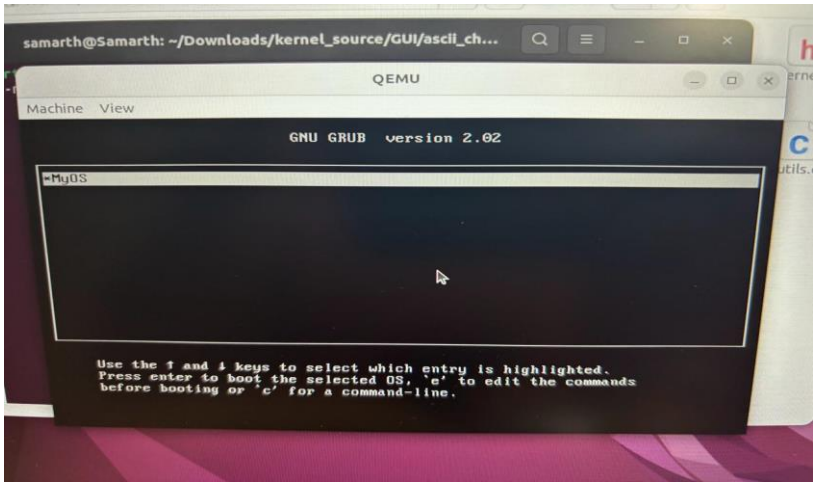
```
  if(grid_inner_box_x <= 43){
    grid_inner_box_x += 13;
    col++;
  }
  break;

case KEY_LEFT :
  if(grid_inner_box_x >= 43){
    grid_inner_box_x -= 13;
    col--;
  }else{
    grid_inner_box_x = 30;
    col = 0;
  }
  break;

case KEY_DOWN :
  if(grid_inner_box_y <= 9){
    grid_inner_box_y += 7;
    row++;
  }
  break;

case KEY_UP :
  if(grid_inner_box_y >= 9){
    grid_inner_box_y-= 7;
```

```
      row--;
    }
    break;

  case KEY_SPACE :
    if(grid[row][col] > 0)
      error = TRUE;

    if(turn == PLAYER_1){
      grid[row][col] = PLAYER_1;
      player_1_moves++;
      turn = PLAYER_2;
    }else if(turn == PLAYER_2){
      grid[row][col] = PLAYER_2;
      player_2_moves++;
      turn = PLAYER_1;
    }
    break;
}
clear_screen(WHITE, BLACK);
draw_game_board();
if(player_1_moves == 3 && player_2_moves == 3){
  if(get_winner() == PLAYER_1){
    draw_box(BOX_DOUBLELINE, 3, 20, 16, 1, BRIGHT_GREEN, BLACK);
    gotoxy(6, 21);
    print_color_string("Player 1 Wins", BRIGHT_GREEN, BLACK);
```

```
      }else if(get_winner() == PLAYER_2){
        draw_box(BOX_DOUBLELINE, 3, 20, 16, 1, BRIGHT_GREEN, BLACK);
        gotoxy(6, 21);
        print_color_string("Player 2 Wins", BRIGHT_GREEN, BLACK);
      }else{
        draw_box(BOX_DOUBLELINE, 3, 20, 16, 1, CYAN, BLACK);
        gotoxy(6, 21);
        print_color_string("No one Wins", BRIGHT_CYAN, BLACK);
      }
    }
    if(player_1_moves + player_2_moves == 9)
      return;

    //change sleep value if game is working so fast or slow
    sleep(0x02FFFFFF);
  }while(keycode > 0);
}
```

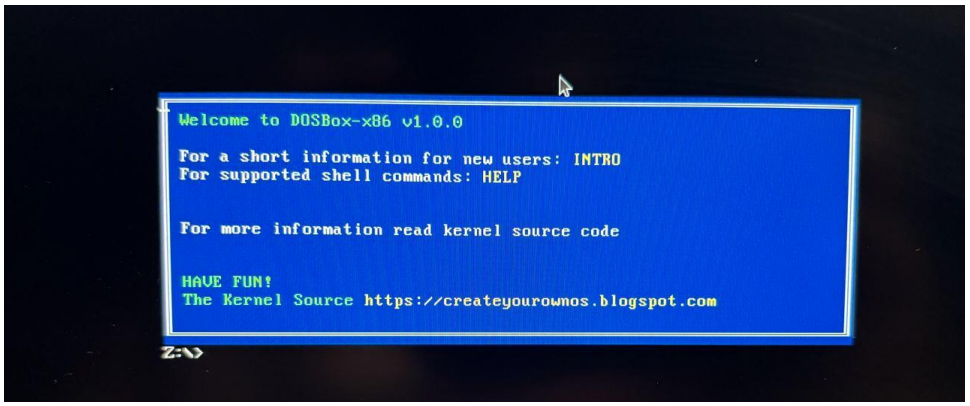
OUTPUT:

1.GNU Grub

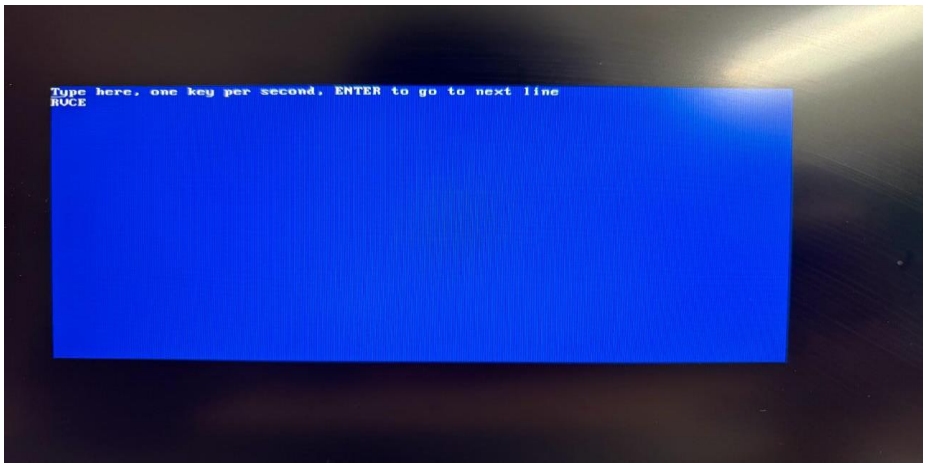


2.Box Drawing GUI

3.DOSBox



4.Keyword



5.Tic-Tac Toe Game