 ****

**Experiential Learning**

**Create your own Kernel Implementation**

**Submitted By**

**Suyash Alva – 1RV22CS212**

**Vansh Goel - 1RV22CS223**

***Submitted in***

***partial fulfillment for the award of degree of***

**BACHELOR OF ENGINEERING**

**in**

**Computer Science and Engineering,**

**Course: Operating Systems – CS235AI DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

**2023-24**

**CONTENTS:**

· Problem Statement

· Introduction

· Relevant Operating Systems Concepts

· Methodology

· System Architecture

· Source Code

· Output

**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 constructing agent given as a component of the GNU Compiler Collection (GCC). It is a part of the GNU toolchain utilized for gathering programs written in dialects like C, C++, and Fortran, into machine code that can be executed by a PC's processor. The GNU Assembler deciphers low level computing construct code, which is a low-level programming language that intently relates to the machine code guidelines grasped by the central processor, into parallel machine code that the PC can execute straightforwardly. Low level computing construct gives an intelligible portrayal of machine guidelines, permitting software engineers to compose code that cooperates straightforwardly with the equipment of a PC framework.

2. GNU/Linux: it is utilized to allude to the blend of the Linux bit with the GNU working framework, created by the Free Software Foundation (FSF) and its pioneer, Richard Stallman. GNU/Linux appropriations come in different flavors, like Ubuntu, Fedora, Debian, and CentOS, every one of which might incorporate different programming bundles and designs, however all depend on the Linux part and the GNU userland devices. The mix of the Linux bit with the GNU working framework and different parts has come about in a strong, flexible, and broadly utilized working framework that powers everything from servers and PCs to implanted frameworks and cell phones. In addition, GNU/Linux is well-known for its stability, safety, and wide selection of free and open-source software.

3. GRUB-MKRESCUE: it is a command-line utility used in the GNU GRUB (Grand Unified Bootloader) bootloader system. Its essential capability is to make a bootable ISO picture that contains the GRUB bootloader alongside determined design records and bootable pieces. This ISO picture can be copied onto a Disc/DVD or kept in touch with a USB drive to make a bootable media. grub-mkrescue permits clients to make salvage circles or establishment media for their working frameworks, giving a helpful method for booting into a framework or perform framework recuperation undertakings. Also, it's generally expected utilized during the time spent making custom Linux disseminations or live Albums/DVDs. Generally speaking, grub-mkrescue is an incredible asset for making bootable media with GRUB bootloader usefulness.

4. QEMU (Quick Emulator [3]): it is a free and open-source emulator. Through dynamic binary translation, it mimics the processor of a computer and provides a variety of hardware and device models for the machine, allowing it to run a variety of guest operating systems. It can interoperate with Part based Virtual Machine (KVM) to run virtual machines at close local speed. QEMU can likewise imitate client level cycles, permitting applications arranged for one engineering to run on another. QEMU supports the imitating of different structures, including x86, ARM, PowerPC, RISC-V, and others. QEMU can save and reestablish the condition of the virtual machine with all projects running. Visitor working frameworks don't require fixing to run inside QEMU.

5. Bootloader: A bootloader is a little program or piece of code that is executed when a PC framework is turned on or restarted. Its basic role is to introduce the equipment parts of the PC and burden the working framework (operating system) or other fundamental projects into the PC's memory (Smash) so the framework can fire up and become functional. During the boot process, the bootloader basically serves as a link between a computer system's software and hardware layers.

**METHODOLOGY:**

1. Setting Up the Development Environment:Install the necessary development tools, including a compiler (such as GCC), an assembler (like NASM or GNU as), and a linker (like GNU ld). Set up a virtual machine or an emulator environment for testing the kernel. Tools like QEMU or VirtualBox are commonly used for this purpose.

2. Understanding GRUB Configuration: GRUB (GRand Unified Bootloader) is a commonly used bootloader for x86 systems. Learn how to configure GRUB to boot your kernel. This involves creating or modifying the GRUB configuration file (usually named grub.cfg) to specify the kernel's location and parameters.

3. Bootloader Initialization: GRUB initializes the system and loads the kernel into memory.Understand how GRUB sets up the environment for the kernel, including setting up the Global Descriptor Table (GDT), enabling protected mode, and transitioning control to the kernel's entry point.

4. Kernel Entry Point:Define the entry point of your kernel code. This is typically where execution begins after control is transferred from the bootloader. Initialize essential data structures and set up the environment necessary for kernel execution.

5. Hardware Interaction:Use x86 assembly language or low-level C code to interact with hardware devices. Implement routines to communicate with the keyboard controller and read input from the keyboard buffer. Understand the PS/2 protocol commonly used by keyboards on x86 systems.

6. Interrupt Handling:Configure interrupt handling to respond to keyboard interrupts.Set up the Interrupt Descriptor Table (IDT) to handle keyboard interrupts (IRQ1). Write interrupt service routines (ISRs) to handle keyboard interrupts and process incoming keystrokes.

7. Buffering and Input Processing: Implement a buffer to store keyboard input temporarily.Process incoming keystrokes to convert scan codes into ASCII characters or other representations. Handle special keys (e.g., function keys, control keys) as needed.

8. Kernel Output: Optionally, implement functionality to display output on the screen, allowing feedback or interaction with the user.

Implement basic text-mode output routines or use BIOS or VESA framebuffer for graphics output.

9. Testing and Debugging: Test the kernel in a virtual machine or emulator environment.Use debugging tools such as GDB, QEMU's built-in debugger, or printf-style debugging to identify and fix issues

**SYSTEM ARCHITECTURE:**

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. **tic\_tac\_toe.c:**

#include "kernel.h"

#include "utils.h"

#include "keyboard.h"

#include "tic\_tac\_toe.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};

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 clear\_screen(uint8 fore\_color, uint8 back\_color)

{

clear\_vga\_buffer(&vga\_buffer, fore\_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\_color\_string(char \*str, uint8 fore\_color, uint8 back\_color)

{

uint32 index = 0;

uint8 fc, bc;

fc = g\_fore\_color;

bc = g\_back\_color;

g\_fore\_color = fore\_color;

g\_back\_color = back\_color;

while(str[index]){

print\_char(str[index]);

index++;

}

g\_fore\_color = fc;

g\_back\_color = bc;

}

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));

}

byte get\_input\_keycode()

{

byte keycode = 0;

while((keycode = inb(KEYBOARD\_PORT)) != 0){

if(keycode > 0)

return keycode;

}

return keycode;

}

/\*

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 gotoxy(uint16 x, uint16 y)

{

vga\_index = 80\*y;

vga\_index += x;

}

void kernel\_entry()

{

byte ans = KEY\_Y;

init\_vga(WHITE, BLACK);

while(ans == KEY\_Y){

clear\_screen(WHITE, BLACK);

launch\_game();

sleep(0x02FFFFFF);

clear\_screen(WHITE, BLACK);

print\_string("Do you want to play again?(y/n) : ");

ans = get\_input\_keycode();

sleep(0x04FFFFFF);

if(ans == KEY\_Y){

continue;

}else if(ans == KEY\_N){

print\_new\_line();

print\_string("Thank you for playing!");

break;

}else{

print\_new\_line();

print\_string("Invalid choice!");

print\_new\_line();

print\_string("Exited ...");

break;

}

}

}

1. **box\_demo\_kernel.c:**

#include "kernel.h"

#include "utils.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};

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\_color\_string(char \*str, uint8 fore\_color, uint8 back\_color)

{

uint32 index = 0;

uint8 fc, bc;

fc = g\_fore\_color;

bc = g\_back\_color;

g\_fore\_color = fore\_color;

g\_back\_color = back\_color;

while(str[index]){

print\_char(str[index]);

index++;

}

g\_fore\_color = fc;

g\_back\_color = bc;

}

void print\_int(int num)

{

char str\_num[digit\_count(num)+1];

itoa(num, str\_num);

print\_string(str\_num);

}

uint16 get\_box\_draw\_char(uint8 chn, uint8 fore\_color, uint8 back\_color)

{

uint16 ax = 0;

uint8 ah = 0;

ah = back\_color;

ah <<= 4;

ah |= fore\_color;

ax = ah;

ax <<= 8;

ax |= chn;

return ax;

}

void gotoxy(uint16 x, uint16 y)

{

vga\_index = 80\*y;

vga\_index += x;

}

void draw\_generic\_box(uint16 x, uint16 y,

uint16 width, uint16 height,

uint8 fore\_color, uint8 back\_color,

uint8 topleft\_ch,

uint8 topbottom\_ch,

uint8 topright\_ch,

uint8 leftrightside\_ch,

uint8 bottomleft\_ch,

uint8 bottomright\_ch)

{

uint32 i;

//increase vga\_index to x & y location

vga\_index = 80\*y;

vga\_index += x;

//draw top-left box character

vga\_buffer[vga\_index] = get\_box\_draw\_char(topleft\_ch, fore\_color, back\_color);

vga\_index++;

//draw box top characters, -

for(i = 0; i < width; i++){

vga\_buffer[vga\_index] = get\_box\_draw\_char(topbottom\_ch, fore\_color, back\_color);

vga\_index++;

}

//draw top-right box character

vga\_buffer[vga\_index] = get\_box\_draw\_char(topright\_ch, fore\_color, back\_color);

// increase y, for drawing next line

y++;

// goto next line

vga\_index = 80\*y;

vga\_index += x;

//draw left and right sides of box

for(i = 0; i < height; i++){

//draw left side character

vga\_buffer[vga\_index] = get\_box\_draw\_char(leftrightside\_ch, fore\_color, back\_color);

vga\_index++;

//increase vga\_index to the width of box

vga\_index += width;

//draw right side character

vga\_buffer[vga\_index] = get\_box\_draw\_char(leftrightside\_ch, fore\_color, back\_color);

//goto next line

y++;

vga\_index = 80\*y;

vga\_index += x;

}

//draw bottom-left box character

vga\_buffer[vga\_index] = get\_box\_draw\_char(bottomleft\_ch, fore\_color, back\_color);

vga\_index++;

//draw box bottom characters, -

for(i = 0; i < width; i++){

vga\_buffer[vga\_index] = get\_box\_draw\_char(topbottom\_ch, fore\_color, back\_color);

vga\_index++;

}

//draw bottom-right box character

vga\_buffer[vga\_index] = get\_box\_draw\_char(bottomright\_ch, fore\_color, back\_color);

vga\_index = 0;

}

void draw\_box(uint8 boxtype,

uint16 x, uint16 y,

uint16 width, uint16 height,

uint8 fore\_color, uint8 back\_color)

{

switch(boxtype){

case BOX\_SINGLELINE :

draw\_generic\_box(x, y, width, height,

fore\_color, back\_color,

218, 196, 191, 179, 192, 217);

break;

case BOX\_DOUBLELINE :

draw\_generic\_box(x, y, width, height,

fore\_color, back\_color,

201, 205, 187, 186, 200, 188);

break;

}

}

void fill\_box(uint8 ch, uint16 x, uint16 y, uint16 width, uint16 height, uint8 color)

{

uint32 i,j;

for(i = 0; i < height; i++){

//increase vga\_index to x & y location

vga\_index = 80\*y;

vga\_index += x;

for(j = 0; j < width; j++){

vga\_buffer[vga\_index] = get\_box\_draw\_char(ch, 0, color);

vga\_index++;

}

y++;

}

}

void kernel\_entry()

{

const char\*str = "Box Demo";

init\_vga(WHITE, BLACK);

gotoxy((VGA\_MAX\_WIDTH/2)-strlen(str), 1);

print\_color\_string("Box Demo", WHITE, BLACK);

draw\_box(BOX\_DOUBLELINE, 0, 0, BOX\_MAX\_WIDTH, BOX\_MAX\_HEIGHT, BRIGHT\_GREEN, BLACK);

draw\_box(BOX\_SINGLELINE, 5, 3, 20, 5, YELLOW, BLACK);

gotoxy(10, 6);

print\_color\_string("Hello World", BRIGHT\_RED, BLACK);

// NULL for only to fill colors, or provide any other character

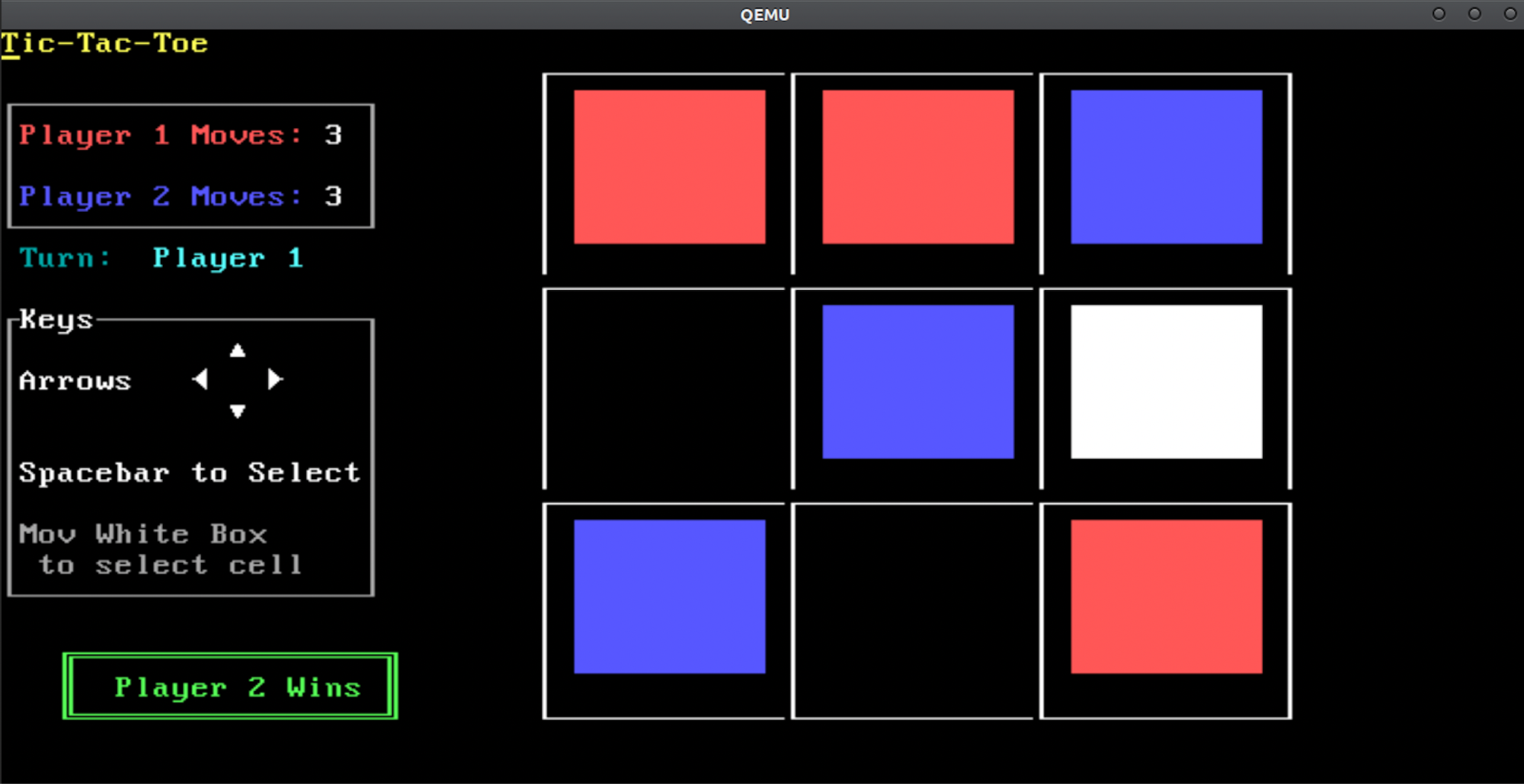
fill\_box(NULL, 36, 5, 30, 10, RED);

fill\_box(1, 6, 16, 30, 4, GREEN);

draw\_box(BOX\_DOUBLELINE, 6, 16, 28, 3, BLUE, GREEN);

}

**OUTPUT:**

****

****