RV College of Engineering®

Mysore Road, RV Vidyaniketan Post,
Bengaluru - 560059, Karnataka, India

Experiential Learning

Go, change the world®

Kernel Development in C

Implementation of keyboard and Ping pong game

Submitted By

Siri A Bhat – 1RV22CS197

Yash Lohia – 1RV22CS237

Submitted to Prof Jyothi SHetty

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

2023-2024

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:

KERNEL - The kernel serves as the core component of an operating system (OS), acting as the central hub that manages system resources and facilitates communication between hardware and software components. It represents the heart of the operating system, responsible for essential tasks such as process management, memory management, device drivers, and system calls. Essentially, the kernel serves as the intermediary between user applications and the underlying hardware, providing a layer of abstraction that enables programmers to interact with system resources in a standardized and efficient manner.

BOOTLOADER -A bootloader serves as a critical component of computer systems, acting as the initial program that loads the operating system into memory during the system startup process. Essentially, it bridges the gap between the hardware of the computer and the software, ensuring the smooth transition from the hardware initialization phase to the execution of the operating system. Upon powering on the computer, the bootloader is typically the first code executed, residing in a special location known as the boot sector of the storage device. Its primary function is to locate the operating system kernel on the storage device, load it into memory, and transfer control to the kernel to begin the operating system's execution. Bootloaders come in various forms, ranging from simple programs embedded in read-only memory (ROM) to more sophisticated boot managers capable of managing multiple operating systems or boot configurations. Overall, bootloaders play a crucial role in the bootstrapping process of computer systems, ensuring their proper initialization and enabling the seamless launch of the operating system.

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

SYSTEM ARCHITECTURE:

1) PING PONG GAME

kernel Core (Ping-Pong Game Engine):

- **Process Management**: Manages processes (ping and pong players) including creation, scheduling, and context switching.
- **Memory Management**: Allocates and manages memory (the ping-pong table) for processes.
- **I/O Handling**: Facilitates communication between processes and external devices (the ball).
- **Interrupt Handling**: Handles interrupts (player reactions) from hardware devices and system events.

Process Management (Players):

- **Process Control Block (PCB)**: Data structure representing each process, containing information such as process state, program counter, and memory allocation.
- **Scheduler**: Algorithm for scheduling processes, determining which process to execute next (deciding who serves the ball).
- **Context Switching**: Mechanism for saving and restoring the state of processes during context switches (changing players).

Memory Management (Ping-Pong Table):

- **Memory Allocator**: Allocates and deallocates memory for processes, ensuring memory integrity and protection.
- **Memory Protection**: Enforces memory protection to prevent unauthorized access to memory regions.

I/O Handling (Ball):

- **Device Drivers**: Software components interacting with hardware devices (the ball), handling I/O operations such as reading input and writing output.
- **I/O Scheduler**: Coordinates I/O operations to optimize performance and resource utilization.

Interrupt Handling (Player Reactions):

- **Interrupt Handlers**: Functions responding to hardware interrupts and system events, such as timer interrupts or I/O interrupts.

- **Interrupt Controller**: Manages interrupt signals from hardware devices, prioritizing and routing interrupts to appropriate handlers.

User Interface (Game Interface):
- **Command Line Interface (CLI)**: Simple interface for users to interact with the kernel, issuing commands and viewing system information.
- **System Calls**: Interface allowing user processes to request services from the kernel, such as process creation or I/O operations.

Kernel Initialization and Bootstrapping:
- **Bootloader**: Initial program loaded by the hardware, responsible for loading the kernel into memory and starting its execution.
- **Initialization Routine**: Initializes the kernel and its components at system boot-up, setting up essential data structures and preparing the system for user interaction.

System Services:
- **System Calls Handling**: Kernel functions providing services to user processes, such as file operations, process management, and inter-process communication.
- **System Libraries**: Libraries providing high-level abstractions and utilities for user programs, facilitating development and portability.

2) KEYBOARD IMPLEMENTATION

This architecture provides a high-level overview of the components and interactions involved in implementing a simple kernel using the ping-pong analogy. Each component plays a crucial role in managing system resources, facilitating communication, and ensuring the proper functioning of the kernel.

- keyboard. When a key is pressed, the keyboard driver reads the corresponding keycode from the keyboard hardware and stores it in the input buffer.

Input Processing Module:
- The input processing module within the kernel is responsible for interpreting the keypress events stored in the input buffer. It converts keycodes into characters or higher-level input events and processes them accordingly.

Kernel Services:

- The kernel provides various services and interfaces for interacting with input devices, including the keyboard. These services may include functions for reading input from the keyboard, registering input event handlers, and managing input device configuration.

Application Interface:

- Applications running on top of the kernel interact with the keyboard through system calls or higher-level APIs provided by the kernel. Applications can request keyboard input, register callback functions for handling keypress events, and perform other input-related operations.

System Call Interface:

- The kernel exposes system calls that allow user-space applications to interact with input devices indirectly. For example, applications can use system calls to read input from the keyboard or register event handlers for specific keypress events.

By implementing this system architecture, the kernel can effectively communicate with the keyboard hardware, process incoming keypress events, and provide keyboard input functionality to user-space applications running on the system.

METHODOLOGY:

1) PING_PONG GAME

Implementing a simple kernel using the ping-pong game analogy can be a creative and educational approach. Below is a methodology outlining the steps to develop such a kernel:

Define Kernel Structure: Begin by outlining the structure of the kernel. Define the main components, such as process management, memory management, and I/O handling, analogous to players, the ball, and the playing field in the ping-pong game.
Process Management (Players): Implement process management functionalities to create and manage processes (players) within the kernel. Define data structures to represent processes and develop functions for process creation, scheduling, and context switching.
Memory Management (Playing Field): Develop memory management capabilities to allocate and manage memory (playing field) for processes. Implement functions for memory allocation, deallocation, and protection to ensure the integrity and security of the kernel's memory space.
I/O Handling (Ball): Implement I/O handling functionalities to facilitate communication between processes and external devices (the ball). Develop device drivers to interact with hardware components and handle I/O operations, such as reading input and writing output.
Kernel Initialization: Create an initialization routine to initialize the kernel and its components at system boot-up. This routine sets up essential data structures, initializes hardware devices, and prepares the system for executing user programs.
Main Loop (Game Loop): Implement the main loop of the kernel, analogous to the game loop in the ping-pong game. This loop continuously schedules and executes processes, handles interrupts, and manages system events.
Process Scheduling (Game Strategy): Develop a process scheduling algorithm to determine the order in which processes are executed, similar to the strategy used in the ping-pong game to decide which player serves the ball next.
Interrupt Handling (Player Reactions): Implement interrupt handling mechanisms to respond to hardware interrupts and system events, such as timer interrupts or I/O interrupts. Define interrupt handlers to handle these events and perform appropriate actions.

User Interface (Game Interface): Create a simple user interface to interact with the kernel, allowing users to execute commands, view system information, and interact with processes. This interface serves as the means for users to interact with the kernel, similar to how players interact with the ping-pong game.
Testing and Debugging: Test the kernel thoroughly to ensure proper functionality and reliability. Use debugging tools and techniques to identify and fix any issues or bugs in the implementation.
Documentation and Optimization: Document the kernel's design, implementation, and usage to facilitate understanding and future development. Additionally, optimize the kernel for performance and resource utilization, ensuring efficient operation on target hardware platforms.

By following this methodology, you can develop a simple kernel using the ping-pong game analogy, providing a creative and engaging approach to learning about operating system fundamentals.

2) KEYBOARD IMPLEMENTATION

Implementing a kernel that interacts with a keyboard involves several steps. Here's a methodology for developing such a kernel:

Understanding Keyboard Hardware:

- Gain a thorough understanding of how keyboards interface with computer hardware. Learn about the keyboard controller, scan codes, key matrices, and how keystrokes are transmitted to the computer.

Setting Up the Development Environment:

- Set up a development environment for kernel development. This typically involves installing a toolchain, an emulator or virtual machine for testing, and a debugger.

Initializing the Keyboard Controller:

- Write code to initialize the keyboard controller. This may involve configuring ports, setting up interrupts, and enabling keyboard input.

Implementing Interrupt Handling:

- Set up interrupt handlers to respond to keyboard events. When a key is pressed or released, the keyboard controller generates an interrupt, which the kernel must handle appropriately.

Reading Keycodes:

- Write code to read scan codes or keycodes from the keyboard controller. These codes represent the keys pressed by the user and need to be interpreted by the kernel.

Processing Key Presses:

- Develop logic to process key presses and releases. Determine how the kernel should respond to different key events, such as generating characters, triggering system events, or passing input to user-space applications.

Buffering Input:

- Implement an input buffer to store incoming key presses. This buffer ensures that keystrokes are not lost if the kernel is busy processing other tasks.

Handling Special Keys:

- Handle special keys such as function keys, modifier keys (Shift, Ctrl, Alt), and multimedia keys. Define how these keys should be interpreted and processed by the kernel.

Kernel Services:

- Expose kernel services or system calls for interacting with the keyboard. Define interfaces that allow user-space applications to read keyboard input, register event handlers, and control keyboard behavior.

Testing and Debugging:

- Test the kernel's keyboard functionality extensively using emulators, virtual machines, or physical hardware. Debug any issues related to keyboard input handling, interrupt handling, or buffer management.

Integration with User-space Applications:

- Integrate keyboard input functionality with user-space applications. Develop applications that utilize the kernel's keyboard services for text input, command execution, or other interactions.

Optimization and Performance Tuning:

- Optimize the keyboard input handling routines for efficiency and responsiveness. Minimize latency between key presses and application response to ensure a smooth user experience.

Documentation and Maintenance:

- Document the kernel's keyboard handling mechanisms, APIs, and usage guidelines. Maintain the codebase by addressing bugs, adding features, and keeping it compatible with newer hardware or software environments.

By following this methodology, you can implement a kernel that effectively interacts with a keyboard, enabling user input and interaction in your operating system environment.

SYSTEM CALLS:

`mov` (Move):

- This is an instruction used to move data between memory locations or between memory and registers. In this code, `mov $stackTop, %esp` moves the address of `stackTop` to the stack pointer register `%esp`, effectively setting up the stack.

`call` (Call):

- This instruction is used to call a subroutine or function. In the code, `call kernel_entry` is used to call the `kernel_entry` function, which likely contains the main logic for the kernel.

`hlt` (Halt):

- This instruction halts the CPU until the next interrupt occurs. In this code, it's used in a loop to halt the system indefinitely, creating an infinite loop that effectively stops further execution.

`jmp` (Jump):

- This instruction is used to transfer program control to a different location. In this code, `jmp hltLoop` is used to jump back to the `hltLoop` label, effectively restarting the infinite loop.

Drawing Functions:

- Functions like `draw_string()` and `draw_rect()` are likely defined elsewhere in the codebase and are used to draw text and shapes on the screen. While not system calls in the operating system sense, they serve similar purposes by interacting with the graphics hardware to display visual elements.

Input Handling:

- The `get_input_keycode()` function retrieves keyboard input, presumably from the user, to control the game. This function simulates user input and interacts with input devices, resembling the functionality of system calls related to input/output operations.

Memory Management:

- Although not explicitly shown in this code snippet, memory management functions like `malloc()` and `free()` may be used elsewhere in the codebase to manage dynamic memory allocation. These functions resemble system calls related to memory management in operating systems.

<u>Sleep Function</u>:

- The `sleep()` function is used to introduce a delay in the execution of the program, which is commonly used for animation or timing purposes in games. While not a system call in the operating system sense, it behaves similarly to system calls that suspend program execution for a specified duration.

<u>Clearing the Screen</u>:

- The `clear_screen()` function clears the screen, likely by manipulating the display buffer or interacting with the graphics hardware. Again, while not a system call, it performs a similar function to system calls related to screen management.

<u>Other Functions</u>:

- Functions like `fill_rect()` and `itoa()` serve various purposes within the game, such as drawing rectangles and converting integers to strings. While not system calls, they represent common operations performed in graphics programming and game development.

<u>`inb` Function</u>:

- This function reads a byte from an input port. It uses inline assembly to execute the `inb` assembly instruction, which reads a byte from the specified port. This function directly interacts with hardware I/O ports, which are commonly used for communication with hardware devices such as keyboards or serial ports.

<u>`outb` Function</u>:

- This function writes a byte to an output port. Similar to `inb`, it uses inline assembly to execute the `outb` assembly instruction, which writes a byte to the specified port. It's used for sending data to hardware devices, such as sending commands to a display adapter or writing data to a serial port.

<u>`get_input_keycode` Function</u>:

- This function retrieves a keycode from the keyboard. It continuously polls the keyboard port until a keycode is received, indicating that a key has been pressed. Once a keycode is obtained, it returns the corresponding character. This function interacts directly with the keyboard hardware to retrieve user input.

<u>`wait_for_io` and `sleep` Functions</u>:

- These functions introduce delays or wait for a specified duration. They contain loops that execute `nop` instructions (no operation), effectively causing the CPU to idle for a certain period. While waiting, the CPU is not processing any instructions, allowing other hardware operations to proceed uninterrupted.

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. pong.c

```
#include "pong.h"
// pads position y, will change on keys
uint16 pad_pos_y = 2;
// score count
uint32 score_count = 0;
// initialize game with into
static void init_game()
{
    uint8 b = 0;
```

```
  draw_string(120, 13, BRIGHT_CYAN, "PONG GAME");
   draw_rect(100, 4, 120, 25, BLUE);
   draw_string(10, 50, BRIGHT_MAGENTA, "HOW TO PLAY");
   draw_rect(2, 40, 235, 80, BROWN);
   draw_string(10, 70, BRIGHT_RED, "ARROW KEY UP");
   draw_string(30, 80, WHITE, "TO MOVE BOTH PADS UP");
   draw_string(10, 90, BRIGHT_RED, "ARROW KEY DOWN");
   draw_string(30, 100, WHITE, "TO MOVE BOTH PADS DOWN");
   draw_string(60, 160, BRIGHT_GREEN, "PRESS ENTER TO START");
#ifdef VIRTUALBOX
   sleep(10);
#endif
   while (1)
   {
      b = get_input_keycode();
      sleep(5);
      if (b == KEY_ENTER)
         break;
      b = 0;
   }
   clear_screen();
}
// update score count text
```

```
static void update_score_count()
{
    char str[32];
    itoa(score_count, str);
    draw_string(150, 2, WHITE, str);
}
// if lose then display final score & restart game
static void lose()
{
    uint8 b = 0;
    char str[32];
    itoa(score_count, str);
    clear_screen();
    draw_string(120, 15, BRIGHT_GREEN, "NICE PLAY");
    draw_string(125, 45, WHITE, "SCORE");
    draw_string(180, 45, WHITE, str);
    draw_string(45, 130, YELLOW, "PRESS ENTER TO PLAY AGAIN");
#ifdef VIRTUALBOX
    sleep(10);
#endif
    while (1)
    {
        b = get_input_keycode();
        sleep(5);
        if (b == KEY_ENTER)
            break;
        b = 0;
    }
    score_count = 0;
    clear_screen();
    pong_game();
}
// move both pads simultaneously on pressed keys
void move_pads()
{
    uint8 b;
    // draw both pads
    fill_rect(0, pad_pos_y, PAD_WIDTH, PAD_HEIGHT, YELLOW);
    fill_rect(PAD_POS_X, pad_pos_y, PAD_WIDTH, PAD_HEIGHT, YELLOW);
    b = get_input_keycode();
    // if down key pressed, move both pads down
    if (b == KEY_DOWN)
    {
        if (pad_pos_y < VGA_MAX_HEIGHT - PAD_HEIGHT)
            pad_pos_y = pad_pos_y + PAD_SPEED;
        fill_rect(0, pad_pos_y, PAD_WIDTH, PAD_HEIGHT, YELLOW);
        fill_rect(PAD_POS_X, pad_pos_y, PAD_WIDTH, PAD_HEIGHT, YELLOW);
    }
    // if up key pressed, move both pads up
    else if (b == KEY_UP)
    {
        if (pad_pos_y >= PAD_WIDTH)
            pad_pos_y = pad_pos_y - PAD_SPEED;
        fill_rect(0, pad_pos_y, PAD_WIDTH, PAD_HEIGHT, YELLOW);
        fill_rect(PAD_POS_X, pad_pos_y, PAD_WIDTH, PAD_HEIGHT, YELLOW);
    }
#ifdef VIRTUALBOX
    sleep(1);
#endif
}
void pong_game()
{
    uint16 rect_pos_x = RECT_SIZE + 20;
    uint16 rect_pos_y = RECT_SIZE;
    uint16 rect_speed_x = RECT_SPEED_X;
    uint16 rect_speed_y = RECT_SPEED_Y;

    init_game();
    while (1)
    {
        // add speed values to positions
        rect_pos_x += rect_speed_x;
        rect_pos_y += rect_speed_y;
        // check if position x < left pad position x
        if (rect_pos_x - RECT_SIZE <= PAD_WIDTH + 1)
        {
            // if position of rect is not between left pad position,
            // then lose, bounced rect is not in y range of pad
            if ((rect_pos_y > 0 && rect_pos_y < pad_pos_y) ||
                (rect_pos_y <= VGA_MAX_HEIGHT && rect_pos_y > pad_pos_y +
PAD_HEIGHT))
            {
                lose();
            }
            else
            {
                // set speed x to negative, means move opposite direction
                rect_speed_x = -rect_speed_x;
                // set position x to rect size
                rect_pos_x = PAD_WIDTH + RECT_SIZE;
                // increase score
                score_count++;
            }
        }
        // check if position x >= right pad position x
        else if (rect_pos_x + RECT_SIZE >= PAD_POS_X + RECT_SIZE - 1)
        {
           // in range of y pad position
           if ((rect_pos_y > 0 && rect_pos_y < pad_pos_y) ||
              (rect_pos_y <= VGA_MAX_HEIGHT && rect_pos_y > pad_pos_y +
    PAD_HEIGHT) ||
              (rect_pos_y + RECT_SIZE > 0 && rect_pos_y + RECT_SIZE < pad_pos_y))
           {
              lose();
           }
           else
           {
              // set speed x to negative, means move opposite direction
              rect_speed_x = -rect_speed_x;
              // set position x to minimum of pad position x - rect size
              rect_pos_x = PAD_POS_X - RECT_SIZE;
              // increase score
              score_count++;
           }
        }
        // change rect y position by checking boundries
        if (rect_pos_y - RECT_SIZE <= 0)
        {
            rect_speed_y = -rect_speed_y;
            rect_pos_y = RECT_SIZE;
        }
        else if(rect_pos_y + RECT_SIZE > VGA_MAX_HEIGHT + RECT_SIZE)
        {
            rect_speed_y = -rect_speed_y;
            rect_pos_y = VGA_MAX_HEIGHT - RECT_SIZE;
        }
        // clear screen for repaint
        clear_screen();
        // move pads on keys
        move_pads();
        // update score count
        update_score_count();
        // fill bounced rect
        fill_rect(rect_pos_x - RECT_SIZE, rect_pos_y - RECT_SIZE, RECT_SIZE,
RECT_SIZE, WHITE);
        // change sleep value if running in VirtualBox or on bare metal
        sleep(1);
    }
```

3. kernel.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++;
```

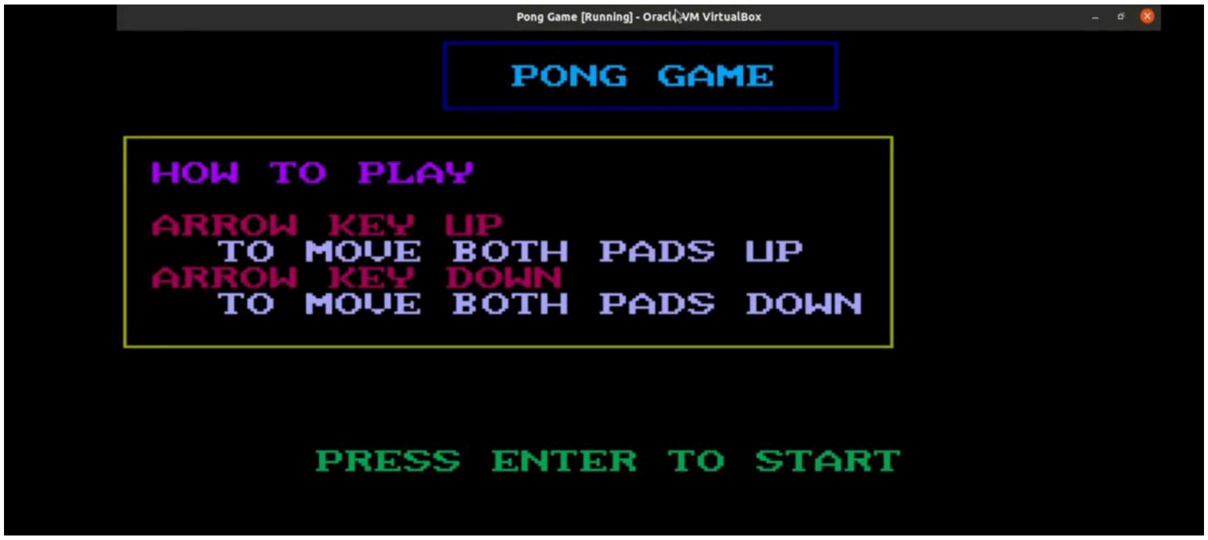
```
}

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;
}
```

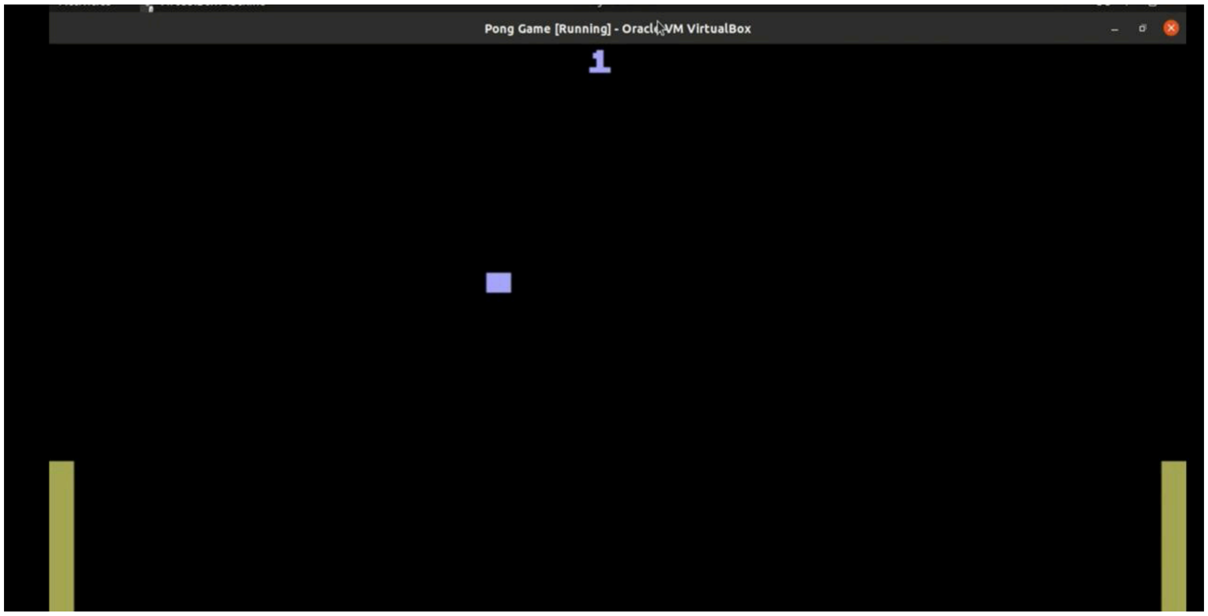
```
/*
keep the cpu busy for doing nothing(nop)
so that io port will not be processed by cpu
here timer can also be used, but lets do this in looping counter
*/
void wait_for_io(uint32 timer_count)
{
  while(1){
    asm volatile("nop");
    timer_count--;
    if(timer_count <= 0)
      break;
    }
}

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();


}
```

OUTPUT:

1) **PING PONG**

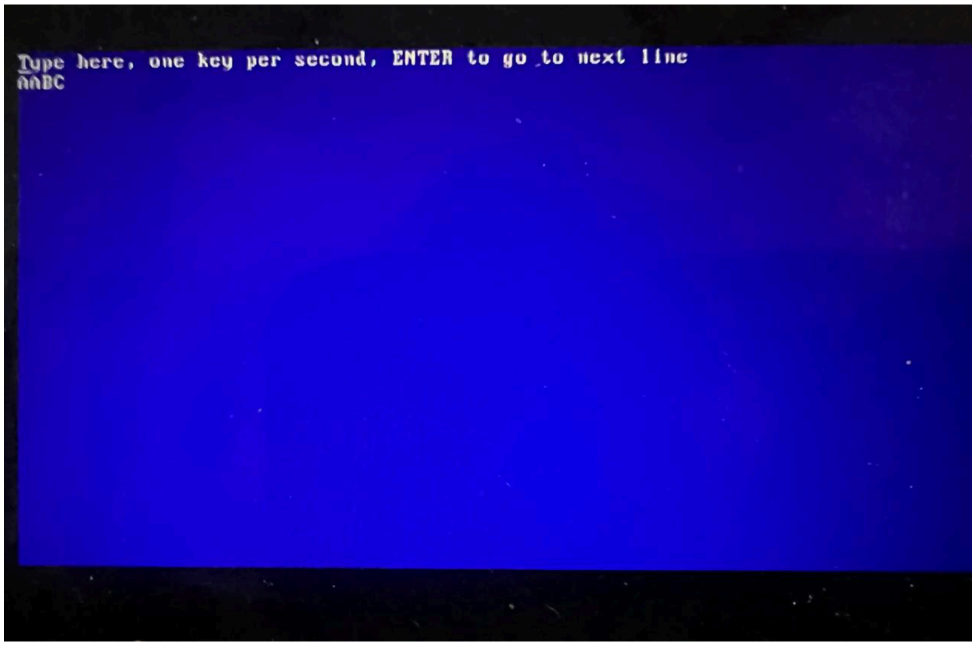


```
Pong Game [Running] - Oracle VM VirtualBox

PONG GAME

HOW TO PLAY

ARROW KEY UP
   TO MOVE BOTH PADS UP
ARROW KEY DOWN
   TO MOVE BOTH PADS DOWN

PRESS ENTER TO START
```



```
Pong Game [Running] - Oracle VM VirtualBox
1
```

2) KEYBOARD



```
Type here, one key per second, ENTER to go to next line
AABC
```

Office on the web Frame

```
, ENTER to go to next line
```

Conclusion:

In conclusion, the process of creating a kernel that incorporates keyboard functionality and a ping pong game involves a meticulous and iterative approach. By understanding the intricacies of keyboard hardware and implementing efficient input handling mechanisms, we can enable users to interact with the system effectively. Additionally, integrating a fun and interactive game like ping pong showcases the versatility and entertainment potential of the kernel.

Throughout the development process, attention to detail, rigorous testing, and optimization are essential to ensure smooth operation and responsiveness. Moreover, documenting the implementation details and maintaining the codebase facilitate future enhancements and troubleshooting.

Ultimately, the creation of such a kernel not only demonstrates technical proficiency but also fosters creativity and innovation in the realm of operating system development. It provides a solid foundation for further exploration and expansion of functionality, paving the way for the development of more sophisticated operating systems and interactive user experiences.