



RV Educational Institutions®
RV College of Engineering®

Autonomous
Institution Affiliated
to Visvesvaraya
Technological
University, Belagavi

Approved by AICTE,
New Delhi, Accredited
By NAAC, Bengaluru
And NBA, New Delhi

Go, change the world

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Implementation Of Kernel With Integrated VGA Array And I/O Controls To Showcase Ping Pong Real Time Game



EL REPORT

OPERATING SYSTEMS

Submitted by,

SNEHIL VUKKUSILA

1RV22CS241

PRANAV DARSHAN

1RV22CS143

Introduction

The Linux kernel is like the boss behind the scenes of your computer, making sure everything works together. It's usually pretty complicated, but this project is like building a fun mini-game inside the boss's office! We'll make a special version of the Linux kernel that lets you play a simple game of ping pong, all within the system itself. This will help us learn more about how the kernel works in a fun and interesting way.

Problem Statement

Design and implement a kernel from scratch for an operating system course project, emphasizing core operating system concepts such as process management, memory management, and I/O operations.

Integrate keyboard input/output functionality and utilize the VGA graphics array for graphical display within the kernel environment

Real-time playing of the Pong game, implementing game logic for paddle movement and ball collision detection.

The objective is to develop a fully functional operating system kernel capable of managing processes, handling keyboard interactions, rendering graphics using VGA support, and facilitating real-time gameplay, showcasing proficiency in kernel development and system-level programming.

Tools & APIS SystemCalls Required

Tools:

GNU/Linux :- Any distribution(Ubuntu/Debian/RedHat etc.).

Assembler :- GNU Assembler(gas) to assemble the assembly language file.

GCC :- GNU Compiler Collection, C compiler. Any version 4, 5, 6, 7, 8 etc.

grub-mkrescue :- Make a GRUB rescue image, this package internally calls the xorriso functionality to build an iso image.

QEMU :- Quick EMUlator to boot our kernel in virtual machine without rebooting the main system.

Executable Kernel Image ISO: The final output of the kernel build process is an executable kernel image

Header Files: Header files (.h) contain declarations

Linker Script: A linker script (.ld file) is used to specify the layout of the kernel image in memory,

Iso Image Generation and Execution

Different APIs and System Calls Utilized:

Kernel Architecture and Design

Understanding the structure and organization of the kernel, including process management, memory management, and device drivers.

Related API/System Calls:

Process Management: fork(), exec(), exit()

Memory Management: malloc(), free()

Device Drivers: open(), read(), write()

Implementation: Design a modular kernel architecture that facilitates the integration of the Ping Pong game, ensuring efficient process and memory management

Graphics Rendering with VGA Buffer

Concept: Utilizing the VGA graphics array buffer for rendering graphical interfaces within the kernel environment.

Related API/System Calls:

open(), read(), and write()

ioctl(): Useful for configuring and controlling device-specific settings, such as manipulating the VGA graphics buffer for drawing game elements.

signal(): Enables setting up signal handlers to handle asynchronous events, such as keyboard interrupts for processing user inputs during gameplay.

VGA Graphics: Direct access to VGA hardware registers for setting display modes, writing pixel data, and manipulating screen buffers.

Implementation: Implement low-level functions to interact with the VGA buffer, enabling the drawing of game elements such as paddles, ball, and score display.

Keyboard Input Handling

Concept: Managing keyboard input for user interactions and game control within the kernel environment.

Related API/System Calls:

kbd_init(): Initializes the keyboard device and sets up the necessary data structures and interrupt handlers for keyboard input.

kbd_read(): Reads input from the keyboard device, translating scan codes into characters or key codes.

Keyboard Input: Directly accessing keyboard hardware registers to read scan codes and handle key presses.

Implementation: Develop functions to read keyboard input asynchronously, translating key presses into game actions (e.g., moving paddles in response to arrow key presses).

Game Logic and Event Handling

Concept: Implementing the core game logic and event handling mechanisms within the kernel environment.

Related API/System Calls:

event_init(): Initializes the game event system, setting up event queues and data structures for managing game events.

event_queue(): Adds a new event to the game event queue, such as paddle movement or ball collision.

event_process(): Processes events from the event queue, updating the game state accordingly.

Event Handling: Custom event loop implementation for managing game state transitions and user inputs.

Implementation: Design and implement the Ping Pong game logic, including ball movement, paddle control, collision detection, and scoring mechanisms.

Relevance To The Course

Input/Output Handling from Keyboard:

The project involves implementing functionality to handle input from the keyboard within the kernel environment.

This includes reading scan codes generated by keystrokes and translating them into meaningful inputs for the game.

By directly interfacing with the keyboard hardware and processing its input at the kernel level, students gain a deeper understanding of low-level input/output handling.

Efficient handling of keyboard input is crucial for providing responsive user interactions in the game, such as moving paddles or controlling game elements.

- **Keyboard Input Handling:**

Conceptual Relevance: Keyboard input handling directly applies concepts learned in the operating systems course, such as interrupt handling and device drivers.

Practical Application: Implementing keyboard input handling in the kernel environment translates theoretical knowledge into real-world scenarios.

Low-Level Interaction: Interfacing with keyboard hardware at the kernel level reinforces understanding of low-level input/output operations and device interactions.

Process Synchronization: Managing keyboard input often involves synchronization mechanisms like semaphores or mutexes, which align with course concepts.

Utilizing VGA Graphics Array:

Integrating VGA support entails interacting with the VGA graphics array to render graphical elements within the kernel environment.

This involves writing pixel data directly to the VGA buffer to display game graphics, such as the game board, paddles, and ball.

VGA graphics within the kernel allows for the creation of a graphical user interface (GUI) for the game, enhancing the user experience and providing visual feedback to players.

- **VGA Graphics Integration:**

Hardware Interaction: Integrating VGA graphics requires understanding of hardware interfacing and memory mapping, concepts taught in the operating systems course.

Kernel-Level Graphics: Developing GUIs within the kernel space reinforces understanding of kernel-level programming and system architecture.

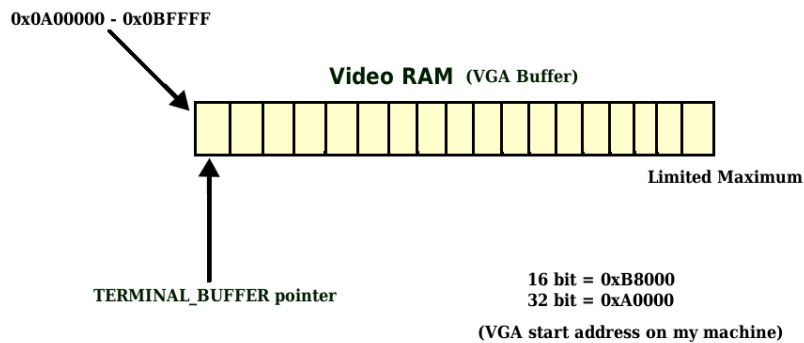
Device Drivers: Treating the VGA graphics array as a device involves concepts similar to those used in developing device drivers, a key topic in the course curriculum.

Interrupt Handling: VGA graphics manipulation may require interrupt handling for efficient screen updates, aligning with course concepts regarding interrupt-driven I/O.

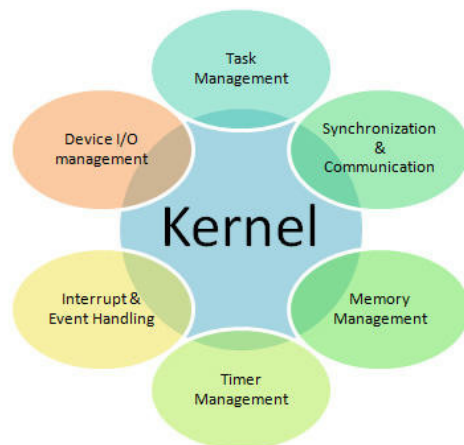
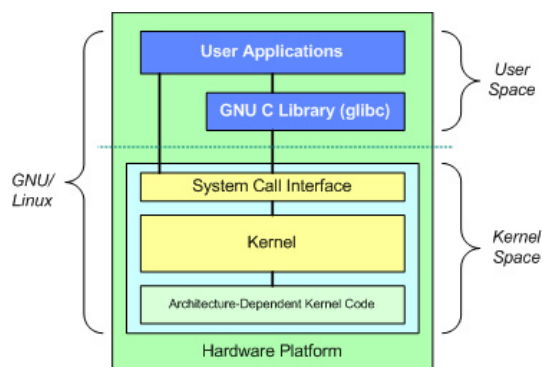
Diagram/Architecture Flowchart

Keyboard code mappings are essentially a way to represent the physical keys on a keyboard in a digital format that can be interpreted by a computer system. Each key press generates a unique code that the system can recognize and respond to. In a kernel implementation for a Ping Pong game, keyboard code mappings are used to capture user input from the keyboard, such as pressing arrow keys to move the paddle or hitting the spacebar to start the game.

Key	KeyCode	Key	KeyCode	Key	KeyCode	Key	KeyCode	Key	KeyCode
A	0x1E	Q	0x10	7	0x08	F2	0x3C	Esc	0x01
B	0x30	R	0x13	8	0x09	F3	0x3D	Home	0x47
C	0x2E	S	0x1F	9	0x0A	F4	0x3E	Insert	0x52
D	0x20	T	0x14	0	0x0B	F5	0x3F	Keypad (5)	0x4C
E	0x12	U	0x16	-	0x0C	F6	0x40	Keypad (*)	0x37
F	0x21	V	0x2F	=	0x0D	F7	0x41	Keypad (-)	0x4A
G	0x22	W	0x11	[0x1A	F8	0x42	Keypad (+)	0x4E
H	0x23	X	0x2D]	0x1B	F9	0x43	Keypad (/)	0x35
I	0x17	Y	0x15	;	0x27	F10	0x44	Left Arrow	0x4B
J	0x24	Z	0x2C	'	0x28	F11	0x45	Page Down	0x51
K	0x25	1	0x02	`	0x29	F12	0x86	Page Up	0x49
L	0x26	2	0x03	\	0x2B	BackSpace	0x0E	Print Screen	0x37
M	0x32	3	0x04	,	0x33	Delete	0x53	Right Arrow	0x4D
N	0x31	4	0x05	.	0x34	DownArrow	0x50	Space Bar	0x39
O	0x18	5	0x06	/	0x35	End	0x4F	Tab	0x0F
P	0x19	6	0x07	F1	0x3B	Enter	0x1C	Up Arrow	0x48



The kernel interacts with the VGA graphics hardware by writing pixel data directly to the VGA memory buffer. This involves manipulating memory addresses corresponding to individual pixels to create and update visual representations of game elements. For example, setting specific pixel values in the buffer can draw lines for the game board or move the paddles and ball across the screen.



VGA Interaction:

The kernel interacts with the VGA graphics array by writing pixel data directly to the VGA memory buffer.

It manipulates memory addresses corresponding to individual pixels to render graphical elements such as the game board, paddles, and ball.

By leveraging VGA support, the kernel provides a graphical user interface for the Pong game, ensuring smooth and responsive rendering of game elements on the screen.

Keyboard Interaction:

The kernel listens for keyboard interrupts to capture user input from the keyboard.

It reads the corresponding key codes generated by key presses and translates them into game commands or actions.

For instance, pressing arrow keys to move the paddles or hitting the spacebar to start the game triggers corresponding actions within the game logic.

Game Logic Integration:

The kernel integrates the VGA graphics and keyboard input/output functionalities with the game logic module.

It communicates game state changes, such as paddle movement or ball collisions, to the VGA graphics for rendering on the screen.

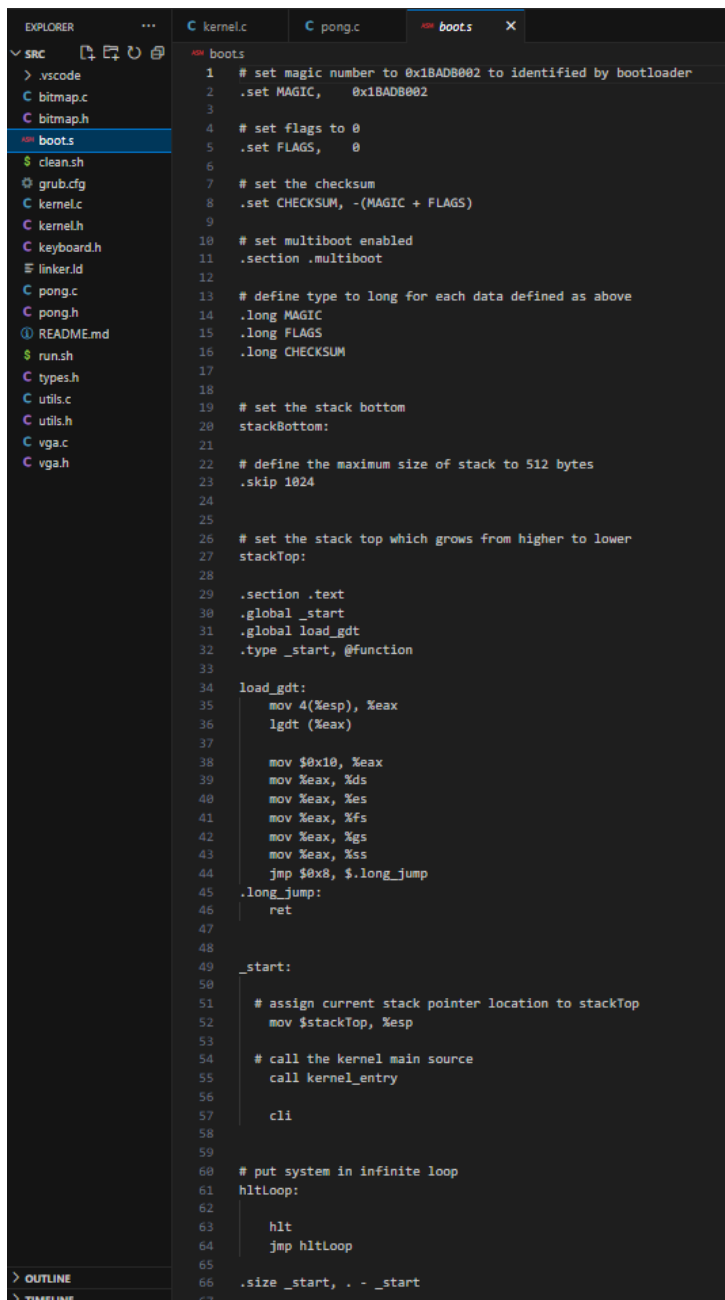
Similarly, user input captured from the keyboard is relayed to the game logic module to control game elements and progress the gameplay.

The kernel structure serves as the foundation of the operating system, providing essential functionalities such as process management, memory management, and input/output handling. It acts as the intermediary between hardware and software, managing system resources and facilitating communication between user programs and hardware devices. The structure is modular, with distinct components responsible for specific tasks, ensuring efficient operation and scalability of the operating system.

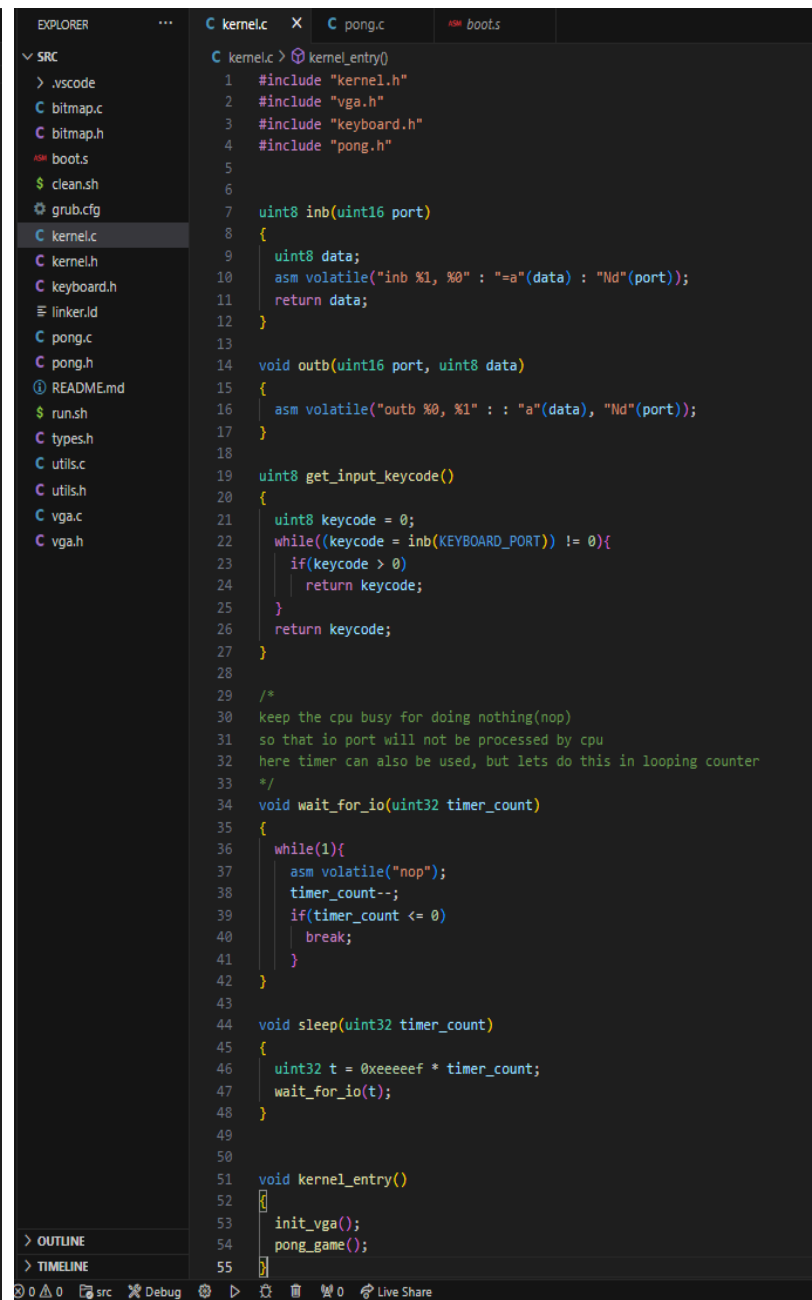
Source Code:

We have around 15 files out of which the main files are:

1. boot.S
2. kernel.c
3. pong.c
4. linker.ld
5. vga.c
6. keyboard.h



```
1 # set magic number to 0x1BAD0002 to identified by bootloader
2 .set MAGIC, 0x1BAD0002
3
4 # set flags to 0
5 .set FLAGS, 0
6
7 # set the checksum
8 .set CHECKSUM, ~(MAGIC + FLAGS)
9
10 # set multiboot enabled
11 .section .multiboot
12
13 # define type to long for each data defined as above
14 .long MAGIC
15 .long FLAGS
16 .long CHECKSUM
17
18
19 # set the stack bottom
20 stackBottom:
21
22 # define the maximum size of stack to 512 bytes
23 .skip 1024
24
25
26 # set the stack top which grows from higher to lower
27 stackTop:
28
29 .section .text
30 .global _start
31 .global load_gdt
32 .type _start, @function
33
34 load_gdt:
35     mov 4(%esp), %eax
36     lgdt (%eax)
37
38     mov $0x10, %eax
39     mov %eax, %ds
40     mov %eax, %es
41     mov %eax, %fs
42     mov %eax, %gs
43     mov %eax, %ss
44     jmp $0x8, $.long_jump
45 .long_jump:
46     ret
47
48
49 _start:
50
51 # assign current stack pointer location to stackTop
52 mov $stackTop, %esp
53
54 # call the kernel main source
55 call kernel_entry
56
57 cli
58
59
60 # put system in infinite loop
61 hltLoop:
62     hlt
63     jmp hltLoop
64
65
66 .size _start, . - _start
67
```



```
1 #include "kernel.h"
2 #include "vga.h"
3 #include "keyboard.h"
4 #include "pong.h"
5
6
7 uint8 inb(uint16 port)
8 {
9     uint8 data;
10    asm volatile("inb %1, %0" : "=a"(data) : "Nd"(port));
11    return data;
12 }
13
14 void outb(uint16 port, uint8 data)
15 {
16    asm volatile("outb %0, %1" : "=a"(data), "Nd"(port));
17 }
18
19 uint8 get_input_keycode()
20 {
21     uint8 keycode = 0;
22     while((keycode = inb(KEYBOARD_PORT)) != 0){
23         if(keycode > 0)
24             return keycode;
25     }
26     return keycode;
27 }
28
29 /*
30 keep the cpu busy for doing nothing(nop)
31 so that io port will not be processed by cpu
32 here timer can also be used, but lets do this in looping counter
33 */
34 void wait_for_io(uint32 timer_count)
35 {
36     while(1){
37         asm volatile("nop");
38         timer_count--;
39         if(timer_count <= 0)
40             break;
41     }
42 }
43
44 void sleep(uint32 timer_count)
45 {
46     uint32 t = 0xeeeeef * timer_count;
47     wait_for_io(t);
48 }
49
50
51 void kernel_entry()
52 {
53     init_vga();
54     pong_game();
55 }
```



```
C kernel.c C pong.c C keyboard.h X
C keyboard.h > ...
1 #ifndef KEYBOARD_H
2 #define KEYBOARD_H
3
4 #define KEYBOARD_PORT 0x60
5
6
7 #define KEY_A 0x1E
8 #define KEY_B 0x30
9 #define KEY_C 0x2E
10 #define KEY_D 0x20
11 #define KEY_E 0x12
12 #define KEY_F 0x21
13 #define KEY_G 0x22
14 #define KEY_H 0x23
15 #define KEY_I 0x17
16 #define KEY_J 0x24
17 #define KEY_K 0x25
18 #define KEY_L 0x26
19 #define KEY_M 0x32
20 #define KEY_N 0x31
21 #define KEY_O 0x18
22 #define KEY_P 0x19
23 #define KEY_Q 0x10
24 #define KEY_R 0x13
25 #define KEY_S 0x1F
26 #define KEY_T 0x14
27 #define KEY_U 0x16
28 #define KEY_V 0x2F
29 #define KEY_W 0x11
30 #define KEY_X 0x2D
31 #define KEY_Y 0x15
32 #define KEY_Z 0x2C
33 #define KEY_1 0x02
34 #define KEY_2 0x03
35 #define KEY_3 0x04
36 #define KEY_4 0x05
37 #define KEY_5 0x06
38 #define KEY_6 0x07
39 #define KEY_7 0x08
40 #define KEY_8 0x09
41 #define KEY_9 0x0A
42 #define KEY_0 0x0B
43 #define KEY_MINUS 0x0C
44 #define KEY_EQUAL 0x0D
45 #define KEY_SQUARE_OPEN_BRACKET 0x1A
```

```
C kernel.c C vga.c X C pong.c
C vga.c > set_gc()
32 void set_misc()
33 {
34     outb(VGA_MISC_WRITE, 0x63);
35 }
36
37 void set_seq()
38 {
39     // write sequence data to index of 0-4
40     for(uint8 index = 0; index < 5; index++){
41         // first set index
42         outb(VGA_SEQ_INDEX, index);
43         // write data at that index
44         outb(VGA_SEQ_DATA, seq_data[index]);
45     }
46 }
47
48 void set_crtc()
49 {
50     // write crtc data to index of 0-24
51     for(uint8 index = 0; index < 25; index++){
52         outb(VGA_CRTC_INDEX, index);
53         outb(VGA_CRTC_DATA, crtc_data[index]);
54     }
55 }
56
57 void set_gc()
58 {
59     // write gc data to index of 0-8
60     for(uint8 index = 0; index < 9; index++){
61         outb(VGA_GC_INDEX, index);
62         outb(VGA_GC_DATA, gc_data[index]);
63     }
64 }
65
66 void set_ac()
67 {
68     uint8 d;
69     // write ac data to index of 0-20
70     for(uint8 index = 0; index < 21; index++){
71         outb(VGA_AC_INDEX, index);
72         outb(VGA_AC_WRITE, ac_data[index]);
73     }
74     d = inb(VGA_INSTANT_READ);
75     outb(VGA_AC_INDEX, d | 0x20);
76 }
77
```

```
C kernel.c C vga.c X C pong.c
C vga.c > ...
1 #include "vga.h"
2 #include "kernel.h"
3
4 // a global vga buffer
5 uint8* g_vga_buffer;
6
7 /*
8 See Intel® OpenSource HD Graphics PRM pdf file
9 for following defined data for each vga register
10 and its explanation
11 */
12 static uint8 seq_data[5] = {0x03, 0x01, 0x0F, 0x00, 0x0E};
13 static uint8 crtc_data[25] = {0x5F, 0x4F, 0x50, 0x82,
14                               0x54, 0x80, 0xBF, 0x1F,
15                               0x00, 0x41, 0x00, 0x00,
16                               0x00, 0x00, 0x00, 0x00,
17                               0x9C, 0x0E, 0xBF, 0x28,
18                               0x40, 0x96, 0xB9, 0xA3,
19                               0xFF};
20
21 static uint8 gc_data[9] = {0x00, 0x00, 0x00, 0x00,
22                             0x00, 0x40, 0x05, 0x0F,
23                             0xFF};
24
25 static uint8 ac_data[21] = {0x00, 0x01, 0x02, 0x03,
26                             0x04, 0x05, 0x06, 0x07,
27                             0x08, 0x09, 0x0A, 0x0B,
28                             0x0C, 0x0D, 0x0E, 0x0F,
29                             0x41, 0x00, 0x0F, 0x00,
30                             0x00};
31
32 void set_misc()
33 {
34     outb(VGA_MISC_WRITE, 0x63);
35 }
36
37 void set_seq()
38 {
39     // write sequence data to index of 0-4
40     for(uint8 index = 0; index < 5; index++){
41         // first set index
42         outb(VGA_SEQ_INDEX, index);
43         // write data at that index
44         outb(VGA_SEQ_DATA, seq_data[index]);
45     }
46 }
```

```
C kernel.c C pong.c linker.ld X
linker.ld
1 /* The bootloader will look at this image and start execution at the symbol
2  * designated as the entry point. */
3 ENTRY(_start)
4
5 /* Tell where the various sections of the object files will be put in the final
6  * kernel image. */
7 SECTIONS
8 {
9     /* Begin putting sections at 1 MiB, a conventional place for kernels to be
10      * loaded at by the bootloader. */
11     . = 1M;
12
13     /* First put the multiboot header, as it is required to be put very early
14      * in the image or the bootloader won't recognize the file format.
15      * Next we'll put the .text section. */
16     .text BLOCK(4K) : ALIGN(4K)
17     {
18         *(.multiboot)
19         *(.text)
20     }
21
22     /* Read-only data. */
23     .rodata BLOCK(4K) : ALIGN(4K)
24     {
25         *(.rodata)
26     }
27
28     /* Read-write data (initialized) */
29     .data BLOCK(4K) : ALIGN(4K)
30     {
31         *(.data)
32     }
33
34     /* Read-write data (uninitialized) and stack */
35     .bss BLOCK(4K) : ALIGN(4K)
36     {
37         *(COMMON)
38         *(.bss)
39     }
40
41     /* The compiler may produce other sections, by default it will put them in
42      * a segment with the same name. Simply add stuff here as needed. */
43 }
```

```
C kernel.c C pong.c X linker.ld
C pong.c > init_game()
46 static void lose()
47 {
48     uint8 b = 0;
49     char str[32];
50
51     itoa(score_count, str);
52     clear_screen();
53     draw_string(120, 15, BRIGHT_GREEN, "NICE PLAY!");
54     draw_string(125, 45, WHITE, "SCORE");
55     draw_string(100, 45, WHITE, str);
56     draw_string(45, 150, YELLOW, "PRESS ENTER TO PLAY AGAIN!");
57 }
58 #ifdef VIRTUALBOX
59 sleep(10);
60 #endif
61 while (1)
62 {
63     b = get_input_keycode();
64     sleep(5);
65     if (b == KEY_ENTER)
66         break;
67     b = 0;
68 }
69 score_count = 0;
70 clear_screen();
71 pong_game();
72 }
73
74 // move both pads simultaneously on pressed keys
75 void move_pads()
76 {
77     uint8 b;
78
79     // draw both pads
80     fill_rect(0, pad_pos_y, PAD_WIDTH, PAD_HEIGHT, YELLOW);
81     fill_rect(PAD_POS_X, pad_pos_y, PAD_WIDTH, PAD_HEIGHT, YELLOW);
82
83     b = get_input_keycode();
84     // if down key pressed, move both pads down
85     if (b == KEY_DOWN)
86     {
87         if (pad_pos_y < VGA_MAX_HEIGHT - PAD_HEIGHT)
88             pad_pos_y = pad_pos_y + PAD_SPEED;
89         fill_rect(0, pad_pos_y, PAD_WIDTH, PAD_HEIGHT, YELLOW);
90         fill_rect(PAD_POS_X, pad_pos_y, PAD_WIDTH, PAD_HEIGHT, YELLOW);
91     }
92     // if up key pressed, move both pads up
93     else if (b == KEY_UP)
94     {
95         if (pad_pos_y >= PAD_WIDTH)
96             pad_pos_y = pad_pos_y - PAD_SPEED;
97         fill_rect(0, pad_pos_y, PAD_WIDTH, PAD_HEIGHT, YELLOW);
98         fill_rect(PAD_POS_X, pad_pos_y, PAD_WIDTH, PAD_HEIGHT, YELLOW);
99     }
100 #ifdef VIRTUALBOX
101 sleep(1);
102 }
```

```
C kernel.c C pong.c X linker.ld
C pong.c > init_game()
1 #include "pong.h"
2
3 // pads position y, will change on keys
4 uint16 pad_pos_y = 2;
5 // score count
6 uint32 score_count = 0;
7
8 // initialize game with into
9 static void init_game()
10 {
11     uint8 b = 0;
12
13     draw_string(120, 13, BRIGHT_CYAN, "PONG GAME");
14     draw_rect(100, 4, 120, 25, BLUE);
15     draw_string(10, 50, BRIGHT_MAGENTA, "HOW TO PLAY");
16     draw_rect(2, 40, 235, 50, BROWN);
17     draw_string(130, 70, BRIGHT_RED, "ARROW KEY UP");
18     draw_string(30, 80, WHITE, "TO MOVE BOTH PADS UP");
19     draw_string(10, 90, BRIGHT_RED, "ARROW KEY DOWN");
20     draw_string(30, 100, WHITE, "TO MOVE BOTH PADS DOWN");
21     draw_string(60, 160, BRIGHT_GREEN, "PRESS ENTER TO START");
22 #ifdef VIRTUALBOX
23 sleep(10);
24 #endif
25 while (1)
26 {
27     b = get_input_keycode();
28     sleep(5);
29     if (b == KEY_ENTER)
30         break;
31     b = 0;
32 }
33 clear_screen();
34
35 // update score count text
36 static void update_score_count()
37 {
38     char str[32];
39
40     itoa(score_count, str);
41     draw_string(150, 2, WHITE, str);
42 }
43
44 // if lose then display final score & restart game
45 static void lose()
46 {
47     uint8 b = 0;
48     char str[32];
49
50     itoa(score_count, str);
51     clear_screen();
52     draw_string(120, 15, BRIGHT_GREEN, "NICE PLAY!");
53     draw_string(125, 45, WHITE, "SCORE");
54     draw_string(100, 45, WHITE, str);
55     draw_string(45, 150, YELLOW, "PRESS ENTER TO PLAY AGAIN!");
56 }
```

Output



