

**“AZƏRBAYCAN HAVA YOLLARI” CJSC NATIONAL AVIATION ACADEMY**

**Individual Work № : 5**

**Topic:** **Linux (kernel) programming in C language**

**Subject: OS2**

**Teacher: Mamed Shahmaliyev**

**Group: 1459i Student: Rəvan Yusifov**

**Date: Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_**

**­ Baku 2022**

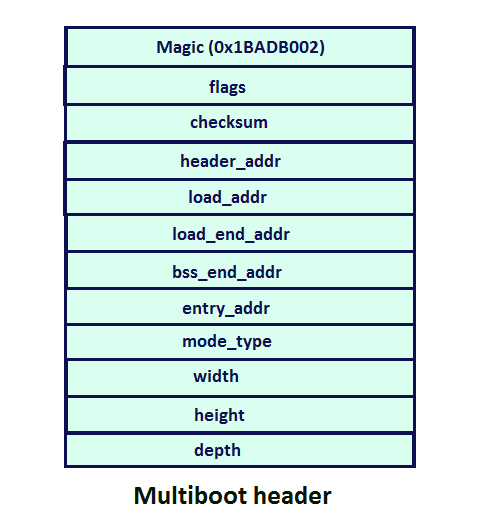
## Using the code

Alright, writing a kernel from scratch is to print something on screen.  
So we have a VGA(Visual Graphics Array), a hardware system that controls the display.

VGA has a fixed amount of memory and addresssing is **0xA0000** to **0xBFFFF**.

**0xA0000** for EGA/VGA graphics modes (64 KB)  
**0xB0000** for monochrome text mode (32 KB)  
**0xB8000** for color text mode and CGA-compatible graphics modes (32 KB)

First you need a multiboot bootloader file that instruct the GRUB to load it.  
Following fields must be define.



**Magic :-**A fixed hexadecimal number identified by the bootloader as the header(starting point) of the kernel to be loaded.  
**flags :-**If bit 0 in the flags word is set, then all boot modules loaded along with the operating system must be aligned on page (4KB) boundaries.  
**checksum :-** which is used by special purpose by bootloader and its value must be the sum of magic no and flags.

We don't need other information,   
but for more details  <https://www.gnu.org/software/grub/manual/multiboot/multiboot.pdf>

Ok lets write a GAS assembly code for above information.  
we dont need some fields as shown in above image.

**boot.S**

Python

Shrink ▲   Copy Code

*# 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

We have defined a stack of size 1024 bytes and managed by stackBottom and stackTop identifiers.  
Then in \_start, we are storing a current stack pointer, and calling the main function of a kernel(kernel\_entry).

As you know, every process consists of different sections such as data, bss, rodata and text.  
You can see the each sections by compiling the source code without assembling it.

e.g.: Run the following command  
        **gcc -S kernel.c**  
      and see the kernel.S file.

And this sections requires a memory to store them, this memory size is provided by the linker image file.  
Each memory is aligned with the size of each block.  
It mostly require to link all the object files together to form a final kernel image.  
Linker image file provides how much size should be allocated to each of the sections.  
The information is stored in the final kernel image.  
If you open the final kernel image(.bin file) in hexeditor, you can see lots of 00 bytes.  
the linker image file consists of an entry point,(in our case it is \_start defined in file boot.S) and sections with size defined in the BLOCK keyword aligned from how much spaced.

**linker.ld**

C++

Shrink ▲   Copy Code

*/\* entry point of our kernel \*/*

ENTRY(\_start)

SECTIONS

{

*/\* we need 1MB of space atleast \*/*

. = 1M;

*/\* text section \*/*

.text BLOCK(4K) : ALIGN(4K)

{

\*(.multiboot)

\*(.text)

}

*/\* read only data section \*/*

.rodata BLOCK(4K) : ALIGN(4K)

{

\*(.rodata)

}

*/\* data section \*/*

.data BLOCK(4K) : ALIGN(4K)

{

\*(.data)

}

*/\* bss section \*/*

.bss BLOCK(4K) : ALIGN(4K)

{

\*(COMMON)

\*(.bss)

}

}

Now you need a configuration file that instruct the grub to load menu with associated image file  
**grub.cfg**

C++

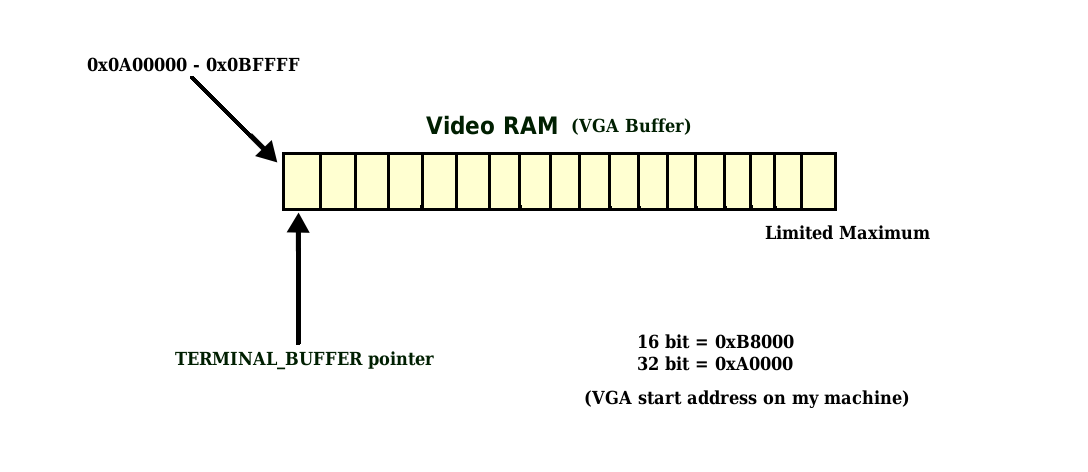
Copy Code

menuentry "MyOS" {

multiboot /boot/MyOS.bin

}

Now let's write a simple HelloWorld kernel code.



### **Simple :-**

**kernel\_1 :-**

**kernel.h**

C++

Shrink ▲   Copy Code

#ifndef KERNEL\_H#define KERNEL\_Htypedef unsigned char uint8;typedef unsigned short uint16;typedef unsigned int uint32;

#define VGA\_ADDRESS 0xB8000#define BUFSIZE 2200

uint16\* vga\_buffer;

#define NULL 0enum vga\_color {

BLACK,

BLUE,

GREEN,

CYAN,

RED,

MAGENTA,

BROWN,

GREY,

DARK\_GREY,

BRIGHT\_BLUE,

BRIGHT\_GREEN,

BRIGHT\_CYAN,

BRIGHT\_RED,

BRIGHT\_MAGENTA,

YELLOW,

WHITE,

};

#endif

Here we are using 16 bit vga buffer, on my machine the VGA address is starts at **0xB8000** and 32 bit starts at **0xA0000**.  
An unsigned 16 bit type terminal buffer pointer that points to VGA address.  
It has 8\*16 pixel font size.  
see above image.

**kernel.c**

C++

Shrink ▲   Copy Code

#include "kernel.h"*/\**

*16 bit video buffer elements(register ax)*

*8 bits(ah) higher :*

*lower 4 bits - forec olor*

*higher 4 bits - back color*

*8 bits(al) lower :*

*8 bits : ASCII character to print*

*\*/*

uint16 vga\_entry(unsigned char ch, uint8 fore\_color, uint8 back\_color)

{

uint16 ax = 0;

uint8 ah = 0, al = 0;

ah = back\_color;

ah <<= 4;

ah |= fore\_color;

ax = ah;

ax <<= 8;

al = ch;

ax |= al;

return ax;

}

*//clear video buffer array*void clear\_vga\_buffer(uint16 \*\*buffer, uint8 fore\_color, uint8 back\_color)

{

uint32 i;

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

(\*buffer)[i] = vga\_entry(NULL, fore\_color, back\_color);

}

}

*//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);

}

The value returned by **vga\_entry()** function is the **uint16** type with highlighting the character to print it with color.  
The value is stored in the buffer to display the characters on a screen.  
First lets point our pointer **vga\_buffer** to VGA address **0xB8000**.

**Segment : 0xB800 & Offset : 0(our index variable(vga\_index))**  
Now you have an array of VGA, you just need to assign specific value to each index of array according to what to print on a screen as we usually do in assigning the value to array.  
See the above code that prints each character of HelloWorld on a screen.

Ok lets compile the source.  
type sh run.sh command on terminal.

**run.sh**

Python

Copy Code

*#assemble boot.s file*as --32 boot.s -o boot.o

*#compile kernel.c file*

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

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

ld -m elf\_i386 -T linker.ld kernel.o boot.o -o MyOS.bin -nostdlib

*#check MyOS.bin file is x86 multiboot file or not*

grub-file --is-x86-multiboot MyOS.bin

*#building the iso file*

mkdir -p isodir/boot/grub

cp MyOS.bin isodir/boot/MyOS.bin

cp grub.cfg isodir/boot/grub/grub.cfg

grub-mkrescue -o MyOS.iso isodir

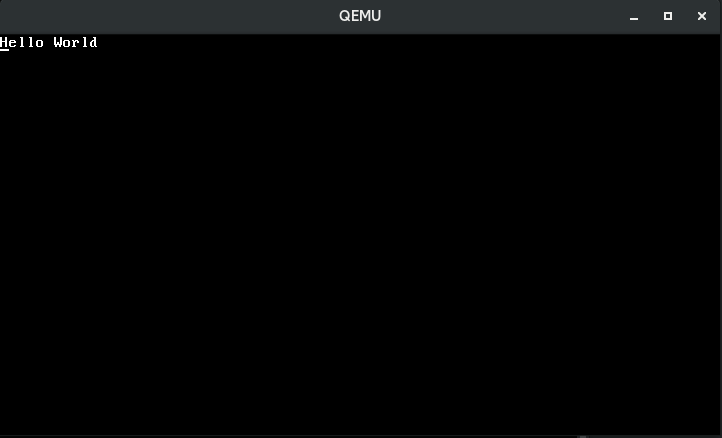
*#run it in qemu*

qemu-system-x86\_64 -cdrom MyOS.iso

Make sure you have installed all the packages that requires to build a kernel.

the output is





As you can see, it is a overhead to assign each and every value to VGA buffer, so we can write a function for that, which can print our string on a screen (means assigning each character value to VGA buffer from a string).

**kernel\_2 :-**

**kernel.h**

C++

Shrink ▲   Copy Code

#ifndef KERNEL\_H#define KERNEL\_Htypedef unsigned char uint8;typedef unsigned short uint16;typedef unsigned int uint32;

#define VGA\_ADDRESS 0xB8000#define BUFSIZE 2200

uint16\* vga\_buffer;

#define NULL 0enum vga\_color {

BLACK,

BLUE,

GREEN,

CYAN,

RED,

MAGENTA,

BROWN,

GREY,

DARK\_GREY,

BRIGHT\_BLUE,

BRIGHT\_GREEN,

BRIGHT\_CYAN,

BRIGHT\_RED,

BRIGHT\_MAGENTA,

YELLOW,

WHITE,

};

#endif

digit\_ascii\_codes are hexadecimal values of characters 0 to 9. we need them when we want to print them on a screen.vga\_index is the our VGA array index. vga\_index is increased when value is assigned to that index.To print an 32 bit integer, first you need to convert it into a string and then print a string.  
BUFSIZE is the limit of our VGA. To print a new line, you have to skip some bytes in VGA pointer(vga\_buffer) according to the pixel font size.  
For this we need another variable that stores the current line index(next\_line\_index).

C++

Shrink ▲   Copy Code

#include "kernel.h"*//index for video buffer array*uint32 vga\_index;*//counter to store new lines*static uint32 next\_line\_index = 1;*//fore & back color values*uint8 g\_fore\_color = WHITE, g\_back\_color = BLUE;*//digit ascii code for printing integers*int digit\_ascii\_codes[10] = {0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39};

*/\**

*16 bit video buffer elements(register ax)*

*8 bits(ah) higher :*

*lower 4 bits - forec olor*

*higher 4 bits - back color*

*8 bits(al) lower :*

*8 bits : ASCII character to print*

*\*/*

uint16 vga\_entry(unsigned char ch, uint8 fore\_color, uint8 back\_color)

{

uint16 ax = 0;

uint8 ah = 0, al = 0;

ah = back\_color;

ah <<= 4;

ah |= fore\_color;

ax = ah;

ax <<= 8;

al = ch;

ax |= al;

return ax;

}

*//clear video buffer array*void clear\_vga\_buffer(uint16 \*\*buffer, uint8 fore\_color, uint8 back\_color)

{

uint32 i;

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

(\*buffer)[i] = vga\_entry(NULL, fore\_color, back\_color);

}

next\_line\_index = 1;

vga\_index = 0;

}

*//initialize vga buffer*void init\_vga(uint8 fore\_color, uint8 back\_color)

{

vga\_buffer = (uint16\*)VGA\_ADDRESS;

clear\_vga\_buffer(&vga\_buffer, fore\_color, back\_color);

g\_fore\_color = fore\_color;

g\_back\_color = back\_color;

}

*/\**

*increase vga\_index by width of row(80)*

*\*/*void print\_new\_line()

{

if(next\_line\_index >= 55){

next\_line\_index = 0;

clear\_vga\_buffer(&vga\_buffer, g\_fore\_color, g\_back\_color);

}

vga\_index = 80\*next\_line\_index;

next\_line\_index++;

}

*//assign ascii character to video buffer*void print\_char(char ch)

{

vga\_buffer[vga\_index] = vga\_entry(ch, g\_fore\_color, g\_back\_color);

vga\_index++;

}

uint32 strlen(const char\* str)

{

uint32 length = 0;

while(str[length])

length++;

return length;

}

uint32 digit\_count(int num)

{

uint32 count = 0;

if(num == 0)

return 1;

while(num > 0){

count++;

num = num/10;

}

return count;

}

void itoa(int num, char \*number)

{

int dgcount = digit\_count(num);

int index = dgcount - 1;

char x;

if(num == 0 && dgcount == 1){

number[0] = '0';

number[1] = '\0';

}else{

while(num != 0){

x = num % 10;

number[index] = x + '0';

index--;

num = num / 10;

}

number[dgcount] = '\0';

}

}

*//print string by calling print\_char*void print\_string(char \*str)

{

uint32 index = 0;

while(str[index]){

print\_char(str[index]);

index++;

}

}

*//print int by converting it into string//& then printing string*void print\_int(int num)

{

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

itoa(num, str\_num);

print\_string(str\_num);

}

void kernel\_entry()

{

*//first init vga with fore & back colors* init\_vga(WHITE, BLACK);

*/\*call above function to print something*

*here to change the fore & back color*

*assign g\_fore\_color & g\_back\_color to color values*

*g\_fore\_color = BRIGHT\_RED;*

*\*/*

print\_string("Hello World!");

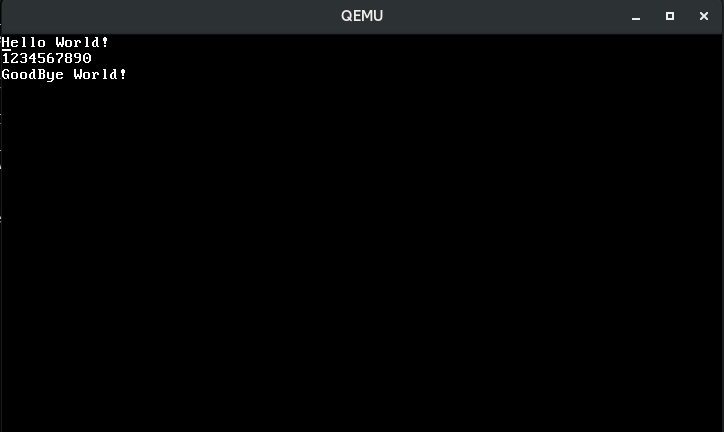
print\_new\_line();

print\_int(123456789);

print\_new\_line();

print\_string("Goodbye World!");

}



As you can see it is the overhead to call each and every function for displaying the values, that's why C programming provides a **printf()**function with format specifiers which print/set specific value to standard output device with each specifier with literals such as \n, \t, \r etc.

### **Keyboard :-**

For keyboard I/O, use port number 0x60 with instructions in/out.Download kernel\_source code fro keyboard. It reads keystrokes from user and displays them on screen.

C++

Shrink ▲   Copy Code

#ifndef KEYBOARD\_H#define KEYBOARD\_H#define KEYBOARD\_PORT 0x60

#define KEY\_A 0x1E#define KEY\_B 0x30#define KEY\_C 0x2E#define KEY\_D 0x20#define KEY\_E 0x12#define KEY\_F 0x21#define KEY\_G 0x22#define KEY\_H 0x23#define KEY\_I 0x17#define KEY\_J 0x24#define KEY\_K 0x25#define KEY\_L 0x26#define KEY\_M 0x32#define KEY\_N 0x31#define KEY\_O 0x18#define KEY\_P 0x19#define KEY\_Q 0x10#define KEY\_R 0x13#define KEY\_S 0x1F#define KEY\_T 0x14#define KEY\_U 0x16#define KEY\_V 0x2F#define KEY\_W 0x11#define KEY\_X 0x2D#define KEY\_Y 0x15#define KEY\_Z 0x2C#define KEY\_1 0x02#define KEY\_2 0x03#define KEY\_3 0x04#define KEY\_4 0x05#define KEY\_5 0x06#define KEY\_6 0x07#define KEY\_7 0x08#define KEY\_8 0x09#define KEY\_9 0x0A#define KEY\_0 0x0B#define KEY\_MINUS 0x0C#define KEY\_EQUAL 0x0D#define KEY\_SQUARE\_OPEN\_BRACKET 0x1A#define KEY\_SQUARE\_CLOSE\_BRACKET 0x1B#define KEY\_SEMICOLON 0x27#define KEY\_BACKSLASH 0x2B#define KEY\_COMMA 0x33#define KEY\_DOT 0x34#define KEY\_FORESLHASH 0x35#define KEY\_F1 0x3B#define KEY\_F2 0x3C#define KEY\_F3 0x3D#define KEY\_F4 0x3E#define KEY\_F5 0x3F#define KEY\_F6 0x40#define KEY\_F7 0x41#define KEY\_F8 0x42#define KEY\_F9 0x43#define KEY\_F10 0x44#define KEY\_F11 0x85#define KEY\_F12 0x86#define KEY\_BACKSPACE 0x0E#define KEY\_DELETE 0x53#define KEY\_DOWN 0x50#define KEY\_END 0x4F#define KEY\_ENTER 0x1C#define KEY\_ESC 0x01#define KEY\_HOME 0x47#define KEY\_INSERT 0x52#define KEY\_KEYPAD\_5 0x4C#define KEY\_KEYPAD\_MUL 0x37#define KEY\_KEYPAD\_Minus 0x4A#define KEY\_KEYPAD\_PLUS 0x4E#define KEY\_KEYPAD\_DIV 0x35#define KEY\_LEFT 0x4B#define KEY\_PAGE\_DOWN 0x51#define KEY\_PAGE\_UP 0x49#define KEY\_PRINT\_SCREEN 0x37#define KEY\_RIGHT 0x4D#define KEY\_SPACE 0x39#define KEY\_TAB 0x0F#define KEY\_UP 0x48

#endif

inb() receives byte from specified port and returns it.

outb() send byte to specified port.

C++

Shrink ▲   Copy Code

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);

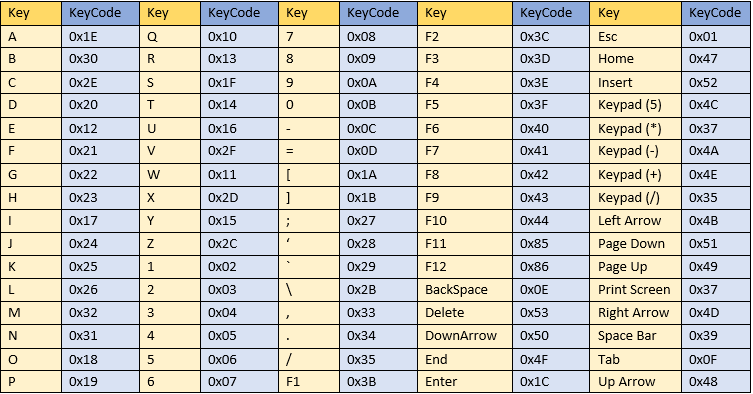
print\_string("Type here, one key per second, ENTER to go to next line");

print\_new\_line();

test\_input();

}

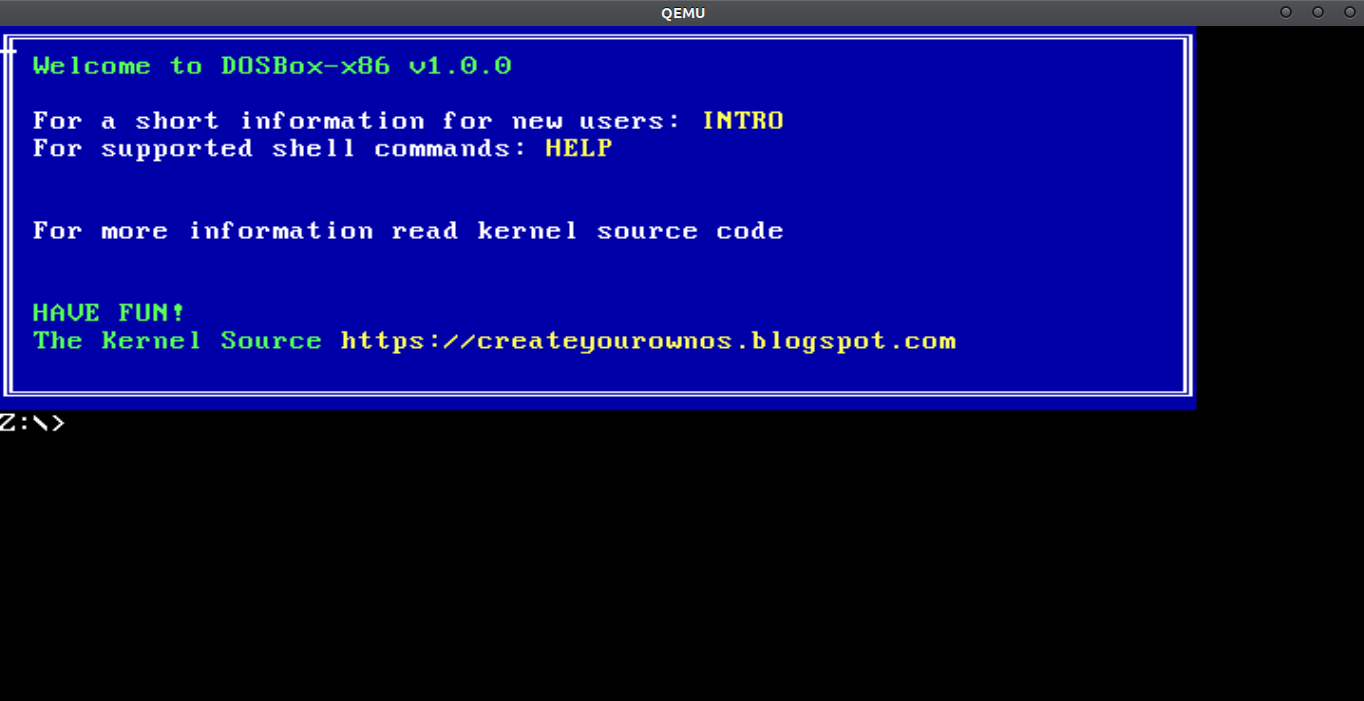
Each keycode is being converted into its ASCII character by function **get\_ascii\_char()**.



### Box-drawing GUI :-

Download the kernel\_source for drawing boxes used in old systems such as DOSBox etc.(kernel\_source/GUI/)

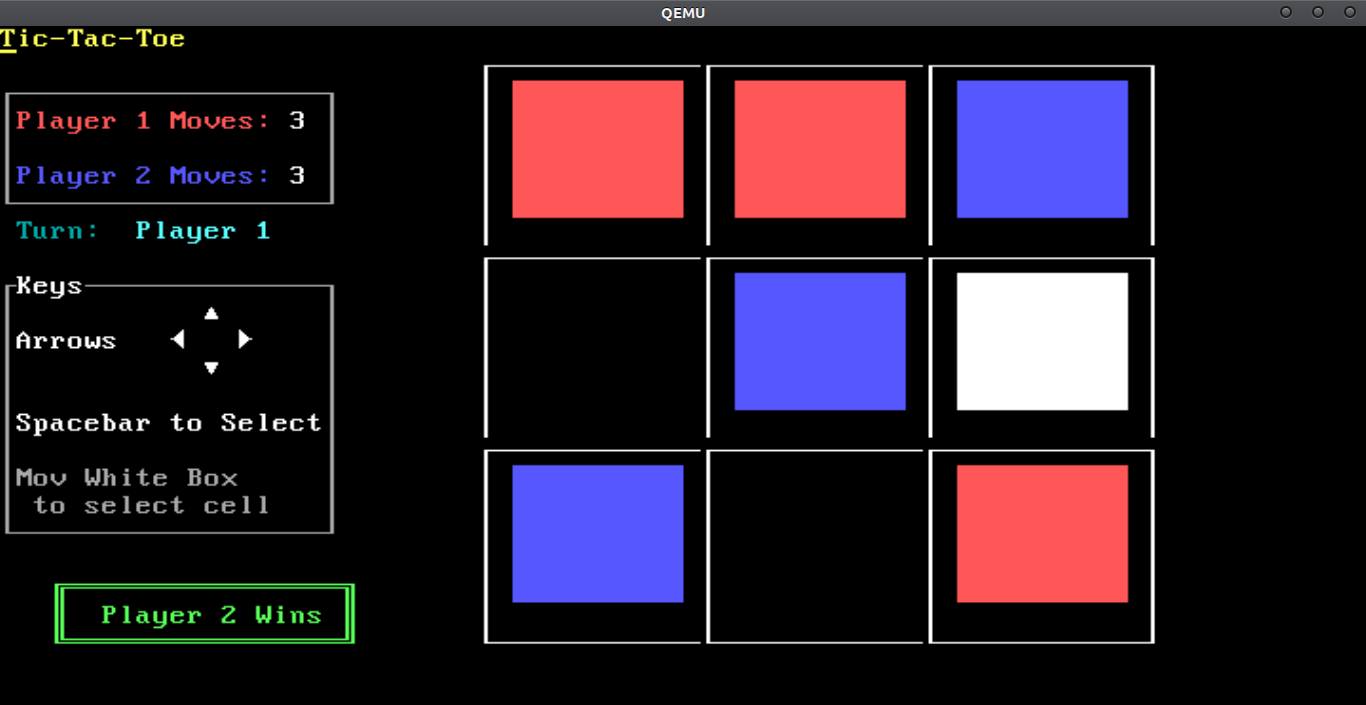




### Tic-Tac-Toe Game :-

We have printing code, keyboard I/O handling and GUI using box drawing characters.So lets write a simple Tic-Tac-Toe game in kernel that can be run on any PC.

Download the kernel\_source code, kernel\_source/Tic-Tac-Toe.



How to Play :

Use arrow keys(UP, DOWN, LEFT, RIGHT) to move white box between cells and press SPACEBAR to select that cell.

RED color for player 1 Box and BLUE color for player 2 box.

See Turn for which player has a turn to select cell.(Turn:    Player 1)

If you are running this on actual hardware then increase the value of **sleep()** function in **launch\_game()** in **tic\_tac\_toe.c** and in **kernel\_entry()** in kernel.c So that will work normally and not too fast. I used 0x2FFFFFFF.