**Налаштування середовища**

Лабораторну роботу було виконано на ОС Windows (без Docker) у IntelliJ IDEA + Rust plugin. Для цього було завантажено додатковий інструментарій та віртуальну машину qemu, але спершу було скопійовано шаблон з github:

*git clone https://github.com/voievodin/unios*

Встановлення інструментарію (має бути виконано в папці з проектом):

*cargo install bootimage*

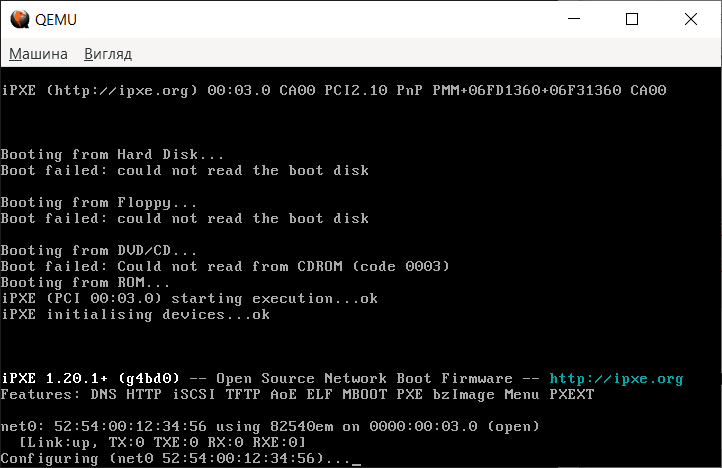
*rustup component add rust-src --toolchain nightly-x86\_64-pc-windows-msvc*

*rustup component add llvm-tools-preview*

*rustup override set nightly*

Перевірка коректності встановлення віртуальної машини:

*qemu-system-x86\_64*



**Програмна реалізація:**

Завдання: Реалізувати драйвер для VGA буферу

Під час написання драйверу для структури Screen було імплементовано наступні додаткові функції (модуль vba\_buf):

|  |  |
| --- | --- |
| print(&mut self, s: &str) | функція для відображення на екрані цілого рядку |
| add\_text(&mut self, row: &[u8]) | функція для збереження тексту до поточного стану екрану |
| scroll(&mut self) | функція, що зміщує всі рядки на один вгору та затирає останній, тим самим звільняючи місце для нового рядка |
| calc\_align(&self, row: &[u8]) -> u32 | функція, що вираховує кількість пробілів, необхідну для конкретного типу вирівнювання конкретного рядка |

Саму структуру Screen також було модифіковано наступним чином:

*pub struct Screen {*

*buffer: \*mut u8, // буфер*

*color: u8,       // колір*

*align: Alignment, // тип вирівнювання*

*cursor\_row: u32, // позиція курсору в рядку*

*cursor\_col: u32, // позиція курсору в стовпчику*

*state : [[u8; BUF\_WIDTH as usize]; BUF\_HEIGHT as usize], // поточний стан екрану*

*cursor\_write: u32 // кількість заповнених рядків*

}

Також було створено додаткову структуру Color для зберігання значень кольорів:

*#[repr(u8)]*

*pub enum Color {*

*BLUE = 0x1,*

*GREEN = 0x2,*

*AZURE = 0x3,*

*RED = 0x4,*

*PURPLE = 0x5,*

*BROWN = 0x6,*

*LIGHT\_GREY = 0x7,*

*DARK\_GREY = 0x8,*

*LIGHT\_BLUE = 0x9,*

*LIGHT\_GREEN = 0xa,*

*LIGHT\_AZURE = 0xb,*

*LIGHT\_RED = 0xc,*

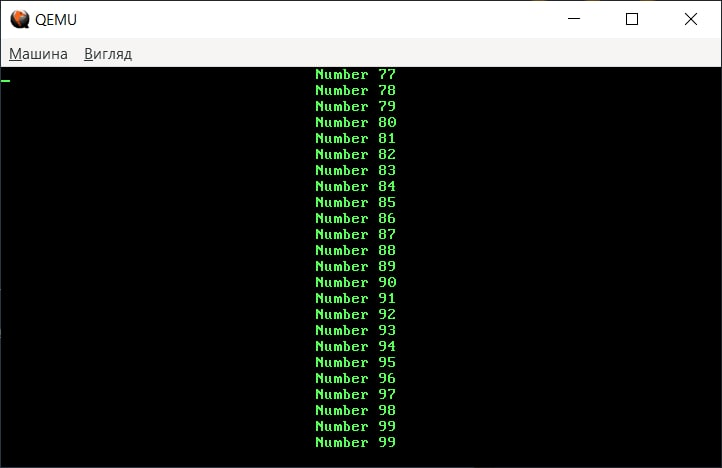
*PINK = 0xd,*

*YELLOW = 0xe,*

*WHITE = 0xf*

*}*

**Результати роботи програми**

**Лістинінг програми**

**Файл main.rs:**

#![no\_std] // don't link the Rust standard library

#![no\_main] // disable all Rust-level entry points

mod vga\_buf;

use core::fmt::Write;

use core::panic::PanicInfo;

// імпортуємо структури Alignment, Color та Screen з модуля vga\_buf

use crate::vga\_buf::{Alignment, Color, Screen};

/// This function is called on panic.

#[panic\_handler]

fn panic(\_info: &PanicInfo) -> ! {

    loop {}

}

#[no\_mangle] // don't mangle the name of this function

pub extern "C" fn \_start() -> ! { // точка входу для нашої ОС

    //створюємо екземпляр структури Screen, передаючи як параметри колір шрифту та тип вирівнювання

    let mut screen = Screen::new(Color::LIGHT\_GREEN as u8, Alignment::Center);

    // 100 раз неявно викликаємо нашу функцію print через макрос write!

    for i in 0..100 {

        write!(screen, "Number {}\n", i);

    }

    loop {}

}

**Файл vba\_buf.rs:**

use core::borrow::Borrow;

const BUF\_ADDR: u32 = 0xb8000; // адреса початку буферу VGA

const BUF\_HEIGHT: u32 = 25; // висота буферу (екрану)

const BUF\_WIDTH: u32 = 80;  // ширина буферу (екрану)

const COLOR\_BLACK: u8 = 0x0; // константа для чорного кольору

// структура, що репрезентує символ ASCII

pub struct AsciiChar {

    pub char\_byte: u8, // значення символу

    pub color\_byte: u8 // значення кольору

}

// enum, що зберігає типи вирівнювання

pub enum Alignment {

    Left,

    Right,

    Center

}

// enum, що зберігає значення кольорів

#[repr(u8)]

pub enum Color {

    BLUE = 0x1,

    GREEN = 0x2,

    AZURE = 0x3,

    RED = 0x4,

    PURPLE = 0x5,

    BROWN = 0x6,

    LIGHT\_GREY = 0x7,

    DARK\_GREY = 0x8,

    LIGHT\_BLUE = 0x9,

    LIGHT\_GREEN = 0xa,

    LIGHT\_AZURE = 0xb,

    LIGHT\_RED = 0xc,

    PINK = 0xd,

    YELLOW = 0xe,

    WHITE = 0xf

}

// структура, що зберігає параметри нашого екрану

pub struct Screen {

    buffer: \*mut u8, // буфер

    color: u8,       // колір

    align: Alignment, // тип вирівнювання

    cursor\_row: u32, // позиція курсору в рядку

    cursor\_col: u32, // позиція курсору в стовпчику

    state : [[u8; BUF\_WIDTH as usize]; BUF\_HEIGHT as usize], // поточний стан екрану

    cursor\_write: u32 // кількість заповнених рядків

}

// імплементуємо поведінку макроса write! для структури Screen

impl core::fmt::Write for Screen {

    fn write\_str(&mut self, s: &str) -> core::fmt::Result {

        self.print(s);

        Ok(())

    }

}

// імплементація структури Screen (можна сказати, аналог методів класу в Java чи C++)

impl Screen {

    // конструктор з параметрами, де ми наділяємо поля відповідними значеннями

    pub fn new(color: u8, align: Alignment) -> Screen {

        return Screen {

            buffer: BUF\_ADDR as \*mut u8,

            color: (COLOR\_BLACK << 4) | color,

            align,

            cursor\_row: 0,

            cursor\_col: 0,

            state : [[0; BUF\_WIDTH as usize]; BUF\_HEIGHT as usize],

            cursor\_write: 0

        }

    }

    // функція, що записує символ до буферу

    pub fn write\_char(&mut self, offset: u32, char: AsciiChar) {

        unsafe {

            \*self.buffer.offset(offset as isize \* 2) = char.char\_byte;

            \*self.buffer.offset(offset as isize \* 2 + 1) = char.color\_byte;

        }

        self.cursor\_write += 1;

    }

    // функція, що зчитує символ з буферу

    pub fn read\_char(&self, offset: u32) -> AsciiChar {

        unsafe {

            return AsciiChar{

                char\_byte: \*self.buffer.offset(offset as isize \* 2),

                color\_byte: \*self.buffer.offset(offset as isize \* 2 + 1)

            }

        }

    }

    // функція для виведення на екран цілого рядку

    pub fn print(&mut self, s: &str) {

            // зберігаємо вхідний рядок до поточного стану екрану

            self.add\_text(s.as\_bytes());

            // встановлюємо похицію для запису символу на 0 (початок)

            self.cursor\_write = 0;

            // йдемо по кожному рядку state, виводимо всі рядки по символу

            for row in self.state {

                /\*

                    кількість пробілів, що треба поставити перед початком рядка, щоб дотримуватися

                    типу вирівнювання

                \*/

                let align = self.calc\_align(&row);

                // вирівнюємо рядок відповідною кількістю пробілів

                for i in 0..align {

                    self.write\_char(

                        self.cursor\_write, AsciiChar{char\_byte : b' ', color\_byte: self.color}

                    );

                }

                // виводимо рядок посимвольно

                for c in row {

                    // якщо зустрічаємо знак кінця рядку, припиняємо його виведення

                    if c == b'\0' {

                        break;

                    }

                    // запис символу c на позицію cursor\_write

                    self.write\_char(

                        self.cursor\_write, AsciiChar{char\_byte : c, color\_byte: self.color}

                    );

                }

                // переведення курсору запису на новий рядок

                self.cursor\_write += BUF\_WIDTH - (self.cursor\_write % BUF\_WIDTH);

            }

    }

    // функція для збереження тексту до поточного стану екрану

    pub fn add\_text(&mut self, row: &[u8]) {

        /\*

            \* записуємо текст посимвольно

            \* якщо довжина рядку перевищує ширину екрану або зустрічається символ переходу на новий рядок,

              то переходимо на новий рядок

        \*/

        for i in 0..row.len() {

            if self.cursor\_row == BUF\_WIDTH - 1 || row[i] == b'\n' {

                self.state[self.cursor\_col as usize ][self.cursor\_row as usize] = b'\0';

                self.cursor\_col += 1;

                self.cursor\_row = 0;

                // якщо рядків стає "забагато"

                // то прогортуємо екран на один рядок

                if self.cursor\_col == BUF\_HEIGHT - 1 {

                    self.scroll();

                }

                continue;

            }

            self.state[self.cursor\_col as usize][self.cursor\_row as usize] = row[i];

            self.cursor\_row += 1;

        }

    }

    /\*

        функція, що зміщує всі рядки на один вгору та затирає останній,

        тим самим звільняючи місце для нового рядка

    \*/

    pub fn scroll(&mut self) {

        // зміщуємо рядки на один вгору

        for i in 0..self.cursor\_col - 1 { // ітеруємося по всіх рядках, крім останнього

            self.state[i as usize] = self.state[(i + 1) as usize]; // зміщуємо всі рядки на 1 вгору

        }

        // затираємо останній рядок

        for i in 0..self.state[self.cursor\_col as usize].len() {

            self.state[(self.cursor\_col) as usize][i] = b' ';

        }

        self.cursor\_col -= 1;

    }

    /\*

        функція, що вираховує кількість пробілів,

        необхідну для конкретного типу вирівнювання конкретного рядка

     \*/

    pub fn calc\_align(&self, row: &[u8]) -> u32 {

        let mut len = 0;

        // знаходимо довжину рядка до символу закінчення

        for c in row {

            if \*c == b'\0' {

                break;

            }

            len += 1;

        }

        // відповідно до типу, вираховуємо кількість пробілів для вирівнювання

        match self.align {

            Alignment::Left => 0,

            Alignment::Right => BUF\_WIDTH - len,

            Alignment::Center => (BUF\_WIDTH - len) / 2

        }

    }

}