

Министерство образования и науки Российской Федерации
Санкт-Петербургский Политехнический Университет Петра Великого

Институт кибербезопасности и защиты информации

ЛАБОРАТОРНАЯ РАБОТА № 1

«ПРИНЦИПЫ РАЗРАБОТКИ ОПЕРАЦИОННЫХ СИСТЕМ»

по дисциплине «Операционные системы»

Выполнил
студент гр. 4851003/10002

Галкин К. К.

Руководитель
К. н. т

Крундышев В. М.

Санкт-Петербург

2023

1. ЦЕЛЬ РАБОТЫ

Цель работы — изучение основ разработки ОС, принципов низкоуровневого взаимодействия с аппаратным обеспечением, программирования системной функциональности и процесса загрузки системы.

2. ХОД РАБОТЫ

Перед выполнением работы были поставлены следующие задачи:

- Написать загрузчик ОС в соответствии с вариантом
- Передавать через загрузчик управление ядру ОС
- Написать ядро ОС в соответствии с вариантом (SolverOS)
- Протестировать разработанное решение

Блок – схема и подробное описание принципа работы загрузчика представлено ниже:

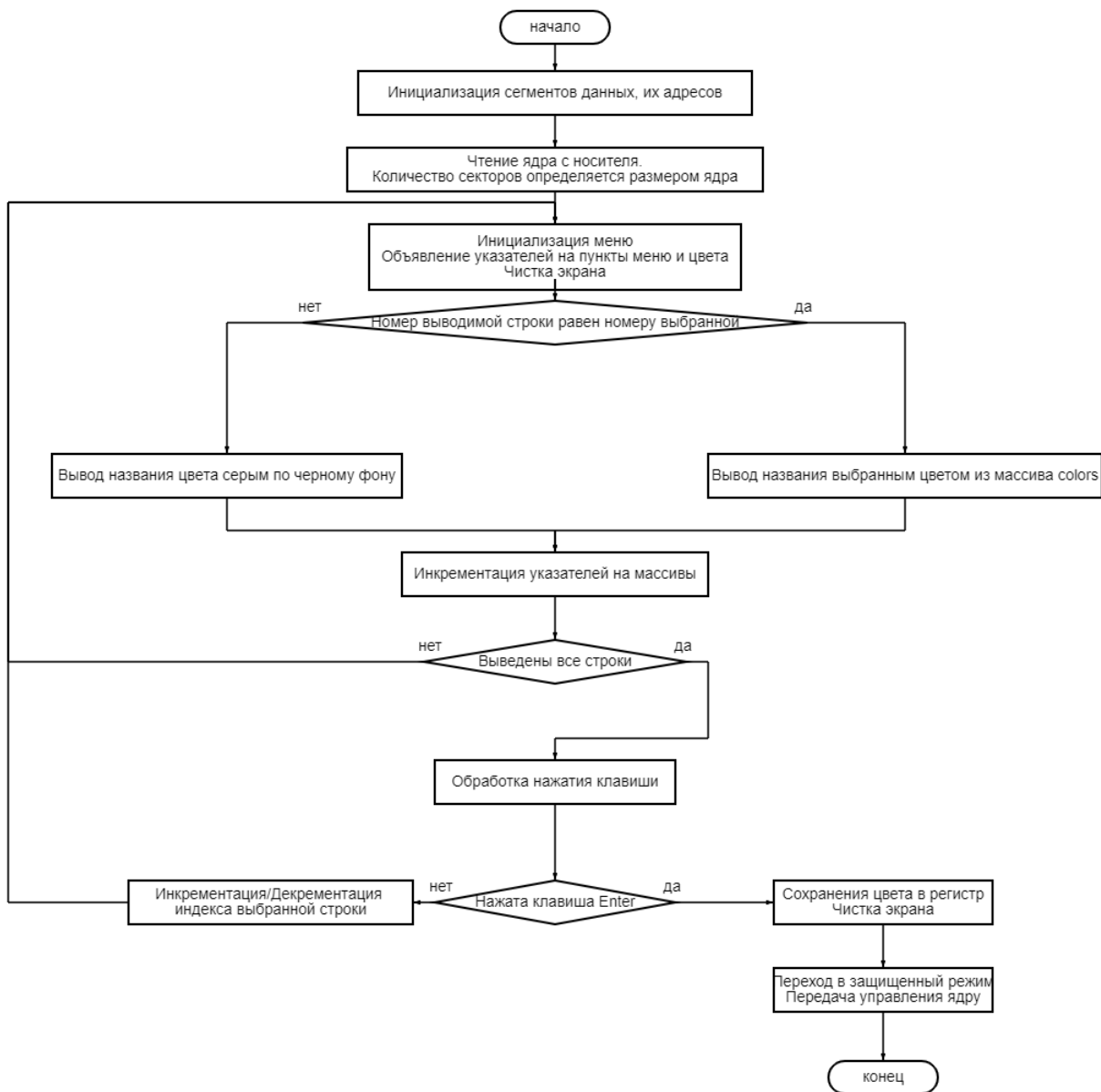


Рисунок 1 Блок-схема работы загрузчика

Несколько подробностей:

- Чистка экрана происходит благодаря прерыванию `int 0x10` и `ax = 3`
- Указатели на пункты меню и массив цветов – регистры `si`, `di` кладутся адреса меток `menu_items` (названия всех цветов) и `colors` (значения байта цвета текста и фона) соответственно.

- Вывод строк происходит посимвольно через функцию `ah = 0x09` и прерывания `int 0x10`. До этого, в прошлых реализациях, была попытка использовать функцию `ah = 0x0E` с тем же прерыванием, но происходила проблема с использованием цвета. Он просто не показывался, даже в графическом режиме.
- При посимвольном выводе используется чтение позиции курсора через `ah = 0x02` и его изменение через функцию `ah = 0x03` и `int 0x10`. При этом, для сохранения данных, использовались команды `pusha` и `popa`, кладущие и забирающие данных регистров со стека
- Нажатия стрелки “вверх” и “вниз”, а так же клавиши `Enter`, происходит через значения регистра `ah` после прерывания `0x16`. Важно отметить, что в регистр `ah` в результате прерывания кладется скан-код нажатой клавиши, а в `al` – само значение. Поэтому можно обрабатывать нажатия через любой из регистров, только значения будут другие.
- Переход в защищенный режим происходит по нажатию клавиши `Enter`. В этом случае происходит прыжок на метку `load_kernel`, где запоминается значение цвета, чистится экран, загружается адрес и размер `gdt` – таблицы в `GDTR` регистр процессора. В таблице содержатся данные о двух сегментах: первый – сегмент кода размером 4 Гб, который начинается с адреса 0, флаг `P = 1` показывает, что сегмент действителен с уровнем привилегий `DPL = 0` и пользовательским режимом (`S = 0`), данный сегмент настроен на чтение и выполнение в 32-битном режиме. Второй сегмент – сегмент данных, инициализация аналогична, правда данных сегмент доступен только на запись.

- После включения GDT-сегмента происходит включение адресной линии A20 через порт 92h, где нужно контролировать только первый разряд, остальные изменять нельзя, и включение CR0 регистра, а конкретнее флага PE (1 бит регистра CR0) в значение 1. Затем идет длинный прыжок, где уже идет передача управления ядру и объявления текущего сегмента в памяти загрузочным.

Пример работы загрузчика можно наблюдать ниже:

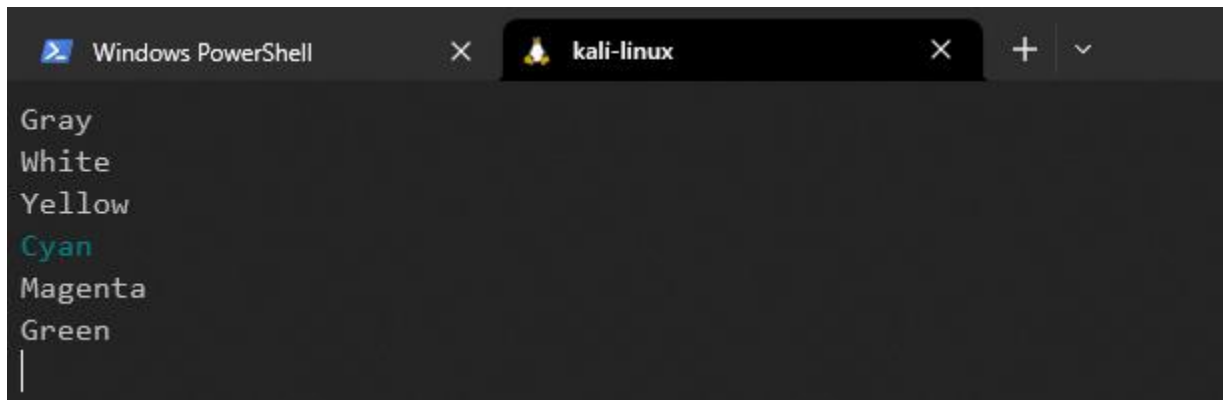


Рисунок 2 Выбор цвета при работе загрузчика

Теперь, при переходе управления ядру, после инициализации прерываний и обработчика нажатия клавиш, значения цвета из регистра bl кладется в переменную color. По условию задания, SolverOS должна обладать возможностью решать линейные уравнения вида $ax+b=c$ и находить НОД двух положительных чисел, не превосходящих максимальное значение типа int (было взято значение unsigned int, поскольку функция работает с положительными значениями). Рассмотрим таблицу, в которой находится описание основных функций ядра ОС, так же на ней будут выделены вспомогательные функции (серым цветом) для работы со строками и обработкой клавиш:

Название функции	Описание
------------------	----------

<code>void on_key(unsigned char scan_code)</code>	Обработка scan_code нажатой клавиши. Проверка на то, что нажатая клавиша является клавишей shift и установка соответствующего флага. Вывод символа, который был нажат на экран.
<code>void backspace_handler()</code>	Обработка нажатия клавиши Backspace. Двигает положение курсора, очищая данные за курсором, до того момента, пока не дойдет до символа ввода команды - #
<code>void read_command()</code>	Чтение команды в строку input_str из видеопамати.
<code>void print_error(const char *error)</code>	Вывод ошибки error и нового символа ввода команды
<code>void command_parse()</code>	Парсинг input_str на команды ОС. Если команда принимает после себя аргументы, то функция записывает их и обрабатывает через вспомогательные функции.
<code>strlen, strcmp, strtok, strrev, swap, _atoi, _itoa</code>	Функции работы со строками из основных библиотек языка Си.
<code>void clear()</code>	Чистка экрана. Изначально была попытка отчистить экран через прерывание 0x10 и функцию <code>ah = 3</code> ,

	но возникли проблемы с отчищаемой областью. Поэтому было принято решение чистить посимвольно, выставляя значения байтов на экране в 0x00.
<code>int gcd(int n1, int n2, char *str)</code>	Нахождение НОД чисел n1 и n2 через расширенный алгоритм Евклида. Для дальнейшей работы предусмотрен возврат результата, как целочисленного значения. Строка с результатом формируется прямо в этой функции и кладется в str. Проверка данных идет до вызова, в функции <code>command_parse</code> .
<code>void solve(int a, int b, int c)</code>	Функция решения линейного уравнения вида $ax+b=c$. Число b переносится вправо с изменением знака. Сначала идет проверка на то, что $\text{right_digit} = b + c$ нацело делится на a. Если да, тогда результат – их целочисленное деление. Если нет, тогда происходит деление каждого из чисел (a, right_digit) на их НОД (функция gcd), пока результат gcd не станет 1. После идет формирование результирующей строки. Проверка

	уравнения так же идет в функции <code>command_parse</code> .
<code>void shutdown()</code>	Отправка команды для отключения питания на ACPI порт 0x604. Важно отметить, что такая реализация команды работает только на новых версиях эмулятора QEMU, для более старых версии нужно использовать другие порты и команды. Если ОС запущена под VmWare, тогда функцию надо полностью менять
<code>void info()</code>	Вывод информации об авторе ОС, используемом компиляторе, синтаксисе и трансляторе.

Тексты программ загрузчика и ядра указаны в приложении

3. РЕЗУЛЬТАТЫ ТЕСТОВ

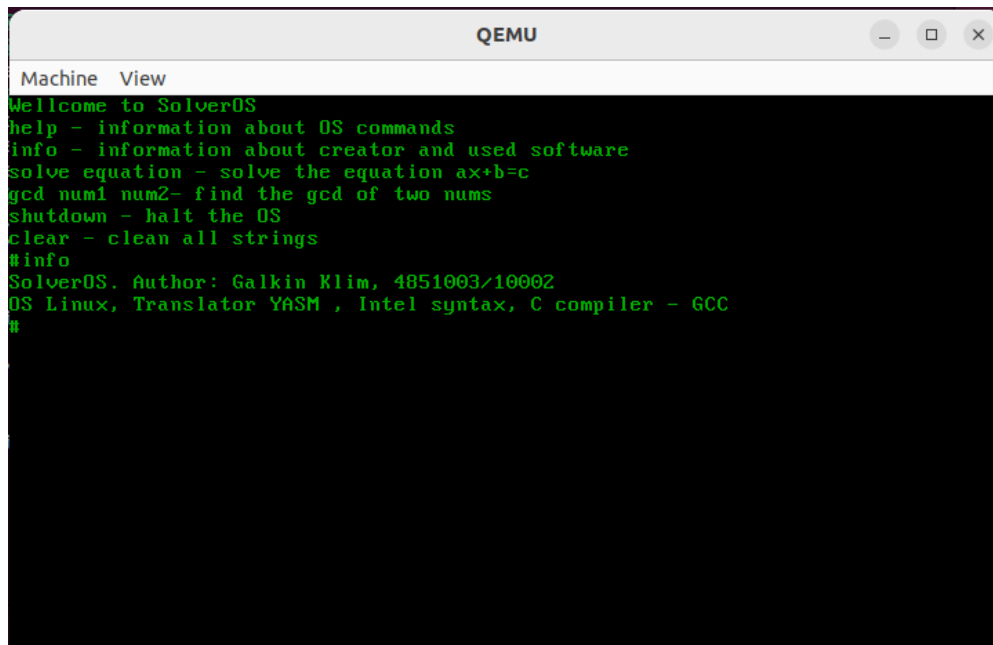


Рисунок 3 Использование функции info

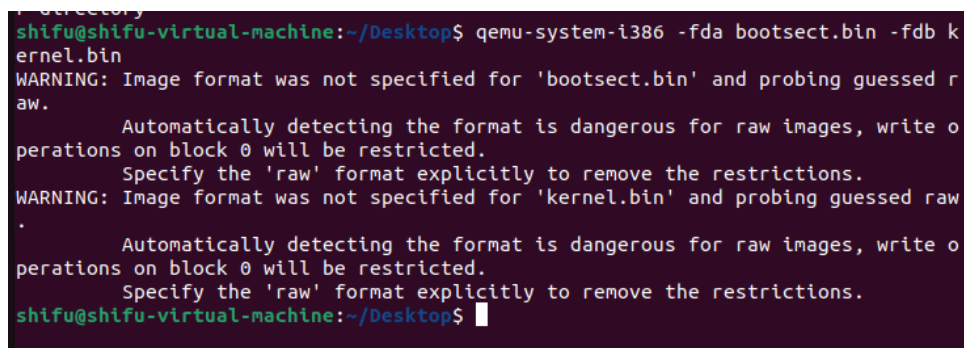


Рисунок 4 Использование функции shutdown.

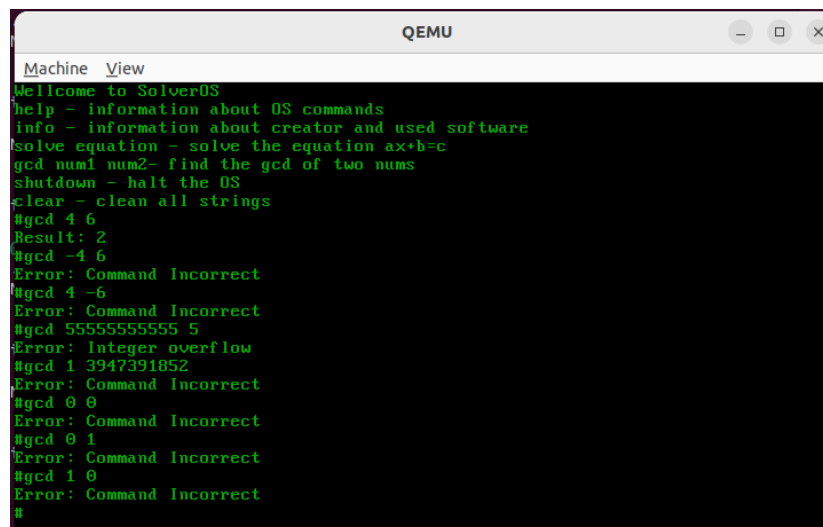
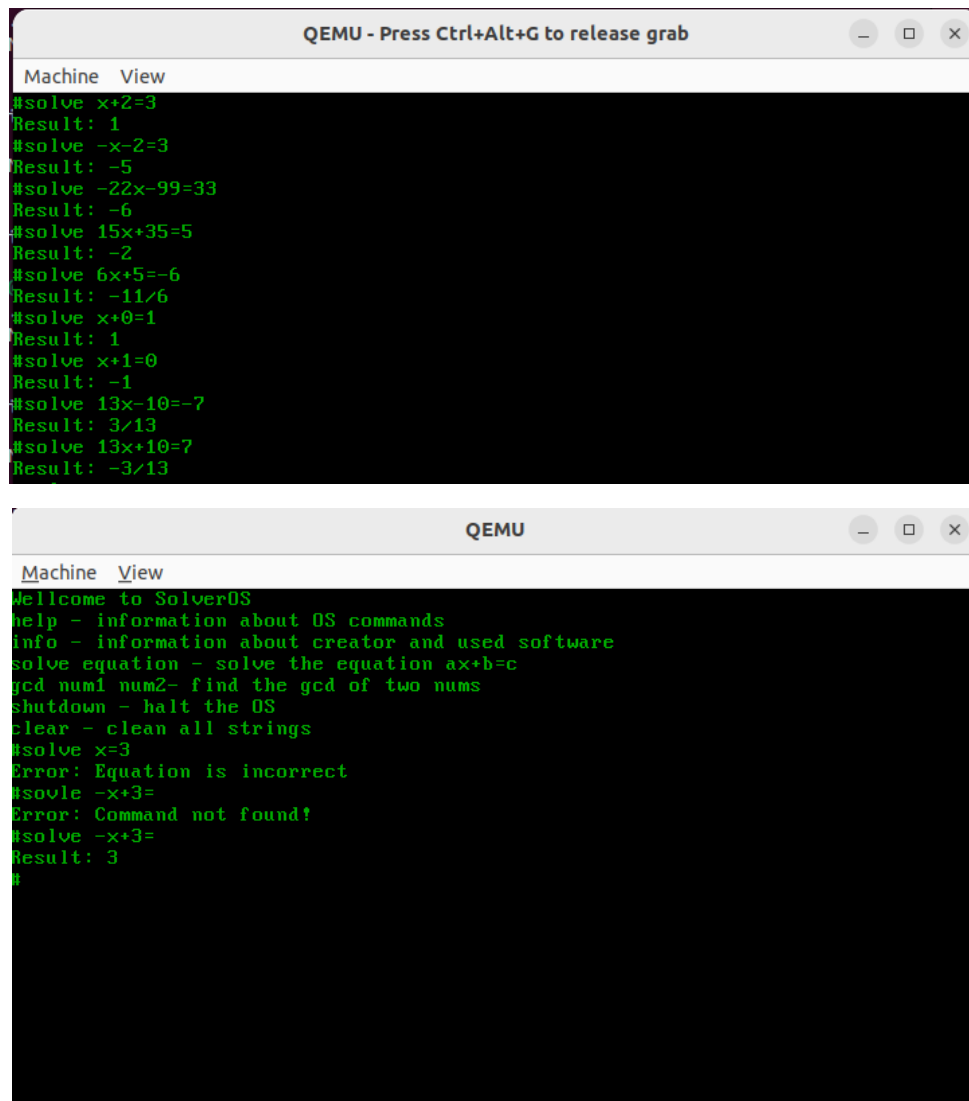


Рисунок 5 Проверка функции gcd и ее аргументов



```
QEMU - Press Ctrl+Alt+G to release grab
Machine View
#solve x+2=3
Result: 1
#solve -x-2=3
Result: -5
#solve -22x-99=33
Result: -6
#solve 15x+35=5
Result: -2
#solve 6x+5=-6
Result: -11/6
#solve x+0=1
Result: 1
#solve x+1=0
Result: -1
#solve 13x-10=-7
Result: 3/13
#solve 13x+10=7
Result: -3/13

QEMU
Machine View
Welcome to SolverOS
help - information about OS commands
info - information about creator and used software
solve equation - solve the equation ax+b=c
gcd num1 num2- find the gcd of two nums
shutdown - halt the OS
clear - clean all strings
#solve x=3
Error: Equation is incorrect
#solve -x+3=
Error: Command not found!
#solve -x+3=
Result: 3
#
```

Рисунок 6 Проверка функции solve

4. ВЫВОД

В ходе выполнения работы были изучены принцип работы загрузчика в памяти, как читаются данные с носителей и как происходит переход в защищенный режим процессора, как происходит передача управления ядру. Были изучены основные прерывания, позволяющие взаимодействовать с компонентами машины (экран, клавиатура и др.), и их аргументы. В ходе разработки ядра поверхностно была изучена теория по ACPI и SMM режимам, которые позволяют управлять состоянием компьютера, посылая на порты

различные команды. Разработанное решение было протестировано на не стандартный ввод, проведено 22 теста.

5. ПРИЛОЖЕНИЕ

Исходный код загрузчика:

[BITS 16]

[ORG 0x7C00]

_start:

; Инициализация сегментов данных и стека

mov ax, cs

mov ds, ax

mov ss, ax

mov sp, _start ; Начало стека - адрес первой команды

; чтение ядра

mov ax, 0x1000

mov es, ax ; размер буфера

mov bx, 0x00 ; адрес буфера(0x0000: 0x1000)

mov ah, 0x02 ; режим чтения

mov dl, 1 ; номер диска

mov dh, 0x00 ; номер головки

mov cl, 0x01 ; номер сектора

mov ch, 0x00 ; номер цилиндра

mov al, 35 ; кол-во секторов. Считается как размер ядра / 512

с округлением в большую сторону

int 0x13

xor ch, ch ; Номер выбранного пункта меню (начинаем с первого)

call menu ; Прыгаем на меню

menu:

; Чистка всего экрана

mov ax, 3

int 0x10

xor dh, dh ; Счетчик цикла

mov di, colors ; Указатель на массив цветов

mov si, menu_items ; Указатель на указатели строк

mov ah, 0x09 ; Вывод символов

mov bl, 0x07 ; Цвет символов (белый на черном фоне)

xor ah, ah

menu_loop:

cmp ch, dh ; Сравниваем

je print_selected ; Если сейчас будет печататься выбранный пункт меню, то прыгаем

jmp print_char ; Иначе просто продолжаем печатать символы с обычным цветом

find_color:

inc ah ; Инкрементируем индекс

```
inc di ; Инкрементируем указатель  
jmp print_selected
```

print_selected:

```
cmp ah, ch ; Ищем индекс цвета в массиве colors в  
соответствии с выбранным пунктом
```

```
jne find_color  
mov bl, byte [di] ; Ставим на bl байт цвета текста  
mov cl, bl  
jmp print_char
```

print_char:

```
mov al, byte [si] ; Выбираем символ из массива  
pusha ; Сохраняем на стек значения регистров  
mov cx, 1 ; Печатаем символ один раз  
mov ah, 0x09 ; Принт символа с цветом  
int 0x10  
popa ; Возврат значения регистров
```

```
; Блок перемещения курсора для корректной работы функции
```

0x09

```
pusha  
mov bh, 0  
mov ah, 0x03  
int 0x10  
inc dl  
mov ah, 0x02
```

int 0x10

popa

inc si ; Бегаем по массиву строк

cmp byte [si], 0x00 ; Пока это не конец строки - печатаем ее

jne print_char

inc si ; Убираем указатель с 0x00 байта

; Блок перемещения курсора для корректной работы функции

0x09

pusha

mov bh, 0

mov ah, 0x03

int 0x10

inc dh

mov dl, 0

mov ah, 0x02

int 0x10

popa

cmp dh, 5 ; Если напечатали последнюю строку - ждем

взаимодействия пользователя

je wait_for_input ; Взаимодействие пользователя

inc dh ; Print string

mov bl, 0x07 ; Восстанавливаем цвет на всякий случай

jmp menu_loop

wait_for_input:

; Ожидание ввода пользователя и обработка нажатий клавиш

"вверх" и "вниз"

mov ah, 0x00 ; Функция чтения клавиши BIOS

int 0x16 ; Вызов прерывания BIOS для чтения клавиши

cmp ah, 0x48 ; Проверка на нажатие клавиши "вверх"

je move_up ; Если это клавиша "вверх", то переходим к
предыдущему пункту меню

cmp ah, 0x50 ; Проверка на нажатие клавиши "вниз"

je move_down ; Если это клавиша "вниз", то переходим к
следующему пункту меню

cmp ah, 0x1C

je load_kernel

jmp wait_for_input ; Если это другая клавиша, то ожидаем
новый ввод

move_up:

cmp ch, 0 ; Проверка на начало списка пунктов меню

je wait_for_input ; Если это начало списка, то ожидаем
новый ввод

dec ch ; Переходим к предыдущему пункту меню

jmp menu ; Выводим меню заново

move_down:

cmp ch, 5 ; Проверка на конец списка пунктов меню

je wait_for_input ; Если это конец списка, то ожидаем новый
ввод

```
inc ch ; Переходим к следующему пункту меню  
jmp menu ; Выводим меню заново
```

colors:

```
db 0x07, 0x0F, 0x0E, 0x03, 0x05, 0x02
```

menu_items:

```
db "Gray", 0  
db "White", 0  
db "Yellow", 0  
db "Cyan", 0  
db "Magenta", 0  
db "Green", 0
```

load_kernel:

```
xor bh, bh  
mov bl, cl ; цвет текста  
mov ax, 3  
int 0x10
```

cli

```
lgdt [gdt_info]
```

```
in al, 0x92
```

```
or al, 2
```

```
out 0x92, al
```



```
mov eax, cr0
or al, 1
mov cr0, eax
jmp 0x8:protected_mode
```

```
gdt:
    db 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
    db 0xff, 0xff, 0x00, 0x00, 0x00, 0x9A, 0xCF, 0x00
    db 0xff, 0xff, 0x00, 0x00, 0x00, 0x92, 0xCF, 0x00
```

```
gdt_info:
    dw gdt_info - gdt
    dw gdt, 0
```

```
; Метка для перехода в защищенный режим
[BITS 32]
```

```
protected_mode:
    mov ax, 0x10
    mov es, ax
    mov ds, ax
    mov ss, ax
```

```
call 0x10000
```

```
times (512 - ($ - _start) - 2) db 0
db 0x55, 0xAA
```

Исходный код ядра:

```
__asm("jmp kmain");
```

```
#define VIDEO_BUF_PTR (0xb8000)
```

```
#define IDT_TYPE_INTR (0x0E)
```

```
#define IDT_TYPE_TRAP (0x0F)
```

```
// Селектор секции кода, установленный загрузчиком ОС
```

```
#define GDT_CS (0x8)
```

```
#define PIC1_PORT (0x20)
```

```
#define NULL 0
```

/*Базовый порт управления курсором текстового экрана. Подходит для

большинства, но может отличаться в других BIOS и в общем случае адрес

должен быть прочитан из BIOS data area.*/

```
#define CURSOR_PORT (0x3D4)
```

```
#define VIDEO_WIDTH (80) // Ширина текстового экрана
```

```
#define ACPI_PORT 0x604
```

```
#define ACPI_CMD 0x2000
```

```
#define UINT_MAX 0xffffffff
```

```
#define HEIGHT 25
```

```
#define CURSOR_COLOR 0x02
```

```
// Структура описывает данные об обработчике прерывания
```

```
struct idt_entry
```

```
{
```

```
    unsigned short base_lo; // Младшие биты адреса обработчика
```

```
unsigned short segm_sel; // Селектор сегмента кода
unsigned char always0; // Этот байт всегда 0
unsigned char flags;      // Флаги тип. Флаги: P, DPL, Типы - это
константы - IDT_TYPE...
```

```
    unsigned short base_hi; // Старшие биты адреса обработчика
} __attribute__((packed)); // Выравнивание запрещено
```

```
// Структура, адрес которой передается как аргумент команды lidt
```

```
struct idt_ptr
{
    unsigned short limit;
    unsigned int base;
} __attribute__((packed)); // Выравнивание запрещено
```

```
typedef void (*intr_handler)();
```

```
struct idt_entry g_idt[256]; // Реальная таблица IDT
struct idt_ptr g_idtp;      // Описатель таблицы для команды lidt
```

```
// https://wiki.osdev.org/PS/2\_Keyboard#Scan\_Code\_Set\_1
```

```
unsigned char no_shift_codes[58] = {
    0, 0, '1', '2', '3', '4', '5', '6', '7', '8', '9', '0', '-', '=', 0, 0,
    'q', 'w', 'e', 'r', 't', 'y', 'u', 'i', 'o', 'p', '[', ']', 0, 0,
    'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l', ';', '\'', '`', 0,
    '\\', 'z', 'x', 'c', 'v', 'b', 'n', 'm', ',', '.', '/', 0, 0, 0, ' '};
```

```
unsigned char shift_on_codes[58] = {
```

```
0, 0, '!', '@', '#', '$', '%', '^', '&', '*', '(', ')', '_', '+', 0, 0,
'Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O', 'P', '{', '}', 0, 0,
'A', 'S', 'D', 'F', 'G', 'H', 'J', 'K', 'L', ':', '\", '~', 0,
'|', 'Z', 'X', 'C', 'V', 'B', 'N', 'M', '<', '>', '?', 0, 0, 0, ' '};
```

```
const char *welcome = "Wellcome to SolverOS";    // Welcome message
const char *default_err = "Error: Command not found!"; // Default
Command message
```

```
const char *gcd_incorrect = "Error: Command Incorrect"; // GCD function
error if integers less than 0
```

```
const char *gcd_overflow = "Error: Integer overflow"; // GCD function
error if integer above than UINT_MAX
```

```
const char *equation_incorrect = "Error: Equation is incorrect";
```

```
int color;           // Color value form bootsector
```

```
unsigned char shift = 0;    // Is shift pressed?
```

```
unsigned int current_string = 0; //
```

```
unsigned int cursor_position = 0;
```

```
unsigned int string_len = 0;
```

```
char input_str[41];
```

```
// Interrupt functions
```

```
void intr_reg_handler(int num, unsigned short segm_sel, unsigned short
flags, intr_handler hndlr);
```

```
void intr_init();
```

```
void intr_start();
```

```
void intr_enable();
```

```
void intr_disable();
void default_intr_handler();

// Keyboard functions
void keyb_handler();
void keyb_init();
static inline unsigned char inb(unsigned short port);
static inline void outb(unsigned short port, unsigned char data);
void keyb_process_keys();
void backspace_handler();
void on_key(unsigned char scan_code);

// Cursor position function
void cursor_moveto(unsigned int strnum, unsigned int pos);

// Command parsing functions
void command_parse();
void read_command();

// Convert functions
char *_itoa(int num, char *str, int base);
unsigned long long int _atoi(char *str);

// String and print functions
void out_str(int color, const char *ptr);
void out_chr(int color, unsigned char ptr);
void strrev(char str[], int length);
```

```
int strcmp(const char *a, const char *b);
int strlen(const char *a);
char *strtok(char *s, const char *delim);
char *strtok_r(char *s, const char *delim, char **last);
void swap(char &a, char &b);
```

```
// OS functions
```

```
void solve(int a, int b, int c);
int gcd(int n1, int n2, char *str);
void shutdown();
void info();
void help();
void clear();
```

```
void intr_reg_handler(int num, unsigned short segm_sel, unsigned short
flags, intr_handler hndlr)
```

```
{
    unsigned int hndlr_addr = (unsigned int)hndlr;
    g_idt[num].base_lo = (unsigned short)(hndlr_addr & 0xFFFF);
    g_idt[num].segm_sel = segm_sel;
    g_idt[num].always0 = 0;
    g_idt[num].flags = flags;
    g_idt[num].base_hi = (unsigned short)(hndlr_addr >> 16);
}
```

```
// Функция инициализации системы прерываний: заполнение массива
с адресами обработчиков
```

```
void intr_init()
```

```

{
    int i;
    int idt_count = sizeof(g_idt) / sizeof(g_idt[0]);
    for (i = 0; i < idt_count; i++)
        intr_reg_handler(i,    GDT_CS,    0x80    |    IDT_TYPE_INTR,
default_intr_handler); // segm_sel=0x8, P=1, DPL=0, Type=Intr
}

```

```

void intr_start()

```

```

{
    int idt_count = sizeof(g_idt) / sizeof(g_idt[0]);
    g_idtp.base = (unsigned int>(&g_idt[0]));
    g_idtp.limit = (sizeof(struct idt_entry) * idt_count) - 1;
    asm("lidt %0" ::"m"(g_idtp));
}

```

```

void intr_enable()

```

```

{
    asm("sti");
}

```

```

void intr_disable()

```

```

{
    asm("cli");
}

```

```

void keyb_handler()

```

```

{
    asm("pusha");
    // Обработка поступивших данных
    keyb_process_keys();
    // Отправка контроллеру 8259 нотификации о том, что прерывание
    обработано
    outb(PIC1_PORT, 0x20);
    asm("popa; leave; iret");
}

void default_intr_handler()
{
    asm("pusha");
    // ... (реализация обработки)
    asm("popa; leave; iret");
}

void keyb_init()
{
    // Регистрация обработчика прерывания
    intr_reg_handler(0x09,    GDT_CS,    0x80    |    IDT_TYPE_INTR,
keyb_handler);
    // segm_sel=0x8, P=1, DPL=0, Type=Intr
    // Разрешение только прерываний клавиатуры от контроллера 8259
    outb(PIC1_PORT + 1, 0xFF ^ 0x02); // 0xFF - все прерывания, 0x02 -
бит IRQ1 (клавиатура).
    // Разрешены будут только прерывания, чьи биты установлены в 0

```



```
}
```

```
static inline unsigned char inb(unsigned short port) // Чтение из порта
```

```
{
```

```
    unsigned char data;
```

```
    asm volatile("inb %w1, %b0"
```

```
        : "=a"(data)
```

```
        : "Nd"(port));
```

```
    return data;
```

```
}
```

```
static inline void outb(unsigned short port, unsigned char data) // Запись
```

```
{
```

```
    asm volatile("outb %b0, %w1" :: "a"(data), "Nd"(port));
```

```
}
```

```
void swap(char &a, char &b)
```

```
{
```

```
    char temp = a;
```

```
    a = b;
```

```
    b = temp;
```

```
}
```

```
int strlen(const char *str)
```

```
{
```

```
    const char *s;
```

```
    for (s = str; *s; ++s)
```

```

        ;
    return (s - str);
}

```

```

int strcmp(const char *s1, const char *s2)
{
    for (; *s1 == *s2; s1++, s2++)
        if (*s1 == '\0')
            return 0;
    return ((*s1 < *s2) ? -1 : +1);
}

```

```

char *strtok(char *s, const char *delim)
{
    static char *last;
    return strtok_r(s, delim, &last);
}

```

```

char *strtok_r(char *s, const char *delim, char **last)
{
    char *spanp;
    int c, sc;
    char *tok;
    if (s == NULL && (s = *last) == NULL)
        return (NULL);
cont:
    c = *s++;

```

```

for (spanp = (char *)delim; (sc = *spanp++) != 0;)
{
    if (c == sc)
        goto cont;
}
if (c == 0)
{ /* no non-delimiter characters */
    *last = NULL;
    return (NULL);
}
tok = s - 1;
for (;;)
{
    c = *s++;
    spanp = (char *)delim;
    do
    {
        if ((sc = *spanp++) == c)
        {
            if (c == 0)
                s = NULL;
            else
                s[-1] = 0;
            *last = s;
            return (tok);
        }
    } while (sc != 0);
}

```

```
    }  
}
```

```
void strrev(char str[], int length)  
{  
    int start = 0;  
    int end = length - 1;  
    while (start < end)  
    {  
        swap(*(str + start), *(str + end));  
        start++;  
        end--;  
    }  
}
```

```
void cursor_moveto(unsigned int strnum, unsigned int pos)  
{  
    if (pos < 1 || pos > string_len || pos > 79)  
        return;  
    unsigned short new_pos = (strnum * VIDEO_WIDTH) + pos;  
    outb(CURSOR_PORT, 0x0F);  
    outb(CURSOR_PORT + 1, (unsigned char)(new_pos & 0xFF));  
    outb(CURSOR_PORT, 0x0E);  
    outb(CURSOR_PORT + 1, (unsigned char)((new_pos >> 8) & 0xFF));  
    cursor_position = pos;  
}
```

```

void out_str(int color, const char *ptr)
{
    unsigned char *video_buf = (unsigned char *)VIDEO_BUF_PTR;
    video_buf += 80 * 2 * current_string;
    while (*ptr)
    {
        video_buf[0] = (unsigned char)*ptr; // Символ (код)
        video_buf[1] = color;           // Цвет символа и фона
        video_buf += 2;
        ptr++;
    }
    if (current_string++ >= 25)
    { // Clear if we fill the window
        clear();
        out_chr(color, (unsigned char) '#');
    }
    string_len = 0;
    cursor_moveto(current_string, 0);
}

```

```

void out_chr(int color, unsigned char char_code)
{
    unsigned char *video_buf = (unsigned char *)VIDEO_BUF_PTR;
    video_buf += 80 * 2 * current_string + cursor_position * 2;
    video_buf[0] = char_code;
    video_buf[1] = color;
    if (string_len <= cursor_position)

```

```

        string_len++;
        cursor_moveto(current_string, cursor_position + 1);
    }

void on_key(unsigned char scan_code)
{
    if (scan_code <= 58)
    {
        switch (scan_code)
        {
            case 42:
                shift = 1;
                break;
            case 54:
                shift = 1;
                break;

            default:
                if (shift)
                {
                    if (shift_on_codes[scan_code])
                        out_chr(color, shift_on_codes[scan_code]);
                }
                else
                {
                    if (no_shift_codes[scan_code])
                        out_chr(color, no_shift_codes[scan_code]);
                }
            }
        }
    }
}

```

```

        }
        break;
    }
}
}

```

/* Обработка нажатия Backspace. Отчистка символов, которые стоят
после курсора. */

```

void backspace_handler()
{
    unsigned char *video_buf = (unsigned char *)VIDEO_BUF_PTR;
    video_buf += 80 * 2 * current_string + 2 * cursor_position;
    if (cursor_position == 1)
        return;
    video_buf = video_buf - 2;
    video_buf[0] = 0;
    cursor_moveto(current_string, cursor_position - 1);
    string_len--;
    return;
}

```

/* Чтение команды из видеопамяти в переменную. */

```

void read_command()
{
    unsigned char *video_buf = (unsigned char *)VIDEO_BUF_PTR;
    video_buf += 80 * 2 * current_string + 2;
    int tmp = 0;

```

```

while (tmp < string_len)
{
    input_str[tmp] = video_buf[0];
    video_buf += 2;
    tmp++;
}
input_str[tmp] = '\0';
}

```

/* Вывод нужной ошибки */

```

void print_error(const char *error)
{
    out_str(color, error);
    out_chr(color, (unsigned char) '#');
}

```

/* Парсинг введенной команды и ее аргументов, если таковые есть */

```

void command_parse()
{
    char delim[] = " ";
    char *command_part = strtok(input_str, delim);
    if (strcmp(command_part, "info") == 0)
    {
        info();
    }
    else if (strcmp(command_part, "help") == 0)
    {

```



```

        help();
    }
    else if (strcmp(command_part, "clear") == 0)
    {
        clear();
        out_chr(color, (unsigned char) '#');
    }
    else if (strcmp(command_part, "shutdown") == 0)
    {
        shutdown();
    }
    else if (strcmp(command_part, "gcd") == 0)
    {
        char operation_number = 0;
        int n1, n2;
        int check_status = 0;
        while (command_part != NULL)
        {
            switch (operation_number)
            {
                case 0:
                    if (strcmp(command_part, "gcd") != 0)
                        print_error(gcd_incorrect);
                    else
                        check_status++;
                    break;
                case 1:

```

```

n1 = _atoi(command_part);
if (n1 <= UINT_MAX && strlen(command_part) >= 10)
{
    print_error(gcd_overflow);
    return;
}
else if (n1 <= 0)
{
    print_error(gcd_incorrect);
    return;
}
else
    check_status++;
break;
case 2:
n2 = _atoi(command_part);
if (n2 == 0 && strlen(command_part) >= 10)
{
    print_error(gcd_overflow);
    return;
}
else if (n2 <= 0)
{
    print_error(gcd_incorrect);
    return;
}
else

```

```

        check_status++;
        break;
default:
        break;
    }
    operation_number++;
    command_part = strtok(NULL, delim);
}
if (check_status == 3)
{
    char result_str[110] = "Result: ";
    gcd(n1, n2, result_str);
    out_str(color, result_str);
    out_chr(color, (unsigned char) '#');
}
}
else if (strcmp(command_part, "solve") == 0)
{
    command_part = strtok(NULL, delim);
    int a, b, c;
    char parameter[50];
    char operation_number = 0;
    int i = 0, param_index = 0;
    if (strlen(command_part) < 5)
    {
        print_error(equation_incorrect);
        return;
    }

```

```

    }
    if (command_part[0] == 'x')
    {
        a = 1;
        i++;
    }
    else
    {
        while (command_part[i] != 'x')
        {
            parameter[param_index] = command_part[i];
            param_index++;
            i++;
        }
        parameter[param_index] = '\0';
        if (param_index == 1 && parameter[0] == '-')
            a = -1;
        else
            a = _atoi(parameter);
    }
    i++;
    param_index = 0;
    while (command_part[i] != '=')
    {
        parameter[param_index] = command_part[i];
        param_index++;
        i++;
    }

```

```

    }
    parameter[param_index] = '\0';
    b = _atoi(parameter);
    param_index = 0;
    i++;
    while (i != strlen(command_part))
    {
        parameter[param_index] = command_part[i];
        param_index++;
        i++;
    }
    parameter[param_index] = '\0';
    c = _atoi(parameter);
    param_index = 0;
    if (a != 0)
        solve(a, b, c);
    else
        print_error(equation_incorrect);
    a = 0;
    b = 0;
    c = 0;
}
else
{
    out_str(color, default_err);
    out_chr(color, (unsigned char) '#');
}

```

```
}
```

```
/* Обработка всех нажатий, проверка на нажатие клавиши enter или  
backspace */
```

```
void keyb_process_keys()
```

```
{
```

```
    // Проверка что буфер PS/2 клавиатуры не пуст (младший бит  
присутствует)
```

```
    if (inb(0x64) & 0x01)
```

```
    {
```

```
        unsigned char scan_code;
```

```
        unsigned char state;
```

```
        scan_code = inb(0x60); // Считывание символа с PS/2 клавиатуры
```

```
        if (scan_code < 128)
```

```
        {
```

```
            // Обработка backspace
```

```
            if (scan_code == 14)
```

```
            {
```

```
                backspace_handler();
```

```
            }
```

```
            if (scan_code == 28)
```

```
            {
```

```
                if (string_len == 1)
```

```
                    return;
```

```
                read_command();
```

```
                current_string++;
```

```
                cursor_position = 0;
```

```

        string_len = 0;
        command_parse();
    }
    on_key(scan_code);
}
else // Скан-коды выше 128 - это отпущение клавиши
{
    if (scan_code == 170 || scan_code == 182) // Shift key up
        shift = 0;
    }
}
}

```

/* Перевод числа в строку, учитывая знак числа */

```
char *_itoa(int num, char *str, int base)
```

```

{
    int i = 0;
    int is_negative = 0;

```

```
    if (num == 0)
```

```

    {
        str[i++] = '0';
        str[i] = '\0';
        return str;
    }

```

```
    if (num < 0 && base == 10)
```

```

    {
        is_negative = 1;
        num = -num;
    }
    while (num != 0)
    {
        int rem = num % base;
        str[i++] = (rem > 9) ? (rem - 10) + 'a' : rem + '0';
        num = num / base;
    }

    if (is_negative)
        str[i++] = '-';
    str[i] = '\0';
    strrev(str, i);
    return str;
}

/* Перевод строки в число с учетом знака */
unsigned long long int _atoi(char *str)
{
    long long int res = 0;
    long long int sign = 1;
    int i = 0;
    if (str[0] == '+')
    {
        sign = 1;

```



```

        i++;
    }
    else if (str[0] == '-')
    {
        sign = -1;
        i++;
    }
    for (; str[i] != '\0'; ++i)
        res = res * 10 + str[i] - '0';
    return sign * res;
}

```

/* Нахождение НОД двух положительных чисел */

```

int gcd(int n1, int n2, char *str)
{
    static int result;
    int temp;
    while (n2 != 0)
    {
        temp = n1 % n2;

        n1 = n2;
        n2 = temp;
    }
    result = n1;
    char solution[100];
    _itoa(n1, solution, 10);
}

```

```

int num_index = 0;
int res_index = strlen(str);

while (num_index != strlen(solution))
{
    str[res_index] = solution[num_index];
    num_index++;
    res_index++;
}
// result_str[res_index] = '\0';
return result;
}

```

/* Решение уравнения вида $ax+b=c$ */

```

void solve(int a, int b, int c)
{
    char result_str[110] = "Result: ";
    char solution[100] = "";
    char a_str[50] = "";
    char right_str[50] = "";
    int delim = 0;
    int sign = -1;
    int right_digit = c + (b * sign);
    int correct_result = 0;
    int append_str_index = 0;
    int result_str_index = strlen(result_str);

```

```

if (right_digit % a == 0)
{
    correct_result = right_digit / a;
    _itoa(correct_result, right_str, 10);
    while (append_str_index != strlen(right_str))
    {
        result_str[result_str_index] = right_str[append_str_index];
        append_str_index++;
        result_str_index++;
    }
}
else
{
    delim = gcd(a, right_digit, NULL);
    while (delim != 1)
    {
        a /= delim;
        right_digit /= delim;
        delim = gcd(a, right_digit, NULL);
    }
    _itoa(a, a_str, 10);
    _itoa(right_digit, right_str, 10);

    while (append_str_index != strlen(right_str))
    {
        result_str[result_str_index] = right_str[append_str_index];
        append_str_index++;
    }
}

```

```

        result_str_index++;
    }

    append_str_index = 0;
    result_str[result_str_index] = '/';
    result_str_index++;

    while (append_str_index != strlen(a_str))
    {
        result_str[result_str_index] = a_str[append_str_index];
        append_str_index++;
        result_str_index++;
    }
}
result_str[result_str_index] = '\0';
out_str(color, result_str);
out_chr(color, (unsigned char) '#');
}

/* Выключение компьютера на эмуляции QEMU последней версии */
void shutdown()
{
    asm volatile("outw %0, %1"
        :
        : "a"((unsigned short)ACPI_CMD), "d"((unsigned
short)ACPI_PORT)); // отправляем команду ACPI на порт
}

```



```

    "mov $0x03, %ah\n" // загрузка значения 3 в регистр ah
    "int $0x10\n"      // вызов прерывания 0x10
);*/
unsigned char *video_buf = (unsigned char *)VIDEO_BUF_PTR;
for (int i = 0; i < VIDEO_WIDTH * HEIGHT; i++)
{
    *(video_buf + i * 2) = '\0';
}
current_string = 0;
cursor_position = 0;
string_len = 0;
return;
}

```

```

extern "C" int kmain()
{
    intr_init();
    keyb_init();
    intr_start();
    intr_enable();
    asm("\t movl %%ebx, %0"
        : "=r"(color));
    // Вывод строки
    clear();
    out_str(color, welcome);
    help();
    // Бесконечный цикл

```

```
while (1)
{
    asm("hlt");
}
return 0;
}
```

Скрипт сборки загрузчика и ядра

```
cat configure.sh
```

```
yasm -f bin bootsec.asm -o bootsec.bin
```

```
g++ -ffreestanding -m32 -o kernel.o -c kernel.cpp -fno-pie
```

```
ld --oformat binary -Ttext 0x10000 -o kernel.bin --entry=kmain -m
elf_i386 kernel.o
```