1. Министерство образования и науки Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. Институт кибербезопасности и защиты информации

**ЛАБОРАТОРНАЯ РАБОТА № 1**

**Принципы разработки операционных систем**

по дисциплине «Операционные системы»

1. Выполнила
2. студентка гр. 4851003/90001 Кулеева А.Г.
3. Руководитель
4. ассистент Крундышев В.М.
6. Санкт-Петербург
7. 2021

# цель работы

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

# ход работы

## Изучение методических рекомендаций

Вариант задания представлен на Рисунке 1.

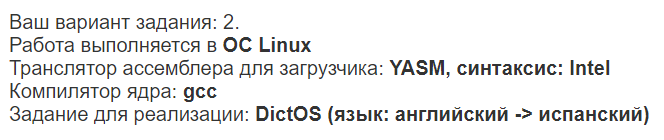


Рисунок 1 ― Вариант задания

Были изучены исходные коды, представленные в методических рекомендациях, и протестированы на виртуальном компьютере. Для начала был запущен файл start.bin, дабы проверить корректность настройки установленных утилит (gcc-multilib, g++, g++-multilib, yasm). Данная примитивнейшая программа является примером загрузки компьютера с дискеты. Она инициализирует адреса сегментов и выводит сообщение “Hello” посредством прерывания 0х10. Пример работы данной программы представлен на Рисунке 2.

; Внимание! Сектор будет считаться загрузочным, если содержит в конце своих 512 байтов два следующих байта: 0x55 и 0xAA

times (512 - ($ - start) - 2) db 0 ; Заполнение нулями до границы 512 - 2 текущей точки

db 0x55, 0xAA ; 2 необходимых байта чтобы сектор считался загрузочным

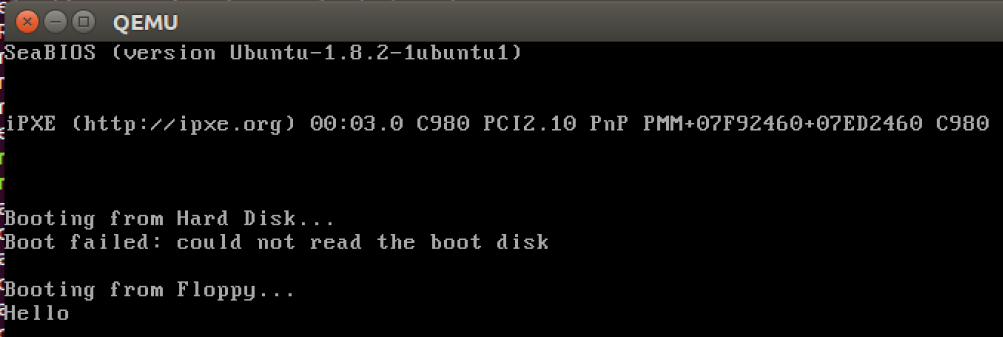


Рисунок 2 ― Запуск start.bin

Далее было начато более доскональное изучения загрузчика операционной системы и его функций. Во-первых, было реализовано считывание заданного количества секторов с диска посредством прерывания BIOS 0х13 и функции 0х02.

mov bx, 0x1000

mov es, bx

xor bx, bx

mov al, 0x30 ;num of sectors to read

mov dl, 0x01 ;drive

mov dh, 0x00 ;head

mov ch, 0x00 ;cylinder

mov cl, 0x01 ;sector

mov ah, 0x02 ;function for int 0x13

int 0x13 ;read data from 0x1000

Далее был произведён переход процессора в защищённый режим. При включении компьютера процессор находится в реальном режиме, однако загрузчик обязан перевести его в защищённый режим. Реальный режим работы процессора предназначен для совместимости с предыдущими моделями процессоров и механизмы защиты в нем отсутствуют. Защищенный режим — основной режим процессора, который должны использовать все современные ОС и который позволяет использовать все возможности процессора (32-битную адресацию памяти, все инструкции, аппаратную многозадачность, защиту). За переход в защищённый режим отвечает бит PE регистра CR0. Перед его установкой необходимо отключить прерывания, включить адресную линию А20, а также загрузить структуру таблицы дескрипторов.

; Отключение прерываний

cli

; Загрузка размера и адреса таблицы дескрипторов

lgdt [gdt\_info] ; Для GNU assembler должно быть "lgdt gdt\_info"

; Включение адресной линии А20

in al, 0x92

or al, 2

out 0x92, al

; Установка бита PE регистра CR0 - процессор перейдет в защищенный режим

mov eax, cr0

or al, 1

mov cr0, eax

jmp 0x8:protected\_mode ; "Дальний" переход для загрузки корректной информации в cs (архитектурные особенности не позволяют этого сделать напрямую).

Основной структурой данных защищенного режима является дескриптор, который имеет размер восемь байт данных и используется для контроля сегментации, аппаратной многозадачности, прерываний. Сегментация обязательно должна использоваться в защищенном режиме. Суть данного механизма заключается в том, что все доступное адресное пространство процессора разделяется на защищенные части, которые называются сегментами и в дальнейшем каждое обращение к памяти осуществляется через один из сегментов. Защита сегментов происходит благодаря распределению уровней привилегий, например, код ядра располагается в сегменте с самым высоким уровнем привилегий, а пользовательская программа ― в самом низко привилегированном сегменте. Таким образом, ядро имеет право вмешаться в пользовательский процесс, обратное неверно.

Итак, для перевода процессора в защищённый режим необходимо инициализировать глобальную таблицу дескрипторов. Минимальная её реализация должна состоять из трех записей: нулевой дескриптор, дескриптор, описывающий область памяти, в которой размещен программный код, дескриптор, описывающий область памяти, в которой размещены данные. Следующий пример демонстрирует описание синтаксисом Intel минимальной таблицы дескрипторов, необходимой для переключения процессора в защищенный режим. Каждый сегмент имеет базу 0 и лимит 4 Гб, что покрывает всю адресуемую процессором память.

gdt:

; Нулевой дескриптор

db 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00

; Сегмент кода: base=0, size=4Gb, P=1, DPL=0, S=1(user),

; Type=1(code), Access=00A, G=1, B=32bit

db 0xff, 0xff, 0x00, 0x00, 0x00, 0x9A, 0xCF, 0x00

; Сегмент данных: base=0, size=4Gb, P=1, DPL=0, S=1(user),

; Type=0(data), Access=0W0, G=1, B=32bit

db 0xff, 0xff, 0x00, 0x00, 0x00, 0x92, 0xCF, 0x00

gdt\_info: ; Данные о таблице GDT (размер, положение в памяти)

dw gdt\_info - gdt ; Размер таблицы (2 байта)

dw gdt, 0 ; 32-битный физический адрес таблицы.

После перехода в защищённый режим основные функции загрузчика выполнены, и управление может быть передано ядру. Для этого необходимо сохранить некоторые загруженные сегменты через регистры и передать управление по адресу 0х10000, на котором должно храниться ядро.

use32

protected\_mode:

; Загрузка селекторов сегментов для стека и данных в регистры

mov ax, 0x10 ; Используется дескриптор с номером 2 в GDT

mov es, ax

mov ds, ax

mov ss, ax

; Передача управления загруженному ядру

call 0x10000 ; Адрес равен адресу загрузки в случае если ядро скомпилировано в "плоский" код

Для проверки корректности написания базового загрузчика можно использовать примеры минимальной реализации ядра из методических указаний, например, ядро на языке ассемблера. Приведённый ниже тривиальный код печатает сообщение "Welcome to HelloWorldOS (asm edition)!", а кроме того переводит процессор в бесконечный цикл. Особое внимание здесь представляет функция video\_puts, которая и печатает сообщение на экран. Если строка равна нулю, то происходит условный переход на метку video\_puts\_end, которая возвращается из функции печати. Иначе в регистр ah кладётся атрибут цвета. Затем очередной символ строки кладётся по адресу регистра edi, цвет записывается по адресу edi + 1. После чего значение регистра увеличивается на два (1 байт на символ, один байт на цвет), а значение регистра esi, хранящего строку, увеличивается на 1 (сдвигаемся на один символ по строке). Изначально в edi хранится значение 0xb8000 ― это адрес начала буфера видеопамяти. На Рисунке 3 представлена успешная проверка работоспособности данного примера.

use32 ; 32-битный код

org 0x10000 ; Адрес, по которому будет загружен этот код загрузчиком ядра

mov edi, 0xb8000 ; В регистр edi помещается адрес начала буфера видеопамяти.

mov esi, str\_hello ; В регистр esi помещается адрес начала строки

call video\_puts ; Вызывается функция для видеовывода в буфер памяти видеокарты

infinite\_loop:

; Перевод процессора в бесконечный цикл

hlt

jmp infinite\_loop

video\_puts:

; Функция выводит в буфер видеопамяти (передается в edi) строку, оканчивающуюся 0 (передается в esi)

; После завершения edi содержит адрес по которому можно продолжать вывод следующих строк

mov al, [esi]

test al, al

jz video\_puts\_end

mov ah, 0x1F ; Цвет символа и фона. Возможные варианты: 0x00 is black-on-black, 0x07 is lightgrey-on-black, 0x1F is white-on-blue

mov [edi], al

mov [edi+1], ah

add edi, 2

add esi, 1

jmp video\_puts

video\_puts\_end:

ret

str\_hello:

db "Welcome to HelloWorldOS (asm edition)!", 0

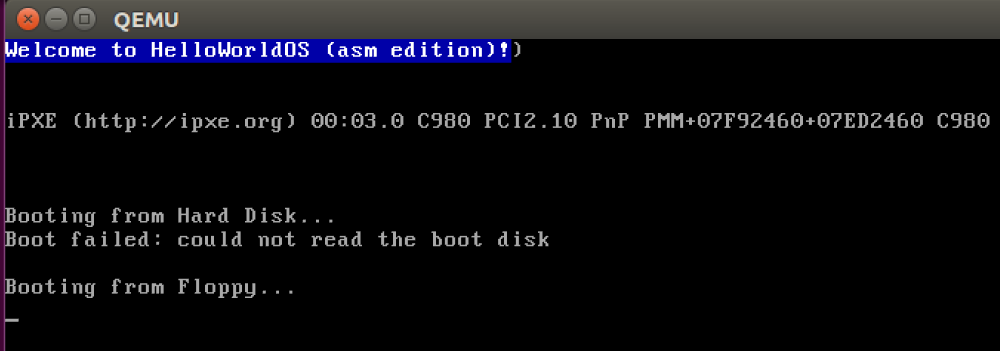


Рисунок 3 ― Тестирование загрузчика с помощью ядра на ассемблере

После проверки реализации загрузчика на корректность можно приступать к разработке ядра. В методических рекомендациях к данной лабораторной работе также есть минимальная реализация ядра на языке С++. Изучим основные функции, необходимые для дальнейшей реализации индивидуального варианта.

В данном примере реализована функция печати сообщения в видеобуфер. Создаётся массив буфера, начало которого находится по уже упомянутому выше адресу 0xb8000. Также сохраняется символ и цвет, этот процесс уже был описан. В функции kmain создаётся приветственное сообщение и происходит переход процессора в бесконечный цикл с помощью ассемблерной вставки asm("hlt"). На Рисунке 4 представлен успешный результат тестирования минимального ядра на языке С++.

// Эта инструкция обязательно должна быть первой, т.к. этот код компилируется в бинарный,

// и загрузчик передает управление по адресу первой инструкции бинарного образа ядра ОС.

\_\_asm("jmp kmain");

#define VIDEO\_BUF\_PTR (0xb8000)

void out\_str(int color, const char\* ptr, unsigned int strnum) {

unsigned char\* video\_buf = (unsigned char\*) VIDEO\_BUF\_PTR;

video\_buf += 80\*2 \* strnum;

while (\*ptr) {

video\_buf[0] = (unsigned char) \*ptr; // Символ (код) 15

video\_buf[1] = color; // Цвет символа и фона

video\_buf += 2;

ptr++;

}

}

const char\* g\_test = "This is test string.";

extern "C" int kmain() {

const char\* hello = "Welcome to HelloWorldOS (gcc edition)!";

// Вывод строки

out\_str(0x07, hello, 0);

out\_str(0x07, g\_test, 1);

// Бесконечный цикл

while(1) asm("hlt");

return 0;

}

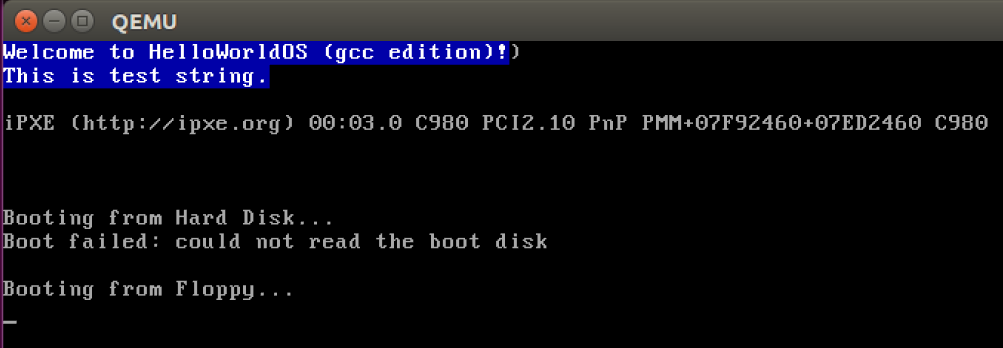


Рисунок 4 ― Тестирования минимального ядра на языке С++

Теперь необходимо добавить ещё некоторые функции, которые помогут в дальнейшей работе ядра, а именно настройка прерываний и исключений процессора. Прерывания и исключения — это события, при которых процессор прекращает выполнение текущей программы и выполняет специальную процедуру — обработчик прерывания. При этом состояние процессора должно быть сохранено, чтобы была возможность продолжить выполнение прерванной программы. Исключения порождаются процессором, когда при выполнении кода происходит ошибка, например, деление на ноль. При возникновении прерывания процессору известен только его номер, который не содержит информации, где находится процедура-обработчик. Эту информацию процессор должен получить из специальной таблицы, которая называется таблицей дескрипторов прерываний IDT (Interrupt Descriptors Table). На ее местонахождение указывает регистр IDTR (interrupt descriptors table register). В таблице находятся не просто адреса обработчиков, а дескрипторы обработчиков.

Для инициализации подсистемы прерываний необходимо вызывать функцию intr\_init. После этого необходимые обработчики (таймер, клавиатура, диск и т.д.) могут быть зарегистрированы с помощью функции intr\_reg\_handler. Регистрация таблицы дескрипторов прерываний осуществляется с помощью функции intr\_start. Включение прерываний осуществляется функцией intr\_enable. Примером пустого обработчика прерываний является функция default\_intr\_handler.

#define IDT\_TYPE\_INTR (0x0E)

#define IDT\_TYPE\_TRAP (0x0F)

// Селектор секции кода, установленный загрузчиком ОС

#define GDT\_CS (0x8)

// 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

struct idt\_entry g\_idt[256]; // Реальная таблица IDT

struct idt\_ptr g\_idtp; // Описатель таблицы для команды lidt

// Пустой обработчик прерываний. Другие обработчики могут быть реализованы по этому шаблону

void default\_intr\_handler() {

asm("pusha");

// ... (реализация обработки)

asm("popa; leave; iret");

}

typedef void (\*intr\_handler)();

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");

}

Теперь необходимо настроить прерывания клавиатуры, с помощью которой и будет производиться вся работа. При нажатии пользователем клавиши клавиатуры контроллер прерываний вызывает прерывание 0x09 процессора. Перечень прерываний, которые ОС обрабатывает, сообщается контроллеру в виде битовой маски. Взаимодействие с оборудованием, в том числе с контроллером прерываний, производится с помощью чтения и записи в порты ввода-вывода. Для этого предназначены следующие служебные функции:

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));

}

Следующий фрагмент программы регистрирует обработчик прерывания клавиатуры и разрешает контроллеру прерываний его вызывать в случае нажатия пользователем клавиши клавиатуры:

#define PIC1\_PORT (0x20)

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

}

Обработчик прерываний может быть реализован следующим образом:

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");

}

Функция keyb\_process\_keys считывает поступивший от пользователя символ. Скан-код клавиши может быть переведен в символ путем применения таблиц, подходящих для используемого оборудования. Например, для многих клавиатур отрицательные скан-коды соответствуют отпусканию клавиши, а положительные — нажатию. Следующий пример демонстрирует считывание скан-кода клавиши для PS/2 клавиатуры:

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) // Скан-коды выше 128 - это отпускание клавиши

on\_key(scan\_code);

}

}

## Реализация индивидуального варианта задания

### Загрузчик

При загрузке ОС появляется латинский алфавит. Пользователю предлагается выбрать, какие буквы будут использованы. Однократное нажатие буквы запрещает её использование, повторное нажатие ― разрешает. Нажатие клавиши Enter должно завершать работу загрузчика и передавать управление ядру. Для этого в функции go\_to\_kernel проверяется нажатие этой клавиши и происходит условный переход на метку save\_buf, где весь алфавит сохраняется в стек с помощью команды push. Если этого не происходит, то произойдёт переход на метку с циклом, в котором происходит переход на метку forbid, запрещающую использование этой буквы. Если буква уже исключена из алфавита, то из forbid будет переход на put\_letter, которая разрешает букву. В конце обеих меток стоит переход на go\_to\_kernel, чтобы проверить, закончил ли пользователь ввод.

mov edi, 0xb8000 ; В регистр edi помещается адрес начала буфера видеопамяти.

mov bx, alphabet ; В регистр bx помещается адрес начала строки

call video\_puts ; Вызывается функция для видеовывода в буфер памяти видеокарты

jmp go\_to\_kernel

save\_buf:

mov bx, alphabet

push bx

go\_to\_kernel:

mov ah, 0x00

int 0x16

cmp al, 0x0d ;if enter

je save\_buf

\_loop:

cmp al, [bx]

je forbid ;if same letter

add bx, 1

add cx, 1

jmp \_loop

forbid:

mov bx, cx

mov ah, [bx]

cmp ah, 0x20 ;space

je put\_letter

mov ah, 0x020

mov [bx], ah

mov bx, alphabet

mov cx, alphabet

call video\_puts

jmp go\_to\_kernel

put\_letter:

mov [bx], al

call clean

call carete

mov bx, alphabet

mov cx, alphabet

call video\_puts

jmp go\_to\_kernel

На Рисунке 5 представлена блок-схема алгоритма загрузчика.

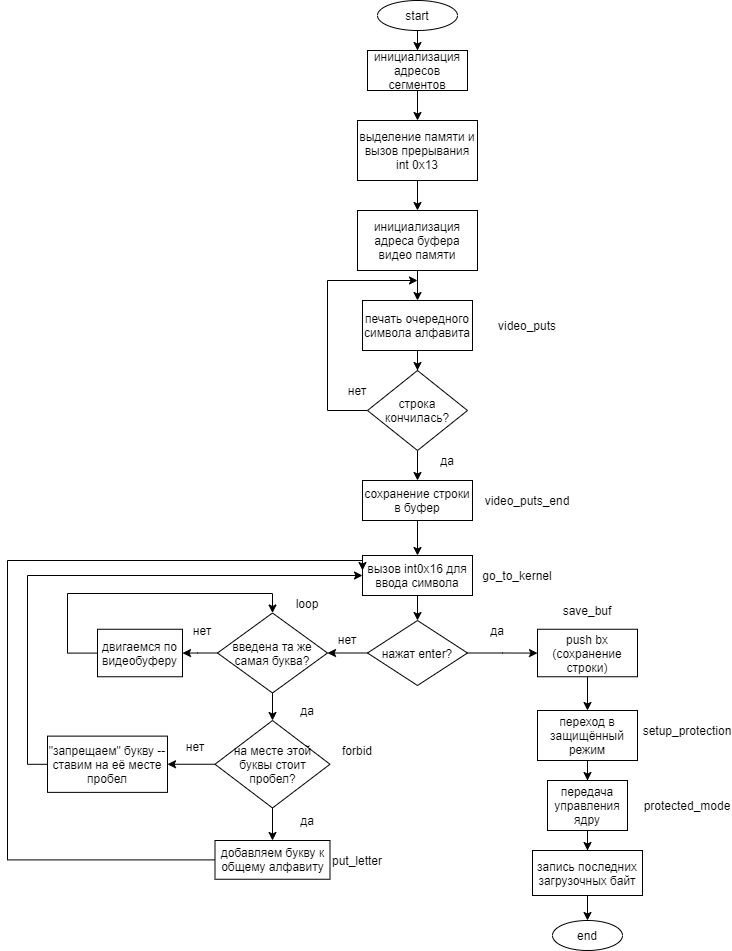


Рисунок 5 ― Блок-схема загрузчика

### Ядро

Для начала создадим глобальные переменные, которые будут хранить алфавит, количество загруженных букв, текущую позицию курсора. Также была создана таблица скан-кодов клавиатуры с учётом специальных клавиш типа backspace, enter, alt и т.д. Кроме того были созданы английский и испанский словари из 52 слов (на каждую букву по два слова).

unsigned char alphabet[26] = { 0 };

unsigned int pos = 0;

unsigned int loaded = 0;

unsigned char syms[] = { 0 /\*no a[0] in ascii\*/, 0, /\*esc\*/

'1', '2', '3', '4', '5', '6', '7', '8', '9', '0', '-', '=', 0, //backspace

0 /\*tab\*/, 'q', 'w', 'e', 'r', 't', 'y', 'u', 'i', 'o', 'p', '[', ']', 0, //enter

0 /\*ctrl\*/, 'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l', ';', '\'', '`', 0, //shift

'\\', 'z', 'x', 'c', 'v', 'b', 'n', 'm', ',', '.', '/', 0, //shift

0 /\*prtsc\*/, 0 /\*alt\*/, ' ' /\*space\*/, 0 /\*caps\*/};

В функции kmain было добавлено извлечение алфавита из загрузчика и подсчёт разрешённых букв.

unsigned char\* video\_buf = (unsigned char\*) VIDEO\_BUF\_PTR; //get alpha from loader

for (unsigned int i = 0; i < 26; i++) {

alphabet[i] = (unsigned char)video\_buf[0];

video\_buf += 2;

}

for (unsigned int i = 0; i < 27; i++) //count loaded letters

if(alphabet[i] != ' ') loaded++;

loaded--;

loaded \*= 2; //two words each letter

Далее пользователь видит приветственное сообщение и справку по командам. Более подробно они будут описаны ниже.

const char\* hello = "Welcome to DictOS (gcc edition)!";

const char\* invite = "Please choose command:";

const char\* inf = "info -- information about creator and development tools";

const char\* di = "dictinfo -- information about language and amount of words";

const char\* trans = "translate [word] -- translation from english";

const char\* ws = "wordstat [letter] -- amount of words on this letter";

const char\* sd = "shutdown -- power off computer";

const char\* kb = "Use Enter to put your arguments, Backspace to delete extra symbols, Alt to finish command and put next arguments.";

// Вывод строки

out\_str(0xDC, hello, 0);

out\_str(0xDC, invite, 1);

out\_str(0xDC, inf, 2);

out\_str(0xDC, di, 3);

out\_str(0xDC, trans, 4);

out\_str(0xDC, ws, 5);

out\_str(0xDC, sd, 6);

out\_str(0xDC, kb, 7);

Курсор передвигается на строку 9, сообщение со справкой остаётся в течение все й работы пользователя. Также были настроены прерывания. После всех этих действий процессор переходит в бесконечный цикл. На Рисунке 6 представлен старт ядра.

cursor\_moveto(9, 0); //user write from here, "hello" msg stay like helper

intr\_disable();

intr\_init(); //set interrupts (for keyboard)

keyb\_init(); //set up keyboard

intr\_start(); //reg desc

intr\_enable(); //start work with intr on

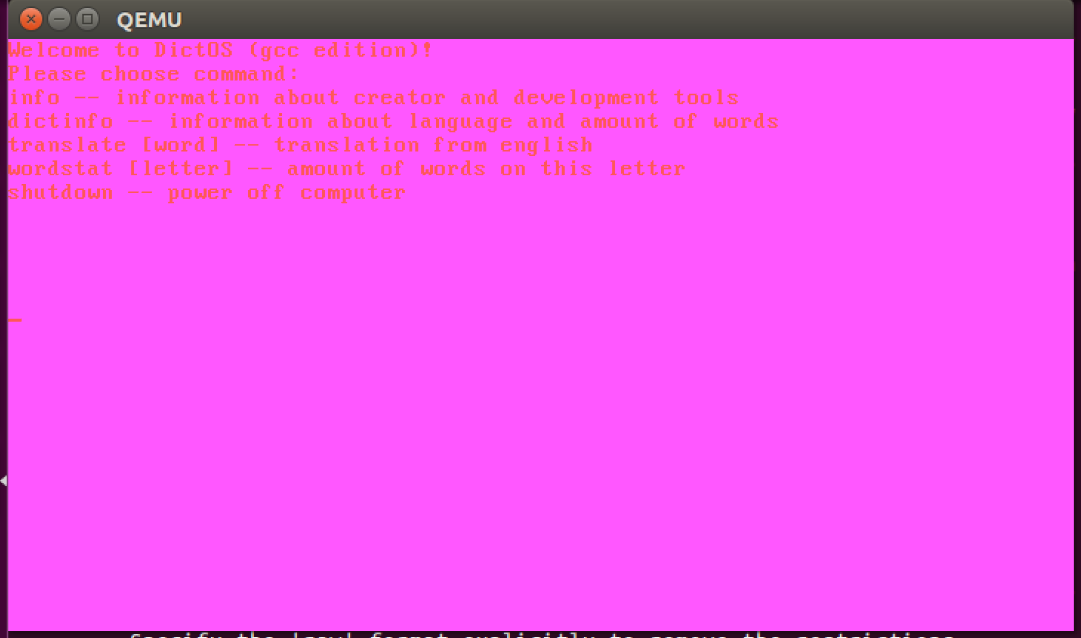


Рисунок 6 ― Стартовое сообщение в ядре

При вводе символов в функции keyb\_process\_keys происходит считывание данных из порта контроллера. Если Скан-код считанного символа меньше 128, то переходим в функцию on\_key. В ней рассмотрены несколько вариантов. Если введён backspace (14), то необходимо заполнить текущую позицию нулем, что будет отображаться как пустой символ, текущая позиция уменьшается на единицу. Если введён enter, то строка сохраняется в буфер и передаётся в обработчик команд. Если введён alt, то весь экран очищается, чтобы можно было ввести следующую команду. Иначе символы просто печатаются в консоль с помощью функции puts.

void on\_key(unsigned char scan\_code) {

switch (scan\_code) {

case 14: { // backspace

unsigned char\* video\_buf = (unsigned char\*) VIDEO\_BUF\_PTR;

video\_buf += 80\*2 \* 9 + 2\*(pos - 1); //move on 2 bytes (color + char)

video\_buf[0] = 0; // Символ (код)

video\_buf[1] = 0xDC; // Цвет символа и фона

//video\_buf -= 2;

pos--;

break;

}

case 28: { //enter

unsigned char\* video\_buf = (unsigned char\*) VIDEO\_BUF\_PTR;

video\_buf += 80\*2 \* 9;

unsigned char\* string;

unsigned char j;

for (unsigned int i = 0, j = 0; i < 40; i+=2, j++)

string[j] = video\_buf[i]; // Символ (код)

//string[j] = '1';

choose\_command(string);

break;

}

case 56: { //alt

clean(9, 15);

pos = 0;

break;

}

default:

puts(0xDC, scan\_code, 9, pos);

pos++;

break;

}

}

В функции puts проверяется, что введено не более 40 символов в строку, иначе они перестанут печаться.

void puts(int color, unsigned char ptr, unsigned int strnum, unsigned int pos) {

if (pos < 40) {

unsigned char\* video\_buf = (unsigned char\*) VIDEO\_BUF\_PTR;

video\_buf += 80\*2 \* strnum + 2\*pos; //move on 2 bytes (color + char)

video\_buf[0] = syms[ptr]; // Символ (код)

video\_buf[1] = color; // Цвет символа и фона

video\_buf += 2;

}

}

В функции choose\_command сравнивается полученный аргумент из консоли с возможными вариантами команд с помощью функции strncmp, которая была реализована самостоятельно. Если такой команды нет, то выводится сообщение об этом.

void choose\_command(unsigned char\* command) {

if (strncmp(command, (unsigned char\*) "info", 5))

info();

else if (strncmp(command, (unsigned char\*) "dictinfo", 9))

dictinfo();

else if (strncmp(command, (unsigned char\*) "shutdown", 9))

shutdown();

else if (strncmp(command, (unsigned char\*) "translate", 10)) {

unsigned char arg[20] = { 0 };

for (unsigned int i = 0; i < 20; i++) {

arg[i] = command[i + 10];

}

translate(arg);

}

else if (strncmp(command, (unsigned char\*) "wordstat", 9)) {

unsigned char c = command[9];

wordstat(c);

}

else {

clean(10, 15);

const char\* empty = "Unknown command";

out\_str(0xDC, empty, 10);

}

}

Функция info выводит информацию о разработчике ОС, технических средствах и возможных для использования буквах. В самом начале необходимо очистить экран от всех предыдущих данных, что выполняется с помощью вспомогательной функции clean. В самом конце курсор возвращается в начало строки. Данные действия происходят в каждой функции, поэтому далее они упоминаться не будут.

void info() {

clean(10, 15);

const char\* author = "Created by Kuleeva Anna, group 4851003/90001, IBKS 2021.";

const char\* os = "OS: Linux";

const char\* tr = "Assemler: YASM, Intel syntax";

const char\* deb = "Debugger: gcc";

const char\* ltr = "You can use these letters:";

out\_str(0xDC, author, 10);

out\_str(0xDC, os, 11);

out\_str(0xDC, tr, 12);

out\_str(0xDC, deb, 13);

out\_str(0xDC, ltr, 14);

out\_str(0xDC, (const char\*) alphabet, 15);

pos = 0; //return carete for next command

}

Функция dictinfo должна выводить информацию о словаре, а именно язык, общее количество слов и количество слов доступных с учётом количества загруженных букв. Если первые два пункта известны разработчику системы, то последний необходимо подсчитать. Изначально количество загруженных букв считается ещё в функции kmain, остаётся перевести число в символы, поскольку печать числовых аргументов не поддерживается. Для этого была использована альтернативная функция itoa, реализованная с помощью оператора switch-case. К сожалению, по непонятным причинам стандартная реализация не увенчалась успехом, поэтому пришлось искать иные пути решения.

void dictinfo() {

clean(10, 15);

const char\* lang = "Language: en -> es";

const char\* all = "MAX words: 52";

unsigned char load[20] = "Loaded words: ";

if (loaded < 10)

load[13] = itoa (loaded);

else {

unsigned int copy = loaded;

copy %= 10; //take first

load[14] = itoa (copy);

copy = loaded;

copy /= 10; //take second

load[13] = itoa (copy);

}

out\_str(0xDC, lang, 10);

out\_str(0xDC, all, 11);

out\_str(0xDC, (const char\*) load, 12);

pos = 0; //return carete for next command

}

unsigned char itoa(unsigned int x) {

switch (x) {

case 0: return '0'; //break;

case 1: return '1'; //break;

case 2: return '2'; //break;

case 3: return '3'; //break;

case 4: return '4'; //break;

case 5: return '5'; //break;

case 6: return '6'; //break;

case 7: return '7'; //break;

case 8: return '8'; //break;

case 9: return '9'; //break;

}

}

Функция wordstat должна выводить количество доступных для перевода слов на заданную букву. Для этого совершается проход по всему массиву с алфавитом. Если какой-то из элементов массива равен поступившему символу, то буква загружена, а значит, на неё есть ровно 2 слова (данное условие было определено разработчиком словаря). Если цикл завершился по условию окончания массива, то такая буква не была найдена, то есть на её месте стоит пробел. Тогда выводится сообщение, что для перевода доступно 0 слов.

void wordstat(unsigned char c) {

clean(10, 15);

unsigned int i = 0;

for (i = 0; i < 27; i++)

if (alphabet[i] == c) {

const char\* yes = "There are 2 words on this letter";

out\_str(0xDC, yes, 10);

break;

}

if (i == 27) {

const char\* no = "There are 0 words on this letter";

out\_str(0xDC, no, 10);

}

pos = 0; //return carete for next command

}

Функция translate переводит введённое пользователем слово с английского на испанский язык. Для начала с помощью самостоятельно реализованной функции strlen проверяется, что слово вообще введено, то есть строка не равна нулю. Затем по аналогии с функцией wordstat реализуется проверка, что слова на такую букву вообще загружены. Если в алфавите найдена первая буква слова, то работаем дальше. Если весь алфавит просмотрен и буквы нет, то печатается сообщение об ошибке и происходит возврат из функции. Когда все проверки пройдены, необходимо найти слово. i-ой букве алфавита соответствует (2\*i)-ое и (2\*i+1)-ое слова. Соответственно сравниваем оба этих элемента массива с помощью strncmp. Также необходимо убедиться, что аргументы имеют одинаковую длину, поскольку при тестировании были обнаружены некоторые ошибки при переводе. Например, в словаре есть слово “i”. При вводе аргумента “ibks” будет предоставлен перевод слова “i”, что является ошибкой. Если все эти условия соблюдены, то выводится слово из испанского словаря под соответствующим индексом. Иначе выводится сообщение, что такого слова в словаре нет.

void translate(unsigned char \*word) {

clean(10, 15);

unsigned int len = strlen((const char\*) word);

if (len == -1) {

pos = 0; //return carete for next command

return; //check arg

}

unsigned int i = 0;

for (i = 0; i < 27; i++)

if (alphabet[i] == word[0])

break;

if (i == 27) {

const char\* no\_let = "This letter wasn't loaded.";

out\_str(0xDC, no\_let, 10);

pos = 0; //return carete for next command

return;

} else { //letter was found

i \*= 2;

if (strncmp(word, (unsigned char\*) english[i], len) && len == strlen(english[i]))

out\_str(0xDC, spanish[i], 10);

else {

if (strncmp(word, (unsigned char\*) english[i + 1], len) && len == strlen(english[i + 1]))

out\_str(0xDC, spanish[i + 1], 10);

else {

const char\* no\_word = "There are no such word in dictionary :(";

}

}

}

pos = 0; //return carete for next command

}

Последняя функция shutdown выключает виртуальный компьютер. Для этого в порт контроллера должны быть пересланы определённые байты, что делается с помощью функции outw. Она реализуется по аналогии с outb.

void shutdown() {

clean(10, 15);

// This is the fix for Qemu support.

// Dmitry V. Reshetov, IKBS, SPbSTU, 2016

//outw (0xB004, 0x2000); // qemu < 1.7, ex. 1.6.2

unsigned int data = 0x2000;

outw (0x604, data); // qemu >= 1.7

}

static inline void outw (unsigned short port, unsigned int data) { // Запись

asm volatile ("outw %w0, %w1" : : "a" (data), "Nd" (port));

}

В Таблице 1 представлен список всех реализованных функций, их аргументов и описание действий.

Таблица 1 ― Описание реализованных функций.

| Название функции | Аргументы | Описание действий |
| --- | --- | --- |
| unsigned int strncmp | (unsigned char \*a, unsigned char \*b, unsigned int n) | Сравнение двух строк на эквивалентность символов |
| unsigned int strlen | (const char \*string) | Возврат длины строки |
| unsigned char itoa | (unsigned int num) | Перевод числа int в символ char |
| void choose\_command | (unsigned char\* command) | Обработчик команд, поступающих о пользователя |
| void clean | (unsigned int begin, unsigned int end) | Очистка экрана от строки begin до строки end |
| void info | (void) | Выводит информацию об авторе и средствах разработки ОС, заданные в загрузчике параметры |
| void dictinfo | (void) | Выводит информацию о загруженном словаре язык, общее кол-во слов, количество доступных для перевода слова |
| void wordstat | (unsigned char c) | Выводит количество загруженных слов в словаре на указанную букву |
| void translate | (unsigned char \*word) | Переводит слово с английского языка на испанский. Если слово не найдено или не загружено — выводит ошибку. |
| void shutdown | (void) | Выключение компьютера |
| void puts | (int color, unsigned char ptr, unsigned int strnum, unsigned int pos) | Посимвольный вывод в консоль |

## Тестирование

Было проведено тестирование программы на работоспособность различных аспектов. Всего было проведено 17 тестов.

Начнём с тестирования основных функций ядра, несвязанных с индивидуальным вариантом. При попытке ввести более 40 символов в строку символы перестают печататься в консоль (см. Рисунок 7).



Рисунок 7 ― Тестирование ограничения по печати символов

При нажатии клавиши alt происходит очищение экрана и перед пользователем оказывается экран, как при старте ядра (см. Рисунок 6).

Протестируем клавишу backspace. На Рисунках 8 и 9 представлен результат до и после нажатия клавиши.



Рисунок 8 ― Ввод символов



Рисунок 9 ― Нажатие клавиши backspace

При вводе неизвестной команды появляется сообщение об ошибке (см. Рисунок 10).

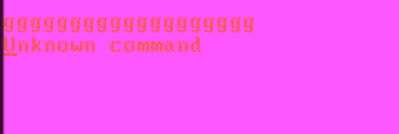


Рисунок 10 ― Ввод неизвестной команды

Теперь перейдём к тестированию

функций, связанных с вариантом задания. Рассмотрим функцию info() в трёх случаях: когда не было загружено ни одной буквы, когда были загружены все буквы, когда было загружено несколько букв. Результаты представлены на Рисунках 11, 12 и 13 соответственно.

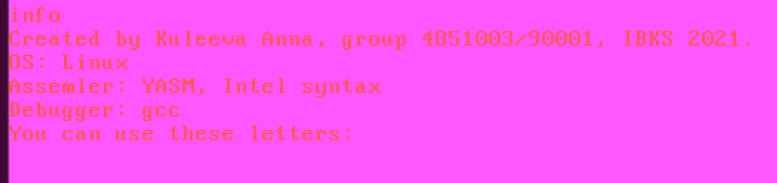


Рисунок 11 ― Вызов info(), когда не загружено ни одной буквы

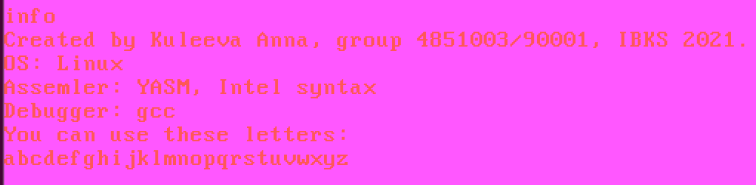


Рисунок 12 ― Вызов info(), когда загружены все буквы

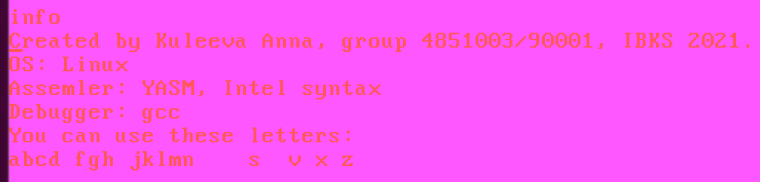


Рисунок 13 ― Вызов info(), когда загружено несколько букв

Аналогичным образом протестируем dictinfo() (см. Рисунки 14, 15, 16).

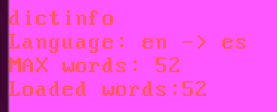


Рисунок 14 ― Вызов dictinfo(), когда загружены все буквы

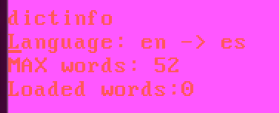


Рисунок 15 ― Вызов dictinfo(), когда не загружено ни одной буквы

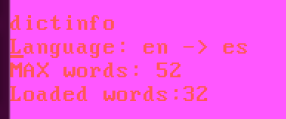


Рисунок 16 ― Вызов dictinfo(), когда загружено несколько букв

Далее протестируем функцию wordstat. Есть два варианта: буква либо загружена и есть 2 слова, либо не загружена и нет слов. Рассмотрим Рисунки 17, 18.

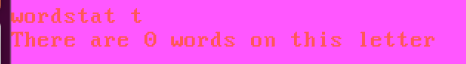


Рисунок 17 ― Вызов wordstat, когда буква не загружена



Рисунок 18 ― Вызов wordstat, когда буква загружена

Для функции translate были введены следующие данные в загрузчике (см. Рисунок 19). На Рисунках 20-23 представлены различные тесты для таких данных.

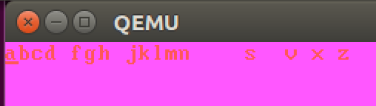


Рисунок 19 ― Данные в загрузчике для теста

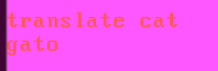


Рисунок 20 ― Успешный перевод

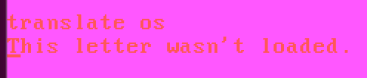


Рисунок 21 ― Перевод на незагруженную буквы

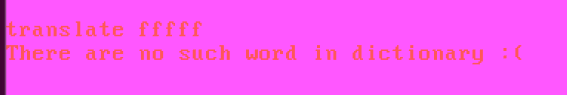


Рисунок 22 ― Перевод несуществующего слова



Рисунок 23 ― Перевод слова, которое есть в словаре лишь частично

При тестировании функции shutdown виртуальный компьютер выключился и окно эмулятора qemu закрылось.

# Вывод

В ходе выполнения данной лабораторной работы были изучены основы разработки операционных систем, а именно написан код на языке Ассемблер для загрузчика системы и код на языке С++ для ядра ОС. Были изучены такие особенности загрузчика, как выделение памяти, переход в защищённый режим работы процессора, в частности познакомились с адресной линией А20. Также были приобретены навыки по написанию программ на языке ассемблера в синтаксисе Intel для транслятора yasm.

В процессе реализации ядра операционной системы были более подробно изучены механизмы прерываний, в частности необходимо было поработать с контроллером прерываний.