



Министерство науки и высшего образования Российской Федерации
МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ имени Н.Э. БАУМАНА
ФАКУЛЬТЕТ ИНФОРМАТИКИ И СИСТЕМ УПРАВЛЕНИЯ
КАФЕДРА «ТЕОРЕТИЧЕСКАЯ ИНФОРМАТИКА И КОМПЬЮТЕРНЫЕ
ТЕХНОЛОГИИ»

Расчетно-пояснительная записка к курсовой работе на тему
«Библиотека для написания программ, запускаемых при помощи GRUB»
по дисциплине *«Операционные системы»*

Студент ИУ9-526 _____ А.Б. Барлука
(Группа) (Подпись, дата) (И.О.Фамилия)

Руководитель курсовой работы _____ А.В. Коновалов
(Подпись, дата) (И.О.Фамилия)

Москва 2018

СОДЕРЖАНИЕ

Аннотация	3
Введение	4
1 Запуск программы из GRUB	5
2 Сборка программы	7
3 Экранный ввод-вывод	8
4 Работа с памятью	11
5 Другие функции	14
6 Тестовое приложение: Тетрис	16
6.1 Организация игры	16
6.2 Игровой цикл	17
Заключение	19
Список литературы	20

АННОТАЦИЯ

Расчетно-пояснительная записка к проекту “Библиотека для написания программ, запускаемых при помощи GRUB” содержит 20 страниц машинописного текста, 16 рисунков.

В расчетно-пояснительной записке приведены подробное описание реализации библиотеки ввода-вывода с иллюстрациями сигнатур функций и исходного кода, использованные алгоритмы выделения и освобождения памяти, принципы программного взаимодействия с периферийными устройствами, архитектура демонстрационного приложения, скриншоты его работы.

ВВЕДЕНИЕ

Сегодня сложно вообразить компьютер без операционной системы. Абсолютное большинство существующих десктопных программ написаны под операционные системы Windows, Linux, MacOS. Даже в основе кроссплатформенного ПО лежит использование библиотек, ранее скомпилированных под определенные ОС. Для большинства нужд разработки достаточно использовать готовые средства и библиотеки. Однако интересно разобраться в принципах написания низкоуровневых программ. Хотя и зачастую таковыми являются операционные системы, тем не менее и по сей день программное обеспечение, не реализующее полноценный функционал ОС, находит свое применение. Наиболее распространенной утилитой, запускаемой из GRUB, является Memtest86 – программа для тестирования оперативной памяти, часто поставляемая в UNIX-подобных системах вместе с загрузчиком.

Цель данной работы – изучить процесс запуска программ из загрузчика, реализовать фреймворк для запуска кода на языке C, поддерживающий экранный ввод/вывод и работу с памятью, из загрузчика GRUB, а также написать приложение, демонстрирующее возможности реализованных библиотек.

Разработка велась на GNU Assembler и ANSI C с использованием ассемблерных вставок. Утилита Make позволила автоматизировать сборку и задание параметров компилятора gcc. Также были использованы различные утилиты Linux для манипуляций с файлами, в том числе grub-install для установки загрузчика в образ.

1 Запуск программы из GRUB

Для начала определим требования к программам, предъявляемые целевым загрузчиком.

GRUB – наиболее распространенный загрузчик операционных систем семейства Unix-подобных. Также он способен передавать управление загрузчикам Windows. На данный момент существуют две версии: grub-legacy и grub2. Он полноценно поддерживает спецификацию Multiboot.

Multiboot создана для унификации интерфейса между ядром операционной системы и загрузчиком, следуя которой любой загрузчик, поддерживающий данную спецификацию, может применяться для загрузки любой совместимой операционной системы. На данный момент существуют две версии: Multiboot и Multiboot 2. В последней были исправлены многие ошибки и недочеты, присущие первой версии спецификации, поэтому обратная совместимость с предыдущей версией отсутствует.

Образ ОС должен содержать «Multiboot header» - специальный заголовок, предназначенный для взаимодействия с загрузчиком. Этот заголовок должен полностью располагаться в первых 8192 байтах образа ОС и должен быть выровнен по длине 32-битного слова. Структура заголовка приведена на Рисунке 1 и имеет следующие поля:

- magic – магическое число, должно быть равно 0x1BADB002;
- flags – требования для корректной работы программы/ОС;
- checksum – контрольная сумма предыдущих двух полей;
- header_addr – физический адрес начала заголовка Multiboot;
- load_addr – физический адрес начала сегмента данных;

- load_end_addr – физический адрес конца сегмента данных;
- bss_end_addr – физический адрес конца сегмента bss;
- entry_addr – физический адрес входа в программу/ядро ОС;
- width – число колонок в видеорежиме (число пикселей по горизонтали в графическом режиме или количество символов в строке в текстовом режиме);
- height – число строк в видеорежиме (число пикселей по вертикали в графическом режиме или количество строк в текстовом режиме);
- depth – значение числа бит на пиксель в графическом режиме (0 в текстовом режиме).

Offset	Type	Field Name	Note
0	u32	magic	required
4	u32	flags	required
8	u32	checksum	required
12	u32	header_addr	if flags[16] is set
16	u32	load_addr	if flags[16] is set
20	u32	load_end_addr	if flags[16] is set
24	u32	bss_end_addr	if flags[16] is set
28	u32	entry_addr	if flags[16] is set
32	u32	mode_type	if flags[2] is set
36	u32	width	if flags[2] is set
40	u32	height	if flags[2] is set
44	u32	depth	if flags[2] is set

Рисунок 1 – Структура Multiboot header

Была поставлена цель реализовать высокоуровневый интерфейс, позволяющий писать код на языке С без необходимости использования низкоуровневых команд. Удобно было сделать функцию main в качестве точки входа в программу. Теперь поясним, каким образом код, содержащийся в main, компилируется в полноценную программу.

2 Сборка программы

Компиляция с использованием написанного Make-файла может производиться из любого дистрибутива Linux, основанного на Debian, например, Ubuntu 18.04 LTS. Далее приведем подробный алгоритм сборки программы.

В качестве носителя создается образ диска `disk.img` (утилита `dd`). Далее размечается таблица разделов (утилита `fdisk`), и создается файловая система `ext2` (утилита `mke2fs`). С помощью `grub-install` в образ устанавливается загрузчик.

Исходные файлы с кодом на C компилируются при помощи `gcc`, а файлы с инструкциями ассемблера (в том числе файл с загрузочным кодом, который вызывает функцию `main` из C) - с помощью `as` (ассемблер проекта GNU). Затем полученные объектные файлы компоуются с помощью линковщика `ld` в двоичный файл ядра `kernel.bin`, который помещается в файловую систему `ext2` образа `disk.img`.

На выходе алгоритма имеем образ `disk.img`, который можно записать на жесткий диск, флеш-накопитель, запустить в эмуляторе и т.п.

Приложения, собранные с помощью реализованного фреймворка, поддерживают первую версию спецификации мультизагрузки. Когда загрузчик передает управление программе, ей доступны поля заголовка Multiboot, как показано на Рисунке 2.

```
void main(multiboot_info_t* mbd, unsigned int magic) {
```

Рисунок 2 – Сигнатура функции `main`. Аргумент “`mbd`” содержит указатель на структуру Multiboot header, а “`magic`” – магическое число Multiboot

На данный момент процесс сборки приложений из исходного кода полностью автоматизирован. Теперь будем расширять возможности фреймворка, добавив возможность взаимодействия с клавиатурой и экраном монитора.

3 Экранный ввод-вывод

По умолчанию GRUB использует текстовый режим 80x25. Это означает, что для отображения на экране доступно 25 строк, по 80 символов в каждой. По физическому адресу 0xb8000 располагается буфер видеопамати VGA. Каждый символ занимает в памяти 2 байта (0-7 биты содержат ASCII-значение, 8-11 биты содержат цвет символа, 12-15 биты содержат цвет фона, 16 бит показывает, является ли символ мигающим). Всего доступно 16 цветов (Рисунок 3).

Код	Пример
0x0	black
0x1	blue
0x2	green
0x3	cyan
0x4	red
0x5	magenta
0x6	brown
0x7	gray
0x8	dark gray
0x9	bright blue
0xA	bright green
0xB	bright cyan
0xC	bright red
0xD	bright magenta
0xE	yellow
0xF	white

Рисунок 3 – Палитра цветов VGA

Для реализации функций ввода/вывода необходимо научиться работать с текущей позицией курсора.

Настраивать курсор можно через чтение+запись в порты VGA 0x3D4 и 0x3D5, соответствующие регистру адреса и регистру данных. Для этого с помощью ассемблерных вставок в C и инструкций GAS outb, inb были реализованы одноименные функции (исходный код приведен на Рисунке 4).

```
unsigned char inb(unsigned short int port) {
    unsigned char res;
    __asm__ volatile (
        "inb %1, %0"
        : "=a"(res)
        : "Nd"(port)
    );
    return res;
}

void outb(unsigned char value, unsigned short int port) {
    __asm__ volatile (
        "outb %0, %1"
        :
        : "a"(value), "Nd"(port)
    );
}
```

Рисунок 4 – Реализация функций inb, outb с использованием ассемблерных вставок

Далее были написаны функции disable_cursor, enable_cursor, move_cursor, update_cursor – отключение, включение, изменение позиции, обновление позиции по координатам курсора соответственно (Рисунок 5). Для задания размера и позиции курсора используются следующие регистры: начала курсора 0x0A, конца курсора 0x0B, старшая и младшая части позиции курсора 0x0E и 0x0F.

```
void disable_cursor();
void enable_cursor(unsigned short int cursor_start, unsigned short int cursor_end);
void move_cursor(int x, int y);
void update_cursor();
```

Рисунок 5 – Объявления функций работы с курсором

Теперь можно приступить непосредственно к реализации ввода-вывода на экран. При разработке функций их именование было ориентировано преимущественно на стандартную библиотеку C с максимально возможным сохранением оригинальных сигнатур.

В функции `putchar` помимо ASCII-символов реализована поддержка основных управляющих символов (“\n”, “\t”, “\r”, “\b”). Учитываются границы экрана: при необходимости автоматически происходит перенос строки. Реализация `puts`, использующая `putchar`, тривиальна. Исходный код функции `printf` был взят из проекта BitVisor[8], так как реализация всех параметров форматирования довольно трудоемка и выходит далеко за рамки данной работы, однако форматированный вывод очень удобен при отладке и написании программ. С учетом способа вывода на экран удобно реализовать очистку экрана функцией `clear_screen` как заполнение всего видеобуфера пробельными символами. На Рисунке 6 представлены сигнатуры функций вывода на экран.

```
void clear_screen();  
void putchar(int c);  
int puts(const char* s);  
int printf (const char *format, ...);
```

Рисунок 6 – Объявления функций вывода на экран

Для написания `getchar` необходимо научиться работать с клавиатурой. Был выбран способ взаимодействия через порты 0x60 и 0x64 (аналогично курсору). Процесс обработки нажатия одной клавиши в `getchar` можно описать следующим образом. С командного порта 0x64 считывается байт статуса, по которому можно определить наличие события, тип события (нажатие/отпускание), а также тип контроллера (мышь/клавиатура). Если событие готово для обработки, с порта данных 0x60 считывается байт, содержащий код нажатой клавиши, который затем транслируется в соответствующий управляющий (например, Tab) символ или

ASCII-символ. Добавлена поддержка английской qwerty-раскладки с учетом клавиш Shift и Caps Lock. Функция gets использует вызовы getchar, пока не встретит символ конца строки “\n” (Рисунок 7).

```
int getchar();  
char* gets(char* s);
```

Рисунок 7 – Объявления функций ввода

На данном этапе имеется достаточный набор инструментов, позволяющий гибко взаимодействовать с клавиатурой и экраном. Перейдем к следующей цели – организации работы с памятью.

4 Работа с памятью

При сборке программы в загрузочном коде размер стека устанавливается в 16 Кбайт параметром STACKSIZE. Если требуется использовать большой размер памяти фиксированного размера, значение STACKSIZE можно увеличить. Однако если потребуется использовать заранее неизвестный объем памяти, данный способ не придет на помощь. Поэтому удобно реализовать привычные любому программисту на С функции malloc/free для работы с динамической памятью.

Главной проблемой, с которой сталкивается менеджер памяти любой ОС при работе с ОЗУ, является “Memory Holes” – наличие “дыр” в памяти. Адресное пространство содержит несколько участков “memory mapped devices” - отображенных в память устройств. Большинство существующих менеджеров памяти решают эту проблему с помощью страничной организации памяти.

Структура заголовка Multiboot, инициализируемая загрузчиком, позволяет определить начальный и конечный адреса наибольшего доступного сегмента в дополнительной памяти. Эта область памяти (исключая первые 0x100000 байт, зарезервированные для программы) становится доступна для выделения и освобождения. Данный подход прост в реализации, однако имеет недостаток: первые 16 Мбайт памяти, содержащие «дыры», становятся недоступны для использования. Впрочем, учитывая современные объемы оперативной памяти и ничтожно малое отношение используемой памяти к неиспользуемой, данный критерий не является критичным.

Для работы с памятью были написаны 3 основные функции – `mem_init`, `malloc` и `free`. `mem_init` должна быть вызвана ровно один раз перед использованием `malloc/free`. Выделенные с помощью `malloc` участки памяти не претерпевают никаких изменений. Алгоритм строится на специальной структуре свободной памяти.

Свободная память делится на блоки, которые организованы в двусвязный список. Каждый блок является узлом списка и состоит из 4 элементов:

- 1) символьная строка “FREE”, занимающая 4 байта (для удобства отладки);
- 2) адрес предыдущего блока;
- 3) адрес следующего блока;
- 4) размер блока.

Всегда доступен указатель на первый блок. В начале работы ядра такой блок всего один – совпадающий со всей доступной свободной памятью. Размер каждого нового блока всегда ненулевой и кратен 16 байт. Это сделано для того, чтобы избежать потерь памяти при фрагментации.

```
void* malloc(size_t size) {
```

Рисунок 8 – Сигнатура функции malloc

Сигнатура функции malloc изображена на Рисунке 8. Размер запрашиваемой памяти, передаваемый в качестве параметра “size”, округляется вверх до ближайшего числа, кратного 16. В зависимости от состояния пространства свободной памяти используется один из двух способа нахождения требуемого объема:

- 1) по указателю на начало последнего свободного блока;
- 2) из списка свободных блоков.

Первый способ применяется, пока не будет израсходована вся доступная память. Это позволяет обеспечить минимальную алгоритмическую сложность $O(1)$. Последний свободный блок разделяется на два: один вычисленного на основе параметра “size” размера, который будет возвращен из функции, и второй – оставшаяся часть.

Второй способ используется, когда длина последнего свободного блока меньше 16 байт. Тогда происходит линейный поиск по двусвязному списку. В худшем случае это займет $O(N)$ времени, где N – количество свободных блоков. Поэтому выделение большого числа блоков малого размера неэффективно.

```
void free(void* ptr, unsigned int size) {
```

Рисунок 9 – Сигнатура функции free

На Рисунке 9 представлена сигнатура функции free. С учетом особенностей алгоритма управления памятью, пришлось изменить оригинальную сигнатуру и добавить параметр “размер”, как и у malloc. Это незначительно усложняет написание программ, заставляя заботиться о размере выделенной памяти, зато

позволяет не хранить размер блока выделенной памяти на куче перед самим блоком. Так же, как и в malloc, размер “size” увеличивается до числа, кратного 16. Освобожденный блок помечается знаком “FREE” и вставляется в список так, чтобы последний всегда оставался отсортированным по возрастанию адресов элементов. Далее вызывается алгоритм объединения смежных блоков. Среднее время работы free для списка из N блоков составляет $O(N)$.

Теперь дополним библиотеку некоторыми другими инструментами, не менее важными для написания интерактивных пользовательских программ.

5 Другие функции

Для организации цикла взаимодействия с пользователем не обойтись без функций задержки. Функция delay использует Programmable Interval Timer (PIT) – программируемый интервальный таймер. При вызове delay с беззнаковым целочисленным параметром (Рисунок 10), означающим количество тактов, через запись и чтение в порт данных 0x40 и только запись в командный порт 0x43 в PIT отправляется количество тактов, которое процессор должен ожидать. Далее в цикле на каждой итерации читается оставшееся количество тактов, и когда оно приближается к 0, происходит возврат из функции. Поскольку PIT работает с частотой приблизительно 1.193182 МГц, называемой частотой синхронизации, 1 секунда ожидания равняется около 1193182 тактам таймера.

```
void delay(unsigned int x) {
```

Рисунок 10 – Сигнатура функции delay

В delay каждые 20000 тактов таймера вызывается обработка новых нажатий клавиатуры. Был написан кольцевой буфер, запоминающий 1024 события клавиатуры – этого вполне достаточно для интервала времени равного 20000 тактов PIT.

Функция sleeps аналогична delay, только в качестве единственного аргумента принимает число реальных секунд, а не тактов таймера (Рисунок 11):

```
void sleeps(unsigned int seconds) {
```

Рисунок 11 – Сигнатура функции sleeps

Иногда возникает потребность в генерации случайных значений. Для этого были реализованы функции rand, srand, rtc_seed.

Вызов rand() возвращает псевдослучайное значение от 0 до 32768, зависящее от значения переменной next (Рисунок 12). srand(seed) устанавливает значение next в seed («семя»). rtc_seed() устанавливает next в значение «секунда * час * день». Текущее время берется из RTC (Real-Time Clock) – часов реального времени. Секундам соответствует смещение адреса 0x0, минутам – 0x4, часам – 0x7. Смещение записывается в порт адреса 0x70, а затем считывается байт с порта данных 0x71, содержащий соответствующее значение времени. Такой подход позволяет обеспечить достаточный процент случайности набора последовательно генерируемых значений из одного «семени».

```
int rand() {  
    next = next * 1103515245 + 12345;  
    return (unsigned int)(next / 65536) % 32768;  
}
```

Рисунок 12 – Исходный код функции rand

6 Тестовое приложение: Тетрис

Для демонстрации ресурсов библиотеки было написано приложение «Тетрис». Выбор пал на это игровое приложения из-за соразмерной объему курсовой работы сложности, а также возможности задействовать большинство реализованных функций. Скриншот программы, запущенной в эмуляторе QEMU, представлен на Рисунке 13.



Рисунок 13 – скриншот приложения «Тетрис»

6.1 Организация игры

Основная логика программы проста: инициализируются буфер клавиатуры, менеджер памяти. В бесконечном цикле инициализируется «объект» игры, запускается игровой цикл, после завершения освобождаются ресурсы и начинается новая итерация.

В качестве игрового поля используется двумерный динамический массив типа `char` размером 10x20 (как в классической версии игры). Для обновления поля на экране используется функция `video_update`, которая записывает массив игрового поля в видеобуфер. Символы “@” обозначают клетки на поле, которые заняты уже лежащими фигурами, а падающая фигура состоит из символов “#”. Незанятые клетки отображаются как пробелы (“ ”), и вокруг поля рисуется рамка из “|”.

В классическом тетрисе имеется 7 типов фигур (которые напоминают латинские буквы “I”, “J”, “L”, “O”, “S”, “T”, “Z”); при всевозможных вращениях образуются множество из 19 поворотов. Интересно заметить, что каждый поворот фигуры состоит из 4 блоков. Поэтому удобно представлять фигуру как 4 клетки на квадратном поле размера 3x3 с максимальным смещением в верхний левый угол. Для падающей фигуры необходимо хранить: ее вид, текущую координату левого верхнего угла заключающего фигуру квадрата, следующую координату, куда пытается переместиться фигура, а также номер следующей фигуры, вычисляемый с помощью функции `rand`. Отображение следующей фигуры можно увидеть на Рисунке 14.

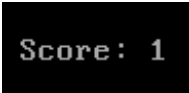


Рисунок 14 – Отображение следующей фигуры

6.2 Игровой цикл

Каждую итерацию игрового цикла (примерно 0,2 секунды) происходит обработка буфера клавиатуры и обновление экрана. Падение блока, применяются нажатые клавиши раз в 1 секунду. Игрок проигрывает, если верхняя строка

игрового поля содержит символ хотя бы один “@”. При заполнении любой строки полностью последняя удаляется, а все строки выше перемещаются вниз на один уровень. При это обновляется счет, как показано на Рисунке 15.




Score: 1

Рисунок 15 – Отображение счета

Стрелками «влево» и «вправо» клавиатуры сдвигается падающая фигура влево и вправо соответственно. «Вниз» означает ускорить падение, а «вверх» - немедленно опустить фигуру.

Нажимая на Enter, фигуру можно поворачивать по часовой стрелке, если это возможно. При нажатии клавиши Esc поле «замораживается», и отображается пауза (Рисунок 16).



paused... press ESC to return to game...

Рисунок 16 – отображение паузы

Исходный код программы занимает 1666 строк кода (включая комментарии).

ЗАКЛЮЧЕНИЕ

Таким образом были достигнуты поставленные в начале работы цели: изучены требования загрузчика к запускаемым программам, реализованы автоматизированная система для сборки приложений и библиотека, содержащая функции для работы с основными устройствами ПК: клавиатурой, экраном монитора, оперативной памятью. Изучена обширная теоретическая база по низкоуровневому программированию и операционным системам. Написано приложение, демонстрирующее реализованные функции.

СПИСОК ЛИТЕРАТУРЫ

1. Таненбаум Э. Архитектура компьютера. 5-е изд. СПб.: Питер, 2007. 844 с.: ил. ISBN 5-469-01274-3.
2. Аблязов Р.З. Программирование на ассемблере на платформе x86-64. М.: ДМ К Пресс, 2011. 304 с.: ил. ISBN 978-5-94074-676-8.
3. GNU GRUB Manual 2.02: [Электронный ресурс]. URL: <http://www.gnu.org/software/grub/manual/grub/grub.html>. (Дата обращения: 16.12.2018).
4. Multiboot Specification version 0.6.96: [Электронный ресурс]. URL: <https://www.gnu.org/software/grub/manual/multiboot/multiboot.html>. (Дата обращения: 16.12.2018).
5. Wilton, Richard. IBM Video Hardware and Firmware // Programmer's Guide to PC and Ps/2 Video Systems.: Microsoft Press, 1987. 544 с.: ISBN 1-55615-103-9.
6. IBM's Keyboard and Auxiliary Device (mouse) Controller documentation: [Электронный ресурс]. URL: http://www.mcarnafia.de/pdf/ibm_hitrc07.pdf. (Дата обращения: 16.12.2018).
7. GNU Make Manual: [Электронный ресурс]. URL: <https://www.gnu.org/software/make/manual/>. (Дата обращения: 16.12.2018).
8. BitVisor: [Электронный ресурс]. URL: <https://www.bitvisor.org/bitvisor/HTML/>. (Дата обращения: 16.12.2018).