Документация Mini OS

# Оглавление

[1. Оглавление 3](#_Toc442035869)

[2. Определения 3](#_Toc442035870)

[3. Рабочая среда 4](#_Toc442035871)

[3.1 Используемые компоненты 4](#_Toc442035872)

[3.2 Подготовка рабочего окружения 4](#_Toc442035873)

[3.3 Команды окружения 4](#_Toc442035874)

[3.3.1 Утилита автоматизации сборки Make 4](#_Toc442035875)

[3.3.2 Вспомогательные bash скрипты 4](#_Toc442035876)

[3.4 Основные инструменты 5](#_Toc442035877)

[3.4.1 Виртуальная машина Qemu 5](#_Toc442035878)

[3.4.1 Отладчик GDB 6](#_Toc442035879)

[3.5 Исследование готовых файлов 6](#_Toc442035880)

[4. Файлы проекта 7](#_Toc442035881)

[4.1 Список файлов 7](#_Toc442035882)

[4.1.1 Папка grub/ 7](#_Toc442035883)

[4.1.2 Папка src/ 7](#_Toc442035884)

[4.2 Файлы для сборки 8](#_Toc442035885)

[4.2.1 Скрипт компоновщика 8](#_Toc442035886)

[4.2.2 Make файлы автоматизации сборки 10](#_Toc442035887)

[4.2.3 Компиляция исходных кодов ядра 10](#_Toc442035888)

[4.2.4 Сборка загрузочного образа (hdd.img) 11](#_Toc442035889)

[4.3 Исходные коды ядра ОС 13](#_Toc442035890)

[4.3.1 boot.s 13](#_Toc442035891)

[4.3.2 main.c 13](#_Toc442035892)

[5. Проектирование исходного кода ядра ОС 14](#_Toc442035893)

[5.1 Таблицы IDT и GDT 14](#_Toc442035894)

[5.1.1 Инициализация таблицы дескрипторов сегментов GDT 16](#_Toc442035895)

[5.1.2 Инициализация таблицы прерываний IDT 18](#_Toc442035896)

[5.1.3 Обработка прерывания 22](#_Toc442035897)

[5.1.4 Установка своих обработчиков прерываний 27](#_Toc442035898)

[6. Диаграммы 28](#_Toc442035899)

[7. Результаты исследований 38](#_Toc442035900)

[7.1 Исполняемый файл ОС 38](#_Toc442035901)

[8. Приложения 41](#_Toc442035902)

[8.1 Архитектура компьютера 41](#_Toc442035903)

[8.2 Режимы работы процессора 41](#_Toc442035904)

[8.2.1 Реальный режим (Real Mode) 41](#_Toc442035905)

[8.2.2 Защищенный режим (Protected Mode) 41](#_Toc442035906)

[8.3 Ассемблер 41](#_Toc442035907)

[8.3.1 Команды работы с флагами 42](#_Toc442035908)

[8.3.2 Дерективы Nasm 42](#_Toc442035909)

[8.4 Взаимодействие кода на ассемблере и кода на Си 43](#_Toc442035910)

[8.5 Ассемблерные вставки в код на Си 43](#_Toc442035911)

[8.5.1 Особенности 43](#_Toc442035912)

[8.5.2 Общая структура вставки 43](#_Toc442035913)

[8.5.3 Синтаксис команд 44](#_Toc442035914)

[8.5.4 Ключевое слово volatile 44](#_Toc442035915)

[8.5.5 Пример вставки 44](#_Toc442035916)

[8.6 Процесс работы загрузчика 44](#_Toc442035917)

[8.7 Карта расположения загрузчика в памяти 45](#_Toc442035918)

[8.8 Состояние машины при старте ОС 46](#_Toc442035919)

[9. Ссылки 48](#_Toc442035920)

# Определения

**Загрузочный образ** (Boot image) - тип образа диска (файл, содержащий в себе полную копию содержания и структуры файловой системы и данных, находящихся на диске), с которого возможна загрузка из BIOS.

**Образ ОС** - бинарный файл, который загрузчик загружает в память и которому передает управление для старта ОС. Содержит ядро ОС.

**Загрузчик** - Программа или набор программ, которые загружают образ операционной системы. ЗАГРУЗЧИК может состоять из нескольких стадий, но это является деталью имплементации, не описываемой в данной спецификации. Только финальная стадия ЗАГРУЗЧИКА - стадия на которой управление передается операционной системе - должна соответствовать требованиям, описываемым в данном документе.

**Сектор диска** — минимальная адресуемая единица хранения информации на дисковых запоминающих устройствах. У большинства устройств размер сектора составляет 512 байт.

# Рабочая среда

Используется операционная система Linux Debian 8.2

## Используемые компоненты

* Компилятор ассемблерного кода nasm
* Компилятор кода на Си gcc
* Утилита make для автоматизации сборки
* Виртуальная машина qemu
* Отладчик cgdb
* Загрузчик GRUB

## Подготовка рабочего окружения

1. Устанавливаем дистрибутив Linux Debian 8.2 32 bit
2. Устанавливаем пакеты: nasm, qemu, cgdb, make, gcc  
   sudo apt-get install nasm qemu cgdb make gcc
3. Для загрузки файла конфигурации отладчика GDB в папке проекта необходимо создать файл “ .gdbinit” с содержанием “ set auto-load safe-path /”

## Команды окружения

### Утилита автоматизации сборки Make

* sudo make compile - компиляция ядра  
  описан в [Компиляция исходных кодов ядра](#_Компиляция_исходных_кодов)
* sudo make image - полное создание образа (с обновлением загрузочной области)  
  описан в [Сборка загрузочного образа (hdd.img)](#_Сборка_загрузочного_образа)
* sudo make - компиляция ядра и обновление его в образе (без обновления загрузочной области ядра)
* sudo clean - очистка файлов компиляции ядра
* sudo make umount -i - высвобождение занятых ресурсов (запускать, если какая-то команда не была завершена из-за ошибок)

### Вспомогательные bash скрипты

* bash run\_qemu.sh - запуск образа в виртуальной машине QEMU
* bash run\_qemu\_dbg.sh - запуск образа в qemu c подключением отладчика CGDB
* bash to\_flash.sh - запись образа на USB flash.

## Основные инструменты

### Виртуальная машина Qemu

Управление

* Ctrl + Alt - вернуть мышку
* Ctrl + Alt + 2 - открыть консоль команд
* Ctrl + Up – перемещение вверх в консоли команд

Команды в консоли команд

* x/152hx 0x7c00 - просмотр 152 слов виртуальной памяти
* xp/152hx 0x7c00 – просмотр 152 слов физической памяти
* pmemsave 0 0xffffff file\_name – сохранение дампа памяти в файл
* info registers - вывод полного содержимого внутренних регистров машины, включая скрытое состояние селекторов сегментов, локальную (LDT) и глобальную (GDT) таблицы дескрипторов и регистр задачи. Виртуальный ЦП считывает скрытое состояние из GDT/LDT, когда загружает селектор сегмента. Ниже приведено содержимое CS при работе ядра JOS и значения каждого поля: CS =0008 10000000 ffffffff 10cf9a00 DPL=0 CS32 [-R-] CS =0008 Видимая часть селектора сегмента. Значение селектора сегмента равно 0x8. Это означает, что мы ссылаемся на глобальную таблицу дескрипторов (0x8&4=0), текущий уровень привилегий (CPL, current privilege level) равен 0x8&3=0. 10000000 Начало этого сегмента. Linear address = logical address + 0x10000000. ffffffff Граница этого сегмента. Линейные адреса более 0xffffffff вызовут исключение «нарушение границы сегмента». 10cf9a00 Флаги для этого сегмента, значения которых QEMU декодирует в следующих нескольких полях. DPL=0 Уровень привилегий для этого сегмента. Только процесс с уровнем 0 может загрузить этот сегмент. CS32 Это 32-разрядный сегмент кода. Другие возможные значения: DS (сегмент данных) и LDT (локальная таблица дескрипторов). [-R-]
* system\_reset — перезапуск исполняемой программы

Запуск загрузочной флешки с именем устройства sdb в виртуальной машине:  
sudo chown username /dev/sdb  
qemu-system-i386 -hda /dev/sdb

Документация по QEMU  
http://www.opennet.ru/docs/RUS/qemu\_doc/

### Отладчик GDB

Отладчик использует скрипт инициализации .gdbinit, в нём можно отредактировать необходимые параметры.

Команды отладки

* с – continue
* n – next
* s – into
* p – вывод значения переменной
* b main – точка останова на имени функции
* b \*0x7C00 – точка остановка на адресе
* watch \*0xb8000 – точка останова на изменение памяти по адресу
* x /2 $pc – посмотреть три следующие инструкции
* si – выполнить одну инструкцию

Инструкция:  
<http://server.179.ru/tasks/gdb/>

## Исследование готовых файлов

* Дизассемблирование исполняемого файла  
  objdump MiniOS.bin.dbg -D > MiniOS.txt
* Просмотр файла в hex  
  hexdump -n 512 -s 10
* Просмотр “program header” исполняемого файла  
  readelf -l
* Просмотр “section header” исполняемого файла  
  readelf -S

# Файлы проекта

## Список файлов

* Makefile  
  основной makefile, автоматизирует создание загрузочного образа hdd.img  
  см. [Сборка загрузочного образа (hdd.img)](#_Сборка_загрузочного_образа)
* hdd.img  
  загрузочный образ MiniOS  
  см. Таблица 1 Структура загрузочного образа (hdd.img)
* run\_qemu.sh  
  запуск загрузочного образа hdd.img в виртуальной машине Qemu
* run\_qemu\_dbg.sh  
  запуск загрузочного образа hdd.img в виртуальной машине Qemu c отладчиком GDB
* to\_flash.sh  
  запись загрузочного образа hdd.img на USB Flash
* .gdbinit  
  скрипт инициализации отладчика GDB

### Папка grub/

Файлы загрузчика

* fat\_stage1\_5  
  промежуточная часть, содержит драйвер файловой системы fat32
* stage1  
  начальная часть, записывается в MBR
* stage2  
  основная часть
* menu.lst  
  файл конфигурации загрузчика

### Папка src/

Исходный код ядра

* MiniOS.bin  
  исполняемый файл ядра  
  см. [Исследование исполняемого файла MiniOS](#_/Исполняемый_файл_ОС)
* MiniOS.bin.dbg  
  исполняемый файл ядра с отладочной информацией для dbg
* Makefile  
  производит компиляцию исходных кодов ядра  
  см. [Компиляция исходных кодов ядра](#_Компиляция_исходных_кодов)
* link.ld  
  скрипт компоновщика  
  описан: Скрипт компоновщика
* common.h
* common.c
* descriptor\_tables.h
* descriptor\_tables.c
* isr.h
* isr.c
* monitor.h
* monitor.c
* timer.c
* multiboot.h
* timer.h
* main.c
* boot.s
* gdt.s
* interrupt.s

## Файлы для сборки

### Скрипт компоновщика

Задача компоновщика — объединить (обработать все связи) несколько скомпилированных объектных файлов (с расширениями .o) в исполняемый бинарный файл формата ELF.

В проекте используется компоновщик ld. Скрипт настроек компоновщика – link.ld В нем указывается точка входа ядра и разделы выходного исполняемого файла.

Определяются разделы:

* .text – исполняемый код
* .data – инициализированные данные (значения переменных)
* .rodata - содержит константные значения, этот раздел «только для чтения»
* .bss – неинициализированные данные  
  Переменные без начальных значений, не занимает места в исполняемом файле, сохраняется только его размер. При запуске программы раздел заполняется нулями.

Компоновщик запускается автоматически из Make файла/src/makefile

Результат работы компоновщика детально рассмотрен в [исследовании исполняемого файла MiniOS](#_/Исполняемый_файл_ОС).

Описание работы компоновщика:  
<http://habrahabr.ru/post/150327/>

**Используемые функции**

ENTRY(start) - позволяет задать входную точку для нашего ядра

ALIGN(4096) – устанавливает выравнивание начала раздела по 4096 байт (4 КБ)

**Содержимое файла**

ENTRY(start)

SECTIONS

{

.text 0x100000 :

{

code = .; \_code = .; \_\_code = .;

\*(.text)

. = ALIGN(4096);

}

.data :

{

data = .; \_data = .; \_\_data = .;

\*(.data)

\*(.rodata)

. = ALIGN(4096);

}

.bss :

{

bss = .; \_bss = .; \_\_bss = .;

\*(.bss)

. = ALIGN(4096);

}

end = .; \_end = .; \_\_end = .;

}

### Make файлы автоматизации сборки

make — утилита, автоматизирующая процесс преобразования файлов из одной формы в другую. Чаще всего это компиляция исходного кода в объектные файлы и последующая компоновка в исполняемые файлы или библиотеки.

Утилита использует специальные make-файлы, в которых указаны зависимости файлов друг от друга и правила для их удовлетворения. На основе информации о времени последнего изменения каждого файла make определяет и запускает необходимые программы.

В проекте используется два make-файла:

• src/makefile — собирает бинарный исполняемый файл ядра  
см. [Компиляция исходных кодов ядра](#_Компиляция_исходных_кодов)

• makefile — основной файл, собирает загрузочный образ. Для сборки бинарного файла ядра использует предыдущий make файл.  
см. [Сборка загрузочного образа](#_Сборка_загрузочного_образа)

Синтаксис файлов описан в  
http://www.gnu.org/software/make/manual/make.html

### Компиляция исходных кодов ядра

Это действие производит src/makefile. Здесь описано его содержимое.

Этот файл запускается только из makefile в папке проекта.

Результатом работы этого скрипта является исполняемый файл MiniOS.bin. Его структура описана в Таблица 2 Структура исполняемого файла (MiniOS.bin).

Описание объектного файла формата ELF  
http://www.sco.com/developers/gabi/latest/contents.html

Служебные переменные:

• OBJFILES — список объектных файлов

• CFLAGS — ключи gcc для компиляции Си файлов

• LDFLAGS — ключи компоновщика ld

• ASFLAGS — ключи nasm для компиляции кода на ассемблере

Ключи gcc:

• Wall — включает вывод предупреждений

• fno-builtin — не использовать встроенные функции компилятора gcc

• fno-stack-protector — не использовать защиту стека от переполнения и выполнения произвольного кода (для защиты добавляются дополнительные интрукции)

• nostdinc — не искать заголовочные файлы в стандартных системных директориях

• nostdlib — не использовать стандартные библиотеки

• ggdb3 — добавить отладочную информацию 3 уровня (максимальную) для отладчика gdb

• m32 — генерировать 32 битный код (важно, если компилируется на 64 битной системе)

Ключи nasm:

• felf — ассемблировать в ELF-объектный файл

Используемые ключи компоновщика ld:

• Tlink.ld — использовать компоновочный скрипт из файла link.ld

• melf\_i386 — эмулировать elf\_i386

Содержимое файла /src/makefile

SOURCES=boot.o main.o monitor.o common.o descriptor\_tables.o isr.o interrupt.o gdt.o timer.o

CFLAGS = -Wall -fno-builtin -fno-stack-protector -nostdinc -nostdlib -ggdb3 -m32

LDFLAGS=-Tlink.ld -melf\_i386

ASFLAGS=-felf

all: $(SOURCES) link

clean:

rm -f \*~ \*.o MiniOS.bin MiniOS.bin.dbg

link:

ld $(LDFLAGS) -o MiniOS.bin $(SOURCES)

cp MiniOS.bin MiniOS.bin.dbg

strip MiniOS.bin

.s.o:

nasm $(ASFLAGS) $

### Сборка загрузочного образа (hdd.img)

Сборку производит make файл в папке проекта. В результате получим загрузочный образ, который может быть записан на flash, или диск.

См. Таблица 1 Структура загрузочного образа (hdd.img)

Makefile, описание цели “image”:

1. Создаем пустой файл, заполняем его нулями

dd (dataset definition) — программа UNIX, предназначенная для копирования файлов

Параметры:

if=файл — читает данные из файла вместо стандартного ввода  
of =файл - пишет данные в файл вместо стандартного ввода  
bs=n - размер блока  
count - сколько блоков скопировать  
512\*16065 байт - минимальный размер, с которым может работать утилита fdisk

dd if=/dev/zero of=./hdd.img bs=512 count=16065 1>/dev/null 2>&1

2. Монтируем этот файл на устройство loop1

losetup /dev/loop1 ./hdd.img

3. Производим с помощью утилиты fdisk создание начального загрузочного раздела FAT32

Вводим команды:  
 с :- Переключения флага совместимости DOS  
 u :- Изменения единиц измерения экрана / содержимого (цилиндры / секторы) - в цилиндры  
 n :- Добавить новый раздел  
 p :- Тип раздела – Первичный  
 1 - номер раздела  
 - первый цилиндр 1

a :- Переключать загрузочный флаг (включаем на первом разделе)  
 t :- Изменить тип раздела (t change a partition type)  
 c :- тип W95 FAT32 (LBA)  
 w :- Таблица записи на диск и выход

(echo c; echo u; echo n; echo p; echo 1; echo ; echo ; echo a; echo t; echo c; echo w;) | fdisk /dev/loop1 1>/dev/null 2>&1 || true

4. Монтируем сам раздел на /dev/loop2 (смотрим в таблице разделов начало и конец раздела)

losetup /dev/loop2 ./hdd.img \  
 --offset `echo \`fdisk -lu /dev/loop1 | sed -n 9p | awk '{print $$3}'\`\*512 | bc` \  
 --sizelimit `echo \`fdisk -lu /dev/loop1 | sed -n 9p | awk '{print $$4}'\`\*512 | bc`

Отмонтируем раздел /dev/loop1

losetup -d /dev/loop1

5. Форматируем раздел в FAT

mkdosfs /dev/loop2

6. Копируем ядро kernel.bin и файлы загрузчика Grub на этот раздел (монтируем раздел в папку tempdir)

@mkdir -p tempdir  
@mount /dev/loop2 tempdir  
@mkdir tempdir/boot  
@cp -r grub tempdir/boot/  
@cp ./src/kernel.bin tempdir/  
@cp ./initrd tempdir/  
@sleep 1  
@umount /dev/loop2  
@rm -r tempdir  
@losetup -d /dev/loop2

7. Устанавливаем загрузчик Grub  
batch - включить неинтерактивный режим

@echo "device (hd0) hdd.img \n \  
root (hd0,0) \n \  
setup (hd0) \n \  
quit\n" | grub --batch 1>/dev/null

## Исходные коды ядра ОС

### boot.s

### main.c

# Проектирование исходного кода ядра ОС

## Таблицы IDT и GDT

Файлы, в которых реализован код этой главы

* descriptor\_tables.c / .h  
  Начальная инициализация (заполнение) таблиц IDT и GDT
* gdt.s  
  Функции на ассемблере для загрузки в соответствующие регистры новых IDT и GDT
* interrupt.s  
  Низкоуровневые (на ассемблере) обработчики прерываний. На них указывают записи в таблице IDT.
* isr.c / .h  
  Высокоуровневые (на Си) обработчики прерываний

На следующей странице приведена общая диаграмма процесса инициализации таблиц. Там видно, что в начале вызывается функция init\_descriptor\_tables. В ней происходит вызов функции инициализации GDT. А затем и IDT.



### Инициализация таблицы дескрипторов сегментов GDT

Структура для одной строки таблицы GDT

// This structure contains the value of one GDT entry.

// We use the attribute 'packed' to tell GCC not to change

// any of the alignment in the structure.

struct gdt\_entry\_struct

{

u16int limit\_low; // The lower 16 bits of the limit.

u16int base\_low; // The lower 16 bits of the base.

u8int base\_middle; // The next 8 bits of the base.

u8int access; // Access flags, determine what ring this segment can be used in.

u8int granularity;

u8int base\_high; // The last 8 bits of the base.

} \_\_attribute\_\_((packed));

typedef struct gdt\_entry\_struct gdt\_entry\_t;

Объявляем массив из структур gdt\_entry\_t, это и будет наша таблица GDT.

gdt\_entry\_t gdt\_entries[5];

Функция заполнения строки под номером “num” в таблице GDT

static void gdt\_set\_gate(s32int num, u32int base, u32int limit, u8int access, u8int gran)

{

gdt\_entries[num].base\_low = (base & 0xFFFF);

gdt\_entries[num].base\_middle = (base >> 16) & 0xFF;

gdt\_entries[num].base\_high = (base >> 24) & 0xFF;

gdt\_entries[num].limit\_low = (limit & 0xFFFF);

gdt\_entries[num].granularity = (limit >> 16) & 0x0F;

gdt\_entries[num].granularity |= gran & 0xF0;

gdt\_entries[num].access = access;

}

Чтобы загрузить новую таблицу в регистр, мы будем использовать специальную команду ассемблера lgdt. Этой команде нужно передавать адрес структуры, в которой содержится указатель на первый элемент нашей таблицы GDT и указатель на конец таблицы. Вот такой формат имеет эта структура:

// This struct describes a GDT pointer. It points to the start of

// our array of GDT entries, and is in the format required by the

// lgdt instruction.

struct gdt\_ptr\_struct

{

u16int limit; // The upper 16 bits of all selector limits.

u32int base; // The address of the first gdt\_entry\_t struct.

} \_\_attribute\_\_((packed));

typedef struct gdt\_ptr\_struct gdt\_ptr\_t;

Функция загрузки новой таблицы в регистр. В ней выполняем команду ldtr и, затем сразу устанавливаем в сегментные регистры селекторы из новой таблицы GDT.

gdt\_flush:

mov eax, [esp+4] ; Get the pointer to the GDT, passed as a parameter.

lgdt [eax] ; Load the new GDT pointer

mov ax, 0x10 ; 0x10 is the offset in the GDT to our data segment

mov ds, ax ; Load all data segment selectors

mov es, ax

mov fs, ax

mov gs, ax

mov ss, ax

jmp 0x08:.flush ; 0x08 is the offset to our code segment: Far jump!

.flush:

ret

Создаем структуру для команды ldtr.

gdt\_ptr\_t gdt\_ptr;

Основная функция. Здесь заполняется структура gdt\_ptr. Далее заполняются строки таблицы GDT через функцию gdt\_set\_gate(). Когда заполнены все строки, вызывается функция gdt\_flush для загрузки новой таблицы в регистр.

static void init\_gdt()

{

gdt\_ptr.limit = (sizeof(gdt\_entry\_t) \* 5) - 1;

gdt\_ptr.base = (u32int)&gdt\_entries;

gdt\_set\_gate(0, 0, 0, 0, 0); // Null segment

gdt\_set\_gate(1, 0, 0xFFFFFFFF, 0x9A, 0xCF); // Code segment

gdt\_set\_gate(2, 0, 0xFFFFFFFF, 0x92, 0xCF); // Data segment

gdt\_set\_gate(3, 0, 0xFFFFFFFF, 0xFA, 0xCF); // User mode code segment

gdt\_set\_gate(4, 0, 0xFFFFFFFF, 0xF2, 0xCF); // User mode data segment

gdt\_flush((u32int)&gdt\_ptr);

}

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Смещение | Base | Limit | Access | Granularity | Описание сегмента |
| 0x0 | 0x0 | 0x0 | 0 | 0x0 | Пустая запись |
| 0x8 | 0x000000000 | 0xFFFFFFFF | DPL=0 Type=Code | 0xCF | Кода в режиме ядра |
| 0x10 | 0x000000000 | 0xFFFFFFFF | DPL=0 Type=Data | 0xCF | Данные в режиме ядра |
| 0x18 | 0x000000000 | 0xFFFFFFFF | DPL=3 Type=Code | 0xCF | Код в режиме пользователя |
| 0x20 | 0x000000000 | 0xFFFFFFFF | DPL=3 Type=Data | 0xCF | Данные в режиме пользователя |

В результате получаем такую таблицу GDT:

### Инициализация таблицы прерываний IDT

Код инициализации IDT достаточно похож на код инициализации GDT: используются аналогичные функции, но их реализация переписана под таблицу прерываний.

Структура строки IDT

// A struct describing an interrupt gate.

struct idt\_entry\_struct

{

u16int base\_lo; // The lower 16 bits of the address to jump to when this interrupt fires.

u16int sel; // Kernel segment selector.

u8int always0; // This must always be zero.

u8int flags; // More flags. See documentation.

u16int base\_hi; // The upper 16 bits of the address to jump to.

} \_\_attribute\_\_((packed));

typedef struct idt\_entry\_struct idt\_entry\_t;

Структура для передачи команде lidt

// A struct describing a pointer to an array of interrupt handlers.

// This is in a format suitable for giving to 'lidt'.

struct idt\_ptr\_struct

{

u16int limit;

u32int base; // The address of the first element in our idt\_entry\_t array.

} \_\_attribute\_\_((packed));

typedef struct idt\_ptr\_struct idt\_ptr\_t;

Объявляем массив idt\_entries из 256 элементов. Это и будет наша IDT. И объявляем структуру для передачи команде lidt.

idt\_entry\_t idt\_entries[256];

idt\_ptr\_t idt\_ptr;

Функция, заполняющая строку таблицы

static void idt\_set\_gate(u8int num, u32int base, u16int sel, u8int flags)

{

idt\_entries[num].base\_lo = base & 0xFFFF;

idt\_entries[num].base\_hi = (base >> 16) & 0xFFFF;

idt\_entries[num].sel = sel;

idt\_entries[num].always0 = 0;

// We must uncomment the OR below when we get to using user-mode.

// It sets the interrupt gate's privilege level to 3.

idt\_entries[num].flags = flags /\* | 0x60 \*/;

}

Функция загрузки в регистр

idt\_flush:

mov eax, [esp+4] ; Get the pointer to the IDT, passed as a parameter.

lidt [eax] ; Load the IDT pointer.

ret

На ассемблере напишем функции для обработки прерываний. Т.к. они все будут однотипными, то напишем макрос для их генерации. Функции сохраняют свой номер и код ошибки в стек. Причем у некоторых прерываний нет кода ошибки, в таком случае мы запишем в стек 0. А затем передают управление следующей функции. Макросы имеют один параметр – номер, который они будут сохранять в стек.

ISR\_NOERRCODE – обработчик программных прерываний без кода ошибки

%macro ISR\_NOERRCODE 1

global isr%1

isr%1:

cli ; Disable interrupts firstly.

push byte 0 ; Push a dummy error code.

push byte %1 ; Push the interrupt number.

jmp isr\_common\_stub ; Go to our common handler code.

%endmacro

ISR\_ERRCODE – обработчик программных прерываний с кодом ошибки

; This macro creates a stub for an ISR which passes it's own

; error code.

%macro ISR\_ERRCODE 1

global isr%1

isr%1:

cli ; Disable interrupts.

push byte %1 ; Push the interrupt number

jmp isr\_common\_stub

%endmacro

IRQ number – обработчик аппаратных прерываний

; This macro creates a stub for an IRQ - the first parameter is

; the IRQ number, the second is the ISR number it is remapped to.

%macro IRQ 2

global irq%1

irq%1:

cli

push byte 0

push byte %2

jmp irq\_common\_stub

%endmacro

Создаем по этим макросам функции, указываем значения параметров:

ISR\_NOERRCODE 0

ISR\_NOERRCODE 1

ISR\_NOERRCODE 2

ISR\_NOERRCODE 3

ISR\_NOERRCODE 4

ISR\_NOERRCODE 5

ISR\_NOERRCODE 6

ISR\_NOERRCODE 7

ISR\_ERRCODE 8

ISR\_NOERRCODE 9

ISR\_ERRCODE 10

ISR\_ERRCODE 11

ISR\_ERRCODE 12

ISR\_ERRCODE 13

ISR\_ERRCODE 14

ISR\_NOERRCODE 15

ISR\_NOERRCODE 16

ISR\_NOERRCODE 17

ISR\_NOERRCODE 18

ISR\_NOERRCODE 19

ISR\_NOERRCODE 20

ISR\_NOERRCODE 21

ISR\_NOERRCODE 22

ISR\_NOERRCODE 23

ISR\_NOERRCODE 24

ISR\_NOERRCODE 25

ISR\_NOERRCODE 26

ISR\_NOERRCODE 27

ISR\_NOERRCODE 28

ISR\_NOERRCODE 29

ISR\_NOERRCODE 30

ISR\_NOERRCODE 31

IRQ 0, 32

IRQ 1, 33

IRQ 2, 34

IRQ 3, 35

IRQ 4, 36

IRQ 5, 37

IRQ 6, 38

IRQ 7, 39

IRQ 8, 40

IRQ 9, 41

IRQ 10, 42

IRQ 11, 43

IRQ 12, 44

IRQ 13, 45

IRQ 14, 46

IRQ 15, 47

И основная функция для инициализации IDT. В ней заполняем параметры структуры для загрузки в регистр idtr. Далее заполняем нулями наш массив idt\_entries (будущую таблицу IDT). Делаем перенастройку аппаратных прерываний на номера 32-47. Это необходимо, т.к. изначально они имеют те же номера что и некоторые программные прерывания. Заполняем строки таблицы, в качестве обработчиков указываем указатели на наши функции. И в конце выполняем загрузку в регистр.

static void init\_idt()

{

idt\_ptr.limit = sizeof(idt\_entry\_t) \* 256 -1;

idt\_ptr.base = (u32int)&idt\_entries;

memset(&idt\_entries, 0, sizeof(idt\_entry\_t)\*256);

// Remap the irq table.

outb(0x20, 0x11);

outb(0xA0, 0x11);

outb(0x21, 0x20);

outb(0xA1, 0x28);

outb(0x21, 0x04);

outb(0xA1, 0x02);

outb(0x21, 0x01);

outb(0xA1, 0x01);

outb(0x21, 0x0);

outb(0xA1, 0x0);

idt\_set\_gate( 0, (u32int)isr0 , 0x08, 0x8E);

idt\_set\_gate( 1, (u32int)isr1 , 0x08, 0x8E);

idt\_set\_gate( 2, (u32int)isr2 , 0x08, 0x8E);

idt\_set\_gate( 3, (u32int)isr3 , 0x08, 0x8E);

idt\_set\_gate( 4, (u32int)isr4 , 0x08, 0x8E);

idt\_set\_gate( 5, (u32int)isr5 , 0x08, 0x8E);

idt\_set\_gate( 6, (u32int)isr6 , 0x08, 0x8E);

idt\_set\_gate( 7, (u32int)isr7 , 0x08, 0x8E);

idt\_set\_gate( 8, (u32int)isr8 , 0x08, 0x8E);

idt\_set\_gate( 9, (u32int)isr9 , 0x08, 0x8E);

idt\_set\_gate(10, (u32int)isr10, 0x08, 0x8E);

idt\_set\_gate(11, (u32int)isr11, 0x08, 0x8E);

idt\_set\_gate(12, (u32int)isr12, 0x08, 0x8E);

idt\_set\_gate(13, (u32int)isr13, 0x08, 0x8E);

idt\_set\_gate(14, (u32int)isr14, 0x08, 0x8E);

idt\_set\_gate(15, (u32int)isr15, 0x08, 0x8E);

idt\_set\_gate(16, (u32int)isr16, 0x08, 0x8E);

idt\_set\_gate(17, (u32int)isr17, 0x08, 0x8E);

idt\_set\_gate(18, (u32int)isr18, 0x08, 0x8E);

idt\_set\_gate(19, (u32int)isr19, 0x08, 0x8E);

idt\_set\_gate(20, (u32int)isr20, 0x08, 0x8E);

idt\_set\_gate(21, (u32int)isr21, 0x08, 0x8E);

idt\_set\_gate(22, (u32int)isr22, 0x08, 0x8E);

idt\_set\_gate(23, (u32int)isr23, 0x08, 0x8E);

idt\_set\_gate(24, (u32int)isr24, 0x08, 0x8E);

idt\_set\_gate(25, (u32int)isr25, 0x08, 0x8E);

idt\_set\_gate(26, (u32int)isr26, 0x08, 0x8E);

idt\_set\_gate(27, (u32int)isr27, 0x08, 0x8E);

idt\_set\_gate(28, (u32int)isr28, 0x08, 0x8E);

idt\_set\_gate(29, (u32int)isr29, 0x08, 0x8E);

idt\_set\_gate(30, (u32int)isr30, 0x08, 0x8E);

idt\_set\_gate(31, (u32int)isr31, 0x08, 0x8E);

idt\_set\_gate(32, (u32int)irq0, 0x08, 0x8E);

idt\_set\_gate(33, (u32int)irq1, 0x08, 0x8E);

idt\_set\_gate(34, (u32int)irq2, 0x08, 0x8E);

idt\_set\_gate(35, (u32int)irq3, 0x08, 0x8E);

idt\_set\_gate(36, (u32int)irq4, 0x08, 0x8E);

idt\_set\_gate(37, (u32int)irq5, 0x08, 0x8E);

idt\_set\_gate(38, (u32int)irq6, 0x08, 0x8E);

idt\_set\_gate(39, (u32int)irq7, 0x08, 0x8E);

idt\_set\_gate(40, (u32int)irq8, 0x08, 0x8E);

idt\_set\_gate(41, (u32int)irq9, 0x08, 0x8E);

idt\_set\_gate(42, (u32int)irq10, 0x08, 0x8E);

idt\_set\_gate(43, (u32int)irq11, 0x08, 0x8E);

idt\_set\_gate(44, (u32int)irq12, 0x08, 0x8E);

idt\_set\_gate(45, (u32int)irq13, 0x08, 0x8E);

idt\_set\_gate(46, (u32int)irq14, 0x08, 0x8E);

idt\_set\_gate(47, (u32int)irq15, 0x08, 0x8E);

idt\_flush((u32int)&idt\_ptr);

}

### Обработка прерывания

При возникновении прерывания, выполнение переходит к обработчику, записанному в IDT. Код начальных обработчиков создается компилятором на основе макросов:

; This macro creates a stub for an ISR which does NOT pass it's own

; error code (adds a dummy errcode byte).

%macro ISR\_NOERRCODE 1

global isr%1

isr%1:

cli ; Disable interrupts firstly.

push byte 0 ; Push a dummy error code.

push byte %1 ; Push the interrupt number.

jmp isr\_common\_stub ; Go to our common handler code.

%endmacro

; This macro creates a stub for an ISR which passes it's own

; error code.

%macro ISR\_ERRCODE 1

global isr%1

isr%1:

cli ; Disable interrupts.

push byte %1 ; Push the interrupt number

jmp isr\_common\_stub

%endmacro

; This macro creates a stub for an IRQ - the first parameter is

; the IRQ number, the second is the ISR number it is remapped to.

%macro IRQ 2

global irq%1

irq%1:

cli

push byte 0

push byte %2

jmp irq\_common\_stub

%endmacro

Эти обработчики сохраняют свой номер в стек и вызывают следующую функцию. Программные и аппаратные обработчики работают по разному.

В начале рассмотрим обработку программных прерываний. Вызывается функция isr\_common\_stub. Она сохраняет в стек значения регистров и передает управление функции на Си isr\_handler.

; This is our common ISR stub. It saves the processor state, sets

; up for kernel mode segments, calls the C-level fault handler,

; and finally restores the stack frame.

isr\_common\_stub:

pusha ; Pushes edi,esi,ebp,esp,ebx,edx,ecx,eax

mov ax, ds ; Lower 16-bits of eax = ds.

push eax ; save the data segment descriptor

mov ax, 0x10 ; load the kernel data segment descriptor

mov ds, ax

mov es, ax

mov fs, ax

mov gs, ax

call isr\_handler

pop ebx ; reload the original data segment descriptor

mov ds, bx

mov es, bx

mov fs, bx

mov gs, bx

popa ; Pops edi,esi,ebp...

add esp, 8 ; Cleans up the pushed error code and pushed ISR number

sti

iret ; pops 5 things at once: CS, EIP, EFLAGS, SS, and ESP

Содержимое isr\_handler. Она вызывает пользовательскую функцию обработки прерывания из массива interrupt\_handlers. Этот массив мы объявляем чуть раньше, в нем хранятся указатели на функции, написанные пользователем. Далее, чтобы установить свой обработчик прерывания, потребуется только записать в этот массив указатель на функцию обработки этого прерывания.

// This gets called from our ASM interrupt handler stub.

void isr\_handler(registers\_t regs)

{

monitor\_write("recieved interrupt: ");

monitor\_write\_dec(regs.int\_no);

monitor\_put('\n');

if (interrupt\_handlers[regs.int\_no] != 0)

{

isr\_t handler = interrupt\_handlers[regs.int\_no];

handler(regs);

}

}

Параметры этой функции – структура:

typedef struct registers

{

u32int ds; // Data segment selector

u32int edi, esi, ebp, esp, ebx, edx, ecx, eax; // Pushed by pusha.

u32int int\_no, err\_code; // Interrupt number and error code (if applicable)

u32int eip, cs, eflags, useresp, ss; // Pushed by the processor automatically.

} registers\_t;

Её сформировали в стеке предыдущие функции.

Если был установлен наш обработчик прерывания, он вызывается.

Далее приведены диаграммы обработки прерываний.





### Установка своих обработчиков прерываний

Для установки своей функции, в качестве обработчика прерывания, написана функция:

void register\_interrupt\_handler(u8int n, isr\_t handler)

{

interrupt\_handlers[n] = handler;

}

Первым параметром она принимает номер прерывания, а вторым – указатель на нашу функцию.

Вот пример использования этой функции для установки обработчика на срабатывание таймера (аппаратное прерывание IRQ0).

register\_interrupt\_handler(IRQ0, &timer\_callback);

Теперь каждый раз при возникновении этого прерывания будет вызываться функция timer\_callback.

# Диаграммы

Диаграмма Структура загрузочного образа (hdd.img)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Сектора | Название области | Содержимое | Описание | |
| 0 | **MBR** (Master Boot Record) Главная загрузочная область | Код загрузчика (446 байт) | GRUB stage 1, находится указатель на следующую часть (GRUB stage 1.5) | |
| Таблица разделов | Содержит информацию о каждом разделе (у нас один раздел) | |
| 1-62 | Не занятая область (пустая, т.к. начало разделов выравнивается по 64 сектора) | GRUB stage 1.5 (32 256 байт), содержит драйверы файловой системы и адрес GRUB stage 2 (через указание полного пути и имени файла) | | |
| 63-16064 | Первый раздел, форматируется в файловую систему FAT32 | **VBR** (Volume Boot Record) Загрузочный сектор раздела | Этот сектор отвечает за информацию о размере кластера, тип и размер используемой таблицы расположения файлов. | |
| Файловая система | /boot/grub | GRUB stage 2, содержит стандартный файл конфигурации и прочие модули |
| /MiniOS.bin | Исполняемый файл ядра Mini OS |

|  |  |  |  |
| --- | --- | --- | --- |
| Смещение | Сегмент | Содержимое | Раздел |
| 0x0000 |  | Заголовок ELF |  |
| 0x002a |  | Программные заголовки   * 1. Первый сегмент, LOAD   2. Второй сегмент, GNU\_STACK |  |
| 0x1000 | Первый сегмент | Структура mboot | .text |
| 0x1020 | Исполняемый код, дополнен пустыми командами |
| 0х3000 | Информация для обработки исключений | .eh\_frame |
| 0х33bc | Значения инициализированных переменных | .data |
| 0x4000 | Неинициализированные переменные (не занимает места, указывается размер в заголовке этого раздела) | .bss |
| 0х4000 | Комментарии (версия компилятора) | .comment |
| 0x401c | Таблица названий разделов | .shstrtab |
| 0x404c |  | Заголовки разделов, каждый по 40 байт, всего 7 (первый пустой) |  |

Таблица Структура исполняемого файла (MiniOS.bin)

Таблица Multiboot information structure

|  |  |  |  |
| --- | --- | --- | --- |
| Начало | Название поля | Описание | Условие корректного заполнения |
| 0 | flags | Флаги, какие части этой структуры корректно заполнены | Обязательно присутствует |
| 4 8 | mem\_lower mem\_upper | Количество памяти (в килобайтах) | Заполнены, если flags[0] (нулевой бит flags) установлен |
| 12 | boot\_device | С какого диска было загружено ядро ОС | flags[1] |
| 16 | cmdline | Содержит физический адрес командной строки передаваемой ядру | flags[2] |
| 20 24 | mods\_count mods\_addr | Загруженные модули ядра и их адреса | flags[3] |
| 28 - 40 | syms | Информация о таблице символов ядра  Или таблице разделов ELF файла ядра | flags[4] или flags[5] |
| 44  48 | mmap\_length  mmap\_addr | Информация о “memory map” | flags[6] |
| 52 56 | drives\_length drives\_addr | Информация о диске | flags[7] |
| 60 | config\_table | Таблица конфигурации, возвращаемая BIOS (INT 15, AH=C0h) | flags[8] |
| 64 | boot\_loader\_name | Адрес названия загрузчика | flags[9] |
| 68 | apm\_table | Advanced Power Management | flags[10] |
| 72 76 80 82 84 86 | vbe\_control\_info vbe\_mode\_info vbe\_mode vbe\_intervace\_seg vbe\_interface\_off vbe\_interface\_len | Информация о поддерживаемых видеорежимах | flags[11] |

## 

Таблица Содержимое Multiboot information structure в виртуальной машине QEMU







Таблица Содержимое GDT

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Смещение | Base | Limit | Access | Granularity | Описание сегмента |
| 0x0 | 0x0 | 0x0 | 0 | 0x0 | Пустая запись |
| 0x8 | 0x000000000 | 0xFFFFFFFF | DPL=0 Type=Code | 0xCF | Кода в режиме ядра |
| 0x10 | 0x000000000 | 0xFFFFFFFF | DPL=0 Type=Data | 0xCF | Данные в режиме ядра |
| 0x18 | 0x000000000 | 0xFFFFFFFF | DPL=3 Type=Code | 0xCF | Код в режиме пользователя |
| 0x20 | 0x000000000 | 0xFFFFFFFF | DPL=3 Type=Data | 0xCF | Данные в режиме пользователя |

Описание полей:

* У нас nonGate  
  <http://www.iakovlev.org/index.html?p=8806&m=1&l1=7>
* Описание бита гранулярности  
  <http://www.wasm.ru/article/190>
* И здесь

<http://habrahabr.ru/post/118881/>

# Результаты исследований

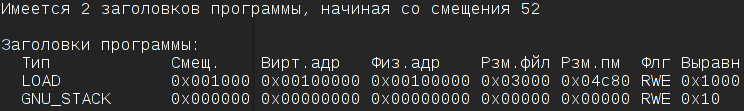
## Исполняемый файл ОС

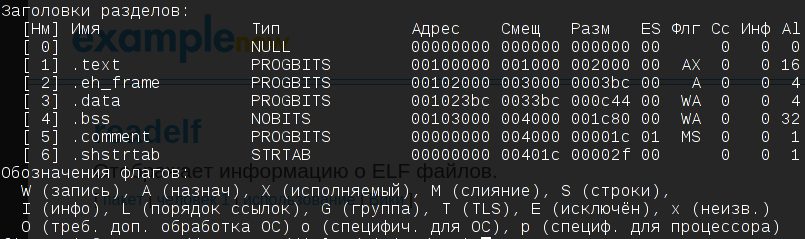
Структура файла:

У исполняемого файла, как результата нашей компиляции, есть два представления: он содержит program header (программный заголовок), описывающий сегменты, которые содержат информацию, используемую во время выполнения, и section header (заголовок разделов), описывающий разделы, которые содержат информацию для линковки и перемещения. Мы можем получить информацию о сегментах или разделах с помощью команд **readelf -l** или **readelf -S** соответственно.

**Заголовок ELF:**

Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00   
 Класс: ELF32  
 Данные: дополнение до 2, little endian  
 Версия: 1 (current)  
 OS/ABI: UNIX - System V  
 Версия ABI: 0  
 Тип: EXEC (Исполняемый файл)  
 Машина: Intel 80386  
 Версия: 0x1  
 Адрес точки входа: 0x100020  
 Начало заголовков программы: 52 (байт в файле)  
 Начало заголовков разделов: 35116 (байт в файле)  
 Флаги: 0x0  
 Размер этого заголовка: 52 (байт)  
 Размер каждого заголовка программы: 32 (байт)  
 Число заголовков программы: 2  
 Размер каждого заголовка раздела: 40 (байт)  
 Число заголовков разделов: 7  
 Индекс записи в таблице разделов, указывающий на таблицу названий разделов: 6

**Программные заголовки** (в каждом описывается один сегмент)

**Заголовки разделов**

Все разделы находятся в первом сегменте (тип LOAD).

Разделы:

* .text — содержит исполняемые инструкции программы
* .data — в этом разделе инициализируются глобальные и локальные переменные
* .rodata — содержит константные значения, этот раздел «только для чтения»
* .bss — раздел неинициализированных глобальных и локальных переменных, то есть заполненных нулями.
* .eh\_frame — информация, необходимая для frame-unwinding во время обработки исключений (облегчает отладку, можно убрать компилированием с флагом -fno-asynchronous-unwind-tables)
* .shstrtab — содержит таблицу названий разделов

Структура исполняемого файла описана в Таблица 2 Структура исполняемого файла (MiniOS.bin), которая была получена в результате исследования файла в hex-редакторе.

# Приложения

## Архитектура компьютера

## Режимы работы процессора

### Реальный режим (Real Mode)

Все инструкции 16-битные

Адресация памяти — сегментная: 16-битный номер сегмента и 16-битное смещение внутри этого сегмента

Размер каждого сегмента 64Кб

Всего доступно 1Мб памяти

В реальном режиме преобразование адреса происходит в соответствии со следующей формулой: physical address = 16 \* segment + offset. Например для адреса 0xf000:fff0 преобразование выглядит так: 16 \* 0xf000 + 0xfff0 = 0xf0000 + 0xfff0 = 0xffff0

### Защищенный режим (Protected Mode)

Основной режим работы процессора

Инструкции 32-битные

Виртуальное адресное пространство

Многозадачность

Защита (изолирование задач, контроль доступа к памяти)

## Ассемблер

Для файлов с расширением “.s” (содержат код на ассемблере), используется компилятор ассемблерного кода Nasm.

Документация по Nasm http://www.codenet.ru/progr/asm/nasm/

В NASM используется Intel-синтаксис записи инструкций. Предложение языка ассемблера NASM (строка программы) может состоять из следующих элементов:

Метка Инструкция Операнды Комментарий

Операнды отделяются между собой запятой. Перед строкой и после инструкции можно использовать любое количество пробельных символов. Комментарий начинается с точки с запятой, а концом комментария считается конец строки. В качестве инструкции может использоваться команда или псевдокоманда (директива компилятора). Если строка очень длинная, то её можно перенести на следующую, используя обратный слеш \, подобно тому, как это делается в языке Си.

### Команды работы с флагами

Флаг прерывания IF

Этот флаг определяет, разрешены в данный момент прерывания или нет (о прерываниях тоже будет отдельная статья).

Команда CLI сбрасывает флаг IF (запрещает прерывания).

Команд STI устанавливает флаг IF в единицу (разрешает прерывания).

### Дерективы Nasm

• Директива BITS указывает, код какой разрядности должен генерировать NASM: для процессора, работающего в 16-битном режиме, или для процессора в 32-битном режиме. Соответствующим синтаксисом будет BITS 16 или BITS 32.

Наиболее вероятным использование директивы BITS представляется при написании 32-битного плоского бинарного файла. (Выходной формат bin по умолчанию предназначен для 16-битного режима, т.к. он наиболее часто используется для написания DOS.COM программ, DOS.SYS драйверов устройств, а также загрузчиков).

• Директива SECTION (SEGMENT это абсолютно эквивалентный синоним) указывает, в какую секцию выходного файла будет ассемблирован код, который вы пишете. В некоторых объектных форматах количество и имена секций фиксированы; в других пользователь может сделать их столько, сколько захочет. Поэтому если вы захотите переключиться на секцию, которая в данный момент не существует, директива SECTION может либо вызвать сообщение об ошибке, либо создать новую секцию.

Объектные форматы Unix и bin поддерживают стандартизованные имена секций: .text для кода, .data для данных и .bss для неинициализированных данных. Формат obj наоборот, не признает эти имена секций специальными и более того, удаляет ведущую точку в имени любой секции.

• EXTERN: импорт символов из других модулей

Директива EXTERN подобна директиве MASM EXTERN и ключевому слову extern в С: она используется для объявления символа, который определен в некотором другом модуле.

• GLOBAL: Экспорт символов в другие модули

Директива GLOBAL это обратная сторона EXTERN: если один модуль объявляет символ как EXTERN и ссылается на него, то для предотвращения ошибок компоновщика необходимо, чтобы некоторый другой модуль определил этот символ и объявил его как GLOBAL.

## Взаимодействие кода на ассемблере и кода на Си

Соглашения, используемые для организации вызова. В GCC для архитектуры x86 при организации вызова используется следующее соглашение \_\_cdecl:

•Все параметры в функцию передаются через стек

•Параметры, перемещаемые в стек, перечисляются справа налево.

•Значение, возвращаемое функцией, помещается в EAX.

Так что вызов функции

d = func(a, b, c);

будет выглядеть следующим образом

push [c]  
push [b]  
push [a]  
call func  
mov [d], eax

## Ассемблерные вставки в код на Си

GCC Inline Assembly — Встроенный ассемблер компилятора GCC, представляющий собой язык макроописания интерфейса компилируемого высокоуровнего кода с ассемблерной вставкой.

### Особенности

Синтаксис и семантика GCC Inline Assembly имеет следующие существенные отличия:

* GCC не интерпретирует никак содержимое ассемблерной вставки.
* Служит явное описание интерфейса с ассемблерной вставкой.
* Даёт компилятору возможность свободы выбора регистров.
* Позволяет явно указать на имеющиеся побочные действия ассемблерного кода.
* Позволяет использовать все инструкции (и директивы тоже) которые распознает ассемблер, а не только те, что знает и применяет gcc

### Общая структура вставки

Общая структура ассемблерной вставки выглядит следующим образом:

asm [volatile] («команды и директивы ассемблера» : выходные параметры : входные параметры : изменяемые параметры);

впрочем, существует и более короткая форма:

asm [volatile] («команды ассемблера»);

### Синтаксис команд

Особенностью ассемблера gas и компилятора gcc есть тот факт, что они используют непривычный для x86 синтаксис AT&T, который существенно отличается от синтаксиса Intel. Основные отличия:

1. Порядок операндов: Операция Источник,Приёмник.
2. Названия регистров имеют явный префикс %, указывающий, что это регистр. Это позволяет работать с переменными, которые имеют то же название, что и какой-либо регистр, что невозможно в Intel-синтаксисе, у которого префиксы для регистров не используются, а их названия являются зарезервированными ключевыми словами.
3. Явное задание размеров операндов в суффиксах команд: b-byte, w-word, l-long, q-quadword. В командах типа movl %edx,%eax это может показаться излишним, однако является весьма наглядным средством, когда речь идёт о incl (%esi) или xorw $0x7,mask
4. Названия констант начинаются с $ и могут быть выражением. Например movl $1,%eax
5. Значение без префикса означает адрес. Например:  
   movl $123,%eax — записать в %eax число 123,  
   movl 123,%eax — записать в %eax содержимое ячейки памяти с адресом 123,  
   movl var,%eax — записать в %eax **значение** переменной var,  
   movl $var,%eax — загрузить **адрес** переменной var
6. Для косвенной адресации необходимо использовать круглые скобки. Например movl (%ebx),%eax — загрузить в %eax значение переменной, по адресу находящемуся в регистре %ebx
7. SIB-адресация: смещение(база, индекс, множитель)

### Ключевое слово volatile

Компилятор в результате оптимизации кода удаляет ассемблерные вставки, которые не изменяют значения переменных. Чтобы компилятор не удалил вставку, используется ключевое слово volatile.

### Пример вставки

asm volatile("         \

     cli;                 \

     mov %0, %%ecx;       \

     mov %1, %%esp;       \

     mov %2, %%ebp;       \

     mov %3, %%cr3;       \

     mov $0x12345, %%eax; \

     sti;                 \

     jmp \*%%ecx           "

                : : "r"(eip), "r"(esp), "r"(ebp), "r"(current\_directory->physicalAddr));

## Процесс работы загрузчика

BIOS

Загружает в память по адресу 0x7c00 первый сектор устройства(MBR, размер 512 байт) и передает ему управление. BIOS и загрузчик ОС работают в [реальном режиме](#_Реальный_режим_(Real).

GRUB stage 1

Находится в MBR (в первом секторе). Используется только чтобы загрузить следующую часть и передать ей управление. При этом, размещение этих файлов кодируется в терминах цилиндр/головка/сектор, и значит на этом этапе еще не требуется распознавание типа файловой системы.

GRUB stage 1.5

Включает в себя драйвер файловой системы (в нашем случае используется fat\_stage1\_5). Производит монтирование файловой системы активного раздела. Находит файлы следующей части загрузчика, загружает в память по адресу 0x8000 и передает им управление.

Эта часть нужна, чтобы GRUB stage 2 был доступен в виде обычных файлов в файловой системе (для упрощения доступа к нему).

GRUB stage 2

Получает информацию о машине от BIOS, читает конфигурационный файл. Выводит окно выбора загружаемой ОС. Производит загрузку ядра ОС в память, переводит процессор в [защищенный режим](#_Защищенный_режим_(Protected) и передает управление ядру ОС ([состояние машины при старте ОС](#_Состояние_машины_при)).

## Карта расположения загрузчика в памяти

* 0 to 4K-1  
  BIOS and real mode interrupts
* 0x07BE to 0x07FF  
  Partition table passed to another boot loader
* down from 8K-1  
  Real mode stack
* 0x2000 to ?  
  The optional Stage 1.5 is loaded here
* 0x2000 to 0x7FFF  
  Command-line buffer for Multiboot kernels and modules
* 0x7C00 to 0x7DFF  
  Stage 1 is loaded here by BIOS or another boot loader
* 0x7F00 to 0x7F42  
  LBA drive parameters
* 0x8000 to ?  
  Stage2 is loaded here
* The end of Stage 2 to 416K-1  
  Heap, in particular used for the menu
* down from 416K-1  
  Protected mode stack
* 416K to 448K-1  
  Filesystem buffer
* 448K to 479.5K-1  
  Raw device buffer
* 479.5K to 480K-1  
  512-byte scratch area
* 480K to 512K-1  
  Buffers for various functions, such as password, command-line, cut and paste, and completion.
* The last 1K of lower memory  
  Disk swapping code and data

## Состояние машины при старте ОС

Когда ЗАГРУЗЧИК передает управление 32-битной операционной системе, машина ДОЛЖНА находиться в следующем состоянии:

* %eax  
  Должен содержать число '0x2BADB002'; наличие этого значения сигнализирует о том, что управление было передано от совместимого мультизагрузчика (в отличие от других загрузчиков, которые также могут использоваться).
* %ebx  
  Должен содержать 32-битный физический адрес специальной структуры (Таблица 3 Multiboot information structure), подготовленной загрузчиком.
* %cs  
  Должен указывать на сегмент кода с соответствующими правами доступа к памяти. Значение может лежать в пределах от '0' до '0xFFFFFFFF'.
* %ds,%es,%fs,%gs,%ss  
  Должны указывать на сегменты данных с соответствующими правами доступа к памяти. Значение может лежать в пределах от '0' до '0xFFFFFFFF'.
* 'A20 gate'  
  Должна быть включена.
* %cr0  
  31 бит (PG) должен быть сброшен. 0 бит (PE) должен быть установлен. Значение остальных битов не определено.
* %eflags  
  17 бит (VM) должен быть сброшен. 9 бит (IF) должен быть сброшен. Значение остальных битов не определено.

Значения всех остальных регистров и флагов процессора не определены. Это, в частности, относится к:

* %esp  
  ОС ДОЛЖНА создать свой собственный стек сразу после получения управления.
* 'GDTR'  
  Хотя сегментные регистры установлены, регистр таблицы глобальных дескрипторов может содержать недействительное значение, поэтому ОС НЕ ДОЛЖНА загружать какие-либо сегментные регистры (даже пытаться перегружать теми же значениями) до тех пор, пока не создаст свою собственную таблицу GDT.
* 'IDTR'  
  ОС не ДОЛЖНА разрешать аппаратные прерывания, пока не настроит свою IDT.

Однако, загрузчику СЛЕДУЕТ оставить все регистры и флаги, не описанные выше, в том состоянии, в которое они были установлены BIOS. Другими словами, ОС должна быть в состоянии обратиться к BIOS после загрузки до тех пор, пока сама не перезапишет структуры данных BIOS. Кроме того, загрузчик ДОЛЖЕН оставить значения контроллера прерываний (PIC) теми, которые были установлены BIOS, даже если он изменил их в процессе переключения в защищенный режим.

# Ссылки

1. James Molloy, руководство по созданию ОС  
   оригинал <http://www.jamesmolloy.co.uk/tutorial_html/>  
   перевод <http://rus-linux.net/MyLDP/kernel/toyos/sozdaem-unix-like-os.html>
2. Статьи по разработке и устройству ОС  
   <http://www.iakovlev.org/>
3. Документация по компилятору Nasm  
   <http://biguniverse.narod.ru/nasm.pdf>