МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ «ВИТЕБСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ П.М. МАШЕРОВА»

Факультет математики и информационных технологий
Кафедра прикладного и системного программирования

Допущено к защите
«__»_____ 20__г.
Заведующий кафедрой
Ермоченко Сергей Александрович

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

Подходы к разработке современного системного программного обеспечения

Специальность *информатика и технологии программирования*

Ашыров Алишер,
2 курс, магистрант

Научный руководитель:
Трубников Юрий Валентинович,
профессор, доктор физико-математических наук

Витебск, 2022

РЕФЕРАТ

Магистерская диссертация 27с., 7 рис., 10 листингов, 6 источников.

ОПЕРАЦИОННАЯ СИСТЕМА, ЗАГРУЗЧИК, BIOS, ПРЕРЫВАНИЕ, РЕАЛЬНЫЙ РЕЖИМ, ЗАЩИЩЕННЫЙ РЕЖИМ, ЯДРО, ЧТЕНИЕ ДИСКА.

Объект исследования – операционные системы, принципы работы современных операционных систем.

Предмет исследования – принципы разработки операционных систем.

Цель работы – изучить и продемонстрировать различные подходы к разработке системного программного обеспечения на примере разработки операционных систем.

Методы исследования: описательно–аналитический, сравнительный.

Элементы новизны: продемонстрированы подходы и технологии разработки современных операционных систем.

Теоретическая и практическая значимость: работа помогает изучении операционных систем и разработки собственной операционной системы, предлагая необходимого набора инструментов и теоретической базы для начального этапе разработки.

СОДЕРЖАНИЕ

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ, УСЛОВНЫХ ОБОЗНАЧЕНИЙ, СИМВОЛОВ И ТЕРМИНОВ

BIOS --- Base input/output system;
UEFI --- Unified extensible firmware system;
NASM --- Netwide assembler;
ELF --- Executable and linking format;
QEMU --- Quick emulator;
GDT --- Global description table;

ВВЕДЕНИЕ

Никому не секрет, что без программного обеспечения любой компьютер — просто бесполезная груда железа. Благодаря программам компьютер может хранить, обрабатывать и искать информацию, воспроизводить музыку и видео, отсылать сообщения электронной почты, вести поиск в Интернете и решать множество других важных задач, для которых он и предназначен. Программное обеспечение можно условно разбить на две большие группы: системные программы, управляющие работой самого компьютера, и прикладные программы, предназначенные для решения пользовательских задач. Самая главная системная программа — это операционная система, она управляет всеми системными ресурсами и обеспечивает основу для работы прикладных программ.

Доменные области далеко не всех прикладных программных обесепчений пересекаются с управлением ресурсами компьютера. Кроме того, как показывает практика, интерфейсы аппаратного обеспечения не всегда остаются постоянными. Если бы каждому программисту приходилось задумываться о том, как работают жесткие диски, помнить о множестве нюансов, которые могут произойти при чтении блока данных, то многие программы, скорее всего, вообще не были бы написаны.

Если оглянуться назад в историю, то компьютеры первого поколения не имели операционных систем. Программы на первых ЭВМ включали в себя код для непосредственной работы системы, связи с периферийными устройствами и вычислений, для выполнения которых эта программа и писалась. Из-за такого расклада даже простые по логике работы программы были сложны в программной реализации. По мере того как компьютеры становились более разнообразными и сложными, писать программы, которые работали и как ОС, и как приложение, стало попросту неудобно.

Ещё на заре разработки прикладных программ стало очевидно, что нужно как-то оградить программистов от тонкостей, связанных с аппаратным обеспечением. Со временем был выработан следующий путь: поверх аппаратного обеспечения работает дополнительная программная прослойка, которая управляет всем оборудованием и предоставляет пользователю интерфейс, более простую для понимания и программирования, чем аппаратура. Операционная система и является этой программной прослойкой.

Операционную систему можно рассматривать с двух точек зрения: как менеджер ресурсов и как расширенную машину. Как менеджер ресурсов операционная система рационально управляет различными частями системы. С точки зрения расширенной машины, работа операционной системы состоит в предоставлении пользователям некой виртуальной машины, более удобной, чем настоящий компьютер с неудобным аппаратным интерфейсом.

Современные процесссоры и микроконтроллеры очень разнообразны по архитектуре и возмодностям, ноо представляют собой достаточно универсаьные вычислительные устройства способные решать широкий круг задач. В этой связи актуальным вопросом является разработка операционной системы общего

назначения, которая позволит более эффективно использовать возможности вычислительных устройств. Изучение и систематизация подходов по разработке такого рода программного обеспечения имеет практическую пользу, поскольку позволяет быстро адаптировать программное обеспечение под новые устройства и платформы.

В работе будет продемонстрированы основные методы и технологии разработки загрузчика и ядра операционных систем.

ОСНОВНАЯ ЧАСТЬ

1 Инструментарий разработки операционных систем

В начале исследовательской работы стоял вопрос выбора инструментарий разработки. Выбор был между системами Linux и Windows. После некоторого сравнения этих систем, выбор был сделан в пользу систем Linux, а именно Ubuntu 20.04 LTS (далее, Ubuntu) с архитектурой x86. Теоретически, разработку операционных систем можно вести на любой современной ОС, но большинство свободных инструментов рассчитаны на UNIX-подобные системы. Более того, поскольку WSL (Windows Subsystem for Linux) не поддерживает модули ядра, смонтировать образ диска не получится, и придется настраивать коммуникация между WSL и Windows. На этом этапе уже становится проще поставить виртуальную машину с Linux.

Для полного цикла разработки понадобятся система сборки, виртуальная машина и, желательно, отдельный реальный компьютер для конечного тестирования. Рассмотрим их отдельно.

1.1 Система сборки

Поскольку мы будем разработать собственный загрузчик для операционной системы, то для этих целей нам понадобится ассемблер. В ходе работ мы будем использовать ассемблер NASM. Поскольку он является самым популярным ассемблером для Linux систем, в следствии чего, по нему можно быстро найти ту или иную информацию. Кроме того, NASM является кроссплатформенным ассемблером и поддерживает программы на 16-bit, 32-bit и 64-bit пллатформах.

Язык NASM похож на другие ассемблерные языки, что позволяет быстро изучить разобраться в его синтаксисе. Мы не будем подробно останавливаться на его обзоре, а лишь приведем инструкции для установки и компиляции через NASM.

Для установки ассемблера NASM на ОС ubuntu 20.04:

1. Открыть терминал (что можно сделать через сочетания клавиш Ctrl+Alt+T);
2. Набрать в терминале следующие команды:

```
sudo apt update
sudo apt install
```

Как видно из рисунка 1.1 для генерации объектного файла из ассемблерного кода будет достаточно набрать команду в формате `nasm <filename> -f <format> -o <output>.`

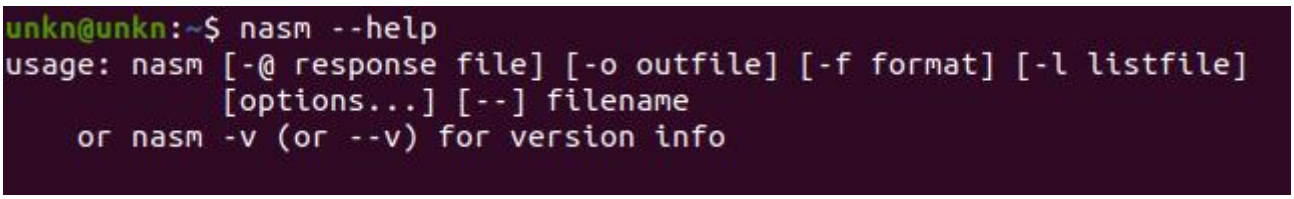

Рисунок 1.1 Часть справочного вывода команды nasm

Поскольку ядро операционной системы будем разрабатывать на высокоуровневом языке C, нам необходимо слинковать код ядра с кодом загрузчика системы. С другой стороны код загрузчика в 16-bit формате, то кроме ассемблера для сборки кода нам потребуется кросс-компилятор gcc для i386 архитектуры процессора. Для компьютеров на Linux с x86_64 архитектурой компильятор можно установить набрав в терминале следующие команды:

```
wget http://newos.org/toolchains/i386-elf-4.9.1-Linux-x86_64.tar.xz

mkdir /usr/local/i386elfgcc

tar -xf i386-elf-4.9.1-Linux-x86_64.tar.xz -C /usr/local/i386elfgcc
--strip-components=1

export PATH=$PATH:/usr/local/i386elfgcc/bin
```

Только что установленный компилятор gcc позволяет компилировать код на языке C в 16-bit формате, что позволяет линковать код загрузки с кодом ядра операционной системы.

1.2 Эмулятор устройств QEMU.

Для запуска загрузчика, ядра операционной системы нам потребуется виртуальная машина. На системах Linux на место виртуальной машиины лучше всего подходят Bochs и QEMU, поскольку они быстро запускаются и предоставляют возможность отладки ядра. В ходе разработки был выбран QEMU.

QEMU (Quick Emulator) позволяет запускать операционные системы, предназначенным под одну архитектуру, на другой. Кроме процессора, QEMU эмулирует различные перифирийные устроства: сетевые карты, HDD, видео карты, PCI, USB и т.п.

Для установки эмуляторв необходимо набрать в терминале следующие команды:

```
sudo apt update
sudo apt install qemu-kvm qemu
```

Образ ОС на эмуляторе можно запустить набрав команду

```
qemu <os boot disk image>
```

Вместо команды `qemu` можно запустить другие эмуляторы, такие как `qemu-system-i386, qemu-system-x86_64,` каждый из которых представляют эмуляцию архитектуру, которая указана постфиксом в конце названия каждого эмулятора.

1.3 Автоматизация сборки с помощью Makefile

Как и в других областях программного обеспечения, во время разработки операционной системы приходится много раз тестировать. Как мы увидим далее, для сборки загрузчика, её линковки с ядром операционной системы и, наконец, запуска на виртуальной машине необходимо набирать около десяти команд в терминале. Ясно, что такое положение дел обречено многочисленным ошибкам и обретает нежелательную рутину.

Для автоматизации сборки и запуска во время разработки применялось система сборки make. Основным принципом make является то, что он необходимо описать правила сборки в файле с названием Makefile. Makefile, помимо своего синтаксиса, допускает набирать в нем обычные команды в терминале, что позволяет нам легко автоматизировать сборку и запуск системы с минимумом знаний о его синтаксисе.

В общем виде, правило сборки в Makefile имеет следующий вид

```
<new rule>: <required rule 1>, <required rule 2>,..., <required rule n>
     <command 1>
     <command 2>
        ...
     <command n>
```

Первая строка описывает новое правило и после двоеточие приводится список необходимых правил для сборки текущего правила, при этом список может быть пустым. Порядок выполнения правил в списке совпадает с порядком записи. Далее приводится список команд для текущего правила.

Для запуска сборки необходимо набрать в папке, где находится файл Makefile, команду `make <rule name optional>`. По умолчанию `make` запускает правило под названием `all`.

2 Программирование загрузчика операционной системы

Прежде чем приступать к написанию ядра операционной системы, посмотрим как компьютер загружается и передает управление ядру. Запуск операционной системы можно условно разделить на несколько этапов, отлючающиеся выполняемой ими задачами: *пре-загрузочная, загрузка, запуск ядра.*

На *пре-загрузочной* этапе*,* запускается система BIOS (UEFI), которая предоставляет автоматическое определение и проверку всех подключенных устройств, такие как, монитор, клавиатура и жесткие диски. Система UEFI, которая пришла на замену BIOS, кроме всего функционала BIOS, предоставляет допольнительные функции, такие как, «Secure Boot», что предотвращает взлом и насанкционированный доступк ОС. В действительности, все современные компьютеры используют систему UEFI вместо BIOS. Но по историческим причинам, название BIOS используется и для UEFI. Поскольку в рамках работы разница между BIOS и UEFI не будет играет роли, далее будем использовать название BIOS.

Остановимся подробнее на задачах выполняемых системой BIOS. У BIOS три главных задач: обнаружение и проверка всех подключенных устройств, предоставление операционной системе базового набора функций для работы с аппаратурой и запуск загрузчика операционной системы.

Обнаружение всех подключенных устройств (процессор, клавиатуру, монитор, оперативную память, видеокарту) и проверка их на работоспособность. Отвечает за это программа POST (Power On Self Test – самотестирование при нажатии ВКЛ). Если жизненно важное железо не обнаружено, то никакая программа не сможет работать, и на этом этапе система подает храктерный звук.

Предоставление операционной системе базового набора функций для работы с аппаратурой. Например, как мы убедимся далее, через функции BIOS можно вывести текст на экране или считать данные с клавиатуры. Потому она и называется базовой системой ввода-вывода. Обычно операционная система получает доступ к этим функциям посредством прерываний.

Запуск загрузчика операционной системы. При этом, как правило, считывается загрузочный сектор — первый сектор носителя информации (дискета, жесткий диск, компакт-диск, флэшка). Порядок опроса носителей можно задать в настройках BIOS SETUP. В загрузочном секторе содержится программа, иногда называемая первичным загрузчиком. Условно говоря, задача загрузчика — начать запуск операционной системы. Процесс загрузки операционной системы может быть весьма специфичен и сильно зависит от её особенностей. Поэтому первичный загрузчик пишется непосредственно разработчиками ОС и при установке записывается в загрузочный сектор. В момент запуска загрузчика процессор находится в реальном режиме.

По историческим причинам сложилось так, что размер начального загрузчика всего 512 байт. На рисунке 2.1 изображена поверхность дискового накопителя. На каждой из двух поверхностей накопителя имеются дорожки, которые в свою

очередь делится на маленькие дугообразные кусочки, называемые секторами. Исторически сложилось, что один сектор на диске имеет размер 512 байт. Самый первый сектор на накопителе считается загрузочным. Единственное требование к загрузочному сектору -- это содержание в двух последних байтах значений 0x55 и 0xAA.

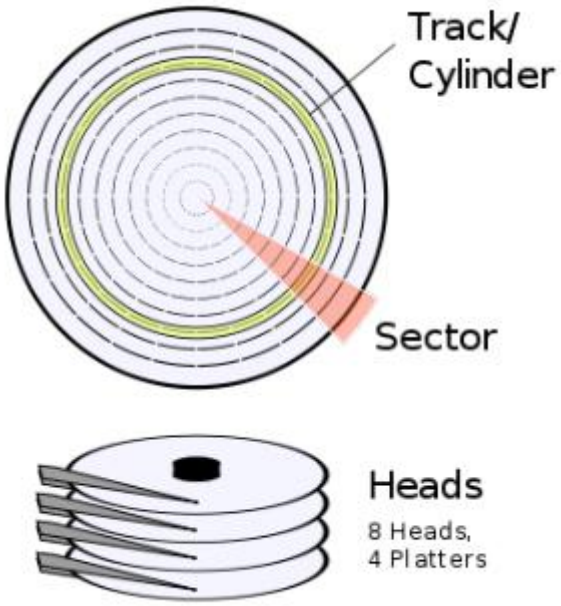

Рисунок 2.1 Поверхность дискового накопителя

На рисунке 2.2 приведен пример бинарного кода загрузочного сектора. Как видно из рисунка, действительно, этот сектор заканчивается на 0xAA55. Стоит отметить, что порядок байтов 0xAA и 0x55 на рисунке с записью 0xAA55. Такая разница связана с тем, что архитектура x86 обрабатывает данные в little-endian формате представления бинарных файлов.

```
00000000  88 16 82 7D BD 00 90 89 EC BB 83 7D E8 08 00 E8 4C 01 E8 FC 00 EB FE 60 B4 0E 8A 07 3C 00 74 07 EB 00 CD 10 43 EB F3 B4 0E B0 0A CD 10
0000002D  B0 0D CD 10 EB 00 61 C3 60 B9 00 00 83 F9 04 7C 02 EB 21 89 D0 83 E0 0F 83 F8 09 7F 02 EB 05 83 C0 27 EB 00 83 C0 30 BB 6E 7C 29 CB 88
0000005A  07 C1 CA 04 41 EB D8 BB 69 7C E8 B0 FF 61 C3 30 78 30 30 30 30 00 52 B4 02 88 F0 B5 00 B6 00 B1 02 CD 13 72 11 5A 38 C6 75 19 EB 02 EB
00000087  6D BB A3 7C E8 89 FF EB 65 BB BF 7C E8 81 FF 88 C6 E8 9A FF EB 56 BB D1 7C E8 74 FF 44 69 73 6B 20 77 61 73 20 73 75 63 63 65 73 73 66
000000B4  75 6C 6C 79 20 72 65 61 64 20 00 44 69 73 6B 20 72 65 61 64 20 65 72 72 6F 72 21 20 00 49 6E 63 6F 72 72 65 63 74 20 6E 75 6D 62 65 72
000000E1  20 6F 66 20 73 65 63 74 6F 72 73 20 72 65 61 64 20 00 EB FE C3 60 BA 00 80 0B 00 8A 03 B4 0F 3C 00 74 0B 66 89 02 83 C3 01 83 C2 02 EB
0000010E  ED 61 C3 FA 0F 01 16 58 7D 0F 20 C0 66 83 C8 01 0F 22 C0 EA 26 7D 08 00 66 B8 10 00 8E D8 8E D0 8E C0 8E E0 8E E8 BD 00 00 09 00 89 EC
0000013B  E8 31 00 00 00 00 00 00 00 00 00 00 00 00 FF FF 00 00 00 9A CF 00 FF FF 00 00 00 92 CF 00 17 00 40 7D 00 00 BB CC 7D E8 B3 FE BB 00 10 B6
00000168  10 8A 16 82 7D E8 00 FF C3 BB 9F 7D 00 00 E8 7B FF FF FF E8 80 92 FF FF EB FE 00 53 74 61 72 74 65 64 20 69 6E 20 31 36 2D 62 69 74 20
00000195  52 65 61 6C 20 4D 6F 64 65 00 53 75 63 63 65 73 73 66 75 6C 6C 79 20 6C 61 6E 64 65 64 20 69 6E 20 33 32 2D 62 69 74 20 50 72 6F 74 65
000001C2  63 74 65 64 20 6D 6F 64 65 00 4C 6F 61 64 69 6E 67 20 6B 65 72 6E 65 6C 20 69 6E 74 6F 20 56 49 44 45 4F 5F 4D 45 4D 4F 52 59 00 00 00
000001EF  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 AA
```

Рисунок 2.2 Пример бинарного кода загрузочного сектора. Каждому шестнадцатеричному числу соответствует один байт.

Как только BIOS найдет загрузочное устройство, она скопирует содержимое первого сектора в оперативную память по смещению 0x7C00, а затем переведет испольнение на этот адрес и начнет испольнение того кода, который только что загрузила. Этот код всё ещё не является операционной системой. А является тем самым загрузчиком. С этого момента начинается этап *загрузки* операционной системы.

Стоит отметить, что вне зависимости от разрядности центрального процессора, загрузчик работает в 16-bit режиме, что означает все адреса, переменные используемые загрузчиком имеют размерность 4 байта. Этот режим называется *16-bit реальным режимом.* После загрузки, ядро операционной системы переключается на *32-bit или 64-bit защищенный режим* в зависимости от операционной системы. Эти и другие особенности загрузки рассмотри в последующих главах. Кроме этого, напишем свой загрузчик операционной системы, что дает более глубокое представление о принципах его работы.

2.1 Программирование в 16-bit реальном режиме. Взаимодействие с дисковым накопителем

Как в других областях программирования, разработчикам ЦП необходимо обеспечивать обратную совместимость своих продуктов, что означает возможность запуска старых программ на более новых процессорах. Вопрос об обратной совместимости возник сразу при разработке 32-bit процессоров. Одно из решений этой проблемы, предложенное и реализованное компанией Intel, состоит в том, чтобы в новых 32-bit процессорах эмулировать работу старых 16-bit процессоров. А именно, семейство процессоров *Intel 8086,* которые поддерживают инструкции 16-bit процессоров и работают в незащищенном режиме.

Защищенный режим (memory protection mode) является ключевым понятием для современных операционных систем, так как позволяет операционной системе ограничивать пользовательский процесс от доступа, скажем, к памяти ядра, который случайно или преднамеренно обойти механизмы безопасности или даже повредить работе всей системы.

Таким образом, для обратной совместимости, необходимо, чтобы процессоры загружались в 16-bit реальном режиме (*16-bit real mode*) и потом явно переключались в 32-bit защищенный режим. Это позволяет старым операционным системам работать на новых процессорах.

Поскольку все современные операционные системы начинают свою работу с 16-bit реального режима, подробно рассмотрим сначала этот режим и далее более подробно рассмотрим переход с 16-bit реального режима в 32-bit защищенный режим.

Рассмотрим следующий код бесконечного цикла на ассемблере `nasm`:

```
jmp $                   ; бесконечный цикл

times 510-($-$$) db 0   ; заполняем нулями 510 байтов минус размер предыдущего
                        ; кода
dw 0xaa55               ; магическое число для определение загрузочного сектора
```

Листинг 2.1 Минимальный пример кода загрузочного сектора с бесконечным циклом

Чтобы скомпилировать данный код в бинарный файл загрузки нужно сохранить в файл, например с названием `boot_sector_simple.asm`, набрать в терминале следующую команду

```
nasm -f bin boot_sector_simple.asm -o boot_sector_simple.bin
```

В результате выполнения этой команды сгенерируется бинарный файл `boot_sector_simple.bin`, что является минимальным кодом для загрузчика. При запуске этого кода на QEMU через команду

```
qemu-system-x86_64 boot_sector_simple.bin
```

мы ничего кроме записи «`Booting from Hard Disk...`» не увидим. Как и ожидалось.
При просмотре бинарного файла увидим уже знакомый нам результат: файл, последние два байта которого равны числу 0xAA55.

Поскольку бесконечный цикл неинтересен, то попробуем вывести на экран сообщение. Рассмотрим следующий листинг кода из файла `print.asm`.

```
print:
       pusha            ; помещаем в стек значения всех 16-битных регистров
                        ; общего  назначения
  start:
       mov al, [bx]     ; ‘0bx’ указатель на строку
       cmp al, 0        ; если в значение ‘al’ равно 0
       je done          ; то переходим в done.

       mov ah, 0x0e     ; запись сивола в режима телетайпа
       int 0x10         ; вызоваем прерывание BIOS 0x10
                        ; ‘al’ already contains the char

       add bx, 1        ; инкрементируем указатель ‘bx’
       jmp start        ; переходим в start

  done:
       popa ; освобождаем из стека значения всех 16-битных регистров общего назначения
       ret  ; возвращаем управление


println:
       pusha

       mov ah, 0x0e     ; запись символа в режиме телетайпа
       mov al, 0x0a     ; символ перехода на новую строку
       int 0x10
       mov al, 0x0d     ; возврат каретки
       int 0x10

       popa
       ret
```

Листинг 2.2 Файл `print.asm`. Функции вывода на экран через 0x10 прерывания BIOS

Как нетрудно догадаться, в коде определены две функции вывода: `print` -- для сообщения на экран без перехода на новую строку; `println` -- для вывода сообщения

с переходом на новую строку. Как видно из листинга, для вывода символа на экран BIOS предоставляет прерывание 0x10. Данное прерывание на вход принимает параметры через регистры `al, ah, bh, bl`. BIOS, как правило, связывает с этим вектором обработчик прерывания в реальном режиме, предоставляющий видеосервис. Он включает установку видеорежима видеоадаптера, вывод символов и строк, графические примитивы.

Регистр `ah` указывает режим работы видеоадаптера, в нашем случае выбран режим записи телетайпа: символ отображается в текущей позиции курсор, после чего курсор сдвигается вправо на одну позицию. При необходимости курсор автоматически перемещается на новую строк, а когда весь экран заполняется, происходит вартикальная свертка экрана.

Регистр `al` хранит ASCII-код записываемого символа, `bh` -- номер страницы видеопамяти в текстовых режимах, `bl` -- цвет символа в графическом режиме.

```
[org 0x7c00]

MESSAGE:                    ; метка к начале строки
    db 'Hello, World!', 0   ; константная строка с нулевым символом

mov bx, MESSAGE             ; записываем адрес начала строки

call print                  ; вызываем функцию print
call println                ; вызываем функцию println


%include "print.asm"        ; подключение файла print.asm

jmp $                       ; бесконечный цикл

times 510-($-$$) db 0       ; заполняем нулями пустые области памяти
dw 0xaa55                   ; записываем в последние два байта значения 0xaa55
```

Листинг 2.3 Файл `boot.asm`. Пример вывода сообщения на экран через функции из фала `print.asm`

В листинге 2.3 приведен пример использования функций `print` и `println` для вывода сообщения “Hello, World!” на экран. Скомпилировать и запустить можно файл через следующие команды:

```
nasm -f bin boot.asm -o boot.bin
qemu-system-i386 boot.bin
```

Результат запуска файла загрузки `boot.bin` на QEMU показан на рисунке 2.3.

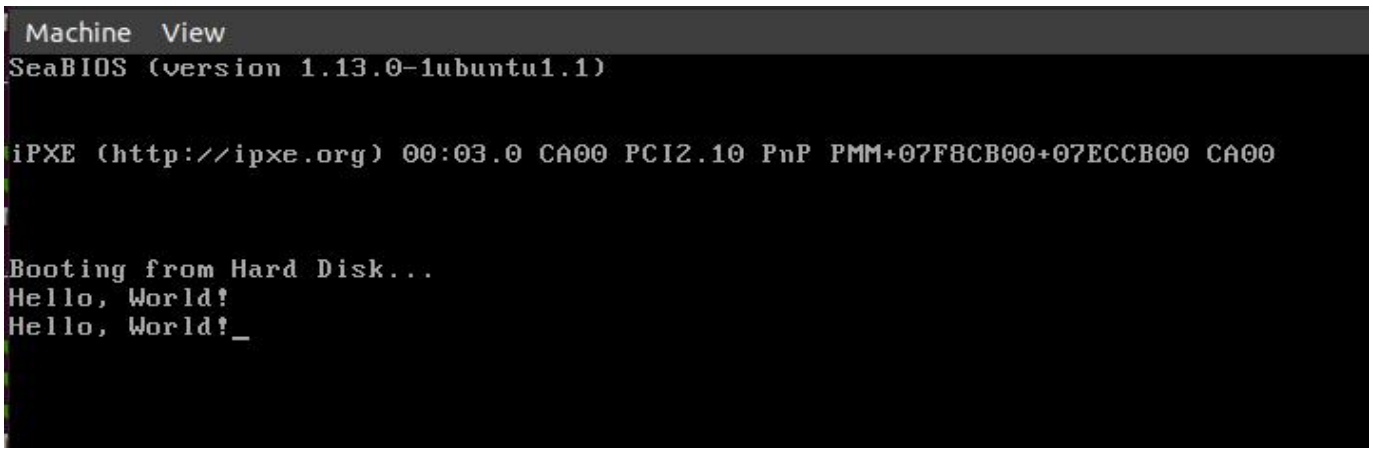

Рисунок 2.3 Результат работы файла `boot.bin`

Поскольку код современных операционных систем имеют размер намного больше чем размер загрузочного сектора, для дальнейшей полноценной работы нам требуется научиться взаимодействовать с дисковыми накопителями. Если заглянуть в «зоопарк» дисковых накопителей, увидим, различные устройства требуют различные и специфичные только для них, взаимодействия с ними. Например, чтобы использовать floppy диски, пользователю нужно явным образом включить и отключить двигатель, который вращает диск подголовкой чтения и записи, в то время другие дисковые накопители имеют автоматизированных функций такого рода.

Кроме того, существуют множество технологий шин, через которых дисковые накопители подключаются к ЦП. Например, ATA/IDE, SATA, SCSI, USB и т.п. Все эти шины имеют специфичный интерфейс, что влияет на взаимодействие с ними. К счастью, BIOS предоставляет несколько дисковых подпрограмм, которые абстрагируют все различия между интерфейсами для конечного пользователя. Подпрограмму BIOS для чтения можно вызвать через прерывание под номером 0x13, предварительно записав в регистр `al` значение `0x02`. Она принимает на вход некоторые другие заранее установленные регистры, которые в свою очередь указывают с какие устройством взаимодействовать, блоки которые необходимо считывать и адрес в оперативной памяти куда сохраняются эти блоки.

Для лучшего понимание рассмотрим приведенный ниже листинг 2.3.

```
; загрузка количество секторов dh с диска dl в es:bx
disk_load:
    pusha
    push dx       ; записываем значение dx в стек для дальнейшего использования

    mov ah, 0x02  ; записываем в ah номер метода чтения 0x02 = 'read'
    mov al, dh    ; записываем в al номер сектора для чтения
    mov ch, 0x00  ; номер цилиндра
    mov dh, 0x00  ; записиваем в dh номер головки
    mov cl, 0x02  ; начинаем считывать со второго сектора,
                  ; т.е. сразу после загрузочного сектора

    int 0x13      ; вызываем прерывание 0x13 BIOS
    jc disk_error ; при возникновении ошибки переходим в disk_error

    pop dx        ; удаляем из стека dx
```

```
    cmp al, dh    ; если номер фактически считанного сектора не
                  ; равен номеру ожидаемого сектора
    jne sectors_error  ; переходим в sectors_error
    popa          ; очищаем стек
    ret           ; возвращаем управление

disk_error:
    mov bx, DISK_ERROR
    call print
    call println
    mov dh, ah
    call println
    jmp disk_loop

sectors_error:
    mov bx, SECTORS_ERROR
    call print

disk_loop:
    jmp $     ; бесконечный цикл

DISK_ERROR: db "Disk read error", 0
SECTORS_ERROR: db "Incorrect number of sectors read", 0
```

Листинг 2.3 Файл `disk_load.asm`. Процедура загрузки указанного сектора из данного диска

На листинге 2.3 приведен пример реализации процедуры `disk_load` чтения указанного сектора с диска. Данная процедура принимает на вход регистры `dh` -- номер сектора, `dl` -- номер диска, `es:bx` -- область в оперативной памяти для сохранения данных сектора. Как отметили выше, перед вызовом прерывания 0x13 требуется указать в `al` -- номер сектора для чтения, `ch` -- номер цилиндра, `dh` -- номер головки, `cl` -- начало считывания. Нетрудно догадаться, выходом подпрограммы 0x02 прерывания 0x13 служит регистр `al`, который записывается номер фактически считанного сектора дискового накопителя. Это позволяет проверить результат считывания сектора, что и делается в приведенном выше примере.

Следующий пример на листинге 2.4 показывает загрузку следующих секторов сразу после загрузочного сектора.

```
[org 0x7c00]
    mov bp, 0x8000          ; устанавливаем указатель на начало стека
    mov sp, bp              ; в безопасном месте

    mov bx, 0x9000          ; es:bx = 0x0000:0x9000 = 0x09000
    mov dh, 3               ; считываем 3 следующих сектора

    call disk_load          ; вызываем процедуру загрузки сегментов диска

    mov dx, [0x9000]        ; выводим значение первого загруженного слова
    call printhex           ; процедура вывода слова в hex формате

    call println

    mov dx, [0x9000 + 512]  ; выводим первое слово со второго загруженного сектора
    call printhex

    call println

    mov dx, [0x9000 + 1024] ; выводим первое слово с третьего загруженного сектора
    call printhex
    jmp $                   ; бесконечный цикл

%include "print.asm"
%include "disk_load.asm"

times 510 - ($-$$) db 0
dw 0xaa55                   ; устанавливаем сигнатуру загрузочного сектора

times 256 dw 0xdada         ; сектор 2 = 512 байтов
times 256 dw 0xface         ; сектор 3 = 512 байтов
times 256 dw 0xdaad         ; сектор 4 = 512 байтов
```

Листинг 2.4 Файл `boot.asm` Пример использования процедуры загрузки диска из файла `disk_load.asm`

В приведенном примере загружается три следующих сектора после загрузочного, заранее заполненных значениями 0xdada, 0xface, 0xdaad. Далее выводятся на экран значения первых слов из загруженных секторов, соответсвенно. Следует отметить, что для вывода слов в формате hex используется специально написанная процедура `printlhex`, код которой приведен в приложенных к работе исходных файлах.

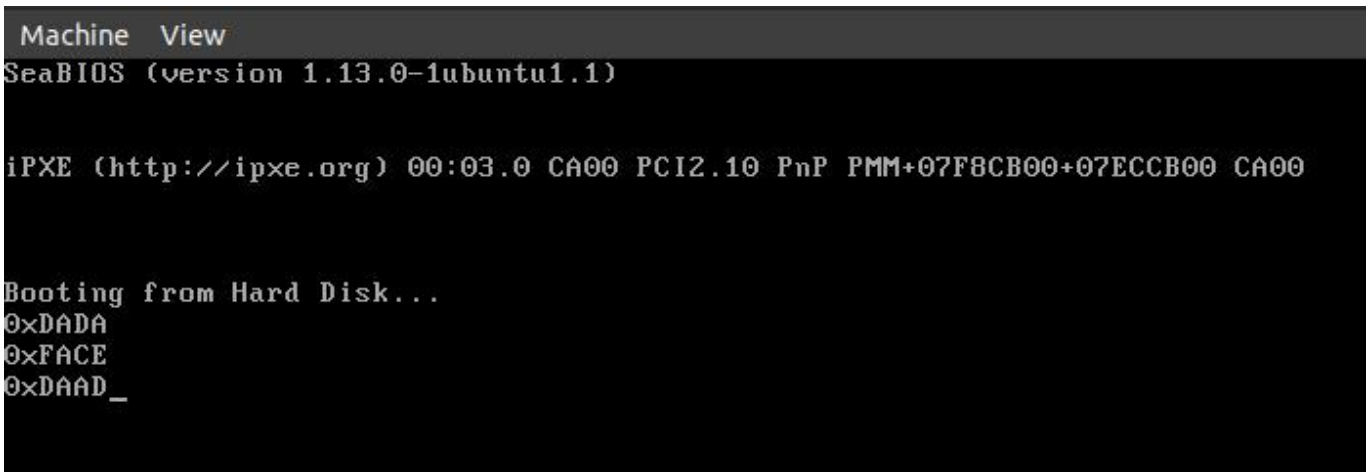

Рисунок 2.4 Пример возможного вывода кода из листинга 2.4

На рисунке 2.4 приведен возможный вывод запуска загрузочного файла `boot.asm`, что показывает успешность загрузки в оперативную память следующих 3 секторов.

Резумируя вышеприведенные примеры, можно отметить, что BIOS, через механизмы прерываний и их подпрограмм, предоставляет достаточно хорошую абстракцию для взаимодействия с устройствами, в том числе и с дисковыми носителями.

2.2 Таблица глобальных дескрипторов GDT

Несмотря на то, что нам хочется продолжать работать в окружении 16-bit реального режима, с котором хорошо знакомы, чтобы использовать все возможности ЦП, нам необходимо переключиться в 32-bit защищенный режим. Наряду с первоначальными неудобствами, данный режим нам дает множество преимуществ.

Одним из основных различий 32-bit защищенного режима от 16-bit реального режима, являются расширенные регистры. Для каждого регистра общего назначения существует его расширенная пара, название которой отличается префиксом `e`. Например, расширенный вариант регистра `bx` является регистр `ebx`. Кроме того, в 32-bit процессорах добавлены дополнительно 2 сегментных регистра: `fs` и `gs`.

Самым важном этапом при переходе 16-bit режима в 31-bit является подготовка *таблицы глобальных дескрипторов.* Поэтому остановимся на этом подробнее.

Как заметим ниже, таблица глобальных дескрипторов (далее GDT) является важной составляющей при работе в 32-bit защищенном режиме. Но, прежде чем перейти на обсуждение GDT, остановимся на адресации ячеек паамяти в реальном режиме.

В наследство от процессоров 8086/88 достался своеобразный способ задания адреса ячейки памяти в виде указателя `seg:offset`, состоящего из двух слов: *сегмента* и *смещения.* Такая запись предполагает вычисление полного адреса по формуле:

```
Address = 16 x seg + offset.
```

Такое представление 20-битного адреса двумя 16-битными числами в процессорах 8086/88 поддерживается и в реальном режиме всех последующих процессоров x86. Здесь сегмент указывает адрес *параграфа* – 16-байтной области памяти. Выравнивание адреса по границе параграфа означает, что он кратен 16. Нетрудно видеть, что один и тот же адрес можно задавать разными сочетаниями этих двух компонентов. Например, адрес начала области данных BIOS (BIOS Data Area) 00400h представляют как 0000:0400, так и 0040:0000.

Хотя общая идея сегментации памяти и использования смещений для доступа к этим сегментам в 32-bit режиме осталась прежней, способ её реализации в защищенном режиме полностью изменился. После переключения ЦП в 32-bit защищенный режим, сегментный регистр становится индексом для определнного дескриптора сегмента в GDT.

Дескриптор сегмента -- это 8-байтовая структура, определяющая следующие свойства сегмента защищенного режима:

1. Base address (32 bits), определяющий откуда сегмент начинается в физической памяти;
2. Segment Limit (20 bits), определяющий размер сегмента;
3. Различные флаги, указывающие уровень привелегий, гранулярность сегмента и т.п.

На рисунке приведен структура сегментного дескриптора

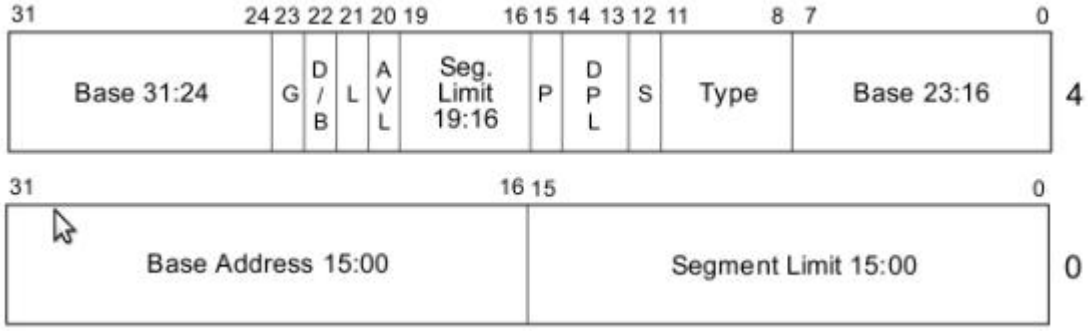

Рисунок 3.1 Структура сегментного дескриптора

где,

L	-- 64-bit сегментный код;
AVL	-- флаг разрешающий использование сегмента пользовательскими программами;
BASE	-- база сегмента;
D/B	-- определяет разрядность сегмента (16 бит или 32 бит)
DPL	-- флаг привелегий;
G	-- гранулярность;
LIMIT	-- размер сегмента;
P	-- определяет доступность сегмента;
S	-- тип дескриптора (0--системный, 1--код или данные)
TYPE	-- тип сегмента.

Как могли уже заметить, что база и лимит сегментного дескриптора распределена по всей структуре, что выглядит довольно запутанно. Как отмечается в руководстве программистов от Intel, это было сделано по той причине, что при определнных сочетаниях единичных флагов, какждая часть базы и лимита воспринимается определенным образом. В дополнение к сегментам кода и данных, ЦП требует, чтобы первая запись в GDT преднамеренно был недопустимым нулевым дескриптором, т.е. структурой из 8 нулевых байтов. Пустой дескриптор выступает как простой механизм обнаружения ошибок. При попытке адресации с нулевым дескриптором, ЦП вызывает исключение, что является ничто иное как обычное прерывание.

На листинге 3.1 приведен пример определения GDT.

```
gdt_start:              ; Эта пустая метка нужна для того чтобы удобнее
                        ; посчитать размер GDT для ее дескриптора
                        ; (end - start)
```

```
gdt_null:               ; Необходимый нулевой дескриптор для GDT
    dd 0x0              ; dd - define double (двойное слово, т.е. 4 байта)
    dd 0x0

gdt_code:               ; Определяем дескриптор сегмента кода
    dw 0xffff           ; Limit (bits 0-15)
    dw 0x0              ; Base (bits 0-15)
    db 0x0              ; Base (bits 16-23)
    db 10011010b        ; Первые флаги + флаги типа (смотрим по битам)
                        ; present: 1, privilege: 00, descriptor type: 1
                        ; code: 1, conforming: 0, readable: 1, accessed: 0
    db 11001111b        ; Вторые флаги + длина сегмента (bits 16-19):
                        ; granularity: 1, 32-bit default: 1,
                        ; 64-bit default: 0, AVL: 0
    db 0x0              ; Base (bits 24-31)

gdt_data:               ; Определяем дескриптор сегмента кода
    dw 0xffff           ; Limit (bits 0-15)
    dw 0x0              ; Base (bits 0-15)
    db 0x0              ; Base (bits 16-23)
    db 10010010b        ; Первые флаги + флаги типа (смотрим по битам)

    db 11001111b        ; Вторые флаги + длина сегмента (bits 16-19)
     db 0x0                 ; Base (bits 24-31)

gdt_end:                ; Пустая метка

gdt_descriptor:                     ; дескриптор GDT
    dw gdt_end - gdt_start - 1      ; Размер GDT
    dd gdt_start                    ; Адрес начала GDT

CODE_SEG equ gdt_code - gdt_start  ; Определяем некоторые константы
DATA_SEG equ gdt_data - gdt_start  ; для дальнейшего использования
```
Листинг 3.1 Файл `gdt.asm`. Пример опередения GDT

2.3 Переход в 32-bit защищенный режим

К этому моменту у нас есть всё, чтобы разобраться с переходом в 32-bit защищенный режим. Первое, что нужно сделать при переходе в защищенный режим -- это отключение прерываний. В NASM это делается через команду `cli (clear interrupt)`. При отключении прерываний ЦП будет игнорировать любое поступающее к нему прерывание. Отключение прерываний необходимо по той причине, что их обработка в 32-bit защищенном режиме принципиально отличается от обработки в 16-bit реальном режиме. Если бы не отключали прерываний, то при их появлении подпрограммы BIOS будут выполнять 16-bit код который имеют представления о 32-bit сегментах и в итоге это привело бы к краху всей системы.

Следующим шагом является загрузка GDT, с которым познакомились в предыдущей главе. Это делается с помощью одной инструкции:

```
lgdt[gdt_descriptor].
```

Остался последний «штрих» перехода в 32-bit защищенный режим. Для этого необходимо переключить первый бит специального регистра контроля `cr0` в 1. Поскольку ЦП не предоставляет механизмы прямого доступа к отдельным битам, воспользуемся оператором `or` (побитового ИЛИ):

```
mov eax, cr0
mov eax, 0x1
mov cr0, eax
```

После переключения первого бита регистра `cr0`, ЦП входит в 32-bit защищенный режим.

Последнее утверждение является не совсем верным. Дело в том, что современные ЦП используют технологию конвейерной обработки, что позволяет им параллельно обработать различные этапы выполнения инструкций. На первый взгляд может показаться, что такая обработка инструкций является небезопасной, поскольку некоторые команды запущенные в 16-bit реальном режиме, могут выполняться уде в 32-bit защищенном режиме. Но, в ЦП существует специальный механизм для защиты от таких ситуаций. Единственное что необходимо сделать -- это ожидать выполнения команд до переключения регистра `cr0.`

Для ожидания выполнения инструкций в конвейере необходимо выполнить *дальний прыжок* с помощью команды `jmp,` что заставляет ЦП ожидать выполнения всех задач в конвейере. *Дальний прыжок* -- это переход в дальний сегмент. Для этого используется метка определенная в GDT:

```
jmp CODE_SEG:init_pm

[bits 32]

init_pm:
    ...
    ...
```

Директива `[bits 32]` указывает ассемблеру, что с этой точки команды нужно компилировать в 32-bir режиме. Следует отметить, это не означает, что далее нельзя использовать команды 16-bit режима, а значит лишь то, что ассемблер дальше будет скомпоновать код несколько иначе чем в 16-bit режиме.

Приведем весь код переключения в защищенный режим как итог данной главы.

```
[bits 16]

switch_to_pm:
    cli                         ; Отключаем прерывания (cli = clear interrupts)

    lgdt [gdt_descriptor]       ; Загружаем GDT дескриптор (lgdt = load GDT)

    mov eax, cr0                ; Чтобы перейти в PM, нужно чтобы первый бит
    or eax, 0x1                 ; регистра управления cr0 был 1
    mov cr0, eax
```

```
    jmp CODE_SEG:init_pm      ; Делаем "дальний прыжок" в 32-битный
                              ; сегмент кода. Это так же заставляет процессор
                              ; завершить обрабатываемые в конвейере инструкции.

[bits 32]

init_pm:                      ; в PM, наши старые сегменты бесполезны, поэтому
    mov ax, DATA_SEG          ; мы делаем так, чтобы регистры всех сегментов
    mov ds, ax                ; указывали на сегмент данных, который мы определили
    mov ss, ax                ; в GDT (см. ./gdt.asm)
    mov es, ax
    mov fs, ax
    mov gs, ax

    mov ebp, 0x90000          ; Обновляем позицию стека, чтобы он был на самом
    mov esp, ebp              ; верху свободного места
```
Листинг 3.2 Файл `switch.asm`. Пример переключения в защищенный режим

3 Разработка ядра ОС

Основное различие между операционной системой и ядром состоит в том, что операционная система --- это системная программа, которая управляет ресурсами системы, а ядро -- это важная часть (программа) в операционной системе.

Стоит отметить, что само ядро --- это не процесс, а диспетчер процессов. Модель процесс / ядро предполагает, что процессы, которым требуется служба ядра, используют определенные программные конструкции, называемые системными вызовами.

2.1 Вызов функции на языке С

Поскольку разарботка ядра представляет собой сложный процесс, то целесообразном является перейти на высокоуревной язык такой как С. Поэтому перед тем как приступим к реализации ядра, необходимо разобраться с вызовом функции написанной на языке С из кода ассемблера.

Рассмотрим следующий листинг:

```
    [bits 32]
    [extern main]  ; Определяем 'внешний' символ main - она понадобится
                   ; линкеру чтобы собрать все вместе

    call main      ; Вызываем определенную выше функцию, которая будет доступна
                   ; после линковки. Это функция main из core.c
    jmp $
```
Листинг 4.1 Файл `launch.asm` Пример перехода на точку входа ядра

Как видим из листинга 4.1, переход на ядро выполняется через команду `call`. Стоит отметить, что во время компиляции файла фукнция процедура main неопределена. В этой связи, до вызова функции, она определяется внешним символом через ключевое слово `extern`. Она позволяет ассемблеру считать, что

символ будет определен в другом файле и во время компановки линкер имеет возможность связать вызов с точкой определения функции `main`.

Следует скопилировать данный файл через следующую команду

```
nasm launch.asm -f elf -o launch.o
```

Опция `-f elf` указывает ассемблеру, что результирующий файл является частью модуля с `elf` форматом, что является значением по умолчанию у компилятора `gcc`.

Напишем простое ядро, на C которое только выведет на экран символ ‘A’.

```
void main () {
      char * video_memory = ( char *) 0 xb8000 ;
      *video_memory = ‘A’;
}
```

Листинг 4.2 Файл core.c.

Для компиляции кода и создания бинарного кода для линковки с `launch.o` нам понадобится набрать следующие команды:

```
gcc -ffreestanding -c core.c -o core.o
ld -o core.bin -Ttext 0x1000 launch.o core.o --oformat binary
```

Чтобы запустить полученный код нам необходимо скомпоновать наше ядро с зарузчиком. Для простоты записываем всё содержимое ядра записываем сразу после кода загрузчика:

```
cat boot.bin core.bin > yaos
```

Запустив образ, убедимся что наше ядро выводит символ ‘A’ на экран.

Рисунок 4.1 Вывод символа на экран

3.2 Реализация ввода-вывода

Логическим продолжением разработки является реализация операции низкуровневого ввода-вывода. На уровне ядра обычно используется один из двух способов взаимодействия с устройствами: через отображения в ОЗУ *(memory-mapped i/o)* и через порты *(port-mapped i/o)*.

Ввод-вывод с отображением в ОЗУ -- это способ организации взаимодействия процессора с внешним устройством, при котором регистры ввода-вывода устройства представляются для программиста обычными ячейками оперативной пам яти с фиксированными адресами, которые жёстко закреплены за каждым устройство м. Не требует наличия специальных команд ввода-вывода - чтение из такой ячейки с оответствует вводу, а запись в неё - выводу данных из ЦП, при этом данные без лиш -них пересылок могут обрабатываться ЦП непосредственно в ячейке. Недостатком м етода является необходимость резервирования части адресного пространства памяти под регистры ввода-вывода и обеспечение возможности прямого доступа контролле ра устройства к шине памяти.

Ввод-вывод с отображением портов используется специальным набором инструкций ЦП. Поддерживающие ввод-вывод с отображением портов устройства имеют отдельное адресное пространство от общей памяти, что обеспечивается либо допольнительным выводом на физическом интерфейсе ЦП, либо всей шиной, выделенной для чтения и записи.

Для получения доступа к регистрам воспользуемся таким механизмом как *встроенное ассемблирование.* Язык С позволяет вставлять ассемблерные выражения прямо в код. Но, поскольку компилятор С работает только с ассемблером GAS, на этом этапе нам придется его использовать. Для взаимодействия с устройствами GAS представляет следующие инструкции:

вывод: `in <port address>, <destination>` ;
ввод: `out <destination>, <source>`

Рассмотрим следующий пример на языке С:

```
unsigned char port_byte_in(unsigned short port) {
   unsigned char result;
    __asm__("in %%dx, %%al" : "=a"(result) : "d"(port));
   return (result);
}

void port_byte_out(unsigned short port, unsigned char data) {
  __asm__("out %%al, %%dx" : : "a"(data), "d"(port));
}
```

Листинг 4.3 Пример низкоуровневого чтения и записи на языке С

Как видим, для вставки ассемблерных выражений на С используется выражение `__asm__`, в котором, запись “`=a”(result)` означает записать значение регистра `al`, а “`d”(port)` -- загрузить значение port в регистр `dx`.

Для польноценного взаимодействия с монитором нам необходимо получить и устанавливать позицию курсора. На листинге 4.4 приведен пример, в котором для получения и установки позиции курсора используютя функции из листинга 4.3. В примере макросы `REG_SCREEN_CTRL`, `REG_SCREEN_DATA` определяют порты для взаимдействия с экраном.

```
uint16_t get_cursor() {
  port_byte_out(REG_SCREEN_CTRL, 14);
```

```
  uint8_t high_byte = port_byte_in(REG_SCREEN_DATA);
  port_byte_out(REG_SCREEN_CTRL, 15);
  uint8_t low_byte = port_byte_in(REG_SCREEN_DATA);
  return (((high_byte << 8) + low_byte) * 2);
}

void set_cursor(uint16_t pos) {
  pos /= 2;
  port_byte_out(REG_SCREEN_CTRL, 14);
  port_byte_out(REG_SCREEN_DATA, (uint8_t)(pos >> 8));
  port_byte_out(REG_SCREEN_CTRL, 15);
  port_byte_out(REG_SCREEN_DATA, (uint8_t)(pos & 0xff));
}
```

Листинг 4.4. Функции для работы с курсором

Далее используя вышеприведынные функции можно легко реализовать высокоуровневую функцию `print`, которая печатает строку на экран.

ЗАКЛЮЧЕНИЕ

В рамках данной работы, были изучены методы и инструменты разработки операционных систем. Описанные в ходе работы технологии разработки были выбраны с учетом дальнейшей автоматизации сборки и отладки системы.

В ходе дальнейшей исследовательской разработки были выделены основные этапы и работы системы BIOS. До загрузки загрузочного сектора в оперативную память BIOS запускает процедуру POST для обнаружения и проверки необходимых устройств. После успешного окончания тестирования загружается в оперативную память первый найденный загрузочный сектор. BIOS накладывает на загрузочный сектор единственное условие: последнее слово загрузочного сектора должно содержать значение 0xAA55.

В начале работы загрузчика, в целях обратной совместимости, ЦП процессор работает в 16-bit реальном режиме. Чтобы использовать все возможности ЦП при поддержки разрядности выше 16-bit, до загрузки ядра в оперативную память, помимо отключения прерываний BIOS, загрузчику необходимо переключить ЦП в защищенный режим. В этой связи, после загрузки таблицы GDT, загрузчик переключает первый бит регистра `cr0`.

Для разработки ядра операционных систем, в основном, используются верхнеуровневые языки программирования как C, C++, Rust и т.п. Поскольку язык C предоставляет достаточно близкие механизмы к ассемблеру, при демонстративной разработке ядра было решено использовать его, в роли переходного языка.

Стоит отметить, что операционная система работает при защищщенном режиме ЦП и не имеет доступа к прерыванием BIOS для обращения к устройствам. В рамках работы, как функционал ядра, были разработаны низкоуровновое взаимодействия с устройствами через механизм отображения портов. Для демонстрации результатов продемнстрирована печать строки на манитор.

Разработка основных функционалов, как многозадачность, менеджер задач, файловая система, требует больших временных ресурсов. В этой связи, далее будет целосообразным переходить ещё более высокоуревновый язык программирования. Поэтому, как направление дальнейшего развития проекта, переход на язык программирования Rust позволит использовать современные механизмы управления памятью уже во время компиляции. Это позволяет, в свою очередь, позволяет в разы ускорить дальнейшую разработку.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Таненбаум, Э. Бос, Х. – Современные операционные системы. 4-е изд. –СПб.: Питер 2021. –1120 с.: ил.
2. Таненбаум, Э. Вудхалл, А. – Операционные системы Разарботка и реализцаия. Классика CS. 3-е изд. –СПб.: Питер 2007. –704 с.: ил.
3. Столяров, А.В. – Программирование на языке ассемлебера для ОС UNIX. Уч. пособие. – 2-е изд. – М.: МАКС Пресс. 2011. – 188 с.: ил.
4. Real Mode // Статья о реальном режиме ЦП [Электронный ресурс] – Режим доступа: https://wiki.osdev.org/Real_Mode. – Дата доступа: 20.09.2021
5. Protected Mode // Статья о защищенном режиме ЦП [Электронный ресурс] – Режим доступа: https://wiki.osdev.org/Protected_Mode. – Дата доступа: 23.11.2021
6. Global Descriptor Table // Статья о таблице GDT [электронный ресурс] – Режим доступа: https://wiki.osdev.org/Global_Descriptor_Table. – Дата доступа: 12.11.2021