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

Специальность *информатика и технологии программирования*

Ашыров Алишер,

2 курс, магистрант

Научный руководитель:

Трубников Юрий Валентинович,

профессор, доктор физико-матема- тических наук

Витебск, 2022

РЕФЕРАТ

Магистерская диссертация СТРАНИЦЫ с., РИСУНКИ рис., ТАБЛИЦЫ табл., ИСТОЧНИКИ источников, ПРИЛОЖЕНИЯ прил.

ОПЕРАЦИОННАЯ СИСТЕМА, ЗАГРУЗЧИК, BIOS, ПРЕРЫВАНИЕ, РЕАЛЬНЫЙ РЕЖИМ, ЗАЩИЩЕННЫЙ РЕЖИМ.

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

Предмет исследования – принципы разработки операционных систем

Цель работы – НАПИСАТЬ ЦЕЛЬ

Методы исследования: описательно–аналитический, сравнительно–сопоставительный, статистический.

Элементы новизны:

Результаты внедрения: ????

Теоретическая и практическая значимость:

**СОДЕРЖАНИЕ**

ПЕРЕЧЕНЬ СОКРАЩЕНИЕ, УСЛОВНЫХ ОБОЗНАЧЕНИЙ, СИМВОЛОВ И ТЕРМИНОВ.........................................................................................................................4

ВВЕДЕНИЕ..............................................................................................................5

ОСНОВНАЯ ЧАСТЬ...............................................................................................6

1 Инструментарий разработки операционных систем.........................................6

1.1 Система сборки............................................................................................6

1.2 Эмулятор устройств QEMU........................................................................?

2 Программирование загрузчика ОС.....................................................................?

* 1. 16-bit реальный режим...............................................................................?
  2. Переход с 16-bit реального режима в 32-bit (64-bit) защищенный режим........................................................................................................................?
  3. Вызов функции с кода на языке С............................................................?

3 Разработка ядра ОС..............................................................................................?

3.1

3.2

3.3

ЗАКЛЮЧЕНИЕ.......................................................................................................?

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ.............................................?

ПРИЛОЖЕНИЕ.......................................................................................................?

**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;

**ВВЕДЕНИЕ**

**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) c 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>.

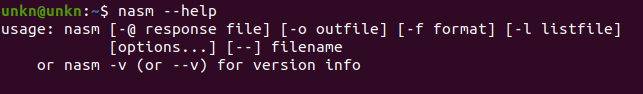


Рисунок 1.1 Часть справочного вывода команды nasm

Поскольку ядро операционной системы будем разарабатывать на высокоуровневом языке С, 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 OC 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.

**2 Программирование загрузки 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 минимально необходимых 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 играть роли и в дальнейщем будет использоваться обозначение BIOS.

После низкоуровневого тестирования системы, BIOS запускает проверку дисков. Поскольку на этом этапе не поддерживается файловая система, для запуска ОС BIOS проверяет первые сектора всех дисков на наличие «магических» байтов, которые называются *загрузочными секторами*. А конкретно, если байты 511 и 512 в первом секторе равны 0xAA55, то диск считается загрузочным.

На рисунке 1.1 приведен пример бинарного кода загрузочного сектора. Как видно из рисунка, действительно этот сектор заканчивается на 0xAA55. Стоит отметить, что порядок байтов AA и 55 на рисунке с записью 0xAA55. Такая разница связана с тем, что архитектура x86 обрабатывает данные в little-endian формате представления бинарных файлов.

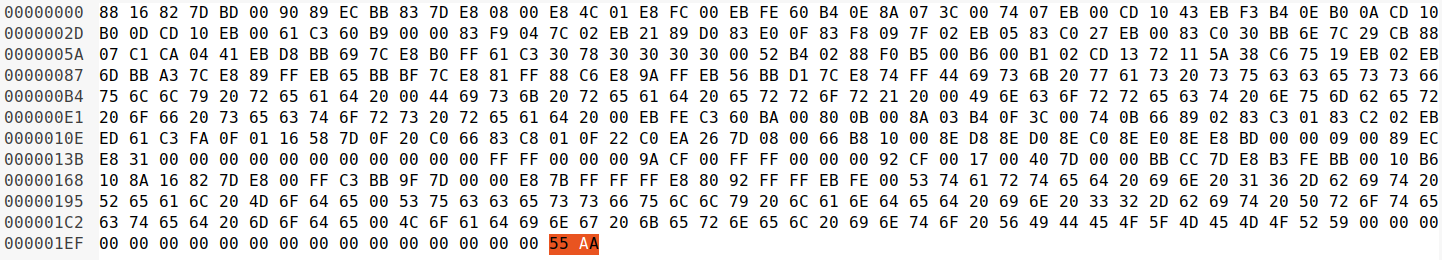


Рисунок 2.1 Пример бинарного кода загрузочного сектора. Каждому шестнадцатеричному числу соответствует один байт.

Как только BIOS найдет загрузочное устройство, она скопирует содержимое первого сектора в оперативную память. Начиная с адреса 0x7C00, а затем переведет испольнение на этот адрес и начнет испольнение того кода, который только что загрузила. С этого момента начинается этап *загрузки* операционной системы.

**2.1 Программирование в 16-bit реальном режиме**

Как в других областях программирования, разработчикам процессоров (CPU) 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 переключение с 16-bit реального режима в 32-bit защищенный режим.

Рассмотрим следующий код бесконечного цикла на ассемблере nasm:

|  |
| --- |
| loop:  jmp loop ; бесконечный цикл (e9 fd ff)  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 компилировать 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 равны числу 0xaa55.

Поскольку бесконечный цикл никому 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 кода, для вывода символа на экран BIOS предоставляет прерывание 0x10. Данное прерывание на вход принимает параметры через регистры al, ah, bh, bl.

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 main.asm -o boot.bin  qemu-system-i386 boot.bin |

Результат запуска файла boot.bin на QEMU показан на следующем рисунке

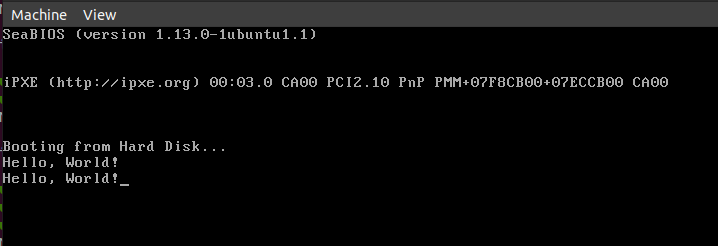


Рисунок 2.2 Результат работы файла boot.bin