Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ

КАФЕДРА ТЕОРЕТИЧЕСКАЯ ИНФОРМАТИКА И КОМПЬЮТЕРНЫЕ ТЕХНОЛОГИИ

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К КУРСОВОЙ РАБОТЕ ПО КУРСУ
Конструирование компиляторов

НА ТЕМУ:

Портирование компилятора BTPC64
на операционную систему macOS

Студент ИУ9-72Б _____ А. А. Мамаев
(Группа) (Подпись, дата) (И.О.Фамилия)

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

Москва 2021 г.

Содержание

Введение

Самоприменимый компилятор BeRo TinyPascal, разработанный для платформы Windows 32-битной разрядности [1], а также его портированная версия BTPC64 для платформы Linux 64-битной разрядности [2] используются на кафедре ИУ9 для проведения лабораторных работ в рамках курса «Конструирование компиляторов» по изучению раскрутки (bootstrapping) и самоприменения компилятора.

Студентам же, использующим операционную систему macOS, при выполнении данной лабораторной работы приходится пользоваться нестандартными методами разработки, такими как использование виртуальной операционной системы или удаленного сервера с нужной ОС.

В рамках выполнения данной курсовой работы требуется портировать существующий компилятор BTPC64 на платформу macOS 64-битной разрядности, сохранив при этом самоприменимость компилятора и основные принципы его архитектуры.

1. Обзор предметной области

1.1. Форматы исполняемых файлов

Принципиально важную роль в данном проекте играют форматы исполняемых файлов исходной и целевой платформ, поэтому прежде всего рассмотрим устройство исполняемых файлов.

Исполняемый файл (Executable file) представляет из себя программу в таком виде, в котором она может быть загружена в память и после этого выполнена непосредственно CPU. Перед исполнением могут быть также выполнены некоторые подготовительные операции, например, загрузка динамических библиотек и настройка окружения.

Внутри файла данные хранятся в определенном формате, который разделяет их на несколько составляющих частей. Общими для большинства форматов частями являются заголовки, таблицы с информацией о смещениях и адресах, метаданные и инструкции.

Заголовки представляют из себя структуры, определяющие, что находится в определенной части файла, как данные из этой части должны быть интерпретированы, как они должны быть исполнены. В заголовках могут быть указаны параметры окружения, исполнители инструкций, настройки этих исполнителей, а также формат инструкций и данных. Исполнителями в данном случае могут являться конкретные процессоры конкретной архитектуры, микроконтроллеры, интерпретаторы как программные, так и аппаратные, а также виртуальные машины. В рамках данной работы целевыми исполнителями будут процессоры архитектуры x86-64.

Часть исполняемого файла с инструкциями может содержать как машинные инструкции, так и исходный код или байт-код виртуальной машины.

Помимо указанных, исполняемые файлы могут содержать и другие составляющие, такие как данные для отладки, константы, описание окружения, иконки ярлыков и любые другие данные, установленные форматом.

Наиболее распространенными исполняемыми файлами к настоящему моменту являются PE (Microsoft Windows), ELF (Unix-подобные ОС), Mach-O (macOS и iOS).

Исходным форматом в рамках данной работы является ELF, целевым – Mach-O. Далее они будут рассмотрены подробнее, так как значительная часть работы связана с обеспечением исходной архитектуры BeRo TinyPascal и BTPC64 в рамках формата Mach-O.

1.1.1. Формат ELF

ELF (Executable and Linkable Format) – формат исполняемых файлов во многих Unix-подобных системах, таких как Linux, FreeBSD, Solaris.

Файл формата ELF состоит из заголовка файла, таблицы программных заголовков, секций и таблицы заголовков секций в конце файла. Схематично устройство формата ELF приведено на рисунке 1.

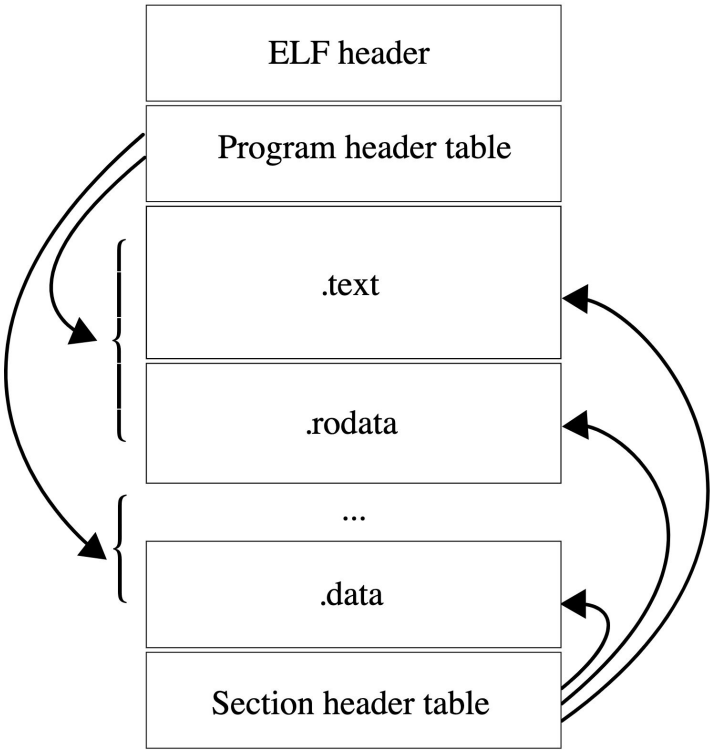

Рисунок 1 – Схема устройства ELF

Заголовок файла (ELF header) расположен в самом начале файла и содержит общее описание структуры файла и его основные характеристики, такие как тип, версия, архитектура, точка входа.

Таблица заголовков программы (Program header table), также известная как таблица заголовков сегментов (Segment table) следует за заголовком

файла. Каждый элемент этой таблицы содержит в себе тип сегмента, расположение сегмента, точку входа, размер и некоторые другие параметры [3].

Сегменты необходимы для загрузки файла в память и исполнения. Только данные, загруженные в сегменты, необходимы для работы с файлом. Остальные данные с позиции загрузчика – отладочная информация либо метаданные.

Далее следуют непосредственно секции. Секции необходимы при линковке (компоновке) файла. Одному выгружаемому сегменту могут соответствовать несколько секций, при этом некоторые секции могут не отображаться в сегменты вообще.

За секциями следует таблица заголовков секций (Section header table). Таблица заголовков секций используется только при линковке файла и содержит атрибуты секций, использующиеся для оптимального размещения секций по сегментам.

Опишем также процесс загрузки ELF-файла:

1. Считывание и валидация заголовков.
2. Выделение виртуальной памяти под приложение и попытка выгрузить его по предпочтительному адресу.
3. Вычисление адреса и размера секций в виртуальной памяти, вычисление смещений и установка атрибутов страниц согласно атрибутам секций, обнуление секций до нужной длины.
4. Анализ таблицы импортов и загрузка соответствующих библиотек, связывание импортируемых симоволов.

1.1.2. Формат Mach-O

Mach-O (Mach object) – формат исполняемых файлов в операционных системах корпорации Apple, а также в некоторых других, основанных на ядре Mach. В настоящее время наиболее распространенные операционные системы, использующие формат Mach-O – Mac OS X (macOS) и iOS.

Интересной особенностью формата является полное отсутствие официальной документации: соответствующая страница на сайте developer.apple.com была удалена, однако можно найти архивированную версию данной страницы [4]. Исчерпывающая информация о внутреннем устройстве формата может быть найдена в исходных файлах загрузчика ядра

XNU [5], а также, например, в статьях разработчика утилиты htool, использующей собственный парсер файлов формата Mach-O [6].

Файл формата Mach-O состоит из заголовка файла, списка загрузочных команд (load commands), отступа (padding), сегментов и секций, таблиц идентификаторов и строк.

Схематичное устройство файлов формата Mach-O приведено на рисунке 2. Несмотря на кажущуюся громоздкость и сложность, устройство файлов Mach-O во многом похоже на устройство ELF и в некотором смысле проще и прямолинейнее.

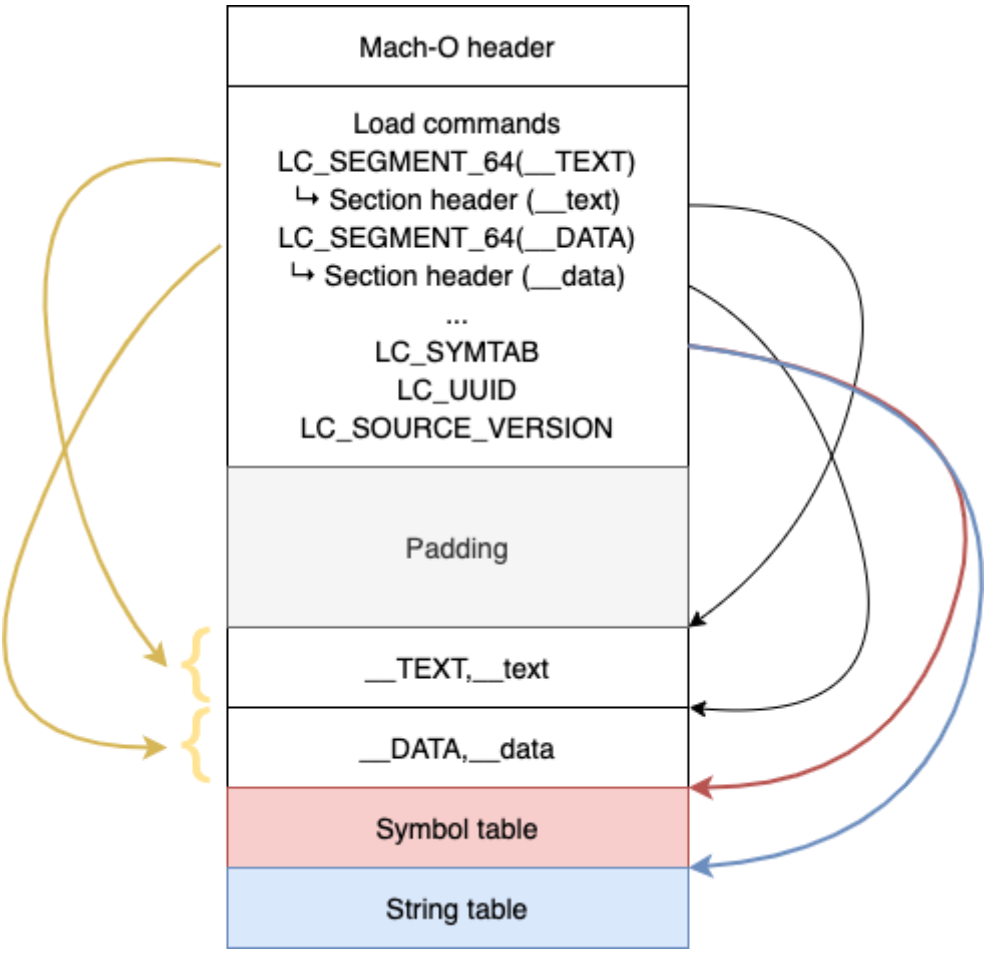

Рисунок 2 – Схема устройства Mach-O

Как и в ELF-файле, заголовок файла (Mach-O header) расположен в самом начале файла и содержит основные его характеристики, такие как тип файла, архитектура платформы, «магическая» константа, а также размер блока загрузочных команд и количество загрузочных команд.

Загрузочные команды (Load commands) – общий список структур, описывающих все составляющие загрузочного файла. Практически во всех Mach-O файлах присутствуют загрузочные команды сегментов

(LC_SEGMENT_64 в случае 64-битной платформы) и таблицы символов и строк (LC_SYMTAB), а также тривиальные загрузочные команды LC_UUID и LC_SOURCE_VERSION, включающие соответственно уникальный идентификатор исполняемого файла и его версию.

Загрузочные команды сегментов включают имя сегмента, предпочтительный адрес, размер и смещение в загрузочном файле, а также список заголовков секций, которые быть загружены в данный сегмент. Принципиально, что отображение секций в сегменты в Mach-O в общем случае задается в исполняемом файле, а при компиляции ассемблерного кода – в самом ассемблерном коде. Заголовки секций содержат также предпочтительный адрес, размер и смещение относительно начала файла, а также другую метаинформацию. Определения структур загрузочной команды сегмента и заголовка секции приведены на листинге 1.

Листинг 1 – определения структур segment_command_64 и section_64

```
struct segment_command_64 {
    uint32_t   cmd;           /* LC_SEGMENT_64 */
    uint32_t   cmdsize;
    char       segname[16];   /* segment name */
    uint64_t   vmaddr;         /* memory address of this segment
*/
    uint64_t   vmsize;        /* memory size of this segment */
    uint64_t   fileoff;       /* file offset of this segment */
    uint64_t   filesize;      /* amount to map from the file */
    vm_prot_t  maxprot;       /* maximum VM protection */
    vm_prot_t  initprot;      /* initial VM protection */
    uint32_t   nsects;         /* number of sections in segment
*/
    uint32_t   flags;         /* flags */
};

struct section_64 {
    char       sectname[16];  /* name of this section */
    char       segname[16];   /* segment this section goes in */
    uint64_t   addr;           /* memory address of this section
*/
    uint64_t   size;           /* size in bytes of this section
*/
    uint32_t   offset;        /* file offset of this section */
    uint32_t   align;          /* section alignment (power of 2)
*/
};
```

Загрузочные команды таблиц символов и строк содержат только смещения таблиц символов и строк, а также количество идентификаторов в таблице символов и размер таблицы строк.

За загрузочными командами следует блок, заполненный нулевыми байтами – отступ (Padding). Отступ никак не используется загрузчиком, но может быть использован для подписи исполняемых файлов, а также для вставки инъекций в исполняемый файл при должном переопределении аттрибутов заголовка секции, следующей за отступом. Тем не менее, размер отступа ограничен, поэтому данный подход не будет использован в настоящем проекте.

Сразу после отступа следуют секции, объединенные в соответствующие им сегменты. В общем случае первой секцией всегда выступает секция __text сегмента __TEXT. Переопределить порядок секций и сегментов можно только в -preload режиме линковки, но такой файл не будет исполняемым.

Выравнивание сегментов всегда составляет 0x1000=4096 байт, выравнивание секции по умолчанию 2 байта, но может быть переопределено в заголовке секции.

Процесс загрузки Mach-O файлов в память во многом аналогичен выгрузке в память ELF-файла.

1.2. Обзор компилятора BTPC

Самоприменимый компилятор BeRo TinyPascal (BTPC) языка Pascal, рассматриваемый в данной работе, был разработан в 2006 году немецким программистом Бенджамином Россо (Benjamin Rosseaux, «BeRo») и распространяется под лицензией zlib для платформы Windows x32 [1].

В 2017 году компилятор был портирован студентом кафедры ИУ9 Антоном Беляевым на платформу Linux 64-bit [2]. Архитектура и основные принципы работы компилятора при переносе были сохранены, поэтому в дальнейшем при подробном рассмотрении компилятора детали будут описываться в контексте портированного компилятора, так называемого BTPC64, если не указано иное.

Компилятор преобразует исходный код из подмножества языка Pascal в двоичный код платформы Windows x86 либо в двоичный код платформы Linux x86-64. Более формально, входной язык компилятора составляет Pascal диалекта Delphi, не содержащий шаблоны, дженерики, перегрузку операторов и все нововведения языка позднее 2006 года.

Устройство компилятора можно условно разделить на две части:

- библиотека RTL (RunTime Library), содержащую низкоуровневые реализации базовых функций BTPC;
- непосредственно компилятор BTPC, реализующий фазы анализа и синтеза.

Рассмотрим подробнее каждую из данных составляющих, а затем опишем взаимодействие данных компонентов в контексте устройства компилятора.

1.2.1. Библиотека времени выполнения RTL

Библиотека времени выполнения RTL (RunTime Library) отвечает за низкоуровневое устройство BTPC – распределение памяти, пользовательский ввод и вывод, завершение программы. Ее исходный код находится в файле RTL.s.

Библиотека содержит девять основных функций:

1. RTLHalt – остановка программы.

2. RTLWriteChar – запись символа на stdout.
3. RTLWriteInteger – запись целого числа на stdout, принимает число и ширину вывода в символах.
4. RTLWriteLn – запись на stdout символа новой строки.
5. RTLReadChar – считывание символа из stdin.
6. RTLReadInteger – считывание целого из stdin.
7. RTLReadLn – пропуск в stdin символов до конца файла или ближайшего перевода строки.
8. RTLEOF – установка в регистр RAX ненулевого значения в случае конца файла и нуля в противном случае.
9. RTLEOLN – установка в регистр DL единицы, если следующий символ – перенос строки, либо нуля в противном случае.

Аргументы функций вывода устанавливаются в стеке, считанные из stdin символы и числа устанавливаются в регистр RAX.

Взаимодействие с библиотекой осуществляется посредством таблицы встроенных функций RTLFunctionTable, исходный код которой приведен на листинге 2. В таблице в описанном выше порядке находятся указатели на соответствующие функции, указатель на данную таблицу устанавливается в регистр RSI.

Листинг 2 – Таблица встроенных функций RTLFunctionTable

```
RTLFunctionTable:
    .quad RTLHalt
    .quad RTLWriteChar
    .quad RTLWriteInteger
    .quad RTLWriteLn
    .quad RTLReadChar
    .quad RTLReadInteger
    .quad RTLReadLn
    .quad RTLEOF
    .quad RTLEOLN
```

Принципиальную особенность библиотеки RTL составляет тот факт, что точка входа в программу указывает на метку StubEntryPoint, после которой происходит лишь установка таблицы встроенных функций в соответствующий регистр и метка ProgramEntryPoint. Исходный код последних строк файла RTL.s приведен на листинге 3.

Листинг 3 – Заключительные строки файла RTL.s

```
StubEntryPoint:
    movq %rsp, %rbp     # BeRo's legacy
    movq RTLFunctionTable@GOTPCREL(%rip), %rsi

ProgramEntryPoint:

#-----------------------------------------
# code generated by btpc.pas goes here
```

Таким образом, после компиляции и запуска RTL происходит неминуемая ошибка сегментирования (Segmentation fault). Как будет установлено позже, такое поведение является ожидаемым и обусловлено спецификой взаимодействия RTL и высокоуровневого компилятора.

1.2.2. Компилятор BTPC

Логика компилятора полностью реализована в файле BTPC.pas, содержащим и стадию анализа – чтение входного потока, лексический анализ, синтаксический анализ, генерация промежуточного представления, и фазу синтеза – генерация кода на целевом языке. Стадия оптимизации в компиляторе отсутствует. Кратко опишем все шаги, происходящие при компиляции, в нотации компилятора BTPC64. Схематичное представление процесса компиляции приведено на рисунке 3.

Вначале вручную на целевой платформе компилируется библиотека времени выполнения RTL в виде полноценного, но падающего при запуске с ошибкой сегментирования ELF-файла. С помощью скрипта линковки при

этом меняется порядок следования секций так, что секция .text, содержащая непосредственно исполняемые инструкции, оказывается после секции .data.

Затем также вручную, с помощью парсера rtl2pas.cpp, полученный исполняемый файл разбивается на два набора Pascal-строк фиксированной длины (в оригинале – 255) таких, что первый набор строк содержит все байты до секции .text включительно, а второй – оставшимися байтами до конца файла, при этом байты задаются десятичными числами. В случае, если длина последней строки первого набора оказывается меньше 255, строка дополняется байтами #$90 (тривиальная инструкция NOP – No operation) в hex формате, а последняя строка второго набора дополняется нулями #0.

Наконец, полученные Pascal-строки вручную добавляются в файл компилятора BTPC.pas через процедуры OutputCodeString(s:TOutputCodeString). Первый набор строк при этом объединен в процедуру EmitStubCode, второй – в процедуру EmitEndingStub. Принцип работы данных функций будет описан несколько позже. Исходный код первой функции с сокращениями приведен на листинге 4.

Листинг 4 – процедура EmitStubCode

```
procedure EmitStubCode;

begin
  OutputCodeString(#207#250#237#254#7#0#0#1...#0#0);
  OutputCodeString(#0#25#0#0#0#152#0#0#0#95#95...#24#0);
  OutputCodeString(#0#0#229#42#152#99#63#19#48...#0#0);
  ...
  OutputCodeString(#255#232#224#254#255#255#72...#$90#$90);
end;
```

Вернемся теперь к самому компилятору BTPC.pas. Над полученной на вход программой фронтенд компилятора производит необходимые стадии анализа и генерирует промежуточное представление – авторский байт-код, преобразующийся следующим образом: каждой инструкции (или нескольким инструкциям) байт-кода ставится в соответствие набор ассемблерных

инструкций. Данные инструкции, как и инструкции скомпилированного RTL.s, преобразуются в Pascal-строки.

Таким образом, получены три набора Pascal-строк: начало скомпилированной библиотеки RTL.s до секции .text включительно, окончание скомпилированной библиотеки и порожденные ассемблерные инструкции входной программы. Данные три набора строк конкатенируются в единую Pascal-строку следующим образом: сначала идет начало библиотеки RTL, затем порожденные ассемблерные инструкции, и наконец – окончание библиотеки RTL. Назовем данную строку *шаблоном* выходного файла.

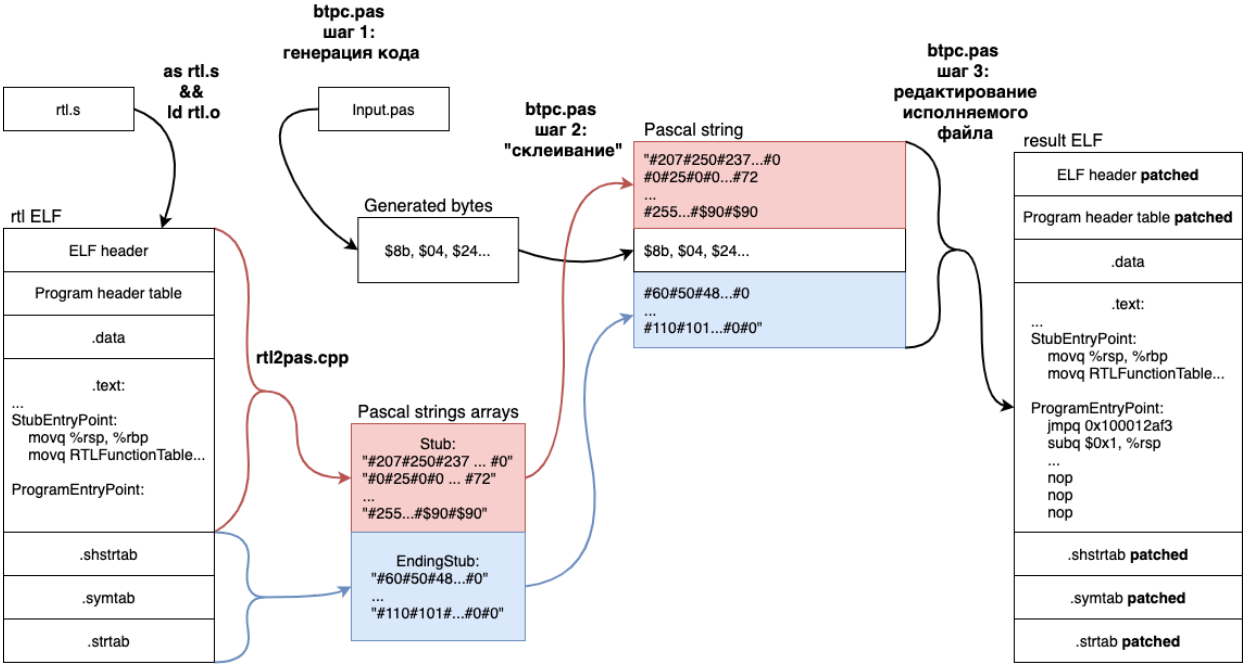

Рисунок 3 – схема работы компилятора BTPC64

Вспомним, что последняя метка ProgramEntryPoint секции .text файла RTL.s суть последняя строка этой секции и указывает в пустоту. После конкатенации же за меткой ProgramEntryPoint появляется полноценный «скомпилированный» код, являющийся программой, поданной на вход компилятору. Таким образом после конкатенации получается строка,

содержащая полноценное байтовое представление ELF-файла, включающего все функции библиотеки RTL и инструкции, порожденные по входной программе компилятора.

Единственную проблему теперь составляет неработоспособность полученного шаблона: все смещения, размеры и адреса в ELF-файле остались неизменными и потому после расширения скомпилированного RTL потеряли свою актуальность.

Для преодоления данной проблемы также в файле BTPC.pas применяется редактирование (патчевание, жарг.) полученного шаблона: все поля структур ELF-файла должны быть изменены в соответствии с произведенным изменением, как правило – увеличены на длину кода, порожденного по входной программе.

Так, например, должно быть увеличено на размер порожденного кода смещение таблицы заголовков секций, находящееся в поле e_shoff заголовка ELF-файла. Для этого требуется вычислить смещение (индекс первого байта) данного поля в строке-шаблоне (это 0x28 = 40) и увеличить его посредством Pascal-процедуры OutputCodePutInt32(offset,value:integer).

Подобную процедуру необходимо произвести со всеми потерявшими актуальность значениями, поэтому вероятность ошибки крайне велика, и проще предвычислить смещения всех данных значений при парсинге ELF-файла через rtl2pas.cpp.

Хотя данный подход крайне нетривиален и при малейшей ошибке делает невозможным запуск выходного файла, в компиляторах BTPC64 и BTPC порождаются работоспособные файлы в форматах ELF и PE32 соответственно, отчего можно сделать вывод, о том, что при корректном выполнении редактирования данная проблема преодолима.

Наконец, отредактированный шаблон выводится в стандартный поток вывода, на этом компиляция заканчивается.

2. Портирование

Вначале требуется добиться корректной сборки и работы библиотеки RTL на платформе macOS 64-bit, затем переписать парсер rtl2pas.cpp для работы с форматом Mach-O. Наконец, нужно отредактировать получаемый Mach-O файл и убедиться в работоспособности и самоприменимости компилятора.

2.1. Библиотека RTL

Портированная на Linux 64-bit библиотека RTL уже написана в AT&T-синтаксисе, поддерживаемом на macOS и порождает инструкции x86-64, совместимые с требуемой платформой. Тем не менее, имеется ряд особенностей ассемблерного кода для macOS, требующих изменения RTL.

Рассмотрим, например, функцию RTLReadChar и вспомогательную для нее функцию ReadCharEx. Первоначальный код данных функций приведен на листинге 5.

Листинг 5 – Исходный код функций ReadCharEx и RTLReadChar в BTPC64

```
ReadCharEx:
    xorq    %rax, %rax              #syscall #0 == Read();
    xorq    %rdi, %rdi              #p1 == read_from stdin
    movq    $ReadCharBuffer, %rsi   #p2 == write_to == buffer
    movq    $1, %rdx                #p3 == count == single_byte
    syscall
    ...
    ret
RTLReadChar:
    call    ReadCharInit
    xorq    %rax, %rax
    movb    (ReadCharBuffer), %al
    call    ReadCharEx
    ret
```

Заметим во-первых, что в macOS отличаются номера системных вызовов. Полный их список может быть найден в исходном коде ядра XNU

[7]. Кроме того, к каждому номеру должно быть добавлено число 0x2000000 вследствие наличия четырех уровней системных вызовов в macOS, из которых требуется выбрать уровень Unix/BSD [8].

Кроме того, всякое обращение к буферам, в том числе к собственным, должно происходить только через глобальную таблицу смещений (Global Offset Table, GOT). При необходимости получить значение из буфера из-за этого приходится вначале сохранять в некоторый регистр (например, R8) адрес буфера, и только затем вычитывать его значение.

Код портированных на macOS 64-bit функций RTLReadChar и ReadCharEx, демонстрирующий описанные выше особенности, приведен на листинге 6.

Листинг 6 – исходный код ReadCharEx и RTLReadChar в BTPC64macOS

```
ReadCharEx:
    movq    $0x2000003, %rax        #syscall #3 == Read();
    xorq    %rdi, %rdi              #p1 == read_from stdin
    movq    ReadCharBuffer@GOTPCREL(%rip), %rsi
                                    #p2 == write_to == buffer
    movq    $1, %rdx                #p3 == count == single_byte
    syscall
    ...
    ret
RTLReadChar:
    call    ReadCharInit
    xorq    %rax, %rax
    movq    ReadCharBuffer@GOTPCREL(%rip), %r8
    movb    (%r8), %al
    call    ReadCharEx
    ret
```

После того, как все функции переписаны должным образом, RTL может быть скомпилирован:

```
as rtl.s -o rtl.o && ld rtl.o -e _main -o rtl -lSystem
```

Полученный исполняемый файл может быть выполнен, но «штатно» упадет с ошибкой сегментирования. Если после метки ProgramEntryPoint

добавить вызов какой-либо RTL-функции, например RTLHalt (завершение с кодом 0), файл будет выполнен успешно.

Линковщик ld на macOS не поддерживает скрипты линковки, а опции -section_order и -segment_order, изменяющие порядок следования секций и сегментов соответственно, поддерживаются только в -preload режиме линковки, который не порождает исполняемые файлы в классическом смысле. Вследствие этого сегмент __DATA всегда следует после сегмента __TEXT в исполняемом Mach-O файле, поэтому адреса буферов изменяются после формирования шаблона выходного файла и необходимо редактировать смещения в таблице символов и адреса буферов в инструкциях.

Также требуется модифицировать парсер ELF-файлов rtl2pas.cpp. Главным образом, необходимо изменить чтение основных структур ELF-файла на чтение структур Mach-O файла, которые могут быть найдены в исходных файлах ядра XNU [5]. После парсинга загрузочных команд сегментов может быть найдено смещение начала сегмента __DATA, по которому происходит «раздел» скомпилированного исполняемого файла на наборы строк Stub и EndingStub. Генерация функций EmitStubCode и EmitEndingStub на языке Pascal происходит в данной программе сразу после парсинга, также выделяются смещения основных значений Mach-O файла, которые должны быть отредактированы после добавления сгенерированного кода компилируемой Pascal-программы.

На этом можно считать, что библиотека времени выполнения RTL полностью портирована на macOS 64-bit, поскольку имеется полный работоспособный набор функций, аналогичных оригинальной RTL и необходимый инструментарий для генерации шаблона выходного файла и вычисления смещений редактируемых значений исполняемого файла.

2.2. Редактирование Mach-O файла

К настоящему моменту начало и конец выходного файла могут быть получены посредством функций EmitStubCode и EmitEndingStub, и остается добиться работоспособности выходного файла. Генерируемый код компилируемой программы остается неизменным, поскольку инструкции в компиляторе BTPC64 порождаются для архитектуры x86-64 и без изменений переносятся на macOS 64-bit. Единственным нововведением относительно генерации кода стало дополнение сгенерированного блока кода операциями NOP (No operation, 0x90) до размера, кратного выравниванию сегментов, равному 0x1000, поскольку в выходном файле __text является единственной секцией сегмента __TEXT, а скомпилированная библиотека RTL всегда имеет размер сегмента __TEXT, равный 0x1000 (физически неиспользуемую часть сегмента занимает идущий перед секцией __text пустой блок – отступ «padding»).

Отключив генерацию кода, скомпилируем существующий «компилятор» на платформе Linux 64-bit, затем подадим ему на вход произвольную программу, после чего на выход будет подан шаблон с отсутствующим блоком сгенерированного кода. Полученный исполняемый файл будет корректным Mach-O файлом, совпадающим с портированным на macOS rtl64macOS и может быть запущен на платформе macOS 64-bit, что свидетельствует о том, что логика работы BTPC может быть перенесена на macOS, требуется только патчевание выходного файла.

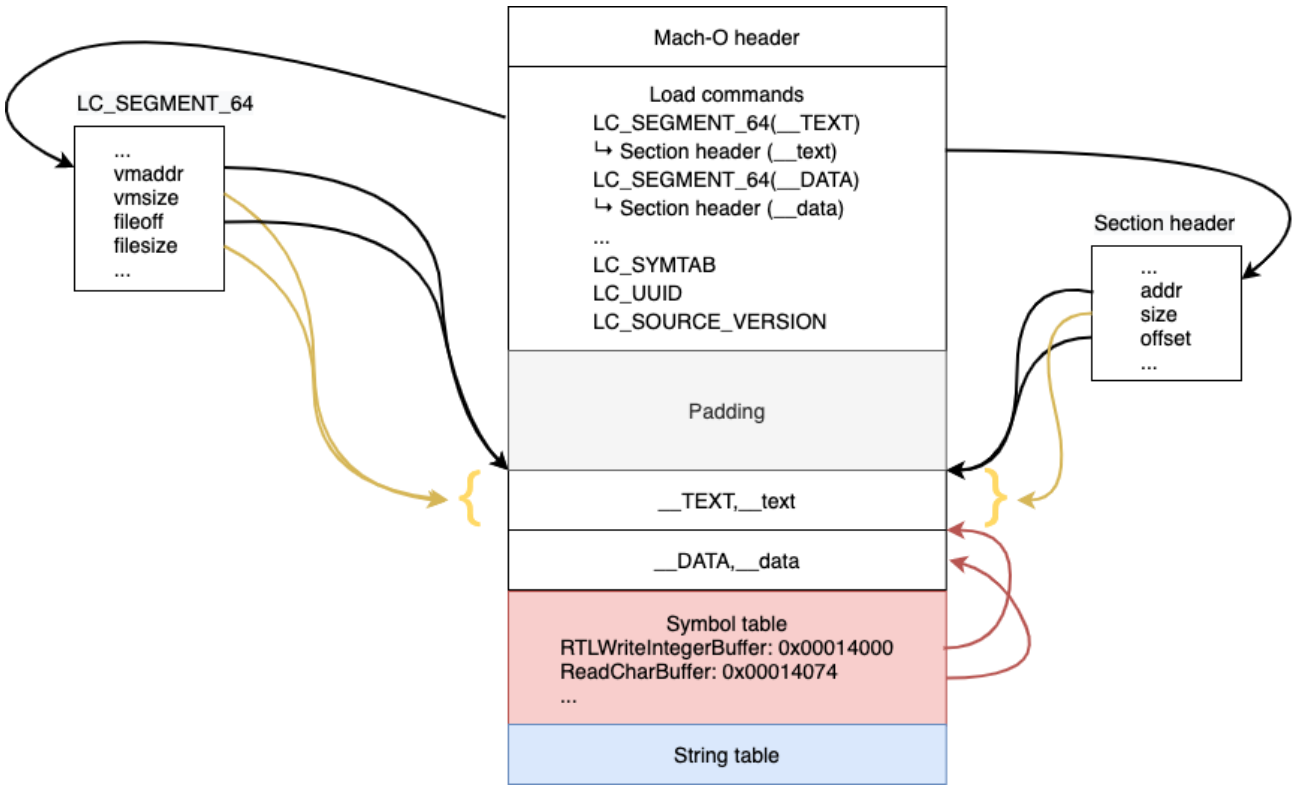

Рисунок 4 – Пример значений, подлежащих патчеванию.

На рисунке 4 приведено более подробное устройство файла формата Mach-O, загрузочная команда сегмента и заголовок секции показаны на примере сегмента __TEXT и секции __text соответственно. Можно заметить, что потеряют свою актуальность как минимум размеры расширяемой секции и сегмента, а также адреса и смещения всех секций и сегментов, следующих за расширяемыми (в нашем случае – секция __data и сегмент __DATA). Также патчевания требуют адреса идентификаторов секции __data (буферов) в таблице символов и обращения к ним в инструкциях секции __text. Полный список всех редактируемых значений приведен в таблице 1. В столбце «Смещение до поля» функция s(X) обозначает размер в байтах структуры X, а сокращения «LC» и «SH» – соответственно «load_command» и «section_header». Заметим, что все перечисленные значения требуют увеличения на размер сгенерированного кода.

Таблица 1 – Редактируемые поля

Имя поля	Смещение до поля
Text_LC.vmsize	s(mach_header) + s(SEG_LC) + 0x20
Text_LC.filesize	s(mach_header) + s(SEG_LC) + 0x30
Text_SH.size	s(mach_header) + 2*s(SEG_LC) + 0x28
Data_LC.vmaddr	s(mach_header) + 2*s(SEG_LC) + s(SH) + 0x18
Data_LC.offset	s(mach_header) + 2*s(SEG_LC) + s(SH) + 0x28
Data_SH.addr	s(mach_header) + 3*s(SEG_LC) + s(SH) + 0x20
Data_SH.offset	s(mach_header) + 3*s(SEG_LC) + s(SH) + 0x30
SymTab_LC.symoff	s(mach_header) + 4*s(SEG_LC) + 2*s(SH) + 0x8
SymTab_LC.stroff	s(mach_header) + 4*s(SEG_LC) + 2*s(SH) + 0x10
SymTab.Data[i].value	SymTab_LC.symoff + (30 + i) * s(n_list_64) + 0x8

Изменение значений происходит уже реализованной на Pascal функцией OutputCodePutInt32(o,i:integer), устанавливающей значение i в 4 байта по смещению o. В случае, если нужно отредактировать 8-байтовое поле, можно дважды воспользоваться данной функцией для младших и старших четверок байтов, однако в общем случае это не требуется, поскольку размер сгенерированного блока и существующие размеры и смещения не достаточно велики. Пример использования данной функции для патчевания полей структуры LC_SEGMENT сегмента __TEXT приведен на листинге 7.

Листинг 7 – редактирование структуры LC_SEGMENT

```
{__TEXT}
  OutputCodePutInt32(OffsSegTextVMSize + $1,
                     ValSegTextVMSize + InjectionSize);
  OutputCodePutInt32(OffsSegTextFileSize + $1,
                     ValSegTextFileSize + InjectionSize);
```

Кроме описанных в таблице 1 значений, отредактировать нужно также адреса в ассемблерных инструкциях скомпилированной библиотеки RTL. Поскольку инструкции представляются исключительно байтами, а самописное дизассемблирование опасно неточностями и ошибками, смещения данных полей приходится считать вручную, используя существующие инструменты дизассемблирования, например утилиту MachOView, предоставляющую возможность просмотра и редактирования структур и инструкций исполняемых файлов формата Mach-O.

После успешного редактирования всех значений можно сказать, что портирование компилятора на платформу macOS 64-bit завершено, и можно переходить к тестированию. Схема работы портированного компилятора приведена на рисунке 5.

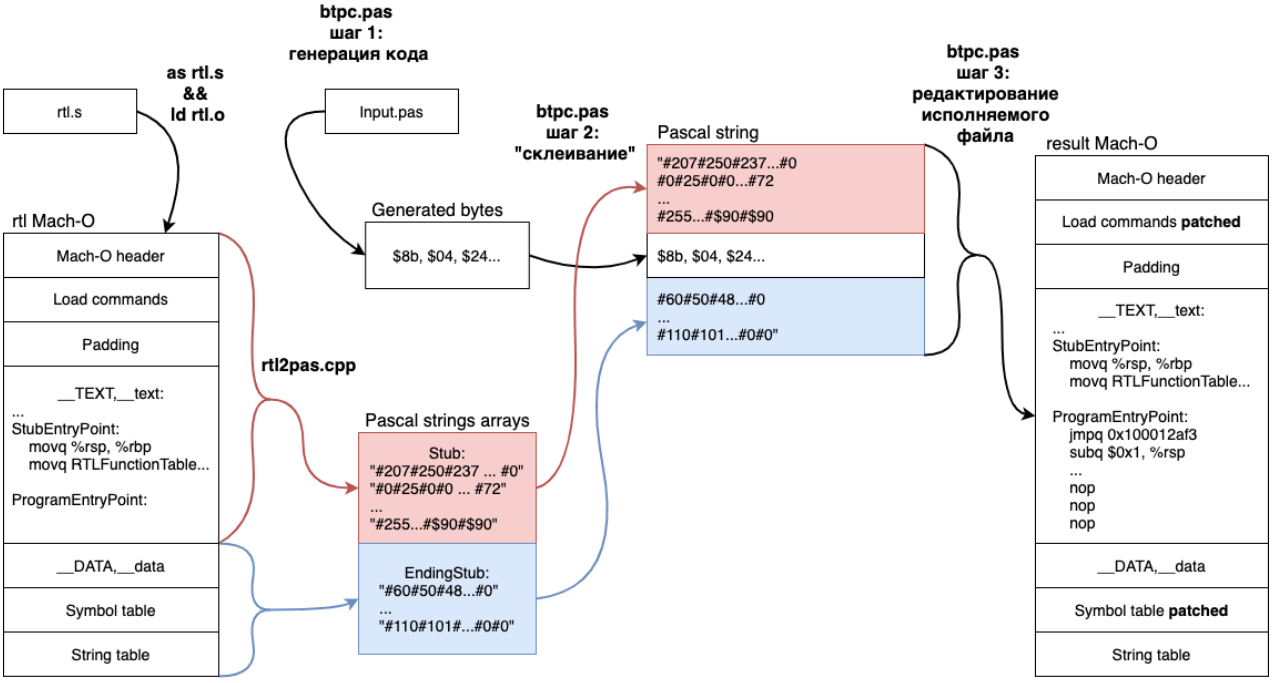

Рисунок 5. Схема работы портированного компилятора

3. Тестирование и перенос

3.1. Юнит-тестирование

Первым этапом тестирования портированной версии компилятора стало модульное тестирование. Необходимо протестировать работоспособность как библиотеки RTL в частности, так и компилятора в целом. Все тесты предыдущих авторов были унаследованы, что подтверждает сохранение архитектуры и основных принципов компилятора BTPC, были добавлены также и новые тесты.

Язык ассемблера не предназначен для написания тестов в классическом смысле, все тесты воспроизводят некоторые пользовательские сценарии и должны валидироваться самим тестировщиком. Пример простейшего теста вывода числа в стандартный поток и его вызов приведен на листинге 8.

Листинг 8 – Тестовая функция Test4 и окончание файла RTL64macOS.s

```
Test4:
    pushq $-1234
    pushq $6
    call RTLWriteInteger
    call RTLWriteLn
    call RTLHalt

StubEntryPoint:

    call Test4           # вызов теста

    movq %rsp, %rbp
    movq RTLFunctionTable@GOTPCREL(%rip), %rsi

ProgramEntryPoint:
#----------------------------------------
# code generated by btpc.pas goes here
```

Тесты библиотеки времени выполнения запускаются добавлением инструкций call на соответствующую метку после метки StubEntryPoint или

ProgramEntryPoint. Поскольку в этом случае скомпилированная RTL будет содержать постороннюю логику, после тестирования при разделении RTL на Pascal-строки все вызовы тестов должны быть удалены из RTL.

Для тестирования компилятора (как кросскомпилятора, так и впоследствии перенесенного на macOS компилятора) используются простейшие программы на языке Pascal, также воспроизводящие пользовательские сценарии работы с какой-либо базовой составляющей языка Pascal – ввод, вывод, арифметические выражения, процедуры, структуры. Пример такой тестовой программы приведен на листинге 9.

Листинг 9 – программа проверки компиляции функций ввода и вывода

```
program RTLWriteTest;

type TSignature = array[1..3] of integer;

var a,i:integer;
    m:TSignature;

begin

  i:=1;
  while i <= 3 do begin
    Read(a);
    WriteLn('->', a);
    m[i]:=a;
    i:=i+1;
  end;

  i:=1;
  while i <= 3 do begin
    WriteLn(m[i]);
    i:=i+1;
  end;
end.
```

Данный тест проверяет корректность компиляции в портированном BTPC ввода и вывода пользовательских значений. Для его выполнения достаточно подать тест на вход компилятору, после чего запустить. Важно, что при тестировании кросскомпилятора запуск компиляции и запуск

скомпилированного файла должен происходить в различных операционных системах – соответственно на Linux 64-bit и macOS 64-bit:

```
$linux> ./btpc64 <btpc64macOS.pas >btpc64CrossLinux
$linux> ./btpc64CrossLinux <./TDD/testRead.pas >testRead
$macOS> chmod 777 testRead && ./testRead
```

Таким образом, корректность работы портированных библиотеки RTL и кросскомпилятора проверена, и можно переходить непосредственно к переносу компилятора на целевую платформу.

3.2. Перенос компилятора

Напомним, что *раскрученным самоприменимым компилятором* языка Pascal можно назвать пару компиляторов (BTPC64.pas, BTPC64), где BTPC64 может быть запущен на исходной платформе Linux 64-bit и порождает по входной программе на языке Pascal код для данной платформы, а BTPC64.pas написан на языке Pascal и переводит программы на языке Pascal в исполняемые на Linux файлы. Тогда применение к BTPC64.pas компилятора BTPC64 порождает полностью идентичный BTPC64 компилятор.

В данном случае для переноса раскрученного самоприменимого компилятора (BTPC64.pas, BTPC64) на платформу macOS 64-bit достаточно:

1. Разработать на основе BTPC64.pas компилятор BTPC64macOS.pas, написанный на языке Pascal и переводящий программы на языке Pascal в исполняемые Mach-O файлы.
2. На исходной платформе скомпилировать на BTPC64 разработанный BTPC64macOS.pas, получить кросскомпилятор BTPC64CrossLinux, запускаемый на платформе Linux 64-bit и порождающий исполняемые на платформе macOS 64-bit файлы.
3. На исходной платформе скомпилировать с помощью BTPC64CrossLinux все тот же BTPC64macOS.pas и получить целевой компилятор BTPC64macOS, который может быть запущен на целевой платформе macOS 64-bit и компилирует программы на языке Pascal в исполняемые на данной платформе файлы. При этом получен раскрученный самоприменимый компилятор (BTPC64macOS.pas, BTPC64macOS).

В виде Т-диаграммы данный процесс представлен на рисунке 6.

Легко видеть, что описанный в первом пункте шаг был выполнен и описан в разделе 2 настоящего проекта, а второй шаг осуществлен и протестирован, описан в разделе 3.1. Таким образом, остается выполнить заключительный, третий шаг переноса:

```
$linux> ./btpc64CrossLinux <btpc64macOS.pas >btpc64macOS
```

Остается физически перенести исполняемый Mach-O файл на устройство под управлением ОС macOS 64-битной разрядности, где он может быть протестирован:

```
$macOS> ./btpc64macOS <./TDD/testRead.pas >testRead
$macOS> chmod 777 testRead && ./testRead
```

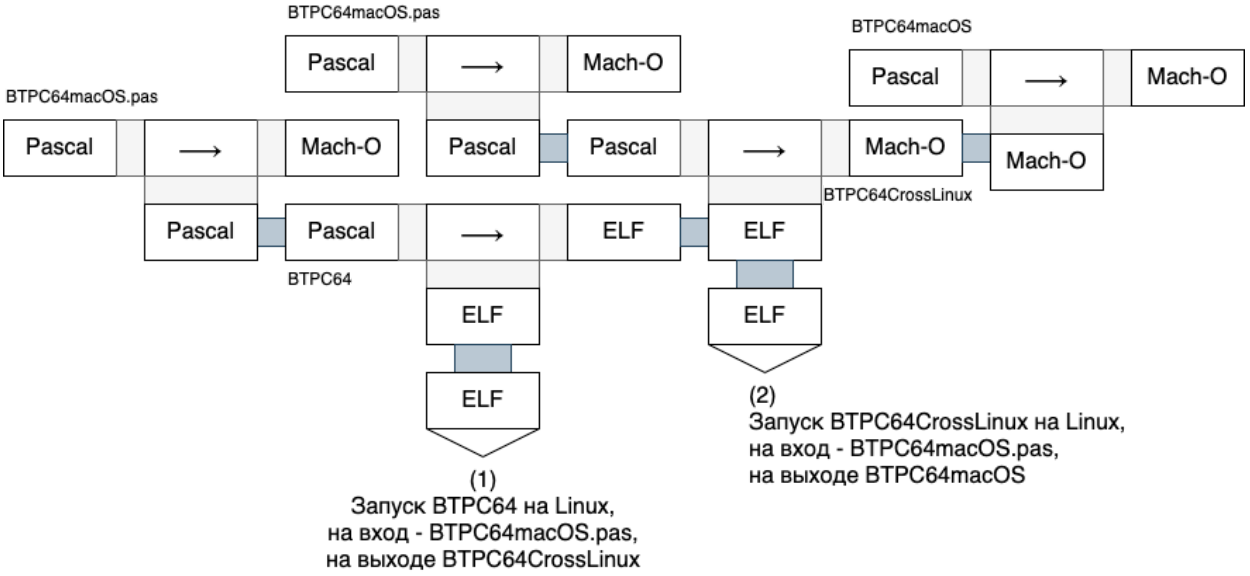

Рисунок 6 – перенос компилятора BTPC64 на macOS

Необходимо проверить и самоприменимость полученного компилятора:

```
$macOS> ./btpc64macOS <btpc64macOS.pas >btpc64macOSCheck
$macOS> diff btpc64macOS btpc64macOSCheck
```

Вывод команды diff пуст, что свидетельствует о полном побайтовом совпадении исполняемых файлов btpc64macOS (оригинальный компилятор) и btpc64macOSCheck (компилятор, полученный после самоприменения). Таким образом, компилятор BTPC портирован на платформу macOS 64-bit, работоспособен и самоприменим.

Заключение

В рамках настоящей курсовой работы был портирован на платформу macOS 64-bit самоприменимый компилятор BTPC.

В ходе работы было изучено внутреннее устройство форматов исполняемых файлов PE32, ELF и Mach-O, архитектура и принципы работы исходного компилятора и его портированной на Linux версии, процессы сборки и линковки на платформах Linux и macOS 64-битной разрядности.

На практическом примере рассмотрены процессы компиляции, кросскомпиляции и самоприменения компилятора, ранее рассмотренные в рамках курса «Конструирование компиляторов».

В результате тестирования установлено, что компилятор работает корректно, подтверждена самоприменимость полученного компилятора, что свидетельствует о выполнении поставленной задачи данного проекта.

Список использованных источников

1. BeRoTinyPascal: A self-hosting capable tiny pascal compiler for the Win32 x86 platform.
   URL: https://github.com/BeRo1985/berotinypascal (дата обращения 08.01.2021)
2. A Bauman BeRo TinyPascal: A ported version of self-hosting capable BeRo Tiny Pascal Compiler for the Linux x64 platform.
   URL: https://github.com/bmstu-iu9/A-Bauman-BTPC-64 (дата обращения 08.01.2021)
3. Executable and Linking Format (ELF) Specification.
   URL: https://refspecs.linuxfoundation.org/elf/elf.pdf (дата обращения 08.01.2021)
4. Mac OS X ABI Mach-O File Format Reference
   URL: https://web.archive.org/web/20090901205800/http://developer.apple.com/mac/library/documentation/DeveloperTools/Conceptual/MachORuntime/Reference/reference.html (дата обращения 08.01.2021)
5. Mach-O loader.h: XNU source code
   URL: https://opensource.apple.com/source/xnu/xnu-2050.18.24/EXTERNAL_HEADERS/mach-o/loader.h (дата обращения 08.01.2021)
6. Mach-O File Format: Introduction
   URL: https://h3adsh0tzz.com/2020/01/macho-file-format/ (дата обращения 08.01.2021)
7. XNU syscalls
   URL: https://github.com/opensource-apple/xnu/blob/master/bsd/kern/syscalls.master#L41 (дата обращения 10.01.2021)
8. XNU syscall classes
   URL: https://opensource.apple.com/source/xnu/xnu-1699.26.8/osfmk/mach/i386/syscall_sw.h (дата обращения 10.01.2021)