Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и технологий

Кафедра компьютерных систем и программных технологий

**Отчёт по лабораторной работе №4**

Дисциплина: Низкоуровневое программирование

Тема: Раздельная компиляция

Вариант: 3

Выполнил студент

гр. 3530901/90002

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Белых Б.А.

(подпись)

Руководитель

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Степанов Д.С.

(подпись)

«\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2021 г.

Санкт-Петербург   
 2021

**Постановка задачи**

На языке C разработать функцию, реализующую сортировку выбором массива чисел in-place. Поместить определение функции в отдельный исходный файл, оформить заголовочный файл. Разработать тестовую программу на языке C. Собрать программу «по шагам». Проанализировать выход препроцессора и компилятора. Проанализировать состав и содержимое секций, таблицы символов, таблицы перемещений и отладочную информацию, содержащуюся в объектных файлах и исполняемом файле. Выделить разработанную функцию в статическую библиотеку. Разработать make-файлы для сборки библиотеки и использующей ее тестовой программы. Проанализировать ход сборки библиотеки и программы, созданные файлы зависимостей.

**Текст программы**

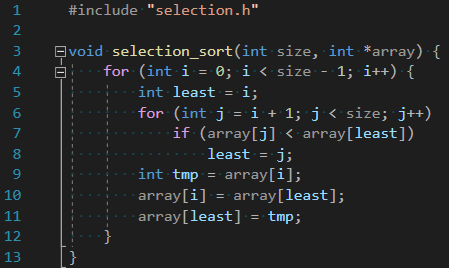


Рис. 1 Основной файл selection.c.

C:\Users\Administrator\Desktop\selh.png

Рис. 2 Заголовочный файл selection.h.

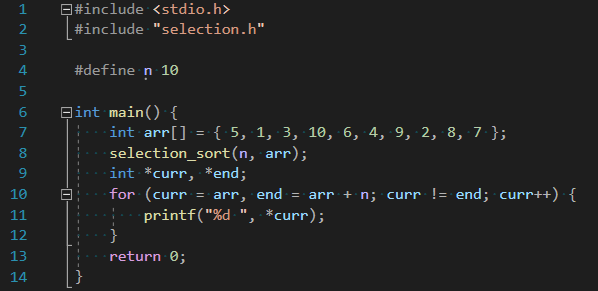


Рис. 3 Тестовая программа main.c.

**Сборка программы “по шагам”**

Первым шагом является препроцессирование файлов исходного текста (файлов “main.c” и “selection.c”), результаты записываются в файлы “main.i” и “selection.i”:

*riscv64-unknown-elf-gcc -march=rv32ic -mabi=ilp32 -O1 -E main.c -o main.i*

*riscv64-unknown-elf-gcc -march=rv32ic -mabi=ilp32 -O1 -E selection.c -o selection.i*

Параметры:

-march=rv32ic -mabi=ilp32 - целевым является процессор с базовой архитектурой системы команд RV32I;

-O1 - выполнять простые оптимизации генерируемого кода (мы используем эту опцию в примерах, потому что обычно генерируемый код получается более простым);

-E - остановить процесс сборки после препроцессирования;

-o - путь к выходному файлу.

Выход препроцессора (фрагменты из файлов main.i и selection.i) приведён на рис. 4 и рис. 5.

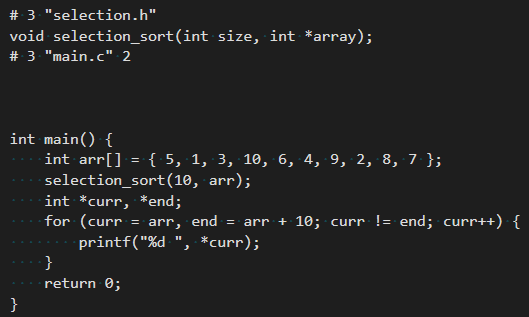


Рис. 4 Фрагмент main.i.

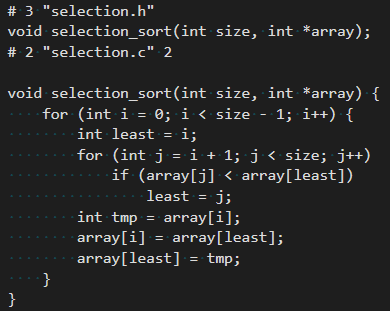


Рис. 5 Фрагмент selection.i.

Результат препроцессирования содержится в файле selection.i и main.i. По причине того, что исходные файлы содержат заголовочные файлы нескольких стандартных библиотек С, результат препроцессирования отличается от исходных файлов и имеет достаточно много добавочных строк, среди которых и исходные программы. Также можно заметить, что препроцессор включил содержимое файла selection.h.

Появившиеся нестандартные директивы, начинающиеся с символа “#”, используются для передачи информации об исходном тексте из препроцессора в компилятор.

Далее необходимо выполнить компиляцию файлов “main.i” и “selection.i”, сохранив результат - сгенерированный код на языке ассемблера - в файлы “main.s” и “selection.s”.

*riscv64-unknown-elf-gcc -march=rv32ic -mabi=ilp32 -O1 -S -fpreprocessed main.i -o main.s*

*riscv64-unknown-elf-gcc -march=rv32ic -mabi=ilp32 -O1 -S -fpreprocessed selection.i -o selection.s*

Параметры:

-S - остановить процесс сборки после компиляции, не запуская ассемблер;

-fpreprocessed - выполняется компиляция файла, уже отработанного процесса;

Все остальные параметры из прошлого пункта означают тоже самое.

Выход компилятора (файл main.s):

|  |
| --- |
| .file "main.c"  .option nopic  .attribute arch, "rv32i2p0\_c2p0"  .attribute unaligned\_access, 0  .attribute stack\_align, 16  .text  .section .rodata.str1.4,"aMS",@progbits,1  .align 2  .LC1:  .string "%d "  .text  .align 1  .globl main  .type main, @function  main:  addi sp,sp,-64  sw ra,60(sp)  sw s0,56(sp)  sw s1,52(sp)  lui a5,%hi(.LANCHOR0)  addi a5,a5,%lo(.LANCHOR0)  lw t3,0(a5)  lw t1,4(a5)  lw a7,8(a5)  lw a6,12(a5)  lw a0,16(a5)  lw a1,20(a5)  lw a2,24(a5)  lw a3,28(a5)  lw a4,32(a5)  lw a5,36(a5)  sw t3,8(sp)  sw t1,12(sp)  sw a7,16(sp)  sw a6,20(sp)  sw a0,24(sp)  sw a1,28(sp)  sw a2,32(sp)  sw a3,36(sp)  sw a4,40(sp)  sw a5,44(sp)  addi a1,sp,8  li a0,10  call selection\_sort  addi s0,sp,8  lui s1,%hi(.LC1)  .L2:  lw a1,0(s0)  addi a0,s1,%lo(.LC1)  call printf  addi s0,s0,4  addi a5,sp,48  bne s0,a5,.L2  li a0,0  lw ra,60(sp)  lw s0,56(sp)  lw s1,52(sp)  addi sp,sp,64  jr ra  .size main, .-main  .section .rodata  .align 2  .set .LANCHOR0,. + 0  .LC0:  .word 5  .word 1  .word 3  .word 10  .word 6  .word 4  .word 9  .word 2  .word 8  .word 7  .ident "GCC: (SiFive GCC-Metal 10.2.0-2020.12.8) 10.2.0" |

Наибольший интерес представляет файл main.s, так как в нем можно заметить обращение к подпрограмме selection\_sort. Следует отметить, что символ selection\_sort используется в файле, но никак не определяется.

Ассемблирование файлов “main.s” и “selection.s” выполняется по следующей команде:

*riscv64-unknown-elf-gcc -march=rv32ic -mabi=ilp32 -O1 -c main.s -o main.o*

*riscv64-unknown-elf-gcc -march=rv32ic -mabi=ilp32 -O1 -c selection.s -o selection.o*

Параметры:

-c - остановить процесс сборки после ассемблирования.

На выходе получили объектные файлы main.o и selection.o. Они содержат коды инструкций, таблицу символов и таблицу перемещений (relocations). В отличие от ранее рассмотренных файлов, объектный файл не является текстовым, для изучения его содержимого используем утилиту objdump, отображающую содержимое бинарных файлов в текстовом виде.

Как известно, содержательная часть объектного файла разбита на «разделы», называемые обычно секциями (section). Следующая команда обеспечивает отображение заголовков секций файлов “main.o” и “selection.o”:

*riscv64-unknown-elf-objdump -h main.o selection.o*

Заголовки секций файлов main.o и selection.o приведены на рис. 6.

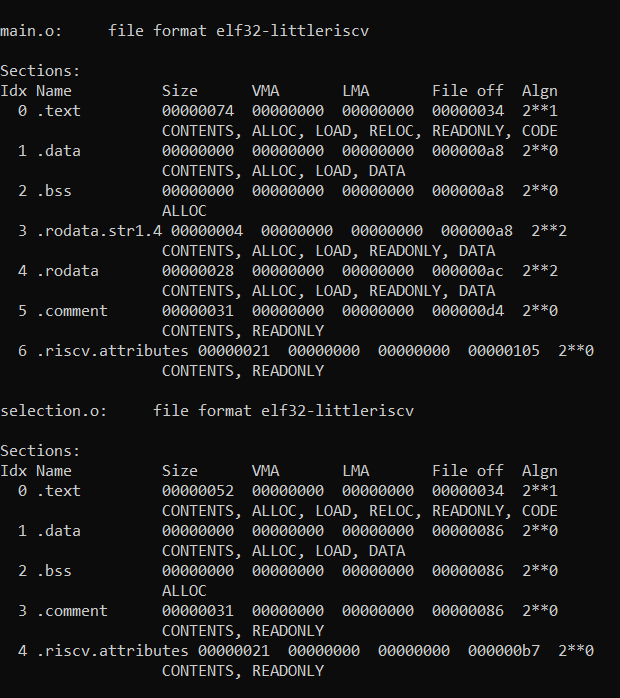


Рис. 6 Заголовки секций полученных файлов.

В файлах “main.o” и “selection.o” имеются следующие секции:

.text - секция кода, в которой содержатся коды инструкций (название секции обусловлено историческими причинами);

.data - секция инициализированных данных;

.bss - секция неинициализированных статических переменных (название секции также обусловлено историческими причинами);

.rodata - аналог .data для неизменяемых данных

.comment - секция данных о версиях размером 12 байт

.riscv.attributes - информация про RISC-V

Изучим содержимое таблиц символов объектных файлов “main.o” и “selection.o”:

*riscv64-unknown-elf-objdump -t main.o selection.o*

Таблица символов файлов main.o и selection.o приведена на рис. 7.

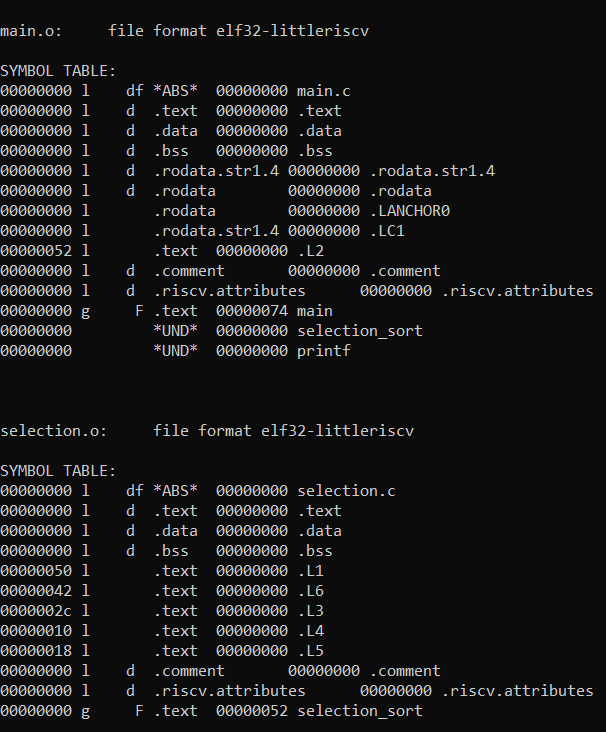


Рис. 7 Таблица символов полученных файлов.

В таблице символов main.o имеется запись: символ “selection” типа \*UND\*. Эта запись означает, что символ “selection” использовался в ассемблерном коде, из которого был получен данный объектный файл, но не был определен; ассемблер сделал вывод о том, что символ должен быть определен где-то еще, и отразил это в таблице символов.

Изучим содержимое секции “.text” объектных файлов “main.o” и “selection.o”:

*riscv64-unknown-elf-objdump -s -j .text main.o selection.o*

Вывод утилиты приведён на рис. 8.

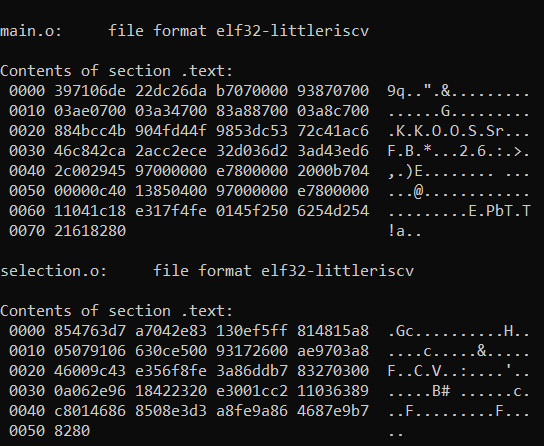


Рис. 8 Коды инструкций программы, сформированные ассемблером и размещенные в секции “.text”.

Процедура декодирования кодов инструкций является «механической» (иначе как бы ее реализовывало техническое устройство - процессор), следовательно, разумно поручить ее выполнение ЭВМ:

*riscv64-unknown-elf-objdump -d -M no-aliases -j .text selection.o main.o*

Опция “-d” инициирует процесс дизассемблирования (disassemble), опция “-M no-aliases” требует использовать в выводе только инструкции системы команд (но не псевдоинструкции ассемблера).

Вывод утилиты (фрагмент из main.o) приведён на рис. 9.

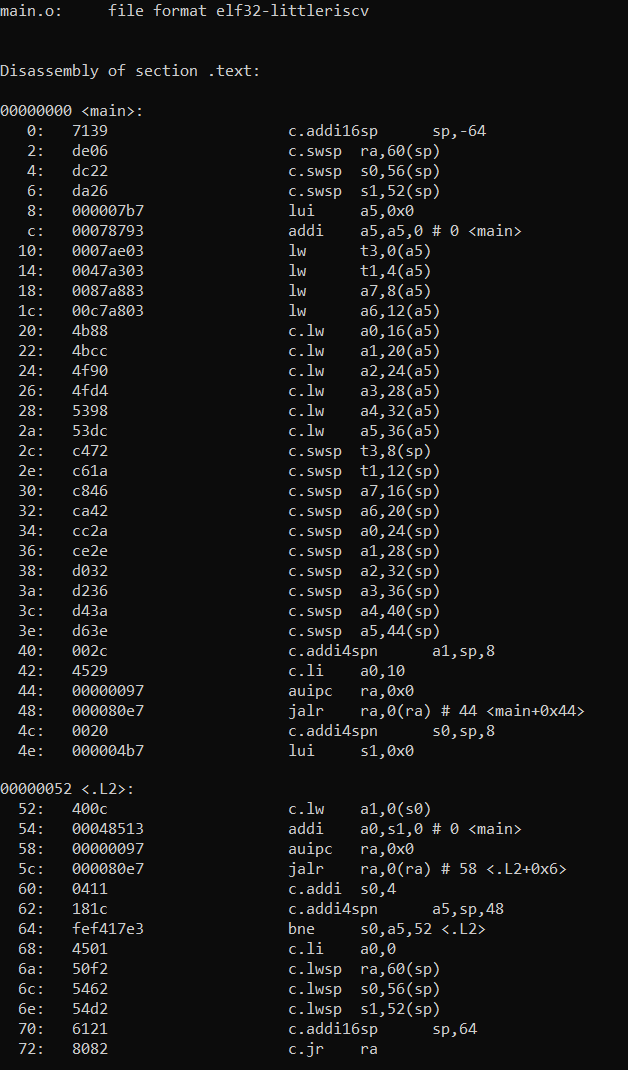


Рис. 9 Дизассемблирование main.o.

Интерес представляет результат дизассемблирования “main.o”: сравнивая его с “main.s”, легко понять, что псевдоинструкция вызова подпрограммы “selection”, транслировалась ассемблером в следующую пару инструкций:



Результатом выполнения этой пары инструкций станет переход на адрес 44, то есть зацикливанием. Ассемблер не имел возможности определить целевой адрес перехода (кроме того, что этот адрес обозначен символом “selection\_sort”), поэтому не мог сформировать корректную инструкцию (пару инструкций) передачи управления. В результате была сформирована пара инструкций с некорректными (нулевыми) значениями непосредственных операндов. Для получения исполняемого кода эта пара инструкций должна быть исправлена компоновщиком.

Информация обо всех «неоконченных» инструкциях передается ассемблером компоновщику посредством таблицы перемещений:

*riscv64-unknown-elf-objdump -r main.o selection.o*

Вывод утилиты (фрагмент) приведён на рис. 10.

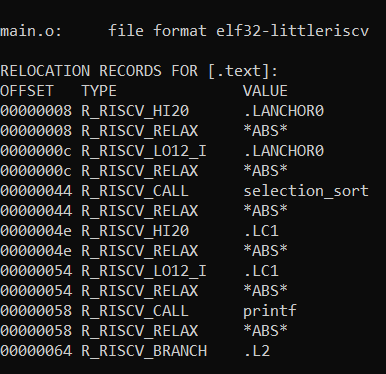


Рис. 10 Таблица перемещения для main.o.

В файле “main.o” имеется две записи, относящиеся к адресу 44 (как мы видели выше, по этому адресу в “main.o” находится первая инструкция пары auipc+jalr). Дизассемблирование и вывод таблицы перемещений можно совместить:

*riscv64-unknown-elf-objdump -d -M no-aliases -r main.o*

Вывод утилиты (фрагмент) приведён на рис. 11.

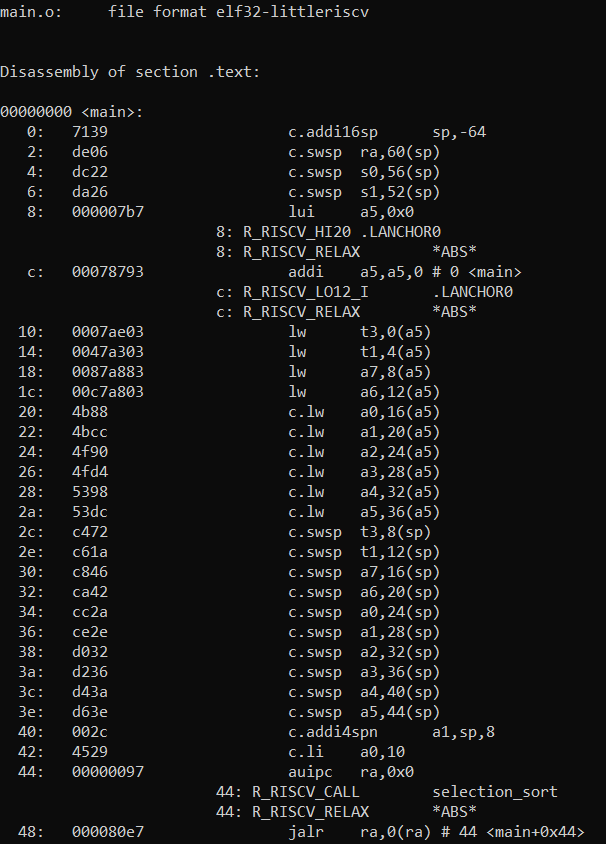


Рис. 11 Дизассемблирование и вывод таблицы перемещения для main.o.

Для того чтобы внести необходимые исправления, требуется знать, что исправить, как исправить и какой символ следует использовать, именно эта информация и содержится в записях о перемещениях. Так, в таблице перемещений указано, что по адресу 44 следует исправить пару инструкций (тип перемещения “R\_RISCV\_CALL”) так, чтобы результат соответствовал вызову подпрограммы “selection”. Типы перемещений специфичны для каждой архитектуры системы команд и обычно определены в ABI (Application Binary Interface).

Следующая запись таблицы перемещений специфична для средств разработки RISC-V. Записи типа “R\_RISCV\_RELAX” заносятся в таблицу перемещений в дополнение к записям типа “R\_RISCV\_CALL” (и некоторым другим) и сообщают компоновщику, что пара инструкций, обеспечивающих вызов подпрограммы, может быть оптимизирована. Для удобства изложения запретим компоновщику выполнять такую оптимизацию:

*riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -Wl,--no-relax main.o selection.o -o main*

Опция “-Wl” драйвера “gcc” позволяет передавать дополнительные аргументы компоновщику. Здесь мы используем опцию компоновщика “--no-relax”, отключающую оптимизацию, о которой шла речь выше.

Компоновка программы выполняется по следующей команде:

*riscv64-unknown-elf-gcc -march=rv32ic -mabi=ilp32 main.o selection.o -o main*

Результатом является исполняемый файл “main”. Изучим содержимое секции “.text” полученного в результате компоновки программы исполняемого файла:

*riscv64-unknown-elf-objdump -j .text -d -M no-aliases main >main.ds*

Нас интересует только фрагмент результирующего файла “main.ds” (рис. 12).

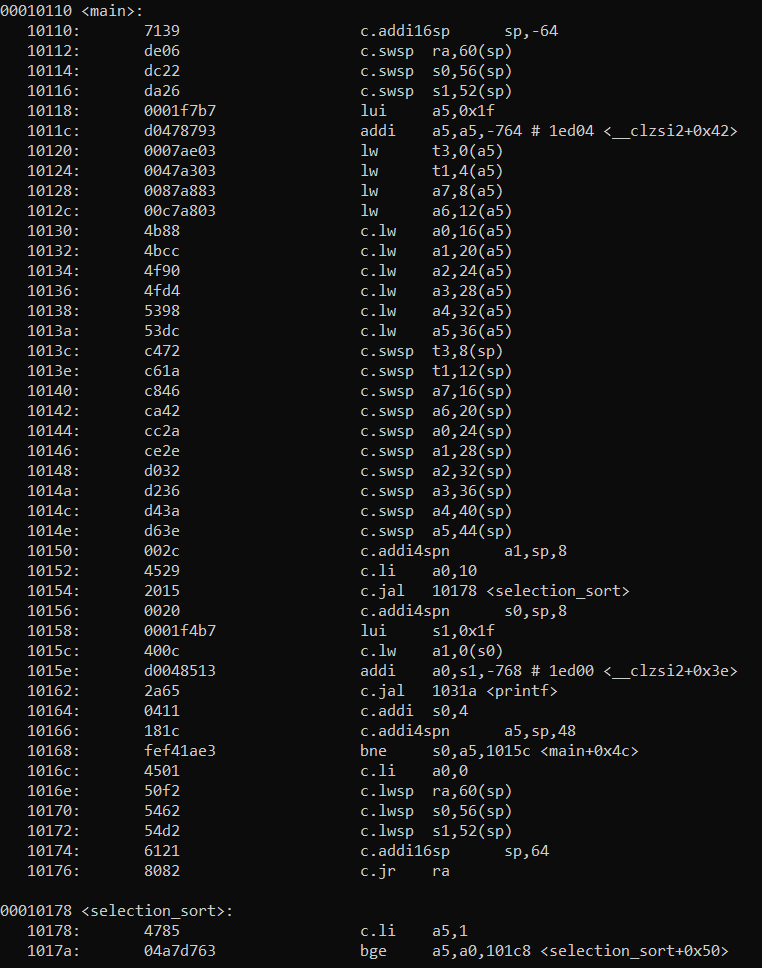


Рис. 12 Фрагмент main.ds.

Можно увидеть, пара инструкций auipc+jalr в подпрограмме main заменена компоновщиком одной инструкцией jal.

**Создание статической библиотеки**

Статическая библиотека (static library) является, по сути, архивом (набором, коллекцией) объектных файлов.

Поместим selection.o в статическую библиотеку lib:

*riscv64-unknown-elf-ar -rsc lib.a selection.o*

Параметры:

-r - заменить старые файлы с такими названиями (selection.o), если они уже есть в архиве;

-s - записать «index» в архив. Index - это список всех символов, объявленных во включенных в архив объектных файлах, и его присутствие ускоряет линковку;

-с - создать архив, если его еще не было.

Используем статическую библиотеку для сборки программ, для этого напишем make-файл. Makefile - это набор инструкций для программы make, которая помогает собирать программу из файлов в один вызов make. Полученный Makefile приведён на рис. 13.

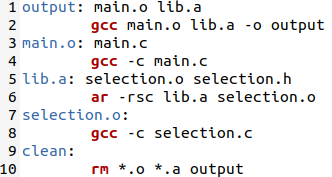


Рис. 13 Файл make.

Что происходит в Makefile:

1. Создаём объектный файл main.o из исходного main.c

2. Создаём объектный файл selection.o из исходного selection.c

3. Архивируем объектный файл selection.o (создаём статическую библиотеку lib.a)

4. Компонуем статическую библиотеку lib.a с объектным файлом main.o и получаем исполняемый файл output

Соберём программу при помощи терминала ОС Linux (рис. 14).

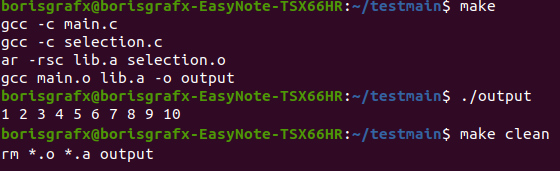


Рис. 14 Тестирование make.

1. С помощью команды make собираем программу: make обращается к Makefile и выполняет инструкции в нём

2. Получаем исполняемый файл output. Проверяем его работу. Он работает верно – программа сортирует входной массив по возрастанию

3. С помощью make clean обращаемся к target clean в Makefile и очищаем все объектные файлы, которые использовались в сборке

Таким образом, поставленная нам задача выполнена. В ходе выполнения данной лабораторной работы была разработана функция на языке C, реализующая сортировку выбором массива чисел in-place. Затем была выполнена сборка программы «по шагам». Проанализированы выход препроцессора и компилятора. Проанализированы состав и содержимое секций, таблицы символов, таблицы перемещений и отладочная информация, содержащаяся в объектных файлах и исполняемом файле. Разработанная функция была выделена в статическую библиотеку. Разработаны make-файлы для сборки библиотеки и использующей ее тестовой программы. Проанализированы ход сборки библиотеки и программы, созданные файлы зависимостей.