Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Кафедра компьютерных систем и программных технологий

Низкоуровневое программирование

Отчет по лабораторной работе №4
Раздельная компиляция

Работу выполнил:
Кечин В.В.
Группа:
3530901/90004
Преподаватель:
Алексюк А.О.

Санкт-Петербург
2021

Содержание

1. Цель работы **3**

2. Программа работы **3**

3. Ход выполнения работы **3**

4. Выводы **11**

1. Цель работы

- На языке C разработать функцию, реализующую определенную вариантом задания функциональность. 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 исполняеммом 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.

2. Программа работы

- Изучить методические материалы, опубликованные на сайте курса.
- Установить пакет средств разработки “SiFive GNU Embedded Toolchain” для RISC-V.
- Разработать программу и проанализировать ее компиляцию.
- Выделить разработанную функцию в статическую библиотеку.

3. Ход выполнения работы

Общая идея заключается в том, что мы меняем максимальный элемент с последним необработанным, последовательно уменьшая количество необработанных элементов. Для этого внешний цикл будет сокращать границу необработанной части массива, а внутренний искать максимальный элемент среди необработанных. Листинг программы:

```
1 #include "sort.h"
2
3 int * sort (int *arr, int n)
4 {
5   for (int i = n - 1; i > 0; i--)
6     {
7       int max = 0;
8       for (int j = 1; j <= i; j++) {
9         if (arr[j] > arr[max])
10             {
11              max = j;
12              }
13   }
14       int t = arr[i];
15       arr[i] = arr[max];
16       arr[max] = t;
17     }
18   return arr;
19 }
```

Теперь разработаем тестовую программу, которая будет вызывать нашу сортировку:

```c
#include "sort.h"

int main (void)
{
  int arr[] = { 0, 5, 1, 4, 6, 2, 3, 7, 9, 8 };
  int n = sizeof(arr) / sizeof(arr[0]);
  sort(arr, n);
  return 0;
}
```

sort.h:

```c
#pragma once

int * sort (int *, int);
```

Программа реализует заданный функционал, в чем можно удостовериться, выполнив их в компиляторе.

Процесс сборки простейшей программы состоит из следующих шагов:

1. Запуск программы cc1 с параметром “-E”. Исполняемая команда в упрощенном виде:

```
cc1.exe –E –v main.c –march=rv32i –mabi=ilp32 –O1 –o main.i
```

На данном шаге выполняется обработка файла исходного текста “main.c” только препроцессором (опция “-E”), результат сохраняется в файле ”main.i” (параметр “-o”).

2. Запуск программы cc1 с параметром “-fpreprocessed”. Исполняемая команда в упрощенном виде:

```
cc1.exe –fpreprocessed main.i –march=rv32i –mabi=ilp32 –O1
```

-o main.s На данном шаге выполняется компиляция файла “main.i”, уже обработанного препроцессором (опция “-fpreprocessed”), результат работы компилятора – код на языке ассемблера – сохраняется в файле ”main.s”.

3. Запуск программы as. Исполняемая команда в упрощенном виде:

```
as.exe –v –march=rv32i –mabi=ilp32 –o main.o main.s
```

На данном шаге выполняется ассемблирование файла “main.s”, результат работы ассемблера – объектный код – сохраняется в файле ”main.o”.

4. Запуск программы collect2. Исполняемая команда в упрощенном виде:

```
collect2.exe lib/rv32i/ilp32/crt0.o rv32i/ilp32/crtbegin.o main.o
 –lgcc –lc –lgloss –lgcc rv32i/ilp32/crtend.o
```

Программа collect2 является утилитой gcc, запускающей компоновщик. На данном шаге выполняется компоновка – формирование исполнимого файла из ранее созданных объектных файлов. Как можно видеть из команды, осуществляется компоновка объектных файлов “crt0.o“, “crtbegin.o“, “crtend.o“, относящихся к реализации среды времени выполнения (C runtime) и созданного на предыдущем шаге объектного файла “main.o”. Кроме того, в компоновке могут участвовать объектные файлы из библиотек “libgcc”, “libc”, “libgloss” (опции “-l...”). Имя выходного файла не указано, и по умолчанию результат работы компоновщика записывается в файл “a.out”.

Подытожим: Обработка файла препроцессором добавляет различные директивы, которые необходимы компилятору, и все записывается в файл .i. Затем код на C переводится на язык ассемблера в файл .s. После происходит ассеблирование файла .s и результат - объектный код (важно заметить, что он бинарный, и мы не сможем его прочитать как обычный текстовый файл) - сохраняется в файл .o. И наконец компоновщик, как и говорит его название, компонует все в исполняемый файл .out.

Теперь выполним компиляцию и проанализируем все эти файлы. Используем следующую команду:

```
riscv64-unknown-elf-gcc --save-temps -march=rv32i -mabi=ilp32 -O1 -v main.c
    ↪ sort.c >log 2>&1

```

Файлы .i по сути содержат исходный код без комментариев и с директивами, которые перередают необходимую информацию об исходном тексте из препроцессора в компилятор; например, последняя директива «1 “main.c”» информирует компилятор о том, что следующая строка является результатом обработки строки 1 исходного файла “main.c”. main.i:

```
# 1 "main.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "main.c"
# 1 "sort.h" 1


int * sort(int *, int);
# 2 "main.c" 2

int main (void)
{
  int arr[] = { 0, 5, 1, 4, 6, 2, 3, 7, 9, 8 };
  int n = sizeof(arr) / sizeof(arr[0]);
  sort(arr, n);
  return 0;
}
```

sort.i:

```
# 1 "sort.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "sort.c"
# 1 "sort.h" 1


int * sort(int *, int);
# 2 "sort.c" 2

int * sort (int *arr, int n)
{
  for (int i = n - 1; i > 0; i--)
    {
      int max = 0;
      for (int j = 1; j <= i; j++) {
       if (arr[j] > arr[max])
           {
            max = j;
            }
 }
```

```
22        int t = arr[i];
23        arr[i] = arr[max];
24        arr[max] = t;
25      }
26    return arr;
27  }
```

Изучим файлы “main.s” и “sort.s сформированные компилятором: sort.s:

```
1    .file "sort.c"
2    .option nopic
3    .attribute arch, "rv32i2p0"
4    .attribute unaligned_access, 0
5    .attribute stack_align, 16
6    .text
7    .align  2
8    .globl  sort
9    .type sort, @function
10 sort:
11   li  a5,1
12   ble a1,a5,.L2
13   mv  a6,a1
14   slli  a1,a1,2
15   add a7,a0,a1
16   li  t1,0
17   li  t3,1
18   j .L3
19 .L4:
20   addi  a4,a4,1
21   addi  a3,a3,4
22   beq a4,a6,.L7
23 .L5:
24   slli  a5,a2,2
25   add a5,a0,a5
26   lw  a1,0(a3)
27   lw  a5,0(a5)
28   ble a1,a5,.L4
29   mv  a2,a4
30   j .L4
31 .L7:
32   lw  a5,-4(a7)
33   slli  a2,a2,2
34   add a2,a0,a2
35   lw  a4,0(a2)
36   sw  a4,-4(a7)
37   sw  a5,0(a2)
38   addi  a6,a6,-1
39   addi  a7,a7,-4
40   beq a6,t3,.L2
41 .L3:
42   addi  a5,a6,-1
43   addi  a3,a0,4
44   li  a4,1
45   mv  a2,t1
46   bgt a5,zero,.L5
47   j .L7
48 .L2:
49   ret
50   .size sort, .-sort
51   .ident  "GCC: (SiFive GCC 8.3.0-2020.04.1) 8.3.0"
```

main.s:

```
  .file "main.c"
  .option nopic
  .attribute arch, "rv32i2p0"
  .attribute unaligned_access, 0
  .attribute stack_align, 16
  .text
  .align  2
  .globl  main
  .type main, @function
main:
  addi  sp,sp,-64
  sw  ra,60(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)
  li  a1,10
  addi  a0,sp,8
  call  sort
  li  a0,0
  lw  ra,60(sp)
  addi  sp,sp,64
  jr  ra
  .size main, .-main
  .section  .rodata
  .align  2
  .set  .LANCHOR0,. + 0
.LC0:
  .word 0
  .word 5
  .word 1
  .word 4
  .word 6
  .word 2
  .word 3
  .word 7
  .word 9
  .word 8
  .ident  "GCC: (SiFive GCC 8.3.0-2020.04.1) 8.3.0"
```

В подпрограмме “main” выполняется обращение к подпрограмме “sort” (значение ре-

гистра ra, содержащее адрес возврата из “main”, сохраняется на время вызова в стеке). Следует отметить, что символ “sort” используется в файле “main.s”, но никак не определяется.

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

```
./riscv64-unknown-elf-objdump -t sort.o main.o

sort.o:     формат файла elf32-littleriscv

SYMBOL TABLE:
00000000 l    df *ABS*  00000000 sort.c
00000000 l    d  .text  00000000 .text
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss 00000000 .bss
00000084 l       .text  00000000 .L2
0000006c l       .text  00000000 .L3
00000048 l       .text  00000000 .L7
00000020 l       .text  00000000 .L4
0000002c l       .text  00000000 .L5
00000000 l    d  .comment 00000000 .comment
00000000 l    d  .riscv.attributes  00000000 .riscv.attributes
00000000 g     F .text  00000088 sort



main.o:     формат файла elf32-littleriscv

SYMBOL TABLE:
00000000 l    df *ABS*  00000000 main.c
00000000 l    d  .text  00000000 .text
00000000 l    d  .data  00000000 .data
00000000 l    d  .bss 00000000 .bss
00000000 l    d  .rodata  00000000 .rodata
00000000 l       .rodata  00000000 .LANCHOR0
00000000 l    d  .comment 00000000 .comment
00000000 l    d  .riscv.attributes  00000000 .riscv.attributes
00000000 g     F .text  00000080 main
00000000         *UND*  00000000 sort
```

В таблице символов “main.o” имеется интересная запись: символ “sort” типа “*UND*”. Эта запись означает, что символ “sort” 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 “zero.o”:

```
riscv64-unknown-elf-objdump -d -M no-aliases -j .text sort.o main.o

sort.o:     file format elf32-littleriscv


Disassembly of section .text:

00000000 <sort>:
   0: 00100793            addi  a5,zero,1
   4: 08b7d063            bge a5,a1,84 <.L2>
   8: 00058813            addi  a6,a1,0
   c: 00259593            slli  a1,a1,0x2
  10: 00b508b3            add a7,a0,a1
  14: 00000313            addi  t1,zero,0
  18: 00100e13            addi  t3,zero,1
```

```
  1c: 0500006f            jal zero,6c <.L3>

00000020 <.L4>:
  20: 00170713            addi  a4,a4,1
  24: 00468693            addi  a3,a3,4
  28: 03070063            beq a4,a6,48 <.L7>

0000002c <.L5>:
  2c: 00261793            slli  a5,a2,0x2
  30: 00f507b3            add a5,a0,a5
  34: 0006a583            lw  a1,0(a3)
  38: 0007a783            lw  a5,0(a5)
  3c: feb7d2e3            bge a5,a1,20 <.L4>
  40: 00070613            addi  a2,a4,0
  44: fddff06f            jal zero,20 <.L4>

00000048 <.L7>:
  48: ffc8a783            lw  a5,-4(a7)
  4c: 00261613            slli  a2,a2,0x2
  50: 00c50633            add a2,a0,a2
  54: 00062703            lw  a4,0(a2)
  58: fee8ae23            sw  a4,-4(a7)
  5c: 00f62023            sw  a5,0(a2)
  60: fff80813            addi  a6,a6,-1
  64: ffc88893            addi  a7,a7,-4
  68: 01c80e63            beq a6,t3,84 <.L2>

0000006c <.L3>:
  6c: fff80793            addi  a5,a6,-1
  70: 00450693            addi  a3,a0,4
  74: 00100713            addi  a4,zero,1
  78: 00030613            addi  a2,t1,0
  7c: faf048e3            blt zero,a5,2c <.L5>
  80: fc9ff06f            jal zero,48 <.L7>

00000084 <.L2>:
  84: 00008067            jalr  zero,0(ra)

main.o:     file format elf32-littleriscv


Disassembly of section .text:

00000000 <main>:
   0: fc010113            addi  sp,sp,-64
   4: 02112e23            sw  ra,60(sp)
   8: 000007b7            lui a5,0x0
   c: 00078793            addi  a5,a5,0 # 0 <main>
  10: 0007ae03            lw  t3,0(a5)
  14: 0047a303            lw  t1,4(a5)
  18: 0087a883            lw  a7,8(a5)
  1c: 00c7a803            lw  a6,12(a5)
  20: 0107a503            lw  a0,16(a5)
  24: 0147a583            lw  a1,20(a5)
  28: 0187a603            lw  a2,24(a5)
  2c: 01c7a683            lw  a3,28(a5)
  30: 0207a703            lw  a4,32(a5)
  34: 0247a783            lw  a5,36(a5)
  38: 01c12423            sw  t3,8(sp)
  3c: 00612623            sw  t1,12(sp)
```

```
  40: 01112823            sw  a7,16(sp)
  44: 01012a23            sw  a6,20(sp)
  48: 00a12c23            sw  a0,24(sp)
  4c: 00b12e23            sw  a1,28(sp)
  50: 02c12023            sw  a2,32(sp)
  54: 02d12223            sw  a3,36(sp)
  58: 02e12423            sw  a4,40(sp)
  5c: 02f12623            sw  a5,44(sp)
  60: 00a00593            addi  a1,zero,10
  64: 00810513            addi  a0,sp,8
  68: 00000097            auipc ra,0x0
  6c: 000080e7            jalr  ra,0(ra) # 68 <main+0x68>
  70: 00000513            addi  a0,zero,0
  74: 03c12083            lw  ra,60(sp)
  78: 04010113            addi  sp,sp,64
  7c: 00008067            jalr  zero,0(ra)
```

Результат дизассемблирования “sort.o” интереса не представляет, в отличие от результата дизассемблирования “main.o”: сравнивая его с “main.s”, легко понять, что псевдоинструкция вызова подпрограммы “sort”, транслировалась ассемблером в следующую пару инструкций:

```
 68:  00000097 auipc ra,0x0
 6c:  000080e7     jalr ra,0(ra) # 68 <main+0x8>

```

Результатом выполнения этой пары инструкций станет переход на адрес 68, т.е. зацикливание! Загадочное поведение ассемблера объясняется очень просто: ассемблер не имел возможности определить целевой адрес перехода (кроме того, что этот адрес обозначен символом “zero”), 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. Компоновщик узнает об этом из таблицы перемещений, которая содержит информацию обо всех неоконченных инструкциях. Таблица перемещений:

```
riscv64-unknown-elf-objdump -r sort.o main.o

sort.o:     file format elf32-littleriscv

RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000004 R_RISCV_BRANCH    .L2
0000001c R_RISCV_JAL       .L3
00000028 R_RISCV_BRANCH    .L7
0000003c R_RISCV_BRANCH    .L4
00000044 R_RISCV_JAL       .L4
00000068 R_RISCV_BRANCH    .L2
0000007c R_RISCV_BRANCH    .L5
00000080 R_RISCV_JAL       .L7



main.o:     file format elf32-littleriscv

RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000008 R_RISCV_HI20      .LANCHOR0
00000008 R_RISCV_RELAX     *ABS*
```

```
0000000c R_RISCV_LO12_I      .LANCHOR0
0000000c R_RISCV_RELAX       *ABS*
00000068 R_RISCV_CALL        sort
00000068 R_RISCV_RELAX       *ABS*
```

В main.o видим две записи, относящиеся к адресу 68 (как мы видели выше, по этому адресу в “main.o” находится первая инструкция пары auipc+jalr).

Создадим библиотеку из объектного файла sort.o:

```
riscv64-unknown-elf-ar -rsc libsort.a sort.o
```

Результирующим файлом является “libsort.a”. Проверим его содержимое:

```
riscv64-unknown-elf-ar -t libsort.a
```

Вывод утилиты: sort.o

Изучим таблицу символов полученных исполняемых файлов:

```
./riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -O1 --save-temps main.c
    ↪ libsort.a -o main11

00000000 l    df *ABS*  00000000 sort.c
...
000101c0 g     F .text  00000088 sort
```

Итого видим, что библиотека успешно создана и используется.

4. Выводы

- Реализована программа соритровки на C с тестовой программой и заголовочным файлом.
- Проанализированы этапы компиляции программы от препроцессора до компановщика, просмотрены и изучены основные его этапы и принципы работы.
- Также создана статическая библиотека с функцией сортировки, которой можно успешно пользоваться.