Смирнов Иван ИУ7-22Б - 2023г.

Отчет

Задание №2

1. Этапы получения исполняемого файла

Целью данной работы является изучение процесса получения исполняемого файла и организации объектных и исполняемых файлов.

2. Задание:

1) В качестве **простой программы на языке Си** была выбрана программа lab_01_04_01. Код программы приведен ниже:

```c
#include <stdio.h>
#include <math.h>

#define OK 0
#define FLOORS 9
#define FLATS_PER_FLOOR 4

int main(void)
{
    int flat_num;

    printf("Введите номер квартиры:\n");
    scanf("%d", &flat_num);

    int entrance = ((flat_num - 1) / (FLOORS * FLATS_PER_FLOOR)) + 1;
    int floor = ((flat_num - (entrance - 1) * (FLOORS * FLATS_PER_FLOOR) - 1) / FLATS_PER_FLOOR) + 1;

    printf("Номер подъезда = %d \nЭтаж = %d \n", entrance, floor);

    return OK;
}
```

Программа выводит на экран номер подъезда и этажа по введенному с клавиатуры номеру квартиры девятиэтажного дома, считая, что на каждом этаже 4 квартиры, а нумерация квартир начинается с первого подъезда.

2) **Этапы получения исполняемого файла:**

A1: Обработка препроцессором

Препроцессор выполняет следующие действия:

a) Удаление комментариев
b) Вставка файлов (директива include)
c) Текстовые замены (директива define)
d) Условную компиляцию (директива if)

Результат исполнения команды ***cpp -o main.i main.c***:

```
. . .
extern int printf (const char *__restrict __format, ...);
. . .
int main(void)
{
    int flat_num;

    printf("P’PIPµPrPëC,Pµ PSPsPjPµCЪ PєPIP°CЪC,PëCЪC‹:\n");
    scanf("%d", &flat_num);

    int entrance = ((flat_num - 1) / (9 * 4)) + 1;
    int floor = ((flat_num - (entrance - 1) * (9 * 4) - 1) / 4) + 1;

    printf("PќPsPjPµCЪ PïPsPrCЉPµP·PrP° = %d \nPC,P°P¶ = %d \n",
entrance, floor);

    return 0;
}
```

P.S. при переносе текстового файла из Ubuntu на Windows возникла проблема с кодировкой (русские буквы). На задание это никак не влияет.

A2: Трансляция на язык ассемблера

Файл main.i (полученный препроцессором) передается на вход транслятору c99, который переводит его с языка Си на язык ассемблера.
Трансляция на язык ассемблера позволяет:

a) Упростить реализацию и отладку транслятора
b) Повысить его переносимость с одной платформы на другую

Результат исполнения команды ***c99 -S -fverbose-asm -masm=intel main.i***:

```
     .file "main.c"

. . .

     .text
     .section   .rodata
     .align 8
.LC0:
     .string
     "\320\222\320\262\320\265\320\264\320\270\321\202\320\265
```

```
\320\275\320\276\320\274\320\265\321\200
\320\272\320\262\320\260\321\200\321\202\320\270\321\200\321\213:"
.LC1:
      .string    "%d"
      .align 8
.LC2:
      .string    "\320\235\320\276\320\274\320\265\321\200
\320\277\320\276\320\264\321\212\320\265\320\267\320\264\320\260 = %d
\n\320\255\321\202\320\260\320\266 = %d \n"
      .text
      .globl     main
      .type main, @function
main:
.LFB0:
      .cfi_startproc
      endbr64
      push  rbp   #
      .cfi_def_cfa_offset 16
      .cfi_offset 6, -16
      mov   rbp, rsp   #,
      .cfi_def_cfa_register 6
      sub   rsp, 32    #,
# main.c:9: {
      mov   rax, QWORD PTR fs:40  # tmp111, MEM[(<address-space-1> long
unsigned int *)40B]
      mov   QWORD PTR -8[rbp], rax     # D.3198, tmp111
      xor   eax, eax   # tmp111
# main.c:12:     printf("P’PIPµPґPëC,Pµ PSPsPjPµCЪ
PєPIP°CЪC,PëCЪC‹:\n");
      lea   rax, .LC0[rip]  # tmp93,
      mov   rdi, rax   #, tmp93
      call  puts@PLT   #
# main.c:13:     scanf("%d", &flat_num);
      lea   rax, -20[rbp]    # tmp94,
      mov   rsi, rax   #, tmp94
      lea   rax, .LC1[rip]  # tmp95,
      mov   rdi, rax   #, tmp95
      mov   eax, 0     #,
      call  __isoc99_scanf@PLT    #
# main.c:15:     int entrance = ((flat_num - 1) / (FLOORS *
FLATS_PER_FLOOR)) + 1;
      mov   eax, DWORD PTR -20[rbp]    # flat_num.0_1, flat_num
      sub   eax, 1     # _2,
. . .
```

A3: Ассемблирование в объектный файл

С языка ассемблера программа переводится в машинный код с помощью транслятора as. На выходе – двоичный файл (а не текстовый!). Этот файл называется объектным.

Результат команды: ***as main.s -o main.o:***

```
□ELF_____________>_____________________@___________@_____@___
_y__ъUH‰eHґм
dH‹_%(___H‰Ешl1АНЌ_____H‰Зи____НЌЕмH‰ЖНЌ_____H‰Зё____и____‹Емґи_HcPHiT9
```

```
Ћг8НБк
Бъ_Бш_‰Б‰Р)ИґА_‰Ер‹Ерґи_kРЬ‹Ем_Рґи_ЌР_…А_НВБш_ґА_‰Еф‹Uф‹Ер‰ЖНЌ_____Н‰З
ё____и____ё____Н‹UшdН+_%(___t_и____ЙГ____Р’РIРµРґРёС,Рµ PSPsPjРµСЂ
РєРIР°СЂС,РёСЂС‹:_%d__РќPsPjРµСЂ РїPsPґСЉРµР·РґР° = %d
РС,Р°Р¶ = %d
__GCC: (Ubuntu 11.3.0-1ubuntu1~22.04)
. . .
```

P.S. так как файл main.o является бинарным, а не текстовым, то если открыть его с помощью текстового редактора, разобрать информацию файла будет невозможно.

Для этого используем команду **hexdump main.o**:

0000000 457f 464c 0102 0001 0000 0000 0000 0000

0000010 0001 003e 0001 0000 0000 0000 0000 0000

0000020 0000 0000 0000 0000 0440 0000 0000 0000

0000030 0000 0000 0040 0000 0000 0040 000e 000d

0000040 0ff3 fa1e 4855 e589 8348 20ec 4864 048b

0000050 2825 0000 4800 4589 31f8 48c0 058d 0000

0000060 0000 8948 e8c7 0000 0000 8d48 ec45 8948

0000070 48c6 058d 0000 0000 8948 b8c7 0000 0000

0000080 00e8 0000 8b00 ec45 e883 4801 d063 6948

0000090 39d2 e38e 4838 eac1 c120 03fa f8c1 891f

00000a0 89c1 29d0 83c8 01c0 4589 8bf0 f045 e883

00000b0 6b01 dcd0 458b 01ec 83d0 01e8 508d 8503

…

A4: Компоновка

Для получения исполняемого файла необходимо вызвать компоновщик.
В процессе получения исполняемого файла компоновщик решает несколько задач

a) Объединяет несколько объектных файлов в единый исполняемый файл
b) Выполняет связывание переменных и функций, которые требуются очередному объектному файлу, но находятся где-то в другом месте
c) Добавляет специальный код, который подготавливает окружение для вызова функции main, а после ее завершения выполняет обратные действия.

Результат команды ***ld -o main.exe main.o*** возвращает исполняемый файл main.exe.

3) **GCC** и **CLANG**

- называются “программами-драйверами”, так как компиляторы умеют выполнять действия четырех разных утилит для получения объектного файла самостоятельно или с помощью вызова внешних утилит (они напрямую взаимодействуют с операционной системой).

4) ***-v*** и ***-save-temps***

Ключ -v печатает (в стандартный вывод ошибок) команды выполняемые для запуска стадий компиляции. Также печатает номер версии управляющей программы компилятора, препроцессора и самого компилятора.

Результат запуска компилятора с ключом **-v**:

Using built-in specs.

COLLECT_GCC=gcc

OFFLOAD_TARGET_NAMES=nvptx-none:amdgcn-amdhsa

OFFLOAD_TARGET_DEFAULT=1

Target: x86_64-linux-gnu

Configured with: ../src/configure -v --with-pkgversion='Ubuntu 11.3.0-1ubuntu1~22.04' --with-bugurl=file:///usr/share/doc/gcc-11/README.Bugs --enable-languages=c,ada,c++,go,brig,d,fortran,objc,obj-c++,m2 --prefix=/usr --with-gcc-major-version-only --program-suffix=-11 --program-prefix=x86_64-linux-gnu- --enable-shared --enable-linker-build-id --libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --libdir=/usr/lib --enable-nls --enable-bootstrap --enable-clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-object --disable-vtable-verify --enable-plugin --enable-default-pie --with-system-zlib --enable-libphobos-checking=release --with-target-system-zlib=auto --enable-objc-gc=auto --enable-multiarch --disable-werror --enable-cet --with-arch-32=i686 --with-abi=m64 --with-multilib-list=m32,m64,mx32 --enable-multilib --with-tune=generic --enable-offload-targets=nvptx-none=/build/gcc-11-xKiWfi/gcc-11-11.3.0/debian/tmp-nvptx/usr,amdgcn-amdhsa=/build/gcc-11-xKiWfi/gcc-11-11.3.0/debian/tmp-gcn/usr --without-cuda-driver --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=x86_64-linux-gnu --with-build-config=bootstrap-lto-lean --enable-link-serialization=2

Thread model: posix

…

ignoring nonexistent directory "/usr/local/include/x86_64-linux-gnu"

ignoring nonexistent directory "/usr/lib/gcc/x86_64-linux-gnu/11/include-fixed"

ignoring nonexistent directory "/usr/lib/gcc/x86_64-linux-gnu/11/../../../../x86_64-linux-gnu/include"

#include "..." search starts here:

#include <...> search starts here:

/usr/lib/gcc/x86_64-linux-gnu/11/include

/usr/local/include

/usr/include/x86_64-linux-gnu

/usr/include

End of search list.

…

COLLECT_GCC_OPTIONS='-std=c99' '-Wall' '-Werror' '-v' '-c' '-o' 'main.exe' '-mtune=generic' '-march=x86-64' '-dumpdir' 'main.'

Ключ -save-temps сохраняет промежуточные “временные” файлы; они помещаются в текущий каталог, а их имена основываются на имени исходного файла (для компиляции файла main.c появятся промежуточные файлы main.i, main.s, main.o).

a) В отличие от действий (из пункта 2), компиляция gcc позволила сразу получить исполняемый файл одной командой, а не четырьмя.
b) Содержимое в файлах *.i, *.s, *.o (по смыслу содержания) осталось неизменным.
c) Однако содержимое файла *.s отличается от того, что был в пункте 2.

Файл **main.s**:

```
main:
.LFB0:
	.cfi_startproc
	endbr64
	pushq %rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq  %rsp, %rbp
	.cfi_def_cfa_register 6
	subq  $32, %rsp
	movq  %fs:40, %rax
	movq  %rax, -8(%rbp)
	xorl  %eax, %eax
	leaq  .LC0(%rip), %rax
…
0:
	.string	"GNU"
1:
	.align 8
	.long 0xc0000002
	.long 3f - 2f
2:
	.long 0x3
```

```
3:
	.align 8
4:
```

В отличие от файла в пункте 2, данный файл не содержит комментарии.

d) Библиотеки: --libexecdir, --libdir, --libstdcxx, libphobos, system-zlib
Объектные файлы: ключ -v не показал объектных файлов (ignoring nonexistent directory).

5) Этапы компиляции **clang**:
   - Препроцессирование
   - Трансляция на язык ассемблера
   - Ассемблирование
   - Компоновка
6) Ключ -S позволяет остановиться после компиляции.
   Команда **gcc -S main.c** (вывод см. пункт 4(c))
7) Ключ -c позволяет остановиться после ассемблирования.
   Команда **gcc -c main.c** (вывод см. пункт 2(A3))
8) Команда **objdump -rdw main.o:**

main.o: формат файла elf64-x86-64

Дизассемблирование раздела .text:

0000000000000000 <main>:

0: f3 0f 1e fa endbr64

4: 55 push %rbp

5: 48 89 e5 mov %rsp,%rbp

8: 48 83 ec 20 sub $0x20,%rsp

c: 64 48 8b 04 25 28 00 00 00 mov %fs:0x28,%rax

15: 48 89 45 f8 mov %rax,-0x8(%rbp)

19: 31 c0 xor %eax,%eax

1b: 48 8d 05 00 00 00 00 lea 0x0(%rip),%rax # 22 <main+0x22> 1e: R_X86_64_PC32 .rodata-0x4

…

9) Global_var_init – глобальная проинициализированная переменная (.data)

   Global_var_notinit – глобальная непроинициализированная переменная (.bss)

Результат команды **objdump -t main.o**:

main.o: формат файла elf64-x86-64

SYMBOL TABLE:

0000000000000000 l df *ABS* 0000000000000000 main.c

0000000000000000 l d .text 0000000000000000 .text

0000000000000000 l d .rodata 0000000000000000 .rodata

0000000000000000 g O .data 0000000000000004 global_var_init

0000000000000000 g O .bss 0000000000000004 global_var_notinit

0000000000000000 g F .text 00000000000000c4 main

0000000000000000 *UND* 0000000000000000 puts

0000000000000000 *UND* 0000000000000000 __isoc99_scanf

0000000000000000 *UND* 0000000000000000 printf

0000000000000000 *UND* 0000000000000000 __stack_chk_fail

Глобальные переменные попали в область g.

Локальные переменные попали в область l.

10) Чтобы добавить отладочную информацию к объектному файлу, необходимо добавить ключ **-g** в команду *gcc -c main.c.*

Результат команды **objdump -t main.o**:

…

0000000000000000 l d .rodata 0000000000000000 .rodata

0000000000000000 l d .debug_info 0000000000000000 .debug_info

0000000000000000 l d .debug_abbrev 0000000000000000 .debug_abbrev

0000000000000000 l d .debug_line 0000000000000000 .debug_line

0000000000000000 l d .debug_str 0000000000000000 .debug_str

0000000000000000 l d .debug_line_str 0000000000000000 .debug_line_str

0000000000000000 g O .data 0000000000000004 global_var_init

…

В символьную таблицу добавились “debug-строки”.

Так же файл можно запустить под отладчиком с помощью команды

gdb main.o

11) **gcc main.o -o main.exe -lm** (если есть объектный файл)

или

gcc -std=c99 -Wall -Werror -o main.exe main.c (если есть только файл *.c)

12) Ответы на вопросы:

(a) – Без отладочной информации:

1) main.o – 2064 байт
2) main.exe – 16176 байт

– С отладочной информацией:

1) main.o – 4952 байт
2) main.exe – 17728 байт

(b) (команда objdump -ht (main.o/main.exe))– Без отладочной информации:

1) main.o – 7 секций
2) main.exe – 26 секций

– С отладочной информацией:

1) main.o – 13 секций
2) main.exe – 32 секций

(c) – Расположение функций, глобальных и локальных переменных при добавлении отладочной информации не изменилось.

13) Динамические библиотеки (команда ldd main.exe):

```
linux-vdso.so.1 (0x00007ffcb1dee000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f4506200000)
/lib64/ld-linux-x86-64.so.2 (0x00007f4506629000)
```

Список литературы:

1. Курс “Проектно-технологическая практика (знакомство с Linux)”: https://e-learning.bmstu.ru/iu7/course/view.php?id=76

2. Крус “Проектно-технологическая практика (тестирование, отладка и профилирование ПО)”:

https://e-learning.bmstu.ru/iu7/course/view.php?id=73

3. Официальная документация gcc:

https://gcc.gnu.org/onlinedocs/gcc/Invoking-GCC.html

4. Документация “Командные опции GCC”:

https://www.opennet.ru/docs/RUS/gcc/gcc1-2.html

5. Динамические библиотеки и зависимости:

https://skeletor.org.ua/?p=1529

6. Документация “От С к Ассемблеру”:

https://www.opennet.ru/base/dev/from_c_to_asm.txt.html

7. Документация “Препроцессор. Подключаемые файлы.”

https://www.opennet.ru/docs/RUS/cpp/cpp-4.html