汇编实验准备知识

陈嘉杰　　费翔

清华大学

2021

一些约定

一些约定：

1. 形如 a{,b,c,d}e 的表示方法，是 ae,abe,ace,ade 的一个缩写。
2. 涉及到执行命令的代码框，如果行首有 $，那么后面表示的是执行的命令；否则，表示的是命令的输出。

实验环境准备

1. 需要是 x86_64 架构的 Linux 系统
2. 如果是 Windows 建议安装 WSL2，在 WSL 内进行实验
3. 如果是 macOS，请 SSH 到 Linux 机器上进行实验
4. 首先安装 gcc 和 gdb 和 objdump，以 Debian 为例子：`sudo apt install gcc gdb binutils`
5. 检验 gdb 是否安装了：`which gdb` 应该会输出 gdb 的路径

C/C++ 代码编译流程

一段 C/C++ 代码编译成二进制的可执行文件，需要下面四个步骤：

1. 预处理（Preprocess）：处理 #include 和宏
2. 编译（Compile）：将 C 代码编译成汇编代码
3. 汇编（Assemble）：将汇编代码编译成指令，保存在对象文件中
4. 链接（Link）：将对象文件和标准库链接为二进制可执行文件

实际编译的时候，调用编译器时，根据命令行参数的不同，会执行上面的部分或者所有步骤。
常用的 C/C++ 编译器有：GCC、Clang、MSVC、ICC。本讲以 GCC 为例子。

GCC 使用

如果要在 GCC 里面编译一个程序，最简单的办法就是 `gcc a.c -o a`，但也可以一步一步地进行四个步骤的生成：

1. 预处理：`gcc -E a.c -o a.i`
2. 编译：`gcc -S a.i -o a.s`
3. 汇编：`gcc -c a.s -o a.o`
4. 链接：`gcc a.o -o a`

举个例子

比如，我们编写一个简单的程序如下，保存为 a.c：

```c
#define RET 0
int main(int argc, char *argv[]) {
  return RET;
}
```

举个例子

首先进行预处理 `gcc -E a.c -o a.i`，可以得到这样的内容：

```
1  # 1 "a.c"
2  # 1 "<built-in>"
3  # 1 "<command-line>"
4  # 31 "<command-line>"
5  # 1 "/usr/include/stdc-predef.h" 1 3 4
6  # 32 "<command-line>" 2
7  # 1 "a.c"
8
9  int main(int argc, char *argv[]) {
10  return 0;
11 }
```

可以看到代码中的宏已经展开。如果 #include 了头文件，头文件的内容也会出现在这里，同学们可以自己进行实验。

举个例子

接着，编译代码到汇编 `gcc -S a.i -o a.s`，可以得到这样的内容，省略了一些注释：

```
1           .text
2           .globl  main
3   main:
4           pushq   %rbp
5           movq    %rsp, %rbp
6           movl    %edi, -4(%rbp)
7           movq    %rsi, -16(%rbp)
8           movl    $0, %eax
9           popq    %rbp
10          ret
```

我们编写的代码编译成了 AT&T 风格的 x86_64 汇编；可以看到，第 4-7 行维护了栈指针，并且把前两个参数（%edi %rsi）保存在栈上，第 9-10 行设置返回值（%eax）为 0，然后返回。

举个例子

得到汇编以后，需要汇编为二进制的指令，并保存在对象文件中：`gcc -c a.s -o a.o`。可以用 `file a.o` 来查看 a.o 文件的格式，可以得到：

```
$ gcc -c a.s -o a.o
$ file a.o
a.o: ELF 64-bit LSB relocatable, x86-64, version 1
 ↪  (SYSV), not stripped
```

接着，链接为可执行程序：

```
$ gcc a.o -o a
$ file a
a: ELF 64-bit LSB pie executable, x86-64, version 1
 ↪  (SYSV), dynamically linked, interpreter
 ↪  /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0,
 ↪  not stripped
```

小贴士：Windows 下可执行文件后缀一般是 exe/msi，其他平台下可执行文件没有后缀。

GCC 总结

总结一下 GCC 的一些参数用法，也给出了其他一些常用的参数：

-o filename	指定输出的文件名
-S	输出汇编代码
-c	编译到对象文件（object file，.o/.obj）
-E	对代码进行预处理
-g	打开调试符号
-Werror	把所有的 warning 当成 error
-O{,1,2,3,s,fast,g}	设置优化选项

为了调试方便，一般需要打开 -g 选项，否则调试的时候可能会遇到无法设置断点，或者在断点无法看到变量信息等问题。

objdump 使用

我们需要使用 objdump 工具来查看编译出来的汇编，例子如下：

```
$ objdump -S a.o
a.o:     file format elf64-x86-64
Disassembly of section .text:
0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   89 7d fc                mov    %edi,-0x4(%rbp)
   7:   48 89 75 f0             mov    %rsi,-0x10(%rbp)
   b:   b8 00 00 00 00          mov    $0x0,%eax
  10:   5d                      pop    %rbp
  11:   c3                      retq
```

可以看到，我们定义的 main 函数被编译成了上面的 7 条指令，这个内容与我们之前看到的 a.s 基本是一致的。区别在于，每条指令出现了地址、二进制表示，并且指令的表示也有细微的差别。

objdump 使用

刚刚只用 objdump 查看了 a.o 对应的汇编，我们再看看 a 可执行文件的汇编：

```
$ objdump -S a
a:     file format elf64-x86-64
Disassembly of section .init:
0000000000001000 <_init>:
    1000:  48 83 ec 08             sub    $0x8,%rsp
    1004:  48 8b 05 dd 2f 00 00    mov
    ↪  0x2fdd(%rip),%rax  # 3fe8 <__gmon_start__>
    100b:  48 85 c0                test   %rax,%rax
...
```

因为输出比较长，省略了一些部分输出。可以看到，这里出现了很多没有见过的代码，这些代码来自于 libc，在链接的时候，a.o 和 libc 的一些代码共同组成了 a 的二进制代码。

objdump 使用

如果我们继续往下找 main 函数：

```
0000000000001125 <main>:
  1125:   55                      push   %rbp
  1126:   48 89 e5                mov    %rsp,%rbp
...
  1135:   5d                      pop    %rbp
  1136:   c3                      retq
  1137:   66 0f 1f 84 00 00 00    nopw
  ↪  0x0(%rax,%rax,1)
  113e:   00 00
```

可以发现 main 函数的地址变了，因为前面出现了其他函数；另外，可以看到 return 之后还有 nop 指令，这是为了让下一个函数的起始地址对齐到某个数（比如 16）的整数倍上。

objdump 与 GCC 关系

objdump 常用参数

objdump 除了上面看到的 -S 以外，还有一些常用的参数：

-d 反汇编可执行的段（如.text）

-D 反汇编所有的段，即使这个段存储的是数据，也当成指令来解析

-S 把反汇编的代码和调试信息里的代码信息合在一起显示

-t 显示函数的符号列表

--adjust-vma OFFSET 把反汇编出来的地址都加上一个偏移，常用于嵌入式开发

-M {intel,att} 用 Intel 或者 AT&T 风格显示 X86 汇编；默认情况下用的是 AT&T 风格

如果用的是 MSVC，那么默认的 X86 汇编风格是 Intel 风格；如果用的是 objdump、gdb、gcc 等工具，一般默认的汇编风格是 AT&T。

gdb 介绍

gdb 是一个调试器，可以在运行程序的时候，设置断点，中途查看程序运行的状态，并且逐步观察程序运行行为。
我们引入下面的代码作为例子，来观察 gdb 的使用方式：

```
1  #include <stdio.h>
2  int number;
3  int main(int argc, char *argv[]) {
4    printf("%d %s\n", argc, argv[0]);
5    scanf("%d", &number);
6    printf("%d\n", number);
7    return 0;
8  }
```

按照前面讲述的方法，编译成二进制：`g++ test.cpp -o test`

gdb 介绍

用 gdb 调试 test 程序，首先运行 gdb test，然后输入 run 命令开始运行：

```
$ gdb test
GNU gdb (Debian 10.1-1.7) 10.1.90.20210103-git
...
Reading symbols from test...
(No debugging symbols found in test)
(gdb) run
Starting program: /home/jiegec/test
1 /home/jiegec/test
1234
1234
[Inferior 1 (process 1773151) exited normally]
(gdb)
```

程序运行以后，输出了 argc argv，这时候可以正常向程序输入数据。程序退出以后，回到 gdb 的命令。接下来，我们要设置断点，来观察程序的行为。

gdb 介绍

首先，我们可以用 b（全称 breakpoint）来设置一个断点，比如这里给 main 函数设置一个断点。

```
(gdb) b main
Breakpoint 1 at 0x555555555149
(gdb) run
Starting program: /home/jiegec/test
Breakpoint 1, 0x000055555555149 in main ()
(gdb) c
Continuing.
1 /home/jiegec/test
1234
1234
[Inferior 1 (process 1775170) exited normally]
(gdb)
```

可以看到，程序在 main 函数的开头暂停了运行，回到了 gdb，并且可以看到当前运行的指令地址和函数。但由于我们在编译的时候没有打开调试信息，输入 c（全称 continue）命令让它继续运行到结束。

gdb 介绍

那么，我们打开调试选项重新编译一次代码 `g++ -g test.cpp -o test`，可以发现 gdb 找到了调试符号：

```
$ gdb test
Reading symbols from test...
(gdb)
```

这时候再设置断点，并且运行，就可以看到程序停在了断点所在的地方。

```
(gdb) b main
Breakpoint 1 at 0x1154: file test.cpp, line 4.
(gdb) run
Starting program: /home/jiegec/test
Breakpoint 1, main (argc=1, argv=0x7fffffffeaf8) at
↪  test.cpp:4
4                   printf("%d %s\n", argc, argv[0]);
(gdb)
```

gdb 介绍

我们可以用 n 命令来执行到下一行代码，也可以用 p 命令来打印出变量的值：

```
(gdb) n
1 /home/jiegec/test
5                   scanf("%d", &number);
(gdb) p argc
$1 = 1
(gdb) p argv
$2 = (char **) 0x7fffffffeaf8
(gdb) p argv[0]
$3 = 0x7fffffffed3d "/home/jiegec/test"
(gdb) p argv[1]
$4 = 0x0
```

gdb 介绍

继续运行，scanf 结束后，可以看到 number 的值和我们输入是一致的：

```
(gdb) n
12345
6                   printf("%d\n", number);
(gdb) p number
$5 = 12345
(gdb) n
12345
7                   return 0;
(gdb)
```

gdb 介绍

上面几个基础的命令已经可以实现很多程序的调试，下面再总结一下常用的 gdb 命令：

run	运行程序，后面可以跟上运行程序的命令行参数
b func/file:line/*addr	设置断点
s	如果当前行是函数调用，进入函数调用；否则执行当前行代码，进入下一行
n	执行当前行代码，进入下一行
si,ni	和上面两个命令的不同是，它的粒度是指令
p var/$reg	输出代码中的变量或者寄存器
info registers	输出所有寄存器的值
disas func/addr/$reg	输出目标函数/地址的汇编
disas /m func/addr/$reg	输出目标函数/地址的汇编和源代码
x/5i func/addr/$reg	输出目标函数/地址的前 5 条汇编
layout src	显示源代码窗口
layout asm	显示汇编窗口