中国科学技术大学计算机学院

《嵌入式系统设计方法》

实验题目： LAB2 交叉编译环境配置

学生姓名：钟书锐

学生学号：PB19000362

完成日期：2021.11.15

嵌入式系统设计方法-LAB2-交叉工具链

- PB19000362
- 钟书锐

一、实验要求

1. 使用 step-by-step 的模式，编译一个你自己的 arm-linux-gcc 编译器，
2. 修改 gcc 的代码，使得 gcc -v 的输出中包含个人的信息。
3. 使用 C 代码测试编译器
4. 使用 gcc 和 arm-gcc 编译，比较生成的目标代码的区别。需要使用 readelf 和 objdump 等工具。重点分析文件头部，分段等信息

二、实验环境

- 处理器 Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz 2.59 GHz
- VMware® Workstation 15 Pro (15.5.6 build-16341506)
- Linux ubuntu 5.11.0-40-generic #44~20.04.2-Ubuntu

三、实验步骤

1.step-by-step 的模式，编译 arm-linux-gcc 编译器

- 1.安装必要的工具 $ sudo apt-get install g++ make gawk
- 2.下载 解压所需的各种程序的源代码

```
$ wget http://ftpmirror.gnu.org/binutils/binutils-2.24.tar.gz
$ wget http://ftpmirror.gnu.org/gcc/gcc-4.9.2/gcc-4.9.2.tar.gz
$ wget https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.17.2.tar.xz
$ wget http://ftpmirror.gnu.org/glibc/glibc-2.20.tar.xz
$ wget http://ftpmirror.gnu.org/mpfr/mpfr-3.1.2.tar.xz
$ wget http://ftpmirror.gnu.org/gmp/gmp-6.0.0a.tar.xz
$ wget http://ftpmirror.gnu.org/mpc/mpc-1.0.2.tar.gz
$ for f in *.tar*; do tar xf $f; done
```

- 3.硬连接

```
$ cd gcc-4.9.2
$ ln -s ../mpfr-3.1.2 mpfr
$ ln -s ../gmp-6.0.0 gmp
$ ln -s ../mpc-1.0.2 mpc
$ cd ../
```

- 4.安装目录

```
$ sudo mkdir -p /opt/cross
$ sudo chown zsr /opt/cross
$ export PATH=/opt/cross/bin:$PATH
```

- 5.Binutils

```
$ mkdir build-binutils
$ cd build-binutils
$ ../binutils-2.24/configure --prefix=/opt/cross --target=aarch64-linux --disable-multilib
$ make -j4
$ make install
$ cd ..
```

- 6.Linux Kernel Headers

```
$ cd linux-3.17.2
$ make ARCH=arm64 INSTALL_HDR_PATH=/opt/cross/aarch64-linux headers_install
$ cd ..
```

- 7.gcc 第一次编译

```
$ mkdir -p build-gcc
$ cd build-gcc
$ ../gcc-4.9.2/configure --prefix=/opt/cross --target=aarch64-linux --enable-languages=c,c++ --disable-multilib
$ make -j4 all-gcc
$ make install-gcc
$ cd ..
```

- 8.glibc 编译

```
$ mkdir -p build-glibc
$ cd build-glibc
$ ../glibc-2.20/configure --prefix=/opt/cross/aarch64-linux --build=$MACHTYPE --host=aarch64-linux --target=aarch64-l
```

```
$ make install-bootstrap-headers=yes install-headers
$ make -j4 csu/subdir_lib
$ install csu/crt1.o csu/crti.o csu/crtn.o /opt/cross/aarch64-linux/lib
$ aarch64-linux-gcc -nostdlib -nostartfiles -shared -x c /dev/null -o /opt/cross/aarch64-linux/lib/libc.so
$ touch /opt/cross/aarch64-linux/include/gnu/stubs.h
$ cd ..
```

- 9.使用编译好的 glibc 第二次编译 gcc

```
$ cd build-gcc
$ make -j4 all-target-libgcc
$ make install-target-libgcc
$ cd ..
```

- 10.C 标准库编译

```
$ cd build-glibc
$ make -j4
$ make install
$ cd ..
```

- 11. 第三次编译 gcc

```
cd build-gcc
make -j 4
make install
cd …
```

2.修改 gcc 的代码，使得 gcc –v 的输出中包含个人的信息。

- 定位到文件 gcc-4.9.2/gcc/gcc.c
- 定位到下面部分

```
if (!strncmp(version_string, compiler_version, n) && compiler_version[n] == 0)
      fnotice(stderr, "gcc version %s %s\n", version_string,
              pkgversion_string);
    else
      fnotice(stderr, "gcc driver version %s %sexecuting gcc version %s\n",
              version_string, pkgversion_string, compiler_version);
```

- 在后面添加代码fnotice(stderr, "This gcc is zsr's compiler\n");

3.使用 C 代码测试编译器,使用 gcc 和 arm-gcc 编译。

- 源代码如下

```
#include <stdio.h>
int main()
{
    printf("Hello World!");
    return 0;
}
```

- 运行 gcc 输出目标文件

```
gcc -c 1.c -o 1.o
arm-linux-gnueabihf-gcc -c 1.c -o 2.o
```

4.比较生成的目标代码的区别。需要使用 readelf 和 objdump 等工具，重点分析文件头部，分段等信息

- readlf 比较 Header 部分 readelf -h 1.o > head1.readelf readelf -h 2.o > head2.readelf
- Header 中主要存放的是一些基本信息

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          640 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           64 (bytes)
```

```
  Number of section headers:         14
  Section header string table index: 13

ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              REL (Relocatable file)
  Machine:                           ARM
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          544 (bytes into file)
  Flags:                             0x5000000, Version5 EABI
  Size of this header:               52 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           40 (bytes)
  Number of section headers:         12
  Section header string table index: 11
```

- 主要区别

  1. Machine 分别是 Advanced Micro Devices X86-64 和 ARM
  2. Start of section headers 分别是 640 (bytesinto file)和 544 (bytesinto file)
  3. Size of this header 分别是 64 (bytes)和 52 (bytes)
  4. Size of section headers 分别是 64 (bytes) 和 40 (bytes)
  5. Number of section headers 分别是 14 和 12
  6. Section header string table index 分别是 13 和 11

- readlf 比较 Section 部分 readelf -S -W 1.o > sec1.readlf readelf -S -W 2.o > sec2.readlf
- Section 主要存放的是机器指令代码和数据
- 通常我们比较的 Section 是.text（存放代码）、.data（存放全局静态变量和局部静态变量）和.bss（存未初始化的全局变量和局部静态变量）

```
There are 14 section headers, starting at offset 0x280:

Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        0000000000000000 000040 000020 00  AX  0   0  1
  [ 2] .rela.text        RELA            0000000000000000 0001c0 000030 18   I 11   1  8
  [ 3] .data             PROGBITS        0000000000000000 000060 000000 00  WA  0   0  1
  [ 4] .bss              NOBITS          0000000000000000 000060 000000 00  WA  0   0  1
  [ 5] .rodata           PROGBITS        0000000000000000 000060 00000d 00   A  0   0  1
  [ 6] .comment          PROGBITS        0000000000000000 00006d 00002b 01  MS  0   0  1
  [ 7] .note.GNU-stack   PROGBITS        0000000000000000 000098 000000 00      0   0  1
  [ 8] .note.gnu.property NOTE            0000000000000000 000098 000020 00   A  0   0  8
  [ 9] .eh_frame         PROGBITS        0000000000000000 0000b8 000038 00   A  0   0  8
  [10] .rela.eh_frame    RELA            0000000000000000 0001f0 000018 18   I 11   9  8
  [11] .symtab           SYMTAB          0000000000000000 0000f0 0000a8 18     12   4  8
  [12] .strtab           STRTAB          0000000000000000 000198 000027 00      0   0  1
  [13] .shstrtab         STRTAB          0000000000000000 000208 000074 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  l (large), p (processor specific)


There are 12 section headers, starting at offset 0x220:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 000018 00  AX  0   0  4
  [ 2] .rel.text         REL             00000000 0001ac 000010 08   I  9   1  4
  [ 3] .data             PROGBITS        00000000 00004c 000000 00  WA  0   0  1
  [ 4] .bss              NOBITS          00000000 00004c 000000 00  WA  0   0  1
  [ 5] .rodata           PROGBITS        00000000 00004c 00000d 00   A  0   0  4
  [ 6] .comment          PROGBITS        00000000 000059 000026 01  MS  0   0  1
  [ 7] .note.GNU-stack   PROGBITS        00000000 00007f 000000 00      0   0  1
  [ 8] .ARM.attributes   ARM_ATTRIBUTES  00000000 00007f 000033 00      0   0  1
  [ 9] .symtab           SYMTAB          00000000 0000b4 0000e0 10     10  12  4
  [10] .strtab           STRTAB          00000000 000194 000017 00      0   0  1
  [11] .shstrtab         STRTAB          00000000 0001bc 000061 00      0   0  1
Key to Flags:
```

```
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  y (purecode), p (processor specific)
```

- 主要区别

  1. - gcc:There are 14 section headers, starting at offset 0x280:
     - sarm-linux-gnueabihf-gcc:There are 12 section headers, starting at offset 0x220: 开始的位置不同

  2. Address 的位数不同，本地 linux 是 64 位的，交叉编译结果是 32 位的

  3. - gcc:.rela.text
     - sarm-linux-gnueabihf-gcc:.rel.text

- objdump 比较 Section 部分 objdump -s 2.o > sec2.objdum objdump -s 2.o > sec2.objdum

```
2.o:     file format elf32-little

Contents of section .text:
 0000 80b500af 034b7b44 1846fff7 feff0023  .....K{D.F.....#
 0010 184680bd 0a000000                    .F......
Contents of section .rodata:
 0000 48656c6c 6f20576f 726c6421 00        Hello World!.
Contents of section .comment:
 0000 00474343 3a202855 62756e74 75203130  .GCC: (Ubuntu 10
 0010 2e332e30 2d317562 756e7475 31292031  .3.0-1ubuntu1) 1
 0020 302e332e 3000                        0.3.0.
Contents of section .ARM.attributes:
 0000 41320000 00616561 62690001 28000000  A2...aeabi..(...
 0010 05372d41 00060a07 41080109 020a0412  .7-A....A.......
 0020 04140115 01170318 0119011a 021c011e  ................
 0030 062201                               ."
```

```
1.o:     file format elf64-x86-64

Contents of section .text:
 0000 f30f1efa 554889e5 488d3d00 000000b8  ....UH..H.=.....
 0010 00000000 e8000000 00b80000 00005dc3  ..............].
Contents of section .rodata:
 0000 48656c6c 6f20576f 726c6421 00        Hello World!.
Contents of section .comment:
 0000 00474343 3a202855 62756e74 7520392e  .GCC: (Ubuntu 9.
 0010 332e302d 31377562 756e7475 317e3230  3.0-17ubuntu1~20
 0020 2e303429 20392e33 2e3000             .04) 9.3.0.
Contents of section .note.gnu.property:
 0000 04000000 10000000 05000000 474e5500  ............GNU.
 0010 020000c0 04000000 03000000 00000000  ................
Contents of section .eh_frame:
 0000 14000000 00000000 017a5200 01781001  .........zR..x..
 0010 1b0c0708 90010000 1c000000 1c000000  ................
 0020 00000000 20000000 00450e10 8602430d  .... ....E....C.
 0030 06570c07 08000000                    .W......
```

- 主要区别

  1. .text 段是程序代码段 2 者区别较大主要是生成的汇编代码差异也较大
  2. .rodata 段 ro 代表 read only，即只读数据(const)，这里是字符串常量" Hello World!"，2 种方法此处没区别
  3. .comment 段是注释信息段，存放的是编译器版本等信息，此处分别是.GCC: (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0. 和 .GCC: (Ubuntu 10.3.0-1ubuntu1) 10.3.0.
  4. 剩下段用处不大，不做分析

五、反思与总结

- 难点主要在附加的 step by step 编译 和 如何修改以输出个人信息
- 实验本体难点在对于 objdump 和 readlf 的输出进行理解