嵌入式系统设计实验报告

实验二

使用交叉编译工具链生成目标代码并分析

学　　号：**PE20060014**

姓　　名：王晨

专　　业：计算机科学与技术

指导老师：张辉

2020 年 11 月 26 日

一、 实验要求

- 熟悉使用交叉编译工具链，基于提供的 hello1.s 和 hello2.s，分别使用 as+ld 和 gcc 生成对应目标代码，并下载到开发板执行
- 使用 objdump 工具分析比较两种编译链接模式所生成的文件的区别。

二、 实验条件

1、 硬件条件：Macbook pro

2、 软件条件：Mac OS Big Sur

VMware fusion pro 虚拟机

Ubuntu 20.04.1 64 位

三、 实验过程

1、使用 as+ld 生成目标代码

1）编写 hello1.s 汇编代码如下：

Open hello1.s ~/Desktop/exp2

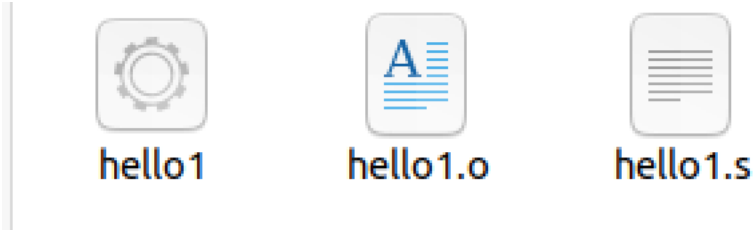
```
.data
    msg:    .asciz  "hello,world\n"
    len = . -msg
.text
        .align  2
.global _start
_start:
    mov     r0, #1
    ldr     r1, =msg
    ldr     r2, =len
    mov     r7, #4
    swi     #0

    mov     r0, #0
    mov     r7, #1
    swi     #0
.end
```

2）执行编译：arm-none-linux-gnueabi-as -o hello1.o hello1.s

3）执行链接：arm-none-linux-gnueabi-ld -o hello1 hello1.o

4）完成后得到如下 arm 可执行文件 hello1：



2、使用 gcc 生成目标代码

1） 使用 gcc 编译 hello1 会看到报错，这是因为 gcc 需要入口函数 main：

```
chen@ubuntu: ~/Desktop/exp2
chen@ubuntu:~/Desktop/exp2$ arm-none-linux-gnueabi-gcc hello1.s -o hello1
/tmp/cc1glvak.o: In function `_start':
(.text+0x0): multiple definition of `_start'
/usr/local/arm/gcc-4.6.4/bin/../arm-arm1176jzfssf-linux-gnueabi/sysroot/usr/lib/
crt1.o::(.text+0x0): first defined here
/usr/local/arm/gcc-4.6.4/bin/../arm-arm1176jzfssf-linux-gnueabi/sysroot/usr/lib/
crt1.o: In function `_start':
:(.text+0x34): undefined reference to `main'
collect2: ld returned 1 exit status
chen@ubuntu:~/Desktop/exp2$
```

2） 修改代码如下，保存为 hello2.s：

```
Open
hello2.s
~/Desktop/exp2
 1 .data
 2     msg:    .asciz  "hello,world\n"
 3     len = . -msg
 4 .text
 5         .align  2
 6 .global main          /*for gcc*/
 7 main:
 8     mov     r0, #1
 9     ldr     r1, =msg
10     ldr     r2, =len
11     mov     r7, #4
12     swi     #0
13
14     mov     r0, #0
15     mov     r7, #1
16     swi     #0
17 .end
18
```

3） 执行编译：arm-none-linux-gnueabi-gcc hello2.s -o hello2

4） 完成后得到 arm 可执行文件 hello2：

3、将可执行文件下载到开发板执行

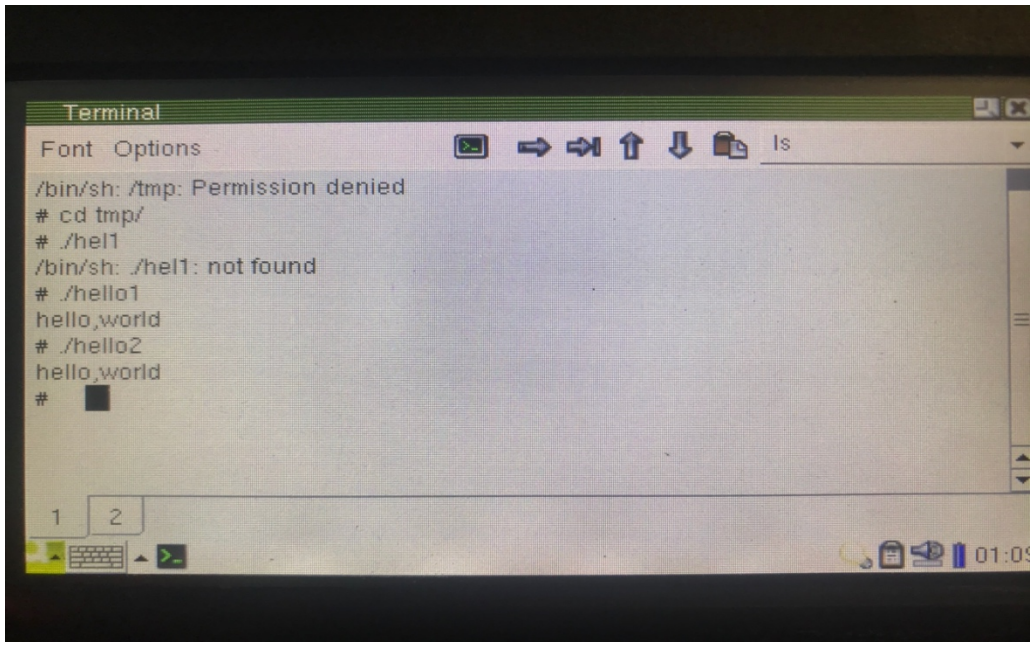
1） 将 hello1,hello2 下载到/tmp 目录下，在终端或串口输入：

cd /tmp

./hello1

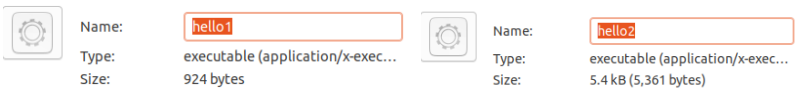
./hello2

2） 在终端界面上可以看到下图，运行成功：



4、分析两种方法生成可执行文件的差异：

1） 首先可以直观的看到，通过 as+ld 方式生成的可执行文件 hello1 的文件大小为 924Byte，而通过 gcc 方式生成的可执行文件 hello2 的文件大小为 5.4KB。执行同样效果的简单程序，gcc 方式生成代码大小是 as+ld 方式的 5 倍之多，差异非常大，因此在嵌入式设备上往往应该采用 as+ld 方式生成可执行文件，因为嵌入式设备往往对容量的大小要求很高。



2） 使用 objdump 工具分析比较两种编译链接模式所生成的文件的区别。

对于 hello1 文件反汇编可以看到生成的反汇编代码基本就是原来写的 hello1.s 的主程序代码段，非常简洁：

```
chen@ubuntu:~/Desktop/exp2$ arm-none-linux-gnueabi-objdump -d hello1

hello1:     file format elf32-littlearm


Disassembly of section .text:

00008074 <_start>:
    8074:       e3a00001        mov     r0, #1
    8078:       e59f1014        ldr     r1, [pc, #20]   ; 8094 <_start+0x20>
    807c:       e3a0200d        mov     r2, #13
    8080:       e3a07004        mov     r7, #4
    8084:       ef000000        svc     0x00000000
    8088:       e3a00000        mov     r0, #0
    808c:       e3a07001        mov     r7, #1
    8090:       ef000000        svc     0x00000000
    8094:       00010098        .word   0x00010098
chen@ubuntu:~/Desktop/exp2$
```

而对于 hello2 文件反汇编可以看到生成的反汇编代码则非常冗长，除了_start 主程序段还有其他的字段，因此导致文件大小非常大：

```
chen@ubuntu:~/Desktop/exp2$ arm-none-linux-gnueabi-objdump -d hello2

hello2:     file format elf32-littlearm


Disassembly of section .init:

00008250 <_init>:
    8250:       e92d4008        push    {r3, lr}
    8254:       eb00001d        bl      82d0 <call_weak_fn>
    8258:       e8bd8008        pop     {r3, pc}

Disassembly of section .plt:

0000825c <.plt>:
    825c:       e52de004        push    {lr}            ; (str lr, [sp, #-4]!)
    8260:       e59fe004        ldr     lr, [pc, #4]    ; 826c <_init+0x1c>
    8264:       e08fe00e        add     lr, pc, lr
    8268:       e5bef008        ldr     pc, [lr, #8]!
    826c:       00008268        .word   0x00008268
    8270:       e28fc600        add     ip, pc, #0, 12
    8274:       e28cca08        add     ip, ip, #8, 20  ; 0x8000
    8278:       e5bcf268        ldr     pc, [ip, #616]! ; 0x268
    827c:       e28fc600        add     ip, pc, #0, 12
    8280:       e28cca08        add     ip, ip, #8, 20  ; 0x8000
    8284:       e5bcf260        ldr     pc, [ip, #608]! ; 0x260
    8288:       e28fc600        add     ip, pc, #0, 12
    828c:       e28cca08        add     ip, ip, #8, 20  ; 0x8000
    8290:       e5bcf258        ldr     pc, [ip, #600]! ; 0x258

Disassembly of section .text:

00008294 <_start>:
```

```
00008294 <_start>:
    8294:       e3a0b000        mov     fp, #0
    8298:       e3a0e000        mov     lr, #0
    829c:       e49d1004        pop     {r1}            ; (ldr r1, [sp], #4)
    82a0:       e1a0200d        mov     r2, sp
    82a4:       e52d2004        push    {r2}            ; (str r2, [sp, #-4]!)
    82a8:       e52d0004        push    {r0}            ; (str r0, [sp, #-4]!)
    82ac:       e59fc010        ldr     ip, [pc, #16]   ; 82c4 <_start+0x30>
    82b0:       e52dc004        push    {ip}            ; (str ip, [sp, #-4]!)
    82b4:       e59f000c        ldr     r0, [pc, #12]   ; 82c8 <_start+0x34>
    82b8:       e59f300c        ldr     r3, [pc, #12]   ; 82cc <_start+0x38>
    82bc:       ebffffeb        bl      8270 <_init+0x20>
    82c0:       ebffffe0        bl      8288 <_init+0x38>
    82c4:       000083c4        .word   0x000083c4
    82c8:       00008340        .word   0x00008340
    82cc:       00008364        .word   0x00008364

000082d0 <call_weak_fn>:
    82d0:       e59f3014        ldr     r3, [pc, #20]   ; 82ec <call_weak_fn+0x1c>
    82d4:       e59f2014        ldr     r2, [pc, #20]   ; 82f0 <call_weak_fn+0x20>
    82d8:       e08f3003        add     r3, pc, r3
    82dc:       e7932002        ldr     r2, [r3, r2]
    82e0:       e3520000        cmp     r2, #0
    82e4:       012fff1e        bxeq    lr
    82e8:       eaffffe3        b       827c <_init+0x2c>
    82ec:       000081f4        .word   0x000081f4
    82f0:       00000018        .word   0x00000018

000082f4 <__do_global_dtors_aux>:
    82f4:       e59f3010        ldr     r3, [pc, #16]   ; 830c <__do_global_dtors_aux+0x18>
    82f8:       e5d32000        ldrb    r2, [r3]
    82fc:       e3520000        cmp     r2, #0
    8300:       03a02001        moveq   r2, #1
    8304:       05c32000        strbeq  r2, [r3]
    8308:       e12fff1e        bx      lr
    830c:       00010505        .word   0x00010505
```

3） 使用 readelf 工具分析比较两种编译链接模式所生成的文件的区别。

可以看到使用 gcc 编译生成的 hello2 的 start of section headers 的字节数非常多，这也反应了在 objdump 中的代码量差异和直观上文件大小的差异：

```
chen@ubuntu:~/Desktop/exp2$ arm-none-linux-gnueabi-readelf -h hello1
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:                              EXEC (Executable file)
  Machine:                           ARM
  Version:                           0x1
  Entry point address:               0x8074
  Start of program headers:          52 (bytes into file)
  Start of section headers:          240 (bytes into file)
  Flags:                             0x5000002, has entry point, Version5 EABI
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         2
  Size of section headers:           40 (bytes)
  Number of section headers:         7
  Section header string table index: 4
chen@ubuntu:~/Desktop/exp2$ arm-none-linux-gnueabi-readelf -h hello2
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:                              EXEC (Executable file)
  Machine:                           ARM
  Version:                           0x1
  Entry point address:               0x8294
  Start of program headers:          52 (bytes into file)
  Start of section headers:          1640 (bytes into file)
  Flags:                             0x5000002, has entry point, Version5 EABI
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
```

四、 实验心得

1. 实验中可以看到 as+ld 生成的可执行文件比较简洁，文件大小比较小，而 gcc 生成的可执行文件往往比较大，这可能是因为使用 gcc 编译链接有一些中间步骤的转化代码，比如在 objdump 可以看到进入主程序前和结束主程序后还有一些准备和收尾代码，导致了可执行文件的冗长。因此在嵌入式开发中，为了满足轻量化的代码需求，应当尽量使用 as+ld 的方式直接生成可执行文件。