

操作系统第二次实验报告-Lab1

以下为学生闫思桥(19241091)的Lab1实验报告

Part1 实验思考题

Thinking 1.1

请查阅并给出前述objdump 中使用的参数的含义。使用其它体系结构的编译器（如课程平台的MIPS交叉编译器）重复上述各步编译过程，观察并在实验报告中提交相应结果。

MyAnswer

首先是objdump中使用参数的含义

完整的参数含义地址在这篇博客中有提及——[linux objdump 反汇编命令](#)

常见的objdump参数含义如下：

- 1) objdump -d：反汇编目标文件中包含的可执行指令。
- 2) 如果需要混合显示源码和汇编代码，需要加上-s选项，并且在编译目标文件时加上-g。
- 3) 如果在编译目标文件时没有加-g选项，则-s相当于-d。
- 4) -s选项生成的混合代码，有时文件结构混乱，可读性较差。推荐使用-d选项，直接阅读汇编代码。

在指导书中一共用了以下几条指令：

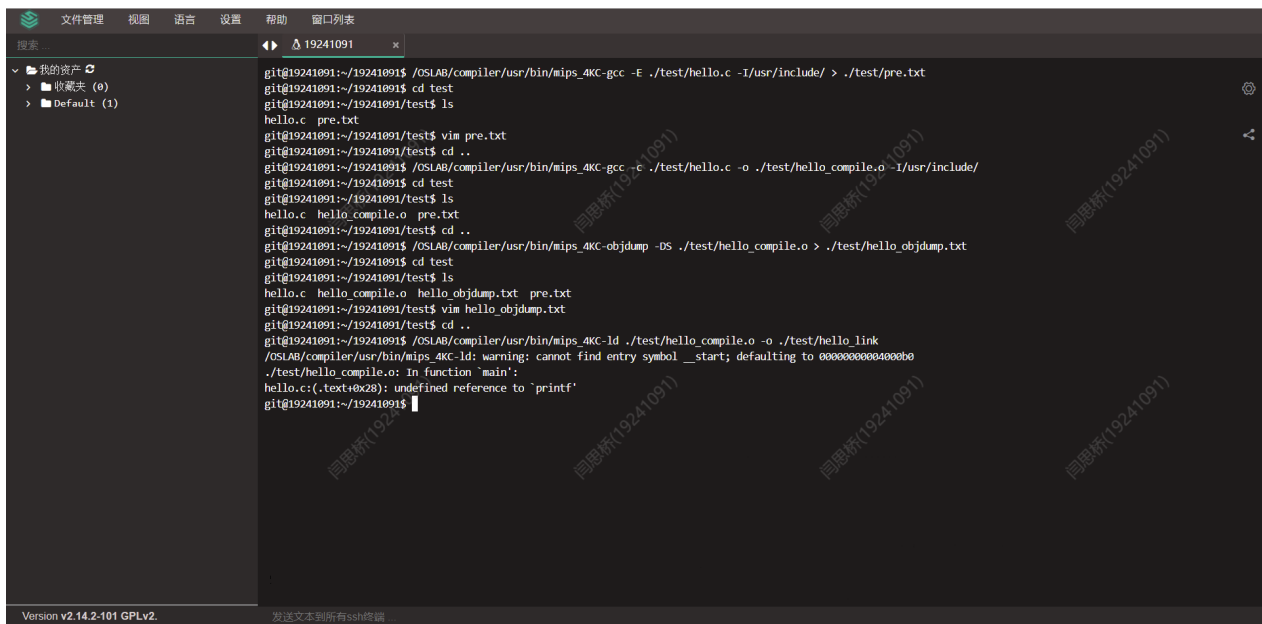
```
objdump -DS 要反汇编的目标文件名> 导出文本文件名
```

```
objdump -DS 输出可执行文件名> 导出文本文件名
```

参数都是-DS，其对应的含义是反汇编目标文件objfile中的所有section，并混合显示源码和汇编指令对应的机器码。

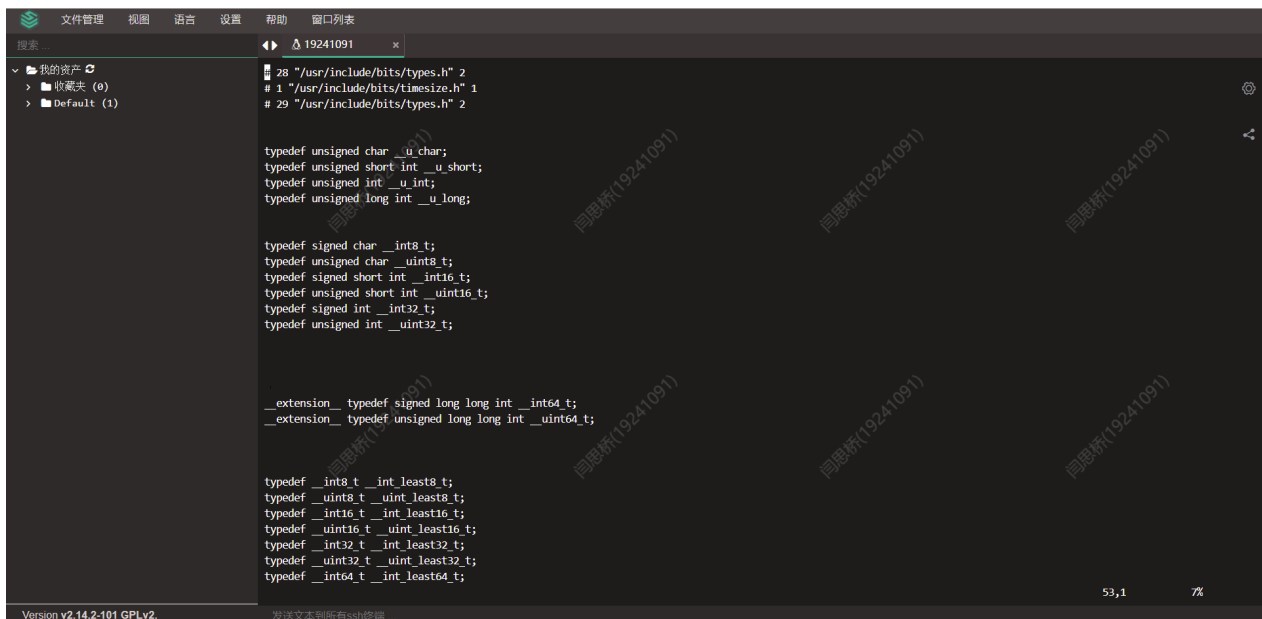
下面是要求的交叉汇编和反汇编练习：

总的命令行



```
git@19241091:~/19241091$ /OSLAB/compiler/usr/bin/mips_4KC-gcc -E ./test/hello.c -I/usr/include/ > ./test/pre.txt
git@19241091:~/19241091$ cd test
git@19241091:~/19241091/test$ ls
hello.c  pre.txt
git@19241091:~/19241091/test$ vim pre.txt
git@19241091:~/19241091/test$ cd ..
git@19241091:~/19241091$ /OSLAB/compiler/usr/bin/mips_4KC-gcc -c ./test/hello.c -o ./test/hello_compile.o -I/usr/include/
git@19241091:~/19241091/test$ ls
hello.c  hello_compile.o  pre.txt
git@19241091:~/19241091/test$ cd ..
git@19241091:~/19241091$ /OSLAB/compiler/usr/bin/mips_4KC-objdump -DS ./test/hello_compile.o > ./test/hello_objdump.txt
git@19241091:~/19241091$ cd test
git@19241091:~/19241091/test$ ls
hello.c  hello_compile.o  hello_objdump.txt  pre.txt
git@19241091:~/19241091/test$ vim hello_objdump.txt
git@19241091:~/19241091/test$ cd ..
git@19241091:~/19241091$ /OSLAB/compiler/usr/bin/mips_4KC-ld ./test/hello_compile.o -o ./test/hello_link
/OSLAB/compiler/usr/bin/mips_4KC-ld: warning: cannot find entry symbol _start; defaulting to 0000000000400000
./test/hello_compile.o: In function 'main':
hello.c:(.text+0x28): undefined reference to 'printf'
git@19241091:~/19241091/test$
```

-E 预处理后的输出文件



```
# 28 "/usr/include/bits/types.h" 2
# 1 "/usr/include/bits/timesize.h" 1
# 29 "/usr/include/bits/types.h" 2

typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
typedef unsigned long int __u_long;

typedef signed char __int8_t;
typedef unsigned char __uint8_t;
typedef signed short int __int16_t;
typedef unsigned short int __uint16_t;
typedef signed int __int32_t;
typedef unsigned int __uint32_t;

__extension__ typedef signed long long int __int64_t;
__extension__ typedef unsigned long long int __uint64_t;

typedef __int8_t __int_least8_t;
typedef __uint8_t __uint_least8_t;
typedef __int16_t __int_least16_t;
typedef __uint16_t __uint_least16_t;
typedef __int32_t __int_least32_t;
typedef __uint32_t __uint_least32_t;
typedef __int64_t __int_least64_t;

extern FILE *popen (const char *__command, const char *__modes) ;

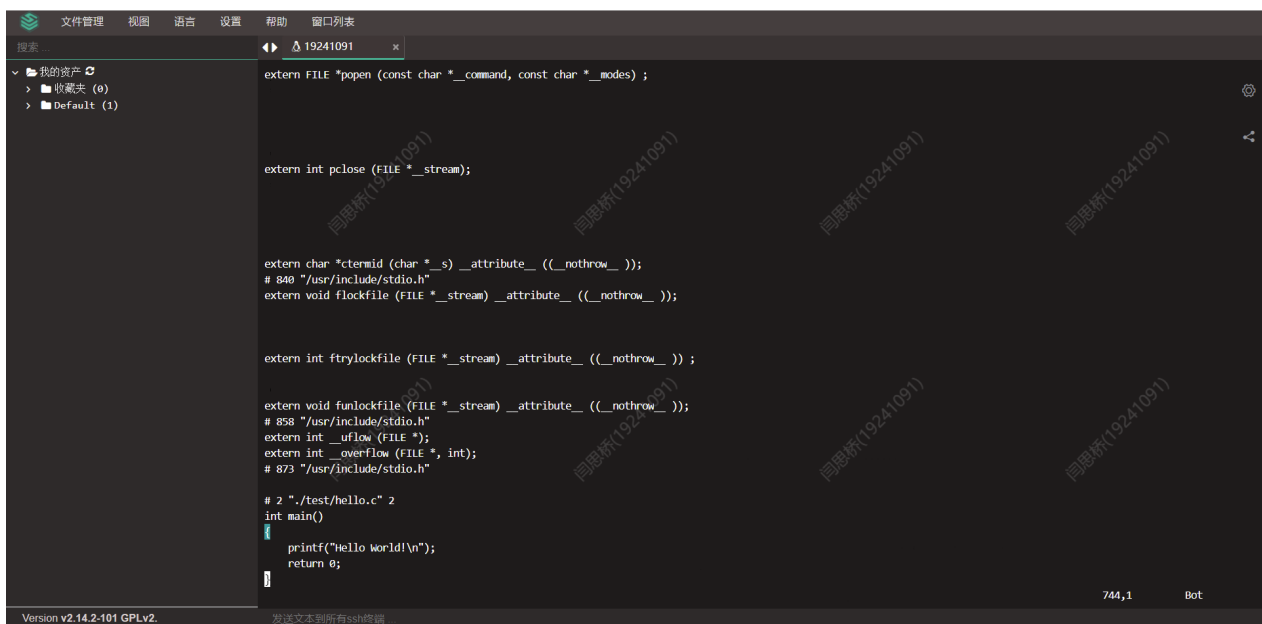
extern int pclose (FILE *__stream);

extern char *ctermid (char *__s) __attribute__ ((__nothrow__ ));
# 840 "/usr/include/stdio.h"
extern void flockfile (FILE *__stream) __attribute__ ((__nothrow__ ));

extern int trylockfile (FILE *__stream) __attribute__ ((__nothrow__ ));

extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__ ));
# 858 "/usr/include/stdio.h"
extern int _uflow (FILE *);
extern int _overflow (FILE *, int);
# 873 "/usr/include/stdio.h"

# 2 "./test/hello.c" 2
int main()
{
    printf("Hello world!\n");
    return 0;
}
```



```
extern FILE *popen (const char *__command, const char *__modes) ;

extern int pclose (FILE *__stream);

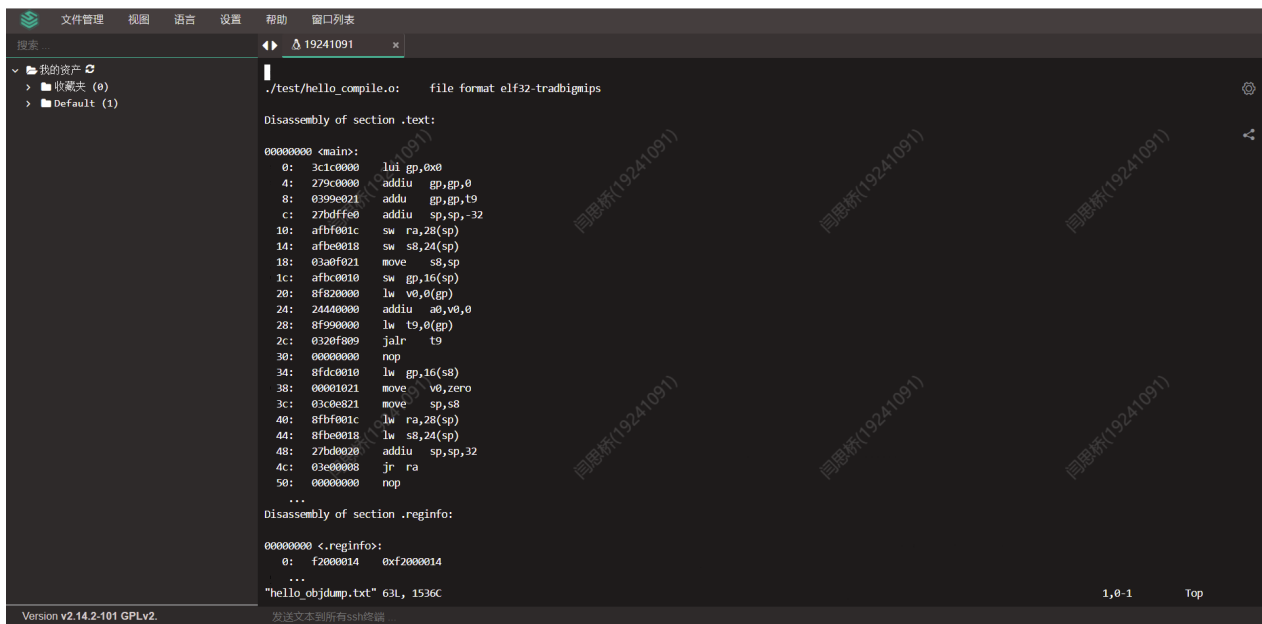
extern char *ctermid (char *__s) __attribute__ ((__nothrow__ ));
# 840 "/usr/include/stdio.h"
extern void flockfile (FILE *__stream) __attribute__ ((__nothrow__ ));

extern int trylockfile (FILE *__stream) __attribute__ ((__nothrow__ ));

extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__ ));
# 858 "/usr/include/stdio.h"
extern int _uflow (FILE *);
extern int _overflow (FILE *, int);
# 873 "/usr/include/stdio.h"

# 2 "./test/hello.c" 2
int main()
{
    printf("Hello world!\n");
    return 0;
}
```

-C 只编译不链接产生.o文件，对.o文件进行反汇编后的输出文件



ld 将编译后的.o文件链接起来，因为缺少链接脚本，会有一个warning提示找不到入口点，报错如下

```
git@19241091:~/19241091$ /OSLAB/compiler/usr/bin/mips_4KC-ld ./test/hello_compile.o -o ./test/hello_link
/OSLAB/compiler/usr/bin/mips_4KC-ld: warning: cannot find entry symbol __start; defaulting to 00000000004000b0
./test/hello_compile.o: In function `main':
hello.c:(.text+0x28): undefined reference to `printf'
```

Thinking 1.2

也许你会发现我们的readelf程序是不能解析之前生成的内核文件(内核文件是可执行文件)的，而我们之后将要介绍的工具readelf则可以解析，这是为什么呢？(提示：尝试使用readelf -h，观察不同)

MyAnswer

testELF的 readelf -h testELF 截图如下：

```
git@19241091:~/19241091/readelf$ readelf -h testELF
```

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:                             Intel 80386
Version:                              0x1
Entry point address:                 0x8048490
Start of program headers:             52 (bytes into file)
Start of section headers:            4440 (bytes into file)
Flags:                                0x0
Size of this header:                 52 (bytes)
Size of program headers:             32 (bytes)
Number of program headers:            9
Size of section headers:             40 (bytes)
Number of section headers:           30
Section header string table index: 27
```

在thinking1.1中生成的 vmlinux 的 readelf -h vmlinux 截图如下:

```
git@19241091:~/19241091/readelf$ readelf -h ../gxemul/vmlinux
```

ELF Header:

```
Magic:  7f 45 4c 46 01 02 01 00 00 00 00 00 00 00 00 00
Class:                               ELF32
Data:                                2's complement, big endian
Version:                             1 (current)
OS/ABI:                              UNIX - System V
ABI Version:                          0
Type:                                EXEC (Executable file)
Machine:                             MIPS R3000
Version:                              0x1
Entry point address:                 0x0
Start of program headers:             52 (bytes into file)
Start of section headers:            36652 (bytes into file)
Flags:                                0x1001, noreorder, o32, mips1
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:           14
Section header string table index: 11
```

可以看到，前者的字节序列的存储格式是little endian，而后者却是big endian。字节序列不同，自然没办法解析。关于两种字节序列的存储格式的具体说明如下：

对于字节序列的存储格式，目前有两大阵营，那就是Motorola的PowerPC系列CPU和Intel的x86系列CPU。PowerPC系列采用big endian方式存储数据，而x86系列则采用little endian方式存储数据。那么究竟什么是big endian，什么又是little endian呢？

- 1) Little-endian：将低序字节存储在起始地址（低位编址）（先读取数据最低的字节）
- 2) Big-endian：将高序字节存储在起始地址（高位编址）（先读取数据最高的字节）

举个例子：
如果我们将0x1234abcd写入到以0x0000开始的内存中，则结果为：

address	big-endian	little-endian
0x0000	0x12	0xcd
0x0001	0x34	0xab
0x0002	0xab	0x34
0x0003	0xcd	0x12

注：每个地址存1个字节，2位16进制数是1个字节（0xFF=11111111）
更加详细的信息参见CSDN博客 [Big Endian](#) 和 [Little Endian](#) 详解

Thinking 1.3

在理论课上我们了解到，MIPS 体系结构上电时，启动入口地址为0xBFC00000（其实启动入口地址是根据具体型号而定的，由硬件逻辑确定，也有可能不是这个地址，但一定是一个确定的地址），但实验操作系统的内核入口并没有放在上电启动地址，而是按照内存布局图放置。思考为什么这样放置内核还能保证内核入口被正确跳转到？
（提示：思考实验中启动过程的两阶段分别由谁执行。）

MyAnswer

在我们的实验中，GXemul支持加载ELF格式内核，所以启动流程被简化为加载内核到内存，之后跳转到内核的入口。

加载内核到内存

首先，GXemul仿真器通过 tools/scse0_3.lds 这个文件中的 Linker Script 语句，将我们前面通过顶层Makefile 链接生成的内核 加载的目标位置调整正确；

跳转到内核入口

然后， tools/scse0_3.lds 文件用 ENTRY(_start) 命令将链接后的程序入口设置为 _start 这个函数。即链接后的程序从_start函数开始执行；

接下来就是内核自己的事情了

接着，内核自己通过 boot/start.S 中的 _start 函数完成初始化CPU和初始化栈指针sp，并跳转到main函数所在的地址；

由此可见，其实是GXemul仿真器简化了启动过程。我们只需要 tools/scse0_3.lds 文件中正确填写 Linker Script脚本，就可以将内核加载到正确的位置。

Thinking 1.4

sg_size 和 **bin_size** 的区别它的开始加载位置并非页对齐，同时 **bin_size** 的结束位置 (**va+i**) 也并非页对齐，最终整个段加载完毕的 **sg_size** 末尾的位置也并非页对齐，请思考，为了保证页面不冲突（不重复为同一地址申请多个页，以及页上数据尽可能减少冲突），这样一个程序段应该怎样加载内存空间中。

MyAnswer

关于这个问题，我十分不确定，但是姑且还是在这里把自己的想法写一下。

我个人的想法是如下：

首先，在加载普通程序时需要通过一些检查，确定有没有和已经加载的程序冲突。这里不仅仅要检查有没有和前面的程序冲突（即 **L2>R1**），也要检查有没有和后面的程序冲突（即 **R2>L3**）。考虑能不能放下这个程序。

然后，如果无法协调，要么将一些已加载的但是是闲置的程序换下，要么将要加载的程序拆分。

但是我并不确定，也会将这个问题写在最后的疑点。

Thinking 1.5

内核入口在什么地方？**main** 函数在什么地方？我们是怎么让内核进入到想要的 **main** 函数的呢？又是怎么进行跨文件调用函数的呢？

MyAnswer

1. 内核入口是地址 **0x8040 0000**。

2. **main** 函数在 **init/main.c** 文件中声明。

3. **tools/scse0_3.lds** 这个文件中有 **ENTRY(_start)** 这一行命令，旁边的注释写着，这是为了把入口定为 **_start** 这个函数。然后发现 **boot/start.S** 中定义了我们的内核入口函数。在 **start.S** 中，前半部分前半段都是在初始化 **CPU**。后半段则是我们需要填补的部分：

```
lui    sp, 0x8040
jal    main
```

在完成填补以后，可以看出：

后半段的功能就是设置栈指针为内核的入口地址（**lui sp, 0x8040**），并且链接并跳转到 **main** 函数的地址（**jal main**）。

综上，通过 **boot/start.S** 中的 **_start** 函数完成初始化 **CPU** 和初始化栈指针 **sp**，并跳转到 **main** 函数所在的地址；再通过 **tools/scse0_3.lds** 这个文件中的 **ENTRY(_start)** 命令将入口设置为 **_start** 这个函数

指导书的佐证提示

在调用main函数之前，我们需要将sp寄存器设置到内核栈空间的位置上。具体的地址可以从mmu.h中看到。这里做一个提醒，请注意栈的增长方向。设置完栈指针后，我们就具备了执行C语言代码的条件，因此，接下来的工作就可以交给C代码来完成了。所以，在start.S的最后，我们调用C代码的主函数，正式进入内核的C语言部分。

main 函数虽然为 c 语言所书写，但是在被编译成汇编之后，其入口点 会被翻译为一个标签，类似于：

```
main:
XXXXXX
```

MyNote：这里栈指针sp的增长方向应该是只能减去，即sp指向栈的顶部。通过让sp减去一个代表尺寸的数字，为进程分配出一块栈空间，即“压栈”这个形象的描述。

Thinking 1.6

查阅《See MIPS Run Linux》一书相关章节，解释boot/start.S 中下面几行对CP0 协处理器寄存器进行读写的意义。具体而言，它们分别读/写了哪些寄存器的哪些特定位，从而达到什么目的？

```
/* Disable interrupts */
mtc0 zero, CP0_STATUS
.....
/* disable kernel mode cache */
mfc0 t0, CP0_CONFIG
and t0, ~0x7
ori t0, 0x2
mtc0 t0, CP0_CONFIG
```

MyAnswer

```
/* Disable interrupts */
mtc0 zero, CP0_STATUS
```

这条指令CP0_STATUS寄存器的所有可写位置零。所以将该寄存器置零的主要作用是：禁用所有中断。

```
/* disable kernel mode cache */
mfc0 t0, CP0_CONFIG
and t0, ~0x7
ori t0, 0x2
mtc0 t0, CP0_CONFIG
```

该操作将config寄存器的低三位置为010。低三位叫做K0域，控制kseg0的可缓存性与一致性属性。置为010代表uncached，即不经过cache。

顺带一提，上面的四行代码是很常见的更新控制寄存器内部的单个域的代码，下面以状态寄存器SR为例记录一下《See MIPS Run Linux》上提供的模板

```
mfc0    t0, SR
and     t0, <要清零的位的反码>
or      t0, <要置1的位>
mtc     SR, t0
```

Part2 实验难点图示

这个实验最大的难点应该还是初次接触一个比之前都复杂的项目结构，如何弄清楚这个项目到底在干什么，总体架构是什么样。下面我在这一部分想梳理一遍整个实验到底做了些什么。

Step1 制作内核文件vmlinux

关键词：顶层**Makefile**宛如地图，指明了内核文件**vmlinux**是如何被构造出来的

在磕磕绊绊做完了整个实验以后，翻回头来再看顶层的Makefile文件，又有了不同的感觉。下面是我的一些注解。

```
# Main makefile
#
# Copyright (C) 2007 Beihang University
# Written by Zhu Like ( zlike@cse.buaa.edu.cn )
#

drivers_dir    := drivers
boot_dir       := boot
init_dir       := init
lib_dir        := lib
tools_dir      := tools
test_dir       :=
#上面定义了各种文件夹名称
vmlinux_elf    := gxemul/vmlinux      #这个是我们最终需要生成的elf文件

link_script    := $(tools_dir)/scse0_3.lds #链接用的脚本

modules        := boot drivers init lib
#modules 定义了内核所包含的所有模块，objects 则表示要编译出内核所依赖的所有.o 文件。
objects        := $(boot_dir)/start.o \
                  $(init_dir)/main.o \
                  $(init_dir)/init.o \
                  $(drivers_dir)/gxconsole/console.o \
                  $(lib_dir)/*.o \
#定义了需要生成的各种文件 *是通配符，*.o表示任意以.o结尾的文件
## 19 到 23 行行末的斜杠代表这一行没有结束，下一行的内容和这一行是连在一起的。这种写法一般用于
#提高文件的可读性。可以把本该写在同一行的东西分布在多行中，使得文件更容易被人类阅读。

# 如果test_dir不为空，则让 objects所表示的要编译的目标 加上test_dir目录下的.o文件
ifneq ($(test_dir),)
objects :=$(objects) $(test_dir)/*.o
endif

.PHONY: all $(modules) clean      # 这个有另外的笔记

all: $(modules) vmlinux          #我们的“最终目标”
## 这说明：构建整个(all)项目依赖于构建好所有的模块以及 vmlinux

vmlinux: $(modules)
        $(LD) -o $(vmlinux_elf) -N -T $(link_script) $(objects)
#vmlinux 的构建依赖于所有的模块
###具体表现为：调用了链接器 将之前构建各模块产生的所有.o 文件在 linker_script 的指导下链接到
#一起，产生最终的 vmlinux 可执行文件
##注意,这里调用了一个叫做$(LD)的程序

$(modules):
        $(MAKE) --directory=$@ #定义了每个模块的构建方法为调用对应模块目录下的 Makefile
#进入各个子文件夹进行make

clean:
        for d in $(modules); \
```



```

do
    \
    $(MAKE) --directory=$$d clean; \
done; \
rm -rf *.o *~ $(vmlinux_elf)

include include.mk

```

下面总结一下这个顶层的Makefile的功能结构：

这个顶层的Makefile，其核心功能只有一个——在学号根目录的gxemul子目录下，创建可执行文件vmlinux。但是这个过程比较复杂。

第一步，我们需要得到变量\$(modules)所代表的各个模块，因此第一步实际上就是在\$(modules)所代表的boot、drivers、init、lib四个目录中，调用各自目录下的Makefile，将对应的.c文件编译成.o文件。这一点的验证：在学号根目录下make clean后，查看tree，再执行make命令后，查看tree。就会发现，在make命令执行前，这四个目录里面只有.c文件，没有.o文件。执行make命令之后，.o文件才出现。

第二步，在目标可执行文件vmlinux所依赖的各个模块的.o文件均已生成后，调用\$(LD)命令将之前各模块所产生的所有.o文件在链接脚本linker_script的指导下链接到一起，最终输出到vmlinux_elf（即在gxemul子目录下，创建可执行文件vmlinux）。

其实Makefile到这里就结束了，all命令只是声明了创建可执行文件vmlinux这一任务目标，需要依赖\$(modules)所代表的各个模块和vmlinux本身。

test_dir的逻辑比较简单，就不提了。

Step2 生成可以解析ELF文件的可执行文件readelf

关键词：ELF文件的结构包含5个部分；Segment包含一段或若干段Sections

ELF 的文件头，就是一个存了关于 ELF 文件信息的 结构体。首先，结构体中存储了 ELF 的魔数，以验证这是一个有效的 ELF。当我们验证了这是个 ELF 文件之后，便可以通过 ELF 头中提供的信息，进一步地解析 ELF 文件了。在 ELF 头中，提供了段头表的入口偏移。假设 binary 为 ELF 的文件头地址，offset 为入口偏移，那么 binary+offset 即为段头表第一项的地址。

主要的信息都在./readelf/kerelf.h 文件，这里不展开说。比较需要注意的一点是：

结构体 Elf32_Shdr 和 Elf32_Phdr 分别是Section Header Table和Program Header Table 两个头表的**每一个成员**的结构信息。表之所以是表，就是因为他们分别含有这两种结构体的表单项。

然后就是Segment的信息。

Offset代表该段(segment)的数据相对于ELF文件的偏移。**VirtAddr**代表该段最终需要被加载到内存的哪个位置。**FileSiz**代表该段的数据在文件中的长度。**MemSiz**代表该段的数据在内存中所应当占的大小。

这里摘录指导书一段重要的文字

VirtAddr是我们尤为需要注意的。由于它的存在，我们就不难推测，GXemul仿真器在加载我们的内核时，是按照内核这一可执行文件中所记录的地址， 将我们内核中的代码、数据等加载到相应位置。并将CPU的控制权交给内核。我们的内核之所以不能够正常运行，显然是因为我们内核所处的地址是不正确的。换句话说，只要我们能够将内核加载到正确的位置上，我们的内核就应该可以运行起来。

那么自然而然地，我们需要进入第三步。

Step3 寻找内核的正确位置

关键词：32位的MIPS CPU的程序地址空间分为4个大区域；kseg0就是内核应该被加载到的正确区域；

根据include/mmu.h中的内存布局图，不难发现内核应该被加载到名为KERNBASE的地方。

Step4 编写加载脚本Linker Script用于控制内核加载目标地址

关键词：section文件中最重要的三个段是.text .data .bss

在链接过程中，目标文件被看成section的集合，并使用section header table来描述各个section的组织。换句话说，section记录了在链接过程中所需要的必要信息。其中最为重要的三个段为.text、.data、.bss。这三种段的意义是必须要掌握的：

- .text
保存可执行文件的操作指令。
- .data
保存已初始化的全局变量和静态变量。
- .bss
保存未初始化的全局变量和静态变量。

前面我们的顶层Makefile中，有

```
link_script := $(tools_dir)/scse0_3.lds
```

即将link_script链接脚本定义为是tools目录下的scse0_3.lds 文件。

那么，通过填写tools/scse0_3.lds中空缺的部分，我们就可以将内核调整到正确的位置上。

之后，我们又完成了Start.S的填写，那么总体的对于内核加载地址的处理就是：

首先，**GXemul仿真器**通过 tools/scse0_3.lds 这个文件中的 **Linker Script** 语句，将我们前面通过顶层**Makefile** 链接生成的内核 加载的目标位置调整正确；

然后，tools/scse0_3.lds 文件用 **ENTRY(_start)** 命令将链接后的程序入口设置为 **_start** 这个函数。即链接后的程序从_start函数开始执行；

接着，**内核自己**通过 boot/start.S 中的 **_start** 函数完成初始化CPU和初始化栈指针sp，并跳转到main函数所在的地址；

结合GXemul仿真器的特点

GXemul仿真器支持直接加载ELF格式的内核，也就是说，GXemul已经提供了bootloader全部功能。我们的小操作系统不需要再实现bootloader的功能了。换句话说，你可以假定，从我们的小操作系统的运行第一行代码前，我们就已经拥有一个正常的C环境了。全局变量、函数调用等等C语言运行所需的功能已经可以正常使用了。

GXemul支持加载ELF格式内核，所以启动流程被简化为加载内核到内存，之后跳转到内核的入口。

不难看出，我们已经完成了加载任务。

加载内核到内存

首先，**GXemul仿真器**通过 tools/scse0_3.lds 这个文件中的 **Linker Script** 语句，将我们前面通过顶层**Makefile** 链接生成的内核 加载的目标位置调整正确；

跳转到内核入口

然后， `tools/scse0_3.lds` 文件用 `ENTRY(_start)` 命令将链接后的程序入口设置为 `_start` 这个函数。即链接后的程序从 `_start` 函数开始执行；

接下来就是内核自己的事情了

接着， **内核自己**通过 `boot/start.S` 中的 `_start` 函数完成初始化CPU和初始化栈指针`sp`，并跳转到`main`函数所在的地址；

最后再区分一波两个地址



Part3 体会与感想

要学的东西很多，指导书提供了不少有用的教程。自己查阅资料会提供不少帮助。在gcc具体指令的例子方面感谢CSDN。

《See_MIPS_Run_Linux》这本书确实很深奥，感觉书中的内容在初次接触的时候有些难以消化。指导书的有些关键的衔接语句对于理解整个实验流程和实验到底在做什么很有帮助，在做完整个课下部分以后，对于整个实验进行总结，能够有效地理清清楚之前理解不清的地方。

我之前一直把实验和系统启动过程的stage1、stage2硬套，结果发现了GXemul仿真器对于系统启动的简化，那么之前其实就是白纠结了。包括一开始，其实对于Makefile的理解也不够透彻。可能说当时凭借注释和自己查询弄清楚了语句本身的含义，但是由于之后的实验是一部分一部分地推进的，整体的对于实验结构框架的认知并没有建立起来——也就是说，对于第一次艰难完成整个实验的我来说，在我的脑海里，顶层的Makefile和后面的部分的联系并没有建立起来。此时，重看一遍指导书，对于顶层的Makefile又有了更深的理解，再次看到那个章节的标题 **Makefile——内核代码的地图**，就有种醍醐灌顶的感觉了。包括还有后面内存布局与Link Script脚本的编写，在第一次写的时候，他们的逻辑关系并没有建立起来。再回过头来总结的时候，我才意识到：他们两个是紧密相连的步骤。具体的理解在之前就已经说了，这里不表。但是真的有种灵魂震颤的感觉哈哈。

评论区和答疑区有不少大佬的交流是很有用的，在整个实验具体的语句的含义诠释上，感谢强生同学留在评论区的博客。

工作量确实很大，希望理论作业能更加和实验相结合。

Part4 指导书反馈

个人感觉指导书质量比较高。硬要说的话，有几点建议：

对于章节之间关键的过渡衔接语句可以设置的更为醒目一些，大部分同学们不可能一天就能搞完，更多的还是零散化碎片化地推进度，因此不能很好地理解整体实验的框架流程。**我相信，即使是现在，依然有很多同学不能很清晰地说出整个实验到底干了什么。因此，一些必要的、整体上的、提纲挈领的引导还是很重要的。**

思考题的问题个人感觉这次是比较大的。我将其概括为：助教的考察本意和同学们的理解之间出现了gap。

首先，关于thinking1.4 的图在页面中显示歪了导致迷惑同学们这种就暂且不提了。**think1.4本身的语言表达也不够清晰**，给人迷惑的感觉。在和助教交流以后我得知，助教只是希望我们写一些自己的理解，但是语言表达本身的意思就误导了同学们去写一个严谨的、完备的加载方案。

其次，thinking1.3的提示在我看来不如改成：“**思考GXemul仿真器在启动过程中扮演的角色**”。因为现在的提示“**思考实验中启动过程的两阶段分别由谁执行**”不能说错，但事实上在本实验里，**两个阶段的界限是不清晰的。GXemul仿真器将实验的流程简化了，硬要去套两个stage是没道理的，只会徒增疑惑。**事实上就是仿真器几乎一人包办了两个阶段，只要内核开始运行main函数，那就和启动没关系了。

然后，一些思考题的提示实在是太少了。thinking1.1在平台上如何操作，如果我不是看到了群里同学的一张半成品，我可能想不到该如何操作。主要是，OSLAB这个文件夹是没办法在学号根目录下看到的，只能说事先知道了这个文件夹的存在才能跳转进去看看。这样无意义的壁垒我认为是没啥意义的。

Part5 残留难点

关于普通程序应该如何加载还是不清楚，思考题只是写了些个人想法。

以及关于设置cp0协处理器的部分还不是很清楚，这部分是在阅读《See_MIPS_Run_Linux》碰壁以后，看强生同学的博客解决的。