实验报告

一、实验目的

> **1.从操作系统角度理解 MIPS 体系结构**
>
> **2. 掌握操作系统启动的基本流程**
>
> **3.掌握 ELF 文件的结构和功能**
>
> **在本章中，我们需要阅读并填写部分代码，使得我们的小操作系统可以正常的运行起来。 这一章节的难度较为简单**

二、实验步骤

`gxemul` **使用方法**

运行

- `-E` 模拟机器的类型
- `-C` 模拟 CPU 的类型
- `-M` 模拟的内存大小
- `-V` 进入调试模式

退出

- 按 `Ctrl+C`，以中断模拟器
- 输入 `quit` 以退出模拟器

1.找到lab1的文件

```
remote: Branch lab1 set up to track remote branch lab1 from origin.
remote: [ lab1 already exists. ]
To 192.168.130.193:ouc21020007131-lab
   bfc55db..6d79749  lab0 -> lab0
jovyan@70234fb21137:~/ouc21020007131-lab$ git pull
remote: Counting objects: 8, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
From 192.168.130.193:ouc21020007131-lab
   5ff11a3..19ef3cb  lab0-result -> origin/lab0-result
Already up to date.
jovyan@70234fb21137:~/ouc21020007131-lab$ git branch -a
* lab0
  remotes/origin/lab0
  remotes/origin/lab0-result
  remotes/origin/lab1
jovyan@70234fb21137:~/ouc21020007131-lab$ ls
dst  src
jovyan@70234fb21137:~/ouc21020007131-lab$ git checkout lab1
Branch 'lab1' set up to track remote branch 'lab1' from 'origin'.
Switched to a new branch 'lab1'
jovyan@70234fb21137:~/ouc21020007131-lab$ ls
boot  drivers  gxemul  include  include.mk  init  lib  Makefile  readelf  tools
jovyan@70234fb21137:~/ouc21020007131-lab$
```

2. Exercise 1.1

> 请修改 `include.mk` 文件，使交叉编译器的路径正确。之后执行 make 指令，如果配置一切正确，则会在 `gxemul` 目录下生成 `vmlinux` 的内核文件。

修改include.mk中CROSS_COMPILE的值为 `/opt/eldk/user/bin/mips_4KC-`

jovyan@70234fb21137: ~/o × include.mk ●

```
1  # Common includes in Makefile
2  #
3  # Copyright (C) 2007 Beihang University
4  # Written By Zhu Like ( zlike@cse.buaa.edu.cn )
5
6
7  CROSS_COMPILE :=  /opt/eldk/user/bin/mips_4KC-
8  CC            := $(CROSS_COMPILE)gcc
9  CFLAGS        := -O -G 0 -mno-abicalls -fno-builtin -Wa,-xgot -Wall -fPIC
10 LD            := $(CROSS_COMPILE)ld
11
```

jovyan@70234fb21137: ~/o × include.mk ×

```
1  # Common includes in Makefile
2  #
3  # Copyright (C) 2007 Beihang University
4  # Written By Zhu Like ( zlike@cse.buaa.edu.cn )
5
6
7  CROSS_COMPILE :=  /OSLAB/compiler/usr/bin/mips_4KC-
8  CC            := $(CROSS_COMPILE)gcc
9  CFLAGS        := -O -G 0 -mno-abicalls -fno-builtin -Wa,-xgot -Wall -fPIC
10 LD            := $(CROSS_COMPILE)ld
11
```

修改好后运行，使用 `make` 命令，运行完后，发现gxemul文件夹下出现了vmlinux文件

```
jovyan@70234fb21137:~/ouc21020007131-lab$ make
make --directory=boot
make[1]: Entering directory '/home/jovyan/ouc21020007131-lab/boot'
/OSLAB/compiler/usr/bin/mips_4KC-gcc -O -G 0 -mno-abicalls -fno-builtin -Wa,-xgot
-Wall -fPIC -I../include/ -c start.S
make[1]: Leaving directory '/home/jovyan/ouc21020007131-lab/boot'
make --directory=drivers
make[1]: Entering directory '/home/jovyan/ouc21020007131-lab/drivers'
make --directory=gxconsole
make[2]: Entering directory '/home/jovyan/ouc21020007131-lab/drivers/gxconsole'
/OSLAB/compiler/usr/bin/mips_4KC-gcc -O -G 0 -mno-abicalls -fno-builtin -Wa,-xgot
-Wall -fPIC   -c -o console.o console.c
make[2]: Leaving directory '/home/jovyan/ouc21020007131-lab/drivers/gxconsole'
make[1]: Leaving directory '/home/jovyan/ouc21020007131-lab/drivers'
make --directory=init
make[1]: Entering directory '/home/jovyan/ouc21020007131-lab/init'
/OSLAB/compiler/usr/bin/mips_4KC-gcc  -O -G 0 -mno-abicalls -fno-builtin -Wa,-xgot
 -Wall -fPIC -I../include -c init.c
/OSLAB/compiler/usr/bin/mips_4KC-gcc  -O -G 0 -mno-abicalls -fno-builtin -Wa,-xgot
 -Wall -fPIC -I../include -c main.c
make[1]: Leaving directory '/home/jovyan/ouc21020007131-lab/init'
make --directory=lib
make[1]: Entering directory '/home/jovyan/ouc21020007131-lab/lib'
/OSLAB/compiler/usr/bin/mips_4KC-gcc -O -G 0 -mno-abicalls -fno-builtin -Wa,-xgot
-Wall -fPIC -I./ -I../ -I../include/ -c print.c
print.c: In function 'lp_Print':
print.c:53: warning: unused variable 'prec'
print.c:77: warning: 'longFlag' is used uninitialized in this function
print.c:82: warning: 'padc' is used uninitialized in this function
print.c:82: warning: 'width' is used uninitialized in this function
print.c:82: warning: 'ladjust' is used uninitialized in this function
/OSLAB/compiler/usr/bin/mips_4KC-gcc -O -G 0 -mno-abicalls -fno-builtin -Wa,-xgot
-Wall -fPIC -I./ -I../ -I../include/ -c printf.c
make[1]: Leaving directory '/home/jovyan/ouc21020007131-lab/lib'
/OSLAB/compiler/usr/bin/mips_4KC-ld -o gxemul/vmlinux -N -T tools/scse0_3.lds boot
/start.o init/main.o init/init.o drivers/gxconsole/console.o lib/*.o
jovyan@70234fb21137:~/ouc21020007131-lab$ ls gxemul
elfinfo  r3000  r3000_test  test  vmlinux
jovyan@70234fb21137:~/ouc21020007131-lab$
```

3. Exercise 1.2

> 阅读`./readelf`文件夹中`kerelf.h`、`readelf.c`以及`main.c`三个文件中的代码，并完成`readelf.c`，`readelf`函数需要输出 elf 文件的所有 section header 的序号和地址信息，对每个 section header，输出格式为:`"%d:0x%x\n"`，两个标识符分别代表序号和地址。

补全`readelf.c`文件中的代码，在`readelf`函数中，我们首先使用`is_elf_format`函数来检查输入的二进制文件是否符合 ELF 文件的格式要求。如果不符合，则打印错误信息并返回 -1。

接下来，我们根据 ELF 文件的头部信息，获得节头表的地址和节头表的条目数量。然后，通过迭代遍历每个节头表条目，在每次迭代中，我们根据当前条目的偏移地址计算出节的地址，并将节的编号和地址打印出来。

通过这样的实现，我们可以在调用`readelf`函数时，通过传入正确的二进制文件和大小，打印出 ELF 文件中每个节的编号和地址。这有助于对 ELF 文件进行进一步的分析和理解。

```
        // get section table addr, section header number and section header
size.
        ptr_sh_table = binary + ehdr->e_shoff;
        sh_entry_count = ehdr->e_shnum;
        sh_entry_size = ehdr->e_shentsize;

        // for each section header, output section number and section addr.
        for (Nr = 0; Nr < sh_entry_count; Nr++) {
            shdr = (Elf32_Shdr *)(ptr_sh_table + Nr * sh_entry_size);
            printf("%d:0x%x\n", Nr, shdr->sh_addr);
        }

        return 0;
}
```

程序成功编译并运行。`make` 命令被用于编译源代码，生成可执行文件 `readelf`。

运行 `./readelf testELF` 命令时，`testELF` 被作为参数传递给可执行文件 `readelf`，并进行解析。

解析结果显示了每个节的编号和地址。例如，编号为 0 的节的地址是 `0x0`，编号为 1 的节的地址是 `0x8048154`，依此类推。

这个解析结果表明，程序成功读取了 ELF 文件，并能够打印出每个节的编号和地址信息。

```
jovyan@70234fb21137:~/ouc21020007131-lab$ cd readelf
jovyan@70234fb21137:~/ouc21020007131-lab/readelf$ make
gcc -I./ -c main.c
gcc -I./ -c readelf.c
gcc main.o readelf.o -o readelf
jovyan@70234fb21137:~/ouc21020007131-lab/readelf$ ./readelf testELF
0:0x0
1:0x8048154
2:0x8048168
3:0x8048188
4:0x80481ac
5:0x80481cc
6:0x804828c
7:0x804830e
8:0x8048328
9:0x8048358
10:0x8048360
11:0x80483b0
12:0x80483e0
13:0x8048490
14:0x804888c
15:0x80488a8
16:0x80488fc
17:0x8048940
18:0x8049f14
19:0x8049f1c
20:0x8049f24
21:0x8049f28
22:0x8049ff0
23:0x8049ff4
24:0x804a028
25:0x804a030
26:0x0
27:0x0
28:0x0
29:0x0
jovyan@70234fb21137:~/ouc21020007131-lab/readelf$
```

运行 `readelf -S testELF` 命令，可以看到 ELF 文件的节头表信息。

节头表的每个条目包含以下信息：

- `[Nr]`：节的序号。
- `Name`：节的名称。
- `Type`：节的类型。
- `Addr`：节的加载地址。
- `Off`：节的文件偏移量。
- `Size`：节的大小。
- `ES`：节的条目大小。
- `Flg`：节的属性标志。
- `Lk`：节的链接。
- `Inf`：节的附加信息。
- `Al`：节的对齐要求。

通过读取节头表的条目，我们可以获取到 ELF 文件的结构和内容信息。例如，`.text` 节是程序的代码段，`.data` 节是程序的数据段，`.plt` 节是过程链接表。

注意，输出的节头表中显示了30个节的信息，与程序中打印的节个数一致。

```
jovyan@70234fb21137:~/ouc21020007131-lab/readelf$ readelf -S testELF
There are 30 section headers, starting at offset 0x1158:

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        08048154 000154 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048168 000168 000020 00   A  0   0  4
  [ 3] .note.gnu.build-i NOTE            08048188 000188 000024 00   A  0   0  4
  [ 4] .gnu.hash         GNU_HASH        080481ac 0001ac 000020 04   A  5   0  4
  [ 5] .dynsym           DYNSYM          080481cc 0001cc 0000c0 10   A  6   1  4
  [ 6] .dynstr           STRTAB          0804828c 00028c 000081 00   A  0   0  1
  [ 7] .gnu.version      VERSYM          0804830e 00030e 000018 02   A  5   0  2
  [ 8] .gnu.version_r    VERNEED         08048328 000328 000030 00   A  6   1  4
  [ 9] .rel.dyn          REL             08048358 000358 000008 08   A  5   0  4
  [10] .rel.plt          REL             08048360 000360 000050 08   A  5  12  4
  [11] .init             PROGBITS        080483b0 0003b0 00002e 00  AX  0   0  4
  [12] .plt              PROGBITS        080483e0 0003e0 0000b0 04  AX  0   0 16
  [13] .text             PROGBITS        08048490 000490 0003fc 00  AX  0   0 16
  [14] .fini             PROGBITS        0804888c 00088c 00001a 00  AX  0   0  4
  [15] .rodata           PROGBITS        080488a8 0008a8 000053 00   A  0   0  4
  [16] .eh_frame_hdr     PROGBITS        080488fc 0008fc 000044 00   A  0   0  4
  [17] .eh_frame         PROGBITS        08048940 000940 000104 00   A  0   0  4
  [18] .ctors            PROGBITS        08049f14 000f14 000008 00  WA  0   0  4
  [19] .dtors            PROGBITS        08049f1c 000f1c 000008 00  WA  0   0  4
  [20] .jcr              PROGBITS        08049f24 000f24 000004 00  WA  0   0  4
  [21] .dynamic          DYNAMIC         08049f28 000f28 0000c8 08  WA  6   0  4
  [22] .got              PROGBITS        08049ff0 000ff0 000004 04  WA  0   0  4
  [23] .got.plt          PROGBITS        08049ff4 000ff4 000034 04  WA  0   0  4
  [24] .data             PROGBITS        0804a028 001028 000008 00  WA  0   0  4
  [25] .bss              NOBITS          0804a030 001030 000008 00  WA  0   0  4
  [26] .comment          PROGBITS        00000000 001030 00002a 01  MS  0   0  1
  [27] .shstrtab         STRTAB          00000000 00105a 0000fc 00      0   0  1
  [28] .symtab           SYMTAB          00000000 001608 0004b0 10     29  46  4
  [29] .strtab           STRTAB          00000000 001ab8 000294 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),
  p (processor specific)
```

4. Thinking 1.1

> 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)

使用 readelf 程序解析testELF文件

```
jovyan@70234fb21137:~/ouc21020007131-lab/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
```

使用 `readelf` 程序解析之前的内核文件时，看到内核的入口点地址为 `0x0`

入口点地址(`Entry point address`)是指程序（包括内核）开始执行时的入口地址。当操作系统启动时，处理器会跳转到该地址开始执行相应的代码，这通常是操作系统内核的启动点。

```
jovyan@70234fb21137:~/ouc21020007131-lab/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:          36716 (bytes into file)
  Flags:                             0x50001001, noreorder, o32, mips32
  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
```

通过对比两次解析的输出结果，可以看到这两个 ELF 文件在数据格式上有所不同。`testFLF` 使用的是小端序（`little endian`），而 `vmlinux` 使用的是大端序（`big endian`）,与 `readelf` 程序使用的小端序不匹配。

打开 `include/mmu.h` 文件，找到内核文件的位置

```
/*
 o     4G ----------->  +------------------------------+-----------0x100000000
 o                      |       ...                    |  kseg3
 o                      +------------------------------+-----------0xe000 0000
 o                      |       ...                    |  kseg2
 o                      +------------------------------+-----------0xc000 0000
 o                      |   Interrupts & Exception     |  kseg1
 o                      +------------------------------+-----------0xa000 0000
 o                      |      Invalid memory          |   /|\
 o                      +------------------------------+----|-------Physics Memory Max
 o                      |       ...                    |  kseg0
 o  VPT,KSTACKTOP-----> +------------------------------+----|-------0x8040 0000-------end
 o                      |       Kernel Stack           |    | KSTKSIZE            /|\
 o                      +------------------------------+----|------                 |
 o                      |       Kernel Text            |    |                     PDMAP
 o      KERNBASE -----> +------------------------------+----|-------0x8001 0000    |
 o                      |   Interrupts & Exception     |   \|/                     \|/
 o      ULIM     -----> +------------------------------+------------0x8000 0000-------
 o                      |         User VPT             |     PDMAP                 /|\
 o      UVPT     -----> +------------------------------+------------0x7fc0 0000    |
 o                      |         PAGES                |     PDMAP                 |
 o      UPAGES   -----> +------------------------------+------------0x7f80 0000    |
 o                      |         ENVS                 |     PDMAP                 |
 o  UTOP,UENVS   -----> +------------------------------+------------0x7f40 0000    |
 o  UXSTACKTOP -/       |     user exception stack     |     BY2PG                 |
 o                      +------------------------------+------------0x7f3f f000    |
 o                      |       Invalid memory         |     BY2PG                 |
 o      USTACKTOP ----> +------------------------------+------------0x7f3f e000    |
 o                      |     normal user stack        |     BY2PG                 |
 o                      +------------------------------+------------0x7f3f d000    |
 a                      |                              |                           |
 a                      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~                           |
 a                      .                              .                          |
 a                      .                              .                       kuseg
 a                      .                              .                          |
```

根据注释和代码内容，可以看到 `KERNBASE` = `0x80010000`

内核基址(`KERNBASE`)是指内核在虚拟地址空间中的位置，用于将内核的各个部分放置在虚拟地址空间的适当位置，以便访问和管理。

则内核的起始地址为`0x80010000` 。以下是内存布局的主要部分：

- `KSEG0`（kernel segment 0）：用于映射内核代码和数据。从 `KERNBASE`（`0x80010000`）开始。
- `KSEG1`（kernel segment 1）：用于映射中断和异常向量表。从 `0xa0000000` 开始。
- `KSEG2`（kernel segment 2）：保留区域。
- `KSEG3`（kernel segment 3）：保留区域。

其中，`KSEG0` 是内核的代码和数据所在的地址空间。通过 `UTEXT`（`0x00400000`）定义了内核的代码起始地址。`UTEXT` 之上是内核的栈空间（kernel stack），由 `KSTACKTOP`（`VPT-0x100`）和 `KSTKSIZE`（8*`BY2PG`）定义。

5. Exercise 1.3

> 填写`tools/scse0_3.lds`中空缺的部分，将内核调整到正确的位置上。

我们使用模拟器`gxemul`把我们实验的内核文件`vmlinux`给加载到内存，然后跳转到内核的起始地址，即前面我们找到的地址`0x80010000`

填写空缺部分：

```
SECTIONS
{

/*To do:
  fill in the correct address of the key section
  such as text data bss ...
*/
    . = 0x80010000;  /* 起始地址 */
    .text : { *(.text) }  /* 可执行文件的操作指令 */
    .data : { *(.data) }  /* 已初始化的全局变量和静态变量 */
    .bss : { *(.bss) }  /* 未初始化的全局变量和静态变量 */

    /* 其他节的定义 */
    end = . ;

}
```

运行命令`readelf -S vmlinux`，查看内核的节信息，可以看到每个节的位置 Addr

```
jovyan@70234fb21137:~/ouc21020007131-lab/gxemul$ readelf -S vmlinux
There are 14 section headers, starting at offset 0x8f6c:

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 000080 000950 00 WAX  0   0 16
  [ 2] .reginfo          MIPS_REGINFO    00000950 0009d0 000018 18   A  0   0  4
  [ 3] .rodata.str1.4    PROGBITS        00000968 0009e8 0000a2 01 AMS  0   0  4
  [ 4] .rodata           PROGBITS        00000a10 000a90 000200 00   A  0   0 16
  [ 5] .data             PROGBITS        00000c10 000c90 000000 00  WA  0   0 16
  [ 6] .data.stk         PROGBITS        00000c10 000c90 008000 00  WA  0   0  1
  [ 7] .bss              NOBITS          00008c10 008c90 000000 00  WA  0   0 16
  [ 8] .pdr              PROGBITS        00000000 008c90 0001a0 00      0   0  4
  [ 9] .mdebug.abi32     PROGBITS        00000000 008e30 000000 00      0   0  1
  [10] .comment          PROGBITS        00000000 008e30 0000c8 00      0   0  1
  [11] .shstrtab         STRTAB          00000000 008ef8 000072 00      0   0  1
  [12] .symtab           SYMTAB          00000000 00919c 000250 10     13  24  4
  [13] .strtab           STRTAB          00000000 0093ec 0000c2 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),
  p (processor specific)
jovyan@70234fb21137:~/ouc21020007131-lab/gxemul$
```

6. Thinking 1.2

> `main`函数在什么地方？我们又是怎么跨文件调用函数的呢？

内核的入口地址为`0x80000000`，而`main`函数在内核通过执行跳转指令`jal`跳转到`main`函数。

跨文件调用函数通常涉及到以下几个步骤：

1. 函数的声明：在需要调用函数的源文件中，需要在文件开头声明要调用的函数。声明包括函数的返回类型、函数名和参数列表等信息。
2. 文件的包含：源文件需要包含要调用的函数所在的头文件。头文件中包含了函数的声明，以便编译器知道函数的存在和如何调用。
3. 编译和链接：将需要调用的函数的源文件和源中的调用部分别编译为目标文件。编译器将检查调的函数是否存在确认函数的参数和值类型等是否。然后，将的目标文件链接一起生成可文件或者库文件。

4. 符号解析和重定位在链接阶段，译器会对和变量进行符解析。它根据函数的声明定义的规则找对应的函数，并生成符号表。然后，在连接进行重定位时，将调用处的函数符号替换为实际的函数入地址。
5. 执行用：在程序过程中，当到调用函数语句时，执行将跳转到函数的入口地址，并将参数传递给函数。函数执行完毕后，会返回到调用处继续执行。

7. Exercise 1.4

> 完成 `boot/start.S` 中空缺的部分。设置栈指针，跳转到 `main` 函数。使 用 `/OSLAB/gxemul -E testmips -C R3000 -M 64 elf-file` 运行（其中 `elf-file` 是你编 译生成的 `vmlinux` 文件的路径，我的路径为`gxemul/vmlinux`）

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。

从`include/mmu.h`中可以看到，内核的栈顶为`0x80400000`

```
o                    +------------------------------+----|-------Physics Memory Max
o                    |       ...                    |  kseg0
o  VPT,KSTACKTOP-----> +------------------------------+----|-------0x8040 0000-------end
o                    |       Kernel Stack           |    | KSTKSIZE            /|\
o                    +------------------------------+----|------                 |
```

完成 `boot/start.S` 中空缺的部分，如下：

```
/*To do:
  set up stack
  you can reference the memory layout in the include/mmu.h
*/
li sp, 0x80400000  //设置栈指针
jal main  //跳转到main函数
nop
```

重新`make`，然后运行命令`gxemul -E testmips -C R3000 -M 64 gxemul/vmlinux`

```
jovyan@70234fb21137:~/ouc21020007131-lab$ gxemul -E testmips -C R3000 -M 64 gxemul/vmlinux
GXemul 0.4.6    Copyright (C) 2003-2007  Anders Gavare
Read the source code and/or documentation for other Copyright messages.

Simple setup...
    net: simulating 10.0.0.0/8 (max outgoing: TCP=100, UDP=100)
        simulated gateway: 10.0.0.254 (60:50:40:30:20:10)
            using nameserver 192.168.224.14
    machine "default":
        memory: 64 MB
        cpu0: R3000 (I+D = 4+4 KB)
        machine: MIPS test machine
        loading gxemul/vmlinux
        starting cpu0 at 0x80010000
--------------------------------------------------------------------------------

main.:  main imain.:    main is start ...

 s start ...

tart ...
```

8. Exercise 1.5

> 阅读相关代码和下面对于函数规格的说明，补全 `lib/print.c` 中 `lp_Print()` 函数中缺失的部分来实现字符输出。

填写代码：

```
    for(;;) {
{
    /* scan for the next '%' */
        while ((*fmt != '%') && (*fmt != '\0'))
        {
            OUTPUT(arg, fmt, 1);
            fmt++;
        }
    /* flush the string found so far */

    /* are we hitting the end? */
        if (*fmt == '\0')  break;
}

/* we found a '%' */
 fmt ++;
/* check for long */

/* check for other prefixes */

/* check format flag */
    if(*fmt == '-'){
        ladjust=1, fmt++;
    } else if(*fmt == '0'){
        padc='0',fmt++;
    }
    for(; IsDigit(*fmt); fmt++){
        width=width*10+Ctod(*fmt);
    }
    if(*fmt=='.'){
        fmt++;
        for(; IsDigit(*fmt); fmt++){
            prec=prec*10+Ctod(*fmt);
        }
    }
    if(*fmt=='l'){
        longFlag=1;
        fmt++;
    }
    longFlag=0;
    negFlag = 0;
    width=0;
    ladjust=0;//默认右对齐
    prec=0;
    padc=' ';
```

重新`make`，然后运行命令`gxemul -E testmips -C R3000 -M 64 gxemul/vmlinux`

```
jovyan@70234fb21137:~/ouc21020007131-lab$ gxemul -E testmips -C R3000 -M 64 gxemul/vmlinux
GXemul 0.4.6    Copyright (C) 2003-2007  Anders Gavare
Read the source code and/or documentation for other Copyright messages.

Simple setup...
    net: simulating 10.0.0.0/8 (max outgoing: TCP=100, UDP=100)
        simulated gateway: 10.0.0.254 (60:50:40:30:20:10)
            using nameserver 192.168.224.14
    machine "default":
        memory: 64 MB
        cpu0: R3000 (I+D = 4+4 KB)
        machine: MIPS test machine
        loading gxemul/vmlinux
        starting cpu0 at 0x80010000
--------------------------------------------------------------------------------

main.c: main is start ...

init.c: mips_init() is called

panic at init.c:24: ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

结果正确!

9. 提交lab1

```
jovyan@70234fb21137:~/ouc21020007131-lab$ git add .
jovyan@70234fb21137:~/ouc21020007131-lab$ git config --global user.email "zym8004@stu.ouc.edu.cn"
jovyan@70234fb21137:~/ouc21020007131-lab$ git config --global user.name "munume"
jovyan@70234fb21137:~/ouc21020007131-lab$ git commit -m "lab1"
[lab1 5cbc049] lab1
 26 files changed, 970 insertions(+), 5 deletions(-)
 create mode 100644 .ipynb_checkpoints/Makefile-checkpoint
 create mode 100644 .ipynb_checkpoints/include-checkpoint.mk
 create mode 100644 boot/.ipynb_checkpoints/Makefile-checkpoint
 create mode 100644 boot/.ipynb_checkpoints/start-checkpoint.S
 create mode 100644 boot/start.o
 create mode 100644 drivers/gxconsole/console.o
 create mode 100755 gxemul/vmlinux
 create mode 100644 include/.ipynb_checkpoints/mmu-checkpoint.h
 create mode 100644 init/init.o
 create mode 100644 init/main.o
 create mode 100644 lib/.ipynb_checkpoints/print-checkpoint.c
 create mode 100644 lib/.ipynb_checkpoints/printf-checkpoint.c
 create mode 100644 lib/print.o
 create mode 100644 lib/printf.o
 create mode 100644 readelf/.ipynb_checkpoints/kerelf-checkpoint.h
 create mode 100644 readelf/.ipynb_checkpoints/main-checkpoint.c
 create mode 100644 readelf/.ipynb_checkpoints/readelf-checkpoint.c
 create mode 100644 readelf/main.o
 create mode 100755 readelf/readelf
 create mode 100644 readelf/readelf.o
 create mode 100644 tools/.ipynb_checkpoints/scse0_3-checkpoint.lds
```

```
jovyan@70234fb21137:~/ouc21020007131-lab$ git push
Counting objects: 32, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (31/31), done.
Writing objects: 100% (32/32), 16.67 KiB | 1.51 MiB/s, done.
Total 32 (delta 6), reused 0 (delta 0)
remote: **********************************************************************
remote:
remote:                     BUAA OSLAB AUTOTEST SYSTEM
remote:                   Copyright (c) BUAA 2015-2019
remote:
remote: **********************************************************************
remote:
remote: [ You are changing the branch: refs/heads/lab1. ]
remote:
remote: Autotest: Begin at Thu Jan  4 01:12:25 CST 2024
remote:
remote: warning: remote HEAD refers to nonexistent ref, unable to checkout.
remote:
remote: Switched to a new branch 'lab1'
remote: Branch lab1 set up to track remote branch lab1 from origin.
remote: lab variable value is lab1
remote: [ readelf.c found ]
remote: rm -rf *.o
remote: rm -rf readelf
remote: gcc -I./ -c main.c
remote: gcc -I./ -c readelf.c
remote: gcc main.o readelf.o -o readelf
remote: [ Compile success! readelf found. ]
remote: [ PASSED:25 ]
remote: [ TOTAL:25 ]
remote: [ You have passed readelf testcase 1/2 ]


remote:
remote: End build at Thu Jan  4 01:12:41 CST 2024
remote: [ PASSED:5 ]
remote: [ TOTAL:5 ]
remote: [ You have passed all testcases of extra printf. ]
remote: [ You got 100 (of 100) this time. Thu Jan  4 01:12:51 CST 2024 ]
remote:
remote:
remote: >>>>>> Collecting autotest results >>>>>>
```

三、实验总结

遇到的问题和解决办法：

区别内核的起始地址和内核的入口点地址：

- 内核基址(`KERNBASE`)是指内核在虚拟地址空间中的位置，用于将内核的各个部分放置在虚拟地址空间的适当位置，以便访问和管理。实验中`KERNBASE` = `0x80010000`
- 入口点地址(`Entry point address`)是指程序（包括内核）开始执行时的入口地址。当操作系统启动时，处理器会跳转到该地址开始执行相应的代码，这通常是操作系统内核的启动点。实验中应该是`0x80000000`

收获与体会

从操作系统角度理解 MIPS 体系结构，简单掌握了操作系统启动的基本流程，掌握了 ELF 文件的结构和功能，学会了使用模拟器`gxemul`运行和调试 MIPS 体系架构下的代码

大概用了五六个小时