

# 操作系统实验

## lab1 内核、Boot和printf

# 内容提要

- 实验概述
- 实验内容
  - 修改交叉编译器的路径
  - 解析ELF文件
  - 调整内核到正确位置
  - 设置栈指针并跳转到主函数
  - 实现printf字符输出
- 测试结果

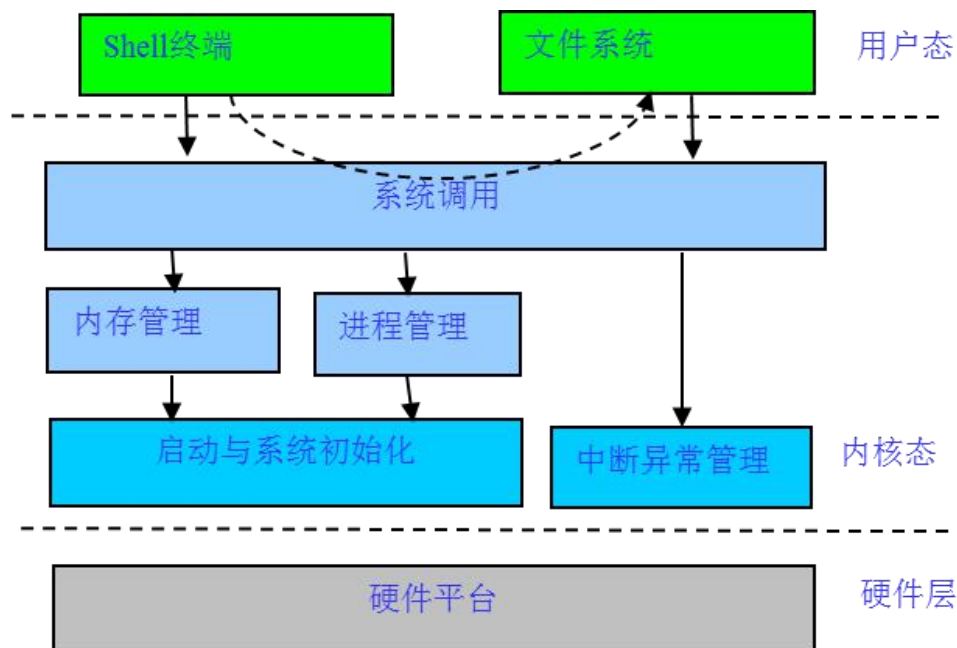
# 实验概述

- 操作系统的启动
  - 了解操作系统的启动原理及流程
- 修改内核并实现一些自定义的功能
  - 了解Makefile——内核代码的地图
  - 了解ELF——深入探究编译与链接
  - 了解MIPS 内存布局——寻找内核的正确位置
  - 了解Linker Script——控制加载地址
- MIPS 汇编与 C 语言
- 实战 printf

# 小操作系统实验的各个部分及相互关系

## ■ 实验设计（七个实验都已在MIPS仿真器上实现）

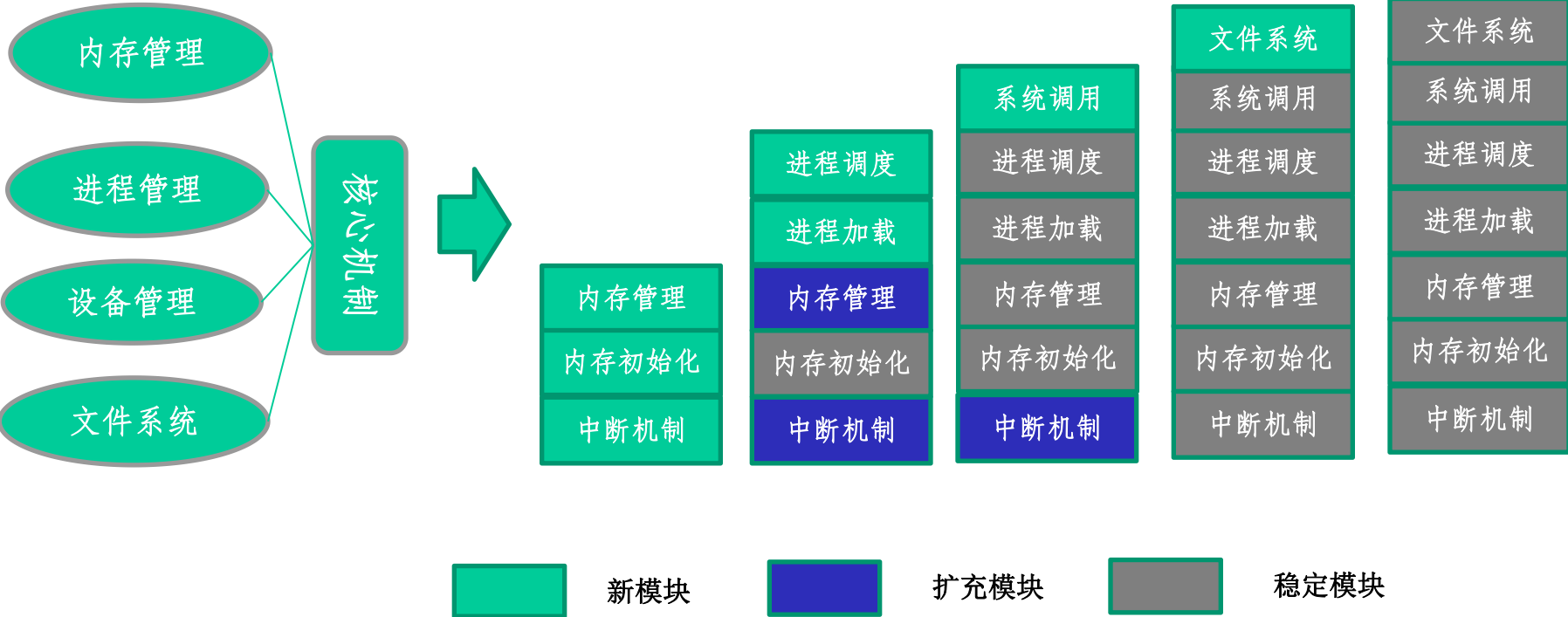
- 基础知识
- 启动和系统初始化
- 内存管理
- 进程管理和中断异常机制
- 系统调用
- 文件系统
- 命令解释程序



# 增量式实现方法

原理

实现



# 操作系统的启动原理及流程

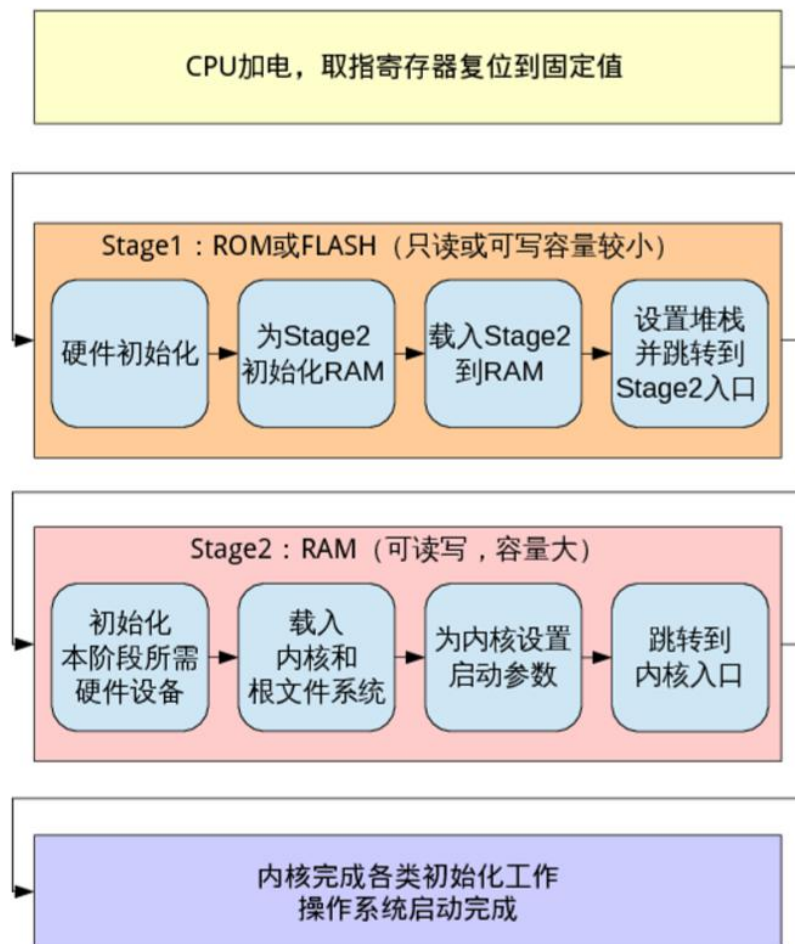
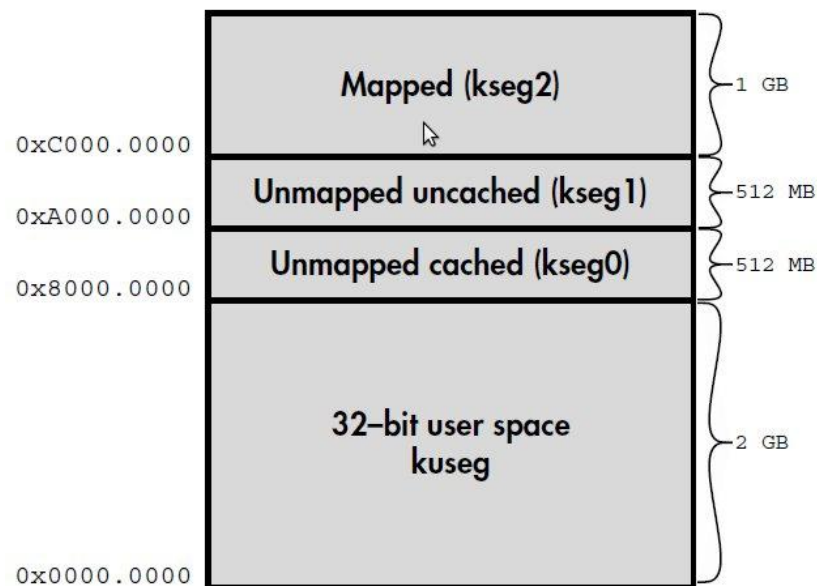


图 1.1: 启动的基本步骤

# gxemul 中的启动流程

- gxemul 仿真器支持直接加载elf 格式的内核
  - gxemul 已经提供了bootloader
- 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。
- 整个启动过程非常简单



# 代码导读

- User Space(kuseg) 0x00000000~0x7FFFFFFF(2G):
  - 用户模式下可用，通过MMU 映射到实际的物理地址上。
- Kernel Space Unmapped Cached(kseg0) 0x80000000~0x9FFFFFFF(512MB):
  - 只需要将地址的高3 位清零，就被转换为物理地址。一般情况下，都是通过cache 对这段区域的地址进行访问。
- Kernel Space Unmapped Uncached(kseg1) 0xA0000000~0xBFFFFFFF(512MB):
  - 通过将高3 位清零的方法将地址映射为相应的物理地址，然后映射到物理内存中低端512MB地址。
- Kernel Space Mapped Cached(kseg2) 0xC0000000~0xFFFFFFFF(1GB):
  - 这段地址只能在内核态下使用并且需要MMU 的转换。



# Makefile——内核代码的地图

- make 工具一般用于 维护工程。

```
target: dependencies  
    command 1  
    command 2  
    ...  
    command n
```

- 其中，target 是我们构建 (Build) 的目标，而 dependencies 是构建该目标所需的其 它文件或其他目标。

# 代码导读

## ■ Makefile

- 代码链接顺序
- 环境变量

Listing 1: 顶层 Makefile

```
1  # Main makefile
2  #
3  # Copyright (C) 2007 Beihang University
4  # Written by Zhu Like ( zlike@cse.buaa.edu.cn )
5  #
6
7  drivers_dir      := drivers
8  boot_dir         := boot
9  init_dir         := init
10 lib_dir          := lib
11 tools_dir        := tools
12 vmlinux_elf       := gxemul/vmlinux
13
14 link_script       := $(tools_dir)/scse0_3.lds
15
16 modules           := boot drivers init lib
17 objects           := $(boot_dir)/start.o \
18                     $(init_dir)/main.o \
19                     $(init_dir)/init.o \
20                     $(drivers_dir)/gxconsole/console.o \
21                     $(lib_dir)/*.o
22
23 .PHONY: all $(modules) clean
24
25 all: $(modules) vmlinux
26
27 vmlinux: $(modules)
28         $(LD) -o $(vmlinux_elf) -N -T $(link_script) $(objects)
29
30 $(modules):
31         $(MAKE) --directory=$@
32
33 clean:
34         for d in $(modules); \
35         do \
36                 $(MAKE) --directory=$$d clean; \
37         done; \
38         rm -rf *.o *~ $(vmlinux_elf)
39
40 include include.mk
```

# Exercise 1.1 修改交叉编译器的路径

Listing 2: include.mk

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

- 阅读顶层Makefile文件后，会发现还有几个关键的变量没有定义，即LD、MAKE等出现在编译指令中的变量。观察其最后一行引用了include.mk文件，显然这些未定义的变量是被定义在了这个文件中，其文件内容如上图所示。
- CROSS\_COMPILE 变量是在定义编译和链接等指令的前缀，是交叉编译器的具体位置。但该路径是错误的，将其修改为正确路径，并执行 make 指令使其在 gxemul 目录下生成 vmlinux 的内核文件。

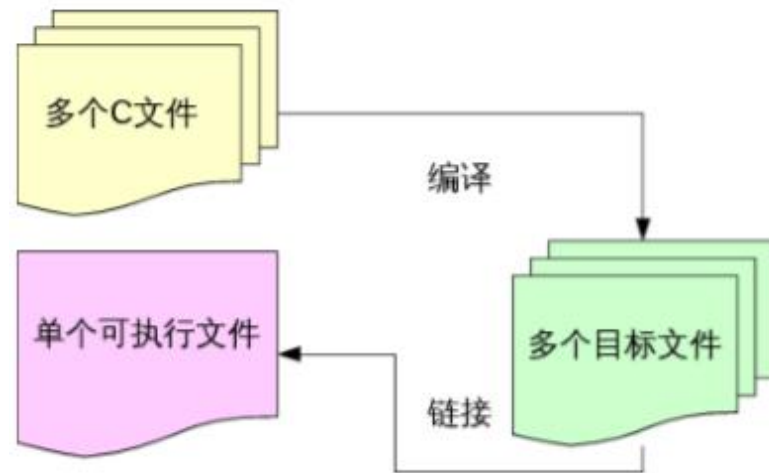
# Exercise 1.1 修改交叉编译器的路径

```
user3@ubuntu14:~/user3-lab$ make
make --directory=boot
make[1]: Entering directory `/home/user3/user3-lab/boot'
bin/mips_4KC-gcc -O -G 0 -mno-abicalls -fno-builtin -Wa,-xgot -Wall
-fPIC -I../include/ -c start.S
make[1]: bin/mips_4KC-gcc: Command not found
make[1]: *** [start.o] Error 127
make[1]: Leaving directory `/home/user3/user3-lab/boot'
make: *** [boot] Error 2
```

```
user3@ubuntu14:~/user3-lab$ make
make --directory=boot
make[1]: Entering directory `/home/user3/user3-lab/boot'
make[1]: Nothing to be done for `all'.
make[1]: Leaving directory `/home/user3/user3-lab/boot'
make --directory=drivers
make[1]: Entering directory `/home/user3/user3-lab/drivers'
make --directory=gxconsole
make[2]: Entering directory `/home/user3/user3-lab/drivers/gxconsole'
make[2]: Nothing to be done for `all'.
make[2]: Leaving directory `/home/user3/user3-lab/drivers/gxconsole'
make[1]: Leaving directory `/home/user3/user3-lab/drivers'
make --directory=init
make[1]: Entering directory `/home/user3/user3-lab/init'
make[1]: Nothing to be done for `all'.
make[1]: Leaving directory `/home/user3/user3-lab/init'
make --directory=lib
make[1]: Entering directory `/home/user3/user3-lab/lib'
make[1]: Nothing to be done for `all'.
make[1]: Leaving directory `/home/user3/user3-lab/lib'
/opt/eldk/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
user3@ubuntu14:~/user3-lab$ ls gxemul/
elfinfo  r3000  r3000_test  test  vmlinux
user3@ubuntu14:~/user3-lab$
```

# ELF——深入探究编译与链接

- 对于拥有多个c文件的工程来说，编译器会首先将所有的c文件以文件为单位，编译成.o文件。最后再将所有的.o文件以及函数库链接在一起，形成最终的可执行文件，整个过程如右图所示。

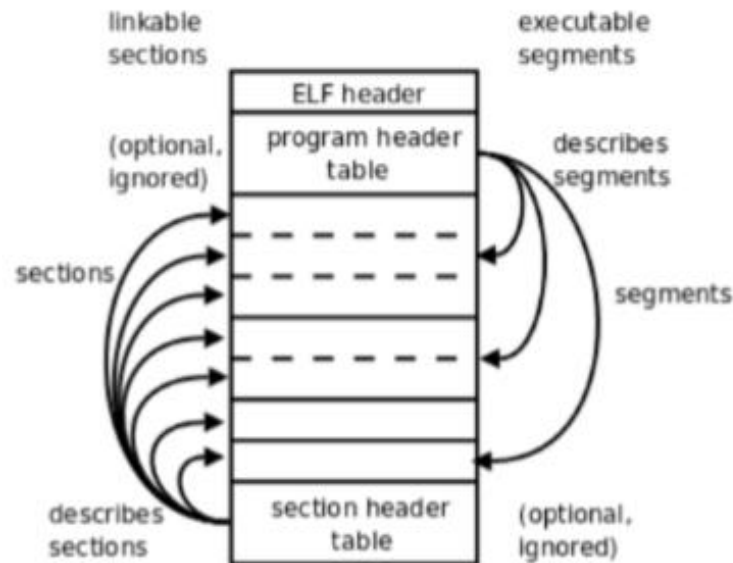


- 在目标文件（也就是我们通过-c选项生成的.o文件）中，记录了代码各个段的具体信息。链接器通过这些信息来将目标文件链接到一起。而ELF(Executable and Linkable Format)正是 Unix 上常用的一种目标文件格式。其实，不仅仅是目标文件，可执行文件也是使用 ELF 格式记录的。这一点通过 ELF 的全称也可以看出来。

# ELF——深入探究编译与链接

- ELF文件大体结构如右图所示，其包括5个部分：

- ELF Header
- Program Header Table
- Section Header Table
- Segments
- Sections



- 观察上图我们可以发现，Program Header Table 和 Section Header Table 指向了同样的地方，这就说明两者所代表的内容是重合的，这意味着什么呢？意味着两者只是同一个东西的不同视图！产生这种情况的原因在于 ELF 文件需要在两种场合使用：
  - 组成可重定位文件，参与可执行文件和可共享文件的链接
  - 组成可执行文件或者可共享文件，在运行时为加载器提供信息。



# Exercise 1.2 解析ELF文件

- 了解ELF文件大体结构之后，阅读一个简易的对32bitELF 文件 (little endian) 的解析程序
- 完成部分代码以了解 ELF 文件各个部分的详细结构。
- 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 中缺少的代码
- 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。

```
1 typedef struct
2 {
3     // some identification information, including magic number
4     unsigned char e_ident[EI_NIDENT];
5     // file type, including relocatable file, executable file and shared object
6     Elf32_Half e_type;
7     // architecture, like MIPS
8     Elf32_Half e_machine;
9     // version info
10    Elf32_Word e_version;
11    // addr of entry point
12    Elf32_Addr e_entry;
13    // program header table offset
14    Elf32_Off e_phoff;
15    // section header table offset
16    Elf32_Off e_shoff;
17    // relating to processor
18    Elf32_Half e_flags;
19    // elf header size
20    Elf32_Half e_ehsize;
21    // program header entry size
22    Elf32_Half e_phentsize;
23    // program header entry number
24    Elf32_Half e_phnum;
25    // section header entry size
26    Elf32_Half e_shentsize;
27    // section header entry number
28    Elf32_Half e_shnum;
29 }Elf32_Ehdr;
```



```
user1@ubuntu14:~/user1-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
```

正确完成后在readelf  
文件夹下使用

./readelf testELF

命令可以观察到右图  
现象

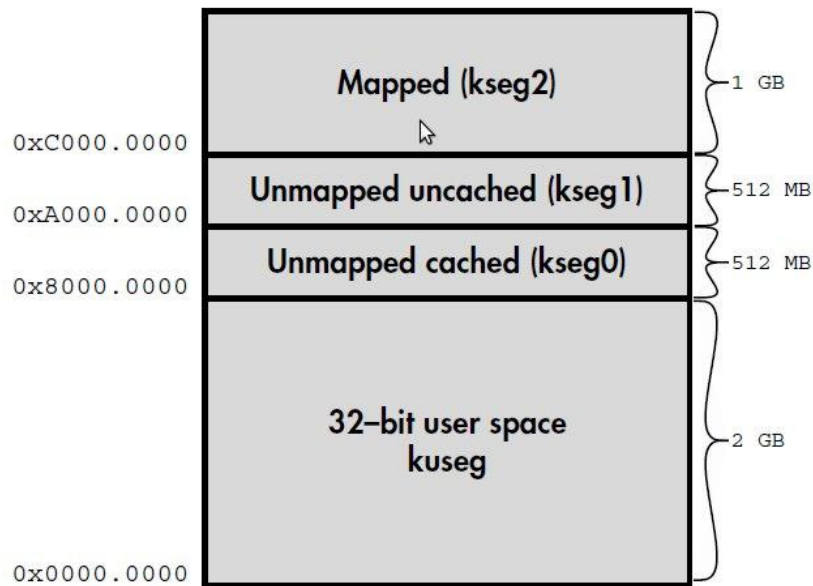
# 课下测试Lab1-1评测说明

```
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: [ PASSED:24 ]
remote: [ TOTAL:24 ]
remote: [ You have passed readelf testcase 2/2 ]
```

**[ You got 50 (of 100) this time. Thu Mar 11 19:44:24 CST 2021 ]**

正确完成Exercise1.1~1.2后提交代码，可以看到如上图所示评测结果，得到40 + 10（Lab1-2成功编译）分。

# MIPS 内存布局——寻找内核的正确位置



Listing 5: include/mmu.h 中的内存布局图

[illegible]

# Linker Script——控制加载地址

- 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。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。

# Linker Script——控制加载地址

- (<https://www.sourceware.org/binutils/docs/ld/Simple-Example.html#Simple-Example>) 该例子的完整代码如下所示：

```
SECTIONS
{
    . = 0x10000;
    .text : { *(.text) }
    . = 0x8000000;
    .data : { *(.data) }
    .bss : { *(.bss) }
}
```

- 在第三行的 “.” 是一个特殊符号，用来做定位计数器，它根据输出段的大小增长。在 SECTIONS 命令开始的时候，它的值为 0。通过设置 “.” 即可设置接下来的 section 的起始地址。“\*” 是一个通配符，匹配所有的相应的段。例如 “.bss:{\*(.bss)}” 表示将所有输入文件中的.bss 段（右边的.bss）都放到输出的.bss 段（左边的.bss）中。

# Exercise 1.3 调整内核到正确位置

- 控制链接器的链接过程让内核被加载到该位置。
- Linker Script 记录了各个 section 应该如何映射到 segment, 以及各个 segment 应该被加载到的位置。在使用了我们自定义的 Linker Script 后, 生成的程序各个 section 的位置就被调整到了我们所指定的地址上, 如右图所示。
- 理解并仿照用例, 填写 tools/scse0\_3.lds 中空缺的部分, 将内核调整到正确的位置上。

```
1  SECTIONS
2  {
3      . = 0x10000;
4      .text : { *(.text) }
5      . = 0x8000000;
6      .data : { *(.data) }
7      .bss : { *(.bss) }
8  }
```



# Exercise 1.3 调整内核到正确位置

```
user1@ubuntu14:~/user1-lab$ readelf -S gxemul/vmlinux
There are 14 section headers, starting at offset 0x90ac:
```

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	30010000	000080	000a80	00	WAX	0	0	16
[ 2]	.reginfo	MIPS_REGINFO	30010a80	000b00	000018	18	A	0	0	4
[ 3]	.rodata.str1.4	PROGBITS	30010a98	000b18	0000a2	01	AMS	0	0	4
[ 4]	.rodata	PROGBITS	30010b40	000bc0	000210	00	A	0	0	16
[ 5]	.data	PROGBITS	30010d50	000dd0	000000	00	WA	0	0	16
[ 6]	.data.stk	PROGBITS	30010d50	000dd0	008000	00	WA	0	0	1
[ 7]	.bss	NOBITS	30018d50	008dd0	000000	00	WA	0	0	16
[ 8]	.pdr	PROGBITS	00000000	008dd0	0001a0	00		0	0	4
[ 9]	.mdebug.abi32	PROGBITS	00000000	008f70	000000	00		0	0	1
[10]	.comment	PROGBITS	00000000	008f70	0000c8	00		0	0	1
[11]	.shstrtab	STRTAB	00000000	009038	000072	00		0	0	1
[12]	.symtab	SYMTAB	00000000	0092dc	000250	10		13	24	4
[13]	.strtab	STRTAB	00000000	00952c	0000c2	00		0	0	1

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 202.112.128.51
machine "default":
  memory: 64 MB
  cpu0: R3000 (I+D = 4+4 KB)
  machine: MIPS test machine
  loading gxemul/vmlinux
  starting cpu0 at 0x80010000
```

# MIPS汇编与C语言

## ■ 函数调用

- 压栈 (push) 和弹栈 (pop) 的操作
- 参数存放在 a0-a3 寄存器、返回值会被保存在 v0-v1 寄存器

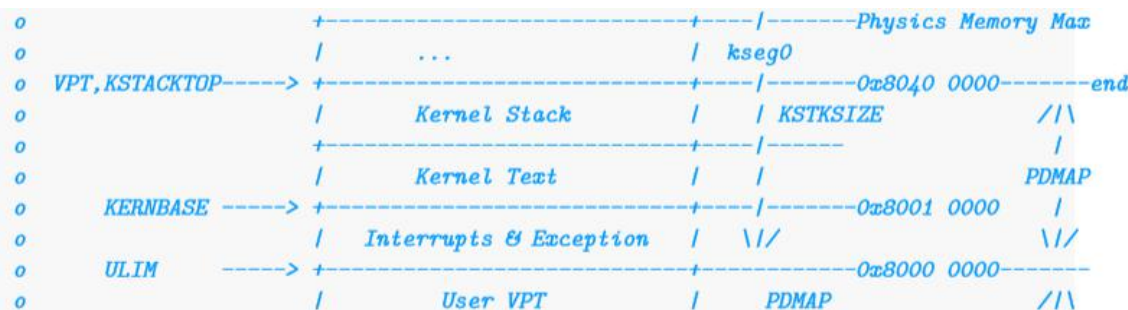
## ■ 循环与判断

- 将循环等高级结构，用判断加跳转的方式替代。
- 判断和循环主要采用 slt、slti 判断两数间的大小关系，再结合 b 类型指令根据对应条件跳转。以这些指令为突破口，我们就能大致识别出循环结构、分支结构了。



## Exercise 1.4 设置栈指针跳转到主函数

- 链接后的程序执行的第一条指令称为 entry point, 在 Linker Script 中已经通过 ENTRY(\_start) 指令来设置了程序入口:
  - `_start`函数。该入口函数定义在 `boot/start.S` 文件中, 补全其空缺的部分。设置栈指针, 跳转到 `main` 函数。
- 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。
  - 设置完栈指针后, 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。
- 使用 `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)。



# 特别提示!!!

若运行 `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) 命令, 得到如下所示反馈的话, 请使用 `gxemul` 的绝对路径。

```
gxemul: 未找到命令
```

即

```
/OSLAB/gxemul -E testmips -C R3000 -M 64 elf-file
```

```
/OSLAB/gxemul -E testmips -C R3000 -M 64 gxemul/vmlinux
```

# Exercise 1.5 实现printf字符输出

- printf 函数全部由 C 语言的标准库提供，而 C 语言的标准库建立在操作系统基础之上，当我们开发操作系统的时候就失去了 C 语言标准库的支持。所以，要在内核中实现一个 printf 函数，需要用到的几乎所有功能性函数都需要我们自己来实现。
- 阅读下列三个文件中的代码：
  - lib/printf.c
  - lib/print.c
  - drivers/gxconsole/console.c

以及指导书中对于函数规格的说明，补全lib/print.c中lp\_Print() 函数缺失的部分来实现字符输出。

# Exercise 1.5 实现printf字符输出

- `/* scan for the next '%' */`
- `/* flush the string found so far */`
- `/* check "are we hitting the end?" */`

# Exercise 1.5 实现printf字符输出

```
/* we found a '%' */
/* format specifiers: %[flags][width][.precision][length]specifier */
//*****
fmt++;

longFlag=0;
negFlag=0;
width=0;
prec=0;
ladjust=0;
padc=' ';
length=0;
```

# Exercise 1.5 实现printf字符输出

- 格式符的原型为：

`%[flags][width][.precision][length]specifier`

flag	描述
-	在给定的宽度 (width) 上左对齐输出，默认为右对齐
0	当输出宽度和指定宽度不同的时候，在空白位置填充 0

# Exercise 1.5 实现printf字符输出

- 格式符的原型为：

`%[flags][width][.precision][length]specifier`

---

width

描述

---

数字 指定了要打印数字的最小宽度，当这个值大于要输出数字的宽度，则对多出的部分填充空格，但当这个值小于要输出数字的宽度的时候则不会对数字进行截断。

---

# Exercise 1.5 实现printf字符输出

---

.precision

描述

---

. 数字      指定了精度，不同标识符下有不同的意义，但在我们实验的版本中这个值只进行计算而没有具体意义，所以不赘述。

---



# Exercise 1.5 实现printf字符输出

- 格式符的原型为：

`%[flags][width][.precision][length]specifier`

length	Specifier	
	d D	b o O u U x X
l	long int	unsigned long int

# Exercise 1.5 实现printf字符输出

- 格式符的原型为：

`%[flags][width][.precision][length]specifier`

Specifier	输出	例子
b	无符号二进制数	110
d D	十进制数	920
o O	无符号八进制数	777
u U	无符号十进制数	920
x	无符号十六进制数，字母小写	1ab
X	无符号十六进制数，字母大写	1AB
c	字符	a
s	字符串	sample

第二处缺失部分：取出参数，输出格式串为“`%[flags][width][.precision][length]d`”的情况

# Lab1课下运行结果

- 如果你正确地实现了前面所要求的全部内容，你将在 gxemul 中观察到如下输出。

```
1  GXemul 0.4.6    Copyright (C) 2003-2007  Anders Gavare
2  Read the source code and/or documentation for other Copyright messages.
3
4  Simple setup...
5      net: simulating 10.0.0.0/8 (max outgoing: TCP=100, UDP=100)
6          simulated gateway: 10.0.0.254 (60:50:40:30:20:10)
7              using nameserver 202.112.128.51
8      machine "default":
9          memory: 64 MB
10         cpu0: R3000 (I+D = 4+4 KB)
11         machine: MIPS test machine
12         loading gxemul/vmlinux
13         starting cpu0 at 0x80010000
14 -----
15
16 main.c: main is start ...
17
18 init.c: mips_init() is called
19
20 panic at init.c:24: ~~~~~
```

# 课下测试Lab1-2评测说明

```
remote: [ PASSED:3 ]  
remote: [ TOTAL:3 ]  
remote: [ You have passed all testcases of default printf. ]
```

```
remote: [ PASSED:5 ]  
remote: [ TOTAL:5 ]  
remote: [ You have passed all testcases of extra printf. ]
```

```
remote: [ PASSED:22 ]  
remote: [ TOTAL:22 ]  
remote: [ You have passed all testcases of powerful printf. ]
```

```
[ You got 100 (of 100) this time. Thu Mar 11 20:04:21 CST 2021 ]
```

正确完成Exercise1.1~1.5后提交代码，可以看到如上图所示评测结果，得到100分。

注意，课下测试为大家提供了三个层次的printf测试，但并不能保证覆盖所有的测试情况，请自己做好相关测试，以免挖坑影响后续lab的完成。

# 特别说明!!!

Lab1-1: Exercise1.1~1.2

第一次课上测试范围: Lab1-1相关内容

提示: 第一次课上测试前请务必完成*Exercise1.1*及*1.2*, 并认真研读*ELF*手册;

Lab1-2: Exercise1.3~1.5

第二次课上测试范围: Lab1-1及Lab1-2相关内容

至此，恭喜完成lab1所有课下任务！  
祝实验顺利！