Lab1 实验报告

> 姓名：文柳懿
> 学号：21351002

Thinking

Thinking 1.1

1.原生 `x86` 工具链：
采用 `gcc` 编译： `gcc -o hello hello.c` 将 `hello.c` 编译为 `x86` 机器代码，并生成名为 `hello` 的可执行文件。
执行 `readelf -h hello`，可见系统架构为： `Advanced Micro Devices X86-64`.
执行 `objdump -d hello`，显示 `hello` 文件格式为 `elf64-x86-64`.

2.MIPS 交叉编译工具链：
采用 `mips-linux-gnu-gcc` 编译： `mips-linux-gnu-gcc -o hello hello.c` 将 `hello.c` 编译为 `MIPS` 机器代码，并生成名为 `hello` 的可执行文件。
执行 `mips-linux-gnu-readelf -h hello`，可见系统架构为： `MIPS R3000`.
执行 `mips-linux-gnu-objdump -d hello`，显示 `hello` 文件格式为 `elf32-tradbigmips`.

Thinking 1.2

原因：
`Makefile` 文件中，执行 `make readelf`,编译为**64位**对象文件：

```
cc -c readelf.c
cc main.o readelf.o -o readelf
```

执行 `make hello`，编译为**32位**文件：
`cc hello.c -o hello -m32 -static -g`

而我们编写的 `readelf.c` 针对32位架构。因此可以解析 `hello`,无法解析 `readelf`.
如果在编译、链接时都使用 `-m32` 选项，即:

```
cc -m32 -c main.c -o main.o
cc -m32 -c readelf.c -o readelf.o
```

```
cc main.o readelf.o -o readelf -m32 -static -g
```

此时，我们编写的`readelf.c`可以解析`readelf`,即：`./readelf readelf`输出节头表信息。

Thinking 1.3

`Bootloader`启动步骤：

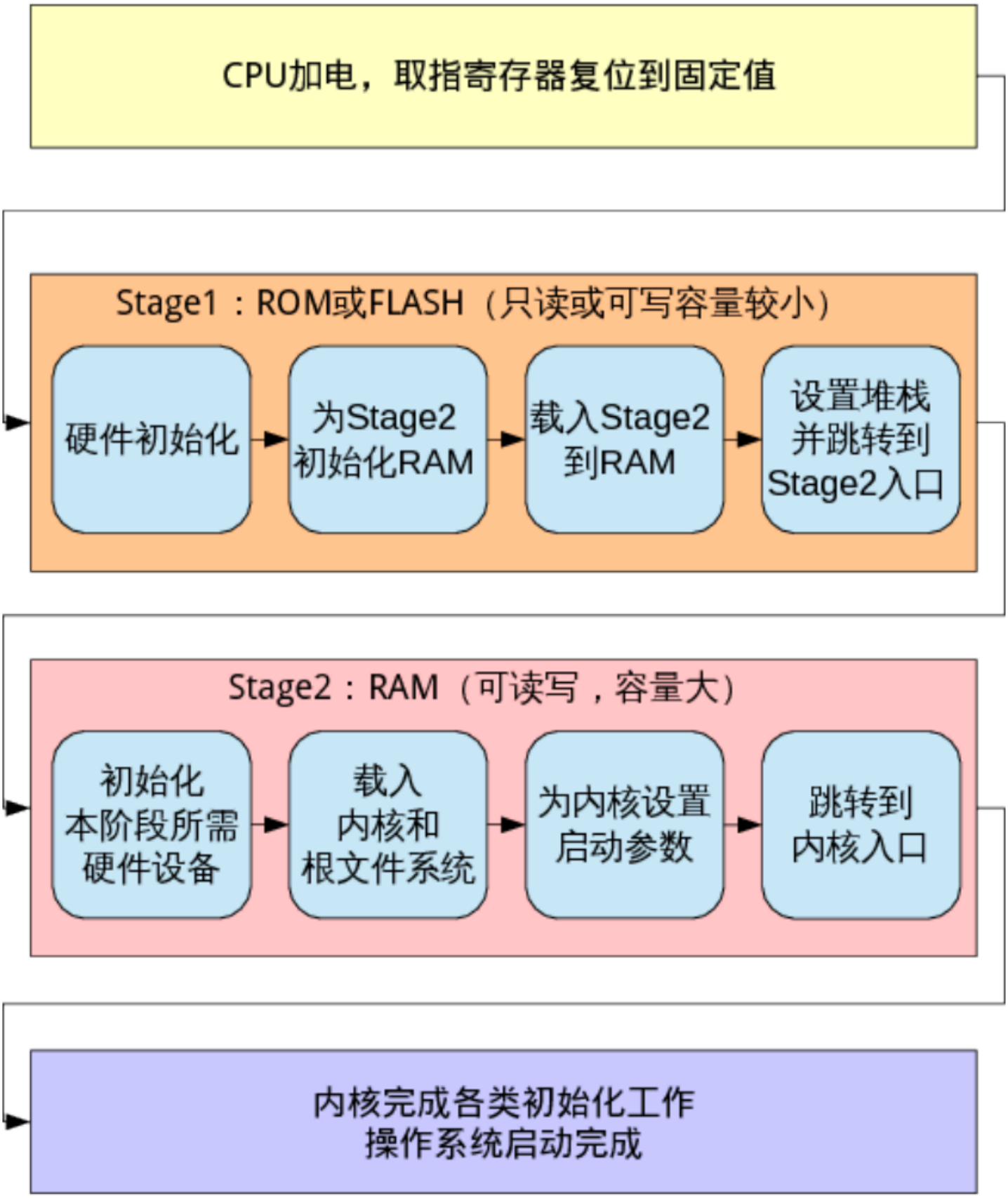


图 A.1：启动的基本步骤

MIPS体系上电时，启动分为两个阶段执行：

- 第一阶段：(stage1+stage2)引导加载程序(U-boot)完成硬件环境的初始化和内核映像的加载，将控制权移交内核。
- 第二阶段：内核本身执行。
  第一阶段中，引导加载程序会解析内核映像的信息，找到**内核入口点地址**，将其写入到CPU的某个寄存器中（如跳转寄存器）；通过执行**跳转**指令，CPU会从该寄存器中读取地址，并跳转到内核入口点开始执行。
  因此，尽管内核入口没有放在上电启动地址，但由于引导加载程序的正确配置和跳转，保证可以跳转至内核入口。

Exercise

Exercise 1.1

要求：输出ELF文件(可执行文件`hello`)所有节头的地址。

- 1.读取ELF文件至`binary`数组;

```
const void *sh_table=binary;
```

- 2.从文件头`Elf32_Ehdr`获取节头表入口相对文件头的偏移`e_shoff`,节头表项个数`e_shnum`.`elf.h`定义的结构体：

```
typedef struct{
    Elf32_Off e_shoff;      //节头表与文件头的偏移
    Elf32_Half e_shnum;     //节头表表项数
    Elf32_Half e_shentsize; //节头表表项大小
}Elf32_Ehdr;      //ELF文件头
```

- 3.读取每个节表项：读取节头的大小`sh_entsize`或`sizeof(Elf32_Shdr)`，随后以指向第一个节头的指针（即节头表第一项的地址）为基地址:`(Elf32_Shdr *)binary+ehdr>e_shoff`，不断累加得到每个节头的地址。
  通过地址读取，转为`Elf32_Shdr *`类型

```
    const Elf32_Shdr *shdr=(Elf32_Shdr *)
(binary+ehdr>e_shoff+sizeof(Elf32_Shdr)*i);
```

- 4.得到节地址

```
    addr=shdr->sh_addr;
```

```
//1. 读取 ELF 文件内容到 binary 数组（类型为 void * ），作为参数传入
int readelf(const void *binary, size_t size) {
    Elf32_Ehdr *ehdr = (Elf32_Ehdr *)binary;        //获取文件头

    // Check whether `binary` is a ELF file.
    if (!is_elf_format(binary, size)) {
        fputs("not an elf file\n", stderr);
        return -1;
    }

    // Get the address of the section table, the number of section headers and the
size of a
    // section header.
//2.从ELF文件头（ehdr）获取节头表的入口偏移（ e_shoff ），节头表的表项个数( e_shnum )
    const void *sh_table;
    Elf32_Half sh_entry_count;
    Elf32_Half sh_entry_size;
    /* Exercise 1.1: Your code here. (1/2) */
    sh_table=binary;
    sh_entry_count=ehdr->e_shnum;   //节头表项数

    // For each section header, output its index and the section address.
    // The index should start from 0.
//3. 由于节头表表项是连续地在文件中存储的，因此 binary + e_shoff 是第一个节头表表项， binary
+e_shoff + sizeof(Elf32_Shdr) * i 是第 i 个表项， binary + e_shoff +
sizeof(Elf32_Shdr) *(e_shnum - 1) 是最后一个表项。
    for (int i = 0; i < sh_entry_count; i++) {
        const Elf32_Shdr *shdr;     //节头表表项
        unsigned int addr;
        /* Exercise 1.1: Your code here. (2/2) */
        shdr=(Elf32_Shdr *)(binary+ehdr->e_shoff+sizeof(Elf32_Shdr)*i);
//4.通过节头表表项(shdr)，获得节地址(sh_addr),节大小(sh_size)
        addr=shdr->sh_addr;
        printf("%d:0x%x\n", i, addr);
    }
    return 0;
}
```

Exercise 1.2

区域	可用性	地址映射	存取
kuseg(2GB)	用户态唯一可	MMU的TLB：虚拟地址->物理地址	通过cache

	用		
kseg0(512MB)	内核态可用	MMU将虚拟地址最高位清零，得到物理地址(连续映射至物理地址低512MB空间)	通过cache
kseg1(512MB)	内核态可用	MMU将虚拟地址高三位清零，得到物理地址(连续映射至物理地址低512MB空间)	不通过cache(使用MIMO访问外设)
kseg2(1GB)	只能在内核态使用	MMU的TLB：虚拟地址->物理地址	通过cache

内核放在`kseg0`.

在`kernel.lds`中调整内核位置(`.text`,`.data`,`.bss`节):
`kernel.lds`记录各节如何映射到段，各段应被加载到的位置。

```
/*
 * Set the ENTRY point of the program to _start.
 */
 //_start函数设置软硬件环境，跳转至 MOS 的初始化函数 ( mips_init )
ENTRY(_start)

SECTIONS {
        /* Step 1: Set the loading address of the text section to the location
counter ".". */
    . = 0x80020000;     //
    .text : { *(.text) }
    .data : { *(.data) }

     bss_start = .;
    .bss : { *(.bss) }
     bss_end = .;
     . = 0x80400000;   //栈顶
     end = . ;
}
```

Exercise 1.3

构建内核:

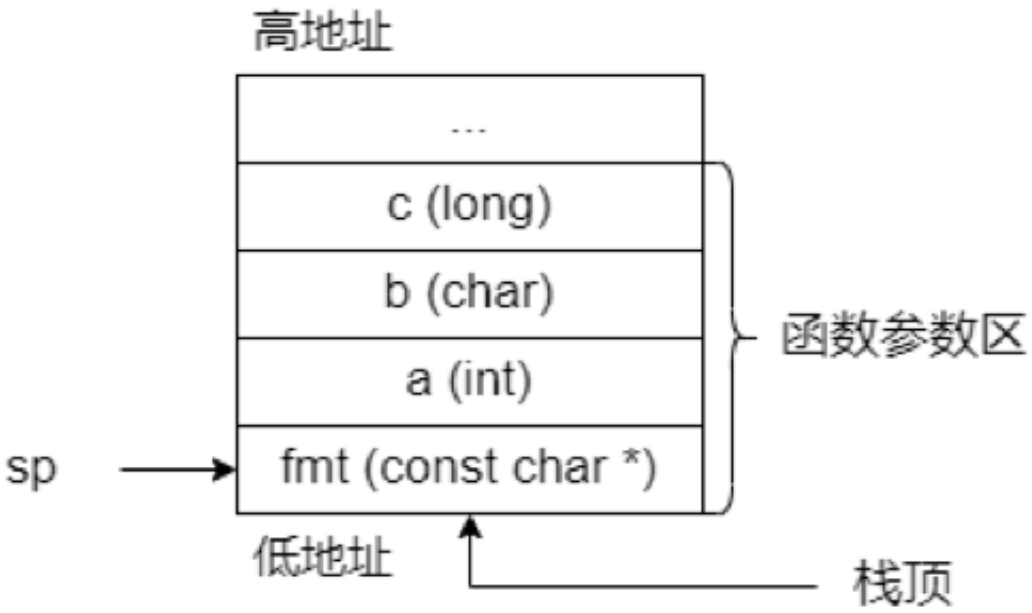
- 1.执行`make`->构建目标`all`->构建`$(targets)`->构建`$(mos_elf)`->构建`$(modules)`

- 2.对 `$(modules)` 中的每个目录执行一次 `$(MAKE) --directory=$@`.
  看见形如：`make[1]: Entering directory '/home/git/xxxxxxxx/init'`的输出.
  `$(modules)`包含：`lib`,`init`,`kern`这三个构建目标。
- 3.`$(mos_elf)`构建：执行 `$(LD) -o $(mos_elf) -N -T $(link_script) $(objects)`.
  `-o --output`:设置输出文件名
  `-T --script`:读取链接脚本
  使用`$(link_script)`将`$(objects)`链接，输出到`$(mos_elf)`位置.
- 4.将组合好的 `$(modules)`和`$(user_modules)` 的内容对应的生成文件赋值给`$(objects)`.
  (1) `objects`:将用户程序和内核程序的目标文件分为不同的后缀保存;
  (2) `modules`:设置 `$(modules)` 为所有需要依赖的构建目标
- 5.`$(mos_elf)`下可查看内核文件.

Exercise 1.4

- C语言的变长参数：

```
printk("%d%c%ld", a, b, c);
//从栈顶依次取出参数：每次需标明类型，以确定后续参数的地址
```



- 格式:`%[flags][width][length]<specifier>`