BUAA_OS_笔记_Lab1

基本语法

`:=` 表示**立即赋值（立即求值）**，也叫 **简单变量赋值。**它和 `=` 的区别是：

写法	名称	求值时机	例子
`=`	延迟赋值	用到时才求值	`A = $(B)`，**B变化A也跟着变**
`:=`	立即赋值	定义时就求值	`A := $(B)`，B当时是多少A就**永远**是多少

`?=` 是"**如果尚未定义 lab，则赋值为...**"

\ 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。

可以使用 `file` 命令来获得文件的类型。

实验概览

系统梳理项目结构，明确任务要求。高亮的是涉及到的补全。我们 Lab1 实验中所有涉及到的代码文件为：`tools/readelf/readelf.c`、`kernel.lds(linker script)`、`init/start.S`、`lib/print.c`。

运行说明：

🌟 在根目录下运行 `make` 可以构建 `target/mos`（内核镜像文件）。也是我们的"最终目标"。

🌟 正确完成 `readelf.c` 之后，在 `tools/readelf` 目录下执行 `make` 命令，即可生成可执行文件 `readelf`，它接受文件名作为参数，对 ELF 文件进行解析。可以执行 `make hello` 生成测试用的 ELF 文件 `hello`，然后运行 `./readelf hello` 来测试 `readelf`。对拍：执行 `readelf -S hello` 命令后，`hello` 文件中各个节的详细信息将以列表的形式为我们展示出来。

🌟 `readelf` 测试：在 `tools/readelf` 里 `make && make hello && ./readelf hello`；`printk` 测试：`make test lab=1_2 && make run`。

工程架构说明：

🌟 `init` 目录中主要有两个代码文件 `start.S` 和 `init.c`，其作用是**初始化内核。** **`start.S` 初始化 CPU 和栈指针，最后跳转到 `init.c` 文件中定义的 `mips_init` 函数。** Lab1 中 `mips_init` 函数只是简单的打印输出，后续会进行扩展。

🌟 `include` 目录中存放系统头文件。 `mmu.h` 文件有一张内存布局图，**在填写 `kernel.lds(linker script)` 和 `init/start.S` 的时候需要根据这个图来设置相应节的加载地址。** `elf.h` 文件定义了几个 `tools/readelf/readelf.c` 中需要用到的结构体。

🌟 `lib` 目录存放一些常用**库函数**，Lab1 主要存放用于**格式化输出**的函数。我们要完成 `lib/print.c`。

🌟 kern 目录中存放内核的主体代码，本章中主要存放的是终端输出相关的函数。

🌟 tests 目录中存放本地测试的测试用例。

启动与 ELF 文件

我们实验中启动的方法：**加载 ELF 格式内核到内存，之后跳转到内核的入口，启动就完成了。**

ELF（Executable and Linkable Format）是一种 Linux 下的一种**用于可执行文件、目标文件和库的文件格式**，它的结构大致是：

```
+-------------------------+
| ELF Header              |  ← Elf32_Ehdr
+-------------------------+
| Program Header Table    |  ← 可选：加载时用
+-------------------------+
| Section Header Table    |  ← 编译器/链接器用来组织节
+-------------------------+
| Section 1               |
+-------------------------+
| Section 2               |
+-------------------------+
| ...                     |
+-------------------------+
```

注意段头表、节头表是独立于段和节之外的，我们这个部分全程只关注表，而不是段或节的具体内容。即：**段头表、节头表并不是段或节本身，而是描述它们的结构体表格。**按照指导书中的定义，明确名称：

🌟段头表（或程序头表，**program** header table），主要**包含**程序中各个段（segment）的信息，段的信息需要在**运行**时刻使用。

🌟节头表（**section** header table），主要**包含**程序中各个节（section）的信息，节的信息需要在程序**编译和链接**的时候使用。

对象名	中文叫法	用于	在 ELF 中位置
Segment	段	**程序运行时加载**	Program Header Table（由 `e_phoff` 指向）
Section	节	**链接、编译、调试用**	Section Header Table（由 `e_shoff` 指向）

下面进入课下解析。

readelf.c

功能是输出 ELF 文件中所有节头中的地址信息（遍历）。

首先，在 `readelf` 函数中有这一行：

```
Elf32_Ehdr *ehdr = (Elf32_Ehdr *)binary;
```

要理解强制类型转换的意思。这句话 **并不会创建新的 `Elf32_Ehdr` 结构体**，它只是告诉编译器：

> **"请把 `binary` 这块内存按 `Elf32_Ehdr` 的字段排列方式来解释。"**

换句话说，它只是"把视角切换为 ELF 头部格式"，但内存本身并没有改变！

🎯 **用示例数据来直观理解**：

假设 `binary` 指向的是 **一个合法的 ELF 文件头部数据**，它的十六进制数据可能是这样：

```
7F 45 4C 46  01 01 01 00  00 00 00 00  00 00 00 00
02 00 03 00  01 00 00 00  80 00 04 08  34 00 00 00
28 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
```

这些数据在 `Elf32_Ehdr` 结构体中的排列方式如下：

```
| e_ident (16字节) | e_type | e_machine | e_version | e_entry | e_phoff | e_shoff | ...
|------------------|--------|-----------|-----------|---------|---------|---------|---
| 7F 45 4C 46 ... | 02 00  | 03 00     | 01 00 00 00 | 80 00 04 08 | 34 00 00 00 | ...
```

🔍 结构体如何解释这块内存

如果我们把 `binary` 解释为 `Elf32_Ehdr` 结构体：

```
Elf32_Ehdr *ehdr = (Elf32_Ehdr *)binary;
```

那么，结构体的字段就会这样解析：

```
ehdr->e_ident[0]  = 0x7F  // ELFMAG0
ehdr->e_ident[1]  = 'E'   // ELFMAG1
ehdr->e_ident[2]  = 'L'   // ELFMAG2
ehdr->e_ident[3]  = 'F'   // ELFMAG3
ehdr->e_ident[4] = 0x01  // ELFCLASS32
ehdr->e_ident[5] = 0x01  // Little endian (小端存储)
...
ehdr->e_ident[15] = 0x00
ehdr->e_type      = 0x0002  // ET_EXEC (可执行文件)
ehdr->e_machine   = 0x0003  // x86 架构
ehdr->e_version   = 0x00000001  // 版本号
ehdr->e_entry     = 0x08040080  // 入口地址
ehdr->e_shoff     = 0x00000034  // Section Header 表的偏移量
```

这里明确一下：为什么上述数据为 `02 00`，下面就变成了 `0x0002`？ 因为是**小端存储**！字段按“低位在前，高位在后”来排放字节。

接着，假设 binary 为 ELF 的文件头地址，`shoff` 为入口偏移，那么 `binary + shoff` 即为节头表第一项的地址。

然后，我们要遍历每个节头表，输出地址。内存中的节头结构是这样：

```
sh_table → +-------------------------+ ← 第 0 个节头
           | Elf32_Shdr              |
           +-------------------------+ ← 第 1 个节头
           | Elf32_Shdr              |
           +-------------------------+ ← 第 2 个节头
           | Elf32_Shdr              |
           +-------------------------+
```

每次循环就：

```
shdr = (Elf32_Shdr *)sh_table + i;
```

指针跳到第 i 个段头，然后访问：

```
shdr->sh_addr
```

去取地址。

再来加深理解一些：节头表本质上就是**一个**每个元素是 `Elf32_Shdr` 类型（每个元素是结构体）的**数组**。数组的每一项描述一个 `Section`。因此，想要遍历所有节，就是遍历整个数组。在 C 语言里，指针加整数 `i` 表示：在原始指针的基础上，跳过 `i` 个结构体的大小。也就是说这行代码：

```
shdr = (Elf32_Shdr *)sh_table + i;
```

就是：**从节头表起始地址开始，跳过 `i` 个 `Elf32_Shdr`，然后取出第 `i` 个节头。**

再通俗一点：

`sh_table` 是一大片混杂的信息。`(Elf32_Shdr *)sh_table` 成功划分出了一个个单元，每个单元是 `(Elf32_Shdr *)sh_table` 结构体类型，因此形成了一个数组。接下来就是取这个数组的每一个元素。

```
sh_table → [节头0][节头1][节头2][节头3]...
             ↑     ↑     ↑     ↑
           i=0   i=1   i=2   i=3
```

本部分总结：

ELF 文件内存布局（简化版）：

```
文件起始地址(binary)
↓
┌──────────────────────┐
| ELF Header (64 bytes)| ← Elf32_Ehdr（结构体）
├──────────────────────┤
| Program Header Table | ← 用于程序加载，可选
├──────────────────────┤
| Section Header Table | ← 我们现在要访问的部分
|  ┌──────────────────┐
|  | Section Header 0 | ← shdr = sh_table + 0
|  ├──────────────────┤
|  | Section Header 1 | ← shdr = sh_table + 1
|  ├──────────────────┤
|  | Section Header 2 | ← shdr = sh_table + 2
|  ├──────────────────┤
|  | Section Header 3 | ← shdr = sh_table + 3
|  └──────────────────┘
├──────────────────────┤
|  Section 的具体内容... |
└──────────────────────┘
```

完全理解了！再整体看一下 Lab1 的第一题（集成版）：

```c
Elf32_Ehdr *ehdr = (Elf32_Ehdr *)binary;    // 读取 ELF 文件头，从中提取节头表的位置

for (int i = 0; i < ehdr->e_shnum; i++) {
    printf("%d:0x%x\n", i, ((Elf32_Shdr *)(binary + ehdr->e_shoff) + i)->sh_addr);
}
```

✔ 整个 ELF 文件的解析过程，**确实就像是"递归下降"解析器的思路**！先用 `Elf32_Ehdr *ehdr = (Elf32_Ehdr *)binary;` 解析 `ELF` 表，再用 `Elf32_Shdr *shdr = (Elf32_Shdr *)sh_table + i;` 解析 `Section` 表。

```
parse_ELF(binary):
    parse_ELF_Header(binary)
    parse_Section_Table(binary + ehdr->e_shoff)
        for each section:
            // maybe you can do more...
```

这和递归下降的逻辑如出一辙！

✅补充一下， **Section Header 的格式是固定的（结构体相同大小），但每个 Section（节）的实际内容长度是不固定的。**

节的内容（比如 `.text`、`.data`、`.bss` 等）是任意长的。长度保存在：

```c
shdr->sh_size   // 当前节的长度（单位：字节）
shdr->sh_offset // 当前节在 ELF 文件中的偏移位置
```

所以你要读取节的内容应该这么做：

```c
const void *section_data = binary + shdr->sh_offset;
size_t section_size = shdr->sh_size;
```

节的格式是不固定的，长度也不固定。

---

kernel.lds

这个任务让我们写 `linker script`。`Linker Script` 是告诉链接器（ld）：**你生成的可执行文件中，每个节（如 .text、.data、.bss）将来要被加载到内存中的哪些位置。**

🧩 `.` 是什么?

> `.` 叫做**定位计数器**，表示**当前位置**（location counter）

可以理解为：

- 编译器在告诉你："我现在正在放内容的位置是多少地址"
- 每写一段东西，`.` 的值就会**自动往后移动**，这里是往**高地址**移动！

这里说明一下为什么一定往高地址堆，我们的内存图如下：

```
 o      KSEG1    -----> +---------------------------+-----------0xa000
0000
 o                     |      Invalid Memory       |   /|\
 o                     +---------------------------+----|-------Physical
Memory Max
 o                     |       ...                 |  kseg0
 o      KSTACKTOP-----> +---------------------------+----|-------0x8040
0000----end
 o                     |       Kernel Stack        |    | KSTKSIZE
   /|\
 o                     +---------------------------+----|------
   |
 o                     |       Kernel Text         |    |
 PDMAP
 o      KERNBASE -----> +---------------------------+----|-------0x8002 0000
   |
 o                     |      Exception Entry      |   \|/
   \|/
```

可以看到，代码节 `Text` 只能从下方 `0x8002 0000` 开始（不然中间那个位置地址也没标啊），因此，代码节、数据节、 `bss` 节依次向上增长，形成**堆**。而下一部分 `start.S` 中要写的汇编中则要求初始化**栈**，是从 `KSTACKTOP` 开始的。

📌 举例：

```
. = 0x80020000;  // 设置当前位置为 0x80020000
.text : {
    *(.text)     // 放入所有 .text 节（代码节）
}
```

这表示：

- 把所有的 `.text` 内容（代码节）放在地址 `0x80020000` 开始
- 放完之后，`.` 会**自动**向后移动，再次强调是往上面的高地址移动。

🧩 `*(.text)` 是什么意思？

> `*` 表示**所有文件**（object files）
> `.text` 表示所有 `.text` 段的内容

也就是说：

`*(.text)` 表示：“把**所有目标文件**中 `.text` 段的内容，通通塞到这里来。”

同样的还有：

```
*(.data)     // 所有目标文件的 .data 段
*(.bss)      // 所有目标文件的 .bss 段
```

这里注意！前后的 `.text` 都是个 **节名（section name）**，不是那个 location counter 的 `.`！以 `.` 开头是一种 **约定俗成的命名规范**，不是强制要求。

可以把这整块看作是：

```
节名 : { 内容 }
```

系统在链接的时候会把这些节按照提供的规则摆放到内存中，而 `.`（location counter）用于控制节的摆放地。

写法	含义
`. = 0x80020000`	设置当前位置为 `0x80020000`（这是“当前位置”的 `.`）✅
`.text : { ... }`	定义一个新的段叫 `.text`（这全都是是 Section 名） ❗
`*(.text)`	把所有 `.o` 文件中的 `.text` 节塞进来

回到我们的实验：

```
. = 0x80020000;    /* 设置第一个节的起始位置，千万记得结尾分号！ */

.text : {          /* 这是定义节 */
    *(.text)       /* 这是节内容 */
}

.data : {
    *(.data)
}
```

第一行是设置 `. = 0x80020000;`，即告诉链接器：我的第一个节从这里开始放。**千万记得结尾分号！**

第二行开始定义 `.text` 节（节的名字叫 `.text`，可以随意取）括号里放的是这个节的实际内容（来自各个 .o 文件）。

还有一个问题：`Set the loading address of the text section to the location counter "."` 时，为什么从 `KERNBASE = 0x80020000` 开始，而不是 `UTEXT = 0x00400000`？

✅ `UTEXT` 是**用户态**程序的地址空间（用户程序的入口）

- 是给用户进程（比如执行 `hello.c` 这样的用户程序）运行用的
- 是属于**用户态地址空间**的一部分（低地址）
- 对应的是段 `UTEXT`、`USTACKTOP`、`UXSTACKTOP` 那一部分

✅ 但是我们现在写的是 **内核** 的链接脚本！

- 是 `kernel.c`、`trap.c`、`pmap.c` ……这些都是**内核代码**
- 内核代码运行在**高地址空间**，`KERNBASE = 0x80020000;` 是**内核虚拟地址空间**的起点

还有一个问题：为什么只有 `.bss` 节需要 `start` 和 `end`？

我们知道 `.bss` 保存未初始化的全局变量和静态变量， `.bss` 在 ELF 文件（硬盘）中**不占空间**，但运行时需要内核**手动清零初始化**，所以要明确它的起始和结束位置。

比如内核启动代码中通常会这么用：

```
extern char bss_start[], bss_end[];
memset(bss_start, 0, bss_end - bss_start);  // 内核自己手动清空 .bss
```

手动清零必须要知道 `.bss` 的范围。

最后的 `. = 0x80400000;` 手动推进地址指针，为之后的数据留空间。

start.S

用一张表明确注意点：

问题	正确用法	原因说明
为什么用 `li` 而不是 `la`	`li sp, KSTACKTOP`	`KSTACKTOP` 是立即数，不是符号地址
为什么用 `j` 而不是 `jal`	`j mips_init`	`mips_init` 是内核主入口，不需要返回
汇编中怎么能调用 C 函数?	链接器解决符号跳转	`mips_init` 是全局符号，链接器会处理地址跳转

为了加深巩固，下面画出一张 **MIPS 启动流程图**，展示 `start.S` 是如何从 `_start` 执行开始，清空 `.bss`，设置栈，然后跳转到 C 函数 `mips_init()`。

```
+------------------------------+
|  start.S: _start 标签        |
|  (Entry point of kernel)     |
+------------------------------+
               |
               v
+----------------------------------------+
| 清空 .bss 段                            |
|                                        |
| la v0, bss_start      ← 加载起始地址     |
| la v1, bss_end        ← 加载结束地址     |
| clear_bss_loop:                        |
|   sb zero, 0(v0)     ← 把0写到地址 v0    |
|   addiu v0, v0, 1                      |
|   beq v0, v1, clear_bss_done           |
|   j clear_bss_loop                     |
+----------------------------------------+
               |
               v
+------------------------------+
| clear_bss_done:              |
| 禁用中断：mtc0 zero, $12     |
+------------------------------+
               |
               v
+------------------------------+
| 设置内核栈指针（栈顶）         |
| li sp, KSTACKTOP             |
| → KSTACKTOP = 0x80400000     |
+------------------------------+
```

```
                |
                v
+-------------------------------+
| 跳转到 C 函数 mips_init()     |
| j mips_init                   |
+-------------------------------+
                |
                v
+-------------------------------+
| C 语言世界开始：mips_init()   |
| → 内存管理初始化              |
| → 页表建立                    |
| → 调用 user 程序              |
+-------------------------------+
```

print.c

这部分是最核心的部分。先明确几个基本概念：

printk.c

在 `print.c` 中的 `vprintfmt` 函数里，我们要填写的代码段中频繁出现 `out` 函数。`out` 是通过**参数表**传进来的一个 **函数指针**，类型是 `fmt_callback_t`。回到 `print.c` 中可以发现其实例化对象为 `outputk`：

```c
/* Lab 1 Key Code "outputk" */
void outputk(void *data, const char *buf, size_t len) {
    for (int i = 0; i < len; i++) {
        printcharc(buf[i]);
    }
}
```

功能是**输出从 `*buf` 开始，长度为 `len` 的字符串。**课下实验中没有用到 `data` 参数。`printk` 函数中对应 `data` 的实参也是 `NULL`。可是，它非常重要！

`data` 是传入 `out` 回调函数（A "callback" is any function that is called by another function which takes the first function as a parameter）的 **上下文指针（context pointer）**。记录了回调函数 `out` 的**额外的上下文信息**（Lab1视频中提到**可以是输出到目标内存地址等等**）。它允许你传入任何你想传递给 `outputk()` 的额外数据。这是一个**抽象设计技巧**，用来支持将来扩展出 **多种输出方式。**

例如未来扩展，要把 `vprintfmt` 的输出：

应用场景	data 可以传入什么?	out 函数（回调）可以怎么用它?
输出到屏幕/串口	NULL	类似 `outputk`，直接 `putchar()`
输出到内存缓冲区	指向一个 struct 或 char 数组指针	`data` 里包含当前写入位置或目标缓冲区指针
输出到内核日志	指向日志缓冲结构体指针	追加日志内容到 ring buffer 中
输出到网络/远程串口	包含 socket、串口句柄的 struct 指针	发送字符串数据到网络/串口设备

💡 举个更复杂的例子：**输出到内存缓冲区**

假设想实现一个 `snprintf()` 内核版本（从工程角度，和 `printk.c` 同级，应该叫做 `snprintk.c`），让格式化结果写到一个**字符数组**中，而不是打印出来，可以这样做：

```c
// snprintk.c

#include <stdarg.h>
#include <stddef.h>
#include <print.h>      // 提供 vprintfmt 声明

// 内部结构：表示写入缓冲区的上下文
struct BufferCtx {
    char *buf;      // 指向目标缓冲区
    size_t pos;     // 当前写入位置
    size_t size;    // 缓冲区总长度（包括 '\0'，注意不是写入总长度）
};

// 回调函数：将字符写入内存缓冲区
static void buffer_out(void *data, const char *str, int len) {
    struct BufferCtx *ctx = (struct BufferCtx *)data;
    for (int i = 0; i < len; i++) {
        if (ctx->pos < ctx->size - 1) { // 预留 '\0'
            ctx->buf[ctx->pos++] = str[i];
        }
    }
    if (ctx->pos < ctx->size) {
        ctx->buf[ctx->pos] = '\0';  // 保证字符串结尾
    } else if (ctx->size > 0) {
        ctx->buf[ctx->size - 1] = '\0';  // 强制终结
    }
}

// 实现类似 snprintf 的函数
int snprintk(char *buf, size_t size, const char *fmt, ...) {
    va_list ap;
    va_start(ap, fmt);
    struct BufferCtx ctx = {
        .buf = buf,
        .pos = 0,
        .size = size
    };
    vprintfmt(buffer_out, &ctx, fmt, ap);
    va_end(ap);
    return ctx.pos;  // 返回写入的字符数（不含终结符）
}
```

在测试模块或 `main()` 中使用：

```c
char buf[100];
snprintk(buf, sizeof(buf), "Hello, %s! num=%d\n", "world", 123);
// buf now contains: "Hello, world! num=123\n"
```

📌 这样就实现了一个“把格式化内容写入内存”的版本！

补充：简化版（`sprintf`）：

```c
#include <stddef.h>
#include <stdarg.h>
#include <print.h>

typedef struct {
    char *addr;        // 当前写入位置
    size_t length;     // 写入总长度
} Data;

static void myoutputk(void *data, const char *str, int len) {
    Data *ctx = (Data *)data;
    for (int i = 0; i < len; ++i) {
        ctx->addr[i] = str[i];     // 写入字符
    }
    ctx->addr += len;              // 更新地址（关键！）
    ctx->length += len;            // 更新写入长度
    *(ctx->addr) = '\0';           // 确保以 '\0' 结尾（关键！）
}
```

```c
int sprintf(char *buf, const char *fmt, ...) {
    va_list ap;
    va_start(ap, fmt);
    Data ctx = {
        .addr = buf,
        .length = 0
    };
    vprintfmt(myoutputk, &ctx, fmt, ap);
    va_end(ap);
    return ctx.length;  // 返回写入的字符数（不含 \0）
}
```

输出格式符解析

首先，我们明确一下 `printf` 中的格式符包含哪些。我们的实验中格式符的原型为：

$$\%[flags][width][length] < specifier >$$

🌟 `flag` 可以没有、为 `-` 或为 `0`。默认右对齐，为 `-` 表示左对齐；为 `0` 表示当输出宽度和指定宽度不同的时候，在空白位置填充 0。举个例子，假如我们后面附加的要输出的整型参数是 42：

格式控制符	输出	宽度	填充字符 `padc`	对齐方式	原因解释
`%5d`	`" 42"`	5	`' '`（默认）	默认为右对齐	宽度为 5，数字占 2 个字符，填充 3 个空格在左侧
`%05d`	`"00042"`	5	`'0'`	右对齐	宽度为 5，数字占 2 个字符，填充 3 个 `0` 在左侧

格式控制符	输出	宽度	填充字符 `padc`	对齐方式	原因解释
`%-5d`	`"42 "`	5	`' '`	左对齐	宽度为 5，数字占 2 个字符，填充 3 个空格在右侧

🌟 `width` 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。**

🌟 `length` 用于修改数据类型的长度，课下实验中我们只使用 `l`，也就是如果为 `l` 就把这个标志位设为 1，以长形式输出。输出 字符、字符串、整数的代码均已提供，我们不用管。

🌟 `specifier` 这次实验中涉及到的总结如下：

Specifier	含义	是否有符号	进制	大小写区别	示例
`b`	无符号二进制数	无符号	2	无区别	`110`
`d`, `D`	有符号十进制整数	**有符号**	10	无区别	`920`
`o`, `O`	无符号八进制整数	无符号	8	无区别	`777`
`u`, `U`	无符号十进制整数	无符号	10	无区别	`920`
`x`	无符号十六进制，小写	无符号	16	输出中使用小写 `abcdef`	`1ab`
`X`	无符号十六进制，大写	无符号	16	输出中使用大写 `ABCDEF`	`1AB`
`c`	单个字符	无	-	无区别	`'a'`
`s`	字符串	无	-	无区别	`"sample"`

提供的 `print_num` 中最后一个 `Upcase` 参数就是用于区分十六进制数的大小写的。其他大小写没有区别。

可以看到，`d`，`D` 均为有符号数，所以要**区别单独处理**！题目中也专门设置了区别正负的标志位。

下面我们回到主线任务：来看 `print.c` 代码。

vprintfmt

功能是实现 `printf` 中的形式化格式的解析和输出到控制台。整体逻辑如下：

```
while (true) {
    ① 从 fmt 中找 '%' 前的普通字符串 -> 输出
    ② 如果到末尾，退出
    ③ 初始化标志变量 ladjust padc
    ④ 解析格式控制符（如 '-', '0', 宽度, 'l'）
    ⑤ 根据格式字符类型执行：
        - 数值格式 -> 取出参数并调用 print_num
        - 字符串格式 -> 调用 print_str
        - 字符格式 -> 调用 print_char
        - 其他 -> 原样输出
    ⑥ 后移 fmt，继续下一轮
}
```

前面扫描到 `%` 的部分我自己重写了，开源答案写的不好。我们从 4/8 开始看：

现在遇到了 `%`，已经把之前东西全输出了，下面吃掉当前 `%` ，进入后面处理。先初始化**标志变量**，标志变量有5个：`width long_flag neg_flag ladjust padc`。**默认右对齐，默认填充字符是空格，默认宽度为0，默认数据类型不是 l（为0），默认符号标志位为0。**

这里关于为什么默认填充字符是空格做进一步说明：

- 第一，C语言标准默认就是填充空格，填充算法在 `print_num` 中也已提供，我们不用管；
- 第二，再次重复，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。**比如执行 `printf("%d", 42);` 会输出42。不用担心会额外填充0或者被截断等问题。
- 第三，C语言中把一个 `char` 类型的变量赋值为 `“”` （空）会报错（java中可以）。

思考1：printk 与 printf

✅ 一句话区别：

> `printf` **是用户态的格式化输出函数，而** `printk` **是内核态的格式化输出函数。**

我们内核实验实现和测试的都是 `printk`。

思考2：可能的扩展

🌟 `readelf.c` 中 `ph_entry_size` 只定义未使用（段的信息需要在**运行**时刻使用，节的信息需要在程序**编译和链接**的时候使用。）；`elf.h` 中 `Elf32_Phdr` 只定义未使用。

🌟 `printk.c` 中的 `outputk` 函数没有使用 `data` 参数。

🌟 对 `case` 进行基础扩展。首先明确：`va_arg(ap, type)` 每调用一次就会**自动读取下一个变参**，你不需要手动做任何偏移操作。

例如：实现一个自定义的格式化字符串'%R',他的格式和%d完全相同，但是输出的值不同，具体地可以说：
`printf("%...R",a,b);` **等价于：** `printf("(%...d,%...d)",a,b);` %R可以从当前参数位往后读取两个参数。 其中,"..."表示用来控制输出格式的那一堆东西，%R的那一堆东西和%d的完全相同。可以这么实现：

```
case 'R':
    if (long_flag) {
        num1 = va_arg(ap, long int);
        num2 = va_arg(ap, long int);
    } else {
        num1 = va_arg(ap, int);
        num2 = va_arg(ap, int);
    }

    if (num1 < 0) {
        num1 = -num1;
        neg_flag1 = 1;
    }
    if (num2 < 0) {
        num2 = -num2;
        neg_flag2 = 1;
    }
    print_char(out, data, '(', 0, 0);
    print_num(out, data, num1, 10, neg_flag1, width, ladjust, padc, 0);
    print_char(out, data, ',', 0, 0);
    print_num(out, data, num2, 10, neg_flag2, width, ladjust, padc, 0);
    print_char(out, data, ')', 0, 0);

    break;      // 别忘了！！！
```

🌞 大端存储怎么转换为小端存储。例如大端`0x12_34_56_78`，对应小端就是`0x78_56_34_12`，将四个字节`reverse`即可。

```
#define REVERSE_32(n_n) \
    ((((n_n)&0xff) << 24) | (((n_n)&0xff00) << 8) | (((n_n) >> 8) & 0xff00) | (((n_n) >> 24) & 0xff))

#define REVERSE_16(n_n) \
    ((((n_n)&0xff) << 8) | (((n_n) >> 8) & 0xff))
```

调试

在根目录下运行`make && make dbg`即可进入调试；

使用`layout asm`显示选择 ASM 的 UI 方式（显示**汇编**代码），可以看到运行所在位置；

使用`break _start`在程序入口处（_start）位置打一个断点；

使用`continue`，可以看到正常进入了程序入口。

`si`：按汇编指令进行单步调试。

`layout src`：按照**源代码**的文本布局对文件进行调试。但是由于 _start 本来就处于汇编代码中，因此在这里执行用源代码布局显示当然不会出现任何东西。因此，回车之后应该在有源代码（C语言）位置处再打一个断点：

`break mips_init`;

然后再`continue`，就可以进入C语言源代码的`mips_init`实现。

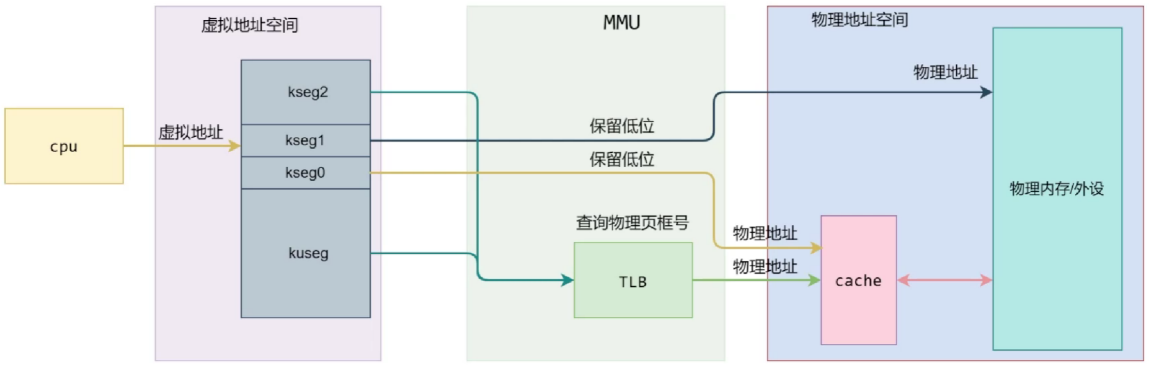
项目 make 选项一览

- `make`：编译产生完整的内核 ELF 文件，不包含任何测试。可以选择在 `init/init.c` 中的 `mips_init` 函数或其他位置，编写自己的测试代码。
- `make test lab=<x>_<y>`：装载 lab<x> 的第 y 个测试用例，编译出相应的内核 ELF。随后可以使用 `make run` 查看运行结果是否符合预期。

  示例：`make test lab=1_2 && make run`
- `make run`：使用 QEMU 模拟器运行内核。
- `make dbg`：使用 QEMU 模拟器以调试模式运行内核，并进入 GDB 调试界面。
- `make objdump`：将项目中的目标文件反汇编。内核的反汇编结果将输出到 `target/mos.objdump` 中。
- `make clean`：清空编译时构建的文件，以待重新编译。

MIPS 内存布局 —— 寻找内核的正确位置

地址空间关系



在操作系统实验中，我们所描述的地址主要是虚拟地址空间，很少直接使用物理地址，注意不要混淆。

tmux 使用

我们在 Shell 下直接输入命令 `tmux`，可以看到终端底部出现一行绿色，这时就已经进入了 tmux 的新会话。tmux 的操作由一系列快捷键组成，下面对重要的快捷键进行介绍。

- Ctrl+B Shift+Num 5（同时按下 Ctrl 和 B，然后松开这两个键，紧接着立刻输入“%”，下面同理），将窗口左右分屏。
- Ctrl+B Shift+'，将窗口上下分屏。
- 重复以上两个快捷键，可以将目前活动的窗格继续分屏。
- Ctrl+B Up / Down / Left / Right 根据按键方向切换到某个窗格。
- Ctrl+B Space，切换窗格布局（上下变成左右，左右变成上下）。
- Ctrl+B X，关闭当前正在使用的窗格（根据提示按 Y 确认关闭）。
- Ctrl+B D，分离（Detach）当前会话，回到 Shell 的终端环境。此时程序仍然保持在 tmux 会话中的状态。

当我们使用 Ctrl+B D 分离了会话或者意外断开了连接，我们该如何恢复到之前的会话中呢？

我们首先使用 `tmux ls` 命令查看当前都有哪些会话。记住会话名（会话名是冒号左边的内容，默认情况下是一个数字），使用 `tmux a -t 会话名` 恢复到原来的会话。

vim 的进阶使用

下列均在命令模式执行：

🌟**复制粘贴：**

v+光标移动 （按字符选择）高亮选中所要的文本，然后进行各种操作（比如，d表示删除）。

用v命令选中文本后，用y进行复制，用p进行粘贴。

🌟**撤销和重做：**

u 撤销，ctrl+R 重做。

重要提醒

- 凡是涉及到 `case` 中的扩展一定要记得 `break` ！
- 用 `gitlab IDE` 编辑，记得先**远程提交**；提交之后，切回跳板机运行前记得 `git pull` ！！