2024 操作系统 Lab1 串讲

2024 操作系统助教组

操作系统的启动

常规的启动流程涉及 bootloader 对软硬件的初始化，十分复杂。

但 MOS 运行的环境是 QEMU 模拟器。 QEMU 支持直接加载 ELF 格式的内核，因此我们只需要把内核编译成正确的 ELF 可执行文件就可以启动了。

使用 QEMU 时，我们在命令中设置 `-kernel` 选项以加载指定的内核 ELF 文件。

```
$ qemu-system-mipsel \
        -m 64 \
        -nographic \
        -M malta \
        -no-reboot \
        -kernel hello_world.elf
```

MOS 内核文件组织

- `init` 目录：内核初始化相关代码
- `include` 目录：存放系统头文件
- `lib` 目录：存放一些常用的库函数，包括 `vprintfmt`
- `kern` 目录：存放内核的主体代码
- `tests` 目录：存放测试程序
- `tools` 目录：存放一些实用工具，包括 `readelf`

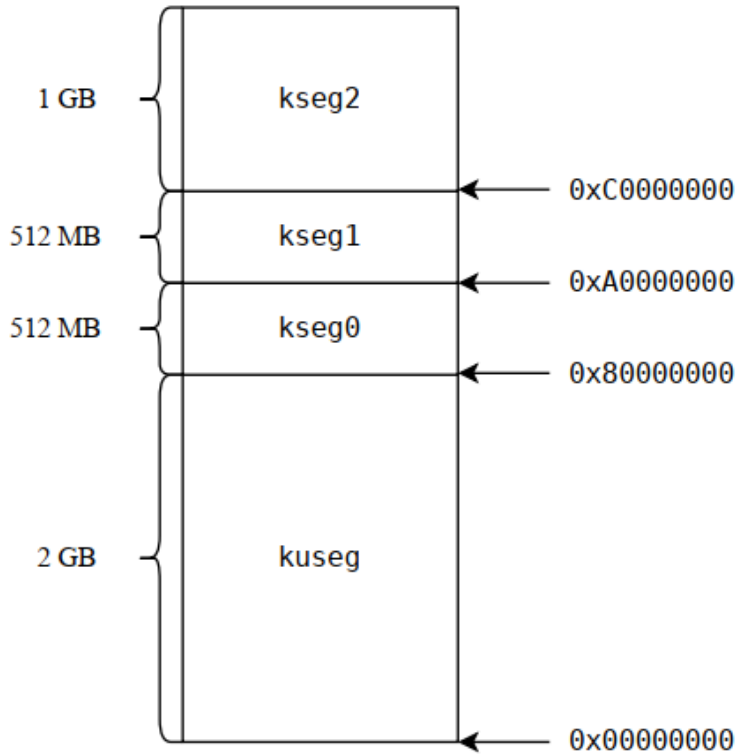
  该目录下的 C 程序使用原生工具链构建（而非交叉编译），在宿主环境（而非 QEMU）下运行
- `target` 目录：存放编译的产物
- `Makefile`：用于编译 MOS 内核的 `Makefile` 文件

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



MIPS 内存布局 —— 寻找内核的正确位置

虚拟地址空间的划分

- `kuseg`：**用户态下唯一可用**的地址空间（内核态下也可用），需要使用 MMU 中的 **TLB** 完成虚拟地址到物理地址的转换。存取都会经过 cache。
- `kseg0`：内核态下的可用地址，MMU 将最高位清零就得到物理地址用于访存。也就是说，这段虚拟地址被**连续地映射到物理地址**的低512MB空间。存取都会**经过 cache**。
- `kseg1`：内核态下的可用地址，高三位清零得到物理地址用于访存。同样连续地映射到物理地址的低 512MB 空间。但是对这段地址的存取**不经过cache**，通常在这段地址上使用 MMIO 技术来访问外设。
- `kseg2`：内核态下的物理地址，**需要MMU中的 TLB** 将虚拟地址转换为物理地址。对这段地址的存取都会经过 cache。

TLE 需要操作系统进行配置管理，因而在载入内核时不能选用需要通过 TLE 的 `kuseg` 和 `kseg2`。而不经过 cache 的 `kseg1` 通常用于访问外设。因此我们**将内核放置在** `kseg0`，其他的区域会在后续实验中用到。

需要注意，`kuseg`、`kseg0`、`kseg1` 以及 `kseg2` 位于**不同的虚拟地址空间**，但是都映射到**同一个物理地址空间**。不同只在于**映射方式**和**访问权限**。

MIPS 内存布局 —— 寻找内核的正确位置

内核的布局

```
/*
 o     4G ----------->  +----------------------------+------------0x100000000
 o                      |       ...                  |  kseg2
 o      KSEG2    -----> +----------------------------+------------0xc000 0000
 o                      |          Devices           |  kseg1
 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       |   \|/                     \|/
 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   |     PTMAP                 |
 o                      +----------------------------+------------0x7f3f f000    |
 o                      |                            |     PTMAP                 |
 o      USTACKTOP ----> +----------------------------+------------0x7f3f e000    |
```

我们为异常处理预留一块空间，将内核镜像的 `.text`、`.data`、`.bss` 这些节安置在 `0x80020000`。同时，由于栈是从高地址向低地址生长的，因此我们为栈预留一块空间，将栈的基地址设置为 `0x80400000`。

Linker Script —— 控制加载地址

Section 的加载

将内核加载到想要的内存地址依赖于 Linker Script。在链接过程中，目标文件被看成 section 的集合，并使用节头表（section header table）来描述各个 section 的组织。换言之， section 记录了链接过程中的必要信息。其中最为重要的三个 section 为 `.text`、 `.data`、 `.bss`

- `.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

```
1  #include <stdio.h>
2
3  char msg[]="Hello World!\n";        .data
4  int count;                          .bss
5
6  int main()
7  {
8      printf("%X\n",msg);
9      printf("%X\n",&count);          .text
10     printf("%X\n",main);
11
12     return 0;
13 }
```

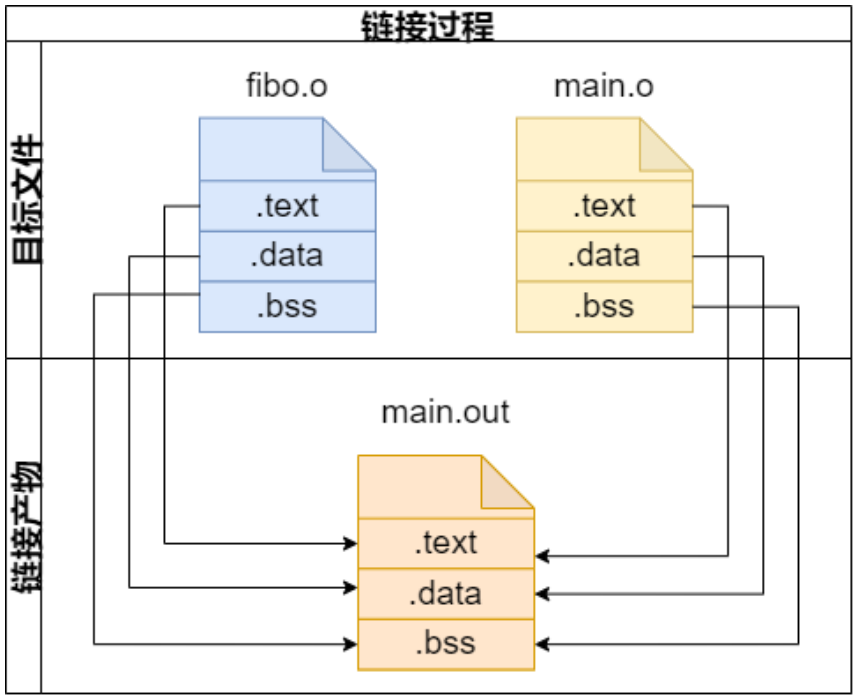
Linker Script —— 控制加载地址

Linker Script 的语法

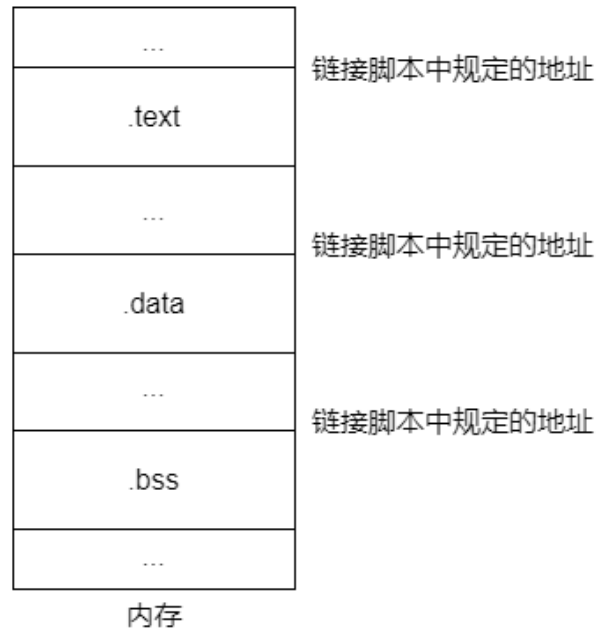
```
SECTIONS
{
    . = 0x10000;
    .text : { *(.text) }
    . = 0x8000000;
    .data : { *(.data) }
    .bss : { *(.bss) }
}
```

- `.` 表示定位计数器，规定当前的位置。通过设定 `.` 的地址即可设置接下来的 section 的起始地址。比如第三行的 `.` 就规定了接下来的 `.text` 需要加载到 `0x10000` 地址处。
- `*` 是一个通配符，表示匹配所有目标文件中相应的节。例如 `.bss:{*(.bss)}` 表示将所有输入文件中的 `.bss` 节（右边的 `.bss`）都放到输出的 `.bss` 节（左边的 `.bss`）中。
- 在 Linker Script 中可以通过 `ENTRY(symbol)` 来设置程序入口。

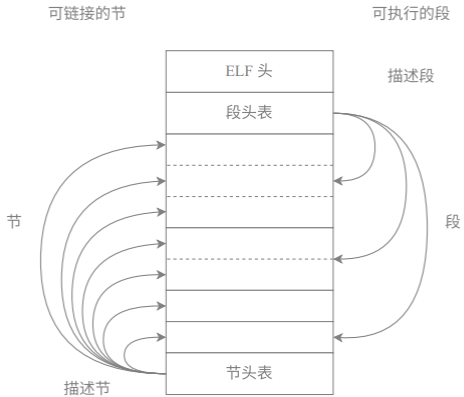
Linker Script —— 控制加载地址



链接脚本指导各个段在内存中的加载位置



ELF 文件的结构



- ELF 文件头：包含程序的基本信息，以及节头表和段头表相对文件的偏移位置。
- 段头表（程序头表）：包含程序中各个段的信息。段的信息需要在**运行时装载**。
- 节头表：包含程序中各个节的信息。节的信息需要在程序**编译链接时使用**。
- 段头表（程序头表）的表项：记录该段数据在文件中的大小、载入内存的位置等，用于指导程序加载。
- 节头表的表项：记录了该节程序的文件内偏移、节地址等信息，主要是链接器在链接的时候需要使用。

示例：32-bit 小端 ELF 文件的解析

ELF 文件头

```
typedef struct {
    // 存放魔数以及其他信息，用于验证ELF文件的有效性
    unsigned char e_ident[EI_NIDENT]; /* Magic number and other info */
    ...
    // 程序头表所在处与此文件头的偏移
    Elf32_Off e_phoff; /* Program header table file offset */
    // 节头表所在处与此文件头的偏移
    Elf32_Off e_shoff; /* Section header table file offset */
    ...
    // 程序头表表项大小
    Elf32_Half e_phentsize; /* Program header table entry size */
    // 程序头表表项数
    Elf32_Half e_phnum; /* Program header table entry count */
    // 节头表表项大小
    Elf32_Half e_shentsize; /* Section header table entry size */
    // 节头表表项数
    Elf32_Half e_shnum; /* Section header table entry count */
    ...
} Elf32_Ehdr;
```

示例：32-bit 小端 ELF 文件的解析

节头表表项

```
typedef struct {
    Elf32_Word sh_name; // 节的名称
    Elf32_Word sh_type; // 节的类型
    Elf32_Word sh_flags; // 节的标志位
    Elf32_Addr sh_addr; // 节的地址(指导链接过程)
    Elf32_Off sh_offset; // 节的文件内偏移
    Elf32_Word sh_size; // 节的大小(以字节计算)
    Elf32_Word sh_link; // 节头表索引链接
    Elf32_Word sh_info; // 额外信息
    Elf32_Word sh_addralign; // 地址对齐
    Elf32_Word sh_entsize; // 此节头表表项的大小
} Elf32_Shdr;
```

示例：32-bit 小端 ELF 文件的解析

遍历 Section Header

1. 读取 ELF 文件内容到 `binary` 数组（类型为 `void *`）。

2. 从文件头（ELF Header）获取节头表的入口偏移（`e_shoff`），节头表的表项个数（`e_shnum`）。

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。

4. 通过节头表表项可以得到节地址（`sh_addr`），节大小（`sh_size`）等信息。

示例：32-bit 小端 ELF 文件的解析

readelf.c

```
int readelf(const void *binary, size_t size) {
    Elf32_Ehdr *ehdr = (Elf32_Ehdr *)binary;
    ...
}
```

疑问：传入的 `binary` 是 `void *` 类型的指针，为什么可以把一个 `void *` 强制转换为 `ELF32_Ehdr *` ？

- 类型只是对某个地址里的内存数据进行的解释，只要符合类型的**大小**和**对齐要求**，一个地址中的数据既可以被看作 `char` ,也可以被看作 `int` 。
- ELF 格式规定了文件起始处含有有效的 ELF 头数据，因此我们就可以把 ELF 数据首部看作一个 `Elf32_Ehdr` 结构体，强制转换之后的指针也就是合法的。
- 指针的算术行为也与其目标类型的大小相关，如 `(Elf32_Shdr *)p + 1` 表示的地址与 `(void *)p + sizeof(Elf32_Shdr)` 相同。

从零开始搭建MOS

`_start` 函数

`_start` 函数的主要任务是设置硬件和软件环境，然后跳转至 MOS 的初始化函数（`mips_init`）。

```
.text
EXPORT(_start)
.set at
.set reorder
    // omit...
    /* disable interrupts */
    mtc0    zero, CP0_STATUS

    /* hint: you can refer to the memory layout in include/mmu.h */
    /* set up the kernel stack */
    /* Exercise 1.3: Your code here. (1/2) */

    /* jump to mips_init */
    /* Exercise 1.3: Your code here. (2/2) */
```

内核栈空间的地址可以在 `include/mmu.h` 中看到，请注意栈的增长方向。

实战 `printk`

C 语言中的变长参数

```
void printk(const char *fmt, ...) {
    va_list ap;
    va_start(ap, fmt);
    vprintfmt(outputk, NULL, fmt, ap);
    va_end(ap);
}
```

- 当函数参数列表末尾有省略号时，该函数即有变长的参数表。由于需要定位变长参数表的起始位置，函数需要含有至少一个固定参数，且变长参数必须在参数表的末尾。
- `stdarg.h` 头文件中为处理变长参数表定义了一组宏和变量类型如下：

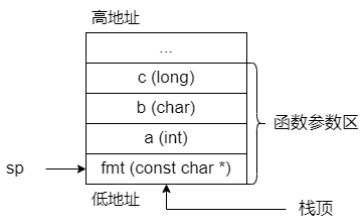
  1. `va_list` ，定义变长参数表的变量类型，代码中的 `ap` 就是 `va_list` 类型的；
  2. `va_start(va_list ap, lastarg)` ，用于初始化变长参数表的宏；
  3. `va_arg(va_list ap, 类型)` ，用于取变长参数表**下一个参数**的宏；
  4. `va_end(va_list ap)` ，结束使用变长参数表的宏。

实战 `printk`

C 语言中的变长参数

- 通过 MIPS 汇编我们可以了解到，参数是在栈上放置的，具体的实现依赖平台的规定。如果按照右边参数先入栈的顺序，那么按照下面的例子：

```
printk("%d%c%ld", a, b, c);
```



- 从栈顶到栈底参数的顺序依次是 `fmt`、`a`、`b`、`c`。由于变长参数占用的空间是不确定的，因此其必须跟在最后一个参数后面，以便用最后一个参数的位置去定位变长参数的位置。

- 首先，以 `fmt` 位置加上 `sizeof(int)`（同时考虑对齐）就得到 `a` 的位置，然后再加上 `sizeof(int)` 就得到 `b` 的位置，依次类推又得到 `c` 的位置。

实战 `printk`

C 语言中的变长参数

- `b` 对应格式化字符串中的 `%c`，之所以其位置要加上 `sizeof(int)` 而不是 `sizeof(char)`，是因为由于 C 语言中的**整数提升**，即使变量 `b` 本身是 `char` 类型，作为实参传入时也会被隐式转换为 `int`。
- 由于类型提升的存在，C 语言中的数值类型在作为右值使用（如参与算术运算、作为实参传递等）时往往都会发生提升，因此在使用 `char` 和 `float` 等类型时需要注意。具体细节可以阅读 C 参考手册中的 隐式转换。
- 可见，变长参数是通过栈空间实现的。由于需要确定后续参数的地址，所以每次取参数都需要标明类型，如下所示：

```
if (long_flag) {
    // 下一个参数类型是 long int
    num = va_arg(ap, long int);
} else {
    // 下一个参数类型是 int
    num = va_arg(ap, int);
}
```

实战 `printk`

`vprintfmt` 函数

```
void vprintfmt(fmt_callback_t out, void *data, const char *fmt, va_list ap) {
    ...
}
```

- `vprintfmt` 是一个公共链接库函数，用于解析格式化字符串，通过调用**回调函数** `out` 完成输出。这样，实现了解析逻辑和输出逻辑的解耦，使程序更加可维护；同时 `printk` 等上层函数可传入不同的回调函数实现**不同的输出行为**（比如输出到文件）。
- `data` 参数是回调函数 `out` 需要的额外上下文信息，需要被 `vprintfmt` 按原样传入 `out` ，可以是**输出的目标内存地址**等。

可以类比面向对象语言中的设计，将 `out` 视为“继承自接口的方法实现”，`data` 则类似方法中的 `this` 指针。

实战 printk

解析格式化字符串

- 格式：`%[flags][width][length]<specifier>`
- `vprintfmt` 中定义了一些解析格式化字符串时需要用到的变量：

```
int width; // 标记输出宽度
int long_flag; // 标记是否为 long 型
int neg_flag; // 标记是否为负数
int ladjust; // 标记是否左对齐
char padc; // 填充多余位置所用的字符
```

- `vprintfmt` 函数的主体是一个循环。这个循环中，主要有两个逻辑部分，第一部分：找到格式符 `%` ，并分析输出格式；第二部分，根据格式符分析结果进行输出。
- 我们在使用 `printf` 输出信息时，`%ld`、`%-b` 等等会按相应输出格式被替换为变长参数的值，这些就是格式符。在解析 `fmt` 字符串时，如果遇到了不需要转换为变量的字符，就直接输出。但如果遇到了以 `%` 开头的格式符，就要按规则解析并输出。常见的规则有是要左对齐还是右对齐，是否为 `long` 型，是否有输出宽度要求等。

实验正确结果

- **基础测试**

  使用 `make && make run` 构建并运行内核：

  ```
  init.c: mips_init() is called
  ```

- **进阶测试**

  - 使用 `make test lab=1_2 && make run` ，加载测试点 1_2 进行自动测试：

  ```
  999999
  12
  -12
  -1234
  00012
  12
  -12
  -0012
  11111111111111111111111111111000
  ...
  ```

课下习题提示

- Exercise1.1 完成 `readelf.c`

  查看指导书中关于 ELF 格式的部分，重点关注 `Elf32_Ehdr` 结构体的 `e_shoff` 和 `e_shnum` 字段

- Exercise1.2 填写 `kernel.lds` 中空缺的部分

  查看 `include/mmu.h` 中的内存布局图

- Exercise1.3 完成 `init/start.S`

  栈指针 `sp` 的位置可参考 `include/mmu.h` ，注意栈空间从高地址向低地址增长

- Exercise1.4 完成 `vprintfmt()` 函数

  1. 格式化字符串的处理规则可以参见指导书附录
  2. 在 Exercise 1.4 第 8 个空处，对 `%d` 的处理可以参考对 `%b` 、 `%o` 、 `%u` 、 `%x` 的处理来实现