Lab3 释疑解惑

张万聪

2020 年 4 月 4 日

目录

一、实验概述

1、基本步骤

1、为进程控制块数组 Envs 分配内核空间（mips_vm_init，alloc）；

2、将 Envs 中各元素串起来，放到空闲链表 env_free_list 中（env_init）；

3、创建进程（env_create），分为以下两步：

- 进程初始化（env_alloc），先为进程初始化页表（env_setup_vm），再分配一个进程号（mkenvid）
- 加载二进制镜像（load_icode），也就是按页分配内存空间（注意这里的空间是用户空间）并将 ELF 的二进制码载入分配好的内存空间中（load_elf），另外还要为进程栈空间分配一页的内存，并设置程序入口 PC 值。

4、准备运行进程（env_run）。运行进程需要如下操作：

- 将当前进程 curenv 的寄存器现场存入 Trapframe 中，并将其运行位置 pc 设置为异常返回地址 cp0_epc
- 当前进程切换为需要运行的进程，并将其状态设置为可调度 ENV_RUNNABLE
- 切换进程的内存空间（页表）（lcontext）
- 恢复待运行进程的上下文，并从异常返回（env_pop_tf）

5、处理中断

- 异常向量组已经为我们设置好，不用管
- 异常分发代码填写到 boot/start.S 的合适位置
- 修改链接脚本 lds，使其可以在处理异常时跳转到异常处理代码
- 打开时钟中断（kclock_init）

6、运行进程：以上步骤都做对之后，就可以对创建好的进程进行 env_run 操作。

7、实现进程调度算法：1-5 步已经实现进程创建、运行操作和中断异常机制。可以让 CPU 发生时钟中断时切换进程，这样就能实现并发。现在唯一需要填写的是 sched_yield 函数，实现时间片轮转调度算法，下面会详细讲解。

2、基本数据结构--进程控制块

查找 include/env.h，可以找到这个原始定义：

```
struct Env {
        struct Trapframe env_tf;        // 异常发生时保存现场区域
        LIST_ENTRY(Env) env_link;       // 空闲进程链表指针域
        u_int env_id;                   // 进程号
        u_int env_parent_id;            // 父进程号
        u_int env_status;              // 进程状态，包括
                                    //   ENV_FREE--空闲，
                                    //   ENV_RUNNABLE--可调度，
                                    //   ENV_NOT_RUNNABLE--阻塞
        Pde  *env_pgdir;                // 进程页目录`内核虚拟地址`
                                      //   注意不是用户虚拟地址！
                                      //   对于用户来说页目录
                                      //   虚拟地址就是 0x7fdff000
        u_int env_cr3;                  // 进程页目录物理地址
        LIST_ENTRY(Env) env_sched_link; // 待调度进程链表指针域
    u_int env_pri;                  // 进程优先级，或时间片长度
    ...
        // 以下为 lab4 及以后需要接触的字段，这里略去
};
```

还有一个 Trapframe 结构体，它是用于发生异常时保存现场的，我们来找一下：

```
struct Trapframe {
        /* 32 个通用寄存器 */
        unsigned long regs[32];

        /* 特殊寄存器 */
        unsigned long cp0_status; // CP0 状态寄存器
        unsigned long hi; // 乘（除）法高位（模）寄存器
        unsigned long lo; // 乘（除）法低位（商）寄存器
        unsigned long cp0_badvaddr; // 异常发生地址
        unsigned long cp0_cause; // CP0 cause 寄存器
        unsigned long cp0_epc; // 异常返回地址
        unsigned long pc; // 程序运行地址
};
```

二、基本函数调用关系

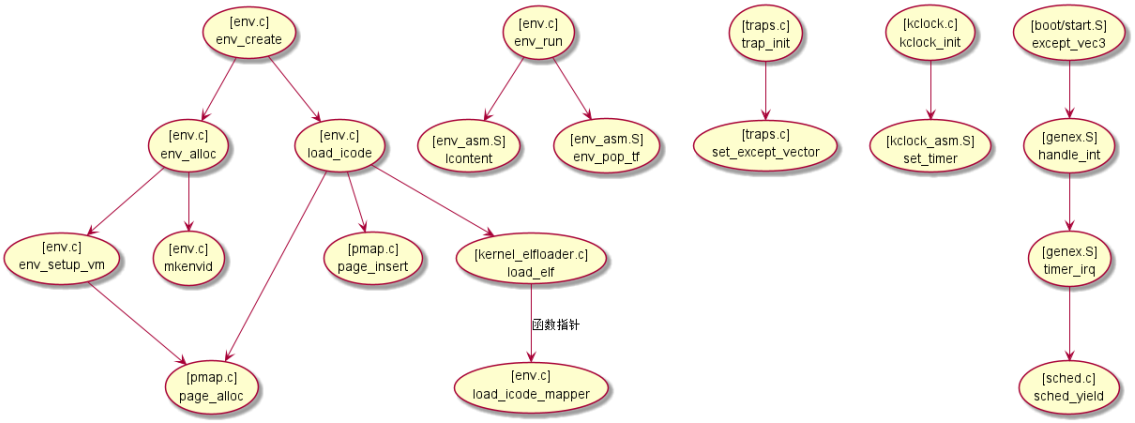

图 1:

三、填写的函数详解

在看这个详解之前，一定要把上图函数调用树厘清，不然下面就会一头雾水。

1、mips_vm_init

前两步在 lab2 中已经写过，现在需要写的就是第 3 步了，明显的照猫画虎，初始化 `Envs` 数组。注意页对齐，还有，我们这是内核态，所以页表的映射关系应当用内核态映射函数完成。

2、env_init

这个答案已经出来了，要初始化进程空闲链表，不用多说了吧。唯一要注意的是插入顺序，如何实现 $O(n)$ 时间的插入呢？

3、env_setup_vm（难点）

这个函数用于初始化进程的页表，首先要初始化页目录，然后就是页目录项的权限问题。这里有一个重要问题：页目录初始化的时候为什么要以内核页目录为模板。我们看到注释中提示：将 `PDX(UTOP)` 之前的页目录项都初始化为 0，之后的页目录项照搬内核页目录。但是，页表区，也就是 `UVPT` 除外，需要单独设置。

我们来回顾一下 lab2。其中 `UTOP` 以上的空间，从低地址到高地址分别为：进程控制块、页表，页表的顶端就是用户区的顶端；然后再往上是内核空间的中断异常、代码段、页表等等。

注意到 UVPT 是进程页表区，其中有一个字节刚好为页目录项，要将其赋值为进程的页目录物理地址。另外，请你思考：页表区是谁在管理？可以给用户写权限吗？

至于 UTOP 以上的其的区域为什么照搬内核页目录为模板，这个问题等你们做到 lab4 系统调用之后，就会彻底明白。

4、env_alloc（难点）

这是新进程初始化的总函数。第一步初始化进程的内存空间，那该用哪个函数呢？我们刚才费了那么大力气，现在该用了吧。第二步就是进程号，好好读代码，哪个是生成新进程号的函数。

可是这完了吗？并没有，只有内存和进程号的进程是跑不起来的。我们还需要给它设置运行前的参数。一直未用的字段 env_status，env_parent_id，env_tf 这些，分别表示进程运行状态、父进程 id 号和保存的寄存器现场。好好思考一下该设置成什么。尤其是 env_tf，这是保存运行现场的，运行现场也就是各种寄存器值和 PC 值。它的 cp0_status 寄存器（还记得计组中的东西吧，CP0 也叫协处理器，一般和中断异常有关）已经帮我们设置好，而栈顶位置却未设置。回顾一下 MIPS 体系结构中，栈指针位置存放在哪个寄存器？我们应该把栈的初始位置，也就是栈顶设置到哪里（回顾 MIPS 内存空间地图）？注意，**进程中的栈是用户栈还是内核栈？不要搞混了哦！**

另外，思考一下，PC 值是在这里设置的吗？如果不是，应该在哪里设置？（提示：回顾 lab1 中编译链接的过程，link script 中设置了 start 点，也就是说链接器需要使用这个设置来生成 ELF 文件，那么程序入口显然是从 ELF 文件本身中获得的。而目前我们尚未加载 ELF 文件，所以现在设置 PC 的初始值为时尚早）

最后别乐极生悲，行百里者半九十，记不记得 env_free_list 里面都是怎样状态的进程？一个进程经过 env_alloc 之后，它还应该在这里吗？**如果不应该，请把它从这里移除**。

5、load_icode_mapper（难点中的难点）

这个函数集前三个 lab 知识之大成，一定要熟悉 lab1 中的 ELF 结构和 lab2 中内存分配的方法！

在填写这个函数之前，要先知道它是干什么的。我们现在已经完成实验的第三步 --- 创建进程中的第一小步，现在已经有了空进程，但只有骨架没有灵魂怎么行？一个进程是一个程序的一次运行，所以现在就要把**程序装进给进程分配的内存空间中**。

这一任务由 load_icode 函数来完成，它的步骤就是分配内存，并将二进制代码装入分配好的内存中。但它一个函数要承担这么大的任务，有点吃不消啊。

所以它就把装入内存的任务交给了 load_elf 函数。但装入内存的任务还是有点艰巨，不仅要**解析 ELF 结构**，还要把 ELF 的内容**复制到分配好的内存**中。这函数比较懒，就又把内容

复制的任务交给了 load_icode_mapper。这下终于不再嵌套了，好像身体被掏空是不是？别急，先来看看这个函数。这个函数如果完成了，lab3 也就没啥难的了。

这个函数大体上也是两步走，第一步，复制 ELF 的内容（当然，必须是**代码段和全局数据段**）。第二步就是难中之难，给 ELF 的内容分配页面。

现在二进制码长度已经由它的参数 bin_size 传入了。那又跑出一个段长度 sgsize 是什么鬼？还记得 lab1 的 readelf 吧，二进制码长度 bin_size 等于代码段.text 和全局数据段.data 长度之和，但不一定等于 ELF 要占用的内存大小。回顾 lab1 中的 ELF 结构，代码段是通过 program header 定位的，每个头部都有一个 filesize 和 memsize，就分别对应 bin_size 和 sgsize。别忘了 ELF 中还有一个.bss 段哦，这.bss 段是全部要置零的，所以无需在 ELF 中体现，但并不代表它就不占内存。那么请你想一下，bin_size 和 sgsize 满足什么样的不等式？.bss 段的起始地址和 bin_size 是什么关系？这是该函数的第一个难点。

前门狼刚走，后门虎又来。看看函数前面的说明，pre_condition 中有一条提示：va may NOT aligned 4KB。这提示了什么？lab2 中我们实现的是页式内存管理，一页的大小是 4KB（BY2PG），也就是说，一个页的首地址的**十六进制表示的后三位**都是 0。如果 va 不是一个页的首地址，比如 0x0003f2d4，该如何处理呢？那么同样的问题，如果.bss 段的首地址不是页对齐的，又该如何处理呢？提示：善于利用 offset 变量。再给一个提示，**在一段内存不满一个页（4KB）的情况下，仍然要分配一整个页来存储**，就像出租车计费，就算你多走了 100 米也会按 1 公里算。

下面通过一张图来加深理解：

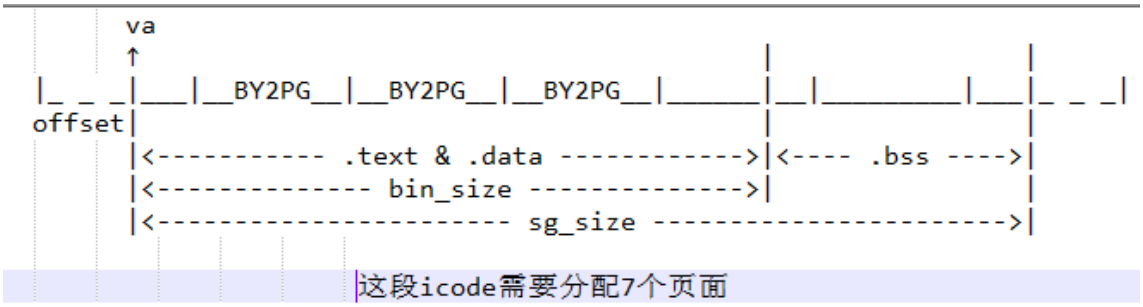

图 2:

6、load_elf

填写完上面那个老大难函数，下面就是 load_elf 了。别找了，就在 lib/kernel_elfloader.c 中。

这个函数需要填写的部分，就是将各个程序段映射到相应的虚拟内存中。回忆刚才我们填写的 load_icode_mapper 函数，再观察一下 load_elf 的最后一个参数 map，它是一个函数指针，其参数表和 load_icode_mapper 完全相同。是不是想到了什么？没错，这就是我们在 load_elf 中需要调用的函数。

再来看下 ELF 文件的程序头结构：

```
typedef struct {
        Elf32_Word        p_type;                    /* Segment type */
        Elf32_Off         p_offset;                  /* Segment file offset */
        Elf32_Addr        p_vaddr;                   /* Segment virtual address,
                                                        这就是需要映射的虚拟地址 */
        Elf32_Addr        p_paddr;                   /* Segment physical address */
        Elf32_Word        p_filesz;                   /* Segment size in file */
        Elf32_Word        p_memsz;                   /* Segment size in memory */
        Elf32_Word        p_flags;                   /* Segment flags */
        Elf32_Word        p_align;                   /* Segment alignment */
} Elf32_Phdr;
```

如何获取每个程序段的偏移量？哪两个参数分别对应 bin_size 和 sgsize？这些是需要自己思考的部分。

时刻记住，我们的工作是把每个程序段的**逻辑虚拟地址 p_vaddr**（还记得 lab1 中编译链接的过程吗？p_vaddr 是链接器经过重定位产生的）一一对应到**分配好的内存**中，所以刚才那个函数起名为 map，意思是“映射”。

最后注意，我们还要从 ELF 文件中提取出进程入口地址，也就是 entry_point，为后面设置 PC 初始值做准备。

7、load_icode

各个文件来回跳，又晕了？回到 env.c 吧。

这是真正的加载二进制镜像的函数，这里，我们就要做运行前的最后一步工作了。

概述里已经说的很清楚了，三步走。首先就是分配进程的运行栈空间，注意这里是进程栈，不是内核栈！为栈空间预分配一个页面。

第二步就是 load_elf。

最后一步，程序运行之前，一定要有一个开始位置，所以，这里才是设置 PC 的正确地点！刚才 load_elf 返回的 entry_point 就是我们想要的了。

好了，接下来就是激动人心的运行进程。

8、env_run

概述中关于 env_run 的步骤已经说的很明白了，四步走。其实精确地讲，这个函数不应该叫做进程运行，而应该叫做进程切换，因为它完成的是 curenv 切换前的准备工作。

首先，进程切换之前必须对运行现场进行处理，因为 env_run 不一定是首先开始运行某个进程，而有可能是其他进程运行一段时间之后，又切换到该进程运行。而如果是从其他进程切换而来的，那么就要先把之前那个进程的运行现场保存起来。所以简而言之，第一步就是保存现场。而保存现场时，如何获取待保存进程的寄存器信息呢？注释里要我们参考 env_destroy 函数的行为，我们看到其中有一个函数调用 bcopy((void *)KERNEL_SP - sizeof(struct Trapframe), (void *)TIMESTACK - sizeof(struct Trapframe), sizeof(struct Trapframe)); 大概已经明白了，TIMESTACK 附近有异常发生时的寄存器信息。

保存完现场之后，还要把之前那个进程的 PC 值重设一下，因为稍后有可能切换回那个进程，还要接着被中断时的地方运行。计组告诉我们，要把之前那个进程的 PC 值设为 cp0_epc。

其次，毫无疑问，是把当前进程 curenv 切换为需要运行的进程，并把它的状态设置为 ENV_RUNNABLE。

接下来，设置进程的地址空间。别忘了每个进程的地址空间是相互独立的，所以，这一步直接告诉你们吧，就是调用 lcontext 函数，其参数为即将运行的进程的页目录 env_pgdir，注意要强制类型转换为 int 型整数哦。

最后就是恢复现场、异常返回了。这一步就是简单地调用 env_pop_tf 函数，写法是 env_pop_tf(&curenv->env_tf, GET_ENV_ASID(curenv->env_id));。至于为什么要用到 GET_ENV_ASID(curenv->env_id)，请你自己思考（提示：阅读 see mips run linux 的 135-144 页，这个设计和 TLB 有关；当然读不懂也没关系，这个和实验关键部分关系不大）。

四、进程调度算法

完成了上面所有的步骤之后，还有一个重要任务，那就是开启中断机制，包括三个步骤：

- 把异常分发代码填写到 boot/start.S 中
- 修改链接脚本 lds，使得中断或异常发生时能够跳转到异常处理代码
- 在 lib/kclock.c 中合适的位置调用 set_timer() 函数，开启时钟中断

接下来就剩下本实验的最后一个任务–进程调度，也就是 sched_yield 函数了。这个函数是用来切换运行的进程的，当然也就是调度算法的实现位置。指导书上让我们实现的是时间片轮转调度算法。

我们首先思考一下，sched_yield 是在内核态还是用户态呢？当然是在内核态，因为内核态是用户态陷入异常之后进入的，切换进程相当于一个从异常返回的过程，显然用户态是不可能从异常返回并切换进程的。

然后我们了解一下这个函数是如何被调用的。当时钟中断发生时，首先跳转到异常处理代码，也就是 boot/start.S 中的 except_vec3 函数。这个函数用来判断异常类型，并根据异常类型跳转到相应的异常处理函数。我们这里异常类型是中断，所以跳转到 lib/genex.S 中的 handle_int

函数。接下来这函数将判断发生的中断是否为时钟中断，如果是时钟中断，将跳转到 lib/genex.S 中的 `timer_irq` 函数。这个函数就会直接跳转到 `dched_yield` 执行，后者将会根据调度算法，执行进程切换和返回到用户态的操作。

了解了进程调度的基本过程之后，我们讲一下时间片轮转算法。**时间片就是时钟中断的周期**，所以每个时间片发生一次中断，进入一次 `sched_yield` 函数。`Env` 结构体有一个字段叫做 `env_pri`，是进程优先级，在这个算法中不能当做“优先级”理解，而应该当作**最长连续时间片**理解，也就是它**最多连续**运行多少个时间片就必须切换为其它进程。

基本步骤如下：

1. 假设有一个待调度进程队列 `q1 = {e1, e2, ..., en}`，最长连续时间片分别为 `t1, t2, ..., tn`（均由对应的 `env_pri` 得出）。还需要另外设一个队列 `q2 = {}`。当前队列指针 `p = &q1`。
2. 检查 `p` 指向队列的第一个进程的 `env_status`，如果为 `ENV_FREE`，则将其移除；如果为 `ENV_NOT_RUNNABLE`，则放入另一个队列尾部；如果为 `ENV_RUNNABLE` 且时间片已用完，则将其时间片恢复为它的 `env_pri`，并放入另一个队列尾部。重复以上步骤，直到当前队列为空，或者找到一个进程，其状态为 `ENV_RUNNABLE` 且时间片未用完。
3. 如果第 2 步结束后当前队列为空，则将当前队列指针 `p` 切换到另一个队列，返回第 2 步。如果两个队列均为空，则 CPU 挂起等待。
4. 将当前队列指针 `p` 指向队列的第一个进程的时间片-1，并调用 `env_run` 运行之。
5. 待时钟中断再次发生时，转到第 2 步。

注意，函数头部 hint 中说善用 `static` 局部变量。静态局部变量的存储方式和全局变量相同，都是在全局数据段中，只不过作用域仅限于函数内部。既然是全局数据段中，那就不随函数消亡而消亡。它只在程序装入时被赋一次初值，然后每次函数运行时就像全局变量一样，不会重新初始化，而是保留上次函数返回时的值。

我们再看一下时间片轮转的过程，由于一个进程的时间片必须是连续消耗完才放入另一个队列（除非中途被阻塞，但 lab3 没有进程间通信，无需考虑这种情况），所以不必保存每个进程的剩余时间片，只保存当前待运行进程的剩余时间片即可。这刚好符合静态局部变量的特点。思考一下如何做。如果实在不会，也可以像上面一样，给 `Env` 结构体加一个字段，来保存每个进程的剩余时间片。

这是一个博客，里面总结了多种进程调度算法可供参考。
https://blog.csdn.net/zh13487/article/details/83928284

填写完这个函数之后，我们的 lab3 就算圆满完成了。