CHAPTER 3

进程与异常

3.1 实验目的

1. 创建一个进程并成功运行
2. 实现时钟中断，通过时钟中断内核可以再次获得执行权
3. 实现进程调度，创建两个进程，并且通过时钟中断切换进程执行

在本次实验中你将运行一个用户模式的进程。你需要使用数据结构进程控制块 Env 来跟踪用户进程。通过建立一个简单的用户进程, 加载一个程序镜像到进程控制块中, 并让它运行起来。同时，你的 MIPS 内核将拥有处理异常的能力。

3.2 进程

> **Note 3.2.1** 进程既是基本的分配单元，也是基本的执行单元。第一，进程是一个实体。每一个进程都有它自己的地址空间，一般情况下，包括代码段、数据段和堆栈。第二，进程是一个“执行中的程序”。程序是一个没有生命的实体，只有处理器赋予程序生命时，它才能成为一个活动的实体，我们称其为进程。

3.2.1 进程控制块

这次实验是关于进程和异常的，那么我们首先来结合代码看看进程控制块是个什么东西。

进程控制块 (PCB) 是系统为了管理进程设置的一个专门的数据结构，用它来记录进程的外部特征，描述进程的运动变化过程。系统利用 PCB 来控制和管理进程，所以**PCB 是系统感知进程存在的唯一标志。进程与 PCB 是一一对应的**。通常 PCB 应包含如下一些信息:

Listing 9: 进程控制块

```
struct Env {
    struct Trapframe env_tf;        // Saved registers
    LIST_ENTRY(Env) env_link;       // Free LIST_ENTRY
    u_int env_id;                   // Unique environment identifier
    u_int env_parent_id;            // env_id of this env's parent
    u_int env_status;               // Status of the environment
    Pde  *env_pgdir;                // Kernel virtual address of page dir
    u_int env_cr3;
};
```

为了集中你的注意力在关键的地方，我们暂时不介绍其他实验所需介绍的内容。下面是 **lab3** 中需要用到的这些域的一些简单说明：

- env_tf：Trapframe 结构体的定义在 **include/trap.h** 中，env_tf 的作用就是在进程因为时间片用光不再运行时，将其当时的进程上下文环境保存在 env_tf 变量中。当从用户模式切换到内核模式时，内核也会保存进程上下文，因此进程返回时上下文可以从中恢复。
- env_link：env_link 的机制类似于实验二中的 pp_link，使用它和 env_free_list 来构造空闲进程链表。
- env_id：每个进程的 env_id 都不一样，env_id 是进程独一无二的标识符。
- env_parent_id：该变量存储了创建本进程的进程 id。这样进程之间通过父子进程之间的关联可以形成一颗进程树。
- env_status：该变量只能在以下三个值中进行取值：
  - ENV_FREE：表明该进程是不活动的，即该进程控制块处于进程空闲链表中。
  - ENV_NOT_RUNNABLE：表明该进程处于阻塞状态，处于该状态的进程往往在等待一定的条件才可以变为就绪状态从而被 CPU 调度。
  - ENV_RUNNABLE：表明该进程处于就绪状态，正在等待被调度，但处于 RUNNABLE 状态的进程可以是正在运行的，也可能不在运行中。
- env_pgdir：这个变量保存了该进程页目录的虚拟地址。
- env_cr3：这个变量保存了该进程页目录的物理地址。
- env_sched_link：这个变量来构造就绪状态进程链表。
- env_pri：这个变量保存了该进程的优先级。

这里值得强调的一点就是 **ENV_RUNNABLE 状态并不代表进程一定在运行中，它也可能正处于调度队列中**。而我们的进程调度也只会调度处于 RUNNABLE 状态的进程。

既然我们知道了进程控制块，我们就来认识一下进程控制块数组 **envs**。在我们的实验中，存放进程控制块的物理内存在系统启动后就要被分配好，并且这块内存不可被换出。所以需要在系统启动后就要为 envs 数组分配内存，下面你就要完成这个重任

Exercise 3.1
- 修改 pmap.c/mips_vm_init 函数来为 envs 数组分配空间。
- envs 数组包含 NENV 个 Env 结构体成员，你可以参考 pmap.c 中已经写过的 **pages** 数组空间的分配方式。
- 除了要为数组 envs 分配空间外，你还需要使用 pmap.c 中你填写过的一个内核态函数为其进行段映射，envs 数组应该被 **UENVS** 区域映射，你可以参考**./include/mmu.h**。

■

当然我们光有了存储进程控制块信息的 envs 还不够，我们还需要像 lab2 一样将空闲的 env 控制块按照链表形式“串”起来，便于后续分配 ENV 结构体对象，形成 env_free_list。一开始我们的所有进程控制块都是空闲的，所以我们要把它们都“串”到 env_free_list 上去。

Exercise 3.2 仔细阅读注释，填写 **env_init** 函数，注意链表插入的顺序 (函数位于 lib/env.c 中)。 ■

在填写完 env_init 函数后，我们对于 envs 的操作暂时就告一段落了，不过我们还有一个小问题没解决，为什么严格规定链表的插入顺序？我们需要你来仔细思考，注释中已经给出了提示。

Thinking 3.1 为什么我们在构造空闲进程链表时必须使用特定的插入的顺序？(顺序或者逆序) ■

3.2.2 进程的标识

你的电脑中经常有很多进程同时存在，每个进程执行不同的任务，他们之间也经常需要相互协作、通信，那操作系统是如何识别每个进程呢？

在上一部分中，我们了解到每个进程的信息存储在该进程对应的进程控制块中，细心的你一定已经发现 struct Env 中的 env_id 这个域，它就是每个进程独一无二的标识符。我们在创建每个新的进程的时候必须为它赋予一个与众不同的 id 来作为它的标识符。

你可以在 env.c 文件中找到一个叫做 mkenvid 的函数，它的作用就是生成一个新的进程 id。

如果我们知道一个进程的 id，那么如何才能找到该 id 对应的进程控制块呢？

Exercise 3.3 仔细阅读注释，完成 env.c/envid2env 函数，实现通过一个 env 的 id 获取该 id 对应的进程控制块的功能。 ■

Thinking 3.2 思考 env.c/mkenvid 函数和 envid2env 函数:

- 请你谈谈对 mkenvid 函数中生成 id 的运算的理解，为什么这么做？
- 为什么 envid2env 中需要判断 e->env_id != envid 的情况？如果没有这步判断会发生什么情况？

■

3.2.3 设置进程控制块

做完上面那个小练习后，那么我们就可以开始利用**空闲进程链表 env_free_list** 创建进程来玩了。下面我们就来具体讲讲你应该如何创建一个进程[1]。

进程创建的流程如下:

第一步 申请一个空闲的 PCB，从 env_free_list 中索取一个空闲 PCB 块，这时候的 PCB 就像张白纸一样。

第二步 “纯手工打造”打造一个进程。在这种创建方式下，由于没有模板进程，所以进程拥有的所有信息都是手工设置的。而进程的信息又都存放于进程控制块中，所以我们需要手工初始化进程控制块。

第三步 进程光有 PCB 的信息还没法跑起来，每个进程都有独立的地址空间。所以，我们要为新进程分配资源，为新进程的程序和数据以及用户栈分配必要的内存空间。

第四步 此时 PCB 已经被涂涂画画了很多东西，不再是一张白纸，把它从空闲链表里除名，就可以投入使用了。

当然，第二步的信息设置是本次实验的关键，那么下面让我们来结合注释看看这段代码

Listing 10: 进程创建

```
1  /* Overview:
2   *  Allocates and Initializes a new environment.
3   *  On success, the new environment is stored in *new.
4   *
5   * Pre-Condition:
6   *  If the new Env doesn't have parent, parent_id should be zero.
7   *  env_init has been called before this function.
8   *
9   * Post-Condition:
10  *  return 0 on success, and set appropriate values for Env new.
11  *  return -E_NO_FREE_ENV on error, if no free env.
12  *
```

[1]这里我们创建进程是指系统创建进程，不是指 fork 等进程“生”进程。我们将在 lab4 中接触另一种进程创建的方式。

```
 * Hints:
 *  You may use these functions and defines:
 *      LIST_FIRST,LIST_REMOVE,mkenvid (Not All)
 *  You should set some states of Env:
 *      id , status , the sp register, CPU status , parent_id
 *      (the value of PC should NOT be set in env_alloc)
 */

int
env_alloc(struct Env **new, u_int parent_id)
{
        int r;
        struct Env *e;

    /*Step 1: Get a new Env from env_free_list*/


    /*Step 2: Call certain function(has been implemented) to init kernel memory
     * layout for this new Env.
     *The function mainly maps the kernel address to this new Env address. */


    /*Step 3: Initialize every field of new Env with appropriate values*/


    /*Step 4: focus on initializing env_tf structure, located at this new Env.
     * especially the sp register,CPU status. */
    e->env_tf.cp0_status = 0x10001004;


    /*Step 5: Remove the new Env from Env free list*/


}
```

相信你一眼看到第三条注释的时候一定会吐槽：“什么鬼，什么叫合适的值啊”。淡定，先别着急吐槽，先花半分钟的时间看一下第二条注释。

env.c 中的 env_setup_vm 函数就是你在第二部中要使用的函数，该函数的作用是初始化新进程的地址空间。在使用该函数之前，我们必须先完成这个函数，这部分任务是这次试验的难点之一。

在你动手开始完成 env_setup_vm 之前，为了便于理解你需要在这个函数中所做的事情，请先阅读以下提示:

在我们的实验中，对于不同的进程而言，虚拟地址 ULIM 以上的地方，虚拟地址到物理地址的映射关系都是一样的。因为这 2G 虚拟空间，不是由哪个进程管理的，而是由内核管理的！如果你仔细思索这句话, 可能会产生疑惑：“那为什么不能该内核管的时候让内核进程去管，该普通进程管的时候给普通进程去管, 非要混在一个地址空间里搞来搞去的呢?”这种想法是很好的，但是问题来了，在我们这种布局模式下，有严格意义上的内核进程吗?

答案当然是否定的，这里我们要再讲清楚几个概念，方可解决这个问题。

首先来科普下虚拟空间的分配模式。我们知道，每一个进程都有 4G 的逻辑地址可以访问，我们所熟知的系统不管是 Linux 还是 Windows 系统，都可以支持 3G/1G 模式或者 2G/2G 模式。3G/1G 模式即满 32 位的进程地址空间中，用户态占 3G，内核态占

1G。这些情况在进入内核态的时候叫做陷入内核，因为**即使进入了内核态，还处在同一个地址空间中，并不切换 CR3 寄存器**。但是！还有一种模式是 4G/4G 模式，内核单独占有一个 4G 的地址空间，所有的用户进程独享自己的 4G 地址空间，这种模式下，在进入内核态的时候，叫做切换到内核，**因为需要切换 CR3 寄存器**，所以进入了**不同的地址空间**！

> **Note 3.2.2** 用户态和内核态的概念相信大家也不陌生，内核态即计组实验中所提到的特权态，用户态就是非特权态。mips 汇编中使用一些特权指令如`mtc0`、`mfc0`、`syscall`等都会陷入特权态（内核态）。

而我们这次实验，根据`./include/mmu.h`里面的布局来说，我们是 2G/2G 模式，用户态占用 2G，内核态占用 2G。接下来，也是最容易产生混淆的地方，进程从用户态提升到内核态的过程，操作系统中发生了什么？

是从当前进程切换成一个所谓的“内核进程”来管理一切了吗？不！还是一样的配方，还是一样的进程！改变的其实只是进程的权限！我们打一个比方，你所在的班级里没有班长，任何人都可以以合理的理由到老师那申请当临时班长。你是班里的成员吗？当然是的。某天你申请当临时班长，申请成功了，那么现在你还是班里的成员吗？当然还是的。那么你前后有什么区别呢？是权限的变化。可能你之前和其他成员是完全平等，互不干涉的。那么现在你可以根据花名册点名，你可以安排班里的成员做些事情，你可以开班长会议等等。那么我们能说你是班长吗？不能，因为你并不是永久的班长。但能说你拥有成为班长的资格吗？当然可以，这种**成为临时班长的资格**，我们可以粗略地认为它就是内核态的精髓所在。

而在操作系统中，每个完整的进程都拥有这种成为临时内核的资格，即所有的进程都可以发出申请变成内核态下运行的进程。内核态下进程可访问的资源更多，更加自由。在之后我们会提到一种“申请”的方式，就叫做“系统调用”。

那么你现在应该能够理解为什么我们要将内核才能使用的虚页表为每个进程都拷贝一份了，在 2G/2G 这种布局模式下，每个进程都会有 **2G 内核态**的虚拟地址，以便临时变身为“内核”行使大权。但是，在变身器使用之前，就算是奥特曼，一样也只能访问自己的那 2G 用户态的虚拟地址。

那么这种微妙的关系应该类似于下面这种：(图中灰色代表不可用，白色代表可用)

好的，现在结合注释你已经可以开始动手完成 env_setup_vm 函数了。

> **Exercise 3.4** 仔细阅读注释，填写 env_setup_vm 函数 ■

> **Thinking 3.3** 结合 include/mmu.h 中的地址空间布局，思考 env_setup_vm 函数：
>
> - 我们在初始化新进程的地址空间时为什么不把整个地址空间的 pgdir 都清零，而是复制内核的`boot_pgdir`作为一部分模板？(提示:mips 虚拟空间布局)
> - UTOP 和 ULIM 的含义分别是什么，在 UTOP 到 ULIM 的区域与其他用户区相比有什么最大的区别？

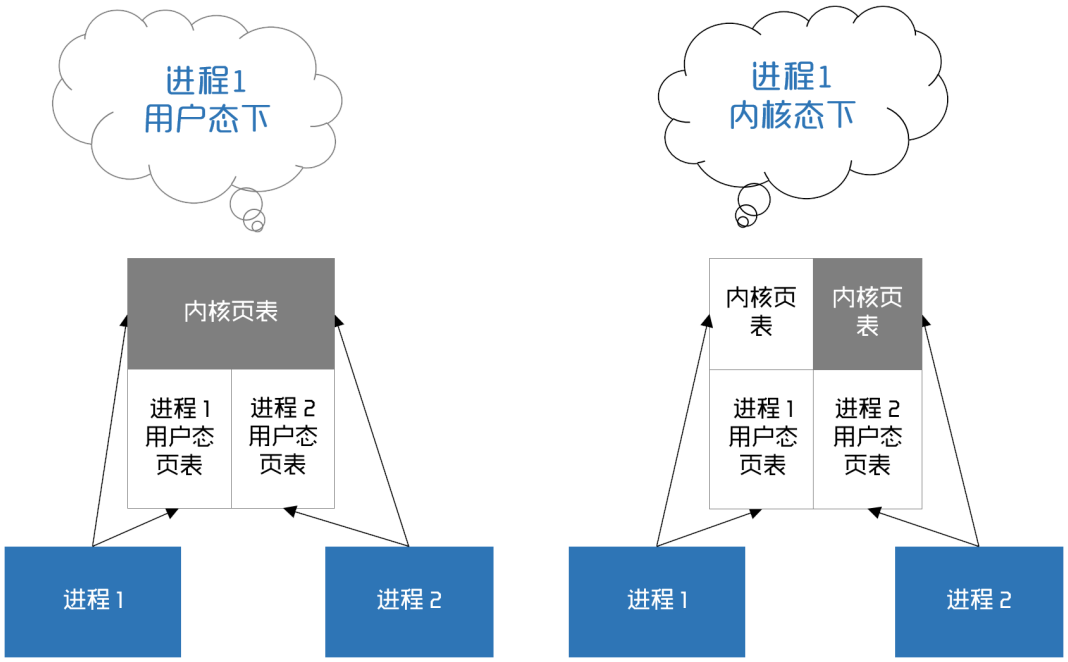

图 3.1: 进程页表与内核页表的关系

- 在 step4 中我们为什么要让`pgdir[PDX(UVPT)]=env_cr3`?(提示: 结合系统自映射机制)
- 谈谈自己对进程中物理地址和虚拟地址的理解

在上述的思考完成后，那么我们就可以直接在 **env_alloc** 第二步使用该函数了。现在来解决一下刚才的问题，第三点所说的合适的值是什么？我们要设定哪些变量的值呢？

我们要设定的变量其实在 **env_alloc** 函数的提示中已经说的很清楚了，至于其合适的值，相信仔细的你从函数的前面长长的注释里一定能获得足够的信息。当然我要讲的重点不在这里，重点在我们已经给出的这个设置`e->env_tf.cp0_status = 0x10001004;`

这个设置很重要，很重要，很重要，重要到我们必须直接在代码中给出。为什么说它重要，各位看官且听我娓娓道来。

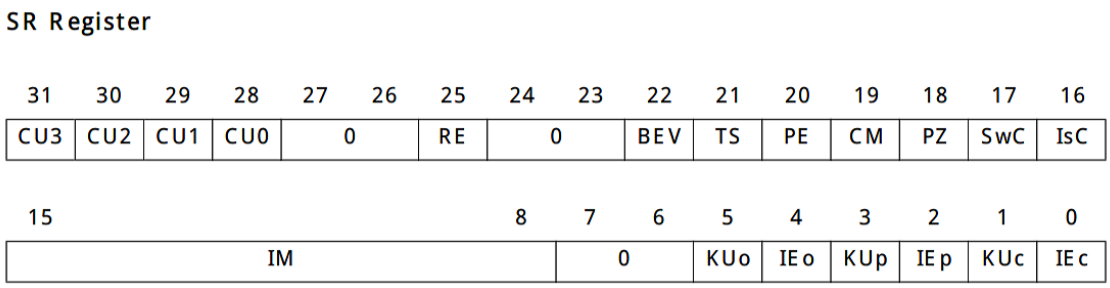

图 3.2: R3000 的 SR 寄存器示意图

图3.2是我们 MIPSR3000 里的 SR(status register) 寄存器示意图，就是我们在 env_tf 里的 cp0_ status。

第 28bit 设置为 1，表示处于用户模式下。

第 12bit 设置为 1，表示 4 号中断可以被响应。

这些都是小 case，下面讲的才是重点！

R3000 的 SR 寄存器的低六位是一个二重栈的结构。KUo 和 IEo 是一组，每当中断发生的时候，硬件自动会将 KUp 和 IEp 的数值拷贝到这里；KUp 和 IEp 是一组，当中断发生的时候，硬件会把 KUc 和 IEc 的数值拷贝到这里。

其中 KU 表示是否位于内核模式下，为 1 表示位于内核模式下；IE 表示中断是否开启，为 1 表示开启，否则不开启[2]。

而每当 rfe 指令调用的时候，就会进行上面操作的**逆操作**。我们现在先不管为何，但是已经知道，下面这一段代码在**运行第一个进程前是一定要执行的**，所以就一定会执行`rfe`这条指令。

```
1 lw   k0,TF_STATUS(k0)        # 恢复 CP0_STATUS 寄存器
2 mtc0 k0,CP0_STATUS
3 j    k1
4 rfe
```

现在你可能就懂了为何我们 status 后六位是设置为`000100b`了。当运行第一个进程前，运行上述代码到`rfe`的时候，就会将 KUp 和 IEp 拷贝回 KUc 和 IEc，令 status 为`000001b`，最后两位 KUc,IEc 为 **[0,1]**，表示开启了中断。之后第一个进程成功运行，这时操作系统也可以正常响应中断，Nice!

> **Note 3.2.3** 关于 MIPSR3000 版本 SR 寄存器功能的英文原文描述：The status register is one of the more important registers. The register has several fields. The current Kernel/User (KUc) flag states whether the CPU is in kernel mode. The current Interrupt Enable (IEc) flag states whether external interrupts are turned on. If cleared then external interrupts are ignored until the flag is set again. In an exception these flags are copied to previous Kernel/User and Interrupt Enable (KUp and IEp) and then cleared so the system moves to a kernel mode with external interrupts disabled. The Return From Exception instruction writes the previous flags to the current flags.

当然我们从注释也能看出，第四步除了需要设置`cp0status`以外，还需要设置栈指针。在 MIPS 中，栈寄存器是第 29 号寄存器，注意这里的栈是用户栈，不是内核栈。

> **Exercise 3.5** 根据上面的提示与代码注释，填写 **env_ alloc** 函数。 ■

3.2.4 加载二进制镜像

我们继续来完成我们的实验。下面这个函数还是蛮难填的呢，所以大家一定要跟紧我的步伐，我们慢慢来分析一下这个函数。

我们在进程创建第三点曾提到过，我们需要为**新进程的程序**分配空间来容纳程序代码。那么下面我需要有两个函数来一起完成这个任务

[2]我们的实验是不支持中断嵌套的，所以内核态时是不可以开启中断的。

Listing 11: 加载镜像映射

```
/* Overview:
 *   This is a call back function for kernel's elf loader.
 * Elf loader extracts each segment of the given binary image.
 * Then the loader calls this function to map each segment
 * at correct virtual address.
 *
 *   `bin_size` is the size of `bin`. `sgsize` is the
 * segment size in memory.
 *
 * Pre-Condition:
 *   bin can't be NULL.
 *   Hint: va may NOT aligned 4KB.
 *
 * Post-Condition:
 *   return 0 on success, otherwise < 0.
 */
static int load_icode_mapper(u_long va, u_int32_t sgsize,
         u_char *bin, u_int32_t bin_size, void *user_data)
{
        struct Env *env = (struct Env *)user_data;
        struct Page *p = NULL;
        u_long i;
        int r;

        /*Step 1: load all content of bin into memory. */
        for (i = 0; i < bin_size; i += BY2PG) {
        /* Hint: You should alloc a page and increase the reference count of it. */


        }
        /*Step 2: alloc pages to reach `sgsize` when `bin_size` < `sgsize`. */
        while (i < sgsize) {


        }
        return 0;
}
```

为了完成这个函数，我们接下来再补充一点关于 ELF 的知识。

前面在讲解内核加载的时候，我们曾简要说明过 ELF 的加载过程。这里，我们再做一些补充说明。要想正确加载一个 ELF 文件到内存，只需将 ELF 文件中所有需要加载的 segment 加载到对应的虚地址上即可。我们已经写好了用于解析 ELF 文件的代码 (lib/kernel_elfloader.c) 中的大部分内容，你可以直接调用相应函数获取 ELF 文件的各项信息，并完成加载过程。该函数的原型如下：

```
// binary 为整个待加载的 ELF 文件。size 为该 ELF 文件的大小。
// entry_point 是一个 u_long 变量的地址 (相当于引用)，解析出的入口地址会被存入到该位置
int load_elf(u_char *binary, int size, u_long *entry_point, void *user_data,
             int (*map)(u_long va, u_int32_t sgsize,
                        u_char *bin, u_int32_t bin_size, void *user_data))
```

我们来着重解释一下 load_elf() 函数的设计，以及最后两个参数的作用。为了让你

有机会完成加载可执行文件到内存的过程，load_elf() 函数只完成了解析 ELF 文件的部分，而把将 ELF 文件的各个 segment 加载到内存的工作留给了你。为了达到这一目标，load_elf() 的最后两个参数用于接受一个你的自定义函数以及你想传递给你的自定义函数的额外参数。每当 load_elf() 函数解析到一个需要加载的 segment，会将 ELF 文件里与加载有关的信息作为参数传递给你的自定义函数。你的自定义函数完成加载单个 segment 的过程。

为了进一步简化你的理解难度，我们已经为你定义好了这个“自定义函数”的框架。如代码11所示。load_elf() 函数会从 ELF 文件文件中解析出每个 segment 的四个信息：va(该段需要被加载到的虚地址)、sgsize(该段在内存中的大小)、bin(该段在 ELF 文件中的内容)、bin_size(该段在文件中的大小)，并将这些信息传给我们的“自定义函数”。

接下来，你只需要完成以下两个步骤：

第一步 加载该段在 ELF 文件中的所有内容到内存。

第二步 如果该段在文件中的内容的大小达不到该段在内存中所应有的大小，那么余下的部分用 0 来填充。

也许机灵的你发现了一个很无语的情况：我们并没有真正解释清楚 user_data 这个参数的作用。最后一个参数是一个函数指针，用于将我们的自定义函数传入进去。但这个诡异的 user_data 到底是做什么用的呢？这样设计又是为了什么呢？很不幸，这个问题我们决定留给你来思考。

Thinking 3.4 思考 user_data 这个参数的作用。没有这个参数可不可以？为什么？(如果你能说明哪些应用场景中可能会应用这种设计就更好了。可以举一个实际的库中的例子) ■

思考完这一点，我们就可以进入这一小节的练习部分了。

Exercise 3.6 通过上面补充的知识与注释，填充 **load_icode_mapper** 函数。 ■

Thinking 3.5 结合 load_icode_mapper 的参数以及二进制镜像的大小，考虑该函数可能会面临哪几种复制的情况？你是否都考虑到了？ ■

现在我们已经完成了补充部分最难的一个函数，那么下面我们完成这个函数后，就能真正实现把二进制镜像加载进内存的任务了。

Listing 12: 完整加载镜像

```
1 /* Overview:
2  *  Sets up the the initial stack and program binary for a user process.
3  *  This function loads the complete binary image by using elf loader,
4  *  into the environment's user memory. The entry point of the binary image
5  *  is given by the elf loader. And this function maps one page for the
6  *  program's initial stack at virtual address USTACKTOP - BY2PG.
```

```
 *
 * Hints:
 *  All mappings are read/write including those of the text segment.
 *  You may use these :
 *      page_alloc, page_insert, page2kva , e->env_pgdir and load_elf.
 */
static void
load_icode(struct Env *e, u_char *binary, u_int size)
{
    /* Hint:
     *  You must figure out which permissions you'll need
     *  for the different mappings you create.
     *  Remember that the binary image is an a.out format image,
     *  which contains both text and data.
     */
    struct Page *p = NULL;
    u_long entry_point;
    u_long r;
    u_long perm;

    /*Step 1: alloc a page. */

    /*Step 2: Use appropriate perm to set initial stack for new Env. */
    /*Hint: The user-stack should be writable? */

    /*Step 3:load the binary by using elf loader. */

    /***Your Question Here***/
    /*Step 4:Set CPU's PC register as appropriate value. */
    e->env_tf.pc = entry_point;
}
```

现在我们来根据注释一步一步完成这个函数。在第二步我们要用第一步申请的页面来初始化一个进程的栈，根据注释你应当可以轻松完成。这里我们只讲第三步的注释所代表的内容，其余你可以根据注释中的提示来完成。

第三步通过调用 load_elf() 函数来将 ELF 文件真正加载到内存中。这里仅做一点提醒：请将 load_icode_mapper() 这个函数作为参数传入到 load_elf() 中。其余的参数在前面已经解释过了，就不再赘述了。

Exercise 3.7 通过补充的 ELF 知识与注释，填充 **load_elf** 函数和 **load_icode** 函数。 ■

这里的`e->env_tf.pc`是什么呢？就是在我们计组中反复强调的甚为重要的`PC`。它指示着进程当前指令所处的位置。你应该知道，冯诺依曼体系结构的一大特点就是：程序预存储，计算机自动执行。我们要运行的进程的代码段预先被载入到了 **entry_ point** 为起点的内存中，当我们运行进程时，CPU 将自动从 pc 所指的位置开始执行二进制码。

Thinking 3.6 思考上面这一段话，并根据自己在 **lab2** 中的理解，回答：

- 我们这里出现的” 指令位置” 的概念，你认为该概念是针对虚拟空间，还是物理内存所定义的呢？

- 你觉得entry_point其值对于每个进程是否一样？该如何理解这种统一或不同？

■

思考完这一点后，下面我们来准备准备可以真正创建进程了。

3.2.5 创建进程

创建进程的过程很简单，就是实现对上述个别函数的封装，**分配一个新的 Env 结构体，设置进程控制块，并将二进制代码载入到对应地址空间**即可完成。好好思考上面的函数，我们需要用到哪些函数来做这几件事？

Exercise 3.8 根据提示，完成 **env_create** 函数与 **env_create_priority** 的填写。

■

当然提到创建进程，我们还需要提到一个封装好的宏命令

```
#define ENV_CREATE_PRIORITY(x, y) \
{ \
    extern u_char binary_##x##_start[];\
    extern u_int binary_##x##_size; \
    env_create_priority(binary_##x##_start, \
        (u_int)binary_##x##_size, y); \
}
```

```
#define ENV_CREATE(x) \
{ \
    extern u_char binary_##x##_start[];\
    extern u_int binary_##x##_size; \
    env_create(binary_##x##_start, \
        (u_int)binary_##x##_size); \
}
```

这个宏里的语法大家可能以前没有见过，这里解释一下##代表拼接，例如[3]

```
#define CONS(a,b) int(a##e##b)
int main()
{
    printf("%d\n", CONS(2,3));  // 2e3 输出:2000
    return 0;
}
```

好，那么现在我们就得手工在我们的init/init.c里面加两句话来初始化创建两个进程

```
ENV_CREATE_PRIORITY(user_A, 2);
ENV_CREATE_PRIORITY(user_B, 1);
```

这两句话加在哪里呢？那就需要你翻代码来寻找啦～

[3]这个例子是转载的，出处为http://www.cnblogs.com/hnrainll/archive/2012/08/15/2640558.html，想深入了解的同学可以参考这篇博客

Exercise 3.9 根据注释与理解，将上述两条进程创建命令加入 **init/init.c** 中。 ■

做完这些，是不是迫不及待地想要跑个进程看看能否成功？等我们完成下面这个函数，就可以开始第一部分的自我测试了！

3.2.6 进程运行与切换

Listing 13: 进程的运行

```
1  extern void env_pop_tf(struct Trapframe *tf, int id);
2  extern void lcontext(u_int contxt);
3
4  /* Overview:
5   *  Restores the register values in the Trapframe with the
6   *  env_pop_tf, and context switch from curenv to env e.
7   *
8   * Post-Condition:
9   *  Set 'e' as the curenv running environment.
10  *
11  * Hints:
12  *  You may use these functions:
13  *      env_pop_tf and lcontext.
14  */
15 void
16 env_run(struct Env *e)
17 {
18     /*Step 1: save register state of curenv. */
19     /* Hint: if there is a environment running,you should do
20     *  context switch.You can imitate env_destroy() 's behaviors.*/
21
22
23     /*Step 2: Set 'curenv' to the new environment. */
24
25
26     /*Step 3: Use lcontext() to switch to its address space. */
27
28
29     /*Step 4: Use env_pop_tf() to restore the environment's
30      * environment registers and drop into user mode in the
31      * the environment.
32      */
33     /* Hint: You should use GET_ENV_ASID there.Think why? */
34
35 }
```

刚刚说到的 load_icode 是为数不多的坑函数之一，env_run 也是，而且其实按程度来讲可能更甚一筹。

env_run，是进程运行使用的基本函数，它包括两部分：

- 一部分是保存当前进程上下文 (**如果当前没有运行的进程就跳过这一步**)
- 另一部分就是恢复要启动的进程的上下文，然后运行该进程。

Note 3.2.4 进程上下文说来就是一个环境，相对于进程而言，就是进程执行时的环境。具体来说就是各个变量和数据，包括所有的寄存器变量、内存信息等。

其实我们这里运行一个新进程往往意味着是进程切换，而不是单纯的进程运行。进程切换，人如其名，就是当前进程停下工作，让出 CPU 处理器来运行另外的进程。那么要理解进程切换，我们就要知道进程切换时系统需要做些什么。Alt+Tab 可以吗？当然不可以。实际上进程切换的时候，为了保证下一次进入这个进程的时候我们不会再“从头来过”，而是有记忆地从离开的地方继续往后走，我们要保存一些信息，那么，需要保存什么信息呢？理所当然地想想，你可能会想到下面两种需要保存的状态：

进程本身的状态

进程周围的环境状态

那么我们先解决一个问题，进程本身的状态需要记录吗？进程本身的状态无非就是进程块里面那几个东西，包括

env_id,env_parent_id,env_pgdir,env_cr3...

或许你会有所疑问，**env_tf** 不算是进程本身的状态吗？按笔者的理解来说，是不算的。env_tf 保存的是进程上下文，相当于我们的第二点，进程周围的环境状态。

我们仔细思索一下，就能发现，进程本身的状态在进程切换的时候是不会变化的。(我们不会去毁灭一个进程块，进程块跟我们又没仇。) 会变的也是需要我们保存的实际上是进程的环境信息。

这样或许你就能明白 run 代码中的第一句注释的含义了：/*Step 1: save register state of curenv. */

那么你可能会想，进程运行到某个时刻，它的上下文——所谓的 CPU 的寄存器在哪呢？我们又该如何保存？在 lab3 中，我们在本实验里的寄存器状态保存的地方是 TIMESTACK 区域。

```
struct Trapframe  *old;

old = (struct Trapframe *)(TIMESTACK - sizeof(struct Trapframe));
```

这个 old 就是当前进程的上下文所存放的区域。那么第一步注释还说到，让我们参考`env_destroy`，其实就是让我们把 old 区域的东西**拷贝到当前进程的 env_ tf 中**，以达到保存进程上下文的效果。那么我们还有一点很关键，就是当我们返回到该进程执行的时候，应该从哪条指令开始执行？即当前进程上下文中的 pc 应该设为什么值？这将留给聪明的你去思考。

Thinking 3.7 思考一下，要保存的进程上下文中的`env_tf.pc`的值应该设置为多少？为什么要这样设置？ ■

思考完上面的，我们沿着注释再一路向下，后面好像没有什么很难的地方了。根据提示也完全能够做出来。但是我们还有一点坑没填，我们忽略了 **env_pop_tf** 函数。

env_pop_tf 是定义在 lib/env_asm.S 中的一个汇编函数。这个函数也可以用来解释: 为什么启动第一个进程前一定会执行 `rfe`汇编指令。但是我们本次思考的重点不在这里，重点在于 TIMESTACK。请仔细地看看这个函数，你或许能看出什么关于 TIMESTACK 的端倪。TIMESTACK 问题可能将是你在本实验中需要思考时间最久的问题，希望你能和小伙伴积极交流，努力寻找实验代码来支撑你的看法与观点，鼓励提出新奇的想法!

Thinking 3.8 思考 TIMESTACK 的含义，并找出相关语句与证明来回答以下关于 TIMESTACK 的问题:

- 请给出一个你认为合适的 TIMESTACK 的定义
- 请为你的定义在实验中找出合适的代码段作为证据 (请对代码段进行分析)
- 思考 TIMESTACK 和第 18 行的 KERNEL_SP 的含义有何不同

■

Exercise 3.10 根据补充说明，填充完成 **env_run** 函数。 ■

至此，我们第一部分的工作已经完成了! 第二部分的代码量很少，但是不可或缺! 休息一下，我们继续奋斗!

3.3 中断与异常

之前我们在学习计组的时候已经学习了异常和中断的概念，所以这里我们不再将概念作为主要介绍内容。

Note 3.3.1 我们实验里认为凡是引起控制流突变的都叫做异常，而中断仅仅是异常的一种，并且是仅有的一种异步异常。

我们可以通过一个简单的图来认识一下异常的产生与返回（见图3.3)。

3.3.1 异常的分发

每当发生异常的时候，一般来说，处理器会进入一个用于分发异常的程序，这个程序的作用就是检测发生了哪种异常，并调用相应的异常处理程序。一般来说，这个程序会被要求放在固定的某个物理地址上（根据处理器的区别有所不同），以保证处理器能在检测到异常时正确地跳转到那里。这个分发程序可以认为是操作系统的一部分。

代码3.3.1就是异常分发代码，我们先将下面代码填充到我们的`start.S`的开头，然后我们来分析一下。

```
1 .section .text.exc_vec3
2 NESTED(except_vec3, 0, sp)
3       .set noat
```

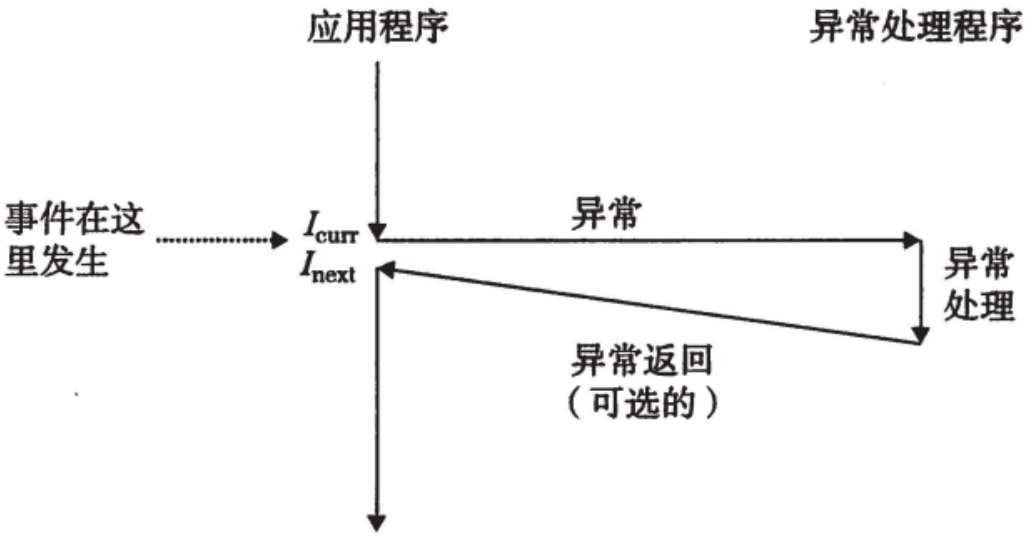

图 3.3: 异常处理图示

```
4       .set noreorder
5    1:
6       mfc0 k1,CP0_CAUSE
7       la   k0,exception_handlers
8       andi k1,0x7c
9       addu k0,k1
10      lw   k0,(k0)
11      NOP
12      jr   k0
13      nop
14   END(except_vec3)
15      .set at
```

Exercise 3.11 将异常分发代码填入 **boot/start.S** 合适的部分。 ■

这段异常分发代码的作用流程简述如下：

1. 取得异常码，这是区别不同异常的重要标志。
2. 以得到的异常码作为索引去 exception_handlers 数组中找到对应的中断处理函数，后文中会有涉及。
3. 跳转到对应的中断处理函数中，从而响应了异常，并将异常交给了对应的异常处理函数去处理

图3.4是 MIPS3000 中 Cause Register 寄存器。其中保存着 CPU 中哪一些中断或者异常已经发生。bit2 bit6 保存着异常码，也就是根据异常码来识别具体哪一个异常发生了。bit8 bit15 保存着哪一些中断发生了。其他的一些位含义在此不涉及，可参看 MIPS 开发手册。

这个.text.exec_vec3 段将通过链接器放到特定的位置，在 R3000 中要求是放到 0x80000080 地址处，这个地址处存放的是异常处理程序的入口地址。一旦 CPU 发生异常，就会自

Cause Register

31	30	29 28	27 16	15 8	7	6 2	1 0
BD	0	CE	0	IP	0	ExcCode	0

图 3.4: CR 寄存器

动跳转到 0x80000080 地址处，开始执行，下面我们将.text.exec_vec3 放到该位置，一旦异常发生，就会引起该段代码的执行，而该段代码就是一个分发处理异常的过程。

所以我们要在我们的 lds 中增加这么一段，从而可以让 R3000 出现异常时自动跳转到异常分发代码处。

```
. = 0x80000080;
.except_vec3 : {
    *(.text.exc_vec3)
}
```

Exercise 3.12 将 lds 代码补全使得异常后可以跳到异常分发代码。 ■

3.3.2 异常向量组

我们刚才提到了异常的分发，要寻找到 exception_handlers 数组中的中断处理函数，而 exception_handlers 就是所谓的异常向量组。

下面我们跟随`trap_init(lib/traps.c)`函数来看一下，异常向量组里存放的是些什么？

```
extern void handle_int();
extern void handle_reserved();
extern void handle_tlb();
extern void handle_sys();
extern void handle_mod();
unsigned long exception_handlers[32];

void trap_init()
{
    int i;

    for (i = 0; i < 32; i++) {
        set_except_vector(i, handle_reserved);
    }

    set_except_vector(0, handle_int);
    set_except_vector(1, handle_mod);
    set_except_vector(2, handle_tlb);
    set_except_vector(3, handle_tlb);
    set_except_vector(8, handle_sys);
}
void *set_except_vector(int n, void *addr)
{
    unsigned long handler = (unsigned long)addr;
```

```
25      unsigned long old_handler = exception_handlers[n];
26      exception_handlers[n] = handler;
27      return (void *)old_handler;
28  }
```

实际上呢，这个函数实现了对全局变量 exception_handlers[32] 数组初始化的工作，其实就是把相应的处理函数的地址填到对应数组项中。主要初始化

0 号异常 的处理函数为 handle_int，

1 号异常 的处理函数为 handle_mod，

2 号异常 的处理函数为 handle_tlb，

3 号异常 的处理函数为 handle_tlb，

8 号异常 的处理函数为 handle_sys，

一旦初始化结束，有异常产生，那么其对应的处理函数就会得到执行。而在我们的实验中，我们主要是要使用 0 号异常，即中断异常的处理函数。因为我们接下来要做的，就是要产生时钟中断。

3.3.3 时钟中断

希望你还没有忘记在计组实验中所接触到的**定时器**这个概念。或许你当时对定时器的作用会有疑惑，为了防止你继续迷糊不清，我们下面来简单介绍一下时钟中断的概念。

时钟中断和操作系统的时间片轮转算法是紧密相关的。时间片轮转调度是一种很公平的算法。每个进程被分配一个时间段，称作它的时间片，即该进程允许运行的时间。如果在时间片结束时进程还在运行，则该进程将挂起，切换到另一个进程运行。那么 CPU 是如何知晓一个进程的时间片结束的呢？就是通过定时器产生的时钟中断。当时钟中断产生时，当前运行的进程被挂起，我们需要在调度队列中选取一个合适的进程运行。如何“选取”，就要涉及到进程的调度了。

要产生时钟中断，我们不仅要了解中断的产生与处理。我们还要了解 gxemul 是如何模拟时钟中断的。初始化时钟主要是在 kclock_init 函数中，该函数主要调用 set_timer 函数，完成如下操作：

- 首先向 0xb5000100 位置写入 1，其中 0xb5000000 是模拟器 (gxemul) 映射实时钟的位置。偏移量为 0x100 表示来设置实时钟中断的频率，1 则表示 1 秒钟中断 1 次，如果写入 0，表示关闭实时钟。实时钟对于 R3000 来说绑定到了 4 号中断上，故这段代码其实主要用来触发了 4 号中断。注意这里的中断号和异常号是不一样的概念，我们实验的异常包括中断。

- 一旦实时钟中断产生，就会触发 MIPS 中断，从而 MIPS 将 PC 指向`0x80000080`，从而跳转到`.text.exc_vec3`代码段执行。对于实时钟引起的中断，通过 text.exc_vec3 代码段的分发，最终会调用 handle_ int 函数来处理实时钟中断。

- 在 handle_ int 判断CP0_CAUSE寄存器是不是对应的 4 号中断位引发的中断，如果是，则执行中断服务函数 timer_ irq。
- 在 timer_ irq 里直接跳转到 sched_ yield 中执行。而这个函数就是我们将要补充的调度函数。

以上就是我们时钟中断的产生与中断处理流程，我们在这里要完成下面的任务以顺利产生时钟中断。

Exercise 3.13 通过上面的描述，补充 **kclock_init** 函数。 ■

Thinking 3.9 阅读 kclock_asm.S 文件并说出每行汇编代码的作用 ■

3.3.4 进程调度

通过上面的描述，我们知道了，其实在时钟中断产生时，最终是调用了 sched_ yield 函数来进行进程的调度，这个函数在lib/sched.c中所定义。这个函数就是我们本次最后要写的调度函数。调度的算法很简单，就是时间片轮转的算法。这里优先级就有用了，我们在这里将优先级设置为时间片大小: 1 表示 1 个时间片长度, 2 表示 2 个时间片长度，以此类推。不过寻找就绪状态进程不是简单遍历进程链表, 而是用两个链表存储所有就绪状态进程。每当一个进程状态变为 ENV_RUNNABLE , 我们要将其插入第一个就绪状态进程链表。调用 sched_yield 函数时, 先判断当前时间片是否用完。如果用完, 将其插入另一个就绪状态进程链表。之后判断当前就绪状态进程链表是否为空。如果为空, 切换到另一个就绪状态进程链表。

Exercise 3.14 根据注释，完成 **sched_yield** 函数的补充，并根据调度方法对 env.c 中的部分函数进行修改，使得进程能够被正确调度。 ■

Thinking 3.10 阅读相关代码，思考操作系统是怎么根据时钟周期切换进程的。 ■

至此，我们的实验三就算是圆满完成了。

3.4 实验正确结果

如果你按流程做下来并且做的结果正确的话，你运行之后应该会出现这样的结果

```
1  init.c: mips_init() is called
2
3  Physical memory: 65536K available, base = 65536K, extended = 0K
4
5  to memory 80401000 for struct page directory.
6
7  to memory 80431000 for struct Pages.
8
9  mips_vm_init:boot_pgdir is 80400000
10
```

```
pmap.c:  mips vm init success

panic at init.c:27: ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

当然不会这么整齐，且没有换行，只是交替输出 2 和 1 而已～不过 1 的个数几乎是 2 的 2 倍。

3.5 实验思考

- 思考-init 的逆序插入
- 思考-mkenvid 的作用
- 思考-地址空间初始化
- 思考-user_data 的作用
- 思考-load-icode 的不同情况
- 思考-位置的含义
- 思考-进程上下文的 PC 值
- 思考-TIMESTACK 的含义
- 思考-时钟的设置
- 思考-进程的调度