Lab3 实验报告

零、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 Lab3 中将运行一个用户模式的进程。
本实验需要使用数据结构进程控制块 `Env` 来跟踪用户进程，并建立一个简单的用户进程，加载一个程序镜像到指定的内存空间，然后让它运行起来。
同时，实验实现的 MIPS 内核具有处理异常的能力。

一、思考题

Thinking 3.1

> 请结合 MOS 中的页目录自映射应用解释代码中 `e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_V` 的含义。

`UVPT`：用户页表的起始处的内核虚拟地址。（`UVPT~ULIM` 是 `User VPT`）
`PDX(UVPT)`：这个虚拟地址的页目录号。
`e->env_pgdir`：进程 `e` 的页目录的内核虚拟地址。
`PADDR(e->env_pgdir)`：进程 `e` 的页目录的物理地址。
`PADDR(e->env_pgdir) | PTE_V`：页目录的物理基地址，加上权限位。
页表基地址：`UVPT`
页目录基地址：`UVPT | (UVPT >> 10)`
映射到页目录的页目录项的基地址：`UVPT | (UVPT >> 10) | (UVPT >> 20)`
该页表项对应于第几个页目录项：`(UVPT >> 20) >> 2 = UVPT >> 22 = PDX(UVPT)`

Thinking 3.2

> `elf_load_seg` 以函数指针的形式，接受外部自定义的回调函数 `map_page`。 请你找到与之相关的 `data` 这一参数在此处的来源，并思考它的作用。没有这个参数可不可以？为什么？

`data` 是传入的进程控制块指针，在 `load_icode_mapper` 和 `load_icode` 函数中被调用。作用是增加虚拟地址到物理地址映射时提供 `env_pgdir` 和 `env_asid`。
如果没有 `data`，`load_icode_mapper` 就不能知道当前进程空间的页目录基地址和 `asid`，所以必须要有。

Thinking 3.3

> 结合 `elf_load_seg` 的参数和实现，考虑该函数需要处理哪些页面加载的情 况。

首先判断 `va` 是否页对齐，不对齐则将多余地址标记为 `offset`；
然后依次将段内页映射到物理空间；

最后如果其在内存的大小大于在文件中的大小，则需要将多余的空间用 0 填充。

Thinking 3.4

> 思考上面这一段话，并根据自己在 Lab2 中的理解，回答：
>
> - 你认为这里的 `env_tf.cp0_epc` 存储的是物理地址还是虚拟地址?

存储的是**虚拟地址**。
因为 `epc` 存储的是发生错误时 CPU 所处的指令地址，对于 CPU，可见的都是虚拟地址。

Thinking 3.5

> 试找出 0、1、2、3 号异常处理函数的具体实现位置。8 号异常（系统调用） 涉及的 `do_syscall()` 函数将在 Lab4 中实现。

0 号异常 **`handle_int`**： `kern/genex.S` 中处理时钟中断
1 号异常 **`handle_mod`**： `kern/tlbex.c` 中处理
2、3 号异常 **`handle_tlb`**： `kern/tlb_asm.S` 中执行 `do_tlb_refill` 来重填

Thinking 3.6

> 阅读 `entry.S` 、 `genex.S` 和 `env_asm.S` 这几个文件，并尝试说出时钟中断在哪些时候开启，在哪些时候关闭。

`entry.S` 中使用 `handle_int` 开启，过程中进入 `schedule` 函数，调用 `env_run` ，再调用 `env_pop_tf` （具体实现在 `env_asm.S` ）其中调用 `RESET_KCLOCK` 。

Thinking 3.7

> 阅读相关代码，思考操作系统是怎么根据时钟中断切换进程的。

时钟中断后，调用 `schedule` 切换线程。

二、难点分析

项目结构

```
.
├── include
│   ├── env.h // lab3：进程相关函数
│   ├── kclock.h // lab3：RESET_KCLOCK 宏
│   ├── trap.h // lab3：Trapframe 结构体
│   └── types.h
├── include.mk
├── init
├── kern
│   ├── entry.S // lab3：异常分发代码
│   ├── env.c // lab3：env 相关函数
```

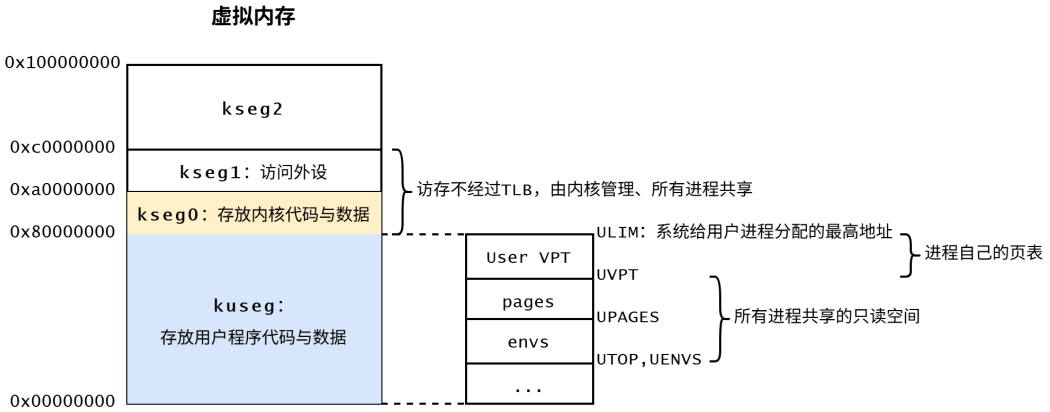
```
│   ├── genex.S // lab3：0 号异常
│   ├── tlb_asm.S // lab3：2、3 号异常
│   ├── tlbex.c // lab3：1 号异常
│   └── traps.c
├── kernel.lds
├── lib
│   ├── elfloader.c // lab3：elf_load_seg 函数
│   └── string.o
├── Makefile
├── target
│   └── mos
├── tools
└── user
```

`env_id` 和 `env_asid`

前者是每个进程唯一标识符，后者用于 TLB 切换（`ASID` 标识虚拟地址空间，同时并发执行的多个进程具有不同的 `ASID` 以方便 TLB 标识其虚拟地址空间）。MOS 使用位图法管理 256 个可用的 `ASID`，如果耗尽时仍要创建进程，内核会发生崩溃（`panic`）。

> TLB 实质构建了一个 `<VPN, ASID>-><PFN, C, D, V, G>` 的映射。其中 `VPN` 是虚拟页号，`PFN` 是物理页框号。每一个进程都有自己虚拟空间下的一套独立 TLB 缓存，每次切换页表无需清空所有 TLB 表项。

进程与虚拟内存



`Status` 寄存器

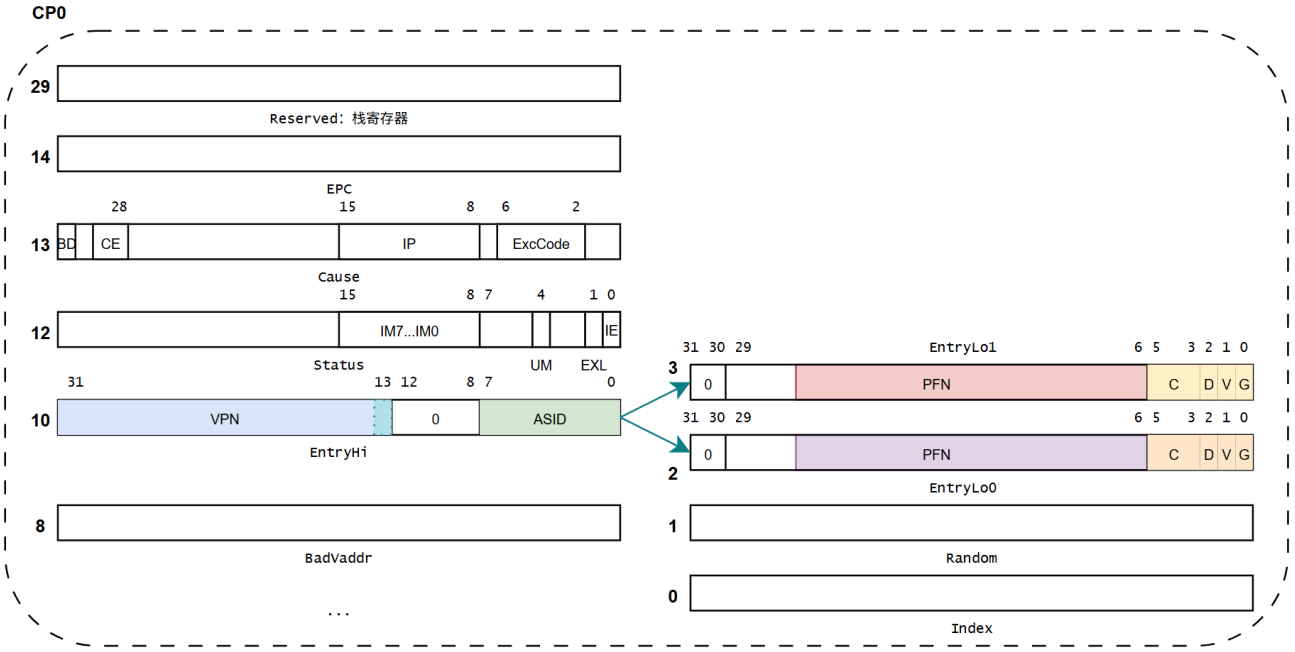
即为 `e->env_tf.cp0_status` 。`env_tf` 即 `Trapframe` 结构体：位于 `include/trap.h` ，用于保存当前进程的上下文信息。

```
 8 struct Trapframe {
 9         /* Saved main processor registers. */
10         unsigned long regs[32];
11
12         /* Saved special registers. */
```

```
13          unsigned long cp0_status;
14          unsigned long hi;
15          unsigned long lo;
16          unsigned long cp0_badvaddr;
17          unsigned long cp0_cause;
18          unsigned long cp0_epc;
19  };
```

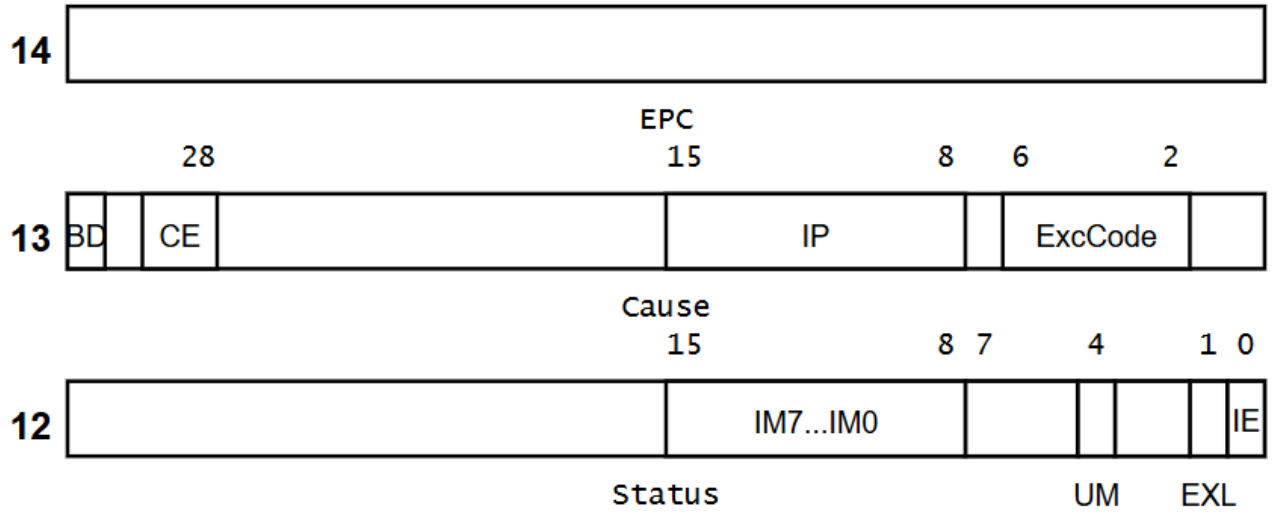
这个结构体存储了 CP0 的 32 个寄存器，其中就包含 status、cause、epc 等。
在 Status 中，包含几个特殊位：

- `IE` 表示中断是否开启，1 表示开启。
- `IM7` 表示 7 号中断（时钟中断），如果将 `IE` 和 `IM7` 设置为 1，表示中断使能，7 号中断可以被响应。
- `EXL` 和 `UM` 控制用户态和内核态切换。当前者为 0 后者为 1 时处于用户模式，其余都是内核模式。异常发生时，`EXL` 设置为 1，执行 `eret` 时，`EXL` 设置为 0。



中断与异常

CP0 中的三个寄存器 Status、Cause、EPC 于此紧密相关。
`Status` **寄存器**：包含中断屏蔽位，在上面介绍过，不再赘述。
`Cause` **寄存器**：保存着 CPU 中哪一些中断或者异常已经发生。`IP` 位保存哪些中断已经发生，当 Status 中相同位（对于 `IM`）允许中断（置 1）时，Cause 寄存器这一活动就会导致中断。`ExcCode` 记录发生了什么异常。
`EPC` **寄存器**：保存异常结束后程序恢复执行的位置。



当异常发生时：

- 设置 `EPC` 指向异常返回的地址。
- 设置 `EXL` 强制进入内核态禁止中断。
- 设置 `Cause` 记录异常发生的原因。
- CPU 开始从异常入口位置取指，此后由软件处理。

进程调度

采用时间片轮转算法，时间片长度量化为 `N * TIMER_INTERVAL` ，其中 `N` 是由进程的优先级决定的，而一个静态变量 `count` 存储了剩余时间片长度（也就是 `N`）

三、实验体会

相比于上一个 Lab，这个还是比较好理解的，并且更加完善了上个 Lab 整理的内存和 CP0 世界观（）。但 Lab3 的 MIPS 汇编代码部分比较难以理解。

四、原创说明

1. BUAA-OS-lab3 | YannaのBlog
2. os-lab3实验报告 | hugo