MOS操作系统实验

lab3 进程与中断
课下任务讲解

内容提要

- 背景知识
- 实验概述
- 实验内容
  - 创建一个进程并成功运行
  - 实现时钟中断，通过时钟中断内核可以再次获得执行权
  - 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
- 测试结果

背景知识

- 进程的引入：
- “程序”与“计算”不是一一对应的关系：一个程序可能对应多个“计算”
- 多道程序＋资源的限制：执行-暂停-执行
  - 直接制约：逻辑上相互依赖
  - 间接制约：等待资源
- 使用“程序”不能揭示多道程序、分时系统引发的动态特性，因此引入“进程”（Process）

背景知识

- 一个进程应该包括
  - 程序的代码；
  - 程序的数据；
  - PC中的值，用来指示下一条将运行的指令；
  - 一组通用的寄存器的当前值，堆、栈；
  - 一组系统资源（如打开的文件）

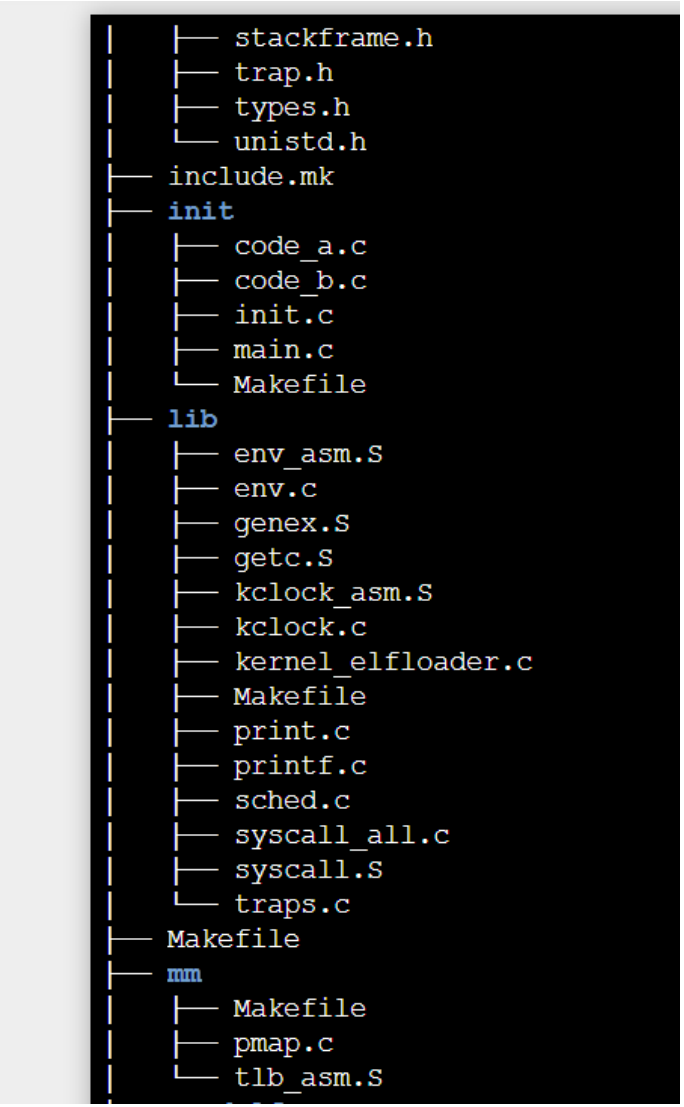
实验概述

- 1. 创建一个进程并成功运行
  - 进程控制块数据结构（Env）的初始化与管理
  - 进程控制块创建（生成envid、进程地址空间建立）
  - 加载进程代码和数据
- 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

代码目录结构

- 比较，在lab2基础上，增加了什么？

- lib目录下多个文件
- init/code_a.c、code_b.c



实验内容——进程控制块

- **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。

```
struct Env {
        struct Trapframe env_tf;        // Saved registers
        LIST_ENTRY(Env) env_link;       // Free list
        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;
        LIST_ENTRY(Env) env_sched_link;
        u_int env_pri;
        // Lab 4 IPC
        u_int env_ipc_value;            // data value sent to us
        u_int env_ipc_from;             // envid of the sender
        u_int env_ipc_recving;          // env is blocked receiving
        u_int env_ipc_dstva;            // va at which to map received page
        u_int env_ipc_perm;             // perm of page mapping received

        // Lab 4 fault handling
        u_int env_pgfault_handler;      // page fault state
        u_int env_xstacktop;            // top of exception stack

        // Lab 6 scheduler counts
        u_int env_runs;                 // number of times been env_run'ed
        u_int env_nop;                  // align to avoid mul instruction
};
```

我们**MOS**操作系统中的**PCB**的结构

进程控制块数组envs

- 实验中，存放进程控制块数组envs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。

- 类比lab2内存管理中的pages数组的初始化工作，完成envs数组和env_free_list的初始化工作。

- Exercise 3.1

修改pmap.c/mips_vm_init 函数完成envs数组的初始化工作

本次不需要填写，但请阅读

- Exercise 3.2

填写env_init 函数完成env_free_list的初始化工作

进程的标识——envid

- 与envid相关的两个函数：
- 1.u_int mkenvid(struct Env *e)
- 功能:生成一个envid

- 2.int envid2env(u_int envid, struct Env **penv, int checkperm)
- 功能：通过envid获取对应的进程控制块

- 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。

- 提示：阅读mkenvid函数体会envid的生成过程。

设置进程控制块

进程创建的流程如下：

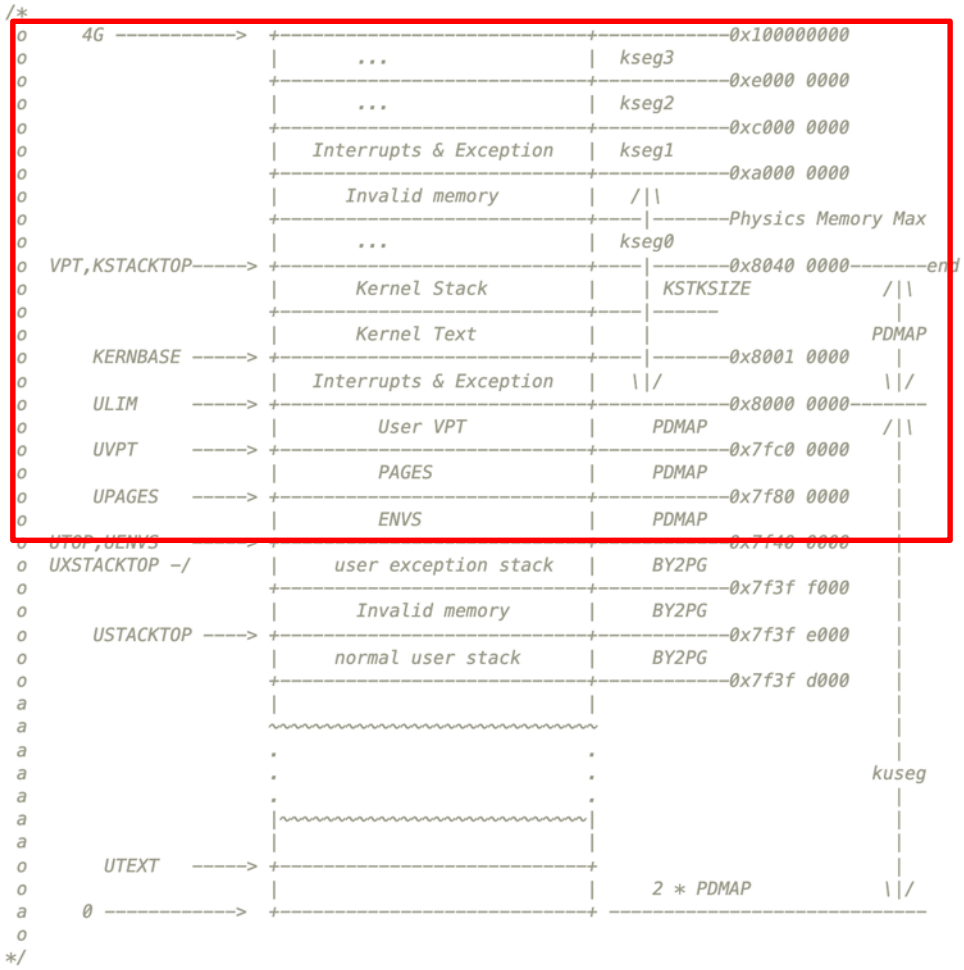
- 第一步申请一个空闲的PCB，从env_free_list 中索取一个空闲PCB 块，这时候的PCB 就像张白纸一样。
- 第二步“纯手工打造”打造一个进程。在这种创建方式下，由于没有模板进程,所以进程拥有的所有信息都是手工设置的。而进程的信息又都存放于进程控制块中，所以我们需要手工初始化进程控制块。
- 第三步进程光有PCB 的信息还没法跑起来，每个进程都有独立的地址空间。所以，我们要为新进程分配资源，为新进程的程序和数据以及用户栈分配必要的内存空间。
- 第四步此时PCB 已经被涂涂画画了很多东西，不再是一张白纸，把它从空闲链表里除名,就可以投入使用了

核心函数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*/


}
```

初始化新进程的地址空间

- Exercise 3.4
- env.c 中的env_setup_vm 函数的功能是初始化新进程地址空间中的内核部分。



- 具体任务:

1.为新进程申请页目录，设置e->env_pgdir 和e->env_cr3 两个域

2.初始化新进程地址空间中的内核部分（红框）

3.结合系统自映射机制，设置指向页目录自身的页目录项（页表映射在UVPT起始的虚拟地址）

我们MOS操作系统的地址空间结构

进程的SR(status register) 寄存器

SR Register

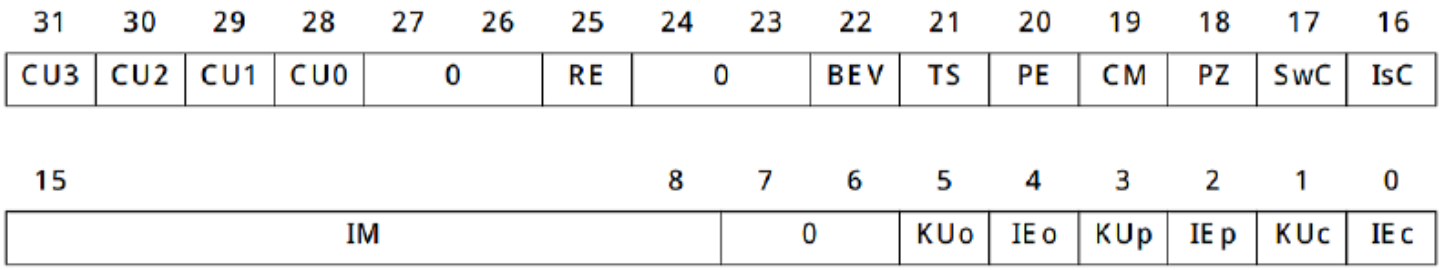


图 3.2: R3000 的 SR 寄存器示意图

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。

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。
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
```

我们status 后六位是设置为001100b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000011b，最后两位KUc,IEc 为[1,1]，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。

完成env_alloc

- Exercise 3.5
- 从空闲进程控制块链表中获得一个空闲的进程控制块（先不删除）。
- 调用env_setup_vm 函数初始化新进程的地址空间。
- 初始化进程控制块中的一些域
- 从空闲进程控制块链表中删除该进程控制块。

```
/*** exercise 3.5 ***/
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 completed just now) 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 the sp register and cp0_status of env_tf field, located at this new Env. */
    e->env_tf.cp0_status = 0x10001004;


    /*Step 5: Remove the new Env from env_free_list. */


}
```

加载二进制镜像

- 进程是程序的一次运行，所以我们要为进程准备即将执行的代码和数据。

- 右侧两个练习需要完成的函数的作用是加载ELF（文件）中的所有内容到内存。

- Exercise 3.6

填充load_icode_mapper函数。

- Exercise 3.7

填充load_elf 函数和load_icode 函数。

load_icode_mapper 函数

- 这是一个回调函数，会被load_elf函数调用，它的作用是映射二进制镜像的一段长度为bin_size的内容到指定虚拟地址va。
- 如果该段在文件中的内容的大小达不到ELF中该段在内存中所应有的大小(sgsize)，那么余下的部分用0来填充。

```
/*** exercise 3.6 ***/
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;
    u_long offset = va - ROUNDDOWN(va, BY2PG);

    /*Step 1: load all content of bin into memory. */
    for (i = 0; i < bin_size; i += BY2PG) {
        /* Hint: You should alloc a new page. */
    }
    /*Step 2: alloc pages to reach `sgsize` when `bin_size` < `sgsize`.
    * hint: variable `i` has the value of `bin_size` now! */
    while (i < sgsize) {


    }
    return 0;
}
```

load_elf 函数

- load_elf() 函数完成了解析ELF 文件的任务，并且通过调用load_icode_mapper函数将ELF 文件的各个segment 加载到内存。
- 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load_icode_mapper函数。
- load_icode_mapper函数完成加载单个segment 的过程。

```
/*** exercise 3.7 ***/
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))
{
        Elf32_Ehdr *ehdr = (Elf32_Ehdr *)binary;
        Elf32_Phdr *phdr = NULL;
         /* As a loader, we just care about segment,
           * so we just parse program headers.
           */
        u_char *ptr_ph_table = NULL;
        Elf32_Half ph_entry_count;
        Elf32_Half ph_entry_size;
        int r;

         // check whether `binary` is a ELF file.
        if (size < 4 || !is_elf_format(binary)) {
                return -1;
        }

        ptr_ph_table = binary + ehdr->e_phoff;
        ph_entry_count = ehdr->e_phnum;
        ph_entry_size = ehdr->e_phentsize;

        while (ph_entry_count--) {
                phdr = (Elf32_Phdr *)ptr_ph_table;

                if (phdr->p_type == PT_LOAD) {
         /* Your task here!  */
        /* Real map all section at correct virtual address.Return < 0 if error. */
        /* Hint: Call the callback function you have achieved before. */




                }

                ptr_ph_table += ph_entry_size;
        }

        *entry_point = ehdr->e_entry;
        return 0;
}
```

load_icode 函数

- 申请一页内存。
- 用第一步申请的页面来初始化一个进程的栈。
- 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。
- 设置进程的pc寄存器。

```
/*** exercise 3.7 ***/
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: Should the user-stack be writable? */


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


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

创建进程

- env_create_priority函数就是对之前部分函数的封装。
- 主要有以下几步：
  - 分配一个新的Env 结构体。
  - 设置进程控制块。
  - 为新进程设置优先级。
  - 并将二进制代码载入到对应地址空间。
- env_create函数则是对env_create_priority函数的封装。

- Exercise 3.8

完成env_create 函数与env_create_priority 的填写

```
/*** exercise 3.8 ***/
void
env_create_priority(u_char *binary, int size, int priority)
{
        struct Env *e;
    /*Step 1: Use env_alloc to alloc a new env. */

    /*Step 2: assign priority to the new env. */

    /*Step 3: Use load_icode() to load the named elf binary,
      and insert it into env_sched_list using LIST_INSERT_HEAD. */

}
/* Overview:
 * Allocates a new env with default priority value.
 *
 * Hints:
 *  this function calls the env_create_priority function.
 */
/*** exercise 3.8 ***/
void
env_create(u_char *binary, int size)
{
     /*Step 1: Use env_create_priority to alloc a new env with priority 1 */

}
```

创建进程

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

- 真正创建进程，我们还需要一个封装好的宏命令，如下图所示：

```
1  #define ENV_CREATE_PRIORITY(x, y) \
2  { \
3      extern u_char binary_##x##_start[];\
4      extern u_int binary_##x##_size; \
5      env_create_priority(binary_##x##_start, \
6          (u_int)binary_##x##_size, y); \
7  }
```

```
1  #define ENV_CREATE(x) \
2  { \
3      extern u_char binary_##x##_start[];\
4      extern u_int binary_##x##_size; \
5      env_create(binary_##x##_start, \
6          (u_int)binary_##x##_size); \
7  }
```

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]

```
1  #define CONS(a,b) int(a##e##b)
2  int main()
3  {
4      printf("%d\n", CONS(2,3));  // 2e3 输出:2000
5      return 0;
6  }
```

- 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

进程运行与切换

- env_run是进程运行使用的基本函数，它包括两部分：

1.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)

2.恢复要启动的进程的上下文，然后运行该进程。

- Exercise 3.10 根据补充说明，填充完成env_run函数

```
env_run(struct Env *e)
{
    /*Step 1: save register state of curenv. */
    /* Hint: if there is a environment running,you should do
    *  context switch.You can imitate env_destroy() 's behaviors.*/


    /*Step 2: Set 'curenv' to the new environment. */


    /*Step 3: Use lcontext() to switch to its address space. */


    /*Step 4: Use env_pop_tf() to restore the environment's
     * environment registers and drop into user mode in the
     * the environment.
     */
    /* Hint: You should use GET_ENV_ASID there.Think why? */

}
```

进程运行与切换

- 实际上进程切换的时候，为了保证下一次恢复这个进程执行的时候能从之前离开的地方继续往后执行，我们要保存一些“现场”信息，也就是所谓的进程上下文
- 进程上下文就是进程执行时的环境。具体来说就是所有的寄存器变量、内存页表（基地址）等信息。
- 进程本身的状态（进程块里面的内容），包括：
  - env_id
  - env_parent_id
  - env_pgdir
  - env_cr3
  - ...

进程运行与切换

- 进程周围的环境状态：
- CPU 的寄存器
- 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。
- 保存进程上下文就是把old 区域的东西拷贝到当前进程的env_ tf 中。

异常处理

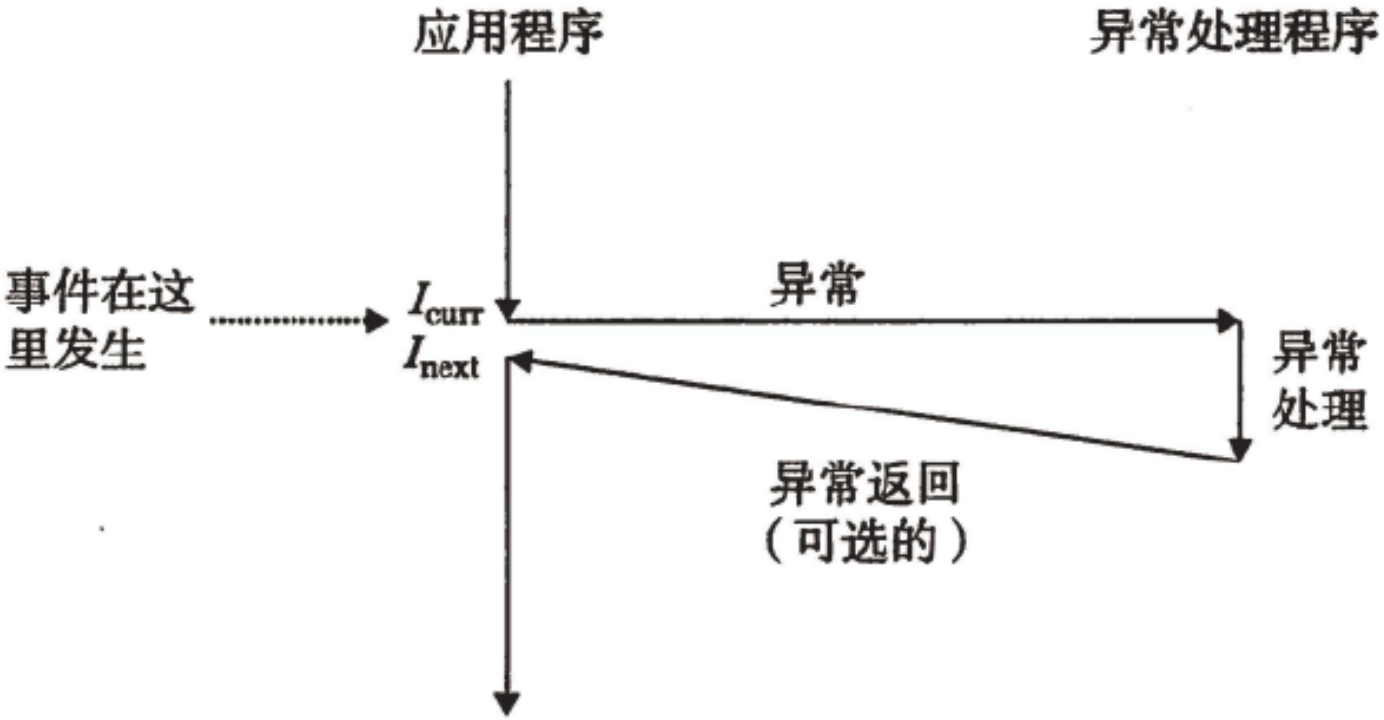


图 3.3: 异常处理图示

异常分发

- Exercise 3.11 将异常分发代码填入boot/start.S 合适的部分
- Exercise 3.12 将lds 代码补全使得异常后可以跳到异常分发代码。

- 在计组的学习过程中，我们知道，当异常触发时，CPU 会将 PC 直到一个特定的地址，表示进入异常处理程序。
- 针对不同的异常，CPU需要启用不同的异常处理程序，因此它需要一个异常分发的入口函数，用于甄别不同的异常类型，并将PC跳转到对应异常的处理程序入口地址。

异常分发

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

第 1 行是指定这段代码的地址。而这个地址的值设定在 tools/scse0_3.lds 中。

第 3 行是条伪指令，它的作用是在 assemble 的时候禁用 at 寄存器（有些扩展指令 assemble 的时候会用到 at 寄存器）。因为进入异常处理程序后，需要保存现场，如果这里利用 at 寄存器来实现一些扩展指令，会破坏现场。

第 4 行仍然是条伪指令，其作用是取消乱序执行。

异常分发

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

- 第 6 行这一个操作在计组中也多次遇见，mfc0 会把 CP0_CAUSE 寄存器的值存到 k1 寄存器中。因此现在 k1 寄存器存储着异常发生的原因。
- 第 7 行是将异常处理程序入口地址数组的首地址赋值给 k0 寄存器。我们知道，这段异常分发代码是为了跳转到特定的异常处理程序中去。而exception_handlers，便是存储了不同异常处理程序的入口地址。

异常分发

- 如何加载异常分发代码到特定位置?
- Linkscript

```
1  . = 0x80000080;
2  .except_vec3 : {
3      *(.text.exc_vec3)
4  }
```

异常分发

- exception_handlers 数组定义在 lib/traps.c 中。里面的具体的值，我们可以参考 trap_init() 函数。

```
4
5   extern void handle_int();
6   extern void handle_reserved();
7   extern void handle_tlb();
8   extern void handle_sys();
9   extern void handle_mod();
10  unsigned long exception_handlers[32];
11  void trap_init(){
12          int i;
13          for(i=0;i<32;i++)
14          set_except_vector(i, handle_reserved);
15          set_except_vector(0, handle_int);
16          set_except_vector(1, handle_mod);
17          set_except_vector(2, handle_tlb);
18          set_except_vector(3, handle_tlb);
19          set_except_vector(8, handle_sys);
20  }
21  void *set_except_vector(int n, void * addr){
22          unsigned long handler=(unsigned long)addr;
23          unsigned long old_handler=exception_handlers[n];
24          exception_handlers[n]=handler;
25          return (void *)old_handler;
26  }
```

异常分发

- 在《See MIPS Run LINUX》中我们可以查阅到：

ExcCode	助记符	描述
0	Int	中断
1	Mod	存储操作时，该页在 TLB 中被标记为只读。
2	TLBL	没有 TLB 转换（读写分别），也就是 TLB 中没有和程序地址匹配的有效入口。
3	TLBS	当根本没有匹配项（连无效的匹配项都没有）并且 CPU 尚未处于异常模式—— SR(EXL) 置位——即 TLB 失效时，为高效平滑处理这种常见事件而采用的特殊异常入口点。
4	AdEL	（取数、取指或者存数时）地址错误：这要么是在用户态试图存取 kuseg 以外的空间，或者是试图从未对齐的地址读取一个双字、字或者半字。
5	AdES	
6	IBE	总线错误（取指或者读取数据）：外部硬件发出了某种出错信号；具体该怎么做与系统有关。因存储而导致的总线错，只能作为一个为了获取要写入的高速缓存行而执行的高速缓存读操作的结果间接出现。
7	DBE	
8	Syscall	执行了一条 syscall 指令。

- 其中与本次实验相关的便是 0 号异常 —— 中断

异常处理

- 回到之前的那份代码，第 8 行是取出 ExcCode 放到 k1 寄存器中。使用 0x7c 的原因如下：

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

Cause Register

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

异常分发

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

- 第 9 行是获取到对应的异常处理程序。有些同学可能会疑惑，这里得到的 k1 的值不是后面有两个多余的零嘛，数组下标会不会有问题。

- 产生这个问题的原因是对高级语言的数组下标和mips 中的 lw 指令产生了混淆。每个异常处理程序的入口都以一个字的形式存储在数组里，因此两个相邻的元素之间地址的差就是 4。因此这样的处理是毫无问题的。第 10 行，便是将这个特定的异常处理程序存入到了 k0 寄存器中。最后在第 12 行跳到该异常处理程序中。

中断处理

- Exercise 3.13 通过上面的描述，补充kclock_init 函数。
- 接下来我们详细看下对于中断的处理：
- 在计组中我们学到，中断是由定时器产生的信号，它的作用是提供了时间片的机制。首先我们看下中断信号是怎么产生的。 kclock_init 是对定时器初始化 的函数，它主要就调用了 set_timer 这个函数，因此我们移步至该函数来具体分析它的代码。

```
17          .text
18  LEAF(set_timer)
19
20          li t0, 0x01
21          sb t0, 0xb5000100
22          sw      sp, KERNEL_SP
23  setup_c0_status STATUS_CU0|0x1001 0
24          jr ra
25
26          nop
27  END(set_timer)
```

中断处理

- 前两步是向 0xb5000100 写入数值 1。其中0xb5000000 是MOS操作系统内核访问硬件时钟（实际上是gxemul仿真器模拟的时钟）时用到的虚拟地址（硬件地址是多少？）。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。
- 另外需要注意的是实时钟对于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。中断号对应着计组中 HWInt 的位置。

gxemul仿真的外设：

http://gavare.se/gxemul/gxemul-stable/doc/experiments.html#testmachines

中断处理

- 接下来我们看下触发中断程序后，代码会如何执行。在 genex.S 中，我们找到 handle_int 函数。

```
46  .set noreorder
47  .align  5
48  NESTED(handle_int, TF_SIZE, sp)
49  .set    noat
50
51  //1: j 1b
52  nop
53
54  SAVE_ALL
55  CLI
56  .set    at
57  mfc0    t0, CP0_CAUSE
58  mfc0    t2, CP0_STATUS
59  and     t0, t2
60
61  andi    t1, t0, STATUSF_IP4
62  bnez    t1, timer_irq
63  nop
64  END(handle_int)
```

```
64  END(handle_int)
```

中断处理

- 先看前两句 mfc0，它分别将 CPU_CAUSE 和 CPU_STATUS 存入了 t0 和 t2 两个寄存器中。接着将 t0 and t2 的值存入 t0 中。两个 CP0 的各个域意义如下：

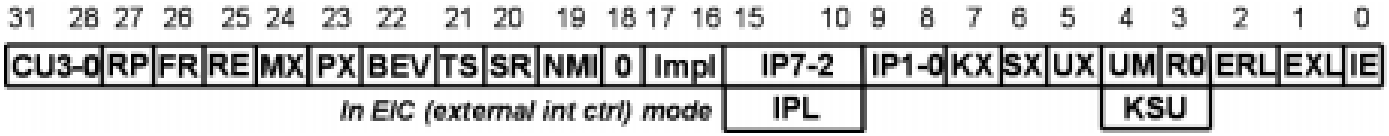


图 3.1: SR（状态）寄存器的各个域

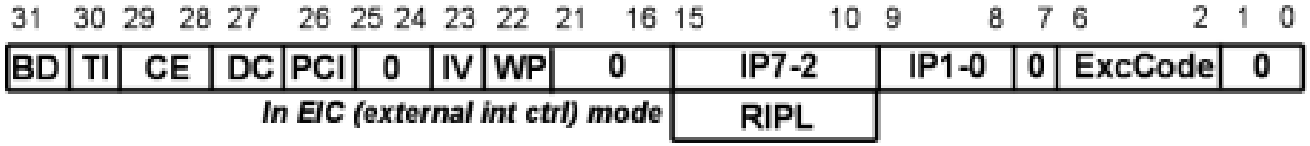


图 3.2: Cause 寄存器的各个域

- 这样操作后，我们就可以得到具体的中断号了。最后通过判断中断号。我们确定是否是实时钟触发的中断，如果是，跳转到 time_irq 中。

中断处理

- time_irq 的代码如下：

```
68  timer_irq:
69
70  1:      j       sched_yield
71          nop
72          /*li t1, 0xff
73          lw    t0, delay
74          addu  t0, 1
75          sw      t0, delay
76          beq     t0,t1,1f
77          nop*/
78          j       ret_from_exception
79          nop
```

- 其中最核心的一步便是跳转到 sched_yield 。这段代码也是时间片轮转机制的基础。当时钟产生中断时，这段汇编代码会调用调度程序。

带优先级的双队列调度

- Exercise 3.14 根据注释，完成sched_yield 函数的补充

- 调度程序：按优先级调度，使用了双队列调度来减少调度中的不公平。难点主要在于第二步，它所带来的影响不仅是在 lab3 中，还包括后面 lab4 内容。

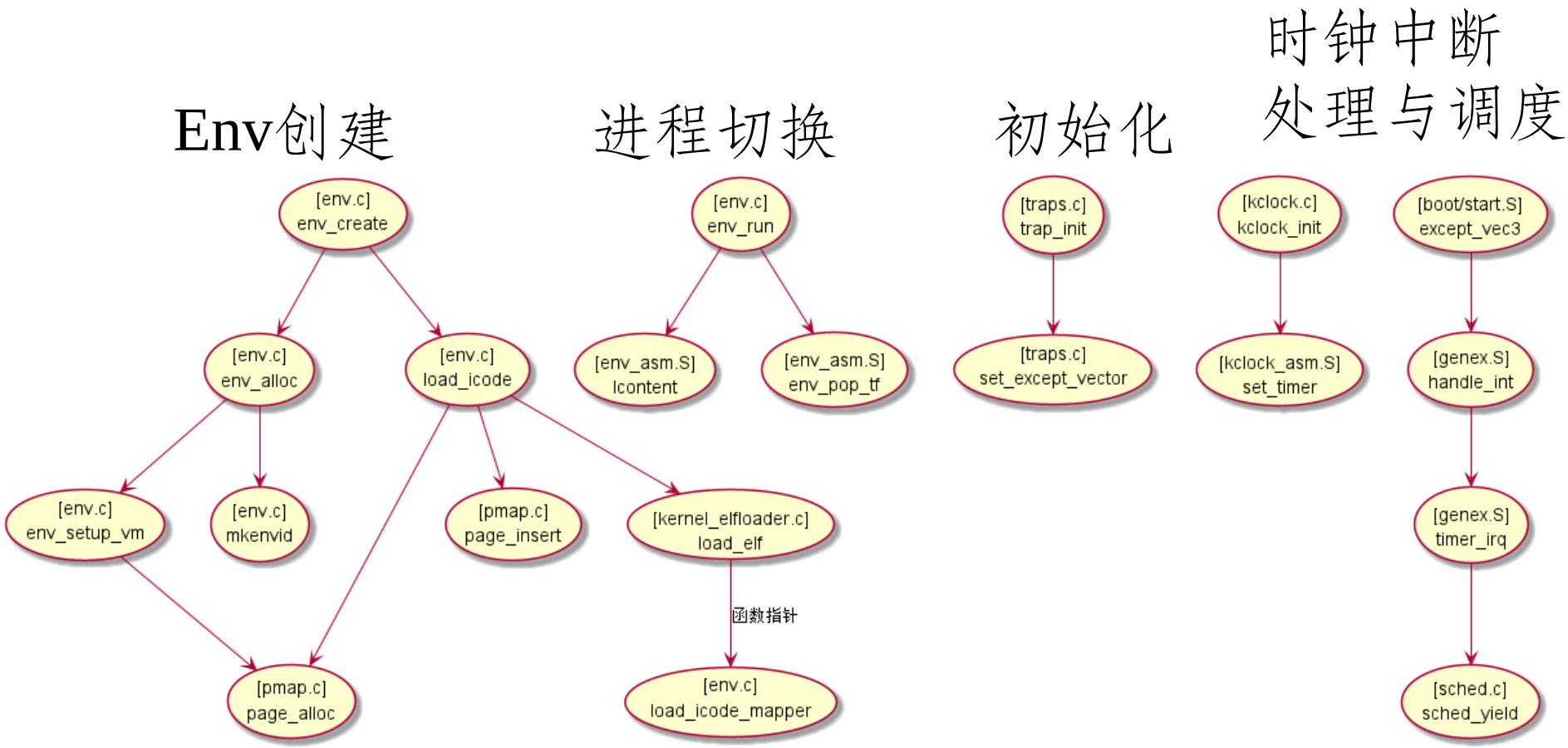
带优先级的调度

- 我们先讨论优先级的机制。在 Struct Env 中有这样一个域: env_pri ，用来表示优先级的大小。在本实验中，优先级表示着所占用的连续时间片的长短。一种思路是修改定时器的参数，另一种思路是让该进程连续执行 env_pri 次。我们用静态变量 times 来存储当前进程剩余执行次数，并且用静态变量 e 来存储当前进程。每次调用 sched_yield 时，我们先判断 times 是否为 0，如果不为 0 ，我们将 times 减 1 ，并 env_run(e) 来接着执行该进程。如果为 0，我们需要利用调度程序，切换到下一个进程。

带优先级的双队列调度

- 两个队列仅存储 runnable 的进程，减少了遍历 env_list 的时间。需要在设置 env 的 status 为 runnable 的时候将该 Env 插入到第一个队列中。
- 相应的，我们需要在 destroy 的时候，将其从队列中移除。
- 而调度的时候，如果当前进程的时间片用完了，我们需要利用 LIST_INSERT_TAIL 宏将其插入到另一个队列的队尾。
- 静态变量 point表示当前正在遍历的队列。因此另一个队列，可以通过 1-point 来表示（或者1^point）。
- 如果当前队列的程序遍历完了，我们利用 pos^=1 移步至另一个队列。
- 至此，本实验的进程调度就全部完成了。

函数调用关系


Env创建
[env.c]
env_create
[env.c]
env_alloc
[env.c]
load_icode
[env.c]
env_setup_vm
[env.c]
mkenvid
[pmap.c]
page_insert
[kernel_elfloader.c]
load_elf
[pmap.c]
page_alloc
函数指针
[env.c]
load_icode_mapper
进程切换
[env.c]
env_run
[env_asm.S]
lcontent
[env_asm.S]
env_pop_tf
初始化
[traps.c]
trap_init
[traps.c]
set_except_vector
时钟中断
处理与调度
[kclock.c]
kclock_init
[kclock_asm.S]
set_timer
[boot/start.S]
except_vec3
[genex.S]
handle_int
[genex.S]
timer_irq
[sched.c]
sched_yield

测试结果

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

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

Physical memory: 65536K available, base = 65536K, extended = 0K

to memory 80401000 for struct page directory.

to memory 80431000 for struct Pages.

mips_vm_init:boot_pgdir is 80400000

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
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
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
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。

谢谢！

祝实验顺利！