

OS Assignment 3: Documentation

December 25, 2013

Group No.12

Group members: Ma Xuan(1110580), Wang Shaoyao(1110539)
Wu Zhengwei(1110548), Yuan Zhaozheng(1110556), Zhang Chao(1110557)

Introduction

本次实验主要内容为在一个多用户任务同时活跃的进程环境上实现一个抢占式多任务调度机制，由三部分组成：

第一个部分中，首先要实现轮转调度算法 Round-Robin Scheduling 实现基本的用户环境管理系统调用(用于创建和销毁用户环境，分配和映射相应的内存)。

第二个部分中，需要完成一个类似于 Unix 中的 `fork()` 函数，它允许一个用户进程去创建一个新的进程——子进程。我们可以使用克隆的方式实现，但是必须让它可以独立于自己的父进程正常的运行。

第三个部分，需要增加对进程通信 (IPC) 功能的实现。允许不同的进程之间进行通信，并且显示的同步的执行任务。还需要增加一个支持硬件时钟中断和抢占式的任务调度。

I. User-level Environment Creation and Cooperative Multitasking

Round-Robin Scheduling

在实验的第一个部分，需要增加一部分新的 JOS 内核系统调用让用户进程去创建新的进程。需完成一个协作式轮转调度算法，当用户的进程退出的时候，允许内核去切换到其它的进程继续执行。

Exercise 1

Description

在 `sched_yield()` 中完成循环调度策略，不要忘记修改 `syscall()` 函数用来调度 `sys_yield()` 修改 `kern/init.c` 来创建两个或者多个进程，让他们都开始调度 `user/yield.c` 下面的程序。你将会看到我们的进程环境不断的前后循环的切换，知道五次循环结束。结果示例如下：

```
...
Hello, I am environment 00001001.
Hello, I am environment 00001002.
Back in environment 00001001, iteration 0.
5 Back in environment 00001002, iteration 0.
Back in environment 00001001, iteration 1.
Back in environment 00001002, iteration 1.
.....
```

在 `yield` 程序退出之后，空闲进程(idle environment)此时应该进入运行，并且唤醒内核的debugger。如果这些没有出现，请仔细检查一下你的程序在做后续操作。

Solution

`sched_yield()` 代码，注意 `idle environment` 的运行条件。

Listing 1: `kern/sched.c: sched_yield()`

```
// Choose a user environment to run and run it.
void
sched_yield(void)
{
5   // Implement simple round-robin scheduling.
   // Search through 'envs' for a runnable environment,
   // in circular fashion starting after the previously

   // running env,
10  // and switch to the first such environment found.
   // It's OK to choose the previously running env
   // if no other env is runnable.
   // But never choose envs[0], the idle environment,
   // unless NOTHING else is runnable.

15  // LAB 4: Your code here.
   int i;

   struct Env *e = (curenv == NULL ||
20                    curenv >= envs + NENV-1) ?
                    envs+1 : curenv+1;

   for (i = 1; i < NENV; i++) {
       if (e->env_status == ENV_RUNNABLE)
25         env_run(e);
       e = (e == envs+NENV-1) ? envs+1 : e+1;
   }

   // Run the special idle environment
30  // when nothing else is runnable.

   if (envs[0].env_status == ENV_RUNNABLE)
       env_run(&envs[0]);
   else {
35       cprintf("Destroyed all environments \
               - nothing more to do!\n");
       while (1)
           monitor(NULL);
   }
40 }
```

修改 `kern/syscall.c`, 添加相关的分发机制, 然后在 `kern/init.c` 中系统启动时创建 `user_idle`, 之后创建 `user_yield`, 其功能是调用 5 次 `sys_yield()`, 并在且切换时打印相关信息:

Listing 2: `kern/init.c: i386 init()`

```
// Should always have an idle process as first one.
ENV_CREATE(user_idle);
```

```
ENV_CREATE(user_yield);  
ENV_CREATE(user_yield);  
5 ENV_CREATE(user_yield);
```

Exercise 2

Description

在 kern/syscall.c 中实现如上描述的系统调用。你需要用到在 kern/pmap.c 和 kern/env.cc 下的很多函数,特别是函数 `envid2env()`。从现在开始,不论你如何调用函数,请将 `envid2env()` 的 `checkperm` 参数设置为 1, 请确定你检查了所有返回 `EINVAL` 的无效的系统调用。利用 JOS 内核中的 `user/dumbfork` 来确认你的工作是正确的完成了。

Solution

`sys_exofork()`: 建立一个新的几乎空白的用户环境: 没有任何东西被映射到用户部分的地址空间, 而且非runnable。

新的用户环境将会拥有和父进程相同的在 `sys_exofork` 调用下的寄存器状态。在父进程中, `sys_exofork` 将会返回新创建的环境的 `envid_t` (或者是一个负的内存分配失败标识)。在子进程中, 将会返回 0。

以 `pid = 0` 表示当前的用户环境, 该定义在 `envid2env()` 中实现。

Listing 3: kern/syscall.c: `sys_exofork()`

```
static envid_t  
sys_exofork(void)  
{  
    envid_t ret;  
    5 struct Env *e;  
    ret = env_alloc(&e, curenv->env_id);  
  
    if (ret < 0)  
        10 return ret;  
  
    e->env_status = ENV_NOT_RUNNABLE;  
  
    // copy trap frame  
    e->env_tf = curenv->env_tf;  
    15  
    // make the child env return value zero  
    e->env_tf.tf_regs.reg_eax = 0;  
  
    20 return e->env_id;  
}
```

`env_set_status()`: 设置特定的环境为 `ENV_RUNNABLE` 或者是 `ENV_NOT_RUNNABLE`。

Listing 4: kern/syscall.c: `env_set_status()`

```

static int
sys_env_set_status(envid_t envid, int status)
{
    struct Env *e;
5   if (status != ENV_RUNNABLE &&
        status != ENV_NOT_RUNNABLE)
        return -E_INVAL;

    if (envid2env(envid, &e, 1) < 0)
10    return -E_BAD_ENV;
    e->env_status = status;
    return 0;
}

```

sys_page_alloc(): 分配一个页的物理内存并且将它映射到给定的虚拟地址空间中去。

Listing 5: kern/syscall.c: sys_page_alloc()

```

static int
sys_page_alloc(envid_t envid, void *va, int perm)
{
    struct Env *e;
5   struct Page *page;
    if ((uint32_t)va >= UTOP ||
        PGOFF(va) != 0 ||
        (perm & 5) != 5 ||
        (perm & (~PTE_USER)) != 0 )
10    return -E_INVAL;
    // PTE_USER = PTE_U | PTE_P | PTE_W | PTE_AVAIL

    if (envid2env(envid, &e, 1) < 0)
        return -E_BAD_ENV;
15    if (page_alloc(&page) < 0)
        return -E_NO_MEM;
    if (page_insert(e->env_pgdir, page, va, perm) < 0) {
        page_free(page);
        return -E_NO_MEM;
20    }
    //fill the new page with 0
    memset(page2kva(page), 0, PGSIZE);
    return 0;
}

```

sys_page_map(): 复制一块页内存映射 (不是页的内容) 并且将它映射到一块给定的虚拟地址环境空间中。

Listing 6: kern/syscall.c: sys_page_map()

```

static int
sys_page_map(envid_t srcenvid, void *srcva,
              envid_t dstenvid, void *dstva,
              int perm)
5 {

```

```

    if (srcva >= (void *)UTOP ||
        ROUNDUP(srcva, PGSIZE) != srcva ||
        dstva >= (void *)UTOP ||
        ROUNDUP(dstva, PGSIZE) != dstva)
10     return -E_INVAL;

    if ((perm & PTE_U) == 0 || (perm & PTE_P) == 0)
        return -E_INVAL;

15     if ((perm & ~PTE_USER) > 0)
        return -E_INVAL;

    struct Env *srcenv;
    if (envid2env(srcenvid, &srcenv, 1) < 0)
20     return -E_BAD_ENV;

    struct Env *dstenv;
    if (envid2env(dstenvid, &dstenv, 1) < 0)
        return -E_BAD_ENV;

25     pte_t *pte;
    struct Page *p = page_lookup(srcenv->env_pgdir,
                                srcva, &pte);

    if (p == NULL ||
30     ((perm & PTE_W) > 0 &&
     (*pte & PTE_W) == 0))
        return -E_INVAL;

    if (page_insert(dstenv->env_pgdir, p, dstva, perm) < 0)
35     return -E_NO_MEM;
    return 0;
}

```

sys_page_unmap(): 取消一个到一块虚拟地址环境空间中的页映射。

Listing 7: kern/syscall.c: sys_page_unmap()

```

static int
sys_page_unmap(envid_t envid, void *va)
{
    struct Env *e;
5     if (envid2env(envid, &e, 1) < 0)
        return -E_BAD_ENV;
    if ((uint32_t)va >= UTOP || PGOFF(va) != 0)
        return -E_INVAL;
    page_remove(e->env_pgdir, va);
10     return 0;
}

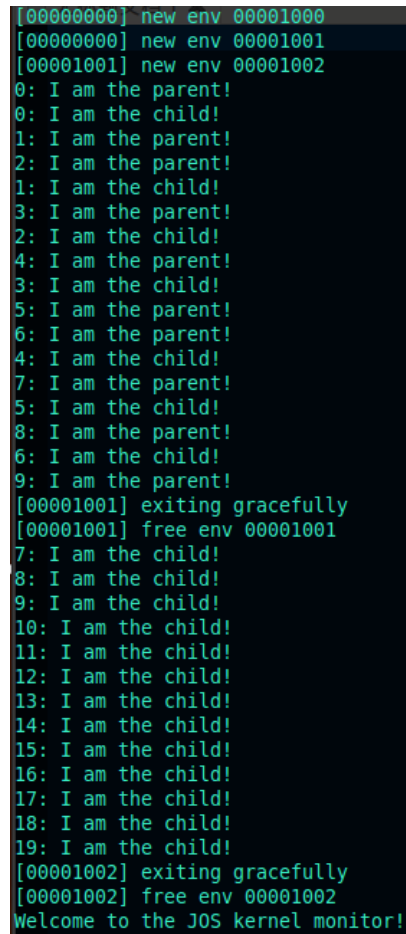
```

测试程序 user/dumbfork: 父进程创建一个子进程 (用户环境), 然后每次打印一条信息后交出控制权, 两个进程间利用 sys_yield 进行前后切换。在 10 次迭代之后, 父进程退出, 子进程将迭代 20 次。

Listing 8: user/dumbfork.c

```
envid_t dumbfork(void);  
void  
umain(void)  
{  
5   envid_t who;  
   int i;  
  
   // fork a child process  
   who = dumbfork();  
10  
   // print a message and yield to the other a few times  
   for (i = 0; i < (who ? 10 : 20); i++) {  
       cprintf("%d: I am the %s!\n", i,  
               who ? "parent" : "child");  
15       sys_yield();  
   }  
}
```

输入 `make run-dumbfork` 运行，结果如下：



```
[00000000] new env 00001000  
[00000000] new env 00001001  
[00001001] new env 00001002  
0: I am the parent!  
0: I am the child!  
1: I am the parent!  
2: I am the parent!  
1: I am the child!  
3: I am the parent!  
2: I am the child!  
4: I am the parent!  
3: I am the child!  
5: I am the parent!  
6: I am the parent!  
4: I am the child!  
7: I am the parent!  
5: I am the child!  
8: I am the parent!  
6: I am the child!  
9: I am the parent!  
[00001001] exiting gracefully  
[00001001] free env 00001001  
7: I am the child!  
8: I am the child!  
9: I am the child!  
10: I am the child!  
11: I am the child!  
12: I am the child!  
13: I am the child!  
14: I am the child!  
15: I am the child!  
16: I am the child!  
17: I am the child!  
18: I am the child!  
19: I am the child!  
[00001002] exiting gracefully  
[00001002] free env 00001002  
Welcome to the JOS kernel monitor!
```

II. Copy-on-Write Fork

Setting the Page Fault Handler

Exercise 3

Description

请实现 `sys_env_set_pgfault_upcall` 系统调用。当我们查询目标环境的环境 ID 时请确认你的权限检查, 因为我们的这样的系统调用时比较“危险”的。

Solution

为了处理用户态的缺页中断, 首先需要在JOS内核中注册缺页中断处理函数, 就是实现这个 `set_pgfault_upcall` 函数, 将用户自定义的中断处理程序注册到进程结构中。

```
static int
sys_env_set_pgfault_upcall(envid_t envid, void *func)
{
    struct Env *e;
5   if (envid2env(envid, &e, 1) < 0)
        return -E_BAD_ENV;
    e->env_pgfault_upcall = func;
    return 0;
}
```

Invoking the User Page Fault Handler

Exercise 4

Description

完成在 `kern/trap.c` 中的代码, 建立一个用户模式下的错误处理机制。确认采取一个合适的方法来写入你的异常栈中。

Solution

完成 `trap.c` 里的 `page_fault_handler`, 在 lab3 里面我们已经完成了一部分, 即判断缺页中断是否发生在内核中, 如果是, 那么直接 `panic`, 陷入内核; 现在我们要处理如果 `page_fault` 发生在用户态该如何处理。

```
void
page_fault_handler(struct Trapframe *tf)
{
5   uint32_t fault_va;
```



```

// Read processor's CR2 register to find
// the faulting address
fault_va = rcr2();

10 // Handle kernel-mode page faults.

// LAB 3: Your code here.

15 if ((tf->tf_cs & 0x3) == 0)
    panic("page fault handler: in kernel mode\n");

// LAB 4: Your code here.

20 if (curenv->env_pgfault_upcall != NULL) {
    struct UTrapframe *utf;

    if (UXSTACKTOP - PGSIZE <= tf->tf_esp &&
        tf->tf_esp < UXSTACKTOP)
        utf = (struct UTrapframe *) (tf->tf_esp -
25         sizeof (struct UTrapframe) - 4);
    else
        utf = (struct UTrapframe *) (UXSTACKTOP -
        sizeof (struct UTrapframe));
    user_mem_assert(curenv,
30         (void*) utf,
        sizeof(struct UTrapframe),
        PTE_U | PTE_W);

    utf->utf_esp      = tf->tf_esp;
35 utf->utf_eflags     = tf->tf_eflags;
    utf->utf_eip       = tf->tf_eip;
    utf->utf_regs      = tf->tf_regs;
    utf->utf_err        = tf->tf_err;
    utf->utf_fault_va  = fault_va;

40
    curenv->env_tf.tf_eip = (uint32_t)
        curenv->env_pgfault_upcall;
    curenv->env_tf.tf_esp = (uint32_t) utf;
    env_run(curenv);
45 }

// Destroy the environment that caused the fault.
cprintf("[%08x] user fault va %08x ip %08x\n",
        curenv->env_id, fault_va, tf->tf_eip);
50 print_trapframe(tf);
    env_destroy(curenv);
}

```

源代码解释：

先是一次的判断，如果CS的后两位是0，就陷入内核，如果不是，先判断用户是否和用户态错误栈进行了映射，如果没有注册过，则进行一些处理并返回，

然后构造数据结构，并赋值，这个数据结构将会传递给用户态处理函数。

这里我们新定义了一个数据结构UTrapframe，当用户堆栈切换成用户错误栈时，会把他压栈，保存触发缺页中断的进程的上下文的信息，以便之后能回到源程序继续运行它和Trapframe类似，他保存了eip、err、esp的信息，另外还保存了出错指令涉及的具体地址，但没有存寄存器，因涉及的前后两个是两个相同的用户态，不会用段的切换。

然后，我们先要判断这个缺页中断是不是递归，我们知道，那个用户态错误栈的有效地址是UXSTACKTOP - PGSIZE 到UXSTACKTOP-1，所以，我们只要判断tf->tf_esp是不是在这个范围内，如果在这个范围内，则说明已经在用户态错误栈了，是嵌套的缺页错误，所以，我们要esp-4，压入一个（32位）空字，在压入一个UTrapframe，如果不在那个范围里，则只需要压入一个UTrapframe。

然后就是切换到用户定义的错误处理程序中处理，这里我们会把eip设置为pgfault_upcall，这是我们下一题要完成的函数，他是所有用户出错程序的入口，然后可以直接调用env_runn来跳转到错误处理程序。

Exercise 5

Description

在 lib/pfentry.S 中完成pgfault_upcall 函数。有意思的是，我们必须返回到原来的用户程序发生缺页错误的程序位置。你无须通过内核返回，仅仅直接返回就可以了。

Solution

pgfault_upcall 是所有用户页错误处理程序的入口，由这里调用用户自定义的处理程序，并在处理完成后，从错误栈中保存的UTrapframe中恢复相应信息，然后跳回到发生错误之前的指令，恢复原来的进程运行。

```
.text
.globl _pgfault_upcall
_pgfault_upcall:
    // Call the C page fault handler.
5     pushl %esp           // function argument: pointer to UTF
    movl _pgfault_handler, %eax
    call *%eax
    addl $4, %esp        // pop function argument

10    // LAB 4: Your code here.

    movl    0x30(%esp), %eax
    subl    $0x4, %eax
    movl    %eax, 0x30(%esp)
15    movl    0x28(%esp), %ebx
    movl    %ebx, (%eax)
```

```
20 // Restore the trap-time registers.
    // LAB 4: Your code here.

    addl    $0x8, %esp
    popal

25 // Restore eflags from the stack.
    // LAB 4: Your code here.

    addl    $0x4, %esp
    popfl

30 // Switch back to the adjusted trap-time stack.
    // LAB 4: Your code here.

    pop     %esp

35 // Return to re-execute the instruction that faulted.
    // LAB 4: Your code here.

    ret
```

源代码解释：

line 5：这一段就是调用具体的pagefault处理程序，把page_fault_handler移到寄存器eax里面，然后再调用eax，接下来就是要从用户的错误处理程序中返回，在前面设置的page_fault_handler函数里面，有个操作是，把 UTrapframe 压入了堆栈里面，然后我们现在现在就可以在从栈里pop出来，得到 UTrapframe 记录的原程序的，返回调用前的状态。

line 12：这一段是把esp减去4，然后传递给eax使用，留出来4个字节空白区域，这是为下一步做准备的。

line 15：这一步就是把缺页中断处理进程的eip放入前面空出来的4个字节中，以便恢复到原程序继续运行。这一步为我们同时变换堆栈和eip提供了可能。

line 21：这段是esp+8，即跳过utf_fault_va和 errcode，指向reg.edi，然后使用popal恢复所有的寄存器

line 27：然后esp+4，此时已经正好指向eip，并popfl恢复eflags标志寄存器。之后就是弹出栈。

最后调用ret，切换到原来出错程序运行位置。

Exercise 6

Description

完成 `set_pgfault_handler()`, 相应的函数原型在 `lib/pgfault.c` 中。

Solution

前面我们就知道了缺页中断处理的过程，如果发生缺页中断，则调用刚刚写的 `_pgfault_upcall`，即调用缺页处理程序，并返回用户态继续执行。其中，调用缺页处理程序先要运行 `set_pgfault_handler`，即申请用户错误栈的空间，然后再运行 `sys_env_set_pgfault_upcall`，即注册用户错误处理程序。接下来就是实现 `set_pgfault_handler` 这个函数。

```
void
set_pgfault_handler(void (*handler)(struct UTrapframe *utf))
{
    int r;

    if (_pgfault_handler == 0) {
        // First time through!

        // LAB 4: Your code here.

        if ((r = sys_page_alloc(0,
            (void*) (UXSTACKTOP - PGSIZE),
            PTE_U | PTE_P | PTE_W)) < 0)
            panic("set_pgfault_handler: %e", r);

        sys_env_set_pgfault_upcall(0, _pgfault_upcall);
    }

    // Save handler pointer for assembly to call.
    _pgfault_handler = handler;
}
```

先判断 `pgfault_handler` 是不是第一次赋值，如果是则要先申请分配用户错误栈的空间，再调用 `sys_env_set_pgfault_upcall` 进行注册，然后直接运行 `handler` 程序。

Exercise 7

Description

完成 `fork` 和 `pgfault` 的相应处理，相应的函数原型在 `lib/fork.c` 中。

Solution

```
envid_t
fork(void)
```

```

{
    // LAB 4: Your code here.
    5    envid_t envid;
    uint8_t *addr;
    int r;
    extern unsigned char end[];
    set_pgfault_handler(pgfault);
    10    envid = sys_exofork();
    if (envid < 0)
        panic("sys_exofork: %e", envid);
    //child
    // can't set pgh here, must set before child running,
    // because when child runs, it will make a page fault.
    15    if (envid == 0) {
        env = &envs[ENVX(sys_getenvid())];
        return 0;
    }
    20    //parent
    for (addr = (uint8_t*) UTEXT; addr < end; addr += PGSIZE)
        duppage(envid, VPN(addr));
    duppage(envid, VPN(&addr));

    25    //copy user exception stack

    if ((r = sys_page_alloc(envid,
                            (void *) (UXSTACKTOP - PGSIZE),
                            PTE_P | PTE_U | PTE_W)) < 0)
        30    panic("sys_page_alloc: %e", r);
    r = sys_env_set_pgfault_upcall(envid,
                                   env->env_pgfault_upcall);

    //set child status
    35    if ((r = sys_env_set_status(envid, ENV_RUNNABLE)) < 0)
        panic("sys_env_set_status: %e", r);
    return envid;
}

```

该函数是创建新进程的总入口，他先调用前面已经完成的set pgfault handler，对缺页错误处理函数进行检查有没有分配错误栈空间以及进行注册，然后调用sys_exofork创建一个新的进程，接着可写的页或者COW页都映射为COW页，然后就是给错误栈创建新的物理页，然后为子进程设置页错误处理程序，即，把pgfault upcall设置为所有用户页错误处理程序的总入口，最后把子进程标记为runnable。

```

static int
duppage(envid_t envid, unsigned pn)
{
    5    int r;
    void *addr;
    pte_t pte;

    // LAB 4: Your code here.

```

```

10     addr = (void *) ((uint32_t) pn * PGSIZE);
    pte = vpt[VPN(addr)];
    if ((pte & PTE_W) > 0 || (pte & PTE_COW) > 0) {
        if ((r = sys_page_map(0, addr, envid, addr,
                               PTE_U | PTE_P | PTE_COW)) < 0)
            panic("duppage: page re-mapping failed at 1 :\n
15                %e", r);

        if ((r = sys_page_map(0, addr, 0, addr,
                               PTE_U | PTE_P | PTE_COW)) < 0)
            panic("duppage: page re-mapping failed at 2 :\n
20                %e", r);
    } else {
        if ((r = sys_page_map(0, addr, envid, addr,
                               PTE_U | PTE_P)) < 0)
            panic("duppage: page re-mapping failed at 3 :\n
25                %e", r);
    }
    return 0;
}

```

缺页错误处理流程：如果发生缺页错误，先会调用_pgfault_upcall，它将继续调用fork的pgfault函数，pgfault首先会检查这个错误是不是写错误(检查FEC_WR)并且查看PTE位是否标志着PTE_COW。如果不是，则panic。不然，则会分配一个新的页，先将地址和物理页大小对齐，然后复制旧页的数据，并且把新页映射到旧页的地址处，并且设置读写权限以代替原来的标志。

```

static void
pgfault(struct UTrapframe *utf)
{
    void *addr = (void *) utf->utf_fault_va;
    uint32_t err = utf->utf_err;
5     int r;

    // LAB 4: Your code here.

10     if (!(err & FEC_WR))
        panic("Page fault: not a write access.");

    if (!(vpt[VPN(addr)] & PTE_COW))
        panic("Page fault: not a COW page.");
15     if ((r = sys_page_alloc(0, PFTEMP,
                              PTE_U|PTE_W|PTE_P)) < 0)
        panic("Page fault: sys_page_alloc err %e.", r);

    memmove(PFTEMP, (void *)PTE_ADDR(addr), PGSIZE);

20     if ((r = sys_page_map(0, PFTEMP, 0,
                            (void *)PTE_ADDR(addr),
                            PTE_U|PTE_W|PTE_P)) < 0)
25         panic("Page fault: sys_page_map err %e.", r);
}

```

```
if ((r = sys_page_unmap(0, PFTEMP)) < 0)
    panic("Page fault: sys_page_unmap err %e.", r);
}
```

III. Preemptive Multitasking and IPC

Exercise 8

Description

修改 kern/trapentry.S 和 kern/trap.c 来初始化适当的 IDT 的入口和相应的 IRQ 处理程序。然后修改在 env_alloc() 下面的程序, 确认你的用户环境总是运行在中断允许的情况之下。

Solution

外部的中断称为 IRQ(interrupt requests), 编号从0 到15 一共16 个, 其中0 号即对应时钟中断。在 JOS 中, 为了区分外部中断和处理器的异常中断, 定义了一个 `IRQ_OFFSET = 32D`。因此, 在 IDT 中, 时钟中断的内核处理程序位于 `IDT [IRQ_OFFSET]`, 即 `IDT [32]`。其它 IRQ 依此类推。

由此可知, 我们需要在 kern/trapentry.S 和 kern/trap.c 中初始化 IDT 的入口和相应的 IRQ 处理程序。

以下给出了设置处理器异常中断在内的完整的 IDT 入口及相应处理程序的初始化代码。

Listing 9: kern/trapentry.S

```
TRAPHANDLER_NOEC(trap_mchk,      T_MCHK)
TRAPHANDLER_NOEC(trap_simderr,   T_SIMDERR)

TRAPHANDLER_NOEC(trap_syscall,   T_SYSCALL)

5
// IRQ
TRAPHANDLER_NOEC(inter_irq_0,    IRQ_OFFSET+0);
TRAPHANDLER_NOEC(inter_irq_1,    IRQ_OFFSET+1);
TRAPHANDLER_NOEC(inter_irq_2,    IRQ_OFFSET+2);
10 TRAPHANDLER_NOEC(inter_irq_3,    IRQ_OFFSET+3);
TRAPHANDLER_NOEC(inter_irq_4,    IRQ_OFFSET+4);
TRAPHANDLER_NOEC(inter_irq_5,    IRQ_OFFSET+5);
TRAPHANDLER_NOEC(inter_irq_6,    IRQ_OFFSET+6);
TRAPHANDLER_NOEC(inter_irq_7,    IRQ_OFFSET+7);
15 TRAPHANDLER_NOEC(inter_irq_8,    IRQ_OFFSET+8);
TRAPHANDLER_NOEC(inter_irq_9,    IRQ_OFFSET+9);
TRAPHANDLER_NOEC(inter_irq_10,   IRQ_OFFSET+10);
TRAPHANDLER_NOEC(inter_irq_11,   IRQ_OFFSET+11);
TRAPHANDLER_NOEC(inter_irq_12,   IRQ_OFFSET+12);
20 TRAPHANDLER_NOEC(inter_irq_13,   IRQ_OFFSET+13);
```

```
TRAPHANDLER_NOEC(inter_irq_14, IRQ_OFFSET+14);
TRAPHANDLER_NOEC(inter_irq_15, IRQ_OFFSET+15);
```

Listing 10: kern/trap.c

```

extern void inter_irq_0();
extern void inter_irq_1();
extern void inter_irq_2();
extern void inter_irq_3();
5 extern void inter_irq_4();
extern void inter_irq_5();
extern void inter_irq_6();
extern void inter_irq_7();
extern void inter_irq_8();
10 extern void inter_irq_9();
extern void inter_irq_10();
extern void inter_irq_11();
extern void inter_irq_12();
extern void inter_irq_13();
15 extern void inter_irq_14();
extern void inter_irq_15();

SETGATE(idt[IRQ_OFFSET+0], 0, GD_KT, inter_irq_0, 0);
SETGATE(idt[IRQ_OFFSET+1], 0, GD_KT, inter_irq_1, 0);
20 SETGATE(idt[IRQ_OFFSET+2], 0, GD_KT, inter_irq_2, 0);
SETGATE(idt[IRQ_OFFSET+3], 0, GD_KT, inter_irq_3, 0);
SETGATE(idt[IRQ_OFFSET+4], 0, GD_KT, inter_irq_4, 0);
SETGATE(idt[IRQ_OFFSET+5], 0, GD_KT, inter_irq_5, 0);
SETGATE(idt[IRQ_OFFSET+6], 0, GD_KT, inter_irq_6, 0);
25 SETGATE(idt[IRQ_OFFSET+7], 0, GD_KT, inter_irq_7, 0);
SETGATE(idt[IRQ_OFFSET+8], 0, GD_KT, inter_irq_8, 0);
SETGATE(idt[IRQ_OFFSET+9], 0, GD_KT, inter_irq_9, 0);
SETGATE(idt[IRQ_OFFSET+10], 0, GD_KT, inter_irq_10, 0);
SETGATE(idt[IRQ_OFFSET+11], 0, GD_KT, inter_irq_11, 0);
30 SETGATE(idt[IRQ_OFFSET+12], 0, GD_KT, inter_irq_12, 0);
SETGATE(idt[IRQ_OFFSET+13], 0, GD_KT, inter_irq_13, 0);
SETGATE(idt[IRQ_OFFSET+14], 0, GD_KT, inter_irq_14, 0);
SETGATE(idt[IRQ_OFFSET+15], 0, GD_KT, inter_irq_15, 0);

35 // Setup a TSS so that we get the right stack
// when we trap to the kernel.
ts.ts_esp0 = KSTACKTOP;
ts.ts_ss0 = GD_KD;
```

在 JOS 的实现中, 外部中断是关闭着的, 我们现在需要在用户模式下开启它。是否响应外部中断是由 %eflags 寄存器中的 FL_IF 标志位控制的, 所以我们只需在用户进程(即 JOS 中的用户环境)中确保 FL_IF 是置位的就可以了。在 kern/env.c 的指定位置完成如下代码即可:

```
// Enable interrupts while in user mode.
// LAB 4: Your code here.
e->env_tf.tf_eflags |= FL_IF;
```


Exercise 9

Description

修改内核的 `trap_dispatch()` 函数, 让它在时钟中断发生的时候调用 `sched_yield()` 来具体的找出和运行不同的进程。

你现在可以使用 `user/spin` 测试程序来测试你的程序啦: 父进程创建子进程, `sys_yield()` 调用它们进行轮转两次, 让他们轮转占用控制 CPU 资源。最后结束子进程, 然后正常的退出。

Solution

完成了练习 8 之后, 运行程序测试, 但都如帮助文档所言, 无一不立即发生了内核错误, 程序无法继续运行。这是由于我们建立了内核响应时钟中断的机制, 但是没有在 `dispatch` 程序中完成相应的 handler, 而这正是练习 9 的容。

本练习中我们需要完成的是时钟中断的处理。根据帮助文档的提示, 我们注意到在 `kern/init.c` 的 `i386_init()` 函数中, 分别新增了 8259A 中断控制器和 8253 定器的初始化程序 `pic_init()` (定义于 `kern/picirq.c`) 和 `kclock_init()` (定义于 `kern/kclock.c`)。操作系统便是基于这两个芯片完成时钟中断的产生和处理的。

查看 `kclock_init()` 的代码可以知道, 其产生时钟中断的频率为 100 times/sec, 即每 10ms 产生一次时钟中断, 我们需要调用调度程序来执行程序的调度以响应此中断。

只需在 `kern/trap.c` 的 `trap_dispatch()` 中为时钟中断添加 handler (此处应添加 `sched_yield()`) 就可以了。

以下是 `trap_dispatch()` 的代码, 方便对比处理外部中断和内核中断的一致性。

```
static void
trap_dispatch(struct Trapframe *tf)
{
    ...

    // Handle clock and serial interrupts.
    // LAB 4: Your code here.

    if (tf->tf_trapno == IRQ_OFFSET + IRQ_TIMER)
        sched_yield ();

    // Unexpected trap: The user process or
    // the kernel has a bug.
    print_trapframe(tf);
    if (tf->tf_cs == GD_KT)
        panic("unhandled trap in kernel");
    else {
        env_destroy(curenv);
        return;
    }
}
```

```
}  

```

Inter-Process Communication

为防止用户程序由于 bug 或恶意攻击(如进行无限循环)而使得系统无法获得 CPU 的控制权,我们需要扩展 JOS 内核使其能响应时钟中断。这样内核可以强制取得 CPU 的控制权,保证了系统的稳定和安全。

Exercise 10

Description

完成 kern/syscall.c 中的 sys_ipc_recv 和 sys_ipc_can_send 函数。当你调用 env_id2env 的时候,应当把 checkperm 的标志位置0。这意味着进程运行向其他任何进程发送我们的 IPC 消息。内核不会对这些做权限检查,当然,必须确定这些目标的 env_id 是有效的。

完成 lib/ipc.c 中的 ipc_recv 和 ipc_send 函数。

使用 user/pingpong 和 user/primes 函数来测试你的 IPC 机制。你会发现 user/primes.c 的有趣地方-隐藏在背后的 forking 和 IPC。

Solution

实现 IPC 机制前首先需要更改 struct env, 增加以下5个成员:

1. env_ipc_recving: 当进程使用 sys ipc recv() 等待信息时, 会将这个成员置为1, 然后阻塞等待; 当一个进程像它发消息解除阻塞后, 发送进程将此成员修改为0;
2. env_ipc_dstva: 如果进程要接受消息, 并且是传送页, 则该地址 \leq UTOP;
3. env_ipc_value: 若等待消息的进程接受到了消息, 发送方将接受方此成员置为消息值;
4. env_ipc_from: 发送方负责设置该成员为自己的env_id号;
5. env_ipc_perm: 如果进程要接受消息, 并且传送页, 那么发送方发送页以后将传送的页权限传给这个成员。

Listing 11: inc/env.h

```
5 // Lab 4 IPC  
// env is blocked receiving  
bool env_ipc_recving;  
// va at which to map received page  
void *env_ipc_dstva;  
// data value sent to us  
uint32_t env_ipc_value;  
// env_id of the sender
```

```

10  envid_t env_ipc_from;
    // perm of page mapping received
    int env_ipc_perm;

```

之后按注释提示完成 两个系统调用 和 两个库函数：

Listing 12: kern/syscall.c

```

static int
sys_ipc_recv(void *dstva)
{
    // LAB 4: Your code here.

5   if (dstva < (void *) UTOP &&
        ROUNDDOWN(dstva, PGSIZE) != dstva)
        return -E_INVAL;

10   curenv->env_ipc_dstva = dstva;
    curenv->env_ipc_recving = 1;
    curenv->env_ipc_from = 0;
    curenv->env_status = ENV_NOT_RUNNABLE;
    sched_yield ();

15   return 0;
}

```

Listing 13: kern/syscall.c

```

static int
sys_ipc_try_send(envid_t envid, uint32_t value,
                 void *srcva, unsigned perm)
{
5   // LAB 4: Your code here.
    struct Env *dstenv;
    int r;

    // target env does not exist
10   if ((r = envid2env (envid, &dstenv, 0)) < 0)
        return -E_BAD_ENV;

    // target env is not blocked or message has been sent
15   if (dstenv->env_ipc_recving == 0 ||
        dstenv->env_ipc_from != 0 )
        return -E_IPC_NOT_RECV;

    // srcva < UTOP but not page aligned
20   if (srcva < (void *) UTOP &&
        ROUNDDOWN(srcva, PGSIZE) != srcva)
        return -E_INVAL;

    if (srcva < (void *) UTOP) {
        // check permission
25   if ((perm & PTE_U) == 0 || (perm & PTE_P) == 0)

```

```

        return -E_INVALID;
        // PTE_USER = PTE_U | PTE_P | PTE_W | PTE_AVAIL
        if ((perm & ~PTE_USER) > 0)
            return -E_INVALID;
30    }

    pte_t *pte;
    struct Page *p;

35    // the page is not mapped in current env
    if (srcva < (void *) UTOP &&
        (p = page_lookup(curenv->env_pgdir,
            srcva, &pte)) == NULL)
        return -E_INVALID;

40    if (srcva < (void *) UTOP &&
        (*pte & PTE_W) == 0 && (perm & PTE_W) > 0)
        return -E_INVALID;

45    // will send a page
    if (srcva < (void *) UTOP &&
        dstenv->env_ipc_dstva != 0) {
        if (page_insert(dstenv->env_pgdir, p,
            dstenv->env_ipc_dstva,
50            perm) < 0)
            return -E_NO_MEM;
        dstenv->env_ipc_perm = perm;
    }

55    dstenv->env_ipc_from = curenv->env_id;
    dstenv->env_ipc_value = value;
    dstenv->env_status = ENV_RUNNABLE;
    dstenv->env_ipc_recving = 0;
    dstenv->env_tf.tf_regs.reg_eax = 0;

60

    curenv->env_ipc_send_to = -1;
    curenv->env_ipc_send_succ++;
    dstenv->env_ipc_recv_min_send_succ = 0xffffffff;
65    struct Env *envptr;
    for (envptr = envs+1; envptr < envs+NENV; envptr++)
        if (envptr->env_ipc_send_to == env_id)
            if (envptr->env_ipc_send_succ <
                dstenv->env_ipc_recv_min_send_succ)
70                dstenv->env_ipc_recv_min_send_succ =
                    envptr->env_ipc_send_succ;

    return 0;
}

```

Listing 14: lib/ipc.c

```

uint32_t
ipc_recv(env_id_t *from_env_store, void *pg, int *perm_store)
{

```

```
5      // LAB 4: Your code here.

    int r;
    if (pg != NULL)
        r = sys_ipc_recv((void *) UTOP);
    else
10        r = sys_ipc_recv(pg);
    struct Env *curenv = (struct Env *) envs +
                        ENVX (sys_getenvid());
    if (from_env_store != NULL)
        *from_env_store = r < 0 ? 0 : curenv->env_ipc_from;
15    if (perm_store != NULL)
        *perm_store = r < 0 ? 0 : curenv->env_ipc_perm;
    if (r < 0)
        return r;
    return curenv->env_ipc_value;
20 }

void
ipc_send(envid_t to_env, uint32_t val, void *pg, int perm)
{
25     // LAB 4: Your code here.

    int r;
    while ((r = sys_ipc_try_send(to_env, val,
        pg != NULL ? pg : (void *) UTOP, perm)) < 0)
30    {
        if (r != -E_IPC_NOT_RECV)
            panic("ipc_send: send message error %e", r);
        sys_yield();
    }
35 }
```

至此 Lab 4 全部完成, 运行 make grade 检验通过。

Acknowledgement

武政伟(1110548) 完成第 1,2 题, 撰写并整理全部文档;

袁兆争(1110556), 张超(1110557) 完成第 3-7 题代码及文档;

马璇(1110580), 王少尧(1110539) 完成第 8-10 题代码及文档。