操作系统JOS实习第四次报告

张弛 00848231,
zhangchitc@gmail.com

April 24, 2011

Contents

1 Introduction

我在实验中主要参考了华中科技大学邵志远老师写的JOS实习指导，在邵老师的主页上http://grid.hust.edu.cn/zyshao/OSEngineering.htm 可以找到。但是这次实验的指导远远不如lab1的指导详尽，所以我这里需要补充的内容会很多。

内联汇编请参考邵老师的第二章讲义，对于语法讲解的很详细。

2 User-level Environment Creation and Cooperative Multitasking

这个部分的MIT文档讲解的比较详细，细节的串接都比较清楚。结合代码的注释写起来不是很困难。

2.1 Round-Robin Scheduling

> **Exercise 1.** Implement round-robin scheduling in sched_yield() as described above. Don't forget to modify syscall() to dispatch sys_yield().
>
> Modify kern/init.c to create three (or more!) environments that all run the program user/yield.c. You should see the environments switch back and forth between each other five times before terminating, like this:
>
> ```
> ...
> Hello, I am environment 00001001.
> Hello, I am environment 00001002.
> Hello, I am environment 00001003.
> Back in environment 00001001, iteration 0.
> Back in environment 00001002, iteration 0.
> Back in environment 00001003, iteration 0.
> Back in environment 00001001, iteration 1.
> Back in environment 00001002, iteration 1.
> Back in environment 00001003, iteration 1.
> ...
> ```
>
> After the yield programs exit, the idle environment should run and invoke the JOS kernel debugger. If any of this does not happen, then fix your code before proceeding.

sched_yield() 函数比较简单，直接贴代码了：

kern/sched.c: sched_yield()

```
1  void
2  sched_yield(void)
3  {
4      struct Env *curenvptr = curenv;
5
6      if (curenv == NULL)
7          curenvptr = envs;
8
9      int round = 0;
```

```
10      for (curenvptr ++; round < NENV; round ++, curenvptr ++) {
11
12          if (curenvptr >= envs + NENV) {
13              curenvptr = envs + 1;
14          }
15
16          if (curenvptr->env_status == ENV_RUNNABLE)
17              env_run (curenvptr);
18      }
19
20      // Run the special idle environment when nothing else is runnable.
21      if (envs[0].env_status == ENV_RUNNABLE)
22          env_run(&envs[0]);
23      else {
24          cprintf("Destroyed all environments - nothing more to do!\n");
25          while (1)
26              monitor(NULL);
27          }
28  }
```

然后修改kern/syscall.c添加相关的分发机制，然后在kern/init.c中系统启动之初创建user_idle以后再创建user_yield，这个用户程序的功能就是作五次`sys_yield()`的系统调用，并且切换时打印相关的消息:

kern/init.c: i386_init()

```
1           // Should always have an idle process as first one.
2           ENV_CREATE(user_idle);
3           ENV_CREATE(user_yield);
4           ENV_CREATE(user_yield);
5           ENV_CREATE(user_yield);
```

那么启动JOS后应该打印出下列消息：（注意，因为是使用Round Robin策略切换，所以顺序应该是确定的）

```
qemu -hda obj/kern/kernel.img -serial mon:stdio
6828 decimal is 15254 octal!
Hooray! Passed all test cases for stdlib!!
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
page_check() succeeded!
check_boot_pgdir() succeeded!
enabled interrupts: 1 2
            Setup timer interrupts via 8259A
enabled interrupts: 0 1 2
            unmasked timer interrupt
[00000000] new env 00001000
[00000000] new env 00001001
[00000000] new env 00001002
[00000000] new env 00001003
Hello, I am environment 00001001.
Hello, I am environment 00001002.
Hello, I am environment 00001003.
Back in environment 00001001, iteration 0.
Back in environment 00001002, iteration 0.
Back in environment 00001003, iteration 0.
Back in environment 00001001, iteration 1.
Back in environment 00001002, iteration 1.
Back in environment 00001003, iteration 1.
Back in environment 00001001, iteration 2.
Back in environment 00001002, iteration 2.
Back in environment 00001003, iteration 2.
Back in environment 00001001, iteration 3.
```

```
Back in environment 00001002, iteration 3.
Back in environment 00001003, iteration 3.
Back in environment 00001001, iteration 4.
All done in environment 00001001.
[00001001] exiting gracefully
[00001001] free env 00001001
Back in environment 00001002, iteration 4.
All done in environment 00001002.
[00001002] exiting gracefully
[00001002] free env 00001002
Back in environment 00001003, iteration 4.
All done in environment 00001003.
[00001003] exiting gracefully
[00001003] free env 00001003
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
```

Question

In your implementation of env_run() you should have called lcr3(). Before and after the call to lcr3(), your code makes references (at least it should) to the variable e, the argument to env_run. Upon loading the %cr3 register, the addressing context used by the MMU is instantly changed. But a virtual address (namely e) has meaning relative to a given address context--the address context specifies the physical address to which the virtual address maps. Why can the pointer e be dereferenced both before and after the addressing switch?

我们先来回顾一下`env_run()`里的具体代码：

kern/env.c: env_run()

```
1  void
2  env_run(struct Env *e)
3  {
4      if (curenv != e) {
5          curenv = e;
6          curenv->env_runs ++;
7          lcr3 (curenv->env_cr3);
8      }
9
10     env_pop_tf (&curenv->env_tf);
11
12     panic("env_run not yet implemented");
13 }
```

我们尝试在未切换到用户页地址之前打印出e在系统页表中的地址，看看是个什么样子：

```
[00000000] new env 00001000
[00000000] new env 00001001
[00000000] new env 00001002
[00000000] new env 00001003
zhangchi: e ptr = f01f207c, KERNBASE = f0000000
zhangchi: e ptr = f01f207c, KERNBASE = f0000000
Hello, I am environment 00001001.
```

```
zhangchi: e ptr = f01f207c, KERNBASE = f0000000
zhangchi: e ptr = f01f20f8, KERNBASE = f0000000
zhangchi: e ptr = f01f20f8, KERNBASE = f0000000
Hello, I am environment 00001002.
```

可以看到e的地址是在KERNBASE以上的系统区，而很明显，在所有用户页地址空间里，KERNBASE以上的空间都是和boot_pgdir是一样的，指向同一片物理内存区域。所以我们在切换页表的前后e是不受影响的。

2.2 System Calls for Environment Creation

这个小节里我一开始最迷惑的就是这个`sys_exofork()`对于父进程和子进程的返回值的区别。因为操作系统应该是可以递归的让子进程不断创建子进程的，**那这里子进程调用返回0又是如何解释？如果他调用老是返回0那如何让其递归的创建子进程？**

这里需要重申一下`fork()`的机制，因为创建出的子进程和父亲是同一进程，即它们在汇编代码级别是一模一样的，父亲创建一个子进程应该是在一个int 30的中断调用里完成的，这个中断去调用相应的系统`fork()`代码。比如我们可以看看obj/user/dumbfork.asm看看相关的代码：

obj/user/dumbfork.asm

```
118 envid_t
119 dumbfork(void)
120 {
121   800119:       55                      push   %ebp
122   80011a:       89 e5                   mov    %esp,%ebp
123   80011c:       53                      push   %ebx
124   80011d:       83 ec 24                sub    $0x24,%esp
125 static __inline envid_t sys_exofork(void) __attribute__((always_inline));
126 static __inline envid_t
127 sys_exofork(void)
128 {
129         envid_t ret;
130         __asm __volatile("int %2"
131   800120:       bb 07 00 00 00          mov    $0x7,%ebx
132   800125:       89 d8                   mov    %ebx,%eax
133   800127:       cd 30                   int    $0x30
134   800129:       89 c3                   mov    %eax,%ebx
135         // The kernel will initialize it with a copy of our register state,
136         // so that the child will appear to have called sys_exofork() too -
137         // except that in the child, this "fake" call to sys_exofork()
138         // will return 0 instead of the envid of the child.
139         envid = sys_exofork();
140         if (envid < 0)
141   80012b:       85 c0                   test   %eax,%eax
142   80012d:       79 20                   jns    80014f <dumbfork+0x36>
143                 panic("sys_exofork: %e", envid);
```

这个时候，中断调用后创建的子进程和父进程有一样的进程状态，父亲的调用完以后因为这个`fork()`有一个返回值代表子进程的pid，这个返回值按照lab3的规定应该是在eax里的，所以int 30的下一句汇编代码应该就是把这个eax赋值给相应的变量，如上面代码中800129地址的指令。但是如果切换到子进程继续运行的话，**不能让它也接受一个和父进程一样的eax（因为这个时**

候eax不就是它自己的pid么），所以为了区分就让它的返回值等于0就好了。即让其eax寄存器设置为0.

其他就没什么太大的疑问了。

> **Exercise 2.** Implement the system calls described above in kern/syscall.c. You will need to use various functions in kern/pmap.c and kern/env.c, particularly envid2env(). For now, whenever you call envid2env(), pass 1 in the checkperm parameter. Be sure you check for any invalid system call arguments, returning -E_INVAL in that case. Test your JOS kernel with user /dumbfork and make sure it works before proceeding.

我们一个函数一个函数的看，首先是刚才提到的`sys_exofork()`

kern/syscall.c: sys_exofork()

```
1  static envid_t
2  sys_exofork(void)
3  {
4      struct Env *newenv;
5      int r;
6
7      if ((r = env_alloc (&newenv, sys_getenvid())) < 0)
8          return r;
9
10     // set not runnnable
11     newenv->env_status = ENV_NOT_RUNNABLE;
12
13     // copy trapframe
14     newenv->env_tf = curenv->env_tf;
15
16     // make the child env's return value zero
17     newenv->env_tf.tf_regs.reg_eax = 0;
18
19     return newenv->env_id;
20 }
```

这里最重要的就是第17行，让创建好的子进程的eax为0，那么它从系统调用得到的返回值就是0了。

然后是`env_set_status()`，很简单，照着注释写就行

kern/syscall.c: env_set_status()

```
1  static int
2  sys_env_set_status(envid_t envid, int status)
3  {
4      if (status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE)
5          return -E_INVAL;
6
7      struct Env *envptr;
8      int r;
9
10     if ((r = envid2env (envid, &envptr, 1)) < 0)
11         return r;
12
13     envptr->env_status = status;
14
15     return 0;
16 }
```

接下来是env_page_alloc()

kern/syscall.c: sys_page_alloc()

```
1  static int
2  sys_page_alloc(envid_t envid, void *va, int perm)
3  {
4      if (va >= (void *)UTOP)
5          return -E_INVAL;
6
7      if ((perm & PTE_U) == 0 || (perm & PTE_P) == 0)
8          return -E_INVAL;
9      // PTE_USER = PTE_U | PTE_P | PTE_W | PTE_AVAIL
10     if ((perm & ~PTE_USER) > 0)
11         return -E_INVAL;
12
13     struct Env *e;
14     if (envid2env (envid, &e, 1) < 0)
15         return -E_BAD_ENV;
16
17     struct Page *p;
18     if (page_alloc (&p) < 0)
19         return -E_NO_MEM;
20
21     if (page_insert (e->env_pgdir, p, va, perm) < 0) {
22         page_free (p);
23         return -E_NO_MEM;
24     }
25
26     memset (page2kva (p), 0, PGSIZE);
27
28     return 0;
29 }
```

注意，这里最重要的是最后一句memset，这个在注释中没有提及，但是在邵老师的讲义里面提到了，**申请了新页面以后注意要给用户进程清空以防止脏数据的情况**。请认真阅读绍老师的讲义。

然后来看sys_page_map()

kern/syscall.c: sys_page_map()

```
1  static int
2  sys_page_map(envid_t srcenvid, void *srcva,
3               envid_t dstenvid, void *dstva, int perm)
4  {
5      if (srcva >= (void *)UTOP || ROUNDUP (srcva, PGSIZE) != srcva
6      ||  dstva >= (void *)UTOP || ROUNDUP (dstva, PGSIZE) != dstva)
7          return -E_INVAL;
8
9      if ((perm & PTE_U) == 0 || (perm & PTE_P) == 0)
10         return -E_INVAL;
11     // PTE_USER = PTE_U | PTE_P | PTE_W | PTE_AVAIL
12     if ((perm & ~PTE_USER) > 0)
13         return -E_INVAL;
14
15     struct Env *srcenv;
16     if (envid2env (srcenvid, &srcenv, 1) < 0)
17         return -E_BAD_ENV;
18
19     struct Env *dstenv;
20     if (envid2env (dstenvid, &dstenv, 1) < 0)
21         return -E_BAD_ENV;
22
23     pte_t *pte;
```

```
24      struct Page *p = page_lookup (srcenv->env_pgdir, srcva, &pte);
25      if (p == NULL || ((perm & PTE_W) > 0 && (*pte & PTE_W) == 0))
26          return -E_INVAL;
27
28      if (page_insert (dstenv->env_pgdir, p, dstva, perm) < 0)
29          return -E_NO_MEM;
30
31      return 0;
32  }
```

没什么需要注意的地方，继续看`sys_page_unmap()`

kern/syscall.c: sys_page_unmap()

```
1   static int
2   sys_page_unmap(envid_t envid, void *va)
3   {
4       if (va >= (void *)UTOP || ROUNDUP (va, PGSIZE) != va)
5           return -E_INVAL;
6
7       struct Env *env;
8       if (envid2env (envid, &env, 1) < 0)
9           return -E_BAD_ENV;
10
11      page_remove (env->env_pgdir, va);
12
13      return 0;
14  }
```

到此为止这个Exercise涉及的全部代码就已经完成了，当然还有我们要对相应的系统调用号添加分发逻辑。然后我们来考虑运行一下user/dumbfork，首先来看看它的代码，这里节选它的主函数段：

user/dumbfork.c: umain()

```
9   void
10  umain(void)
11  {
12          envid_t who;
13          int i;
14
15          // fork a child process
16          who = dumbfork();
17
18          // print a message and yield to the other a few times
19          for (i = 0; i < (who ? 10 : 20); i++) {
20                  cprintf("%d:␣I␣am␣the␣%s!\n", i, who ? "parent" : "child");
21                  sys_yield();
22          }
23  }
```

这个代码的逻辑就是：父进程创建一个子进程，然后每次打印一条信息以后交出控制权，并且让父进程重复10次而子进程重复20次。

这个逻辑已经很明显了，我们尝试这运行一下，输入make run-dumbfork:

```
qemu -hda obj/kern/kernel.img -serial mon:stdio
6828 decimal is 15254 octal!
Hooray! Passed all test cases for stdlib!!
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
```

```
page_check() succeeded!
check_boot_pgdir() succeeded!
enabled interrupts: 1 2
            Setup timer interrupts via 8259A
enabled interrupts: 0 1 2
            unmasked timer interrupt
[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!
1: I am the child!
2: I am the parent!
2: I am the child!
3: I am the parent!
3: I am the child!
4: I am the parent!
4: I am the child!
5: I am the parent!
5: I am the child!
6: I am the parent!
6: I am the child!
7: I am the parent!
7: I am the child!
8: I am the parent!
8: I am the child!
9: I am the parent!
9: I am the child!
[00001001] exiting gracefully
[00001001] free env 00001001
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!
Type 'help' for a list of commands.
```

和程序的预期逻辑是一致的。

3 Copy-on-Write Fork

3.1 User-level page fault handling

3.2 Setting the Page Fault Handler

Exercise 3. Implement the sys_env_set_pgfault_upcall system call. Be sure to enable permission checking when looking up the environment ID of the target environment, since this is a "dangerous" system call.

很简单:

kern/syscall.c: sys_env_set_pgfault_upcall()

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

    e->env_pgfault_upcall = func;

    return 0;
}
```

3.3 Normal and Exception Stacks in User Environments

首先梳理一下几个栈之间的关系。根据inc/memlayout.h中对于虚拟地址空间的描述，他们的布局如下：

inc/memlayout.h

```
/*
 * Virtual memory map:                                Permissions
 *                                                    kernel/user
 *
 *    4 Gig -------->  +------------------------------+
 *                     |                              | RW/--
 *                     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 *                     :              .               :
 *                     :              .               :
 *                     :              .               :
 *                     |~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~| RW/--
 *                     |                              | RW/--
 *                     |   Remapped Physical Memory   | RW/--
 *                     |                              | RW/--
 *    KERNBASE ----->  +------------------------------+ 0xf0000000
 *                     |  Cur. Page Table (Kern. RW)  | RW/--  PTSIZE
 *    VPT,KSTACKTOP--> +------------------------------+ 0xefc00000      --+
 *                     |         Kernel Stack         | RW/--  KSTKSIZE   |
 *                     | - - - - - - - - - - - - - - -|                   PTSIZE
 *                     |      Invalid Memory (*)      | --/--             |
 *    ULIM     ------> +------------------------------+ 0xef800000      --+
 *                     |  Cur. Page Table (User R-)   | R-/R-  PTSIZE
 *    UVPT      ----> +------------------------------+ 0xef400000
 *                     |          RO PAGES            | R-/R-  PTSIZE
 *    UPAGES    ----> +------------------------------+ 0xef000000
 *                     |           RO ENVS            | R-/R-  PTSIZE
 * UTOP,UENVS ------> +------------------------------+ 0xeec00000
 * UXSTACKTOP -/       |     User Exception Stack     | RW/RW  PGSIZE
 *                     +------------------------------+ 0xeebff000
 *                     |       Empty Memory (*)       | --/--  PGSIZE
 *    USTACKTOP  ---> +------------------------------+ 0xeebfe000
 *                     |      Normal User Stack       | RW/RW  PGSIZE
 *                     +------------------------------+ 0xeebfd000
 *                     |                              |
 *                     |                              |
 *                     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 *                     .                              .
 *                     .                              .
 *                     .                              .
```

```
40  *                    |~~~~~~~~~~~~~~~~~~~~~~~~~~~~~|
41  *                    |     Program Data & Heap     |
42  *    UTEXT --------->  +-----------------------------+ 0x00800000
```

一共有三个栈：

[KSTACKTOP, KSTACKTOP – KSTKSIZE) ：
内核态系统栈

[UXSTACKTOP, UXSTACKTOP – PGSIZE) ：
用户态错误处理栈

[USTACKTOP, UTEXT) ：
用户态运行栈

他们之间的几个区别是：

- **内核态系统栈**运行的是内核的相关程序，在这里我们关注的仅仅是**在有中断被触发以后**，CPU会将自动将栈切换到内核栈上来。我们是在kern/trap.c的`idt_init()`中进行相应设置的：

kern/trap.c: idt_init()

```
1         // Setup a TSS so that we get the right stack
2         // when we trap to the kernel.
3         ts.ts_esp0 = KSTACKTOP;
4         ts.ts_ss0 = GD_KD;
5
6         // Initialize the TSS field of the gdt.
7         gdt[GD_TSS >> 3] = SEG16(STS_T32A, (uint32_t) (&ts),
8                                  sizeof(struct Taskstate), 0);
9         gdt[GD_TSS >> 3].sd_s = 0;
10
11        // Load the TSS
12        ltr(GD_TSS);
13
14        // Load the IDT
15        asm volatile("lidt␣idt_pd");
```

那么当一个中断被触发进入kern/trapentry.S中定义的中断服务程序时，所处在的栈就已经是内核栈了：

kern/trapentry.S

```
1  /*
2   * Lab 3: Your code here for _alltraps
3   */
4  _alltraps:
5
6      pushw    $0x0
7      pushw    %ds
8      pushw    $0x0
9      pushw    %es
10     pushal
11
12     movl     $GD_KD, %eax
13
```

```
14      movw    %ax, %ds
15      movw    %ax, %es
16
17      pushl   %esp
18
19      call trap
```

这里push的指令全都是写入KSTACKTOP以下的空间的，为了形成一个Trapframe的内存结构，最后一个pushl %esp是为了按照C Convention传入一个参数的句柄，因为trap 函数的定义为

kern/trap.c

```
1  void trap (struct Trapframe *tf)
```

需要一个参数句柄。

- **用户运行栈**则是用户程序运行中使用的栈，初始是在创建用户进程初始时设置的，在kern/env.c的`env_alloc()`中可以看到：

kern/env.c: env_alloc()

```
1          // Set up appropriate initial values for the segment registers.
2          // GD_UD is the user data segment selector in the GDT, and
3          // GD_UT is the user text segment selector (see inc/memlayout.h).
4          // The low 2 bits of each segment register contains the
5          // Requestor Privilege Level (RPL); 3 means user mode.
6          e->env_tf.tf_ds = GD_UD | 3;
7          e->env_tf.tf_es = GD_UD | 3;
8          e->env_tf.tf_ss = GD_UD | 3;
9          e->env_tf.tf_esp = USTACKTOP;
10         e->env_tf.tf_cs = GD_UT | 3;
11         // You will set e->env_tf.tf_eip later.
```

这个时候其实其虚拟地址对应的地方只有一页的物理地址，在载入ELF文件时分配的：

kern/env.c: load_icode()

```
1      // LAB 3: Your code here.
2      segment_alloc (e, (void*) (USTACKTOP - PGSIZE), PGSIZE);
```

如果用到了更多的空间，那么就会触发缺页中断，而转到内核栈上去运行中断处理程序。

注意用户运行栈是一边用一边分配的，而内核栈则固定了大小，而下面的用户态错误栈也是固定大小的。

- **用户态错误栈**是用户自己定义相应的中断处理程序后，相应处理程序运行时的栈。当用户进程调用前面的`sys_env_set_pgfault_upcall()`向系统注册缺页中断处理程序后，当用户程序触发缺页中断，那么

  1. 首先系统陷入内核态，栈位置从**用户运行栈**切换到**内核栈**，进入trap ()处理中断分发，进入page_fault_handler ()

2. 当确认是用户程序而不是内核触发了缺页中断后，（内核的话就直接panic了），为其在用户错误栈里分配一个UTrapframe的大小（但是这时仍在**内核栈**，我们只是在对应的页表空间里作的这样的操作）
3. 把栈切换到**用户错误栈**，运行相应的用户中断处理程序
4. 最后返回用户程序，栈恢复到**用户运行栈**

那么这个栈是在哪里被指定的呢？**注意，只有注册了自己的缺页中断服务程序的用户进程才会分配用户错误栈**，具体代码看到lib/pgfault.c中：

lib/pgfault.c: set_pgfault_handler()

```
1  void
2  set_pgfault_handler(void (*handler)(struct UTrapframe *utf))
3  {
4          int r;
5
6          if (_pgfault_handler == 0) {
7                  // First time through!
8                  // LAB 4: Your code here.
9                  panic("set_pgfault_handler not implemented");
10         }
11
12         // Save handler pointer for assembly to call.
13         _pgfault_handler = handler;
14 }
```

在设置中断服务程序时，要作的工作之一就是为该进程的用户错误栈分配物理页面。只有这样，在进入内核的trap()中在错误栈里放置相应信息才不会触发缺页中断（细节在下节马上就会说到）

所以这个栈是自己创建的。

弄清楚这些栈的切换和设置关系对于我们下面要实现的系统调用非常重要。

3.4 Invoking the User Page Fault Handler

Exercise 4. Implement the code in page_fault_handler in kern/trap.c required to dispatch page faults to the user-mode handler. Be sure to take appropriate precautions when writing into the exception stack. (What happens if the user environment runs out of space on the exception stack?)

`page_fault_handler()` 在lab3中我们已经完成了对内核态触发的页错误的处理（如果在内核发生页错误，那么直接panic，以免发生更大的问题），这次的任务是完成在用户态中发生的页错误的处理。

有两种可能会造成缺页中断：

1. 用户程序正常运行中访问到一个错误地址，触发页错误中断
2. 用户自己定义的页错误处理程序在运行时访问到错误地址，同样也会触发页错误中断

注意！ 有可能从第1种开始进入中断处理，然后2又触发2，又触发2...造成中断处理的**递归**。

简单的来说，在page_fault_handler()要做的事情，就是将触发页错误的进程信息（可能是正常的用户程序，也可能是用户定义的页错误处理程序）以UTrapframe的形式压入用户的错误栈，然后进入用户的页错误处理程序开始运行。（用户的页错误处理程序会负责回到原来触发错误的地方重新开始执行，在后面我们马上会涉及到）

在处理中有几个东西需要注意的：

运行栈的切换 ：

- 当正常用户进程执行出现页错误时，栈的切换轨迹是，**用户运行栈** $\longrightarrow$内核态系统栈（中断被操作系统捕捉到）$\longrightarrow$用户错误栈（开始错误处理）
- 当错误处理程序执行出现页错误时，栈的切换轨迹是，**用户错误栈** $\longrightarrow$内核态系统栈$\longrightarrow$用户错误栈

虽然错误处理程序在本质上仍是用户进程，但是它们的运行栈不同！具体细节稍候在代码中解释。

UTrapframe ：

UTrapframe放置在用户错误栈中，保存触发页错误中断的进程上下文信息，便于页错误处理程序结束后返回原程序继续运行。那么为什么要提出一个新结构而不用原来的Trapframe呢？我们来看看他们之间的区别：

inc/trap.h

```
56  struct Trapframe {
57          struct PushRegs tf_regs;
58          uint16_t tf_es;
59          uint16_t tf_padding1;
60          uint16_t tf_ds;
61          uint16_t tf_padding2;
62          uint32_t tf_trapno;
63          /* below here defined by x86 hardware */
64          uint32_t tf_err;
65          uintptr_t tf_eip;
66          uint16_t tf_cs;
67          uint16_t tf_padding3;
68          uint32_t tf_eflags;
69          /* below here only when crossing rings, such as from user to kernel
                */
70          uintptr_t tf_esp;
71          uint16_t tf_ss;
72          uint16_t tf_padding4;
73  } __attribute__((packed));
74
75  struct UTrapframe {
76          /* information about the fault */
77          uint32_t utf_fault_va;  /* va for T_PGFLT, 0 otherwise */
78          uint32_t utf_err;
79          /* trap-time return state */
80          struct PushRegs utf_regs;
```

```
81          uintptr_t utf_eip;
82          uint32_t utf_eflags;
83          /* the trap-time stack to return to */
84          uintptr_t utf_esp;
85  } __attribute__((packed));
```

通过结构可以观察到这么几个不同：

- UTrapframe是特别设计给**用户自定义的中断错误处理程序的（不一定是页错误中断处理程序）**的，用以保存错误发生之前的环境信息，所以UTrapframe有了一个特殊的成员fault_va，表示访问出错的指令涉及的具体地址，当然如果不是页错误处理程序，那么这个成员设置成0即可
- UTrapframe和Trapframe相比少了es, ds, ss等段寄存器信息。因为根据刚才提到栈切换的不同，无论是两种情况的哪种，都是从用户态$\longrightarrow$内核态$\longrightarrow$用户态，因为**两个用户态程序实际上是同一个用户进程**，所以我们从后面的中断错误处理程序切换到前面的触发程序就不会涉及到段的切换，自然也不需要保存它们了。

以上是结构上最大的两点不同，而**实际使用上**，Trapframe用于保存进程的完整信息，在中断后被中断程序自动保留下来，并且作为数据结构在操作系统内各个函数中传递，而UTrapframe只是为了有效的组织起触发错误进程的状态放置在错误栈里，以便恢复运行，并没有在操作系统的其他地方使用到。

好，上面已经说明了需要注意的事项，我们来看看具体的代码：

kern/trap.c: page_fault_handler()

```
1   void
2   page_fault_handler(struct Trapframe *tf)
3   {
4       uint32_t fault_va;
5
6       // Read processor's CR2 register to find the faulting address
7       fault_va = rcr2();
8
9       // Handle kernel-mode page faults.
10      // LAB 3: Your code here.
11
12      if ((tf->tf_cs & 3) == 0)
13          panic ("kernel-mode page faults");
14
15
16      // LAB 4: Your code here.
17      if (curenv->env_pgfault_upcall != NULL) {
18
19          struct UTrapframe *utf;
20
21          if (UXSTACKTOP - PGSIZE <= tf->tf_esp && tf->tf_esp < UXSTACKTOP)
22              utf = (struct UTrapframe *)
23                          (tf->tf_esp - sizeof (struct UTrapframe) - 4);
24          else
25              utf = (struct UTrapframe *)
26                          (UXSTACKTOP - sizeof (struct UTrapframe));
27
28          user_mem_assert (
```

```
29              curenv,
30              (void*) utf,
31              sizeof (struct UTrapframe),
32              PTE_U|PTE_W);
33
34          utf->utf_eflags = tf->tf_eflags;
35          utf->utf_eip = tf->tf_eip;
36          utf->utf_err = tf->tf_err;
37          utf->utf_esp = tf->tf_esp;
38          utf->utf_fault_va = fault_va;
39          utf->utf_regs = tf->tf_regs;
40
41          curenv->env_tf.tf_eip = (uint32_t) curenv->env_pgfault_upcall;
42          curenv->env_tf.tf_esp = (uint32_t) utf;
43          env_run (curenv);
44      }
45
46      // Destroy the environment that caused the fault.
47      cprintf("[%08x] user fault va %08x ip %08x\n",
48          curenv->env_id, fault_va, tf->tf_eip);
49      print_trapframe(tf);
50      env_destroy(curenv);
51  }
```

从上到下解释：

1. 第21行：判断发生页错误的原进程是否已经运行在用户错误栈上，如果是，则是中断递归，需要在错误栈中压入一个空字（用处在后面揭晓）和一个UTrapframe，否则的话，则是正常的用户进程发生页错误，则在错误栈的栈顶放入一个UTrapframe即可

2. 第28行：检查新需要的错误栈空间是否已经被按照用户可写的权限正确映射。还记得原来提到用户错误栈是什么时候被申请的么？是用户自己在运行之前注册中断错误处理程序时申请的！所以到了 `page_fault_handler()` 这里应该是已经按照映射好了的，但是因为错误栈大小被固定，所以有可能溢出，需要检查

3. 第34行：将发生错误的进程的运行时信息保存在UTrapframe中。还记得前面提到的Trapframe和UTrapframe在使用上的区别么？所以我们在这里可以看到是将tf中的信息放入utf

4. 第41行：准备切换到用户定义的中断错误处理程序开始运行，还记得前面我们提到过中断处理程序实际上和出错的用户进程是同一程序么？所以它们对应的环境是同一个！需要改变的只有栈顶位置和指令的入口而已，然后就可以直接使用`env_run()` 来跳转到错误处理程序了。

3.5 User-mode Page Fault Entrypoint

> **Exercise 5.** Implement the _pgfault_upcall routine in lib/pfentry.S. The interesting part is returning to the original point in the user code that caused the page fault. You'll return directly there, without going back through the kernel. The hard part is simultaneously switching stacks and re-loading the EIP.

_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。

这段代码非常的有技巧性，因为数据都存储在栈中，如何同时做到恢复EIP和ESP的呢？我是在参考了张磊同学的代码后才知道这么作的原理，所以这里直接结合代码来叙述：

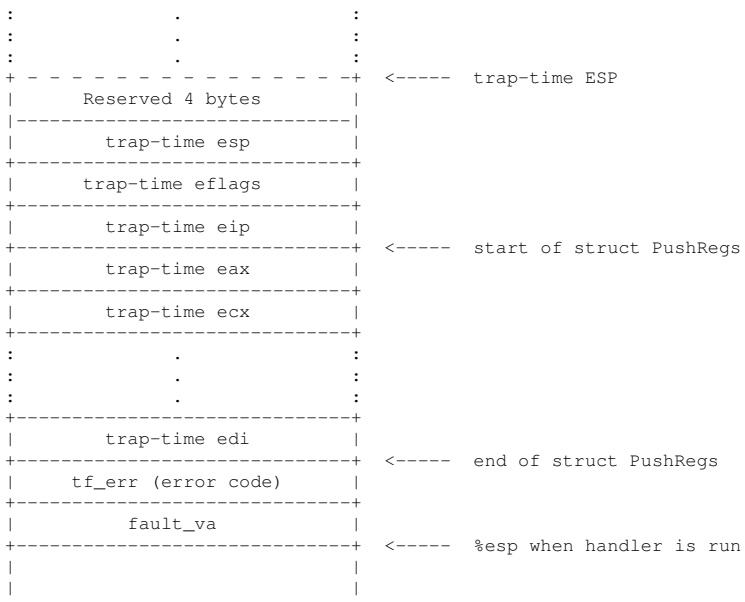
lib/pfentry.S

```
1  .text
2  .globl _pgfault_upcall
3  _pgfault_upcall:
4          # Call the C page fault handler.
5          pushl %esp                      # function argument: pointer to UTF
6          movl _pgfault_handler, %eax
7          call *%eax
8          addl $4, %esp                   # pop function argument
9
10
11         # Add by Chi Zhang (zhangchitc@gmail.com)
12         # subtract 4 from old esp for the storage of old eip(later use for return)
13         movl     0x30(%esp), %eax
14         subl     $0x4, %eax
15         movl     %eax, 0x30(%esp)
16
17
18         # put old eip in the pre-reserved 4 bytes space
19         movl     0x28(%esp), %ebx
20         movl     %ebx, (%eax)
21
22
23         # restore all general-purpose registers
24         addl     $0x8, %esp
25         popal
26
27
28         # Restore eflags from the stack.  After you do this, you can
29         # no longer use arithmetic operations or anything else that
30         # modifies eflags.
31         # LAB 4: Your code here.
32
33         addl     $0x4, %esp
34         popfl
35
36         # Switch back to the adjusted trap-time stack.
37         # LAB 4: Your code here.
38
39         pop     %esp
40
41         # Return to re-execute the instruction that faulted.
42         # LAB 4: Your code here.
43
44         ret
```

我们从第8行开始看起：

1. 这时已从用户自定义错误处理程序中返回，那么观察用户错误栈，应该是和调用前一样的形式：

```
+------------------------------+  <-----  UXSTACKTOP
|                              |
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```



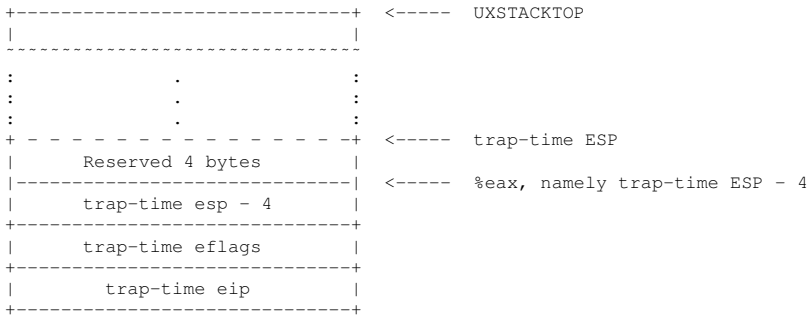
注意这里trap-time esp上的空间，我留出了一个4 bytes保留空间，为了说明这是一个中断递归的情形，有可能没有，那么就是用户进程直接出错了，这个不妨碍说明。

2. 第13行，将栈中的trap-time esp减去4，同时传递给GPR eax以便下面的使用:

lib/pfentry.S

```
11      movl    0x30(%esp), %eax
12      subl    $0x4, %eax
13      movl    %eax, 0x30(%esp)
```

当执行完以后栈情况如下图，当是中断递归的情况，如图，trap-time esp减去4就是Reserved 4 bytes的首地址；如果不是，那么trap-time esp则是原来用户运行栈的栈顶。

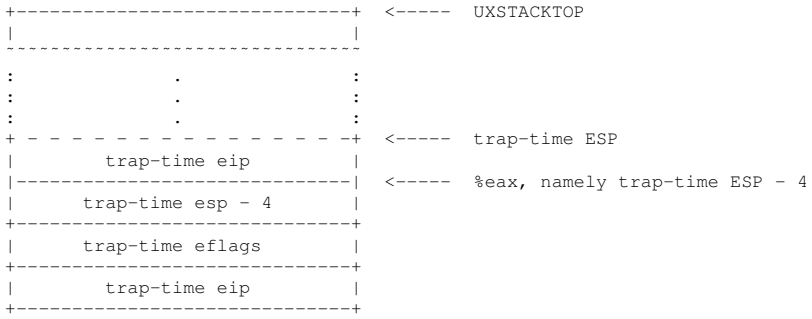


3. 第18行，将原出错程序的EIP（即trap-time eip)放入留出的4字节空白区域，以便**后来恢复运行**，这个地方是最巧妙的地方！也是留出这个4字节的意义所在。稍候我们便会看到它是如何神奇的使我们同时恢复ESP和EIP两个寄存器的。

lib/pfentry.S

```
18          movl    0x28(%esp), %ebx
19          movl    %ebx, (%eax)
```

执行以后栈布局变成：



4. 第24行，恢复所有通用寄存器。从这句话完成以后开始所有的通用寄存器就不能再使用了

lib/pfentry.S

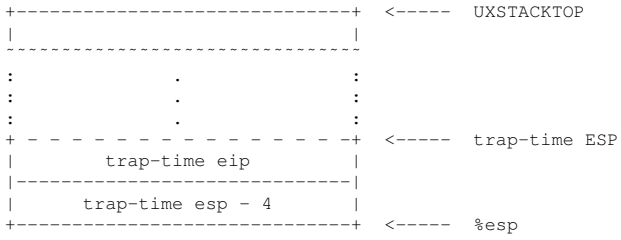
```
24          addl    $0x8, %esp
25          popal
```

5. 第33行，恢复EFLAGS标志寄存器，从这句话以后就不能使用会修改EFLAGS操作的指令了，比如算数指令add, sub或者mov和int等等：

lib/pfentry.S

```
33          addl    $0x4, %esp
34          popfl
```

执行完以后这时栈的指针和布局为：



```
|      trap-time eflags        |
+------------------------------+
|        trap-time eip         |
+------------------------------+
```

6. 第39行，切换回原来出错的程序运行栈：

lib/pfentry.S

```
39      pop     %esp
```

栈的情况如图：

```
+------------------------------+  <-----  UXSTACKTOP
|                              |
~------------------------------~
:              .               :
:              .               :
:              .               :
+ - - - - - - - - - - - - - - -+  <-----  trap-time ESP
|        trap-time eip         |
|------------------------------|  <-----  %esp
|      trap-time esp - 4       |
+------------------------------+
|      trap-time eflags        |
+------------------------------+
|        trap-time eip         |
+------------------------------+
```

这个情况看起来挺二的，我们用一个pop命令实现的仅仅是将栈指针上移4个bytes，其实这里有两点原因：

(a) 前面我们说过，所有算数指令都不能使用了，所以不能直接移动esp

(b) 上图仅仅是中断递归的情况，如果是用户程序直接出错的话，那么这下pop出的值就是**用户栈**的栈顶-4，那么就从错误栈切换到了用户栈

7. 最后，使用ret返回出错程序：

lib/pfentry.S

```
44      ret
```

在lab3中我们就知道，一条ret指令的意义相当于就是pop %eip，那么结合上面栈的布局图：

```
+------------------------------+  <-----  UXSTACKTOP
|                              |
~------------------------------~
:              .               :
:              .               :
:              .               :
+ - - - - - - - - - - - - - - -+  <-----  trap-time ESP
|        trap-time eip         |
|------------------------------|  <-----  %esp
|      trap-time esp - 4       |
```

```
+-----------------------------+
|      trap-time eflags       |
+-----------------------------+
|        trap-time eip        |
+-----------------------------+
```

那么执行完以后，eip被正确赋值成了trap-time eip，而esp也正好得到了trap-time esp！！

Exercise 6. Finish set_pgfault_handler() in lib/pgfault.c.

这个函数即我们在前面提到了很多次的，用户在运行前注册自己的页错误处理程序，里面做的最重要的事情就是申请用户错误栈空间：

lib/pgfault.c

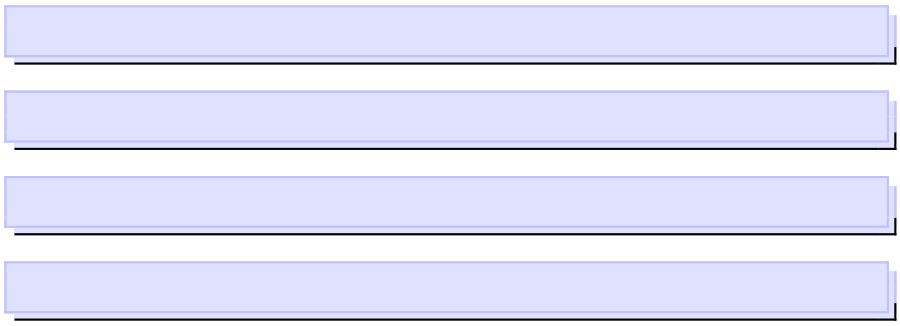
```
1  void
2  set_pgfault_handler(void (*handler)(struct UTrapframe *utf))
3  {
4      int r;
5
6      if (_pgfault_handler == 0) {
7          if ((r = sys_page_alloc (0, (void*) (UXSTACKTOP - PGSIZE), PTE_U|PTE_P|
                PTE_W)) < 0)
8              panic ("set_pgfault_handler:␣%e", r);
9
10         sys_env_set_pgfault_upcall (0, _pgfault_upcall);
11     }
12
13     // Save handler pointer for assembly to call.
14     _pgfault_handler = handler;
15 }
```

当然记得在kern/syscall.c中添加对应的系统调用号分配程序，这样我们的代码就全部完成了！！

3.6 Testing

测试全部顺利通过！

3.7 Implementing Copy-on-Write Fork



4 Preemptive Multitasking and Inter-Process communication (IPC)

inc/mmu.h

kern/trapentry.S