操作系统JOS实习第四次报告

张弛 00848231,
zhangchitc@gmail.com

May 10, 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。

Challenge! Add a less trivial scheduling policy to the kernel, such as a fixed-priority scheduler that allows each environment to be assigned a priority and ensures that higher-priority environments are always chosen in preference to lower-priority environments. If you're feeling really adventurous, try implementing a Unix-style adjustable-priority scheduler or even a lottery or stride scheduler. (Look up "lottery scheduling" and " stride scheduling" in Google.)

Write a test program or two that verifies that your scheduling algorithm is working correctly (i.e., the right environments get run in the right order). It may be easier to write these test programs once you have implemented fork() and IPC in parts B and C of this lab.

这个部分我实现的机制是使用简单的优先级调度算法，相同优先级下使用Round robin。具体有几方面的工作：

1. 首先给Env结构添加一个新成员env_priority
2. 相应在`env_alloc()`中给它赋值为默认优先级ENV_PRIOR_DEFAULT，一共设立了四个基本的优先级：

inc/env.h

```
1 #define ENV_PRIOR_SUPREME   0
2 #define ENV_PRIOR_HIGH      10
3 #define ENV_PRIOR_DEFAULT   100
4 #define ENV_PRIOR_LOW       1000
```

   数字越小优先级越高

3. 修改kern/sched.c的调度策略
4. 添加一个系统调用`sys_env_set_priority (envid, priority)`，允许进程的父亲或者自己修改自己的优先级，具体对优先级的数值没有限制，可调高也可调低。这部分工作很多，添加一个系统调用需要修改若干个文件：
   - inc/syscall.h
   - lib/syscall.c
   - lib/lib.h

- kern/syscall.c

然后我使用了以下的测试程序：

user/priorsched.c

```
#include <inc/lib.h>

void
umain(void)
{
    struct Env *chenv;
    int ch;

    sys_env_set_priority (0, ENV_PRIOR_SUPREME);

    if ((ch = fork ()) != 0) {
        sys_env_set_priority (ch, ENV_PRIOR_LOW);

        if ((ch = fork ()) != 0) {
            sys_env_set_priority (ch, ENV_PRIOR_HIGH);

            if ((ch = fork ()) != 0)
                sys_env_set_priority (ch, ENV_PRIOR_DEFAULT);
        }
    }

    chenv = (struct Env *) envs + ENVX (sys_getenvid ());

    int i;
    for (i = 0; i < 2; i ++) {
        cprintf("%04x: I am running the %dth time with priority = ", sys_getenvid
            (), i + 1);
        if (chenv->env_priority == ENV_PRIOR_LOW) cprintf ("low\n"); else
        if (chenv->env_priority == ENV_PRIOR_DEFAULT) cprintf ("default\n"); else
        if (chenv->env_priority == ENV_PRIOR_HIGH) cprintf ("high\n"); else
        if (chenv->env_priority == ENV_PRIOR_SUPREME) cprintf ("supreme\n");
        sys_yield ();
    }
}
```

大致意思是父进程拥有最高优先级，然后创建三个不同权限的子进程，先创建了最低优先级，然后是HIGH和DEFAULT，运行以后输出：

```
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
[00001001] new env 00001003
[00001001] new env 00001004
1001: I am running the 1th time with priority = supreme
1001: I am running the 2th time with priority = supreme
[00001001] exiting gracefully
[00001001] free env 00001001
1003: I am running the 1th time with priority = high
```

```
1003: I am running the 2th time with priority = high
[00001003] exiting gracefully
[00001003] free env 00001003
1004: I am running the 1th time with priority = default
1004: I am running the 2th time with priority = default
[00001004] exiting gracefully
[00001004] free env 00001004
1002: I am running the 1th time with priority = low
1002: I am running the 2th time with priority = low
[00001002] exiting gracefully
[00001002] free env 00001002
Welcome to the JOS kernel monitor!
```

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

3.1.1 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()

```
1  static int
2  sys_env_set_pgfault_upcall(envid_t envid, void *func)
3  {
4      struct Env *e;
5      if (envid2env (envid, &e, 1) < 0)
6          return -E_BAD_ENV;
7
8      e->env_pgfault_upcall = func;
9
10     return 0;
11 }
```

3.1.2 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

```
1  /*
2   * Virtual memory map:                                Permissions
3   *                                                    kernel/user
4   *
5   *    4 Gig --------->  +------------------------------+
6   *                      |                              | RW/--
7   *                      ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
8   *                      :              .               :
9   *                      :              .               :
10  *                      :              .               :
11  *                      |~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~| RW/--
12  *                      |                              | RW/--
13  *                      |   Remapped Physical Memory   | RW/--
14  *                      |                              | RW/--
15  *    KERNBASE ------>  +------------------------------+ 0xf0000000
16  *                      |  Cur. Page Table (Kern. RW)  | RW/--  PTSIZE
17  *    VPT,KSTACKTOP--> +------------------------------+ 0xefc00000      --+
18  *                      |         Kernel Stack         | RW/--  KSTKSIZE   |
19  *                      | - - - - - - - - - - - - - - -|                 PTSIZE
20  *                      |      Invalid Memory (*)      | --/--             |
21  *    ULIM     ------> +------------------------------+ 0xef800000      --+
22  *                      |  Cur. Page Table (User R-)   | R-/R-  PTSIZE
23  *    UVPT      ----->  +------------------------------+ 0xef400000
24  *                      |          RO PAGES            | R-/R-  PTSIZE
25  *    UPAGES    ----->  +------------------------------+ 0xef000000
26  *                      |           RO ENVS            | R-/R-  PTSIZE
27  * UTOP,UENVS ------>  +------------------------------+ 0xeec00000
28  * UXSTACKTOP -/       |     User Exception Stack     | RW/RW  PGSIZE
29  *                     +------------------------------+ 0xeebff000
30  *                      |       Empty Memory (*)       | --/--  PGSIZE
31  *    USTACKTOP  --->  +------------------------------+ 0xeebfe000
32  *                      |      Normal User Stack       | RW/RW  PGSIZE
```

```
33  *                    +------------------------------+ 0xeebfd000
34  *                    |                              |
35  *                    |                              |
36  *                    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
37  *                    .                              .
38  *                    .                              .
39  *                    .                              .
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。

3.1.3 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。

3.1.4 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。

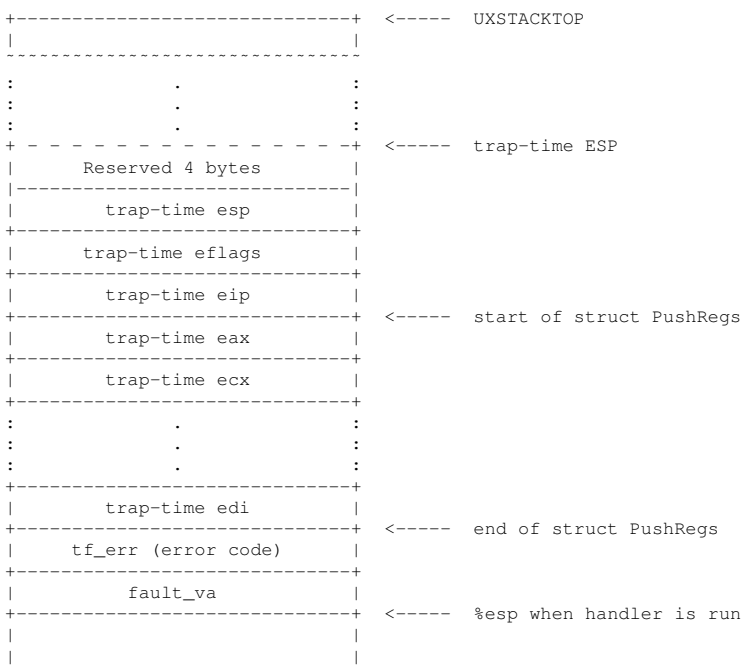
这段代码非常的有技巧性，因为数据都存储在栈中，如何同时做到恢复EIP和ESP的呢？我是在参考了张磊同学的代码后才知道这么作的原理，所以这里直接结合代码来叙述：

lib/pfentry.S

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


        # Add by Chi Zhang (zhangchitc@gmail.com)
        # subtract 4 from old esp for the storage of old eip(later use for return)
        movl    0x30(%esp), %eax
        subl    $0x4, %eax
        movl    %eax, 0x30(%esp)


        # put old eip in the pre-reserved 4 bytes space
        movl    0x28(%esp), %ebx
        movl    %ebx, (%eax)


        # restore all general-purpose registers
        addl    $0x8, %esp
        popal


        # Restore eflags from the stack.  After you do this, you can
        # no longer use arithmetic operations or anything else that
        # modifies eflags.
        # LAB 4: Your code here.

        addl    $0x4, %esp
        popfl

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

        pop     %esp

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

        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：



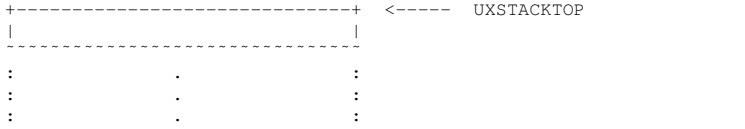
注意这里trap-time esp上的空间，我留出了一个4 bytes保留空间，为了说明这是一个中断递归的情形，有可能没有，那么就是用户进程直接出错了，这个不妨碍说明。

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。



```
+ - - - - - - - - - - - - - - - - +  <-----  trap-time ESP
|      Reserved 4 bytes           |
|-------------------------------- |  <-----  %eax, namely trap-time ESP - 4
|      trap-time esp - 4          |
+---------------------------------+
|      trap-time eflags           |
+---------------------------------+
|        trap-time eip            |
+---------------------------------+
```

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：

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

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

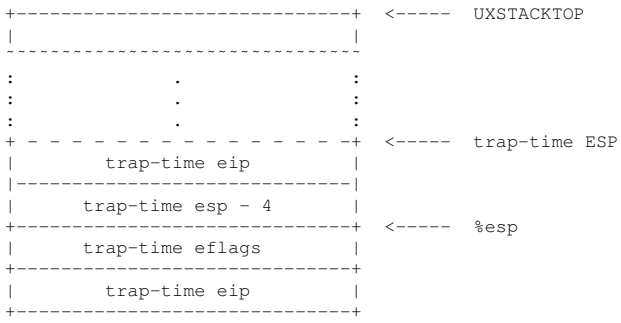
```
24        addl    $0x8, %esp
25        popal
```

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：

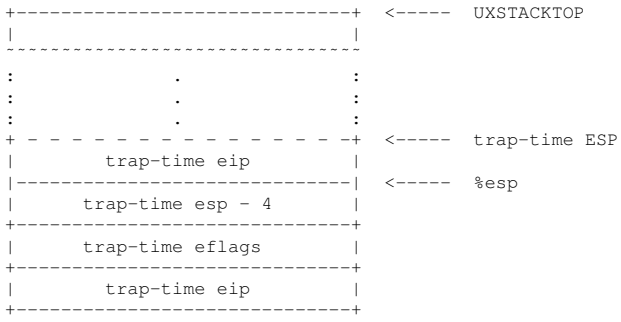


6. 第39行，切换回原来出错的程序运行栈：

lib/pfentry.S

```
39      pop     %esp
```

栈的情况如图：



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

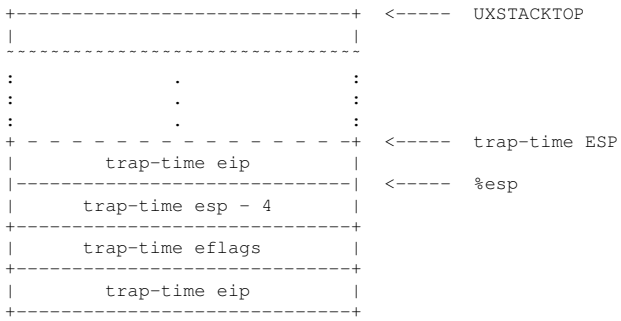
(b) 上图仅仅是中断递归的情况，如果是用户程序直接出错的话，那么这下pop出的值就是**用户栈**的栈顶-4，那么就从错误栈切换到了用户栈

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：



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！！

3.1.5 Testing

测试全部顺利通过！

3.2 Implementing Copy-on-Write Fork

这一部分的MIT材料非常的重要，提到了很多实现上的细节。

```
Exercise 7. Implement fork, duppage and pgfault in lib/fork.c.

Test your code with the forktree program. It should produce the following
messages, with interspersed 'new env', 'free env', and 'exiting gracefully
' messages. The messages may not appear in this order, and the environment
 IDs may be different.

        1001: I am ''
        1802: I am '0'
        2801: I am '00'
        3802: I am '000'
        2003: I am '1'
        5001: I am '11'
        4802: I am '10'
        6801: I am '100'
        5803: I am '110'
        3004: I am '01'
        8001: I am '011'
        7803: I am '010'
        4005: I am '001'
        6006: I am '111'
        7007: I am '101'
```

首先来看看页错误处理程序`pgfault()`，这个函数在材料中提到了需要找到用户程序访问的地址对应页表项来查看它的权限，用户程序要如何访问页目录和页表呢？这就涉及到lab2中的细节了。

请阅读我lab2报告中3.2 Reference counting一节关于[UVPT, ULIM)这段内存区域的详细说明，里面讲述的给定一个虚拟地址，如何构造新的虚拟地址来得到它对应的**页目录表项**和**页表表项**的具体方法。这里稍微总结一下：

假设需要查询的虚拟地址为addr = PDX|PTX|OFFSET的话

- 如果要得到对应页目录表项：令vaddr = UVPT[31:22]|UVPT[31:22]|PDX|00
- 如果要得到对应页表表项：令vaddr = UVPT[31:22]|PDX|PTX|00

那么利用vaddr查询到的值就是相应所需要的数据。在具体代码中，有两处地方使用到了这样的技巧，首先是定义这两处地址，在lib/entry.S中，**每个用户程序都会和此文件进行链接**：

lib/entry.S

```
1 #include <inc/mmu.h>
2 #include <inc/memlayout.h>
3
4 .data
5
6
7 // Define the global symbols 'envs', 'pages', 'vpt', and 'vpd'
8 // so that they can be used in C as if they were ordinary global arrays.
9         .globl envs
```

```
10          .set envs, UENVS
11          .globl pages
12          .set pages, UPAGES
13          .globl vpt
14          .set vpt, UVPT
15          .globl vpd
16          .set vpd, (UVPT+(UVPT>>12)*4)
```

可以看到其中定义的两个变量：

- vpt: 指向UVPT
- vpd: $(\text{UVPT} + (\text{UVPT} >> 12) * 4)$，实际上就是把UVPT的前10位复制到了第11到20位，正好对应上面提到页目录表项地址的构造方法。

后面我们看到在inc/memlayout.h中是怎么描述这两个变量的：

inc/memlayout.h

```
137 /*
138  * The page directory entry corresponding to the virtual address range
139  * [VPT, VPT + PTSIZE) points to the page directory itself.  Thus, the page
140  * directory is treated as a page table as well as a page directory.
141  *
142  * One result of treating the page directory as a page table is that all PTEs
143  * can be accessed through a "virtual page table" at virtual address VPT (to
144  * which vpt is set in entry.S).  The PTE for page number N is stored in
145  * vpt[N].  (It's worth drawing a diagram of this!)
146  *
147  * A second consequence is that the contents of the current page directory
148  * will always be available at virtual address (VPT + (VPT >> PGSHIFT)), to
149  * which vpd is set in entry.S.
150  */
151 typedef uint32_t pte_t;
152 typedef uint32_t pde_t;
153
154 extern volatile pte_t vpt[];     // VA of "virtual page table"
155 extern volatile pde_t vpd[];     // VA of current page directory
```

里面提到，假设需要查询的虚拟地址为va，则

- 其对应的页目录表项为：vpd[VPD(va)]，其中VPD等价于PDX宏
- 其对应的页表表项为：vpt[VPN(va)]，其中VPN等价于PPN宏

因为用数组base[offset]的方式访问内存其实就是访问base+offset×sizeof(base)，例如上面的页表表项，

$$\begin{aligned} vpt[VPN(va)] &= vpt + VPN(va) << 2 \\ &= vpt + (va >> 12) << 2 \\ &= vpt + (va >> 10) \\ &= UVPT + (va >> 10) \end{aligned}$$

即我们前面说到的页表表项地址构造方法vaddr = UVPT[31:22]|PDX|PTX|00！页目录的方法也类似。

好了有了这个的铺垫，我们可以开始看看实际的代码了：

lib/fork.c: pgfault()

```
1  static void
2  pgfault(struct UTrapframe *utf)
3  {
4      void *addr = (void *) utf->utf_fault_va;
5      uint32_t err = utf->utf_err;
6      int r;
7
8      if ((err & FEC_WR) == 0 || (vpd[VPD(addr)] & PTE_P) == 0 ||  (vpt[VPN(addr)] &
           PTE_COW) == 0)
9          panic ("pgfault: not a write or attempting to access a non-COW page");
10
11
12     if ((r = sys_page_alloc (0, (void *)PFTEMP, PTE_U|PTE_P|PTE_W)) < 0)
13         panic ("pgfault: page allocation failed : %e", r);
14
15     addr = ROUNDDOWN (addr, PGSIZE);
16
17     memmove (PFTEMP, addr, PGSIZE);
18
19     if ((r = sys_page_map (0, PFTEMP, 0, addr, PTE_U|PTE_P|PTE_W)) < 0)
20         panic ("pgfault: page mapping failed : %e", r);
21 }
```

里面主要有两点需要注意的：

1. PTE_COW是x86页面机制中在页表表项里留出的几个特殊位供程序员自己定制的位，一共三位，区域为AVAIL，可以参考x86手册获得更详细的资料
2. 第15行，在分配了新的物理页以后准备开始从原页面开始拷贝数据时，记得将地址和物理页大小对齐

接下来看duppage()，这个函数负责进行COW方式的页复制：

lib/fork.c: duppage()

```
1  static int
2  duppage(envid_t envid, unsigned pn)
3  {
4          int r;
5
6      void * addr = (void *) ((uint32_t) pn * PGSIZE);
7      pte_t pte = vpt[VPN(addr)];
8
9      if ((pte & PTE_W) > 0 || (pte & PTE_COW) > 0) {
10         if ((r = sys_page_map (0, addr, envid, addr, PTE_U|PTE_P|PTE_COW)) < 0)
11             panic ("duppage: page re-mapping failed at 1 : %e", r);
12
13         if ((r = sys_page_map (0, addr, 0, addr, PTE_U|PTE_P|PTE_COW)) < 0)
```

```
14              panic ("duppage: page re-mapping failed at 2 : %e", r);
15
16      } else {
17          if ((r = sys_page_map (0, addr, envid, addr, PTE_U|PTE_P)) < 0)
18              panic ("duppage: page re-mapping failed at 3 : %e", r);
19      }
20
21      return 0;
22  }
```

唯一要注意的就是它传入的参数是物理页号，所以要作相应的地址运算。

最后来看fork函数：

lib/fork.c: fork()

```
1   envid_t
2   fork(void)
3   {
4       set_pgfault_handler (pgfault);
5
6       envid_t envid;
7       uint32_t addr;
8       int r;
9
10      envid = sys_exofork();
11
12      if (envid < 0)
13          panic("sys_exofork: %e", envid);
14
15      // We're the child
16      if (envid == 0) {
17          env = &envs[ENVX(sys_getenvid())];
18          return 0;
19      }
20
21          // We're the parent.
22      for (addr = UTEXT; addr < UXSTACKTOP - PGSIZE; addr += PGSIZE) {
23          if ((vpd[VPD(addr)] & PTE_P) > 0 && (vpt[VPN(addr)] & PTE_P) > 0 && (vpt[
                VPN(addr)] & PTE_U) > 0)
24              duppage (envid, VPN(addr));
25      }
26
27      if ((r = sys_page_alloc (envid, (void *)(UXSTACKTOP - PGSIZE), PTE_U|PTE_W|
             PTE_P)) < 0)
28          panic ("fork: page allocation failed : %e", r);
29
30
31      extern void _pgfault_upcall (void);
32
33      sys_env_set_pgfault_upcall (envid, _pgfault_upcall);
34
35      // Start the child environment running
36      if ((r = sys_env_set_status(envid, ENV_RUNNABLE)) < 0)
37          panic("fork: set child env status failed : %e", r);
38
39      return envid;
40  }
```

这里首先需要搞清楚关于页错误处理程序的控制流，请认真阅读绍老师讲义

中的的相关章节，理解下图：

图 6-2.页错误处理流程

然后从上到下梳理一下代码：

1. 第4行，注意这里调用的是 `set_pgfault_handler()` 而不是系统调用 `sys_env_set_pgfault_upcall()` ，**因为前者会检查用户程序的错误栈是否存在**，如果没有则分配相应空间。我们无法得知父进程在调用fork前是否已经为错误栈分配了空间，所以不能直接调用后者
2. 第22行，拷贝从0到UXSTACKTOP之间的所有用户页面。因为我们在创建用户env时，已经在`env_setup_vm()` 中将UTOP（即UXSTACKTOP）之上的所有页面都映射到和内核一样，但是不要映射错误栈的空间，因为我们马上要给它创建新物理页。
3. 第33行，为子进程设置页错误处理程序。因为使用`env_alloc()`创建的env的处理程序指针都为空，但是这时我们已经明确的为其错误栈分配了物理页面，所以可以直接使用系统调用指定错误处理的入口了，_pgfault_upcall为所有用户页错误处理程序的总入口。

至此使用Copy on Write机制的fork函数就完成了，运行用户程序forktree能争取打印出Exercise要求的结果。任务完成！

> **Challenge!** Implement a shared-memory fork() called sfork(). This version should have the parent and child share all their memory pages (so writes in one environment appear in the other) except for pages in the stack area , which should be treated in the usual copy-on-write manner. Modify user/ forktree.c to use sfork() instead of regular fork(). Also, once you have finished implementing IPC in part C, use your sfork() to run user/ pingpongs. You will have to find a new way to provide the functionality of the global env pointer.

这里面主要修改的就是对于页面的映射问题，即如果在父进程中是栈页面，则映射为COW，否则不COW. 那么主要就是修改duppage ()，我写了一个新的页映射函数，增加了参数need_cow，表示是否是需要强制设置为COW:

lib/fork.c: sduppage()

```
1  static int
2  sduppage(envid_t envid, unsigned pn, int need_cow)
3  {
4      int r;
5
6      void * addr = (void *) ((uint32_t) pn * PGSIZE);
7      pte_t pte = vpt[VPN(addr)];
8
9      if (need_cow || (pte & PTE_COW) > 0) {
10         if ((r = sys_page_map (0, addr, envid, addr, PTE_U|PTE_P|PTE_COW)) < 0)
11             panic ("duppage: page re-mapping failed at 1 : %e", r);
12
13         if ((r = sys_page_map (0, addr, 0, addr, PTE_U|PTE_P|PTE_COW)) < 0)
14             panic ("duppage: page re-mapping failed at 2 : %e", r);
15
16     } else
17     if ((pte & PTE_W) > 0) {
18         if ((r = sys_page_map (0, addr, envid, addr, PTE_U|PTE_W|PTE_P)) < 0)
19             panic ("duppage: page re-mapping failed at 3 : %e", r);
20     } else {
21         if ((r = sys_page_map (0, addr, envid, addr, PTE_U|PTE_P)) < 0)
22             panic ("duppage: page re-mapping failed at 4 : %e", r);
23     }
24
25     return 0;
26 }
```

然后sfork() 的大部分代码都和fork()一样，只有在复制页表时，从用户栈从上往下添加，记录栈空间，一直连续存在的页就是栈页面，否则就不是了：

lib/fork.c: sfork()

```
1  // Challenge!
2  int
3  sfork(void)
4  {
5      // LAB 4: Your code here.
6
7      set_pgfault_handler (pgfault);
8
9      envid_t envid;
10     uint32_t addr;
11     int r;
12
```

```
13      envid = sys_exofork();
14
15      if (envid < 0)
16          panic("sys_exofork: %e", envid);
17
18      // We're the child
19      if (envid == 0) {
20          env = &envs[ENVX(sys_getenvid())];
21          return 0;
22      }
23
24          // We're the parent.
25      int stackarea = 1;
26      for (addr = USTACKTOP - PGSIZE; addr >= UTEXT; addr -= PGSIZE) {
27          if ((vpd[VPD(addr)] & PTE_P) > 0 && (vpt[VPN(addr)] & PTE_P) > 0 && (vpt[
                VPN(addr)] & PTE_U) > 0)
28              sduppage (envid, VPN(addr), stackarea);
29          else
30              stackarea = 0;
31      }
32
33      if ((r = sys_page_alloc (envid, (void *)(UXSTACKTOP - PGSIZE), PTE_U|PTE_W|
            PTE_P)) < 0)
34          panic ("fork: page allocation failed : %e", r);
35
36
37      extern void _pgfault_upcall (void);
38
39      sys_env_set_pgfault_upcall (envid, _pgfault_upcall);
40
41      // Start the child environment running
42      if ((r = sys_env_set_status(envid, ENV_RUNNABLE)) < 0)
43          panic("fork: set child env status failed : %e", r);
44
45      return envid;
46  }
```

注意sfork() 中只有除开用户栈之外的页面才是共享的，比如全局变量！所以Challenge才会提到需要重新考虑env的使用。因为env定义在lib/lib.h中，被所有进程共用，这样的话在一个进程里修改了以后在另一个里取值就会发生问题。

我没有考虑这个问题，只是单纯的验证了我的sfork()是否正确，我使用了以下程序：

user/sfork.c

```
1   #include <inc/lib.h>
2
3   int share = 1;
4
5   void
6   umain(void)
7   {
8       int ch = sfork ();
9       if (ch != 0) {
10          cprintf ("I'm parent with share num = %d\n", share);
11          share = 2;
12      } else {
13          cprintf ("I'm child with share num = %d\n", share);
14      }
15  }
```

如果使用fork，那么父进程里修改share变量肯定不影响子进程，但是如果使用sfork的话，那么运行的输出为：

```
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
I'm parent with share num = 1
[00001001] exiting gracefully
[00001001] free env 00001001
I'm child with share num = 2
[00001002] exiting gracefully
[00001002] free env 00001002
Welcome to the JOS kernel monitor!
```

可以看到父亲的修改在子进程了马上得到了反应。

4 Preemptive Multitasking and Inter-Process communication (IPC)

4.1 Clock Interrupts and Preemption

4.1.1 Interrupt discipline

Exercise 8. Modify kern/trapentry.S and kern/trap.c to initialize the appropriate entries in the IDT and provide handlers for IRQs 0 through 15. Then modify the code in env_alloc() in kern/env.c to ensure that user environments are always run with interrupts enabled.

The processor never pushes an error code or checks the Descriptor Privilege Level (DPL) of the IDT entry when invoking a hardware interrupt handler. You might want to re-read section 9.2 of the 80386 Reference Manual, or section 5.8 of the IA-32 Intel Architecture Software Developer's Manual, Volume 3, at this time.

After doing this exercise, if you run your kernel with any test program that runs for a non-trivial length of time (e.g., spin), you should see the kernel print trap frames for hardware interrupts. While interrupts are now enabled in the processor, JOS isn't yet handling them, so you should see it misattribute each interrupt to the currently running user environment and destroy it. Eventually it should run out of environments to destroy and drop into the monitor.

这个部分模仿原来设置默认中断向量即可，先在kern/trapentry.S中定义相关的处理例程：

kern/trapentry.S

```
1  TRAPHANDLER_NOEC(routine_mchk, T_MCHK)
2  TRAPHANDLER_NOEC(routine_simderr, T_SIMDERR)
3
4  TRAPHANDLER_NOEC(routine_syscall, T_SYSCALL)
```

```
5
6
7       # for IRQ handler
8       TRAPHANDLER_NOEC(routine_irq0, IRQ_OFFSET + 0);
9       TRAPHANDLER_NOEC(routine_irq1, IRQ_OFFSET + 1);
10      TRAPHANDLER_NOEC(routine_irq2, IRQ_OFFSET + 2);
11      TRAPHANDLER_NOEC(routine_irq3, IRQ_OFFSET + 3);
12      TRAPHANDLER_NOEC(routine_irq4, IRQ_OFFSET + 4);
13      TRAPHANDLER_NOEC(routine_irq5, IRQ_OFFSET + 5);
14      TRAPHANDLER_NOEC(routine_irq6, IRQ_OFFSET + 6);
15      TRAPHANDLER_NOEC(routine_irq7, IRQ_OFFSET + 7);
16      TRAPHANDLER_NOEC(routine_irq8, IRQ_OFFSET + 8);
17      TRAPHANDLER_NOEC(routine_irq9, IRQ_OFFSET + 9);
18      TRAPHANDLER_NOEC(routine_irq10, IRQ_OFFSET + 10);
19      TRAPHANDLER_NOEC(routine_irq11, IRQ_OFFSET + 11);
20      TRAPHANDLER_NOEC(routine_irq12, IRQ_OFFSET + 12);
21      TRAPHANDLER_NOEC(routine_irq13, IRQ_OFFSET + 13);
22      TRAPHANDLER_NOEC(routine_irq14, IRQ_OFFSET + 14);
23      TRAPHANDLER_NOEC(routine_irq15, IRQ_OFFSET + 15);
24
25  /*
26   * Lab 3: Your code here for _alltraps
27   */
28  _alltraps:
```

然后在IDT里注册，修改`idt_init()`：

kern/trap.c: idt_init()

```
1       // set IDT for irq handler
2       extern void routine_irq0 ();
3       extern void routine_irq1 ();
4       extern void routine_irq2 ();
5       extern void routine_irq3 ();
6       extern void routine_irq4 ();
7       extern void routine_irq5 ();
8       extern void routine_irq6 ();
9       extern void routine_irq7 ();
10      extern void routine_irq8 ();
11      extern void routine_irq9 ();
12      extern void routine_irq10 ();
13      extern void routine_irq11 ();
14      extern void routine_irq12 ();
15      extern void routine_irq13 ();
16      extern void routine_irq14 ();
17      extern void routine_irq15 ();
18
19
20      SETGATE (idt[IRQ_OFFSET + 0], 0, GD_KT, routine_irq0, 0);
21      SETGATE (idt[IRQ_OFFSET + 1], 0, GD_KT, routine_irq1, 0);
22      SETGATE (idt[IRQ_OFFSET + 2], 0, GD_KT, routine_irq2, 0);
23      SETGATE (idt[IRQ_OFFSET + 3], 0, GD_KT, routine_irq3, 0);
24      SETGATE (idt[IRQ_OFFSET + 4], 0, GD_KT, routine_irq4, 0);
25      SETGATE (idt[IRQ_OFFSET + 5], 0, GD_KT, routine_irq5, 0);
26      SETGATE (idt[IRQ_OFFSET + 6], 0, GD_KT, routine_irq6, 0);
27      SETGATE (idt[IRQ_OFFSET + 7], 0, GD_KT, routine_irq7, 0);
28      SETGATE (idt[IRQ_OFFSET + 8], 0, GD_KT, routine_irq8, 0);
29      SETGATE (idt[IRQ_OFFSET + 9], 0, GD_KT, routine_irq9, 0);
30      SETGATE (idt[IRQ_OFFSET + 10], 0, GD_KT, routine_irq10, 0);
31      SETGATE (idt[IRQ_OFFSET + 11], 0, GD_KT, routine_irq11, 0);
32      SETGATE (idt[IRQ_OFFSET + 12], 0, GD_KT, routine_irq12, 0);
33      SETGATE (idt[IRQ_OFFSET + 13], 0, GD_KT, routine_irq13, 0);
34      SETGATE (idt[IRQ_OFFSET + 14], 0, GD_KT, routine_irq14, 0);
35      SETGATE (idt[IRQ_OFFSET + 15], 0, GD_KT, routine_irq15, 0);
36
37
```

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

创建用户进程时，只需要在其EFLAGS寄存器中打开IF位即可，这样等到运行时就会自动开启外部中断响应了：

kern/env.c: env_alloc()

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

运行spin应该得到下面的输出了：

```
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
I am the parent.  Forking the child...
[00001001] new env 00001002
TRAP frame at 0xf01f607c
  edi  0xef400000
  esi  0xef7bd000
  ebp  0xeebfdfb0
  oesp 0xefbfffdc
  ebx  0x9a7ed000
  edx  0x00000000
  ecx  0x00802000
  eax  0x00000000
  es   0x----0023
  ds   0x----0023
  trap 0x00000020 Hardware Interrupt
  err  0x00000000
  eip  0x00801315
  cs   0x----001b
  flag 0x00000246
  esp  0xeebfdf68
  ss   0x----0023
[00001001] free env 00001001
Welcome to the JOS kernel monitor!
```

可以看到打开外部中断响应后Hardware Interrupt被成功捕获

4.1.2 Handling Clock Interrupts

Exercise 9. Modify the kernel's trap_dispatch() function so that it calls sched_yield() to find and run a different environment whenever a clock interrupt takes place.

```
You should now be able to get the user/spin test to work: the parent
environment should fork off the child, sys_yield() to it a couple times
but in each case regain control of the CPU after one time slice, and
finally kill the child environment and terminate gracefully.
```

按照提示处理时钟中断：

kern/trap.c: trap_dispatch()

```
1  static void
2  trap_dispatch(struct Trapframe *tf)
3  {
4          // Handle clock interrupts.
5          // LAB 4: Your code here.
6          if (tf->tf_trapno == IRQ_OFFSET + IRQ_TIMER)
7              sched_yield ();
```

这时运行spin应该就看到子进程交出权限然后被父进程销毁：

```
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
I am the parent.  Forking the child...
[00001001] new env 00001002
I am the parent.  Running the child...
I am the child.  Spinning...
I am the parent.  Killing the child...
[00001001] destroying 00001002
[00001001] free env 00001002
[00001001] exiting gracefully
[00001001] free env 00001001
Welcome to the JOS kernel monitor!
```

运行make grade应该可以得到pingpong以前的所有分数了。

4.2 Inter-Process communication (IPC)

4.2.1 IPC in JOS

4.2.2 Sending and Receiving Messages

4.2.3 Transferring Pages

在读懂JOS中实现IPC的机制以后，我们要来看看实现这个机制需要修改Env的地方：

inc/env.h

```
38 struct Env {
39         struct Trapframe env_tf;        // Saved registers
40         LIST_ENTRY(Env) env_link;       // Free list link pointers
41         envid_t env_id;                 // Unique environment identifier
```

```
42          envid_t env_parent_id;          // env_id of this env's parent
43          unsigned env_status;            // Status of the environment
44          uint32_t env_runs;              // Number of times environment has run
45
46          // Address space
47          pde_t *env_pgdir;               // Kernel virtual address of page dir
48          physaddr_t env_cr3;             // Physical address of page dir
49
50          // Exception handling
51          void *env_pgfault_upcall;       // page fault upcall entry point
52
53          // Lab 4 IPC
54          bool env_ipc_recving;           // env is blocked receiving
55          void *env_ipc_dstva;            // va at which to map received page
56          uint32_t env_ipc_value;         // data value sent to us
57          envid_t env_ipc_from;           // envid of the sender
58          int env_ipc_perm;               // perm of page mapping received
59  };
```

其中增加了5个成员：

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

env_ipc_dstva ：
如果进程要接受消息，并且是传送页，则该地址$\leq$ UTOP

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

env_ipc_from ：
发送方负责设置该成员为自己的envid号

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.

4.2.4 Implementing IPC

> **Exercise 10.** Implement sys_ipc_recv and sys_ipc_try_send in kern/syscall.c . Read the comments on both before implementing them, since they have to work together. When you call envid2env in these routines, you should set the checkperm flag to 0, meaning that any environment is allowed to send IPC messages to any other environment, and the kernel does no special permission checking other than verifying that the target envid is valid.
>
> Then implement the ipc_recv and ipc_send functions in lib/ipc.c.
>
> Use the user/pingpong and user/primes functions to test your IPC mechanism . You might find it interesting to read user/primes.c to see all the forking and IPC going on behind the scenes.

这一部分的代码很好写，因为相对注释都非常完善，按照一步一步来就行了，首先是两个系统调用：

kern/syscall.c

```
1  static int
2  sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned perm)
3  {
4      struct Env *dstenv;
5      int r;
6
7
8      // target env does not exist
9      if ((r = envid2env (envid, &dstenv, 0)) < 0)
10         return -E_BAD_ENV;
11
12
13     // target env is not blocked or message has been sent
14     if (!dstenv->env_ipc_recving || dstenv->env_ipc_from != 0)
15         return -E_IPC_NOT_RECV;
16
17     // srcva < UTOP but not page aligned
18     if (srcva < (void *) UTOP && ROUNDDOWN (srcva, PGSIZE) != srcva)
19         return -E_INVAL;
20
21
22     if (srcva < (void *) UTOP) {
23         // check permission
24         if ((perm & PTE_U) == 0 || (perm & PTE_P) == 0)
25             return -E_INVAL;
26         // PTE_USER = PTE_U | PTE_P | PTE_W | PTE_AVAIL
27         if ((perm & ~PTE_USER) > 0)
28             return -E_INVAL;
29     }
30
31
32     pte_t *pte;
33     struct Page *p;
34
35     // the page is not mapped in current env
36     if (srcva < (void *) UTOP && (p = page_lookup (curenv->env_pgdir, srcva, &pte)
           ) == NULL)
37         return -E_INVAL;
38
39     if (srcva < (void *) UTOP && (*pte & PTE_W) == 0 && (perm & PTE_W) > 0)
40         return -E_INVAL;
41
42
43     // will send a page
44     if (srcva < (void *) UTOP && dstenv->env_ipc_dstva != 0) {
45         if (page_insert (dstenv->env_pgdir, p, dstenv->env_ipc_dstva, perm) < 0)
46             return -E_NO_MEM;
47
48         dstenv->env_ipc_perm = perm;
49     }
50
51     dstenv->env_ipc_from = curenv->env_id;
52     dstenv->env_ipc_value = value;
53     dstenv->env_status = ENV_RUNNABLE;
54     dstenv->env_ipc_recving = 0;
55     dstenv->env_tf.tf_regs.reg_eax = 0;
56
57     return 0;
58 }
59
60 static int
61 sys_ipc_recv(void *dstva)
62 {
63     if (dstva < (void *) UTOP && ROUNDDOWN (dstva, PGSIZE) != dstva)
64         return -E_INVAL;
65
```

```
66      curenv->env_ipc_dstva = dstva;
67      curenv->env_ipc_recving = 1;
68      curenv->env_ipc_from = 0;
69      curenv->env_status = ENV_NOT_RUNNABLE;
70
71      sched_yield ();
72  }
```

完成后记得添加相应的分发机制，把这两个调用号加上。

然后就是两个库函数:

lib/ipc.c

```
1   int32_t
2   ipc_recv(envid_t *from_env_store, void *pg, int *perm_store)
3   {
4       int r;
5
6       if (pg == NULL)
7           r = sys_ipc_recv ((void *) UTOP);
8       else
9           r = sys_ipc_recv (pg);
10
11
12      struct Env *curenv = (struct Env *) envs + ENVX (sys_getenvid ());
13
14      if (from_env_store != NULL)
15          *from_env_store = r < 0 ? 0 : curenv->env_ipc_from;
16
17      if (perm_store != NULL)
18          *perm_store = r < 0 ? 0 : curenv->env_ipc_perm;
19
20      if (r < 0)
21          return r;
22
23          return curenv->env_ipc_value;
24  }
25
26  void
27  ipc_send(envid_t to_env, uint32_t val, void *pg, int perm)
28  {
29      int r;
30
31      while ((r = sys_ipc_try_send (to_env, val, pg != NULL ? pg : (void *) UTOP,
             perm)) < 0) {
32          if (r != -E_IPC_NOT_RECV)
33              panic ("ipc_send:␣send␣message␣error␣%e", r);
34
35          sys_yield ();
36      }
37  }
```

这里要注意的是现在已经是在用户态下使用的库函数，那么在系统调用里的那些内核变量没有了，比如在`ipc_recv()`里需要用到当前环境的env，那么就只能通过envs来查找，envs定义在lib/entry.S 中。

至此lab4全部Exercise完成，可以通过make grade的所有测试。

Challenge! The ipc_send function is not very fair. Run three copies of user/fairness and you will see this problem. The first two copies are both trying to send to the third copy, but only one of them will ever succeed. Make the IPC fair, so that each copy has approximately equal chance of succeeding.

这里我的想法主要是限制那些发送消息次数太多的用户进程，所以我修改了Env的结构，增加了三个成员：

inc/env.h

```
    uint32_t env_ipc_send_succ;          // the number of try times
    envid_t env_ipc_send_to;             // the target envid
    uint32_t env_ipc_recv_min_send_succ; // the min success number of sender
```

- env_ipc_send_succ 表示该进程成功发送消息的次数
- env_ipc_send_to 表示该进程当前如果是正在发送消息，目标进程的envid
- env_ipc_recv_min_send_succ 表示如果当前进程是在接受消息，那么所有向他发送消息的进程中成功次数最少的那个是多少

这样修改以后注意进程初始化的设置，如果用户没有在发送消息，记得send_to设置为-1，如果没有在接受消息，那么min_send_succ设置为0xffffffff，然后 `sys_ipc_try_send()` 修改为：

kern/syscall.c sys_ipc_try_send()

```
static int
sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned perm)
{
    curenv->env_ipc_send_to = envid;

    struct Env *dstenv;
    int r;


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


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

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


    if (srcva < (void *) UTOP) {
        // check permission
        if ((perm & PTE_U) == 0 || (perm & PTE_P) == 0)
            return -E_INVAL;
        // PTE_USER = PTE_U | PTE_P | PTE_W | PTE_AVAIL
        if ((perm & ~PTE_USER) > 0)
```

```
30              return -E_INVAL;
31      }
32
33
34
35      pte_t *pte;
36      struct Page *p;
37
38      // the page is not mapped in current env
39      if (srcva < (void *) UTOP && (p = page_lookup (curenv->env_pgdir, srcva, &pte)
            ) == NULL)
40          return -E_INVAL;
41
42      if (srcva < (void *) UTOP && (*pte & PTE_W) == 0 && (perm & PTE_W) > 0)
43          return -E_INVAL;
44
45
46      // check fairness
47
48      if (curenv->env_ipc_send_succ > dstenv->env_ipc_recv_min_send_succ)
49          return -E_IPC_NOT_RECV;
50
51
52      // will send a page
53      if (srcva < (void *) UTOP && dstenv->env_ipc_dstva != 0) {
54          if (page_insert (dstenv->env_pgdir, p, dstenv->env_ipc_dstva, perm) < 0)
55              return -E_NO_MEM;
56
57          dstenv->env_ipc_perm = perm;
58      }
59
60      dstenv->env_ipc_from = curenv->env_id;
61      dstenv->env_ipc_value = value;
62      dstenv->env_status = ENV_RUNNABLE;
63      dstenv->env_ipc_recving = 0;
64      dstenv->env_tf.tf_regs.reg_eax = 0;
65
66      curenv->env_ipc_send_to = -1;
67      curenv->env_ipc_send_succ++;
68
69      dstenv->env_ipc_recv_min_send_succ = 0xffffffff;
70      struct Env *envptr;
71      for (envptr = envs + 1; envptr < envs + NENV; envptr ++)
72          if (envptr->env_ipc_send_to == envid)
73              if (envptr->env_ipc_send_succ < dstenv->env_ipc_recv_min_send_succ)
74                  dstenv->env_ipc_recv_min_send_succ = envptr->env_ipc_send_succ;
75
76      return 0;
77  }
```

其大致思路为：如果一个进程尝试向目标进程发送消息，如果发现当前进程的成功次数比所有向该目标进程发消息的进程里成功次数最少的要多，那么放弃发送，重新尝试。

在一个进程消息发送成功之后，为目标进程更新相应的值。

测试的时候修改kern/init.c，添加5个user_fairness，那么使用make qemu看到的运行轨迹基本上就是各个进程都是在顺序发送消息的：

```
qemu -nographic -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
[00000000] new env 00001004
[00000000] new env 00001005
1002 loop sending to 1001
1003 loop sending to 1001
1004 loop sending to 1001
1005 loop sending to 1001
1001 recv from 1002
1001 recv from 1002
1001 recv from 1003
1001 recv from 1004
1001 recv from 1005
1001 recv from 1003
1001 recv from 1004
1001 recv from 1005
1001 recv from 1002
1001 recv from 1003
1001 recv from 1004
1001 recv from 1005
1001 recv from 1002
1001 recv from 1003
1001 recv from 1004
1001 recv from 1005
...
```