JOS 实验四实验记录

作者：卓达城
指导老师：邵志远
单位：华中科技大学集群网络与服务计算实验室

备注：本文档重点说明 exofork 函数的返回机制(下面将以黑体标志)，如果有不当，敬请发邮件到我的邮箱。
本文档最精彩的地方在于缺页中断处理函数的返回机制和堆栈的切换，文中以灰底，不同字体、加粗显示。
还有一处就是 vpd 和 vpt 的使用，这里用了回环搜索 pgdir 和二级页表，也用加粗显示。

如果要做实验四，要用 svn 把 mit 提供的实验代码很 lab3 的代码合并。前面的实验也如此。

修改：对于实验三，有部分地方的代码我写错了，要修改,具体如下：
函数：load_icode

```
        struct Elf *env_elf;
        struct Proghdr *ph;
        struct Page *pg;
        int i;
        unsigned int old_cr3;
        env_elf = (struct Elf *)binary;
        old_cr3 = rcr3();
        lcr3(PADDR(e->env_pgdir));
        if( env_elf->e_magic != ELF_MAGIC){
                return;
        }
        ph = (struct Proghdr*)((unsigned int)env_elf + env_elf->e_phoff );
        for(i = 0;i < env_elf->e_phnum; i++){
                if( ph->p_type == ELF_PROG_LOAD){
/                       segment_alloc(e,(void *)ph->p_va, ph->p_memsz);
        //              cprintf("[0x%x,0x%x]\n",ph->p_va,ph->p_va + size);
        //              memset((void *)ph-> p_va, 0, ph->p_memsz - ph->p_filesz);
                        memset((void *)ph->p_va, 0, ph->p_memsz);
        //              cprintf("segment_alloc success !!!\n");

                        memmove((void *)ph->p_va,(void *)((unsigned int )env_elf + ph->p_offset),ph->p_filesz);
/                       memset((void *)ph-> p_va + ph-> p_filesz, 0, ph->p_memsz - ph->p_filesz);

                }
                ph++;
        }
/       cprintf("for success!!\n");
        e->env_tf.tf_eip = env_elf->e_entry;
/       cprintf("env_elf %x\n",env_elf->e_entry);
        // Now map one page for the program's initial stack
        // at virtual address USTACKTOP - PGSIZE.

        // LAB 3: Your code here.
        i = page_alloc(&pg);
        if( i != 0){
                cprintf("load_icode page_alloc fail!!\n");
                                                                    352,
```

修改 user_mem_check 函数：

```
user_mem_check(struct Env *env, const void *va, size_t len, int perm)
{
        // LAB 3: Your code here.
        int i;
        unsigned int begin;
//      begin = (unsigned int)va / PGSIZE * PGSIZE + PGSIZE;
        begin = (unsigned int)va / PGSIZE * PGSIZE + PGSIZE;
//      cprintf("va:0x%x:\n",va);
//      cprintf("len: %d\n",len);
        if((pgdir_walk(env->env_pgdir,va, 0) != NULL) && ((unsigned int)*pgdir_walk(env->env_pgdir, va, 0) & (perm | PTE_P)) == (
perm | PTE_P)){
                for(;begin < ((unsigned int)va + len) / PGSIZE * PGSIZE; begin += PGSIZE){
                        if( begin  >= ULIM){
                                cprintf("user_mem_check: >= ULIM\n");
                                user_mem_check_addr = begin;
                        }
                        if((pgdir_walk(env->env_pgdir, va, 0) != NULL) && ((unsigned int)*pgdir_walk(env->env_pgdir,(void *) begi
n, 0) & (perm | PTE_P)) == (perm | PTE_P)){
                                //NOTHING TO DO
                        } else {
                                user_mem_check_addr = begin;
                                //cprintf("fuck 2!!\n");
                                cprintf("begin:0x%x\n",begin);
                                cprintf("user_mem_check: perm fail!!!\n");
                                return -E_FAULT;
                        }
                }
        }else {
                user_mem_check_addr = (unsigned int)va;
                //cprintf("%x\n",*pgdir_walk(env->env_pgdir,va,0));
                //cprintf("%x\n", PTE_P | PTE_U);
                //cprintf("fuck 1!!\n");
                cprintf("user_mem_check: perm fail!!\n");
                return -E_FAULT;
        }
        return 0;
}
```

现在进入实验 4：

PART A：实现调度算法

第一步：修改 kern/sched.c 里面的函数 sched_yield，具体代码如下：

```
        // LAB 4: Your code here.
        int i,j,k;
        cprintf("sched_yield\n");
        if( curenv == NULL){
                cprintf("First Environment run!\n");
                i = 0;
        }else {
                i = ENVX(curenv -> env_id);
//              cprintf("envno: %d\n",i);
        }
        for ( j = (i + 1) % NENV , k = 0;k < NENV; j ++,k++){
                if( j % NENV == 0){
//                      j++ ;
                } else {
                //      cprintf("%d\n",j);
                        if( envs[j % NENV].env_status == ENV_RUNNABLE){
//                              cprintf("env no: %d\n",j%NENV);
                                env_run(&envs[j % NENV]);
                                return ;
                        }
                }
        }
        cprintf(" No runnable environments!!It mostly will run idle!!\n");
        // the code below is temp
//      char buf[10] = { '\0' };
//      readline(buf);

        // Run the special idle environment when nothing else is runnable.
        if (envs[0].env_status == ENV_RUNNABLE)
                env_run(&envs[0]);
        else {
                cprintf("Destroyed all environments - nothing more to do!\n");
                while (1)
                        monitor(NULL);
        }
}
```

这个函数很简单，就是从当前环境的下一个环境一直遍历到当前环境，如果有可以运行的就开始运行。如果没有，就进入 idle 环境。

然后修改系统调用，具体是修改 syscall.c 里面的 syscall，具体代码如下：

```
//      cprintf("syscallno : %d\n", syscallno);
        switch(syscallno){
                case SYS_cputs:
                        sys_cputs((char*)a1,a2);
                        return 0;
                case SYS_cgetc:
                        sys_cgetc();
                        return 0;
                case SYS_getenvid:
                        return sys_getenvid();
                case SYS_env_destroy:
                        return sys_env_destroy((envid_t)a1);
                case SYS_yield:
                        sys_yield();
                        return 0;
                case SYS_exofork:
                        return sys_exofork();
                case SYS_env_set_status:
                        return sys_env_set_status((envid_t)a1,(int)a2);
                case SYS_page_alloc:
                        //int kkk;
                        return sys_page_alloc((envid_t)a1, (void *)a2, (int)a3);
//                      cprintf("kkk:%d\n",kkk);
//                      return kkk;
                case SYS_page_map:
                        return sys_page_map((envid_t)a1, (void *)a2, (envid_t)a3, (void *)a4, (int)a5);
                case SYS_page_unmap:
                        return sys_page_unmap((envid_t)a1, (void *)a2);
                case SYS_env_set_pgfault_upcall:
                        return sys_env_set_pgfault_upcall((envid_t)a1, (void *)a2);
                case SYS_ipc_try_send:
                        return sys_ipc_try_send((envid_t)a1, (uint32_t)a2, (void *)a3,(unsigned int)a4);
                case SYS_ipc_recv:
                        return sys_ipc_recv((void *)a1);
                default:
                        return -E_INVAL;
        }
//      panic("syscall not implemented");
```

这里要完成 part a 的话不用添加那么多，但是我是做完 lab4 再写的，所以就多加了一些进去。

现在在 init.c 里面多创建几个环境，以供测试，具体如下：

```
#if defined(TEST)
        // Don't touch -- used by grading script!
        ENV_CREATE2(TEST, TESTSIZE);
        cprintf("test!!\n");
#else
        // Touch all you want.
//      ENV_CREATE(user_forktree);
        ENV_CREATE(user_yield);
        ENV_CREATE(user_yield);
//      ENV_CREATE(user_yield);
//      ENV_CREATE(user_yield);

//      cprintf("user_hello!!\n");
#endif // TEST*


        // Schedule and run the first user environment!
        sched_yield();
```

这里为了达到实验效果，在 bochs 开始一两秒后要 bochs 关掉，或者在 sched_yield 里面把多余的调试信息删除掉，才能看到效果，不然程序会不断地刷屏。又或者把

```
//      // the code below is temp
//      char buf[10] = { '\0' };
//      readline(buf);
```

这两句激活，也可以达到效果的。

现在开始实现 sys_exofork 函数：
这个函数主要是用来创建环境，这个函数也是整个实验最为难理解的函数之一，这里将详细解释，如果讲的不好，请莫怪，因为实在比较复杂。函数代码不多，具体如下：

```
static envid_t
sys_exofork(void)
{
        // Create the new environment with env_alloc
        // It should be left as env_alloc created it
        // status is set to ENV_NOT_RUNNABLE, and th
        // from the current environment -- but tweak
        // will appear to return 0.

        // LAB 4: Your code here.
        struct Env *env;
        cprintf("brakpoint1\n");
        if( env_alloc(&env, curenv->env_id) < 0){
                return -E_NO_FREE_ENV;
        }
        env -> env_status = ENV_NOT_RUNNABLE;
        env -> env_tf = curenv -> env_tf;

        env -> env_tf.tf_regs.reg_eax = 0;


//      cprintf("env-> env_id:%d\n",env->env_id);
        return env -> env_id;
//      panic("sys_exofork not implemented");
}
```

要理解这个函数，又要回顾一下中断和异常这方面的知识：

我们可以大体的把 cpu 的中断和异常弄成四类，fault、trap、interrupt（用户调用）、abort 不好用中文翻译 ，关于那个中断号

我们这里都用 call 来做说明，中断和异常进行特权级的切换（只要设计者愿意，可以在任何特权级间切换），而 call 是不能的，call tss 段也可以切换，但是只能同特权级或者高特权级到低特权级的切换。现在我们先不管特权级。

那么如 call function，在 function 执行之前，cpu 先把 cs，eip（如果是段内的话，就只入栈 eip）入栈，这里最值得关注的是 eip，在 call 指令下，eip 是执行 call 的下一条指令的，所以当 function 执行完之后，也就是 ret 之后（弹出所有刚才入栈的东西），程序在 call 的下一条指令执行。

中断不仅入栈 cs、eip，它还会入栈 ss、esp、eflag、cs、eip、errno、interrupt num。

这里的 interrupt 跟 call 是完全一样的，而且是程序员调用的。有些也不是，例如键盘中断，总之它是一些好的情况，下面三个都是坏的情况。

对于 fault 跟 call 不一样的地方就是 eip 的问题，eip 是指向引发异常的指令的，就是如果指令 mov eax ebx，引起 fault，那么 eip 就指向这条指令，然后处理完之后，cpu 重新执

行 mov eax ebx。这里特别注意，fault 都是不能由程序员调用的，是硬件发出的。

对于 trap，trap 跟 interrupt 是一样的，但是它也是硬件发出，例如溢出，处理完后，运行下一条指令，还有就是 trap 执行时 eflag 的 if 位不会置位，interrupt 会的。

对于 abort，这个不允许程序或者任务继续执行，是用来报告错误的。

讲了一大堆之后，现在来看看函数的返回值机制，函数的返回值总是放在 eax 中，这也就是为什么我们的函数只能返回一个值的缘故。

讲了这么多废话，现在开始进入正题：

```
// will return 0 instea
envid = sys_exofork();
```

这句代码是在 dumbfork.c 里面。

我们为了便于理解可以写成：

int i = sys_exofork();

envid = i;

sys_exofork()是一个 interrupt，按照之前写的我们可以知道它入栈的 eip 是指向 envid = i 的，也就是 sys_exofork 执行完之后要执行 envid = i 这句代码。那么 exofork 做了什么呢？这个我们从上面代码可以清楚的看到，它在申请了一个新的环境，然后把父环境的寄存器复制进去，然后把 tf ->eax 变成 0。注意这里不是 eax 寄存器，tf 是用来还原寄存器用的。这时候，由实验 3 我们可以知道父环境中 tf ->eip 是指向 envid = i 这条指令，请看 trapencry.c 这个文件。然后函数返回，中断返回，cs 和 eip，eflag ，ss ，esp 出栈，父环境运行 envid = i 这行代码。这时候，父环境设置子环境的各种东西，地址映射之类，下面会讲，总之就是让子环境可以运行吧。然后父环境调用 sched_yield，然后刚才建的子环境被还原。子环境中的 tf 是 eip 指向 envid = i 这句代码，然后 eax 被设置成 tf ->eax 就是刚才设置的 0，所以 envid = i 也就是 envid = eax = tf->eax=0。

多么绝妙的思想啊!!!!!!!!!!!!!

现在实现 sys_env_set_status 函数：

这个这个函数就是把环境设置成可以运行和不可以运行，代码如下：

```
// LAB 4: Your code here.
struct Env *env;
if( status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE){
        cprintf("parameter error!!\n");
        return -E_INVAL;
}
if( envid2env(envid, &env, 1) < 0){
        return -E_BAD_ENV;
}
env -> env_status = status;
return 0;
```

现在实现函数 sys_page_alloc：

这个函数的功能就是分配一个内存也给 envid 对应的 env，并且映射到地址 va 处。

```
	// LAB 4: Your code here.
	struct Env *env;
	struct Page *pg;
	if( (unsigned int) va >= UTOP || ((unsigned int) va) % PGSIZE != 0){
		return -E_INVAL;
	}
	if( (perm & (PTE_U | PTE_P)) != (PTE_U | PTE_P) ){
		return -E_INVAL;
	}
	if( envid2env(envid, &env, 1) < 0) {
		return -E_BAD_ENV;
	}
	cprintf("cs: %x\n", read_cs());
	cprintf("envid: %x\n", env -> env_id);
	if( page_alloc(&pg) < 0){
		return -E_NO_MEM;
	}
	cprintf("va : %x\n",(unsigned int)va);
	cprintf("*va: %x\n", *(int *)va);
	  cprintf("---->%x\n",*pgdir_walk(env ->env_pgdir, va, 0));
	cprintf("--------------------------page: %x\n",page2pa(pg));
	if( page_insert(env->env_pgdir, pg, va, perm) < 0){
		page_decref(pg);
		return -E_NO_MEM;
	}
```

```
	memset(page2kva(pg), 0, PGSIZE);
	return 0;
```

函数 static int

sys_page_map(envid_t srcenvid, void *srcva,

envid_t dstenvid, void *dstva, int perm)

作用是把 srcenvid 中 srcva 对应的页映射到 dstenvid 中的 dstva 处，具体代码实现如下：

```
	// LAB 4: Your code here.
	struct Env * srcenv;
	struct Env * dstenv;
	struct Page *pg;
	pte_t *pte;
	if( (unsigned int) srcva >= UTOP || (unsigned int) dstva >= UTOP || ((unsigned int)srcva % PGSIZE != 0) || ((unsigned int
)dstva % PGSIZE != 0)){
		return -E_INVAL;
	}
	if( (perm & (PTE_U | PTE_P)) != (PTE_U | PTE_P) ){
		return -E_INVAL;
	}
	if( envid2env( srcenvid, &srcenv, 1) < 0 || envid2env( dstenvid, &dstenv, 1) < 0){
		return -E_BAD_ENV;
	}
	if( (pg = page_lookup(srcenv->env_pgdir, srcva, &pte)) == NULL || ((*pte) & PTE_P) == 0){
		return -E_INVAL;
	}
	if( (*pte | PTE_W) == 0 && (perm | PTE_W) != 0){
		return -E_INVAL;
	}
	if( page_insert(dstenv->env_pgdir, pg, dstva, perm) <0){
		return -E_NO_MEM;
	}
	return 0;
```

函数 static int

sys_page_unmap(envid_t envid, void *va)

解除 envid 中 va 映射的页，具体代码如下：

```
        // LAB 4: Your code here.
        struct Env *env;
        if( (unsigned int) va >= UTOP){
                return -E_INVAL;
        }
        if( envid2env(envid, &env, 1) < 0){
                return -E_BAD_ENV;
        }
        page_remove( env->env_pgdir, va);
        return 0;
//      panic("sys_page_unmap not implemented");
```

这里我们还可以看看 envid2env 这个函数，如果输入是 0 就返回当前环境，代码就不贴了。

此时，在 init.c 里面修改代码

```
        // Touch all you want.
        ENV_CREATE(user_dumbfork);
/       ENV_CREATE(user_forktree);
/       ENV_CREATE(user_yield);
/       ENV_CREATE(user_yield);
/       ENV_CREATE(user_yield);
/       ENV_CREATE(user_yield);
```

就可以看到相应效果。

现在进入 PART B

我们先看看 dumbfork.c 里面的一个函数，这个函数是 duppage 写得很有意思。

```
void
duppage(envid_t dstenv, void *addr)
{
        int r;
//      cprintf("breakpoint2\n");
        // This is NOT what you should do in your fork.
        if ((r = sys_page_alloc(dstenv, addr, PTE_P|PTE_U|PTE_W)) < 0){
                panic("sys_page_alloc: %e", r);
        }
//      cprintf("breakpoint3\n");
        if ((r = sys_page_map(dstenv, addr, 0, UTEMP, PTE_P|PTE_U|PTE_W)) < 0)
                panic("sys_page_map: %e", r);
        memmove(UTEMP, addr, PGSIZE);
        if ((r = sys_page_unmap(0, UTEMP)) < 0)
                panic("sys_page_unmap: %e", r);
}
```

这个函数的具体操作是:

先给子环境分配一个页（我们暂时叫做 pg），并映射到 dstenv 的 addr 上

然后把 pg 映射到父环境的地址 UTEMP 中，由于现在是在父环境中运行，用的是父进程的地址空间，所以需要这样做。然后把父进程中 addr 对应页的内容复制到 UTEMP 中，然后删除父环境中 UTEMP 的映射，这里并没有删除页，因为它还映射到子环境中，就是说 pp_ref > 0。

EX 4

sys_env_set_pgfault_upcall 函数，代码如下：

```
static int
sys_env_set_pgfault_upcall(envid_t envid, void *func)
{
        // LAB 4: Your code here.
//      panic("sys_env_set_pgfault_upcall not implemented");
        struct Env *env;
        if( envid2env(envid, &env, 1) < 0){
                cprintf("sys_env_set_pgfault_upcall fail!!\n");
                return -E_BAD_ENV;
        }
        env -> env_pgfault_upcall = func;
        cprintf("func:%d\n",(unsigned int) func);
        return 0;
}
```

代码很简单，一看就可以明白。

EX 5

要完成 ex5，首先要在中断哪里设置好 page_fault 的入口，这里在 dispatch 函数里面已经设置过了，具体参照本文档前面关于 dispatch 的实现。

然后是：

void

page_fault_handler(struct Trapframe *tf)

这个函数的作用是设置好调用缺页处理函数的参数，缺页处理函数的参数和堆栈，参数是一个命名为 UTrapframe 的数据结构，堆栈是以 UXSTACKTOP 为栈顶的堆栈。

```
        // LAB 4: Your code here.
        if( curenv -> env_pgfault_upcall != NULL){
                struct UTrapframe *utf;
                if(tf -> tf_esp > USTACKTOP){
                        utf = (struct UTrapframe *)((void *)(tf->tf_esp) - sizeof(struct UTrapframe) -4);
                } else {
                        utf = (struct UTrapframe *)((void *)UXSTACKTOP - sizeof(struct UTrapframe));
                }
                user_mem_assert(curenv, utf, sizeof(struct UTrapframe), PTE_P | PTE_U | PTE_W);
                utf->utf_fault_va = fault_va;
                utf->utf_err = tf->tf_err;
                utf->utf_regs = tf->tf_regs;
                utf->utf_eip = tf->tf_eip;
                utf->utf_eflags = tf->tf_eflags;
                utf->utf_esp = tf->tf_esp;
                tf->tf_eip = (uintptr_t)(curenv ->env_pgfault_upcall);
                tf->tf_esp = (uintptr_t)utf;
                env_run(curenv);

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

先看这两句代码：

```
    if(tf -> tf_esp > USTACKTOP){
            utf = (struct UTrapframe *)((void *)(tf->tf_esp) - sizeof(struct UTrapframe) -4);
    } else {
            utf = (struct UTrapframe *)((void *)UXSTACKTOP - sizeof(struct UTrapframe));
    }
```

因为 pagefault 是一个中断，所以 tf 里面放的是什么大家应该比计较清楚，如果不清楚请参照 trapentry.S 以及其它函数。这两句代码的作用是判断是不是迭代的缺页中断，这里有一个

值得注意的地方就是-4),为什么要-4 呢？这是因为我们还要留 4 个字节把指向 utf 的 esp 放到堆栈里面，这样它就可以作为参数传递给以后的函数。如下：

```
pushl %esp
movl _pgfault_handler, %eax
call *%eax
addl $4, %esp
```

至于为什么要这样判断是否迭代，原因很简单，就是因为如果是缺页中断迭代，esp 肯定在 UXSTACKTOP - PGSIZE , UXSTACKTOP-1 之间>USTACKTOP。

然后是下面的代码：

```
        user_mem_assert(curenv, utf, sizeof(struct UTrapframe), PTE_P | PTE_U | PTE_W);
        utf->utf_fault_va = fault_va;
        utf->utf_err = tf->tf_err;
        utf->utf_regs = tf->tf_regs;
        utf->utf_eip = tf->tf_eip;
        utf->utf_eflags = tf->tf_eflags;
        utf->utf_esp = tf->tf_esp;
        tf->tf_eip = (uintptr_t)(curenv ->env_pgfault_upcall);
        tf->tf_esp = (uintptr_t)utf;
        env_run(curenv);

    }else {
        cprintf("curenv -> env_pgfault_upcall == NULL\n");
    }
```

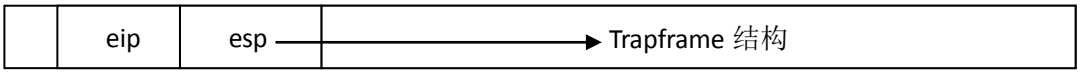
以上代码就是填充 utf 结构的代码，其中值得注意的一句是tf->tf_esp = (uintptr_t)utf;

这里把原来的栈换成新的栈。然后运行 env_run，现在 cpu 切换到用户态（这很重要，因为它会影响到 ret 指令的操作，同级返回 ret 指令只 pop eip，不同特权级会 pop ip 和 pop cs，call 指令道理一样），为什么？因为这里的还原的 cs 是用户态的 cs。刚看这里也不会明白，如果要弄明白它到底是什么葫芦卖什么药，我们要结合下面的代码（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
```

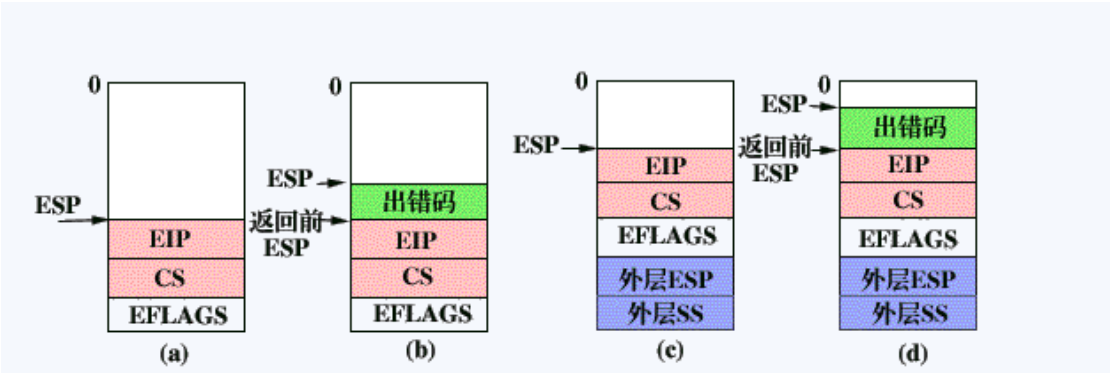
```
//   What registers are available for intermediate calculations?
//
// LAB 4: Your code here.
addl $8, %esp
movl 32(%esp),%eax
movl 40(%esp),%ebx
movl %eax,-4(%ebx)
// Restore the trap-time registers.
// LAB 4: Your code here.
popal
// Restore eflags from the stack.
// LAB 4: Your code here.
addl $4,%esp
popfl
// Switch back to the adjusted trap-time stack.
// LAB 4: Your code here.
popl %esp
subl $4, %esp
// Return to re-execute the instruction that faulted.
// LAB 4: Your code here.
ret
```

现在开始重点研究栈的结构，结合实验 2，我们可以知道进入中断之后系统的堆栈情况，如下图：



实际的 esp

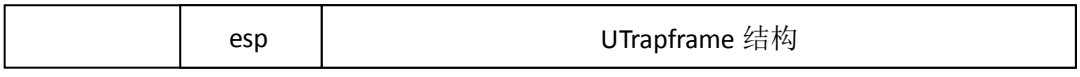
这里又出现一个重要问题,这里看看现在的堆栈情况到底 Trapframe 里面的 esp 和 ss 是什么？由 intel 手册可以知道中断的入栈具体入栈情况如下：



无特权级转换　　有特权级转换

这里我们还有一点要提到的是当 cpu 切换到内核的时候，内核的堆栈从哪里得到，答案就是从 TSS 里面得到，JOS 只定义了一个 TSS 段，索引是 0x28，在 idt_init 里面设置的。

当系统出现缺页中断的时候，cpu 把当前运行指令的 eip（当前指令的 eip）入栈，由实验 2 我们可以知道它是放在 Trapframe 里面的。然后 `utf->utf_eip = tf->tf_eip;` 又把它放到 utf 里面，进入缺页处理函数之后，栈的状态应该是：



关于 UTrapframe 里面的结构，请自己参照 UTrapframe 定义。

```
addl $8, %esp
movl 32(%esp),%eax
movl 40(%esp),%ebx
movl %eax,-4(%ebx)
```

这几句代码是把 eip==tf->eip（缺页中断的 eip）放到 eax 里面。32(%esp)刚好是指向 eip 的，请核对 UTrapframe 结构。然后把 esp ==外部 esp，也就是用户程序的 esp（请看本页第一个图）。然后在 eax（里面是 eip）放到用户栈里面。然后把一大堆寄存器 pop 出来，这里有个问题，为什么没有了 cs 和 ss 等段寄存器呢？因为都在用户态，段都一样的，所以这里不用还原段寄存器。

```
popl %esp
subl $4, %esp
```

popl %esp，这句代码恢复到用户栈，即(USTACKTOP)。在这句代码之前是(UXSTACKTOP)。由于在用户段，没有段切换，所以 ret 指令只把 eip pop 出来（对于不同的情况，ret 指令的操作是不一样的），然后程序就会在 eip 哪里继续运行，这里就是产生缺页中断的代码处。subl $4,%esp 是把 esp 指向刚才压入的 eip，ret 指令将会把它 pop 出来。

EX7

set_pgfault_handler 函数：

```
void
set_pgfault_handler(void (*handler)(struct UTrapframe *utf))
{
	int r;
//	cprintf("_pgfault_handler:%x\n",(unsigned int)_pgfault_handler);
	if (_pgfault_handler == 0) {
		// First time through!
		// LAB 4: Your code here.
		envid_t env_id = sys_getenvid();
//		cprintf("set_pgfault_handler:sys_getenvid finished!!\n");
		if( sys_page_alloc(env_id, (void *)(UXSTACKTOP- PGSIZE), PTE_P | PTE_U | PTE_W) < 0){
			cprintf("set_pgfault_handler: sys_page_alloc fail!!\n");
			return;
		}
//		cprintf("set_pgfault_handler:  sys_page_alloc finished!!\n");
		if( sys_env_set_pgfault_upcall(env_id, _pgfault_upcall) < 0){
			cprintf("set_pgfault_handler: sys_env_set_pgfault_upcall fail!!\n");
			return;
		}
//		cprintf("set_pgfault_handler: sys_env_set_pgfault_upcall!!\n");
//		_pgfault_handler = handler;
//		panic("set_pgfault_handler not implemented");
	}

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

这个函数也是相对简单，但是写完这个函数以后我们应该要明白 pagefault 的调用流程。如下：

缺页中断发生-> 跳到汇编代码_pgfault_upcall ->运行 handler 函数，就是上面的最后一句代码设置的函数。

现在可以进入测试：

修改 init.c，然后编译运行就可以看到想要的结果。

static void

pgfault(struct UTrapframe *utf)

代码如下：

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

        // Check that the faulting access was (1) a write, and (2) to a
        // copy-on-write page.  If not, panic.
        // Hint:
        //   Use the read-only page table mappings at vpt
        //   (see <inc/memlayout.h>).

        // LAB 4: Your code here.
        pte_t pte = ((pte_t *)vpt)[VPN(addr)];
        if(!((err & FEC_WR) != 0 && (pte & PTE_COW) != 0)) {
                panic("lib/pgfault.c: bad page fault at %08x err %08x\n", addr, err);
                return;
        }

        // Allocate a new page, map it at a temporary location (PFTEMP),
        // copy the data from the old page to the new page, then move the new
        // page to the old page's address.
        // Hint:
        //   You should make three system calls.
        //   No need to explicitly delete the old page's mapping.

        // LAB 4: Your code here.
        if ((r = sys_page_alloc(0, PFTEMP, PTE_P | PTE_U | PTE_W)) < 0)
                panic("pgfault page allocate for PFTEMP error\n");
        memmove(PFTEMP, ROUNDDOWN(addr, PGSIZE), PGSIZE);
        if ((r = sys_page_map(0, PFTEMP, 0, ROUNDDOWN(addr, PGSIZE), PTE_P | PTE_U | PTE_W)) < 0)
                panic("pgfault page map for PFTEMP error\n");
        sys_page_unmap(0, PFTEMP);

//      panic("pgfault not implemented");
}
```

这个函数大概就是判断当前缺页的类型是否为 FEC_WR（缺页中断时有错误码的，可以根据错误码判断，cpu 自动压入），当前页是否为 PTE_COW。

然后把要写的页的内容复制到 PFTEMP，然后再把地址映射到 PFTEMP，具体实现跟 duppage 相同，请看上面的描述在 part b 开始哪里。

现在看看 vpt 和 vpd 到底是怎么回事？

先看看 env_setup_vm 里面的两句代码，我们还可以知道 vpt = UVPT 的

```
.globl envs
.set envs, UENVS
.globl pages
.set pages, UPAGES
.globl vpt
.set vpt, UVPT
.globl vpd
.set vpd, (UVPT+(UVPT>>12)*4)
```

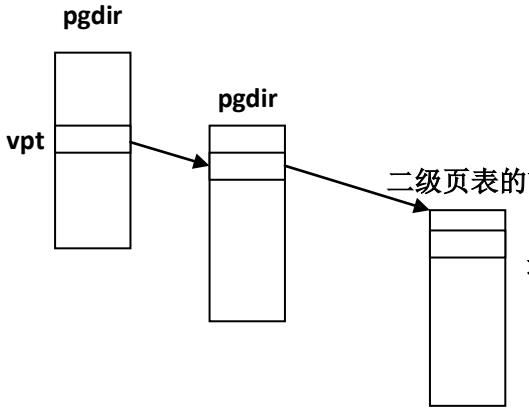
具体的值是 0xef400000，也就是高 10 位以后全部是零。

看下面两句代码：

```
e->env_pgdir[PDX(VPT)]  = e->env_cr3 | PTE_P | PTE_W;
e->env_pgdir[PDX(UVPT)] = e->env_cr3 | PTE_P | PTE_U;
```

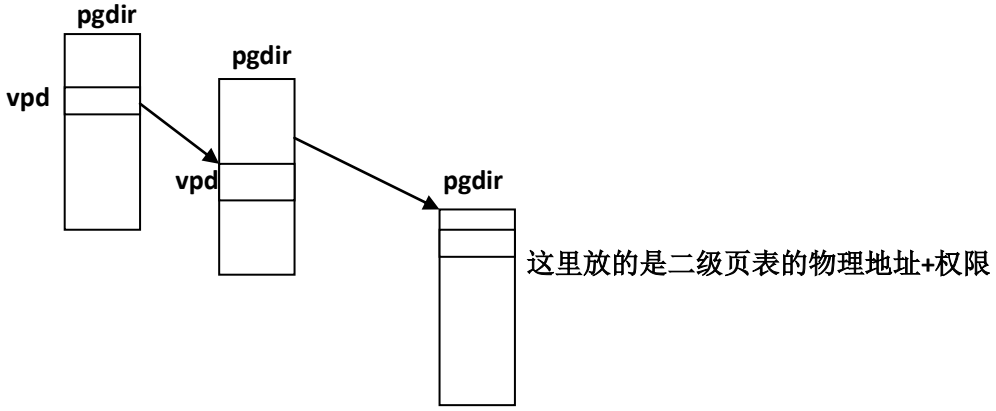
由于对 vpt 进行了映射，所以当代码访问 vpt 的时候会出现下面的情况：

回顾一下 cpu 在分页的情况下的寻址就很容易看懂下面的图了。




```
.set vpd, (UVPT+(UVPT>>12)*4)
```

有代码可以知道 vpd 就是把 UVPT 的头 10 位复制到 UVPT 的中间 10，那样会有什么样的效果呢？



有了以上基础，我们可以看看 fork 到底是怎么样运行的。
主要是理解 for 循环，其它的都不难理解。

```
envid_t
fork(void)
{
        // LAB 4: Your code here.
//      panic("fork not implemented");
        envid_t envid;
        uint8_t * addr;
        int r;
        set_pgfault_handler(pgfault);
        envid = sys_exofork();
        if (envid < 0)
                panic("fork: unable to create new env.\n");
        // child
        if (envid == 0) {
                env = &envs[ENVX(sys_getenvid())];
                return 0;
        }
        // parent
        int i, j;
        //here i < 1024 notice zdc
        for (i = 0; i * PTSIZE < UTOP; i++) {
                if (((pte_t *)vpd)[i] & PTE_P) {
                        for (j = 0; j * PGSIZE + i * PTSIZE < UTOP && j < NPTENTRIES; j++) {
                                if (j * PGSIZE + i * PTSIZE == UXSTACKTOP - PGSIZE)
                                        continue;
                                pte_t p = ((pte_t *)vpt)[i * NPTENTRIES + j];
                                if ((p & PTE_P) && (p & PTE_U))
                                        if ((r = duppage(envid, i * NPTENTRIES + j)) < 0)
                                                panic("fork: duppage error\n");
                        }
                }
        }

        if ((sys_page_alloc(envid, (void *)UXSTACKTOP - PGSIZE, PTE_P | PTE_U | PTE_W)) < 0)
                panic("fork: exception stack allocate error\n");
```

```
        extern void _pgfault_upcall(void);
        if ((sys_env_set_pgfault_upcall(envid, _pgfault_upcall)) < 0)
                panic("fork: set pgfault upcall error\n");
        if ((sys_env_set_status(envid, ENV_RUNNABLE)) < 0)
                panic("fork: set env status error\n");
        return envid;
}
```

先看 for 循环

先对 vpd 进行遍历，其实就是对 pgdir 进行遍历，这个有了上面的图很容易理解。当找到存在的话，然后遍历它对应的页。

这里想一想 vpt + i * NPTENTRIES 对应的是什么？

就是第 i 个页目录对应的二级页表的首地址，再加上 j 就是我们要映射的页。

这里细细想一下就会感到 jos 作者思想之精妙。

现在再回过头来看 pagefault 函数，就很容易明白了。

最后看看 duppage 函数

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

        // LAB 4: Your code here.
//      panic("duppage not implemented");
        //panic("duppage not implemented");
        pte = vpt[pn];
        addr = (void *)(pn * PGSIZE);
        if ((pte & PTE_P) == 0 || (pte & PTE_U) == 0)
                panic("duppage: pte unpresent or non-user\n");
        if ((pte & PTE_W) == 0 && (pte & PTE_COW) == 0) {
                if ((r = sys_page_map(0, addr, envid, addr, PTE_P | PTE_U)) < 0)
                        return r;
        }
        else {
                if ((r = sys_page_map(0, addr, envid, addr, PTE_P | PTE_U | PTE_COW)) < 0)
                        return r;
                if ((r = sys_page_map(0, addr, 0, addr, PTE_P | PTE_U | PTE_COW)) < 0)
                        return r;
        }
        return 0;
}
```

这个函数就是判断当前页是否为用户可写或者 cow（copy on write），如果不是，直接映射，不改属性，如果是就映射两次，并把属性改成 PTE_COW。这里无论父进程还是子进程都要做这个事情。为什么呢？（这里要说明一点，父进程跟子进程是完全隔离的）。
当父进程建立了一个子进程（环境）之后，子进程 addr 映射父进程（环境）的一页 p，如果我们不复制多一次，就是没有下面这句代码：

```
    if ((r = sys_page_map(0, addr, 0, addr, PTE_P | PTE_U | PTE_COW)) < 0)
```

子进程写 addr 之前，如果父进程改变 p 的内容，那么子进程里面的数据也变了，这不是 jos 的设计思想。
但是如果有上面这句代码的话，当父进程要写的时候，同样发生 copy on write，所以不会改变原来的那一页的内容，就是子进程里面的数据也是不会变的。

改变 init.c 里面的内容，就可以看到相应的结果。

```
//      ENV_CREATE(user_primes);
        ENV_CREATE(user_forktree);
//      ENV_CREATE(user_faultalloc);
//      ENV_CREATE(user_faultdie);
//      ENV_CREATE(user_dumbfork);
```

现在进入 PART C
首先实现时钟中断，相对上面的东西，这个太简单了，要改的地方是：
trapentry.S

```
TRAPHANDLER_NOEC(device_not_available, T_DEVICE)
TRAPHANDLER_NOEC(double_fault, T_DBLFLT)

TRAPHANDLER_NOEC(float_point_error, T_FPERR)
TRAPHANDLER_NOEC(timer, IRQ_OFFSET + IRQ_TIMER)

TRAPHANDLER(invalid_TSS, T_TSS)
TRAPHANDLER(segment_not_present, T_SEGNP)
```

trap.c

```
extern void machine_check();
extern void SIMD_float_point_error();
extern void timer();
extern void system_call();
```

idt_init 函数

```
SETGATE(idt[T_FPERR], 0, GD_KT, float_point_error, 0);
SETGATE(idt[T_ALIGN], 0, GD_KT, alignment_check, 0);
SETGATE(idt[T_MCHK], 0, GD_KT, machine_check, 0);
SETGATE(idt[IRQ_OFFSET + IRQ_TIMER], 0, GD_KT, timer, 0);
```

trap_dispatch(struct Trapframe *tf)函数

```
}
if(tf->tf_trapno == T_BRKPT){
        monitor(tf);
        return;
}
if(tf -> tf_trapno == IRQ_OFFSET + IRQ_TIMER){
        cprintf("timer!!\n");
        panic("timer interrupt");
        sched_yield();
        return;
}
// Handle clock and serial interrupts.
// LAB 4: Your code here.
```

env_alloc 函数（env.c)

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

这里设置是让用户程序运行的时候允许硬件中断。

根据 jos 的要求，是内核态不能发生中断，用户态必须发生中断。

我们这里可能会问当系统切换到内核的时候，哪里设置了 eflag 让它不能发生中断呢？答案在中断的机制里面，当系统发生 interrupt 时，eflag 自动把 if 位变 0,trap 不会。我们在我们的代码中把 idt 里面的映射都设为 interrupt，我们看看:

```
extern char coprocessor_segment_overrun[];
SETGATE(idt[T_DIVIDE], 0, GD_KT, divide_error, 0);
SETGATE(idt[T_DEBUG], 0, GD_KT, debug, 0);
SETGATE(idt[T_NMI], 0, GD_KT, nmi, 0);
SETGATE(idt[T_BRKPT], 0, GD_KT, break_point, 3);
SETGATE(idt[T_OFLOW], 0, GD_KT, overflow, 0);
```

0 表示是 interrupt，1 表示是 trap。

具体请看上面关于中断的解释。

实现 IPC

sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned perm)函数：

```
static int
sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned perm)
{
	// LAB 4: Your code here.
//	panic("sys_ipc_try_send not implemented");
	struct Env * env;
	struct Page * page = NULL;
	int r;
	if ((r = envid2env(envid, &env, 0)) < 0)
		return -E_BAD_ENV;
	if (env->env_ipc_recving == 0)
		return -E_IPC_NOT_RECV;
	if (srcva < (void *)UTOP) {
		if ((unsigned int)srcva % PGSIZE != 0)
			return -E_INVAL;
		if ((perm & PTE_U) == 0 ||
			(perm & PTE_P) == 0) //||
			//(perm & ~PTE_U & ~PTE_P & ~PTE_W & PTE_AVAIL) != 0)
			return -E_INVAL;
		if ((page = page_lookup(curenv->env_pgdir, srcva, NULL)) == NULL)
			return -E_INVAL;
	}
	env->env_ipc_recving = 0;
	env->env_ipc_from = curenv->env_id;
	env->env_ipc_value = value;
	if (srcva >= (void *)UTOP)
		env->env_ipc_perm = 0;
	if (srcva < (void *)UTOP && env->env_ipc_dstva < (void *)UTOP) {
		if ((r = page_insert(env->env_pgdir, page, env->env_ipc_dstva, perm)) < 0)
			return -E_NO_MEM;
		env->env_ipc_perm = perm;
	}
	env->env_status = ENV_RUNNABLE;
	return 0;
}
```

判断 srcva 是否为空，如果是直接通过 env->env_ipc_value 传送值，如果不是进行页映射，然后把接收数据的进程的状态改成可以运行，其它判断的不多说了，比较简单。

sys_ipc_recv(void *dstva)

```
sys_ipc_recv(void *dstva)
{
	// LAB 4: Your code here.
//	panic("sys_ipc_recv not implemented");
	if ((unsigned int)dstva % PGSIZE != 0)
		return -E_INVAL;
	curenv->env_ipc_recving = 1;
	curenv->env_ipc_dstva = dstva;
	curenv->env_status = ENV_NOT_RUNNABLE;
	curenv->env_tf.tf_regs.reg_eax = 0;
	sched_yield();
	return 0;
}
```

把当前进程阻塞，然后准备接收数据。

然后是

int32_t

ipc_recv(envid_t *from_env_store, void *pg, int *perm_store)

也比较简单，不做解释

```
int32_t
ipc_recv(envid_t *from_env_store, void *pg, int *perm_store)
{
        // LAB 4: Your code here.
//      panic("ipc_recv not implemented");
        int r;
        if ((r = sys_ipc_recv(pg == NULL ? (void *)UTOP : pg)) < 0) {
                return r;
//              if (from_env_store != NULL)
//                      *from_env_store = 0;
//              if (perm_store != NULL)
//                      *perm_store = 0;
        }
        if (from_env_store != NULL)
                *from_env_store = env->env_ipc_from;
        if (perm_store != NULL)
                *perm_store = env->env_ipc_perm;
        return env->env_ipc_value;
}
```

ipc_send(envid_t to_env, uint32_t val, void *pg, int perm)函数

```
void
ipc_send(envid_t to_env, uint32_t val, void *pg, int perm)
{
        // LAB 4: Your code here.
//      panic("ipc_send not implemented");
        int r;
        while ((r = sys_ipc_try_send(to_env, val, pg == NULL ? (void *)UTOP : pg, perm)) < 0) {
                if(r != -E_IPC_NOT_RECV)
                        panic("lib ipc send: %e", r);
                sys_yield();
        }
        return;
}
```

这以后就可以进行进程通信了

如果前面没有改 syscall 函数,现在可以把它改好，但是如果是按照上面做下来，应该就改了

```
    case SYS_page_map:
            return sys_page_map((envid_t)a1, (void *)a2, (envid_t)a3, (void *)a4, (int)a5);
    case SYS_page_unmap:
            return sys_page_unmap((envid_t)a1, (void *)a2);
    case SYS_env_set_pgfault_upcall:
            return sys_env_set_pgfault_upcall((envid_t)a1, (void *)a2);
    case SYS_ipc_try_send:
            return sys_ipc_try_send((envid_t)a1, (uint32_t)a2, (void *)a3, (unsigned int)a4);
    case SYS_ipc_recv:
            return sys_ipc_recv((void *)a1);
    default:
            return -E_INVAL;
```

然后修改 init.c

```
        // Touch all you want.
        ENV_CREATE(user_primes);
```

就可以见到相应结果。

源代码可以通过邮箱叫我提供!!