

OS Lab4 实验报告

(郑懿 1611361)

一、实验准备

本次试验在 Lab2 基础上进行操作，在了解了用户进程相关知识后，对本次试验进行多次尝试，并结合网上内容，完成本次试验。

依旧存在合并错误，在 pmap.c 和 trap.c 中，出现了合并错误，删除掉合并错误的乱码。

随后，在第一次 make qemu 中，不知道为什么，kern 无发生成一个叫 sendpage 的文件，无奈之下，只好从网上下载了一份 sendpage.c 文件，放进 user 目录下，然后开始第四次实验。

二、实验过程

PART A

练习 1:

lapic_init() 一开始调用 mmio_map_region() 函数，为了实现虚拟地址和物理地址的分配，所以必然会调用 boot_map_region()。返回 kern/pmap.c，查看该函数定义，进行一个简单分配。其次，根据注释要求，要将 size 大小 roundup PGSIZE，如果超过 MMIOLIM，简单的 panic，另外设置的 page 的权限是 PTE_PCD|PTE_PWT|PTE_W，代码如下：

```
static uintptr_t base = MMIOBASE;
size = ROUNDUP(size, PGSIZE);
if (base + size >= MMIOLIM)
    panic("overflow MMIOLIM");
boot_map_region(kern_pgdir,
                base,
                size,
                pa,
                PTE_PCD|PTE_PWT|PTE_W);
uintptr_t result = base;
base += size;
return (void *)result;
//panic("mmio_map_region not implemented");
```

练习 2:

通过阅读两个函数，大概了解程序运行过程：

- 首先是 I386_init() 再完成一切初始化后，调用 boot_aps();
- 随后 boot_aps() 把每个 CPU 代码复制到指定位置；

- 最后 `lapic_startap()` 来开始每一段 CPU。

整个过程都是在 CPU0 上，也就是 BSP 上完成的。

修改代码只是在 `page_init()` 时候，不要把 `MPENTRY_PADDR(0x7000)` 加入到 `free_page_list` 中去，所以只需要一个条件语句进行判断即可，代码如下：

```
size_t i;
size_t mp_page = PGNUM(MPENTRY_PADDR);
page_free_list = NULL;
int num_alloc = ((uint32_t)boot_alloc(0) - KERNBASE) / PGSIZE;
int num_iohole = 96;
for (i = 0; i < npages; i++) {
    if(i == 0){
        pages[i].pp_ref = 1;
    }
    else if(i == mp_page) continue;
    else if(i >= npages_basemem && i < npages_basemem + num_iohole + num_alloc) {
        pages[i].pp_ref = 1;
    }
    else {
        pages[i].pp_ref = 0;
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```

结果通过 `check_page_free_list()`，在 `check_kern_pgdir()` 中报错，结果如图：

```
physical memory: 88350K available, base = 0x0, extended = 0x352K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
kernel panic on CPU 0 at kern/pmap.c:864: assertion failed: check_va2pa(pgdir, base + KSTKGAP + i) == PADDR(percpu_kstacks[ln1] + i)
```

问题 1:

逐行比较 `kern/mpentry.S` 和 `boot/boot.S`。牢记 `kern/mpentry.S` 和其他内核代码一样也是被编译和链接在 `KERNBASE` 之上运行的。那么，`MPBOOTPHYS` 这个宏定义的目的是什么呢？为什么它在 `kern/mpentry.S` 中是必要的，但在 `boot/boot.S` 却不用？换句话说，如果我们忽略掉 `kern/mpentry.S` 哪里会出现问题呢？提示：回忆一下我们在 Lab 1 讨论的链接地址和装载地址的不同之处。

答：观察 `mpentry.S`，中有这样一段话：

```
: This code is similar to boot/boot.S except that
:   - it does not need to enable A20
:   - it uses MPBOOTPHYS to calculate absolute addresses of its
:     symbols, rather than relying on the linker to fill them
```

所以，使用 `MPBOOTPHYS` 是为了获得其他变量的物理地址，因为实模式无法寻址在高地址，但是 `boot.s` 本身就是实模式可以寻址的低地址，不需要再转换成物理地址。

练习 3:

这段代码只需要建立一个循环，依次为每个 CPU 用 boot_map_region 分配内核栈，值得注意的是，需要在 KSTKGAP 这段空间留出空间以防止两个 CPU 内核栈相互覆盖，NCPU 代表 CPU 个数，权限 kernel RW。所以代码如下：

```
int i = 0;
uintptr_t kstacktop_i;

for (i = 0; i < NCPU; i++)
{
    kstacktop_i = KSTACKTOP - i * (KSTKSIZE + KSTKGAP);
    boot_map_region(kern_pgdir,
        kstacktop_i - KSTKSIZE,
        ROUNDUP(KSTKSIZE, PGSIZE),
        PADDR(&percpu_kstacks[i]),
        PTE_W);
}
```

成功通过 check_kern_pgdir(), 结果如图：

```
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2
[00000000] new env 00001000
kernel panic on CPU 0 at kern/trap.c:334: page_fault in kernel mode, fault address 0
```

可以发现，程序报错，内核页发生缺页。目前尚不知原因。

练习 4:

阅读 trap_init_percpu(), 发现由于以前的代码只能适用于单 CPU 情况，可以通过 thiscpu 来访问当前 CPU，cpuid 或者 cpunum() 获得当前 id。随后依次初始化内核栈的 esp, ss, tss。代码如下：

```
thiscpu->cpu_ts.ts_esp0 = KSTACKTOP - cpunum() * (KSTKSIZE + KSTKGAP);
thiscpu->cpu_ts.ts_ss0 = GD_KD;
thiscpu->cpu_ts.ts_lomb = sizeof(struct Taskstate);

// Initialize the TSS slot of the gdt.
gdt[(GD_TSS0 >> 3) + cpunum()] = SEG16(STS_T32A, (uint32_t) (&(thiscpu->cpu_ts)),
    sizeof(struct Taskstate) - 1, 0);
gdt[(GD_TSS0 >> 3) + cpunum()].sd_s = 0;

// Load the TSS selector (like other segment selectors, the
// bottom three bits are special; we leave them 0)
ltr(GD_TSS0 + 8 * cpunum());

// Load the IDT
lidt(&idt_pd);
```

随后输入 make qemu CPUS=4, 获得以下结果：

```
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 4 CPU(s)
enabled interrupts: 1 2
SMP: CPU 1 starting
SMP: CPU 2 starting
SMP: CPU 3 starting
[00000000] new env 00001000
kernel panic on CPU 0 at kern/trap.c:334: page_fault in kernel mode, fault address 0
ss 0
```

CPU1, 2, 3 依次启动，依旧会有内核页缺页报错。

练习 5:

加锁的位置，前三个代码基本已经给定位置，至于释放锁的位置，当前 CPU 释放，所以当寄存器存入现在环境，随后释放锁，代码如下：

```
lcr3(PADDR(curenv->env_pgdir));
unlock_kernel();
env_pop_tf(&curenv->env_tf);
```

值得注意的是，在 mp_main() 添加锁的同时，需要注释掉 for(;;); 这个死循环，因为不再需要。

问题 2:

看起来使用全局内核锁能够保证同一时段内只有一个 CPU 能够运行内核代码。既然如此，我们为什么还需要为每个 CPU 分配不同的内核堆栈呢？请描述一个即使我们使用了全局内核锁，共享内核堆栈仍会导致错误的情形。

答：如果 CPU1 进入内核，压入数据后，让 CPU2 进入内核，就可能占据 CPU1 存放数据的地方，当 CPU2 解锁，CPU1 在进入内核，就无法使用上次压入的数据。而是使用 CPU2 压入的数据，会造成结果混乱。

练习 6:

首先，我们按照要求完成 sched_yield()，寻找一个可运行的环境，如果，没有可运行环境，判断当前环境是否可运行，代码如下：

```
idle = curenv;
int32_t start_envid = (idle)? ENVX(idle->env_id) : 0;
int32_t next_envid;
size_t i;
for(i = 0; i < NENV; i++)
{
    next_envid = (start_envid+i)%NENV;
    if(envs[next_envid].env_status == ENV_RUNNABLE) {
        env_run(&envs[next_envid]);
        return;
    }
}
if (idle && idle->env_status == ENV_RUNNING) {
    env_run(idle);
}
// sched_halt never returns
sched_halt();
```

随后在系统调用中加上轮转调度，代码如下：

```

|case SYS_yield:
|    sys_yield();
|    return 0;

```

在 init.c 中删除旧的环境，添加三个新的环境，代码如下：

```

|else
|    // Touch all you want.
|    //ENV_CREATE(user_primes, ENV_TYPE_USER);
|endif // TEST*
|    ENV_CREATE(user_yield, ENV_TYPE_USER);
|    ENV_CREATE(user_yield, ENV_TYPE_USER);
|    ENV_CREATE(user_yield, ENV_TYPE_USER);
|

```

结果如图：

```

[00000000] new env 00001000
[00000000] new env 00001001
[00000000] new env 00001002
Hello, I am environment 00001000.
Hello, I am environment 00001001.
Hello, I am environment 00001002.
Back in environment 00001000, iteration 0.
Back in environment 00001001, iteration 0.
Back in environment 00001002, iteration 0.
Back in environment 00001000, iteration 1.
Back in environment 00001001, iteration 1.
Back in environment 00001002, iteration 1.
Back in environment 00001000, iteration 2.
Back in environment 00001001, iteration 2.
Back in environment 00001002, iteration 2.
Back in environment 00001000, iteration 3.
Back in environment 00001001, iteration 3.
Back in environment 00001002, iteration 3.
Back in environment 00001000, iteration 4.

```

问题 3:

在你实现的 `env_run()` 中你应当调用了 `lcr3()`。在调用 `lcr3()` 之前和之后，你的代码应当都在引用变量 `e`，就是 `env_run()` 所需要的参数。在装载 `%cr3` 寄存器之后，MMU 使用的地址上下文立刻发生改变，但是处在之前地址上下文的虚拟地址（比如说 `e`）却还能够正常工作，为什么 `e` 在地址切换前后都可以被正确地解引用呢？

答：因为当前是运行在系统内核中的，所有的进程 `env_pgdir` 的高地址的映射跟 `kern_pgdir` 的是一样的，每个进程页表中虚拟地址高于 `UTOP` 之上的地方，只有 `UVPT` 不一样，其余的都是一样的，只不过在用户态下是看不到的。所以虽然这个时候的页表换成了下一个要运行的进程的页表，但是 `curenv` 的地址没变，映射也没变。

问题 4:

无论何时，内核在从一个进程切换到另一个进程时，它应当确保旧的寄存器被保存，以使得以后能够恢复。为什么？在哪里实现的呢？

答：因为通过保存状态，才能保证下次恢复到正确的环境，否则不知道该从哪里进行，通过 `trap()` 把用户态陷入内核态时候，`curenv->env_tf = *tf` 这句话把状态保存到用户环境 `env_tf` 中。

练习 7:

sys_exofork():

该函数的目的是为了创建子进程，如果错误了返回错误码，如果在父进程中，返回子进程 id，如果是子进程调用返回 0，在创造过程中，寄存器等于当前进程，状态改为 not runnable，通过把 `eax` 该为 0，系统调用返回值为 0。代码如下：

```
int32_t result;
if(result = env_alloc(&newenv, curenv->env_id) < 0)
    return result;
newenv->env_status = ENV_NOT_RUNNABLE;
newenv->env_tf = curenv->env_tf;
newenv->env_tf.tf_regs.reg_eax = 0;
return newenv->env_id;
```

sys_env_set_status():

该函数的目的就是更改制定 id 进程的状态，所以先用 `envid2env()` 转换 id 成进程，随后通过权限 1 以及判定目的是否处于可更改状态 runnable 或者 not_unrunnable。代码如下：

```
struct Env *e;
if (envid2env(envid, &e, 1))
    return -E_BAD_ENV;

if (status != ENV_NOT_RUNNABLE && status != ENV_RUNNABLE)
    return -E_INVALID;

e->env_status = status;
return 0;
// LAB 4: Your code here.
```

sys_page_alloc():

该函数目的就是分配物理页，并且分配虚拟内存映射，首先我们将要分配的进程 id 转换为具体的环境，方法同上，其次要求必要权限是 `PTE_U|PTE_P`，如果权限不够，返回 `-E_INVALID`，如果都成功，用 `page_alloc()` 分配，如果没有内存，返回 `-E_NO_MEM`。如果 insert 没有成功，free 掉 page。代码如下：

```

struct Env *e;
if (envid2env(envid, &e, 1))
    return -E_BAD_ENV;

if (status != ENV_NOT_RUNNABLE && status != ENV_RUNNABLE)
    return -E_INVAL;

e->env_status = status;
return 0;
// LAB 4: Your code here.

```

sys_page_map():

该函数目的是为了两个进程共享内存，首先通过 id 寻找到两个进程，随后跟前面函数一样，直到通过 lookup 找到确定的一个内存，把 dst 用 insert 存入。代码如下：

```

struct Env *srcenv, *dstenv;
if (envid2env(srcenvid, &srcenv, 1) || envid2env(dstenvid, &dstenv, 1))
    return -E_BAD_ENV;

if (srcva >= (void *)UTOP || dstva >= (void *)UTOP || PGOFF(srcva) || PGOFF(dstva))
    return -E_INVAL;

pte_t *pte;
struct PageInfo *p = page_lookup(srcenv->env_pgdir, srcva, &pte);
if (!p)
    return -E_INVAL;

int valid_perm = (PTE_U|PTE_P);
if ((perm&valid_perm) != valid_perm)
    return -E_INVAL;

if ((perm & PTE_W) && !(*pte & PTE_W))
    return -E_INVAL;

int result = page_insert(dstenv->env_pgdir, p, dstva, perm);
return result;

```

sys_page_unmap():

该函数目的是释放映射关系，依旧先通过具体的 id 找到环境，然后用 remove，释放掉，代码如下：

```

struct Env *e;
if (envid2env(envid, &e, 1))
    return -E_BAD_ENV;

if (va >= (void *)UTOP)
    return -E_INVAL;

page_remove(e->env_pgdir, va);
return 0;
// LAB 4: Your code here

```

随后，将函数对应到 syscall 里，代码如下：

```

case SYS_exofork:
    return sys_exofork();
case SYS_env_set_status:
    return sys_env_set_status(a1, a2);
case SYS_page_alloc:
    return sys_page_alloc(a1, (void *)a2, a3);
case SYS_page_map:
    return sys_page_map(a1, (void*)a2, a3, (void*)a4, a5);
case SYS_page_unmap:
    return sys_page_unmap(a1, (void *)a2);
case SYS_cpusts:

```

至此，part A 全部完成，make grade 后结果：

```

dumbfork: 0x (1.8s)
(Old jos.out.dumbfork failure log removed)
Part A score: 5/5

```

PART B

练习 8:

该题主要是实现函数 `sys_env_set_pgfault_upcall()`，只需要找到对应环境，把他的 `env_pgfault_upcall` 改成 `func` 即可，代码如下：

```

static int
sys_env_set_pgfault_upcall(envid_t envid, void *func)
{
    // LAB 4: Your code here.
    struct Env *e;
    if (envid2env(envid, &e, 1))
        return -E_BAD_ENV;

    e->env_pgfault_upcall = func;
    return 0;
}

```

随后，添加系统调用。

练习 9:

该函数根据指导书中所写，应该分为两种情况，一种是正常情况下的，会导致内核构造一个 `UTrapframe`，随后陷入异常处理栈；另一种情况是已经在异常栈，那么会在自己的 `esp` 上构造，但首先要先压入一个 32 位空值，也就是 4 字节。随后使用 `user_mem_assert()` 对异常栈进行 `overflow` 的检查，如果发生这种情况，就会发生内核缺页，会进行 `panic`，该函数如果不通过检查，`destroy` 环境，如果是当前环境，则不返回。所以如果通过检查，保存内陷的信息。随后把页面错误处理函数赋值给 `eip`，并切回到用户态处理错误。代码如下：


```

if (curenv->env_pgfault_upcall)
{
    struct UTrapframe *utf;
    if (tf->tf_esp >= UXSTACKTOP-PGSIZE && tf->tf_esp <= UXSTACKTOP-1)
        utf = (struct UTrapframe *) (tf->tf_esp - sizeof(struct UTrapframe) - 4);

    else
        utf = (struct UTrapframe *) (UXSTACKTOP - sizeof(struct UTrapframe));

    user_mem_assert(curenv, (void*)utf, 1, 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;

    curenv->env_tf.tf_eip = (uintptr_t)curenv->env_pgfault_upcall;
    curenv->env_tf.tf_esp = (uintptr_t)utf;
    env_run(curenv);
}

```

练习 10:

对于这部分代码，根据注释，可知之前嵌套中留下 4 字节，就是为了在每一个 UTrapframe 栈顶存储 eip，方便这里返回，所以只需要在常规用户栈减去 4 字节，把这个 eip 取出来；随后依次还原通用寄存器和 eflags；将栈顶弹到 esp，此时 esp 存储的就是 eip；弹出栈顶元素并返回。代码如下：

```

// LAB 4: Your code here.
movl 0x28(%esp), %ebx
subl $0x4, 0x30(%esp)
movl 0x30(%esp), %eax |
movl %ebx, (%eax)
addl $0x8, %esp

// Restore the trap-time registers. After you do this, you
// can no longer modify any general-purpose registers.
// LAB 4: Your code here.
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.
popl %esp
// Return to re-execute the instruction that faulted.
// LAB 4: Your code here.
ret

```

练习 11:

这部分代码很简单，主要实现了先分配一个异常栈，然后调用 `sys_env_set_pgfault_upcall()`，如果失败，进行简单的报错，代码如下：

```

if (_pgfault_handler == 0) {
    // First time through!
    // LAB 4: Your code here.
    if (sys_page_alloc(0, (void*)(UXSTACKTOP - PGSIZE), PTE_W|PTE_U|PTE_P))
        panic("set_pgfault_handler page_alloc failed");

    if (sys_env_set_pgfault_upcall(0, _pgfault_upcall)) {
        panic("set_pgfault_handler set_pgfault_upcall failed");
    }
    //panic("set_pgfault_handler not implemented");
}

```

几个结果如下：

make run-faultread:

```

[00000000] new env 00001000
[00001000] user fault va 00000000 ip 00800039
TRAP frame at 0xf0291000 from CPU 0
edi 0x00000000
esi 0x00000000
ebp 0xeebdfdf0
oesp 0xefffffdc
ebx 0x00000000
edx 0x00000000
ecx 0x00000000
eax 0xeec00000
es 0x---0023
ds 0x---0023
trap 0x0000000e Page Fault
cr2 0x00000000
err 0x00000004 [user, read, not-present]
elp 0x00800039
cs 0x---001b
flag 0x00000096
esp 0xeebdfdb8
ss 0x---0023
[00001000] free env 00001000

```

make run-faultdie:

```

[00000000] new env 00001000
i faulted at va deadbeef, err 6
[00001000] exiting gracefully
[00001000] free env 00001000

```

make run-faultalloc:

```

[00000000] new env 00001000
fault deadbeef
this string was faulted in at deadbeef
fault cafebffe
fault cafec000
this string was faulted in at cafebffe
[00001000] exiting gracefully
[00001000] free env 00001000

```

make run-faultallocbad:

```

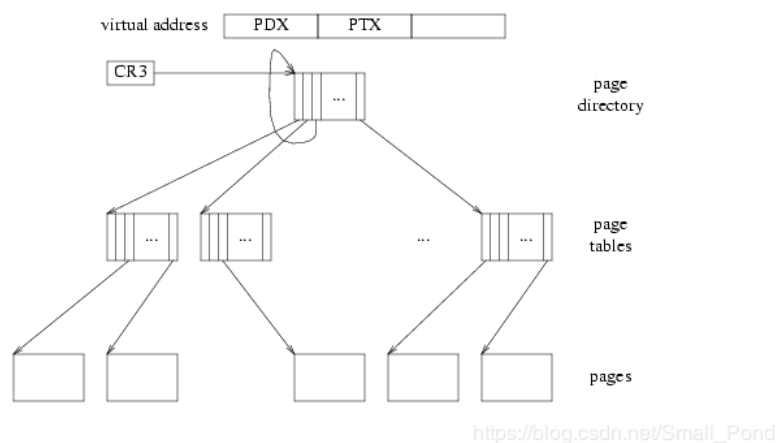
[00000000] new env 00001000
[00001000] user_mem_check assertion failure for va deadbeef
[00001000] free env 00001000
No runnable environments in the system!

```

至于为什么 faultalloc 和 faultallocbad 为什么会有不同，通过看了测试代码，发现唯一不同在于前者用了 printf，后者用的是 sys_puts。而 printf 虽然最后也会调用 sys_puts，但他会先讲信息存入到缓存区，这是已经访问错误地址，而不会等到 sys_puts 再调用 user_mem_check 检查内存地址的正确性。所以，输出会有区别。

练习 12:

通过理解 UVPD 和 UVPT 大概理解为，正常寻址可以寻址到具体的值，但是通过这两种寻址，通过一个指向自己的指针，消耗 CR3 的寻址，最后只能寻找到页目录或者二级页表。所以我们通过这种方式对以下函数进行更改。



pgfault() :

首先按照注释，要求检查缺页操作是不是写操作，以及是否标记为 COW，如果不是就 panic。如果通过检查，分配一个新的页面，随后对这个页面进行指导书上的更改，代码如下：

```
// LAB 4: Your code here.
if (!(err & FEC_WR) && (uvpd[PDX(addr)] & PTE_P)
    && (uvpt[PGNUM(addr)] & PTE_COW) && (uvpt[PGNUM(addr)] & PTE_P)))
    panic("page cow check failed");

addr = ROUNDDOWN(addr, PGSIZE);
// 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.

// LAB 4: Your code here.
if ((r = sys_page_alloc(0, PFTEMP, PTE_P|PTE_U|PTE_W)))
    panic("sys_page_alloc: %e", r);

memmove(PFTEMP, addr, PGSIZE);

if ((r = sys_page_map(0, PFTEMP, 0, addr, PTE_P|PTE_U|PTE_W)))
    panic("sys_page_map: %e", r);

if ((r = sys_page_unmap(0, PFTEMP)))
    panic("sys_page_unmap: %e", r);
... ..
```

duppage() :

该函数的目的就是通过判断旧页面是否是 write 或者 woc 后，将新的页面改成 woc，同时再将旧的页面也改成 woc。代码如下：

```

..
void *addr = (void *) (pn * PGSIZE);
if (uvpt[pn] & (PTE_W|PTE_COW))
{
    if ((r = sys_page_map(0, addr, envid, addr, PTE_COW|PTE_U|PTE_P)) < 0)
        panic("sys_page_map COW:%e", r);

    if ((r = sys_page_map(0, addr, 0, addr, PTE_COW|PTE_U|PTE_P)) < 0)
        panic("sys_page_map COW:%e", r);
}
else
    if ((r = sys_page_map(0, addr, envid, addr, PTE_U|PTE_P)) < 0)
        panic("sys_page_map UP:%e", r);
//panic("duppage not implemented");
return 0;

```

fork():

按照指导书上步骤，依次完成程序：利用 `set_page_fault()` 创建 `pgfault()`；然后利用 `sys_exofork()` 创建子进程，随后遍历每一个页面，调用 `duppage()`，因为在该函数已经相信测定，再分配页面和创建异常栈，随后把状态改为 `runnable`。代码如下：

```

..
envid_t
fork(void)
{
    // LAB 4: Your code here.
    set_pgfault_handler(pgfault);

    envid_t envid = sys_exofork();
    uint8_t *addr;
    if (envid < 0)
        panic("sys_exofork:%e", envid);
    if (envid == 0) {
        thisenv = &envs[ENVX(sys_getenvid())];
        return 0;
    }

    extern unsigned char end[];
    for (addr = (uint8_t *) UTEXT; addr < end; addr += PGSIZE) {
        if ((uvpd[PDX(addr)] & PTE_P) && (uvpt[PGNUM(addr)] & PTE_P)
            && (uvpt[PGNUM(addr)] & PTE_U)) {
            duppage(envid, PGNUM(addr));
        }
    }

    duppage(envid, PGNUM(ROUNDDOWN(&r, PGSIZE)));

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

    extern void pgfault_upcall();
    sys_env_set_pgfault_upcall(envid, pgfault_upcall);

    if ((r = sys_env_set_status(envid, ENV_RUNNABLE))
        < 0)
        panic("sys_env_set_status:%e", r);

    return envid;
    //panic("fork not implemented");
}

```

利用 `make run-forktree`，进行测试，结果如下：

```

2001: I am '000'
[00002001] exiting gracefully
[00002001] free env 00002001
1002: I am '1'
[00001002] new env 00003001
[00001002] new env 00003000
[00001002] exiting gracefully
[00001002] free env 00001002
3000: I am '11'
[00003000] new env 00002002
[00003000] new env 00001005
[00003000] exiting gracefully
[00003000] free env 00003000
3001: I am '10'
[00003001] new env 00004000
[00003001] new env 00001006
[00003001] exiting gracefully
[00003001] free env 00003001
4000: I am '100'
[00004000] exiting gracefully
[00004000] free env 00004000
2002: I am '110'
[00002002] exiting gracefully
[AAAA7AA2] free env AAAA7AA2

```

最后，利用 make grade 进行测试，完成 part B，结果如下：

```
Faultread: OK (1.9s)
Faultwrite: OK (4.9s)
Faultdie: OK (2.0s)
Faultregs: OK (2.5s)
Faultalloc: OK (1.8s)
Faultallocbad: OK (2.4s)
Faultnostack: OK (2.3s)
Faultbadhandler: OK (1.8s)
Faultevilhandler: OK (1.4s)
Forktree: OK (2.3s)
(Old jos.out.forktree failure log removed)
Part B score: 50/50
```

PART C

练习 13:

对于 trapentry.S 和 trap.c 处理的办法类似于 LAB 3 中的处理方式，代码如下：

```
TRAPHANDLER_NOEC(timer_handler, IRQ_OFFSET + IRQ_TIMER);
TRAPHANDLER_NOEC(kbd_handler, IRQ_OFFSET + IRQ_KBD);
TRAPHANDLER_NOEC(serial_handler, IRQ_OFFSET + IRQ_SERIAL);
TRAPHANDLER_NOEC(spurious_handler, IRQ_OFFSET + IRQ_SPURIOUS);
TRAPHANDLER_NOEC(ide_handler, IRQ_OFFSET + IRQ_IDE);
TRAPHANDLER_NOEC(error_handler, IRQ_OFFSET + IRQ_ERROR);

/*
 *
 */

void dblflt_handler();
void timer_handler();
void kbd_handler();
void serial_handler();
void spurious_handler();
void ide_handler();
void error_handler();

SETGATE(idt[IRQ_OFFSET + IRQ_TIMER], 0, GD_KT, timer_handler, 3);
SETGATE(idt[IRQ_OFFSET + IRQ_KBD], 0, GD_KT, kbd_handler, 3);
SETGATE(idt[IRQ_OFFSET + IRQ_SERIAL], 0, GD_KT, serial_handler, 3);
SETGATE(idt[IRQ_OFFSET + IRQ_SPURIOUS], 0, GD_KT, spurious_handler, 3);
SETGATE(idt[IRQ_OFFSET + IRQ_IDE], 0, GD_KT, ide_handler, 3);
SETGATE(idt[IRQ_OFFSET + IRQ_ERROR], 0, GD_KT, error_handler, 3);
// Do CPU setup
```

随后修改 kern/env.c 中 env_alloc(), IDT 表项中的每一项都初始化为中断门，这样在发生任何中断/异常的时候，陷入内核态的时候，CPU 都会将%eflags 寄存器上的 FL_IF 标志位清 0，关闭中断；切换回用户态的时候，CPU 将内核栈中保存的%eflags 寄存器弹回%eflags 寄存器，恢复

原来的状态。所以在 env.c 中找到一句注释 Enable interrupts while in user mode. 所以在这个地方把 eflags 改为 FL_IF, 代码如下:

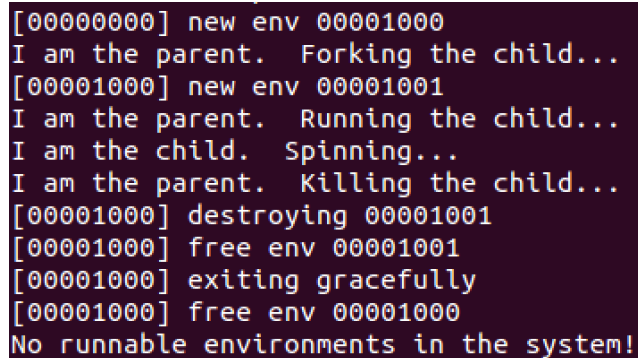
```
// Enable interrupts while in user mode.
// LAB 4: Your code here.s
e->env_tf.tf_eflags |= FL_IF;
```

练习 14:

这个也等同于以前, 直接加一个判断语句即可, 代码如下:

```
if (tf->tf_trapno == IRQ_OFFSET + IRQ_TIMER) {
    lapic_eoi();
    sched_yield();
    return;
}
```

随后运行 spin, 成功结束程序, 结果如图:



```
[00000000] new env 00001000
I am the parent. Forking the child...
[00001000] new env 00001001
I am the parent. Running the child...
I am the child. Spinning...
I am the parent. Killing the child...
[00001000] destroying 00001001
[00001000] free env 00001001
[00001000] exiting gracefully
[00001000] free env 00001000
No runnable environments in the system!
```

随后测试 forktree, 没有杀死正常进程。

练习 15:

sys_ipc_recv():

根据注释我们可以知道, 如果 dstva < UTOP, 但是 dstva 没有页对齐, 返回 -E_INVAL。如果满足, 更改 env_ipc_recving 和 env_ipc_dstva, 并把状态改成 not runnable。代码如下:

```
if (dstva < (void *)UTOP && ((unsigned)dstva % PGSIZE))
    return -E_INVAL;
```

```
curenv->env_ipc_recving = 1;
curenv->env_ipc_dstva = dstva;
curenv->env_status = ENV_NOT_RUNNABLE;
sched_yield();
```

sys_ipc_try_send():

首先是判断能否通过 id 找到环境；环境是否愿意接受；如果 srcva < UTOP，判断是否对齐，然后开始寻找页面，然后判断页面是否存在，权限的问题，如果一切都可以，那么用 insert 把这个页面加到指定环境中去。然后更新环境数据。代码如下：

```
static int
sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned perm)
{
    // LAB 4: Your code here.
    struct Env *e;
    if (envid2env(envid, &e, 0))
        return -E_BAD_ENV;

    if (!e->env_ipc_recving)
        return -E_IPC_NOT_RECV;

    if (srcva < (void *) UTOP)
    {
        if (PGOFF(srcva))
            return -E_INVALID;

        pte_t *pte;
        struct PageInfo *p = page_lookup(curenv->env_pgdir, srcva, &pte);
        if (!p)
            return -E_INVALID;

        if ((*pte & perm) != perm)
            return -E_INVALID;

        if ((perm & PTE_W) && !(*pte & PTE_W))
            return -E_INVALID;

        if (e->env_ipc_dstva < (void *)UTOP)
        {
            int ret = page_insert(e->env_pgdir, p, e->env_ipc_dstva, perm);
            if (ret)
                return ret;
            e->env_ipc_perm = perm;
        }
    }

    e->env_ipc_recving = 0;
    e->env_ipc_from = curenv->env_id;
    e->env_ipc_value = value;
    e->env_status = ENV_RUNNABLE;
    e->env_tf.tf_regs.reg_eax = 0;
    return 0;
}
```

ipc_send():

根据注释，如果是空页面，我们可以传递 UTOP，这样 sys_ipc_try_send() 就不会传递页面。随后进入循环，不停调用 sys_ipc_try_send()，直到收到刀，如果收到错误信息回复，则 panic。代码如下：

```
if (pg == NULL)
    pg = (void *)UTOP;

int ret;
while ((ret = sys_ipc_try_send(to_env, val, pg, perm)))
{
    if (ret != -E_IPC_NOT_RECV)
        panic("ipc_send error %e", ret);
    sys_yield();
}
```

ipc_recv():

同样，如果是空页，传递 UTOP，如果接受成功，更新数据，如果最后 from_env_store 和 perm_store 不为空，那么存储发送方的数据，最后又数据返回数据，没数据返回 0。代码如下：

```
int32_t
ipc_rcv(envid_t *from_env_store, void *pg, int *perm_store)
{
    // LAB 4: Your code here.
    if (pg == NULL)
        pg = (void *)UTOP;

    int r = sys_ipc_rcv(pg);
    int from_env = 0, perm = 0;
    if (r == 0)
    {
        from_env = thisenv->env_ipc_from;
        perm = thisenv->env_ipc_perm;
        r = thisenv->env_ipc_value;
    }
    else
    {
        from_env = 0;
        perm = 0;
    }

    if (from_env_store) *from_env_store = from_env;
    if (perm_store) *perm_store = perm;

    return r;
    //panic("ipc_rcv not implemented");
    //return 0;
}
```

最后，开始测试：
pingpong:

```
[00000000] new env 00001000
[00001000] new env 00001001
send 0 from 1000 to 1001
1001 got 0 from 1000
1000 got 1 from 1001
1001 got 2 from 1000
1000 got 3 from 1001
1001 got 4 from 1000
1000 got 5 from 1001
1001 got 6 from 1000
1000 got 7 from 1001
1001 got 8 from 1000
1000 got 9 from 1001
[00001000] exiting gracefully
[00001000] free env 00001000
1001 got 10 from 1000
[00001001] exiting gracefully
[00001001] free env 00001001
```

primes: 分配到超出整个内存

```
CPU 0: 8081 [00001317] new env 00001318
CPU 0: 8087 [000013f8] new env 000013f9
CPU 0: 8089 [000013f9] new env 000013fa
CPU 0: 8093 [000013fa] new env 000013fb
CPU 0: 8101 [000013fb] new env 000013fc
CPU 0: 8111 [000013fc] new env 000013fd
CPU 0: 8117 [000013fd] new env 000013fe
CPU 0: 8123 [000013fe] new env 000013ff
CPU 0: 8147 [000013ff] user panic in <unknown> at lib/fork.c:112: sys_exofork:out of environments
Welcome to the JOS kernel monitor!
```

三、实验结果

最后通过 `make grade` 完成全部实验，结果如下：

```
spin: OK (1.5s)
stresssched: OK (3.5s)
    (Old jos.out.stresssched failure log removed)
sendpage: OK (3.9s)
pingpong: OK (2.1s)
    (Old jos.out.pingpong failure log removed)
primes: OK (5.8s)
    (Old jos.out.primes failure log removed)
Part C score: 25/25

Score: 80/80
```

通过本次试验加深了对抢占式多任务处理的理解。以及通讯和父子进程已经时钟等的认识。