

实验报告 Lab4

10300240039 章超

December 5, 2013

1 Multiprocessor Support and Cooperative Multitasking

第一部分，我们首先要使 JOS 拓展称为多处理器的系统，然后实现新的系统调用，从而使用户可以创建进程（JOS 的 Enviroment），进程调度将使用 round-robin 轮转式调度，在第三部分中再实现抢占式调度。

1.1 Multiprocessor Support

JOS 采用的是 SMP(Symmetric Multi-Processing) 模型：每一个 CPU 对系统资源具有同等的访问能力。CPU 可以分为两类，一类是 BSP(BootStrap Processor)，也就是启动系统的那个处理器，到现在为止，也就是 Lab1 到 Lab3，我们的所有代码都是在 BSP 上运行的，启动时的 BSP，是由硬件和 BIOS 决定的；另一类是 AP(Application Processor)，这些都是由 BSP 激活，然后和 BSP 一起同等工作的。

在 SMP 中，每个 CPU 有一个 LAPIC(Local APIC) 单元。LAPIC 单元的作用包括在系统传递中断信号，提供 CPU 标识符。Lab4 中在下面这几个地方会涉及 LAPIC 单元：

- 读取 LAPIC 标识符 (APIC ID)，知道运行当前代码的 CPU(`cpunum()`)
- BSP 发送 CPU 间中断STARTUP来激活 AP(`lapic_startap()`)
- 在抢占式调度中，我们对 LAPIC 内部的计时器进行编程 (`apic_init()`)

这些函数都实现好了，我们只要看懂大概意思就行了。

LAPIC 还采用了 MMIO(Memory-mapped I/O) 技术，一部分物理内存硬连接到 IO 设备的寄存器上。我们其实已经将 0xA0000 的物理地址代表的 I/O Hole 作为 CGA 的输出 Buffer 了。而 LAPIC 硬链接到物理内存的 0xFE000000(LAPIC Hole)，此处离 4GB 仅有剩 32MB，物理地址太高了，KERNBASE 处的直接映射已经不可能了。所以我们要建立它的内存映射。

Exercise 1 Implement `mmio_map_region` in `kern/pmap.c`. To see how this is used, look at the beginning of `lapic_init` in `kern/lapic.c`. You'll have to do the next exercise, too, before the tests for `mmio_map_region` will run.

`lapic_init`中为 LAPIC 分配了一页。（这里用 PGSIZE 要比直接用 4096 硬编码要好）。

kern/lapic.c

```
lapic = mmio_map_region(lapicaddr, 4096);
```

我们对于mmio_map_region的实现如下:

kern/pmap.c

```
void *
mmio_map_region(physaddr_t pa, size_t size)
{
    static uintptr_t base = MMIIOBASE;
    size = ROUNDUP(size, PGSIZE);
    if (base + size > MMIOLIM)
        panic("mmio_map_region: Out of memory size");
    boot_map_region(kern_pgdir, base, size, pa, PTE_W | PTE_PCD | PTE_PWT);
    void *ret = (void*)base;
    base += size;
    return ret;
}
```

这里需要将base更新, 以便后续可以继续分配 (实际上 Lab4 里面就没有后续分配了)。然后按照注释提示, 调用boot_map_region就可以了。唯一需要注意的就是要将权限设为 Cache-disable 和 Write-through, 这样在这部分虚拟空间上的操作就会立即生效。

1.2 Application Processor Bootstrap

在 AP 跑起来之前, BSP 会事先从 BIOS 搜集一些系统多处理器的信息, 例如 CPU 的个数, 它们的 APIC ID, LAPIC 单元的 MIMO 地址。参见kern/mpconfig.c中的mp_init()函数。kern/init.c中的boot_aps()函数激活 AP, AP 也是从实模式开始运行。所以需要将 AP 的入口代码 (kern/mpentry.S) 复制到内存中实模式可以寻址的位置, 这里是0x7000(MPENTRY_PADDR)。现在, boot_aps()函数将 AP 依次激活: 它向 AP 的 LAPIC 发送STARTUP中断, 以及起始的CS:IP地址MPENTRY_PADDR。AP 的起始代码在kern/mpentry.S中, 这和我们在boot/boot.S中看到的代码非常相似。在完成一些初始操作以后, AP 开启分页, 进入保护模式, 然后调用mp_main()函数。这时, BSP 的boot_aps()函数等待来自被激活 AP 的CPU_STARTED信号, 然后才激活下一个 AP。

Exercise 2 Read boot_aps() and mp_main() in kern/init.c, and the assembly code in kern/mpentry.S. Make sure you understand the control flow transfer during the bootstrap of APs. Then modify your implementation of page_init() in kern/pmap.c to avoid adding the page at MPENTRY_PADDR to the free list, so that we can safely copy and run AP bootstrap code at that physical address. Your code should pass the updated check_page_free_list() test (but might fail the updated check_kern_pgdir() test, which we will fix soon).

这个练习要求我们阅读kern/init.c中的boot_aps(), mp_main()函数, 以及kern/mpentry.S中的汇编代码。然后在kern/pmap.c的page_init()添加一些代码, 使 AP 启动代码的物理内存区域 (MPENTRY_PADDR) 避免被加入空闲页框链表。

首先，kern/init.c中的boot_aps()函数。这个函数是被 BSP 调用，首先将代码拷贝到MPENTRY_PADDR，然后依次启动 CPU，设置他们的栈，调用lapic_start并等待CPU_STARTED信号。其次，kern/mpentry.S中的汇编代码。这与boot/boot.S几乎一样，区别是一个没有开启 A20，另一个是用MPBOOTPHYS来计算符号的地址。这里面有一个问题，

kern/mpentry.S

```
# Call mp_main(). (Exercise for the reader: why the indirect call?)
movl    $mp_main, %eax
call    *%eax
```

网上搜索了好久，都没有找到答案，照理来说%eax也不会在函数中被用到。所以这样的调用确实比较奇怪。再次，kern/init.c中的mp_main()函数。这个函数是 AP 的第一个 C 函数，这里，AP 初始化它的页表，初始化 LAPIC 单元，初始化 Env，中断等等。然后调用sched_yield，寻找可以执行的进程。最后，page_init函数，我们修改如下，这一过程比较简单，就是在 Lab2 的基础上再禁止MPENTRY_PADDR代表的物理页的分配。

kern/pmap.c

```
static void init_use_page(size_t page) {
    pages[page].pp_ref = 1;
    pages[page].pp_link = NULL;
}

static void init_free_page(size_t page) {
    pages[page].pp_ref = 0;
    pages[page].pp_link = page_free_list;
    page_free_list = &pages[page];
}

void
page_init(void)
{
    size_t i, iop, ext, cur, mp;
    iop = PGNUM(IOPHYSMEM);
    ext = PGNUM(EXTPHYSMEM);
    cur = PGNUM(PADDR(boot_alloc(0)));
    mp = PGNUM(MPENTRY_PADDR);

    for (i = 0; i < npages; i++) {
        if (i == 0)
            init_use_page(i);
        else if (iop <= i && i < ext)
            init_use_page(i);
        else if (ext <= i && i < cur)
            init_use_page(i);
        else if (i == mp)
            init_use_page(i);
        else
            init_free_page(i);
    }
}
```

```
}  
}
```

Question 1 Compare `kern/mpentry.S` side by side with `boot/boot.S`. Bearing in mind that `kern/mpentry.S` is compiled and linked to run above `KERNBASE` just like everything else in the kernel, what is the purpose of macro `MPBOOTPHYS`? Why is it necessary in `kern/mpentry.S` but not in `boot/boot.S`? In other words, what could go wrong if it were omitted in `kern/mpentry.S`? Hint: recall the differences between the link address and the load address that we have discussed in Lab 1.

宏的定义如下:

```
#define MPBOOTPHYS(s) ((s) - mentry_start + MPENTRY_PADDR)
```

它可以计算出一个内核线性地址`s`对应的物理地址。`boot/boot.S`里没有, 是因为它跑在实模式下, 直接对应物理地址。`kern/entry.S`里面没有, 是因为代码中使用了简单页表, 使得`KERNBASE`以上的线性地址被映射到了物理地址上 (即`[KERNBASE, KERNBASE+4MB)=>[0, 4MB)`)。而对于 AP 来说, 我们没有使用简单页表, 所以就需要`MPBOOTPHYS`来计算绝对地址 (物理地址), 而不是让编译器去搞定。

1.2.1 Per-CPU State and Initialization

CPU 之间需要做区分, `kern/cpu.h`中定义了`CpuInfo`结构来表示一个 CPU, `cpunum()` 函数用于返回调用这个函数的当前 CPU 序号, `cpus`用于存储所有 CPU 的信息, `thiscpu` 宏用于表示当前 CPU 的`CpuInfo`结构。每个 CPU 的状态有以下信息:

- Kernel 栈, 储存在`percpu_kstacks`。
- TSS 和 TSS 描述项, 储存在`CpuInfo.cpu_ts`中。
- 当前运行进程的指针, 储存在`CpuInfo.cpu_env`中。
- CPU 的系统寄存器。

Exercise 3 Modify `mem_init_mp()` (in `kern/pmap.c`) to map per-CPU stacks starting at `KSTACKTOP`, as shown in `inc/memlayout.h`. The size of each stack is `KSTKSIZE` bytes plus `KSTKGAP` bytes of unmapped guard pages. Your code should pass the new check in `check_kern_pgdir()`.

这个练习要求修改`mem_init_mp()`函数, 为每一个 CPU 映射 Kernel 栈空间。另外栈空间之间还有保护页作为间隔。代码如下:

`kern/pmap.c`

```
static void  
mem_init_mp(void)  
{  
    int kstacktop_i;  
    int i;  
    for (i = 0; i < NCPU; ++i) {
```

```

    kstacktop_i = KSTACKTOP - i * (KSTKSIZE + KSTKGAP);
    boot_map_region(kern_pgdir, kstacktop_i - KSTKSIZE, KSTKSIZE, PADDR(
    percpu_kstacks[i]), PTE_W | PTE_P);
}
}

```

Exercise 4 The code in `trap_init_percpu()` (`kern/trap.c`) initializes the TSS and TSS descriptor for the BSP. It worked in Lab 3, but is incorrect when running on other CPUs. Change the code so that it can work on all CPUs. (Note: your new code should not use the global `ts` variable any more.)

这个练习要求修改`trap_init_percpu()`，使它能设置每个 CPU 的 TSS 进行设置，包括设置 TSS 的 Kernel 栈指针。其实参考注释，一步一步来就可以了。代码如下：

kern/trap.c

```

void
trap_init_percpu(void)
{
    int i = thiscpu->cpu_id;
    thiscpu->cpu_ts.ts_esp0 = KSTACKTOP - i * (KSTKSIZE + KSTKGAP);
    thiscpu->cpu_ts.ts_ss0 = GD_KD;
    gdt[(GD_TSS0 >> 3) + i] = SEG16(STS_T32A, (uint32_t) &(thiscpu->cpu_ts),
    sizeof(struct Taskstate), 0);
    gdt[(GD_TSS0 >> 3) + i].sd_s = 0;
    ltr((GD_TSS0 + (i << 3)) & ~0x7);
    lidt(&idt_pd);
}

```

1.3 Locking

我们需要确保只有一个 CPU 在运行 Kernel 代码。所以我们采用了 Big Kernel Lock 来限制 Kernel 代码的运行。另外 JOS 还提供了`lock_kernel`和`unlock_kernel`这两个函数来简化我们的操作。

Exercise 5 Apply the big kernel lock as described above, by calling `lock_kernel()` and `unlock_kernel()` at the proper locations.

这个练习就是要求我们实现 Big Kernel Lock，不过其实提示的已经非常明显了。我们只需要记住，在`env_run()`里面，也就是重新回到 User Mode 时，需要释放锁；其余进入 Kernel Mode 时，需要加上锁。代码如下：

kern/init.c

```

void
i386_init(void)
{

```

```

extern char edata[], end[];
memset(edata, 0, end - edata);
cons_init();
cprintf("6828 decimal is %o octal!\n", 6828);
mem_init();
env_init();
trap_init();
mp_init();
lapic_init();
pic_init();
lock_kernel();
boot_aps();
#if defined(TEST)
    ENV_CREATE(TEST, ENV_TYPE_USER);
#else
    ENV_CREATE(user_priorityhigh, ENV_TYPE_USER);
    ENV_CREATE(user_prioritynormal, ENV_TYPE_USER);
    ENV_CREATE(user_prioritylow, ENV_TYPE_USER);
#endif
    sched_yield();
}

```

kern/init.c

```

void
mp_main(void)
{
    lcr3(PADDR(kern_pgdir));
    cprintf("SMP: CPU %d starting\n", cpunum());
    lapic_init();
    env_init_percpu();
    trap_init_percpu();
    xchg(&thiscpu->cpu_status, CPU_STARTED);
    lock_kernel();
    sched_yield();
}

```

kern/trap.c

```

void
trap(struct Trapframe *tf)
{
    asm volatile("cld" ::: "cc");
    extern char *panicstr;
    if (panicstr)
        asm volatile("hlt");
    if (xchg(&thiscpu->cpu_status, CPU_STARTED) == CPU_HALTED)
        lock_kernel();
    assert(!(read_eflags() & FL_IF));
}

```

```

if ((tf->tf_cs & 3) == 3) {
    lock_kernel();
    assert(curenv);
    ...
}
...
}

```

kern/env.c

```

void
env_run(struct Env *e)
{
    if (curenv != e) {
        if (curenv && curenv->env_status == ENV_RUNNING)
            curenv->env_status = ENV_RUNNABLE;
        curenv = e;
        e->env_status = ENV_RUNNING;
        lcr3(PADDR(e->env_pgdir));
    }
    e->env_runs ++;
    unlock_kernel();
    env_pop_tf(&(e->env_tf));
}

```

Question 2 It seems that using the big kernel lock guarantees that only one CPU can run the kernel code at a time. Why do we still need separate kernel stacks for each CPU? Describe a scenario in which using a shared kernel stack will go wrong, even with the protection of the big kernel lock.

显然我们需要给每个 CPU 一个 Kernel 栈，这是因为两个 CPU 可能几乎同时发生中断，这样就会把 Trapframe 压入 Kernel 栈，但是由于我们的 `lock_kernel()` 加在了 `trap()` 里面。所以在中断同时发生的时候，Trapframe 压入栈这一过程就会出现竞争，导致错误。

1.4 Round-Robin Scheduling

Exercise 6 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 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.
...
```

After the `yield` programs exit, there will be no runnable environment in the system, the scheduler should invoke the JOS kernel monitor. If any of this does not happen, then fix your code before proceeding.

这个练习就是要求我们实现`sched_yield()`这一函数。这一函数要求使用 Round-robin 的方式，也就是每个进程轮着跑一遍。实现方式有以下几个要求：

- 需要首先找到一个`ENV_RUNNABLE`的进程，然后通过`env_run()`来运行它。
- 永远不要让两个 CPU 运行同一个进程，可以通过识别`ENV_RUNNING`来判断是否有 CPU 在运行这个进程。
- User Mode 通过`sys_yield()`来调用`sched_yield()`

我们实现如下：

kern/sched.c

```
void
sched_yield(void)
{
    struct Env *idle;
    int i;
    uint32_t min_runs = 0xffffffff;
    int min_choice = -1;
    if (thiscpu->cpu_env) {
        if (env_priority_enabled) {
            for (i = 0; i < NENV; i++)
                if (envs[i].env_status == ENV_RUNNABLE) {
                    if ((envs[i].env_runs >> envs[i].env_priority) < min_runs) {
                        min_runs = envs[i].env_runs >> envs[i].env_priority;
                        min_choice = i;
                    }
                }
        }
        if (min_choice != -1)
            env_run(&envs[min_choice]);
        if (thiscpu->cpu_env->env_status == ENV_RUNNING)
            env_run(thiscpu->cpu_env);
    } else {
        int cur_id = ENVX(thiscpu->cpu_env->env_id);
        for (i = (cur_id + 1) % NENV; i != cur_id; i = (i + 1) % NENV)
            if (envs[i].env_status == ENV_RUNNABLE)
```


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?

不会。对于用户进程，其虚拟空间中的 Kernel Address Space 是保持不变的，也就是说映射到的永远是同一片物理内存区域。又由于指针 `e` 指向的是 Kernel Address Space 中的 `envs` 数组的一项，所以，在运行 `lcr3()` 前后，`e` 指向的是同一个地址，不会随着进程改变而改变。

Question 4 Whenever the kernel switches from one environment to another, it must ensure the old environment's registers are saved so they can be restored properly later. Why? Where does this happen?

如果寄存器的值变了，那么 User Mode 进程自然是无法正常恢复的。因为 User Mode 下的进程并不会意识到自己什么时候 `ENV_RUNNING`，什么时候 `ENV_RUNNABLE`。所以 Kernel 必须保证每个 User 进程运行的正确性。寄存器值记录在 `Env.env_tf` 中，在 `env_pop_tf()` 中被恢复出来。

1.5 System Calls for Environment Creation

JOS 现在有了多进程支持，我们就可以实现一些必要的系统调用，使得用户进程可以创建、运行子进程。

Unix 系统提供了 `fork()` 系统调用，使得父进程可以创建子进程，父进程的整个地址空间的内容也复制一份给子进程。他们的唯一区别时，父进程中 `fork()` 的返回值为子进程的 ID，子进程中返回值为 0。默认情况下，每个进程的地址空间是私有的。

现在，我们首先要实现一些和 `fork()` 相关的一系列系统调用，也就是这个练习的内容。

Exercise 7 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_INVALID` in that case. Test your JOS kernel with `user/dumbfork` and make sure it works before proceeding.

1.5.1 sys_exofork

这个函数要求我们，用 `env_alloc` 创建一个新的进程，而且要标记为 `ENV_NOT_RUNNABLE`，并且 `Trapframe` 也要和父进程一样。还有有趣的一点时，因为在子进程中返回值为 0，所以必须对子进程的 `TrapFrame` 进行修改。代码如下：

kern/syscall.c

```
static envid_t
sys_exofork(void)
{
    struct Env* e;
    int r;
```

```

if ((r = env_alloc(&e, sys_getenvid())) < 0)
    return r;
e->env_status = ENV_NOT_RUNNABLE;
e->env_tf = curenv->env_tf;
e->env_tf.tf_regs.reg_eax = 0;
return e->env_id;
}

```

这里我们通过`e->env_tf.tf_regs.reg_eax = 0;`来让子进程认为返回值是 0。

1.5.2 sys_env_set_status

这个函数可以设置指定进程的状态为`ENV_RUNNABLE`或`ENV_NOT_RUNNABLE`。父进程将子进程初始化完毕后，需要将子进程标记为`ENV_RUNNABLE`时使用这个函数。代码如下：

kern/syscall.c

```

static int
sys_env_set_status(envid_t envid, int status)
{
    struct Env* env;
    int r;
    if (status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE)
        return -E_INVAL;
    if ((r = envid2env(envid, &env, 1)) < 0)
        return r;
    env->env_status = status;
    return 0;
}

```

1.5.3 sys_page_alloc

这个函数是为 User 进程分配一个物理页框，并将它映射到指定的线性地址。虽然注释写的很清楚，但是注释的分支条件太多，有很多错误代码需要返回。我们完成时需要耐心。代码如下：

kern/syscall.c

```

static int
sys_env_set_status(envid_t envid, int status)
{
    struct Env* env;
    int r;
    if (status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE)
        return -E_INVAL;
    if ((r = envid2env(envid, &env, 1)) < 0)
        return r;
    env->env_status = status;
    return 0;
}

```

1.5.4 sys_page_map

这个函数要求的分支就更多了。功能是将一个地址映射从一个 User 进程的地址空间复制到另一个 User 进程的地址空间。然后，它们的虚拟地址就映射到相同的物理地址了。

函数的实现主要依靠page_lookup()和page_insert()函数。实现如下：

kern/syscall.c

```
static int
sys_page_map(envid_t srcenvid, void *srcva,
             envid_t dstenvid, void *dstva, int perm)
{
    struct Env *srcenv, *dstenv;
    int r;
    pte_t *pte;
    struct PageInfo *pp;
    if (srcva >= (void *)UTOP || ROUNDUP(srcva, PGSIZE) != srcva
        || dstva >= (void *)UTOP || ROUNDUP(dstva, PGSIZE) != dstva)
        return -E_INVAL;
    if (!(perm & PTE_U) || !(perm & PTE_P) || (perm & ~PTE_SYSCALL))
        return -E_INVAL;
    if ((r = envid2env(srcenvid, &srcenv, 1)) < 0)
        return r;
    if ((r = envid2env(dstenvid, &dstenv, 1)) < 0)
        return r;
    pp = page_lookup(srcenv->env_pgdir, srcva, &pte);
    if (pp == NULL || ((perm & PTE_W) && !(*pte & PTE_W)))
        return -E_INVAL;
    if ((r = page_insert(dstenv->env_pgdir, pp, dstva, perm)) < 0)
        return r;
    return 0;
}
```

1.5.5 sys_page_unmap

这个函数将 User 进程中映射到某一给定线性地址的页面解除映射。函数实现主要依靠page_remove()。代码如下：

kern/syscall.c

```
static int
sys_page_unmap(envid_t envid, void *va)
{
    struct Env *env;
    int r;
    if (va >= (void *)UTOP || ROUNDUP(va, PGSIZE) != va)
        return -E_INVAL;
    if ((r = envid2env(envid, &env, 1)) < 0)
        return r;
    page_remove(env->env_pgdir, va);
}
```

```
    return 0;
}
```

写完这些函数以后，还要在`sys_call()`中进行分发。代码如下：

kern/syscall.c

```
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:
    return sys_page_alloc((envid_t)a1, (void *)a2, (int)a3);
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);
```

2 Copy-on-Write Fork

UNIX 提供`fork()`函数来创建子进程，父进程会将其地址空间整个拷贝给子进程。这一拷贝过程往往是`fork()`中最花时间的。而且实际上，`fork()`后面一般都跟着调用`exec()`，所以拷贝给子进程的地址空间指向的地址很少被使用，所以这一拷贝显得不那么必要了。

出于这个考虑，UNIX 后来让父子进程共用地址空间映射，直到其中的一方要修改该地址中的内容。这种机制成为 COW（写时复制，copy on write）。在 COW 机制下，`fork()`只将父进程的地址空间映射拷贝给子进程的地址空间。当两个进程试图写入共享页面时，就会产生一个 Page Fault。这样 UNIX 就知道一个新的，私有的，可以写的页面。这样`fork()`和`exec()`就可以变得更快。可能子进程只需要复制一页就够了。

在 Lab 4 接下来的实验中，我们就需要实现一个写时复制的`fork()`库函数。之所以说是库函数，是因为它在 User 模式下运行。这样做可以让 Kernel 更加轻量，也能更可能工作正常。

2.1 User-level page fault handling

一个 User 级别的 COW 版本的`fork()`，需要知道在写保护页面上的 Page Fault。COW 只是用户级 Page Fault 的一种应用。

设置一个地址空间，其中 Page Fault 代表着某些特定的操作是可以的。例如，我们可以在一开始只分配一页内存作为栈区，随着程序对栈的使用的增加，可能会产生 Page Fault，我们再按程序分配并映射新的页面，这种机制称为“按需分配”。

2.1.1 Setting the Page Fault Handler

Exercise 8 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.

这个练习要求我们为每个进程设置 Page Fault 的回调函数。代码如下：

kern/syscall.c

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

2.1.2 Normal and Exception Stacks in User Environments

在 User 进程正常执行时，它所运行的栈称为 Normal User Stack，它的ESP指向USTACKTOP，栈的线性地址范围是[USTACKTOP - PGSIZE, USTACKTOP - 1]。当 Page Fault 发生时，User 进程所在的栈为 User Exception Stack，它的ESP指向UXSTACKTOP，栈的线性地址范围是[UXSTACKTOP - PGSIZE, UXSTACKTOP - 1]。

2.1.3 Invoking the User Page Fault Handler

Exercise 9 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.

User 进程发生 Page Fault 的处理流程如下：

- User 进程发生 Page Fault，先进入 Kernel Mode，进入`trap()`后进入`page_fault_handler()`。
- `page_fault_handler()`检查是否是 User 进程发生中断，如果是，向异常栈中压入一个`UTrapframe`保存现场信息。
- Kernel 切换道用户异常栈，并让用户程序重新运行。从`curenv->env_pgfault_call`开始执行。
- Page Fault 处理完毕后，回到用户运行栈。用户程序重新运行。

所以这个练习要求我们做的就是第二步，我们的实现如下：

env/trap.c

```
void
page_fault_handler(struct Trapframe *tf)
{
    uint32_t fault_va;
    fault_va = rcr2();
}
```

```

if ((tf->tf_cs & 3) != 3) {
    print_trapframe(tf);
    panic("page_fault_handler: Kernel Page Fault");
}

// LAB 4: Your code here.
if (curenv->env_pgfault_upcall) {
    struct UTrapframe *utf;
    if (UXSTACKTOP - PGSIZE <= tf->tf_esp && tf->tf_esp < UXSTACKTOP)
        utf = (struct UTrapframe*)(tf->tf_esp - sizeof(struct UTrapframe) -
4);
    else
        utf = (struct UTrapframe*)(UXSTACKTOP - sizeof(struct UTrapframe));
    user_mem_assert(curenv, (void*)utf, sizeof(struct UTrapframe), PTE_U |
PTE_W | PTE_P);
    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 = (uint32_t)curenv->env_pgfault_upcall;
    curenv->env_tf.tf_esp = (uint32_t)utf;
    env_run(curenv);
}

// 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);
}

```

这里有一个细节需要注意，就是我们压入UTrapframe时，如果是递归出现的 Page Fault，我们需要隔开 4 个字节。这个在后面会用到。

题干后面的那个问题：如果用户异常栈空间不足，可以在page_fault_handler()继续分配新的页表，但页表的回收就成了一个问题。

2.1.4 User-mode Page Fault Entrypoint

Exercise 10 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.

这个练习要求我们填写lib/pfentry.S中的_pgfault_upcall过程的汇编码，_pgfault_upcall将调用page_fault_handler()。这是因为用户中断处理程序返回到用户进程中时，需要同

时更改`ss:esp`和`eip`。这时，如果先切换`ss:esp`，那么`eip`就不知道应该去哪里找了；又如果先切换回了`eip`，那么下一条指令就执行在用户进程中了。

所以我们的解决方法是：

1) 如果是递归出现的 Page Fault，我们因为多留了 4 个字节。那么我们就可以将上一个 `UTrapframe` 的 `eip` 存储在空隙中，这样只要调用 `ret` 指令就能同时恢复 `ss:esp` 和 `eip`。

2) 如果是第一个 Page Fault。我们还是可以把返回的 `eip` 保存在用户运行栈的栈顶下面 4 个字节，因为用户运行栈下面是没有被使用的，这样还是只要调用 `ret` 指令就可以了。

代码如下：

lib/pfentry.S

```
...
_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 argume
    // LAB 4: Your code here.
    movl 0x30(%esp), %eax
    subl $0x4, %eax
    movl %eax, 0x30(%esp)
    movl 0x28(%esp), %ebx
    movl %ebx, (%eax)
    add $0x8, %esp
    popal
    add $0x4, %esp
    popfl
    popl %esp
    ret
```

Exercise 11 Finish `set_pgfault_handler()` in `lib/pgfault.c`.

这个就是完成 User 库函数。代码如下：

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

    if (_pgfault_handler == 0) {
        // First time through!
        // LAB 4: Your code here.
        if ((r = sys_page_alloc(0, (void *) (UXSTACKTOP - PGSIZE), PTE_U | PTE_W
        | PTE_P)) < 0) {
            panic("set_pgfault_handler: %e\n", r);
        }
    }
}
```



```

    }
    sys_env_set_pgfault_upcall(0, _pgfault_upcall);
}

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

```

2.2 Testing

这里分别解释两个用户程序的区别：

user/faultdie.c

```

void
handler(struct UTrapframe *utf)
{
    void *addr = (void*)utf->utf_fault_va;
    uint32_t err = utf->utf_err;
    cprintf("i faulted at va %x, err %x\n", addr, err & 7);
    sys_env_destroy(sys_getenvid());
}

void
umain(int argc, char **argv)
{
    set_pgfault_handler(handler);
    *(int*)0xDeadBeef = 0;
}

```

这个程序试图写入 0xDeadBeef，产生 Page Fault，然后自行 Destroy。

user/faultallocbad.c

```

void
handler(struct UTrapframe *utf)
{
    int r;
    void *addr = (void*)utf->utf_fault_va;

    cprintf("fault %x\n", addr);
    if ((r = sys_page_alloc(0, ROUNDDOWN(addr, PGSIZE),
        PTE_P|PTE_U|PTE_W)) < 0)
        panic("allocating at %x in page fault handler: %e", addr, r);
    snprintf((char*) addr, 100, "this string was faulted in at %x", addr);
}

void
umain(int argc, char **argv)
{

```

```
set_pgfault_handler(handler);
sys_cputs((char*)0xDEADBEEF, 4);
}
```

这个程序访问地址 0xDEADBEEF 时调用的是 `sys_cputs()`，而 `sys_cputs()` 访问地址前会调用 `user_mem_assert()`，而此时地址没有分配，因此产生 Assertion Failure。

2.3 Implementing Copy-on-Write Fork

我们可以根据 `user/dumbfork.c` 来实现 `fork()`。它的基本流程如下：

1. 父进程调用 `set_pgfault_handler()` 将它的 Page Fault 回调函数设为 `pgfault()`。
2. 父进程调用 `sys_exofork()` 创建子进程。
3. 遍历父进程的地址空间的 `UTOP` 以下的部分，对于每个标记为 COW 或 Writable 的页面，父进程调用 `duppage()` 将该页面映射到子进程的地址空间中，并标记为 COW，并且将父进程自己的地址空间中那个标记 COW。需要注意的是，对于用户异常栈，父进程不能使用 COW 策略。
4. 父进程设置子进程的 Page Fault Handler 的起始地址。
5. 父进程将子进程设置为 `RUNNABLE`。

而 User 进程的 Page Fault Handler 的流程如下：

1. User 进程产生 Page Fault 按照一般的中断流程，进入汇编码 `_pgfault_upcall`。调用进入 `page_fault_handler()` 函数后，调用 `lib/fork.c` 中的 `pgfault()` 函数。
2. `pgfault()` 检查错误是否为 `FEC_WR`，对应地址的页目录项是否为 `PTE_COW`，若不是则 panic。
3. `pgfault()` 分配一张新页面，接着将它映射到一个临时的线性地址，然后将原页面内容复制到新页面，最后将临时页面映射到正确的地址，并标记为相应读写权限，替换掉原有的映射。

Exercise 12 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.

```
1000: I am ''
1001: I am '0'
2000: I am '00'
2001: I am '000'
1002: I am '1'
3000: I am '11'
3001: I am '10'
4000: I am '100'
1003: I am '01'
5000: I am '010'
4001: I am '011'
2002: I am '110'
1004: I am '001'
1005: I am '111'
```

```
1006: I am '101'
```

根据以上思路，我们的实现如下：

lib/fork.c

```
envid_t
fork(void)
{
    envid_t envid;
    uint8_t *addr;
    int r;
    extern void _pgfault_upcall();

    set_pgfault_handler(pgfault);
    if ((envid = sys_exofork()) < 0)
        panic("fork: sys_exofork: %e", envid);
    if (envid == 0) {
        thisenv = &envs[ENVX(sys_getenvid())];
        return 0;
    }
    for (addr = 0; addr < (uint8_t*)(UXSTACKTOP - PGSIZE); addr += PGSIZE) {
        if ((uvpd[PDX(addr)] & PTE_P) && (uvpt[PGNUM(addr)] & PTE_P))
            duppage(envid, (unsigned)addr);
    }
    if ((r = sys_env_set_pgfault_upcall(envid, _pgfault_upcall)) < 0)
        panic("fork: sys_env_set_pgfault_upcall: %e", r);
    if ((r = sys_page_alloc(envid, (void*) UXSTACKTOP - PGSIZE, PTE_U | PTE_W
        | PTE_P)) < 0)
        panic("fork: sys_page_alloc: %e", r);
    if ((r = sys_env_set_status(envid, ENV_RUNNABLE)) < 0)
        panic("fork: sys_env_set_status: %e", r);
    return envid;
}
```

lib/fork.c

```
static int
duppage(envid_t envid, unsigned pn)
{
    int r;
    void *addr = (void *) (pn);
    pte_t pte = uvpt[PGNUM(pn)];
    if (!(pte & PTE_P) || !(pte & PTE_U))
        return -E_INVAL;
    if ((pte & PTE_W) || (pte & PTE_COW)) {
        if ((r = sys_page_map(0, addr, envid, addr, PTE_U | PTE_P | PTE_COW)) <
            0)
            return -E_INVAL;
    }
}
```

```

    panic("duppage: sys_page_map: %e", r);
    if ((r = sys_page_map(0, addr, 0, addr, PTE_U | PTE_P | PTE_COW)) < 0)
        panic("duppage: sys_page_map: %e", r);
} else {
    if ((r = sys_page_map(0, addr, envvid, addr, PGOFF(pte))) < 0)
        panic("duppage: sys_page_map: %e", r);
}
return 0;
}

```

lib/fork.c

```

static void
pgfault(struct UTrapframe *utf)
{
    void *addr = (void *) utf->utf_fault_va;
    uint32_t err = utf->utf_err;
    int r;
    pte_t pte = uvpt[PGNUM((uint32_t)addr)];
    addr = (void*)ROUNDDOWN((uint32_t)addr, PGSIZE);
    if (!(err & FEC_WR))
        panic("pgfault: the faulting access was not a write!");
    if (!(pte & PTE_COW))
        panic("pgfault: the faulting access was not to a copy-on-write page");
    if ((r = sys_page_alloc(0, PFTEMP, PTE_U | PTE_P | PTE_W)) < 0)
        panic("pgfault: sys_page_alloc: %e", r);
    memmove(PFTEMP, addr, PGSIZE);
    if ((r = sys_page_map(0, PFTEMP, 0, addr, PTE_U | PTE_P | PTE_W)) < 0)
        panic("pgfault: sys_page_map: %e", r);
}

```

3 Preemptive Multitasking and Inter-Process communication (IPC)

在 part C, 我们将实现抢占式调度, 实现进程之间通信的机制。

3.1 Clock Interrupts and Preemption

如果有一些不合做的用户进程, 像user/spin.c创建出来的子进程一样, 它们通过死循环, 使它们的父进程和 Kernel 都没有方法取得 CPU 控制权。为了避免这种情况, Kernel 需要能抢占运行中的 User 进程, 这需要 JOS 能够支持外部时钟中断。

3.1.1 Interrupt discipline

IRQ 表示外部中断, 一共有 16 种, 编号为 0 至 15。但是外部中断在 IDT 中的映射不是固定的。kern/picirq.c的pic_init()将 0 到 15 号外部中断映射到 IDT 表项的IRQ_OFFSET至IRQ_OFFSET+15。inc/trap.h中IRQ_OFFSET被定义为 32, 因此外部中断

的 IDT 表项为 32-47。其中，IRQ 0 是时钟中断，因此 IDT[IRQ_OFFSET+0] 中包含了时钟中断处理程序的起始地址。

在 JOS 中，我们做了相应简化。Kernel 进程时，外部设备的中断总是被屏蔽；而 User 进程中总是被开启。这是通过设置 %eflags 的 FL_IF 位来控制的。

Exercise 13 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.

这个练习首先要我们增加 Handler，但是 Lab 3 中我已经添加了所有 256 个中断的 handler，`trap_init()` 中也设置了所有 handler 的 IDT 表项。所以唯一的修改就是开启 FL_IF。

kern/env.c

```
int
env_alloc(struct Env **newenv_store, env_id_t parent_id)
{
    ...
    // Enable interrupts while in user mode.
    // LAB 4: Your code here.
    e->env_tf.tf_eflags |= FL_IF;
    ...
}
```

3.1.2 Handling Clock Interrupts

Exercise 14 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.

我们开启了外部中断，但是还没有处理他们。所以这个练习里，我们需要在`trap_dispatch()`函数中处理相应情况，代码如下：

kern/trap.c

```
static void
trap_dispatch(struct Trapframe *tf)
{
    ...
    // Handle clock interrupts. Don't forget to acknowledge the
    // interrupt using lapic_eoi() before calling the scheduler!
    // LAB 4: Your code here.
    if (tf->tf_trapno == IRQ_OFFSET+IRQ_TIMER) {
        lapic_eoi();
        sched_yield();
        return;
    }
    ...
}
```

3.2 Inter-Process communication (IPC)

之前我们一直集中于操作系统如何保证进程之间的独立性，制造了一种一个进程独占一台机器的假象。操作系统的另一个重要服务，是让进程之间能够通信。UNIX 的管道模型就是一个例子。

3.2.1 IPC in JOS

我们要实现两个系统调用`sys_ipc_recv()`和`sys_ipc_try_send()`，以及库函数中的包装函数`ipc_recv()`和`ipc_send()`，来提供简单的进程间的通信机制。

JOS 中进程之间传递的消息包括两个部分：一个 32 位的值，以及一个可选的单页面映射。后者能够让进程之间一次传递更多的值。

3.2.2 Sending and Receiving Messages

接收消息：User 进程调用`sys_ipc_recv`，然后系统将阻塞当前进程，直到收到消息。当一个进程在等待接收消息时，其他任何进程都可以向它发送消息。

发送消息：发送进程指定接收进程的 ID 和发送的内容，并调用`sys_ipc_try_send`，如果接收进程确实在等待消息，那么就传递消息过去并返回 0；否则返回`-E_IPC_NOT_RECV`表示接收进程当前并不在等待接收消息。

User 库函数`ipc_recv`将`sys_ipc_recv`进行封装，并从当前 User 进程的`Env`结构中查询接收到的值。同样的，User 库函数`ipc_send`则不断地尝试调用`sys_ipc_try_send`直到发送成功。

3.2.3 Transferring Pages

当一个接收进程调用`sys_ipc_recv`并传入一个`UTOP`以下的地址变量`dstva`时，这代表着它希望接收一个页面映射。如果发送者发送了一个页面，那么这个页面将被映射到接收者地址空间的`dstva`。如果这个地址已经映射了一个页面，那么之前的页面将被 `unmap`。

当一个发送进程调用`sys_ipc_try_send`并传入一个UTOP以下的地址变量`srcva`时，这代表着发送者希望发送一个当前被映射在`srcva`的页面给接收者，权限指定为`perm`。如果进程通信成功，那么发送者仍在其地址`srcva`保留原有的页面映射，而接收者将在其地址`dstva`处与发送者共享同一个映射。

如果发送者或接收者的任何一方，并没有表示出希望接收一个页面映射的意愿，那么页面映射将不会被传递。在IPC之后，Kernel将接收者`Env`结构中接收到的页面的权限位`env_ipc_perm`进行设置，否则为0。

3.2.4 Implementing IPC

Exercise 15 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.

这里的实现相对就不那么难了，代码如下：

kern/syscall.c

```
static int
sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned perm)
{
    // LAB 4: Your code here.
    struct Env *e;
    struct PageInfo *page = NULL;
    pte_t *pte;
    if (envid2env(envid, &e, 0) < 0)
        return -E_BAD_ENV;
    if (!e->env_ipc_recving)
        return -E_IPC_NOT_RECV;
    if ((uint32_t) srcva < UTOP && ROUNDDOWN(srcva, PGSIZE) != srcva)
        return -E_INVAL;
    if ((uint32_t) srcva < UTOP && (!(perm & PTE_U) || !(perm & PTE_P) || (perm & ~PTE_SYSCALL)))
        return -E_INVAL;
    if ((uint32_t) srcva < UTOP && (page = page_lookup(curenv->env_pgdir, srcva, &pte)) == NULL)
        return -E_INVAL;
    if ((uint32_t) srcva < UTOP && (perm & PTE_W) && !(*pte & PTE_W))
        return -E_INVAL;
    if ((uint32_t) srcva < UTOP && page_insert(e->env_pgdir, page, e->env_ipc_dstva, perm))
        return -E_INVAL;
}
```

```

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

static int
sys_ipc_recv(void *dstva)
{
    // LAB 4: Your code here.
    if ((uint32_t) dstva < UTOP && ROUNDDOWN(dstva, PGSIZE) != dstva)
        return -E_INVAL;
    curenv->env_ipc_recving = 1;
    curenv->env_ipc_dstva = dstva;
    curenv->env_ipc_from = 0;
    curenv->env_status = ENV_NOT_RUNNABLE;
    sched_yield();
    return 0;
}

...
int32_t
syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t
    a4, uint32_t a5)
{
    ...
    case SYS_env_set_pgfault_upcall:
        return sys_env_set_pgfault_upcall(a1, (void *) a2);
    case SYS_ipc_try_send:
        return sys_ipc_try_send(a1, a2, (void *)a3, a4);
    case SYS_ipc_recv:
        return sys_ipc_recv((void *)a1);
    ...
}

```

lib/ipc.c

```

int32_t
ipc_recv(envid_t *from_env_store, void *pg, int *perm_store)
{
    // LAB 4: Your code here.
    int r = sys_ipc_recv((pg) ? pg : (void *)UTOP);
    if (from_env_store)
        *from_env_store = (r < 0) ? 0 : thisenv->env_ipc_from;
}

```



```

    if (perm_store)
        *perm_store = (r < 0 && (uint32_t) pg < UTOP) ? 0 : thisenv->
env_ipc_perm;
    if (r < 0)
        return r;
    return thisenv->env_ipc_value;
}

void
ipc_send(envid_t to_env, uint32_t val, void *pg, int perm)
{
    // LAB 4: Your code here.
    int r;
    while ((r = sys_ipc_try_send(to_env, val, pg ? pg : (void *) UTOP, perm
)) != 0) {
        if (r != -E_IPC_NOT_RECV)
            panic("ipc_send: error %e", r);
        else
            sys_yield();
    }
}

envid_t
ipc_find_env(enum EnvType type)
{
    int i;
    for (i = 0; i < NENV; i++)
        if (envs[i].env_type == type)
            return envs[i].env_id;
    return 0;
}

```

lib/syscall.c

```

int
sys_env_set_pgfault_upcall(envid_t envid, void *upcall)
{
    return syscall(SYS_env_set_pgfault_upcall, 1, envid, (uint32_t) upcall,
0, 0, 0);
}

int
sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, int perm)
{
    return syscall(SYS_ipc_try_send, 0, envid, value, (uint32_t) srcva, perm,
0);
}

```

```

int
sys_ipc_recv(void *dstva)
{
    return syscall(SYS_ipc_recv, 1, (uint32_t)dstva, 0, 0, 0, 0);
}

```

这里需要注意的是,ipc_send应该不停地调用sys_ipc_try_send, 如果返回的值是-E_IPC_NOT_RECV, 那么也应该继续循环。

最后, 我们执行make grade, 结果如下:

4 Challenges

4.1 Challenge 2

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 Challenge 是让我们实现`sched_yield()`的优先级调度算法。这不是简单的 Round-robin 调度, 而是将每个进程赋予一个优先级, 优先级高的应该比优先级低的先运行。

黄睿哲的报告里给出了一种实现, 这种实现每次运行RUNNABLE中优先级最高的进程。但这会有一个问题: Starvation! 如果一个优先级最高的进程不断地再运行, 那么优先级低的就永远不会被运行。所以我们必须用一种防止 Starvation 的方法。

我用的方法是, 考虑每个进程的运行次数`env_runs`, 并且优先级定义如下:

inc/env.h

```
#define ENV_PRIORITY_HIGH 2
#define ENV_PRIORITY_NORMAL 1
#define ENV_PRIORITY_LOW 0

struct Env {
    ...
    // Lab 4 Challenge 2
    uint32_t env_priority;
};
```

那么我每次就取env_runs >> env_priority最低的，这样在高优先级运行了低优先级 4 倍的次数之后，低优先级就可以运行了。这样就防止了 Starvation 的出现。

我的实现代码如下：

kern/env.c

```
...
struct Env *envs = NULL;    // All environments
bool env_priority_enabled = 0; // Lab 4 Challenge 2
```

kern/env.h

```
...
extern struct Env *envs;    // All environments
extern bool env_priority_enabled;
```

kern/sched.c

```
void
sched_yield(void)
{
    struct Env *idle;

    // LAB 4: Your code here.
    int i;
    uint32_t min_runs = 0xffffffff;
    int min_choice = -1;
    if (thiscpu->cpu_env) {
        if (env_priority_enabled) {
            for (i = 0; i < NENV; i++)
                if (envs[i].env_status == ENV_RUNNABLE) {
                    if ((envs[i].env_runs >> envs[i].env_priority) < min_runs) {
                        min_runs = envs[i].env_runs >> envs[i].env_priority;
                        min_choice = i;
                    }
                }
            if (min_choice != -1)
                env_run(&envs[min_choice]);
        }
        if (thiscpu->cpu_env->env_status == ENV_RUNNING)
```

```

        env_run(thiscpu->cpu_env);
    } else {
        int cur_id = ENVX(thiscpu->cpu_env->env_id);
        for (i = (cur_id + 1) % NENV; i != cur_id; i = (i + 1) % NENV)
            if (envs[i].env_status == ENV_RUNNABLE)
                break;
        if (i != cur_id)
            env_run(&envs[i]);
        if (thiscpu->cpu_env->env_status == ENV_RUNNING)
            env_run(thiscpu->cpu_env);
    }
} else {
    for (i = 0; i < NENV; i++)
        if (envs[i].env_status == ENV_RUNNABLE)
            break;
    if (i != NENV)
        env_run(&envs[i]);
}

// sched_halt never returns
sched_halt();
}

```

这里需要注意的是，用户程序应该能够开启/关闭优先级调度，因为这种调度在进程很多时，每次 `yield` 都会扫描所有进程，所以会导致某些 `test` 运行速度奇慢。所以这里我还设置了 `env_priority_enabled` 来让用户开启/关闭优先级调度模式。

我还添加了几个系统调用：

kern/syscall.c

```

static int
sys_env_set_priority(envid_t envid, uint32_t priority) {
    struct Env *env;
    int r;
    if ((r = envid2env(envid, &env, 1)) < 0)
        return r;
    env->env_priority = priority;
    return 0;
}

static int
sys_env_get_runs(envid_t envid) {
    struct Env *env;
    int r;
    if ((r = envid2env(envid, &env, 1)) < 0)
        return r;
    return env->env_runs;
}

```

```
static int
sys_env_enable_priority(bool enabled) {
    env_priority_enabled = enabled;
    return 0;
}
```

lib/syscall.c

```
// Lab 4 Challenge 2
int
sys_env_set_priority(envid_t env, uint32_t priority)
{
    return syscall(SYS_env_set_priority, 1, env, priority, 0, 0, 0);
}

int
sys_env_get_runs(envid_t env)
{
    return syscall(SYS_env_get_runs, 0, env, 0, 0, 0, 0);
}

int
sys_env_enable_priority(bool enabled)
{
    return syscall(SYS_env_enable_priority, 1, enabled, 0, 0, 0, 0);
}
```

有一些比较琐碎的代码修改就不贴出来了，最后还写了几个用户程序来测试

kern/priorityhigh.c

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

#define TIMES 10

void
umain(int argc, char **argv)
{
    int i;
    env_t t = sys_getenv();
    sys_env_enable_priority(1);
    sys_env_set_priority(t, ENV_PRIORITY_HIGH);
    for (i = 0; i < TIMES; i++) {
        cprintf("i am high priority env %08x [runs = %d]\n", t,
            sys_env_get_runs(t));
        sys_yield();
    }
    return;
}
```

kern/prioritynormal.c

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

#define TIMES 10

void
umain(int argc, char **argv)
{
    int i;
    envid_t t = sys_getenvid();
    sys_env_enable_priority(1);
    sys_env_set_priority(t, ENV_PRIORITY_NORMAL);
    for (i = 0; i < TIMES; i++) {
        cprintf("i am normal priority env %08x [runs = %d]\n", t,
            sys_env_get_runs(t));
        sys_yield();
    }
    return;
}
```

kern/prioritylow.c

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

#define TIMES 10

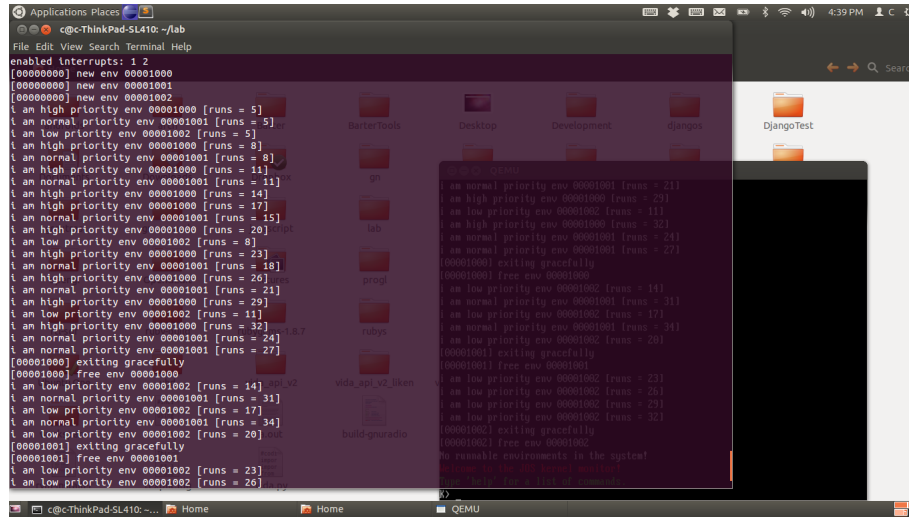
void
umain(int argc, char **argv)
{
    int i;
    envid_t t = sys_getenvid();
    sys_env_enable_priority(1);
    sys_env_set_priority(t, ENV_PRIORITY_LOW);
    for (i = 0; i < TIMES; i++) {
        cprintf("i am low priority env %08x [runs = %d]\n", t, sys_env_get_runs
            (t));
        sys_yield();
    }
    return;
}
```

kern/init.c

```
ENV_CREATE(user_priorityhigh, ENV_TYPE_USER);
ENV_CREATE(user_prioritynormal, ENV_TYPE_USER);
ENV_CREATE(user_prioritylow, ENV_TYPE_USER);
```

这里为什么要采用三个用户程序而没有用fork(), 是因为fork()本身可能会有点慢, 就无法保证高优先级, 中优先级和低优先级三个程序是同时开始等待调用的。

运行结果如下：



5 Tips

1. Kernel 调试信息可以用 warn 输出，User Mode 则要用 cprintf
2. Regression Test 很重要！Exercise 15 就因为 Lab 3 多加了一句话让我调试了好久，就犯了这个错误！
3. GDB 调试基本上是不会有帮助的
4. 参考的实验报告有很多结论是错的，一定要冷静分析。这句话好像说过很多次了。
5. 有些代码实在是难写，比如 Exercise 10，而且还很难理解。还好最后多看几次，渐渐地就明白了 Trick 在哪里。

6 References

北京大学操作系统实习 (实验班) 报告，黄睿哲。操作系统 JOS 实习第一次报告，张弛。