JOS Lab4 实验报告

朱汉峰
1120379059

完成了所有的Exercise，并且还完成了第二个challenge，在JOS上实现了基于优先级的进程调度。以及回答所有问题。

Part A:Multiprocessor Support and Cooperative Multitasking

这部分主要就是对Lab3的JOS进行扩展，使其能够支持多核系统，并提供一些系统调用，使得用户程序可以创建新的环境（进程）。此外，在这部分还需要实现一个比较简单的内核大锁机制以及Round-robin方式的进程调度，使得JOS可以在多个进程之间进行自由切换。

Exercise 1

这个练习的主要任务就是阅读 `kern/init.c` 中的两个函数 `boot_aps()` 和 `mp_main()` ，编码任务反倒是挺简单的，在 `page_init()` 的函数中，如果某个物理内存位于 `MPENTRY_PADDR` ，那么就不能将其加入到 `page_free_list` 中去，要实现这个功能，只需要在将新的内存页插入到的 `page_free_list` 的时候加一个判断即可，部分关键代码如下：

```
for(i = 1; i < npages_basemem; ++i){
    pages[i].pp_ref = 0;

    if(i == PGNUM(MPENTRY_PADDR))
        continue;

    // insert new page into page free list
    pages[i].pp_link = page_free_list;
    page_free_list = &pages[i];
}
```

Exercise 2

这个练习就是将每个CPU映射环境，按次序映射到内存中的KSTACKTOP处，主要就是调用 `boot_map_regin()` ，然后循环NCPU次就可以搞定了，代码如下：

```
int i;
for(i = 0; i < NCPU; ++i){
    uint32_t kstacktop = KSTACKTOP - i*(KSTKSIZE+KSTKGAP);
    boot_map_region(kern_pgdir, kstacktop - KSTKSIZE,
                    KSTKSIZE, PADDR(percpu_kstacks[i]),
                    PTE_W | PTE_P);
}
```

Exercise 3

这个练习就是为每个CPU初始化TSS以及TSS的描述符，和之前单一CPU的情形还是挺像的，只不过这里需要处理的是多CPU的情况，不过流程大致相似，再加上注释的帮助，实现起来还不算太难。代码都位于 `trap_init_percpu()` 中，关键代码如下：

```
int i = thiscput->cpu_id;
thiscpu->cpu_ts.ts_esp0 = KSTACKTOP - i*(KSTKSIZE+KSTKGAP);
thiscpu->cput_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;
```

Exercise 4

这个练习就是为JOS提供一个内核大锁功能，以防止资源竞争。实现起来非常简单，是整个这个Lab中，仅次于Exercise 1的一个练习。这个练习主要设计到4个函数，修改也非常简单，关键代码如下：

```
i386_init():

lock_kernel();
```

```
mp_main():

lock_kernel();
sched_yield();

// for(;;);
```

```
trap():

lock_kernel();
```

```
env_run():

unlock_kernel();
```

Exercise 5

这个练习是实现一个round-robin方式的进程调度算法，round-robin可以说是最基础的进程调度算法，其核心思想就是对所有进程都一视同仁，从当前进程的下一个进程开始，如果该进程的类型不是 `ENV_TYPE_IDLE` ，而且其状态

为 `ENV_RUNNABLE` ，则运行这个进程，否则就检测下一个是否可以运行，依次循环，直到再次循环到当前的进程为止。关键代码如下：

```
if(curenv != NULL){
    int curid = ENVX(curenv->env_id);
    int i = (curid+1) % NENV;

    while(i != curid){
        if(envs[i].env_type != ENV_TYPE_IDLE &&
            envs[i].env_status == ENV_RUNNABLE)
            break;

        i = (i+1) % NENV;
    }

    if(i != curid)
        env_run(envs + i);

    if(curenv->env_status == ENV_RUNNING)
        env_run(curenv);
}
```

Exercise 6

这个练习是在JOS中实现一个简单的fork功能，虽说这个fork功能比较简单，但涉及到的内容却比较多，这个练习可以说是整个Part A中最复杂的一个。这个练习中需要实现的主要就是在用户模式下创建子进程。

sys_exofork()

这个函数不接收任何参数，它的作用就是利用当前的环境，复制一个新的环境，并返回新环境的ID。创建环境时使用之前lab中实现的 `env_alloc` 函数。此外，还需要将这个新环境的状态设置为 `ENV_NOT_RUNNABLE` ，关键代码如下：

```
static envid_t
sys_exofork(void)
{
    struct Env *newenv;
    int r = env_alloc(&newenv, curenv->env_id);
    if(r < 0)
        return r;

    newenv->env_status = ENV_NOT_RUNNABLE;
    newenv->env_tf = curenv->env_tf;
    newenv->env_tf.tf_regs.reg_eax = (uint32_t)0;

    return newenv->env_id;
}
```

sys_env_set_status(envid_t envid, int status)

顾名思义，这个函数的功能就是为环境设置状态，但有一点需要注意的是，这个函数只支持对环境设置两种状态，
即 `ENV_RUNNABLE` 和 `ENV_NOT_RUNNABLE` ，所以在设置状态之前，需要对参数 `status` 进行判断，如果参数不正确，则直接返回错误。关键代码如下：

```
static int
sys_env_set_status(envid_t envid, int status)
{
    if((status != ENV_RUNNALBE) || (status != ENV_NOT_RUNNA
BLE))
        return -E_INVAL;

    struct Env *env;
    int r = envid2env(envid, &env, 1);
    if(r < 0)
        return r;

    env->env_status = status;
    return 0;
}
```

sys_page_alloc(envid_t envid, void *va, int perm)

这个函数的作用是为id为envid的环境申请一个新的物理页，并将其映射到虚拟地址 `va` 处，可以调用之前lab实现的函数 `page_insert` 来实现对物理页的映射。个人认为这个函数的关键在于对 `va` 以及 `perm` 参数的有效性检查，其它都不是太难。关键代码如下：

```
static int
sys_page_alloc(envid_t envid, void *va, int perm)
{
    uint32_t uva = (uint32_t)va;
    if(va >= UTOP || uva%PGSIZE)
        return -E_INVAL;

    if(!(perm & PTE_P) || !(perm & PTE_U) ||
        (perm & (~PTE_SYSCALL)))
        return -E_INVAL;

    struct Env *env;
    int r = envid2env(envid, &env, 1);
    if(r < 0)
        return r;

    struct Page *pp;
    pp = page_alloc(ALLOC_ZERO);
    if(!pp)
        return -E_NO_MEM;

    r = page_insert(env->env_pgdir, pp, va, perm);
    if(r < 0){
        page_free(pp);
        return -E_NO_MEM;
    }

    return 0;
}
```

sys_page_map(envid_t srcenvid, void *srcva, envid_t dstenvid, void *dstva, int perm)

这个函数的功能和上面的函数 `sys_page_alloc` 有点类似，不同点在于，这个函数并不是申请新的物理页，而是讲原来映射在虚拟地址 `srcva` 出的物理页，映射到虚拟地址 `dstva` 处，同时设置其权限为 `perm` 。同样，参数的有

效性检查也是这个函数的重点。关键代码如下：

```
static int
sys_page_map(envid_t srcenvid, void *srcva, envid_t dstenvi
d, void *dstva, int perm)
{
    uint32_t usrcva = (uint32_t)srcva;
    uint32_t udstva = (uint32_t)dstva;

    // check virtual address
    if(usrcva >= UTOP || usrcva%PGSIZE)
        return -E_INVAL;
    if(udstva >= UTOP || udstva%PGSIZE)
        return -E_INVAL;

    // check perm
    if(!(perm & PTE_P) || !(perm & PTE_U) ||
        (perm & (~PTE_SYSCALL)))
        return -E_INVAL;

    // get env
    struct Env *srcenv;
    struct Env *dstenv;
    int r = envid2env(srcenvid, &srcenv, 1);
    if(r < 0)
        return r;
    r = envid2env(dstenvid, &dstenv, 1);
    if(r < 0)
        return r;

    struct Page *pp;
    pte_t *pte;
    pp = page_lookup(srcenv->env_pgdir, srcva, &pte);
    if(pp == NULL)
        return -E_NO_MEM;

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

    r = page_insert(dstenv->env_pgdir, pp, dstva, perm);
    if(r < 0)
        return r;
```

```
    return 0;
}
```

sys_page_unmap(envid_t envid, void *va)

这个函数的作用正好和上面的函数相反，它是用来取消虚拟地址 `va` 处的映射，其实也就是移除原来映射在 `va` 处的物理页。这个函数的实现相比上面两个函数还是比较简单的。关键代码如下：

```
static int
sys_page_unmap(envid_t envid, void *va)
{
    uint32_t uva = (uint32_t)va;
    if(uva >= UTOP || uva%PGSIZE)
        return -E_INVAL;

    struct Env *env;
    int r = envid2env(envid, &env, 1);
    if(r < 0)
        return r;

    page_remove(env->env_pgdir, va);
    return 0;
}
```

syscall

最后，还需要在 `kern/syscall.c` 里的 `syscall` 函数中，将以上四个函数和对应的系统调用号进行绑定，关键代码如下：

```
int32_r = 0;
switch(syscallno){

// other cases

case SYS_exofork:
    r = sys_exofork();
    break;
case SYS_env_set_status:
    r = sys_env_set_status((envid_t)a1, (int)a2);
    break;
case SYS_page_alloc:
    r = sys_page_alloc((envid_t)a1, (void*)a2, (int)a3);
    break;
case SYS_page_map:
    r = sys_page_map((envid_t)a1, (void*)a2, (envid_t)a3, (
void*)a4, (int)a5);
    break;
case SYS_page_unmap:
    r = sys_page_unmap((envid_t)a1, (void*)a2);
    break;

// other cases
   ...
}
```

Part B:Copy-on-Write Fork

这部分主要是在JOS中实现copy-on-write fork，在以前的unix中，当fork函数被调用时，它会复制父进程的所有内容到子进程中去，这个方式就和这个lab中的dumbfork一样，虽然简单，但效率却很低。因为当一个应用程序调用fork之后，通常紧接着的就是调用exec()载入新的数据来执行一个新的程序，那么从父进程复制过来的数据就没用了，也就是说那么数据白复制了。因此，为了提高效率，在之后的unix中，实现fork函数时，通常都是采用copy-on-write的方式。当一个应用程序调用fork时，fork并不会把父进程的数据都复制到子进程，而是采用映射的方式，将父进程的内容映射到子进程，从而实现共享相同的数据。只有当子进程或父进程想要修改共享数据时，才复制一份新的数据，从而大幅度提高了fork的效率。这部分，主要就是实现copy-on-write的fork，以及一系列相关函数。

Exercise 7

这个练习主要时实现 `sys_env_set_pgfault_upcall` 函数，这个函数可以说是整个Part B的基础，它用来设置当发生页错误时，应该调用什么函数来进行处理。这是一个系统调用，每个应用程序都可以设置自己的页错误处理函数。函数的实现倒是不难，关键代码如下：

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

    env->env_pgfault_upcall = func;
    return 0;
}
```

Exercise 8

这个练习主要是完善 `kern/trap.c` 中的 `page_fault_handler` 函数，使其能够支持用户自定义的页错误处理函数。如果是用户态下的页错误，并且该环境（进程）设置upcall的话，那么就将eip指向该upcall。实现这个函数的关键是要理解 `用户异常栈` ，这是运行用户自定义中断处理函数的栈环境。这个栈的范围是[UXSTACKTOP-PGSIZE, UXSTACKTOP-1]，正好是一个页。此外，在调用用户自定义的页错误处理函数之前，需要保护现场，也就是压入一个 `UTrapframe` 。关键代码如下：

```
void *upcall = curenv->env_pgfault_upcall;
if(upcall){
    struct UTrampframe *utf;
    uint32_t esp = tf->tf_esp;
    size_t len = sizeof(struct UTrapframe);
    // 如果已经运行在用户异常栈，那么则从其esp处开始压入参数
    // 否则从UXSTACKTOP处开始压入参数
    if(esp >= (UXSTACKTOP-PGSIZE) && esp < UXSTACKTOP){
        utf = (UTrapframe*)(esp-len-4);
    }else{
        utf = (UTrapframe*)(UXSTACKTOP-len);
    }

    user_mem_assert(curenv, (void*)utf, len, PTE_P | PTE_U
| PTE_W);

    // 复制原来trapframe里的内容
    utf->utf_eflags = tf->tf_eflags;
    utf->utf_eip = tf->tf_eip;
    utf->utf_err = tf->tf_err;
    utf->utf_esp = tf->tf_esp;
    utf->utf_fault_va = fault_va;
    utf->utf_regs = tf->tf_regs;

    curenv->env_tf.tf_eip = (uintptr_t)upcall;
    curenv->env_tf.tf_esp = (uint32_t)utf;

    env_run(curenv);
}
```

Exercise 9

这个练习要做的就是实现用户自定义页错误处理函数的入口函数，`_pgfault_upcall`，当发生用户态下的页错误时，如果用户自定义了处理函数，那么首先这个入口函数会被调用，然后这个入口函数会调用用户的自定义函数。之所以要通过这个入口函数来调用用户自定义的处理函数，是因为这里涉及到堆栈切换，即由用户异常栈切换到用户运行栈。关键代码如下：

```
// save trap-time
movl 0x28(%esp), %eax

movl 0x30(%esp), %ebx
subl $0x4, %ebx
movl %eax, (%ebx)

// change trap-time
movl %ebx, 0x30(%esp)

// skip error code and fault va
addl $0x8, %esp
popal

// skip %esp
add $0x4, %esp
popfl

// switch back to the adjusted trap-time stack
popl %esp

// return to re-execute the instruction that faulted
ret
```

Exercise 10

这个练习主要是通过库函数的方式将设置用户自定义页错误的处理函数的系统调用暴露给用户，方便用户直接在应用程序里调用。关键代码很少，如下：

```
int r = sys_page_alloc(0, (void*)(UXSTACKTOP-PGSIZE), PTE_U
 | PTE_P | PTE_W);
if(r < 0)
    return;

sys_env_set_pgfault_upcall(0, _pgfault_upcall);
```

Exercise 11

这个练习主要是以库函数的方式，为用户程序提供copy-on-write fork相关的函数，涉及到三个函数的实现。

duppage(envid_t envid, unsigned pn)

这个函数的作用就是将页pn映射到id为envid的环境中的相同地址，从而在不复制父进程数据的内存数据的情况下，实现数据共享。相比复制数据，这种方式可以大幅度提高fork的效率。代码如下：

```
static int
duppage(envid_t envid, unsigned pn)
{
    int r;
    uint32_t addr = pn * PGSIZE;
    if(addr >= UTOP)
        panic("duppage: addr above UTOP");

    pte_t pde = vpd[PDX(addr)];

    if(pre & PTE_P){
        pte_t pte = vpt[PGNUM(addr)];
        if(pte & PTE_P){
            if(pre & (PTE_W | PTE_COW)){
                r = sys_page_map(0, (void*)addr, envid, (vo
id*)addr,
                                     PTE_U | PTE_P | PTE_COW
);

                if(r < 0)
                    panic("duppage: %e", r);
                r = sys_page_map(0, (void*)addr, 0, (void*)
addr,
                                     PTE_U | PTE_P | PTE_COW
);
                if(r < 0)
                    panic("duppage: %e", r);
            }else{
                r = sys_page_map(0, (void*)addr, envied, (v
oid*)addr,
                                     PTE_U | PTE_P);

                if(r < 0)
                    panic("duppage: %e", r);
            }
        }
    }
    return 0;
}
```

fork()

这个fork函数，其实就是copy-on-write的fork，它主要就是扩展了原来的 `exofork` ，在其基础上设置了一些用户自定义的错误处理函数，一旦发生页错误的时候，可以由用户自定义的函数来处理。

```
envid_t
fork()
{
    set_pg_fault_handler(pgfault);
    envid_t envid = sys_exofork();
    if(envied < 0)
        painc("panic: %e", envid);

    if(envied == 0){
        thisenv = envs + ENVX(sys_getenvid());
        return 0;
    }

    usigned i;
    for(i = 0; i < UTOP / PGSIZE - 1; ++i)
        duppage(envid, i);

    // 为子进程的用户异常栈申请页
    int r = sys_page_alloc(envid, (void*)(UXSTACKTOP-PGSIZE
),
                                PTE_U | PTE_P | PTE_W);

    if(r < 0)
        painc("fork: %e", r);

    extern void _pgfault_upcall();
    r = sys_env_set_pgfault_upcall(envid, (void*)_pgfault_u
pcall);
    if(r < 0)
        panic("fork: %e", r);

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

    return envid;
}
```

pgfault(struct UTrapframe *utf)

这是用户页的错误处理函数，如果错误页是copy-on-write类型的，那么当错误发生时，就将原来映射在共享空间的地址重新映射到的新的可写的私有备份。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;

    if(!(err & FEC_WR))
        panic("pgfault: wrong utf_err");

    pde_t pde = vpd[PDX(addr)];
    pte_t pte = vpt[PGNUM(addr)];

    if(!(pre & PTE_P))
        panic("pgfault: wrong pre");

    if(!(pre & (PTE_U | PTE_P | PTE_COW)))
        panic("pgfault: wrong permission");

    r = sys_page_alloc(0, (void*)PFTEMP, PTE_U | PTE_P | PT
E_W);
    if(r < 0)
        panic("pgfault: %e", r);

    memmove((void*)PFTEMP, (const void*)((PGNUM(addr)) << P
TXSHFIT),
                PGSIZE);

    r = sys_page_map(0, (void*)PFTEMP, 0,
                        (void*)((PGNUM(addr)) << PTXSHFIT),
                        PTE_U | PTE_P | PTE_W);

    if(r < 0)
        panic("pgfault: %e", r);
}
```

Part C:Preemptive Multitasking and Inter-Process Communication(IPC)

这部分主要是实现抢占式多任务调度以及进程间通信。所谓抢占式，就是不允许某个进程一直占着CPU，通过外部硬件时钟发来的时间中断来进行抢占。

Exercise 12

这个练习比较简单，和之前lab中系统中断的初始化非常相似，只是代码略多，关键代码如下：

```
kern/trapentry.S 定义相关的irq 处理函数，共16个：

TRAPHANDLER_NOEC(t_irq0, IRQ_OFFSET + 0);
…
…
TRAPHANDLER_NOEC(t_irq15, IRQ_OFFSET + 15);
```

```
kern/trap.c的trap_init 函数中，将IRQ号和对应的处理函数进行绑定

extern void t_irq0();
…
…
extern void t_irq15();

SETGATE(idt[IRQ_OFFSET + 0], 0, GD_KT, t_irq0, 0);
…
…
SETGATE(idt[IRQ_OFFSET + 15], 0, GD_KT, t_irq15, 0);
```

```
kern/env.c的env_alloc函数中，enable用户态下的中断：

e->env_tf.tf_eflags |= FL_IF;
```

Exercise 13

这个练习主要是让JOS系统在有时钟中断发生时，调用 `sched_yield` 函数，从而实现抢占时调度。关键代码都位于 `trap_dispatch` 中，如下：

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

Exercise 14

这个练习主要是在JOS系统中实现进程间通信。JOS系统中的进程间通信信息包括两个部分，一个是32位的数据，以及一个可选的页面映射。这个练习主要涉及两个函数：

sys_ipc_recv(void *dstva)

这个函数的作用就是用来接收进程间通信的信息，实现这个函数，有一点需要注意，那就是dstva是可以位于 `UTOP` 之上的。关键代码如下：

```
static int
sys_ipc_recv(void *dstva)
{
    uint32_t uva = (uint32_t)dstva;
    if(uva < UTOP && uva%PGSIZE)
        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;
}
```

sys_ipc_try_send(envid_t envied, uint32_t value, void *srcva, unsigned perm)

顾名思义，这个函数也很好理解，就是用来发送进程间通信信息的。实现这个函数其实也不难，不过由于参数比较多，所以对参数有效性检查的代码相应地也就比较多了，因而代码比较繁琐。具体代码如下：

```
static int
sys_ipc_try_send(envid_t envied, uint32_t value, void *srcv
a, unsigned perm)
{
    struct Env *env;
    int r = envid2env(envid, &env, 0);
    if(r < 0)
        return r;

    if(env->env_ipc_from || !env->env_ipc_recving)
        return -E_IPC_NOT_RECV;

    uint32_t uva = (uint32_t)srcva;
    if(uva < UTOP){
        if(uva % PGSIZE)
            return -E_INVAL;

        // check perm
        if(!(perm & PTE_P) || !(perm & PTE_U) ||
            (perm & (~PTE_SYSCALL)))
            return -E_INVAL;

        pte_t *pte;
        struct Page *pp = page_lookup(curenv->env_pgdir, sr
cva, &pte);
        if(!pp)
            return -E_INVAL;

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

        if((uint32_t)(env->env_ipc_dstva) < UTOP){
            r = sys_page_map(curenv->env_id, srcva, envid,
                             env->env_ipc_dstva, perm);

            if(r < 0)
                return r;
        }
    }

    env->env_ipc_recving = 0;
    env->env_ipc_from = curenv->env_id;
    env->env_ipc_value = value;
```

```
    if((uint32_t)(env->env_ipc_dstva) < UTOP)
        env->env_ipc_perm = perm;
    else
        env->env_ipc_perm = 0;

    env->env_status = ENV_RUNNABLE;
    env->env_tf.tf_regs.reg_eax = 0;

    retur 0;
}
```

Challenge

实现了Lab中的第二个challenge，即在JOS中支持基于优先级的调度，并且写了几个测试程序。为了实现基于优先级的调度，那么进程首先得有优先级这个属性，因此，第一步就是在 `inc/env.h` 中的 `Env` 结构体上增加一个优先级属性：

```
struct Env{
    // other properties

    uint32_t env_prio;

    // other properties
}
```

为了方便使用，还需要在这个文件中定义一些进程的优先级，通过宏的方式定义：

```
#define PRIO_HIGH 0x1000
#define PRIO_NORM 0x0100
#define PRIO_IDLE 0x0010
```

由于原来的JOS系统中并没有优先级的概念，所以在初始化进程的时候，我们还需要为进程设置一个默认的优先级，即 `PRIO_NORM` ，这段代码位于 `kern/env.c` 中的 `env_alloc` 函数中：

```
e->env_prio = PRIO_NORM;
```

现在每个进程都有了默认的优先级，为了支持用户程序对优先级的修改，我们还需要提供一些系统调用，这里主要也就是一个系统调用，让用户设置优先级。涉及到4个文件的修改。

inc/syscall.h 定义新的中断号：

```
SYS_env_set_prio;
```

kern/syscall.c 中定义函数：

```
static int
sys_env_set_prio(envid_t envid, uint32_t prio)
{
    struct Env *env;
    int r = envid2env(envid, &env, 1);
    if(r < 0)
        return r;

    env->env_prio = prio;
    return 0;
}
```

inc/lib.h 声明：

```
int sys_env_set_prio(envid_t envid, uint32_t prio);
```

lib/syscall.h 定义对应的库函数：

```
int
sys_env_set_prio(envid_t envid, uint32_t prio)
{
    return syscall(SYS_env_set_prio, 1, envied, prio, 0, 0,
 0);
}
```

调度的代码代码依然是位于 `kern/sched.c` 中，在进行调度时，和原来一视同仁的方式不同，这里每次都是选择优先级最高的一个来运行，代码如下：

```
if(curenv != NULL){
    int curid = ENVX(curenv->env_id);
    int i = (curid+1) % NENV;

    uint32_t max_prio = 0;
    int torun = -1;
    while(i != curid){
        if(envs[i].env_type != ENV_TYPE_IDLE &&
            envs[i].env_status == ENV_RUNNABLE &&
            envs[i].env_prio > max_prio){

            max_prio = envs[i].env_prio;
            torun = i;
        }
        i = (i+1)%NENV;
    }

    if(torun > -1 && torun != curid)
        env_run(envs + torun);

    if(curenv->env_status == ENV_RUNNING)
        env_run(curenv);
}
```

测试优先级调度

为了验证这个算法是否正确，我写了三个简单的测试程序，都位于 `user` 目录下

- user/prio_high.c
- user/prio_norm.c
- user/prio_idle.c

三个程序的代码大同小异，其中 `user/prio_high.c` 代码如下：

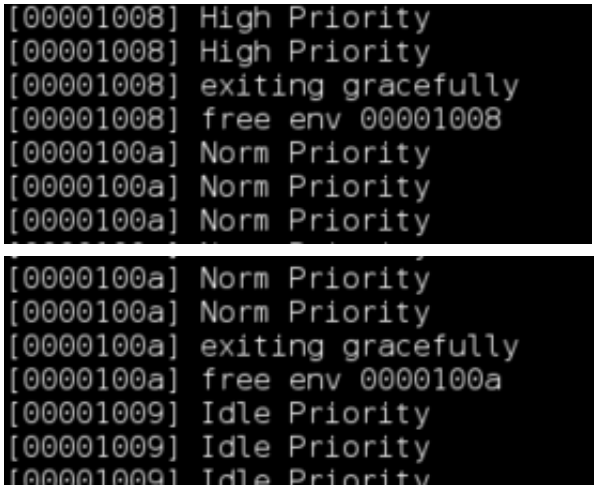
```
#include <inc/lib.h>
#incldue <inc/env.h>

void
umain(int argc, char **argv)
{
    sys_env_set_prio(0, PRIO_HIGH);
    int i = 0;
    for(i = 0; i < 500; ++i)
        cprintf("[%08x] High Priority\n", sys_getenvid());
}
```

最后需要在 `kern/init.c` 中创建这三个进程，使得JOS启动时就可以直接运行测试程序，代码如下：

```
// ENV_CREATE(user_primes, ENV_TYPE_USER);
ENV_CREATE(user_prio_high, ENV_TYPE_USER);
ENV_CREATE(user_prio_idle, ENV_TYPE_USER);
ENV_CREATE(user_prio_norm, ENV_TYPE_USER);
```

测试结果如下：



测试发现 `PRIO_HIGH` 的进程最先结束， `PRIO_IDLE` 的最后结束，这说明这个调度算法是正确的。

Qestions

Q1

> 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?

这个宏的作用就是将内核线性地址转成物理地址，因为所有的内核代码都是链接在 `KERNBASE` 上的线性地址空间里。在Lab1中，我们使用了一个非常简单的页表（其实也就是将链接地址减去 `KERNBASE` ）来将内核线性地址转化成物理地址，这种方式看似简单，其实非常冗余。因而，对于APs，JOS采用了 `MPBOOTPHYS` 宏来直接进行计算。对于内核线性地址va，直接将其减去 `mpentry_start` ，然后再加上AP的起始地址 `MPENTRY_PADDR` ，就得到了va的物理地址。

Q2

> 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.

因为我们必须要考虑到一些边界情况。比如发生中断时，我们在 `trap` 函数中才会调用 `lock_kernel` ，但实际早在lock之前，程序就已经运行在内核栈了，然后我们才会把现场信息保存到栈里。如果此时共用内核栈的话，那么保存现场时就会发生错误。因为这时多CPU对内核栈的使用出现了竞争。所以即使有内核大锁，还是需要为每个CPU分配不同的内核栈。

Q3

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

> address to which the virtual address maps. Why can the pointer e be dereferenced both before and after the addressing switch?

因为这个e指向的是内核地址空间，而这个空间是所有进程共享的，在任何进程中，e都是指向同样的物理地址。所以即使发生了页表切换，e指向的物理地址仍然不变，所以e在前后前后都可以被正确地解引用。