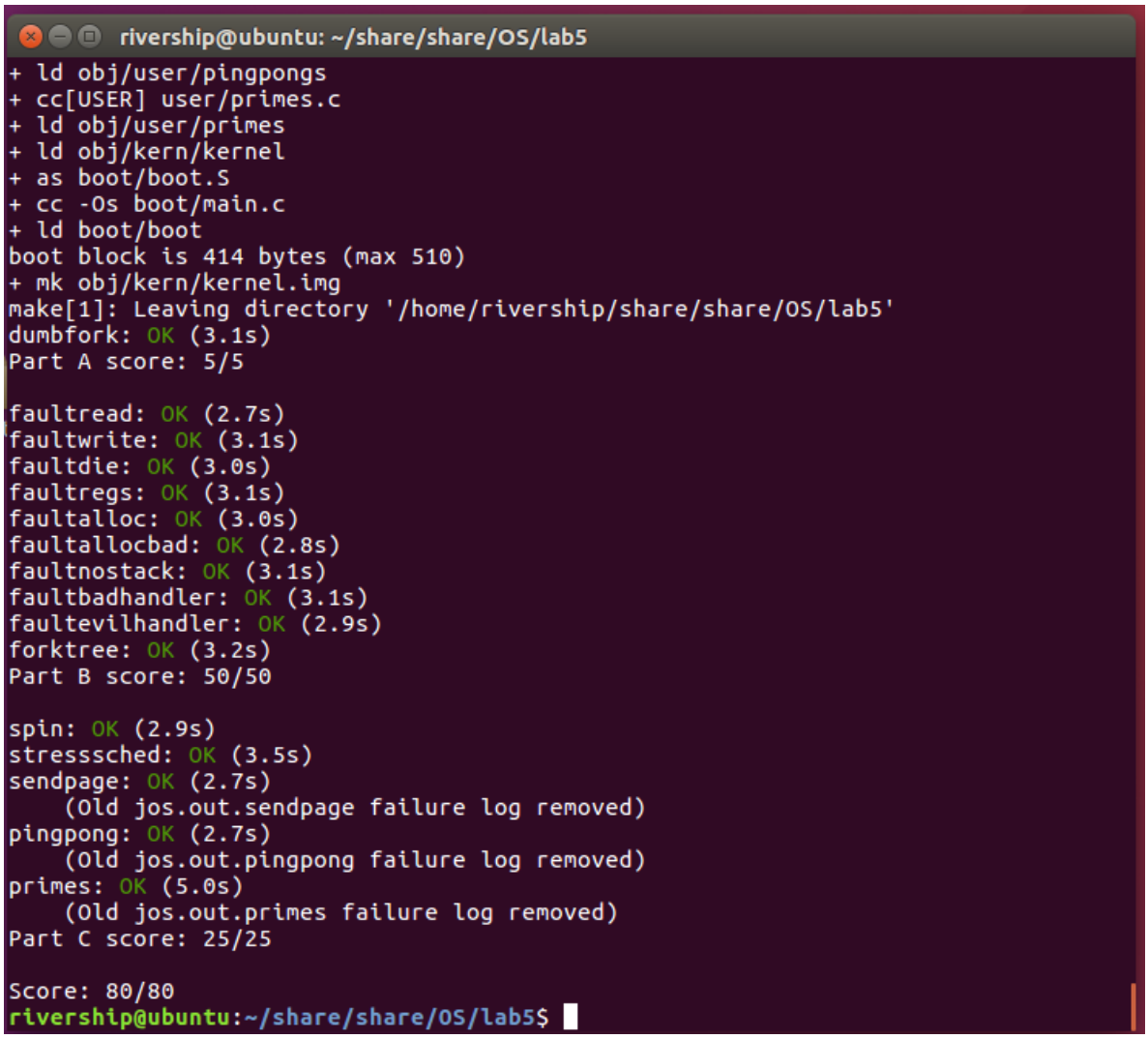
make grade



PART A：Multiprocessor Support and Cooperative Multitasking

excercise1

实现 `void *mmio_map_region(physaddr_t pa, size_t size)`

该函数是在`MMIO region`中给调用者分配虚拟地址空间(map到`[pa,pa+size)`,其中`pa`是IO设备物理地址)。

- 函数中设置了一个指向当前空闲区域的`base`指针，分配时，将`[base,base+size) map --> [pa,pa+size)`（这一步可以采用内存管理lab中的`boot_map_region`）

```
//检查是否 pg-aligned
assert(base%PGSIZE == 0 && pa%PGSIZE==0);
if (base + ROUNDUP(size,PGSIZE) >= MMIOLIM || base < MMIOBASE){
    panic("mmio out of memory");
}
//构造映射[base,base+size) ---> [pa,pa+size)
boot_map_region(kern_pgdir,base,ROUNDUP(size,PGSIZE),pa,PTE_PCD|PTE_PWT|PTE_
W);
```

- 分配成功，更新base指针，指向`base+size`,并且返回未更新前的base指针

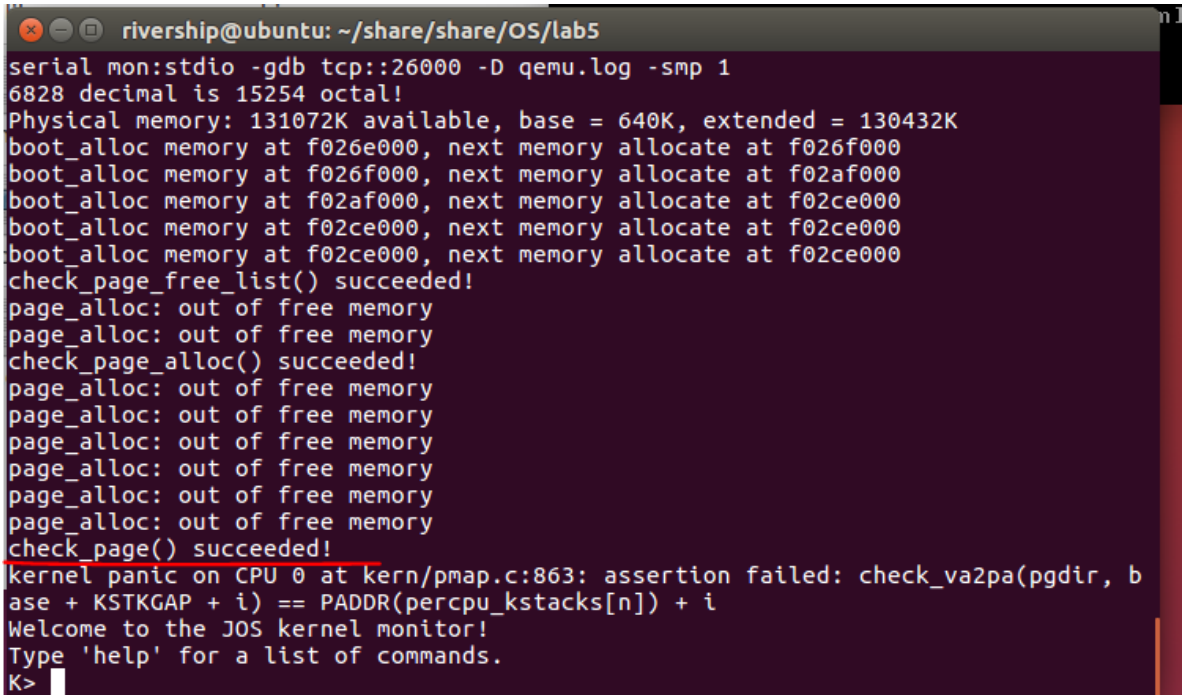
```
base += ROUNDUP(size,PGSIZE);
return (void*)(base-ROUNDUP(size,PGSIZE));
```

exercise2

修改`page_init`，将`MPENTRY_PADDR`所在的物理页设置为已使用，因为在`boot_aps`中会将`mpentry.S`中的代码复制到`code = KADDR(MPENTRY_PADDR)`的位置。由于`MPENTRY_PADDR = 0x7000`，只需要在`page_init`中处理下页号为7的物理页即可。

```
for (i = 0; i < npages; i++) {
    // i==0, for idt;
    // i==7, MPENTRY_PADDR(0x7000)
    if (i == 0 || i==7) {
        pages[i].pp_ref = 1;
        pages[i].pp_link = NULL;
    } else if (i >= io_hole_start_page && i < kernel_end_page) {
        pages[i].pp_ref = 1;
        pages[i].pp_link = NULL;
    } else {
        pages[i].pp_ref = 0;
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }
}
```

完成 exercise2 后，通过`check_page()`检查如下



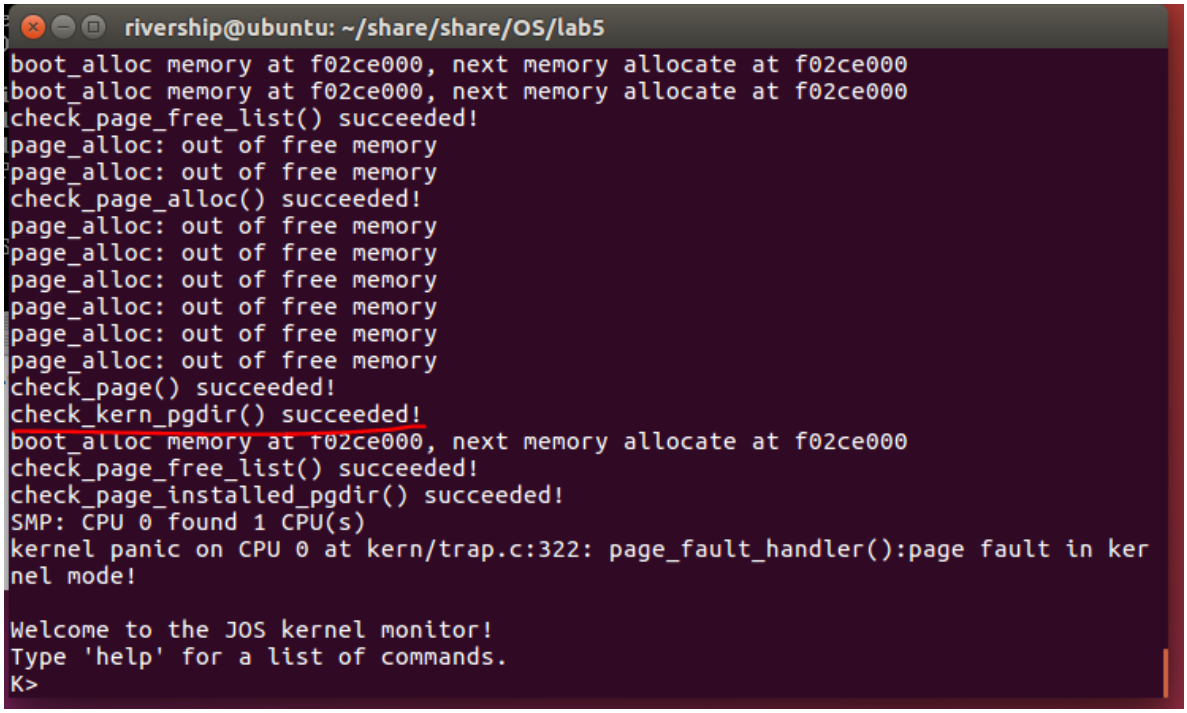
exercise3

在`mem_init_mp()`中为每一个CPU的kernel stack分配虚拟地址空间

- `percpu_kstacks[NCPU][KSTKSIZE]`（定义于`mpconifg.c`）是实际上每个CPU栈的所在地，即`percpu_kstacks[i]`为`cpu[i]`栈的起始地址（这个地址同样是虚拟地址，不过只需要`PADDR`就可以得到物理地址）
- `KSTACKTOP-i*(KSTKSIZE+KSTKGAP)`作为`cpu[i]`在虚拟地址空间的**栈顶**(map时需要从栈底开始，即需要再减去`KSTKSIZE`)
- 使用`boot_map_region`即可

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

完成exercise3后通过了`check_kern_pgdir()`



exercise4

修改`trap_init_percpu()`，适应多CPU的环境。

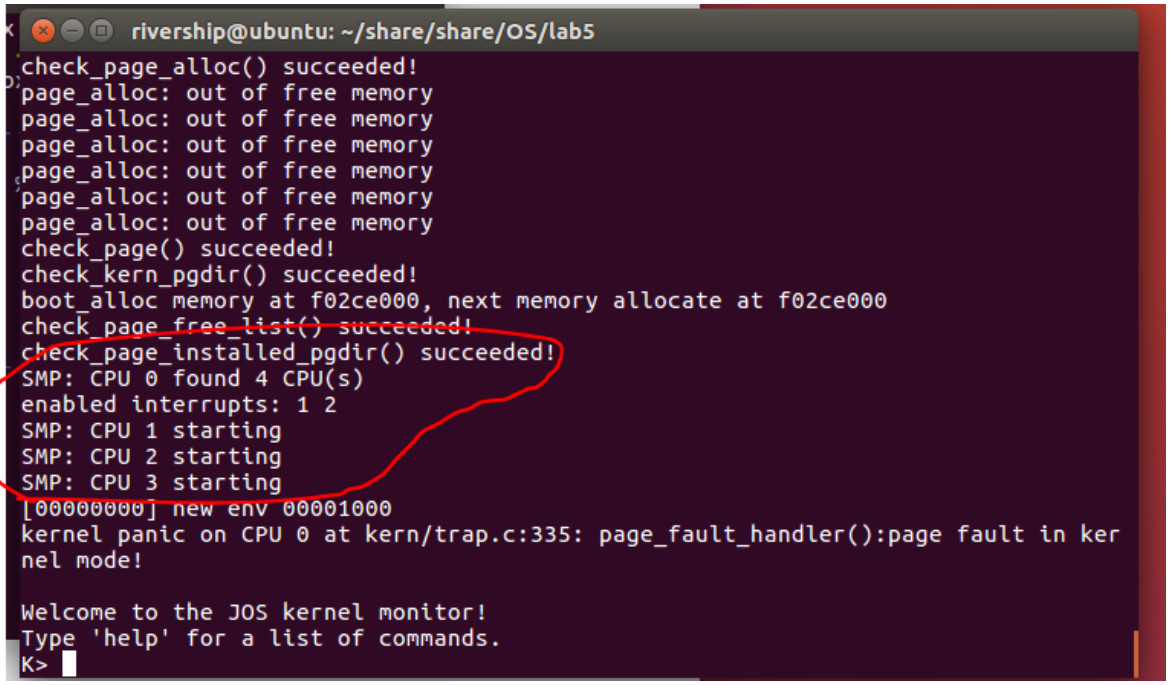
- 每个CPU会自己调用`trap_init_percpu()`，如下图

```
BPS ---(call boot_aps) |---> cpu0 mpentry --> mp_main--> trap_init_percpu
                       |---> cpu1 mpentry --> mp_main--> trap_init_percpu
                       ...
                       |---> cpun mpentry --> mp_main--> trap_init_percpu
```

- 因此`trap_init_percpu()`只需要完成对调用它的`cpu[i]`的任务即可，即设置好栈顶，数据段选择子，gdt表项，idt的相关信息，这一步仿造原来对`bps`的设置流程即可

```
//内核栈顶
(thiscpu->cpu_ts).ts_esp0 = KSTACKTOP - cpunum()*(KSTKSIZE+KSTKGAP);
//内核代码段选择子
(thiscpu->cpu_ts).ts_ss0  = GD_KD;
(thiscpu->cpu_ts).ts_iomb = sizeof(struct Taskstate);
//gdt表项
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;
ltr(GD_TSS0 + 8*cpunum());
lidt(&idt_pd);
```

完成`exercise4`，`make qemu CPUS=4`，发现所有CPU能够正常启动



exercise5

根据文档描述，在各个函数调用时，替调用者cpu给内核加锁，保证同步。

加解锁使用函数`lock_kernel()`和`unlock_kernel`即可，二者均依赖于`xchg`的原子性。

- `i386_init`，在调用`boot_aps`前加锁
  - bps唤醒aps时是一个接一个唤醒的，加锁后可以保证已被唤醒的ap无法操作内核
  - 同时，也因为aps是一个接一个唤醒的，所以即使锁是bps持有的，ap在调用mp_main并且需要访问内核时，实际上也只有一个CPU在操作内核（bps在等待该ap结束初始化）
- `mp_main`，在每个ap cpu结束初始化后，给内核上锁，获得锁后再调用`sched_yield()`
  - ap此处的锁实际上会在bps完成所有ap的唤醒、并且调用了用户程序之后才会获得，因为只有当`bps`调用了`env_run`之后才会`unlock_kernel()`
- `trap`，当trap来自用户程序时，加锁。（`tf_CS & 3`检测trap是否来自用户程序）
- `env_run`，在`env_pop_tf()`之前解锁，这样在开启用户程序之后，一个cpu就能够释放锁，同时另一个试图加锁的cpu能够获得锁，以便继续进行它在`mp_main`或`trap`中的任务。同时cpu在`halted`状态下也会释放锁，这时候cpu没有用户程序可以运行，同时也不需要处理其他事项，于是释放内核。

exercise6

实现采用`Round Robin`调度算法的调度函数`sched_yield()`，同时还要添加一个系统调用`sys_yield()`

- 将envs视为一个循环列表，查找一圈，寻找下一个RUNNABLE的进程
- 若上述过程未找到目标，查看`curenv`能否继续运行（是否为`ENV_RUNNING`）

```
void
sched_yield(void)
{
    struct Env *idle;
    envid_t i,curid;
    idle = curenv;
```

```
    if(idle == NULL)
        curid = NENV-1;
    else
        curid = ENVX(idle->env_id);
    // 将envs视为一个循环列表来查找，找到一个runnable的进程就run
    // 找不到，就查看curenv能否再run(有可能已经结束了)
    for( i=(curid+1)%NENV ; i != curid ; i = (i+1)%NENV){
        if(envs[i].env_status == ENV_RUNNABLE){
            env_run(&envs[i]);
        }
    }
    // 检查curenv是否能够再run
    if(idle && (idle->env_status==ENV_RUNNING || idle-
>env_status==ENV_RUNNABLE)){
        env_run(idle);
    }
    // sched_halt never returns
    sched_halt();
}
```

同时还要在kernle的`syscall()`中给switch添加一个条目指向`sys_yield()`

在`i386_init`中create三个`yield`程序，`make qemu CPUS=2`后如下，调度算法正常运行

```
rivership@ubuntu: ~/share/share/OS/lab5
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 2 CPU(s)
enabled interrupts: 1 2
SMP: CPU 1 starting
[00000000] new env 00001000
[00000000] new env 00001001
[00000000] new env 00001002
Hello, I am environment 00001000.
Hello, I am environment 00001001.
Back in environment 00001000, iteration 0.
Hello, I am environment 00001002.
Back in environment 00001001, iteration 0.
Back in environment 00001000, iteration 1.
Back in environment 00001002, iteration 0.
Back in environment 00001001, iteration 1.
Back in environment 00001000, iteration 2.
Back in environment 00001002, iteration 1.
Back in environment 00001001, iteration 2.
Back in environment 00001000, iteration 3.
Back in environment 00001002, iteration 2.
Back in environment 00001001, iteration 3.
Back in environment 00001000, iteration 4.
Back in environment 00001002, iteration 3.
All done in environment 00001000.
[00001000] exiting gracefully
[00001000] free env 00001000
Back in environment 00001001, iteration 4.
Back in environment 00001002, iteration 4.
All done in environment 00001001.
All done in environment 00001002.
[00001001] exiting gracefully
[00001001] free env 00001001
[00001002] exiting gracefully
[00001002] free env 00001002
No runnable environments in the system!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

exercise7

实现`sys_exofork`，为用户程序创建一个几乎全空的子进程（地址空间为空）

- `env_alloc`一个新的env（child），设置`stacus=ENV_NOT_RUNNABLE`，并且将其trapframe复制为parent的`trapframe`，这一步使子进程与父进程具有相同寄存器上下文，于是在地址空间也设置之后，子进程可以从父进程调用`sys_fork()`处开始继续运行（当然此处是不对子进程的地址空间进行更多操作的）

```
int r;
struct Env* e;
if( (r=env_alloc(&e,curenv->env_id)) < 0){
    cprintf("sys_exofork: %e\n",r);
    // env_alloc失败
    return r;
}
e->env_status = ENV_NOT_RUNNABLE;
e->env_tf = (curenv->env_tf);
```

- 同时在`trapframe`中将child的eax设置为0，这样当内核决定继续运行父/子进程并进行上下文切换时，子进程发现自己的`eax==0`，即`sys_exofork()`的返回值为0；父进程发现`sys_exofork()`返回值为子进程的id。于是一个进程可以区分在调用`sys_exofork()`之后是父进程还是子进程

```
// child返回值为0
(e->env_tf).tf_regs.reg_eax = 0;
// parent返回值为child id
return e->env_id;
```

全部函数代码如下：

```
static envid_t
sys_exofork(void)
{
    int r;
    struct Env* e;
    if( (r=env_alloc(&e,curenv->env_id)) < 0){
        cprintf("sys_exofork: %e\n",r);
        return r;
    }
    e->env_status = ENV_NOT_RUNNABLE;
    e->env_tf = (curenv->env_tf);
    // child返回值为0
    (e->env_tf).tf_regs.reg_eax = 0;
    // parent返回值为child id
    return e->env_id;
    //panic("sys_exofork not implemented");
}
```

同时需要在`syscall()`中添加一条对`sys_exofork()`的dispatch

PART B: Copy-on-Write Fork

exercise 8

完成 `sys_env_set_pgfault_upcall()`，为用户程序设置缺页异常处理的入口。用户的`page_fault`在经过`trap--->page_fault_handler()`之后，能够交给`env_pgfault_upcall`指向的用户空间处理函数。步骤如下：

- 找到`envid`对应的env结构，同时需要确保`curenv`对其有可写的权限（通过设置`envid2env()`的perm参数为1）

```
struct Env* e;
int r=envid2env(envid,&e,1);
if(r<0){
    cprintf("sys_env_set_pgfault_upcall: %e\n",r);
    return r;
}
```

- 随后修改`e->env_pgfault_upcall`使其指向`func`函数的起始地址即可

```
e->env_pgfault_upcall = func;
return 0;
```

全部代码如下：

```
static int
sys_env_set_status(envid_t envid, int status)
{
    //根据注释检查进程status
    if (!(status==ENV_NOT_RUNNABLE || status==ENV_RUNNABLE)){
        return -E_INVAL;
    }
    struct Env* e;
    //获取进程env，同时检查权限
    int r=envid2env(envid,&e,1);
    if(r<0){
        cprintf("sys_env_set_status: %e\n");
        return r;
    }
    e->env_status = status;
    return 0;
    //panic("sys_env_set_status not implemented");
}
```

同时需要在`syscall()`中添加一条对`sys_set_status()`的dispatch

exercise 9

完成`page_fault_handler() in kern/trap.c`

- 当curenv的`env_pgfault_upcall`已经设置时，user pagefault分为两种情况
    - 在用户进程正常运行时发生的pagefault：。这时候需要将**栈迁移**到用户的异常处理栈`UXSTACKTOP`（此为栈顶），在**栈顶**（向下增长）构建一个`UTrapframe`。这里直接将一个`UTrapframe`指针指到对应位置，方便之后进行构建

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

- 在异常栈上又发生了page fault，此时，需要在当前栈底空出一个`blank word(4 bytes)`，接下来再构建`UTrapframe`

```
if(tf->tf_esp < UXSTACKTOP && tf->tf_esp >= UXSTACKTOP-PGSIZE){
    utf = (struct UTrapframe*)(tf->tf_esp - (4 + sizeof(struct
UTrapframe)));
}
```

同时还要检查此时的构建会不会溢出用户异常栈。此处的异常栈用户自己预先申请（如在`set_pgfault_handler()`或者`fork()`时申请）

```
user_mem_assert(curenv,(void*)utf,sizeof(struct UTrapframe),PTE_W);
```

- 构建`UTrapframe`

```
utf->utf_regs = tf->tf_regs;
utf->utf_esp = tf->tf_esp;
utf->utf_eip = tf->tf_eip;
utf->utf_eflags = tf->tf_eflags;
utf->utf_err = tf->tf_err;
utf->utf_fault_va = fault_va;
```

- 接下来`kernel`就可以返回用户模式，并且将执行流交给用户的异常处理程序。由于涉及到内核到用户模式的转换，需要用到`iret`中断返回指令。这里可以使用`env_run`进行上下文切换

```
// env_run之后执行env_pgfault_upcall
tf->tf_eip = (uintptr_t)(curenv->env_pgfault_upcall);
// esp指向构建的 UTrapframe
tf->tf_esp = (uintptr_t)utf;
env_run(curenv);
```

全部代码如下：

```
    struct UTrapframe* utf;
    if(curenv->env_pgfault_upcall){
        if(tf->tf_esp < UXSTACKTOP && tf->tf_esp >= UXSTACKTOP-PGSIZE){
            utf = (struct UTrapframe*)(tf->tf_esp - (4 + sizeof(struct
UTrapframe)));
        }
        else{
            utf = (struct UTrapframe*)(UXSTACKTOP - sizeof(struct
UTrapframe));
        }
        //检查是否溢出异常处理栈
        user_mem_assert(curenv,(void*)utf,sizeof(struct UTrapframe),PTE_W);
        //设置Utrapframe
        utf->utf_regs = tf->tf_regs;
        utf->utf_esp = tf->tf_esp;
        utf->utf_eip = tf->tf_eip;
        utf->utf_eflags = tf->tf_eflags;
        utf->utf_err = tf->tf_err;
        utf->utf_fault_va = fault_va;
```

```
        //切换到用户空间的处理程序
        tf->tf_eip = (uintptr_t)(curenv->env_pgfault_upcall);
        tf->tf_esp = (uintptr_t)utf;
        env_run(curenv);
        return;
    }
```

exercise 10

完成`lib/pfentry.S`

此处的指令的目的为，`call`用户设置的handler，然后从异常处理中返回（返回到用户程序**或者**返回到上一层的异常处理）。返回时，需要恢复的上下文信息有：

- 通用寄存器组，可以使用`popal`指令将`UTrapframe`中保存寄存器组的帧通过弹栈恢复到各个对应的寄存器中，注意popal之后就不可以再使用通用寄存器了
- eflags寄存器，可以使用`popfl`指令将数据弹栈到`eflags`中，注意恢复eflags之后就不能使用算数类的指令了，包括`offset(register)`这样的取地址方式也无法使用。最保守的方法是恢复`eflags`之后，只使用`push\pop`
- 恢复`esp`
- `ret`回上一层（原来的eip）。由于ret指令的本质是`pop eip`，因此我们调用`ret`之前就需要将需要pop的指令地址放在**pagefault发生时的栈顶**，完成这一步需要在恢复以上的上下文信息之前

```
// 在pagefault发生时的栈顶push要恢复的eip
addl $8, %esp            //esp指向通用寄存器帧
subl $4, 0x28(%esp)      //utf->utf_esp -= 4
movl 0x28(%esp),%ebx
movl 0x20(%esp),%eax     //utf_eip取到eax中
movl %eax,(%ebx)         //push utf_eip

// 恢复通用寄存器组
popal
// 恢复eflags
addl $4,%esp
popfl
// esp指向原栈顶
popl %esp
// ret
ret
```

exercise 11

完成`set_pgfault_handler()`，用户调用该函数设置自己的缺页处理函数

- 当前用户程序第一次设置`_pgfault_handler`时
    - 为该用户分配用户异常栈的页面`[UXSTACKTOP-PGSIZE,UXSTACKTOP)`

```
r=sys_page_alloc(sys_getenvid(),(void*)(UXSTACKTOP-
PGSIZE),PTE_W|PTE_U|PTE_P);
// 检查 r<0
```

- 将自己的`env_pgfault_upcall`指向`_pgfault_upcall`

  ```
  r=sys_env_set_pgfault_upcall( sys_getenvid(), _pgfault_upcall );
  // 检查 r<0
  ```

- 最后将处理函数`handler`挂载到函数指针`_pgfault_handler`上，之后由`_pgfault_upcall`进行调用

  ```
  _pgfault_handler = handler;
  ```

- 若当前用户程序已经设置过`_pgfault_handler`（也可能是父进程设置的），就只需要挂载`_pgfault_handler = handler`即可

全部代码如下：

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

    if (_pgfault_handler == 0) {
        // First time through!
        // LAB 4: Your code here.
        int r;
        r=sys_page_alloc(sys_getenvid(),(void*)(UXSTACKTOP-
PGSIZE),PTE_W|PTE_U|PTE_P);
        if(r<0){
            panic("set_pgfault_handler: \n");
        }
        r=sys_env_set_pgfault_upcall( sys_getenvid(), _pgfault_upcall );
        if(r<0)
            panic("set_pgfault_handler: \n");
        //panic("set_pgfault_handler not implemented");
    }

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

exercise 12

完成`lib/fork.c`中的`pgfault()` `duppage()` `fork()`

duppage()

该函数为子进程的一个虚拟页面建立物理页面映射，其中

- 父进程该虚拟页面为只读，子进程页面映射到相同物理页，不作变更
- 父亲程该虚拟页面为可写或者`COW (copy-on-write)`，子进程页面映射到相同物理页，但是permission bit需要设置为`COW`，同时父进程的该虚拟页的permission bit也需要更改为`COW`，保证父子进程在写页面时都会重新copy一份再进行写操作，从而不会更改对方的内存信息。

- 子进程先建立映射，父进程再更改（COW情况下）
- `PTE_COW` 不能与 `PTE_W` 同时存在

读取父进程虚拟页面的perm可以通过读`uvpt`数组，在设置用户进程页表时曾经`env_pgdir[PDX(UVPT)] = env_pgdir`(这里简写表意，不考虑perm)，从而将该进程所有的页表按顺序布置在了`[UVPT,UVPT+4MB)`上（只读），这样可以通过`uvpt[pgnum]`读取所需的页表项。

```
static int
duppage(envid_t envid, unsigned pn)
{
    int r;
    pte_t pte;
    void *va = (void*)(pn<<PGSHIFT);
    // 检查父进程中是否有映射存在
    if (!(uvpd[PDX(va)]))
        return -E_INVAL;
    pte = uvpt[pn];
    if( !pte )
        return -E_INVAL;

    // 设置子进程页面的权限
    int perm = PTE_U | PTE_P ;
    // 是否需要COW
    if ( (pte & PTE_W) || ( pte & PTE_COW ) ){
        perm |= PTE_COW;
    }
    // 为子进程设置映射
    if (( r = sys_page_map(sys_getenvid(),va,envid,va,perm)) < 0)
        return r;
    // 若有COW，remap父进程
    if (perm & PTE_COW){
        if (( r= sys_page_map(sys_getenvid(),va,sys_getenvid(),va,PTE_U | PTE_P
| PTE_COW) ) <0 ){
            return r;
        }
    }
    //panic("duppage not implemented");
    return 0;
}
```

pgfault()

该函数处理写COW导致的用户page fault。

- 先检查page fault的原因是否为`write COW`

  ```
  // 先检查是否有映射存在
  if( !(uvpd[PDX(addr)] & PTE_P) )
      panic("no mapping.\n");
  if ( !((pte= uvpt[PGNUM(addr)]) & PTE_P) )
      panic("no mapping.\n");
  // 检查pgfault是否来自 write COW
  if( !((pte & PTE_COW) && (err & FEC_WR)) )
      panic("not write || copy-on-write.\n");
  ```

- 通过上述检查后，需要执行copy-on-write流程

```
// 申请一个新页面，map到PFTEMP
if ((r = sys_page_alloc(sys_getenvid(), PFTEMP, PTE_U|PTE_W|PTE_P)) < 0)
    panic("pgfault: %e\n",r);

// 把旧页面copy到PFTEMP
void* pgva = ROUNDDOWN(addr,PGSIZE);
memcpy((void*)PFTEMP,pgva,PGSIZE);

// 再把旧页面映射到PFTEMP对应的物理页上
r = sys_page_map( sys_getenvid(), PFTEMP, sys_getenvid(), pgva,
PTE_U|PTE_W|PTE_P );
if(r<0)
    panic("pgfault: %e\n",r);

// unmap PFTEMP
r = sys_page_unmap(sys_getenvid(), PFTEMP);
if(r<0)
    panic("pgfault: %e\n",r);
```

全部代码如下：

```
static void
pgfault(struct UTrapframe *utf)
{
    void *addr = (void *) utf->utf_fault_va;
    uint32_t err = utf->utf_err;
    int r;.
    pte_t pte;
    // 检查addr是否对应一个物理页，并且检查pgfault是不是由write COW引起的
    if( !(uvpd[PDX(addr)] & PTE_P) )
        panic("no mapping.\n");
    if ( !((pte= uvpt[PGNUM(addr)]) & PTE_P) )
        panic("no mapping.\n");
    if( !((pte & PTE_COW) && (err & FEC_WR)) )
        panic("not write || copy-on-write.\n");

    // 申请一个新页面，map到PFTEMP
    if ((r = sys_page_alloc(sys_getenvid(), PFTEMP, PTE_U|PTE_W|PTE_P)) < 0)
        panic("pgfault: %e\n",r);
    // 把旧页面copy到PFTEMP
    void* pgva = ROUNDDOWN(addr,PGSIZE);
    memcpy((void*)PFTEMP,pgva,PGSIZE);
    // 再把旧页面映射到PFTEMP对应的物理页上
    r = sys_page_map( sys_getenvid(), PFTEMP, sys_getenvid(), pgva,
PTE_U|PTE_W|PTE_P );
    if(r<0)
        panic("pgfault: %e\n",r);
    // unmap PFTEMP
    r = sys_page_unmap(sys_getenvid(), PFTEMP);
    if(r<0)
        panic("pgfault: %e\n",r);
    //panic("pgfault not implemented");
}
```

fork()

fork程序有两个视角，一个是父进程的视角，一个是子进程的视角，在父进程调用`fork()`之后，子进程也会从fork中继续运行，但由于`sys_exofork()`的返回值不同，子进程接下来执行的fork和父进程执行的fork是不一样的

- 设置`pgfault_handler`为`pgfault()`

  ```
  set_pgfault_handler(pgfault);
  ```

- 调用`sys_exofork()`，从这里开始子进程将会与父进程的执行流分开

  ```
  envid_t child = sys_exofork();
  if(child < 0){
      panic("fork: %e\n",child);
  }
  ```

  - **子进程接下来的执行流**
    - 设置`thisenv`，由于完全复制的父进程的地址空间，这里的`thisenv`是指向父进程的，子进程需要将其更改为自己

      ```
      if(child == 0){
          thisenv = &envs[ENVX(sys_getenvid())];
      }
      ```

    - 随后子进程就可以从fork中退出了，返回值为0
  - **父亲程的执行流**
    - 注意一点，父进程在完成`fork()`之前，子进程不会开始执行（子进程未被设置为`ENV_RUNNABLE`且没有调用`sched_yield()`），因此接下来的操作在时间先后上是先于子进程的
    - 为子进程的每个页面进行`duppage`，设置好`COW`位

      ```
      uint32_t addr = UTEXT;
      pte_t pte;
      pde_t pde;
      for(; addr != UTOP-PGSIZE; addr += PGSIZE){
          if (!(pde = uvpd[PDX(addr)]))
              continue;
          if (!(pte = uvpt[PGNUM(addr)]))
              continue;
          if ((pde & PTE_P) || (pte & PTE_P)){
              if((r=duppage(child,PGNUM(addr)))<0){
                  panic("duppage error: %e\n",r);
              }
          }
      }
      ```

    - 设置子进程的异常栈、异常处理入口。这里**必须由父进程设置**，若由子进程设置，会在设置完成之前就触发page fault，导致子进程无法正常处理page fault

```
r = sys_page_alloc(child,(void*)(UXSTACKTOP-
PGSIZE),PTE_U|PTE_P|PTE_W);
if(r<0){
    panic("fork: cannot alloc Uxstack\n");
}
r = sys_env_set_pgfault_upcall(child,_pgfault_upcall);
if(r<0){
    panic("fork: cannot set pgfault upcall\n");
}
```

- 最后将子进程标记为`ENV_RUNNABLE`等待调度，给父进程返回`child id`

```
r = sys_env_set_status(child,ENV_RUNNABLE);
if(r<0){
    panic("fork: cannot set_env_status.\n");
}
return child;
```

全部代码如下：

```
envid_t
fork(void)
{
    // LAB 4: Your code here.
    set_pgfault_handler(pgfault);
    envid_t child = sys_exofork();
    int r;
    if(child < 0){
        panic("fork: %e\n",child);
    }
    //子进程
    if(child == 0){
        thisenv = &envs[ENVX(sys_getenvid())];
    }
    //父进程
    else{
        uint32_t addr = UTEXT;
        pte_t pte;
        pde_t pde;
        //进行地址空间复制，注意COW
        for(; addr != UTOP-PGSIZE; addr += PGSIZE){
            if (!(pde = uvpd[PDX(addr)]))
                continue;
            if (!(pte = uvpt[PGNUM(addr)]))
                continue;
            if ((pde & PTE_P) || (pte & PTE_P)){
                if((r=duppage(child,PGNUM(addr)))<0){
                    panic("duppage error: %e\n",r);
                }
            }
        }
        r = sys_page_alloc(child,(void*)(UXSTACKTOP-PGSIZE),PTE_U|PTE_P|PTE_W);
        if(r<0){
            panic("fork: cannot alloc Uxstack\n");
        }
        r = sys_env_set_pgfault_upcall(child,_pgfault_upcall);
```

```
        if(r<0){
            panic("fork: cannot set pgfault upcall\n");
        }
        r = sys_env_set_status(child,ENV_RUNNABLE);
        if(r<0){
            panic("fork: cannot set_env_status.\n");
        }
    }
    return child;
    //panic("fork not implemented");

}
```

PART C: Preemptive Multitasking and Inter-Process communication (IPC)

exercise 13

设置`external interrupt(IRQ) 0-15`的`trapentry`，并且设置好其idt表项

- 设置trapentry，注意IRQ没有error_code

```
TRAPHANDLER_NOEC(th_irq_timer,IRQ_OFFSET+IRQ_TIMER)
TRAPHANDLER_NOEC(th_irq_kbd, IRQ_OFFSET+IRQ_KBD)
TRAPHANDLER_NOEC(th_irq_serial, IRQ_OFFSET+IRQ_SERIAL)
TRAPHANDLER_NOEC(th_irq_spurious, IRQ_OFFSET+IRQ_SPURIOUS)
TRAPHANDLER_NOEC(th_irq_ide, IRQ_OFFSET+IRQ_IDE)
TRAPHANDLER_NOEC(th_irq_error, IRQ_OFFSET+IRQ_ERROR)
TRAPHANDLER_NOEC(th_syscall, T_SYSCALL)
```

- 设置idt表项

```
SETGATE(idt[32],0,GD_KT,th_irq_timer,0);
SETGATE(idt[33],0,GD_KT,th_irq_kbd,0);
SETGATE(idt[36],0,GD_KT,th_irq_serial,0);
SETGATE(idt[39],0,GD_KT,th_irq_spurious,0);
SETGATE(idt[46],0,GD_KT,th_irq_ide,0);
SETGATE(idt[51],0,GD_KT,th_irq_error,0);
```

exercise 14

对于`trapno == IRQ_TIMER`的中断，调用`sched_yield()`，运行其他进程，防止一个进程无限占用cpu

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

exercise 15

完成`sys_ipc_recv()` `sys_ipc_try_send()` `ipc_recv()` `ipc_send()`，实现进程间通信。接收者设置自身env结构中的相应字段，告知其他进程消息可以发送；发送者通过设置接收者env结构中的相应字段，告知接收者消息已发送。接受消息可以通过：1).读取env结构中的value字段，此方法只能接受4bytes的消息；2).接收一个页面，接收者设置一个接受页面，接收消息完成后可以查看页面来获知消息。

- **sys_ipc_recv():**
  - 检查`dstva`是否合法，若合法，设置本进程的`env_ipc_dstva = dstva`，不合法，若是`pgaligned`，则设置为`UTOP`；否则就返回一个`-E_INVAL`

    ```
    if ((uintptr_t)dstva < UTOP){
        if ((uint32_t)dstva % PGSIZE){
            return -E_INVAL;
        }
        curenv->env_ipc_dstva = dstva;
    }
    else{
        curenv->env_ipc_dstva = (void*)UTOP;
    }
    ```

  - 置调用进程的状态为`ipc_recving`并且将其设置为`ENV_NOT_RUNNABLE`，即挂起进程直到收到消息

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

  - 返回值
    - 对于一个正常运行的`sys_ipc_recv()`而言，由于在函数结束前就通过`sched_yield()`到了其他进程，其他进程再调用`sched_yield()`运行该进程时，实际上不是从`sys_ipc_recv()`中返回的，因此此处的返回值需要发送方在内核态（即`sys_ipc_send`）中修改接受方的`Trapframe`中的`tf_eax`来指定返回值为0

全部代码如下：

```
static int
sys_ipc_recv(void *dstva)
{
    if ((uintptr_t)dstva < UTOP){
        if ((uint32_t)dstva % PGSIZE){
            return -E_INVAL;
        }
        curenv->env_ipc_dstva = dstva;
    }
    else{
        curenv->env_ipc_dstva = (void*)UTOP;
    }

    curenv->env_ipc_recving = 1;
    curenv->env_status = ENV_NOT_RUNNABLE;
    sched_yield();
    //panic("sys_ipc_recv not implemented");
```

```
    return 0;
}
```

- **sys_ipc_try_send()**
  - 获得目的进程的`env`结构指针，此处不需要检查权限

    ```
    int r = envid2env(envid,&e,0);
    if(r<0){
        return r;
    }
    ```

  - 检查目的进程是否处于接受消息的状态之中，若否，则**本次尝试失败**

    ```
    if(!(e->env_ipc_recving)){
        return -E_IPC_NOT_RECV;
    }
    ```

  - 检查要求发送的页面是否合法，同时要检查目的进程是否打算接受一个页面，同时还要检查`perm`是否合法
    - 检查发送的页面是否合法（此处已经检查过是否小于`UTOP`）

      ```
      if ((uintptr_t)srcva % PGSIZE){
          return -E_INVAL;
      }
      // 检查是否存在映射
      if(!(pp=page_lookup(curenv->env_pgdir,srcva,&pte))){
          return -E_INVAL;
      }
      ```

    - 检查`perm`是否合法：不能与现有的`permission bit`冲突；不能有`PTE_SYSCALL`以外的权限位

      ```
      if (!(*pte & PTE_W) && (perm & PTE_W)){
          return -E_INVAL;
      }
      if( (perm & (PTE_U | PTE_P)) != (PTE_U | PTE_P)
         || (perm & (~PTE_SYSCALL)) ){
          return -E_INVAL;
      }
      ```

    - 发送页面：将目的进程的`dstva`映射到发送方的物理页上

      ```
      e->env_ipc_perm = perm;
      r = page_insert(e->env_pgdir,pp,e->env_ipc_dstva,perm);
      if(r<0){
          return r;
      }
      ```

  - 剩余的部分为发送`value`，设置接受方的`sys_ipc_recv()`返回值，设置接受方的接受状态

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

全部代码如下：

```
static int
sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned perm)
{
    // LAB 4: Your code here.
    struct Env *e;
    struct PageInfo *pp;
    pte_t *pte;
    int r = envid2env(envid,&e,0);
    if(r<0){
        return r;
    }
    if(!(e->env_ipc_recving)){
        return -E_IPC_NOT_RECV;
    }
    e->env_ipc_perm = 0;
    if ( (uintptr_t)(e->env_ipc_dstva) < UTOP && (uintptr_t)srcva<UTOP){
        if ((uintptr_t)srcva % PGSIZE){
            return -E_INVAL;
        }
        // 检查是否存在映射
        if(!(pp=page_lookup(curenv->env_pgdir,srcva,&pte))){
            return -E_INVAL;
        }
        if (!(*pte & PTE_W) && (perm & PTE_W)){
            return -E_INVAL;
        }
        if( (perm & (PTE_U | PTE_P)) != (PTE_U | PTE_P)
                || (perm & (~PTE_SYSCALL)) ){
            return -E_INVAL;
        }
        e->env_ipc_perm = perm;
        r = page_insert(e->env_pgdir,pp,e->env_ipc_dstva,perm);
        if(r<0){
            return r;
        }
    }
    //发送消息，设置目标env的相应字段
    e->env_ipc_recving = 0;
    e->env_ipc_value = value;
    e->env_ipc_from = curenv->env_id;
    e->env_status = ENV_RUNNABLE;
    e->env_tf.tf_regs.reg_eax = 0;
    return 0;
    //panic("sys_ipc_try_send not implemented");
}
```

- **ipc_recv()**
  - 设置接受页面，若为`null`，则将其设置为`UTOP`

    ```
    int r;
    pg = pg ? pg: (void*)UTOP;
    ```

  - 进行接受，接受完毕后，设置好来源和perm，返回接受到的value（或者返回错误信息）

    ```
    if (from_env_store){
        *from_env_store = r<0 ? 0 : thisenv->env_ipc_from;
    }
    if(perm_store){
        *perm_store = r<0 ? 0 : thisenv->env_ipc_perm;
    }
    return r<0 ? r : thisenv->env_ipc_value;
    ```

  全部代码如下：

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

- **ipc_send()**
  - 检查发送的页面是否为`null`，若为`null`，则将其设置为`UTOP`

    ```
    pg = pg ? pg : (void*)UTOP;
    ```

  - 接下来，反复尝试发送消息，直到成功。失败的标志为`sys_ipc_try_send()`的返回值为`-E_IPC_NOT_RECV`，失败时需要调用`sys_yield()`将，保证不会过度地占用CPU（在`yield`的间隙中目标进程也有可能进入接受消息状态）

    ```
    RETRY:
        r = sys_ipc_try_send(to_env,val,pg,perm);
        if (r<0 && r!=-E_IPC_NOT_RECV){
            panic("ipc_send: %e\n",r);
        }
        if (r == -E_IPC_NOT_RECV){
            sys_yield();
            goto RETRY;
        }
    ```

全部代码如下：

```
void
ipc_send(envid_t to_env, uint32_t val, void *pg, int perm)
{
    // LAB 4: Your code here.
    pg = pg ? pg : (void*)UTOP;
    int r;
    // 反复尝试发送，若接受方不处于接受状态，继续发送
    // 同时需要 yield
RETRY:
    r = sys_ipc_try_send(to_env,val,pg,perm);
    if (r<0 && r!=-E_IPC_NOT_RECV){
        panic("ipc_send: %e\n",r);
    }
    if (r == -E_IPC_NOT_RECV){
        sys_yield();
        goto RETRY;
    }
    //panic("ipc_send not implemented");
}
```

exercise 15结束后，`make grade`获得满分，截图已放在**报告开头**

补充：优先级调度

首先，在`struct Env`中添加一个优先级字段，其中`priority`是`int32`类型的`typdef`

当`env_alloc()`时，将`priority`字段设置为进程在`envs`中的下标

```
e->env_priority = e-envs;
```

在`kern/sched.c`中添加了一个`sched_priority()`函数，每次调度都遍历`envs`，找到优先级最大的`ENV_RUNNABLE`进程，若未找到，就查看`curenv`能否再运行

```
void sched_priority_yield(void){
    struct Env *idle;
    envid_t i,maxi=-1;
    idle = curenv;
    // 寻找可以run的优先级最大的进程
    priority max=-1;
    for(i=0; i<NENV; i++){
        if(envs[i].env_status == ENV_RUNNABLE && envs[i].env_priority>max){
            max = envs[i].env_priority;
            maxi = i;
        }
    }
    // 未找到
    if (maxi<0){
        if(idle && (idle->env_status==ENV_RUNNING || idle-
>env_status==ENV_RUNNABLE)){
            env_run(idle);
        }
    }
```

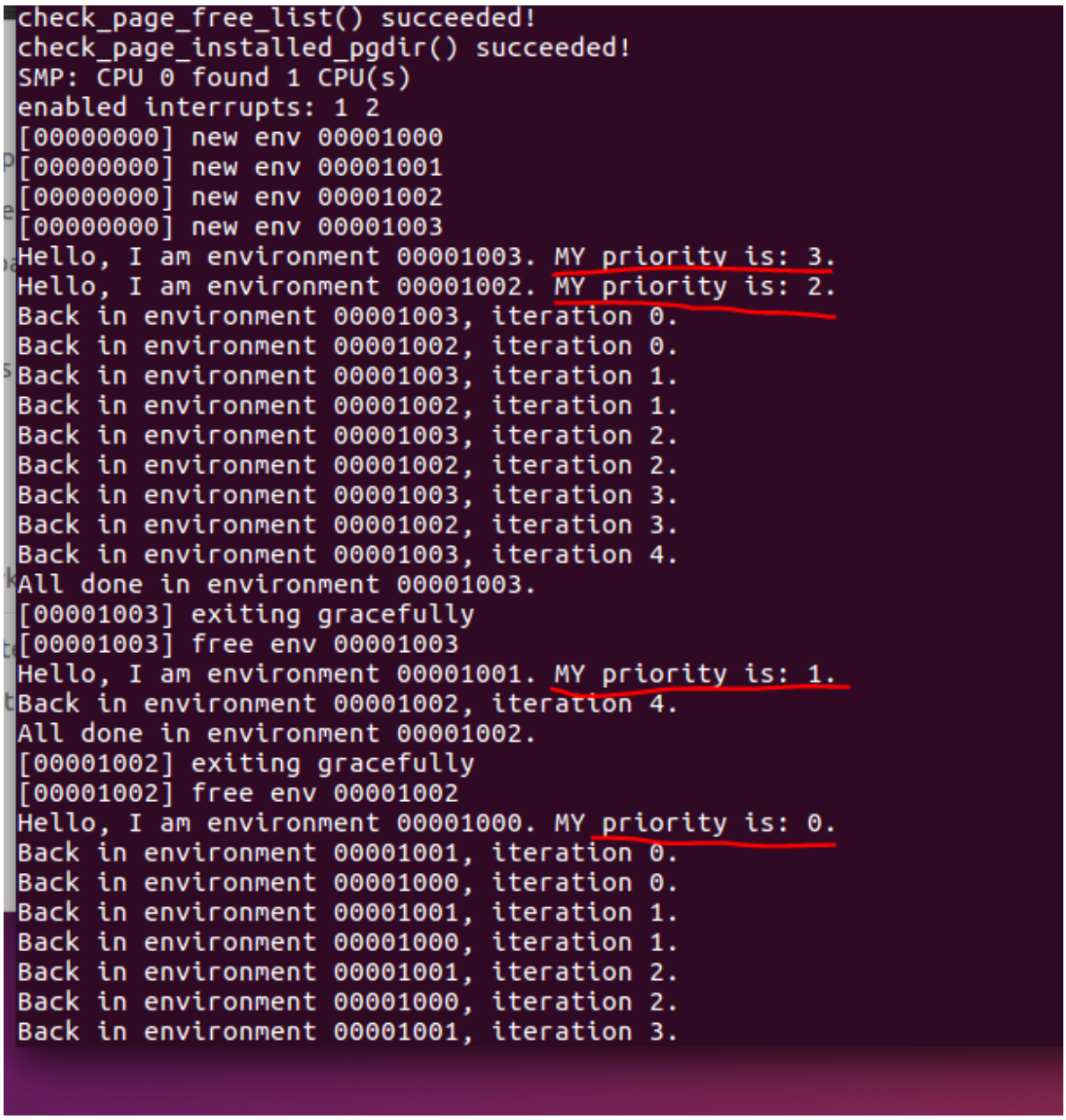
```
    // 找到
    else{
        env_run(&envs[maxi]);
    }
    sched_halt();
}
```

然后再添加一个系统调用`sys_priority_yield()`作为用户调用`sched_priority_yield()`的接口。编写了用户程序`user/priority_yield.c`，如下。将其加入到`kern/Makefrag`中，使得make能够编译它。

```
#include <inc/lib.h>
void
umain(int argc, char **argv)
{
    int i;

    cprintf("Hello, I am environment %08x. MY priority is: %d.\n", thisenv-
>env_id,thisenv->env_priority);
    for (i = 0; i < 5; i++) {
        sys_priority_yield();
        cprintf("Back in environment %08x, iteration %d.\n",thisenv->env_id, i);
    }
    cprintf("All done in environment %08x.\n", thisenv->env_id);
}
```

修改jos中各个调用`sched_yield()`的地方为`sched_priority_yield()`，在`i386_init`中create四个`user-priority_yield`，并且`make qemu`，得到的结果如下：



问题回答

(1) 详细描述 JOS 启动多个 APs（Application Processors）的过程

jos启动，BSP调用`i386_init()`。为了唤醒Aps，BSP先给内核加锁，随后调用了`boot_aps()`。

在`boot_aps()`中，BSP将依次唤醒AP。首先它将`mpentry.S`中定义的指令复制到地址`code`处，随后令每个AP都运行`mpentry.S`中的指令，并且等待该AP运行完毕。

每个执行`mpentry.S`指令的AP，接下来都会调用`mp_main()`，在mp_main中：

- 每个AP先装载内核模式下的页目录
- 调用lapic_init()，初始化自己的`LAPIC`
- 调用`env_init_percpu()`，加载GDT以及段描述符
- 调用`trap_init_percpu()`，初始化并且加载每个CPU的`TSS`以及`IDT`
- 置本CPU的状态为`CPU_STRATED`，使BSP得知该AP已经启动
- 试图占用内核（准备运行用户程序），由于此时内核是被锁的，AP会阻塞在自己的`lock_kernel()`请求上。

之后等待BSP唤醒所有AP，并且运行了用户程序后，BSP会释放内核，APs就可以获得内核锁并运行用户程序了

(2)

a). 调用fork时，用户程序分为父进程和子进程

对于父进程：

- 设置缺页处理函数为`pgfault()`，并且调用`sys_exofork()`来获得自己的子进程，`set_pgfault_handler()`中会调用`sys_env_set_pgfault_upcall()`
- 此时子进程还不完善，父进程接下来：
  - 复制自己的地址空间映射给子进程，其中可写页面在map时要标记为`copy-on-write`。可写页面包括父进程具有`PTE_W 或者 PTE_COW`的页面。子进程标记为`PTE_COW`的页面，父进程随后也要标记自己的相同页面为`PTE_COW`，这一步可以使用我们定义的`duppage()`函数。这个过程中会调用`sys_page_map()`
  - 为子进程的缺页异常处理设置必须的条件，包括
    - 为子进程的异常处理栈分配页面，这一步会调用`sys_page_alloc()`
    - 将子进程env结构中的`env_pgfault_upcall`指向`_pgfault_upcall`，这一步会调用`sys_env_set_pgfault_upcall()`
    - 将子进程的状态设为可以运行，这一步会调用`sys_env_set_status()`

对于子进程：

- 将从`sys_fork()`结束处开始运行，接下来会设置自己的`thisenv`，会调用`sys_getenvid()`

b). 发生COW相关 pagefault 时：

- mmu 产生了page fault中断，cpu自动切换到内核栈上，并且将部分`Trapframe`布置到栈上，同时会压栈一个`error code`（处理器自动完成）
- 随后cpu查找`idt`表，根据`trapno`进行跳转，这里会跳转到page fault对应的`entry`，随后内核会补充完剩下的`Trapframe`，随后调用`trap()`（内核中完成）
- `trap()---> trap_dispatch() ---> page_fault_handler()`，在此处，内核检查用户是否有设置缺页处理函数，若有，则内核会在用户的异常处理栈（`UXSTACKTOP向低地址增长`）上布置`UTrapframe`，随后切换到用户异常处理栈上，并且将控制流返还给用户程序（执行用户的`_pgfault_upcall`）（内核中完成）（接下来的步骤都发生在用户空间中）
- 在`_pgfault_upcall`中，用户调用自己设置的`handler`，也就是`fork.c/pgfault()`，在该函数中，用户通过系统调用，为自己申请一个新的页面，复制旧的`COW`页面到新页面，并且把原来映射到旧页面的地址映射到新页面
- `pgfault()`返回后，回到`_pgfault_upcall`，接下来`_pgfault_upcall`会恢复上下文，直接返回并继续运行用户进程。

(3)

primes.c中，存在一个进程链`envs[0]--->envs[1]--->envs[2]--->···--->envs[NENVS-1]`，其中`envs[i]`是`envs[i+1]`的`parent`。整个计算过程如下

- **a**. 根进程`envs[0]` fork 一个子进程`envs[1]`，并尝试给 envs[1] 发送整数 `2`，此时尝试失败，会唤醒`envs[1]`，随后`envs[1]`进入`ipc_recv`，再度唤醒`envs[0]`，`envs[0]`再次尝试发送2，此次成功；唤醒`envs[1]`，随后进入以下流程
- **b**. 父进程`envs[i], i=1,2,3···,NENVS-1`生成子进程`envs[i+1]`，随后父进程进入recv状态，yield到`envs[i+1]`，由于其为子进程，所以`envs[i+1]`也进入recv状态，yield到`envs[0]`
- **c**. `envs[0]`尝试发送整数`j`，此时`envs[1]`到`envs[i+1]`全为recv状态（可以归纳证明），于是`j`能够沿着进程链传播到`envs[k]`，此时要么`k==i+1`，要么`j % primes[k] == 0`，皆无法继续往下发送。
  - 若`k==i+1`，`envs[i+1]`获得`primes[i+1]`，随后离开`primeproc`中的`top`循环，`i=i+1`并返回过程 **b**。
  - 若`k<i+1 && j % primes[k]==0`，envs[k] 结束recv，并且不会进行send，于是循环又进入recv状态，此时，`envs[1]`到`envs[k]`全部都是send之后又进入了recv状态，`envs[k]`到`envs[i+1]`处于recv状态中仍未退出，于是`envs[1]`到`envs[i+1]`保持了全为`recv`的状态，`j = j+1`，并且`envs[k]`yield到了`envs[0]`，返回 **c**
- **d**. 全部envs用尽，程序结束

以上的执行流程是在`Round robin`调度算法下的具体执行流程。而`primes.c`实际上是为每一个进程`envs[i]`计算了第`i`大的质数`primes[i]`，用图像简化说明如下:

```
// 进程envs[i]向子进程发送一列数
// 这列数均不是 prims[k],for k=1...i 的倍数
// 这列数的第一个数作为 primes[i+1]

envs[0] ---(send 2 3 4 5···)---> envs[1] ---(send 3 5 6···)--> envs[2]----------
>
envs[3] ---(send number sequence) ··· ------> envs[i]---(send number sequence)--
>
envs[i+1] ---> ····· --------> envs[NENV-1]

// 程序中发送的数列没有上界，这里取一个上界为20的例子
envs[0]: 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
envs[1]: 2 | 3 5 7 9 11 13 15 17 19
envs[2]: 3 | 5 7 11 13 17 19
// 这里实际上已经筛完了，剩余全部都为质数
```

这个过程实际上是**埃拉托斯特尼筛法**，即第k轮筛去第k大质数的所有倍数，筛后的数列的第一位为第k+1大的质数

简单归纳证明如下

假设第i大的质数为`primes[i]`，数列`a[0],a[1],····`，满足`任意j、k，a[j] % primes[k] != 0, for k=1...i-1`，即现有数列`a`是已经对`primes[1] primes[2] ··· primes[i-1]`进行筛选得到的数列。

接下来数列`a`对`primes[i]`进行筛选操作，得到数列`b`，则`b[0]`一定是一个质数。若`b[0]`不是质数，则`存在k,使b[0] % primes[k]==0 且 b[0] > primes[k]`，该数应在第k轮对`primes[k]`的筛选中被去除，不可能出现在当前数列中，因此`b[0]`一定是质数，且一定是第`i+1`大的质数。primes.c的流程相当于每次把`primes[i+1] = b[0]`。因此每个进程持有的`primes[i]`确实是第`i`大的质数。