操作系统及实习（实验班）

JOS-Lab4
抢占式多任务
实习报告

姓名 杜若谷 学号 1300012855
日期 2016.4.20

目录

内容一：总体概述

这个 Lab 分为 A、B、C 三个部分。

Part A 主要工作是完成 JOS 对多处理器的支持，并且实现系统调用实现用户创建新进程的功能。随后实现非抢占式的 Round-Robin 调度算法。

Part B 是围绕实现 Copy-on-Write 的 fork 函数，进行一些工作。

Part C 则是实现有关进程间通信的两个系统调用，以及包装它们的库函数。

内容二：任务完成情况

任务完成列表（Y/N）

Exercise 1	Exercise 2	Exercise 3	Exercise 4	Exercise 5	Exercise 6
Y	Y	Y	Y	Y	Y
Exercise 7	Exercise 8	Exercise 9	Exercise 10	Exercise 11	Exercise 12
Y	Y	Y	Y	Y	Y
Exercise 13	Exercise 14	Exercise 15			Challenge2
Y	Y	Y			Y

具体 Exercise 的完成情况

第一部分

这一部分的主要目的是实现 JOS 对多处理器的支持，随后增加系统调用以便用户进程创建新进程，然后实现协同式(Cooperative)轮转调度。所谓协同式调度，就是在操作系统课上所讲的非抢占式调度i，直到用户进程主动交出 CPU，操作系统都不会执行上下文切换，因此这一部分中不用实现时间片的功能。

Exercise 1

我们即将支持的多处理器模型为 SMP 模型，即所有的处理器都可以无差别的访问存储器，并且功能上都是相同的。但是在启动阶段则需要首先启动被称为 Bootstrap Processor(BSP)的存储器来初始化系统并且启动 OS，并且初始化其他的处理器，被 BSP 启动的处理器成为 Application Processor(AP)。这个 BSP 是由 BIOS 设置的。

在 x86 体系结构中，每个处理器都有一个本地 APIC 单元(LAPIC)。LAPIC 单元是为处理器提供服务的中断控制器，其中我们即将使用有几项功能：读取当前 CPU 的编号(cpunum())；接收 BSP 发出的 STRATUP 中断，使 AP 启动；使用 LAPIC 的时钟中断功能实现抢占式调度。

处理器访问 LAPIC 时使用了 Memory-mapped IO(MMIO)模式，物理地址从

0xFE000000 开始，并且映射到从 MMIO 开始的虚拟地址。这个 Exercise 的工作就是完成 kern/pmap.c 中的 mmio_map_region()函数来实现内存映射。值得注意的是这里的页表权限位应当打开 PTE_PCD|PTE_PWT，意为关闭 Cache 和写回。因为这里使用的 MMIO 如果使用 Cache 可能无法保证一致性。

Exercise 2

在进行这项 Exercise 之前，需要了解 kern/mpconfig.c 中 mp_init 函数的功能，即获取 CPU 数，APIC ID 和每个 LAPIC 的 MMIO 地址，这些信息在 BIOS 中的 MP configuration table 中。

随后我们应该完成 boot_aps 函数来启动所有的 AP。AP 启动的过程和 BSP 类似，首先从实模式中启动，随后开始执行类似 bootloader 的代码。但是这里我们已经可以控制开始执行的位置，所以只需将代码(kern/mpentry.s)，拷贝到 MPENTRY_PADDR ，这段代码开启保护模式和分页后调用 kern/init.c 的 mp_main()函数。比较值得注意的是 RELOC 和 MPBOOTPHYS 这两个宏。MPBOOTPHYS 这个宏的功能是计算物理地址，由于连接的位置不固定，所以在代码中的物理地址需要我们自己计算。RELOC 的功能是将 KERNBASE 以上的高地址换算为低地址，因为开启分页之前我们还运行在低地址。

```
# 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

#define RELOC(x) ((x) - KERNBASE)
#define MPBOOTPHYS(s) ((s) - mpentry_start + MPENTRY_PADDR)
```

随后我们需要修改 kern/pmap.c 的 page_init()。防止将 MPENTRY_PADDR 的这一页加入 page_free_list。

```
extern unsigned char mpentry_start[], mpentry_end[];

struct PageInfo *p = pages;
p++;        // skip the first page
for (; p < pa2page((physaddr_t)MPENTRY_PADDR); p++) {
    p->pp_ref = 0;
    p->pp_link = page_free_list;
    page_free_list = p;
}

p = pa2page(ROUNDUP(mpentry_end - mpentry_start + MPENTRY_PADDR, PGSIZE));

for (; p < pa2page((physaddr_t)IOPHYSMEM); p++) {
    p->pp_ref = 0;
    p->pp_link = page_free_list;
    page_free_list = p;
}
for(p = pa2page(PADDR(boot_alloc(0))); p < pages+npages; p++) {
    p->pp_ref = 0;
    p->pp_link = page_free_list;
    page_free_list = p;
}
```

Question 1

由于我们不能确认 mpentry.S 的链接位置，因此链接地址都是不固定的。但是在进入时这一段代码运行在实模式，所以变量的地址应该为物理地址，因此需要用 MPBOOTPHYS 来换算。

Exercise 3

在启动 AP 之前，需要 BSP 针对每个 AP 初始化环境。这些环境被记录在 kern/cpu.h 中。在这个 Exercise 中，我们需要为每个处理器上的内核设置不同的栈。这样做的原因是每个处理器都有可能同时进入内核，所以不能使用相同的栈。这段代码在 kern/pmap.c 的 mem_init_mp 中。

完成代码后，运行结果如下:

```
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
check_page() succeeded!
kernel panic on CPU 0 at kern/pmap.c:489: boot_map_region: PTE already exist
```

```
K> backtrace
Stack backtrace:
  ebp f011dea8  eip f0100be3  args 00000001 f011dec0 00000000 f011df4c f022ba80
         kern/monitor.c:210: monitor+275
  ebp f011df18  eip f01000a6  args 00000000 f011df4c f0107895 000001e9 0022d000
         kern/init.c:155: _panic+102
  ebp f011df38  eip f0101441  args f0107895 000001e9 f0107034 f01015a1 f026e000
         kern/pmap.c:490: boot_map_region+101
  ebp f011df78  eip f0103098  args 0022d000 00000002 effff000 00000002 f010425d
         kern/pmap.c:297: mem_init+6676
  ebp f011dfd8  eip f01000ef  args f01068ac 00001aac 000426a9 00000000 00000000
         kern/init.c:41: i386_init+71
  ebp f011dff8  eip f010003e  args 0011f021 00000000 00000000 00000000 00000000
         kern/entry.S:84: <unknown>+0
```

可以看出，这个函数中出现了 Error，原因则是 PTE 已经存在。其实原因很好理解，因为我们之前已经把 KERNBASE 以上的部分都已经映射到页表了。因此我们需要进行清理。

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

最终结果如下，已经顺利通过测试。

```
6828 decimal is 15254 octal!
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() 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:335: kernel fault va 00000000 ip f0106519

Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
```

Exercise 4

每个 CPU 还需要记录当前 CPU 正在运行的环境，即 Env 结构的指针。这个结构也记录在 kern/cpu.h 的 CpuInfo 结构中。

在这个 Exercise 中，我们需要要为每个 CPU 设置 TSS 和 TSS 描述符。在 JOS 中 TSS 的作用是在用户态进入和离开内核态时切换堆栈。我们需要完成 kern/trap.c 的 trap_init_percpu()函数。

仿照已有的代码和 Hint，很容易就可以完成这个函数。应该注意的是不止有关 CPU 的部分需要修改，所有和堆栈有关的位置也应该修改。

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

// 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 + (cpunum() << 3));

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

最后每个 CPU 的系统有关的寄存器也需要修改，比如 GDTR，IDTR，CR3 等。但是这一部分 JOS 已经实现在 env_init_percpu()和 trap_init_percpu()这两个函数中。

每个 CPU 状态的设置已经完成，这一部分的测试可以使用 make qemu CPUS=4 来模拟出 4 核 CPU 来测试之前的完成情况。

```
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() 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
```

可以看出 CPU1，2，3 已经启动。然而这时应该有 4 个 CPU。这是否意味着有一个 CPU 没有启动呢？当然并不是这样，这个信息是 SMP 发出的，ID 为 0 号，SMP 已经启动编号为 1、2、3 的 AP，所以此时 4 个 CPU 都已经启动了。

Exercise 5

多处理器一个明显的问题就是同步与互斥的问题。目前我们需要解决的就是同时访问内核的问题。根据之前的设置，很容易发现多个处理器可以同时访问内核，由于我们已经为每个 CPU 设置了独立的内核栈，所以不用担心栈冲突的问题。但是内核可能还有其他临界资源需要防止多个 CPU 同时读或写。这些资源包括但不限于页分配、控制台、调度器以及 IPC 部分。

所以我们即将使用加自旋锁的方法，来实现对这些资源的互斥。所有有关自旋锁的函数都在 kern/spin.c 中。具体可以使用的方法包括为这些临界资源分别设置锁，在进入临界区时加锁，在离开时解锁。但是 JOS 使用了更为简单的解决方案，即使用一个唯一的 Big Kernel Lock 在所有进入内核的部分加锁，在所有离开内核的部分解锁。这样就会造成在任意时刻最多只能有 1 个 CPU 在内核中访问临界资源，从而解决了竞争问题。

但是这样做明显会造成很多资源浪费，因为并不是内核的每一个部分都是临界资源，在访问非临界区时明显可以多个 CPU 并发访问而不会导致竞争。所以并非没有更好的解决方案。

在这个 Exercise 中，我们需要完成的工作就是在所有进入内核的部分调用 lock_kernel()加锁，这些位置包括：

1) 函数 i386_init()中，在 BSP 启动其他 AP 之前；
2) 函数 mp_main()中，完成初始化 AP 之后，调用 sched_yield()之前；
3) 函数 trap()中，从用户态进入内核态时。

前两个位置都是系统初始化的时候进入内核时加锁，而第三个位置则是用户程序运行后再次进入内核时加锁。

我们还需要在离开内核时调用 unlock_kernel()解锁。由于离开内核有统一的接口 env_run()，因此只需在这里进入用户态之前调用即可。

接下来我们可以研究一下自旋锁的实现。在 kern/spinlock.c 中，我们可以看到实现加锁的函数为 spin_lock()，其中核心代码只有两行：

```
while (xchg(&lk->locked, 1) != 0)
    asm volatile ("pause");
```

可见这个这个自旋锁就是使用 XCHG 指令来实现的。但是这里的 xchg 是经过包装的函数，并非简单的一条指令。具体内容如下：

```
static inline uint32_t
xchg(volatile uint32_t *addr, uint32_t newval)
{
    uint32_t result;

    // The + in "+m" denotes a read-modify-write operand.
    asm volatile("lock; xchgl %0, %1" :
                 "+m" (*addr), "=a" (result) :
                 "1" (newval) :
                 "cc");
    return result;
}
```

比较值得关注的是 xchgl 指令之前增加了 lock，意为锁住总线，这样就能防止两个 CPU 同时取到锁了。

解锁的函数为 spin_unlock()，核心代码只有一行：

```
xchg(&lk->locked, 0);
```

这样就可以直接解锁了。

Question 2

虽然 Big Kernel Lock 能够保证同时只能有至多一个 CPU 访问临界资源，但是并没有保证所有 CPU 都不能同时进入内核。如果有多个 CPU 同时发生中断，那

么每个 CPU 都会尝试在栈中构建 Trapframe，如果只有一个栈，那么肯定会发生冲突。所以还是要为每个 CPU 设立一个独立的栈。

还有一个原因，即正常使用的系统不可能同时只有一个 CPU 能够访问内核，因此最终还是会向这个方向改进，所以提前设立独立的栈可能还有这个考虑。

Exercise 6

在这一个部分我们需要实现一个 Round-Robin 调度算法。在 kern/sched.c 中的 sched_yield()函数中实现调度。这个算法非常简单，即从 envs 数组中当前运行的进程的下一位开始顺寻寻找状态为 ENV_RUNNABLE 的进程来调度。如果实在没有可以调度的进程，可以查看当前进程状态是否为 ENV_RUNNING，如果是的话必须查看当前 CPU 是否是正在运行这个进程的 CPU,如果是则可以继续运行。如果不是的话也不能调度。因为状态为 RUNNING 的进程的状态还没有保存到 Env 构里，如果这个时候切换 CPU，会造成之前运行的状态丢失。所以不能切换。

```
int i = (ENVX(curenv->envid) + 1) % NENV;

while(i != ENVX(curenv->envid)) {
    if(envs[i].env_status == ENV_RUNNABLE)
        break;
    i = (i + 1) % NENV;
}

if(env[i].env_status == ENV_RUNNABLE) {
    env_run(envs[i]);
}
else if(envs[i].env_status == ENV_RUNNING
    && envs[i].env_cpunum == cpunum()) {
    env_run(envs[i]);
}
```

值得注意的在这段代码中，我们仍然能够使用 curenv 变量。而在之前的 lab 中，这个变量的类型为 struct Env*。但是我们目前已经支持了多处理器，同时可能有不止一个进程在运行，那么 curenv 为什么还能使用呢？

原来，在这个 lab 中偷偷地将 curenv 换成了一个宏：

```
extern struct Env *envs;          // All environments
#define curenv (thiscpu->cpu_env)       // Current environment
extern struct Segdesc gdt[];
```

在这个宏中，从当前 CPU 中的 CpuInfo 读取这个 CPU 上的进程，所以这个 curenv 和之前的版本就能无缝切换了。

在完成这一部分后，我们可以通过在系统初始化时加入多个 user_yield 用户程序来调度，加入办法如下：

```
#if defined(TEST)
    // Don't touch -- used by grading script!
    ENV_CREATE(TEST, ENV_TYPE_USER);
#else
    // Touch all you want.
    //ENV_CREATE(user_primes, ENV_TYPE_USER);
    ENV_CREATE(user_yield, ENV_TYPE_USER);
    ENV_CREATE(user_yield, ENV_TYPE_USER);
    ENV_CREATE(user_yield, ENV_TYPE_USER);
#endif // TEST*
```

课件我们自己所加的进程都必须在 else 部分。这样做的原因是如果#define TEST，表明当前正在使用 make grade 进行测试，所以我们不能修改这一部分，但是下面的部分就是我们自己测试时可以使用的了。

接下来进行测试，出现了如下错误：

```
kernel panic on CPU 0 at kern/trap.c:337: kernel fault va 00000048 ip f0104e3c
(gdb) break *0xf0104e3c
Breakpoint 1 at 0xf0104e3c: file kern/sched.c, line 33.
```

查看这一部分的代码，发现原因非常简单，因为 curenv 不一定存在，所以需要分情况处理：

```
int i, ori_envx;
if(curenv) {
    ori_envx = (ENVX(curenv->env_id) + 1) % NENV;
    i = (ori_envx + 1) % NENV;
}
else {
    ori_envx = NENV - 1;
    i = 0;
}
```

修改后测试结果如下：

```
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2
[00000000] new env 00001000
yield_start
yield: env_cpu: 0 cur_cpu 0
Hello, I am environment 00001000.
yield_start
yield_halt
TRAP frame at 0xefffffbc from CPU 0
  edi  0xf0291044
  esi  0x0000007c
  ebp  0x00000000
  oesp 0xefffffdc
  ebx  0x0000007c
  edx  0xf022d020
  ecx  0x00000002
  eax  0xf0000000
  es   0x----0010
  ds   0x----0010
  trap 0x0000000d General Protection
  err  0x00000102
  eip  0xf0104e02
  cs   0x----0008
  flag 0x00000246
kernel panic on CPU 0 at kern/trap.c:256: unhandled trap in kernel
```

这样很像出现了错误，但是确实正常的，因为我们目前还没有处理 LAPIC 的 IRQ 中断以及时钟中断，因此会出现这样的 unhandled trap，而且出现 General Protection 错误，因为我们根本就没有设置这些中断的 IDT。

Question 3

虽然在 env_run()中即使切换了 CR3 还能引用 Env 结构 e 的原因是:这个结构并非存储在用户部分（UTOP 以下）而是存储在内核中。而且 e 本身也存储在内核的栈中，因为我们生成 Env 的页表页目录时，已经将内核部分都映射到相同的物理页上，因此即使切换到用户态的页表页目录，我们在内核中还是可以引用这些内容。

Question 4

进程切换时，如果不把寄存器状态保存，就会造成将来无法恢复这个进程。保存的位置在每次用户程序通过中断进入内核后保存了 Trapframe，进入 kern/trap.c 的 trap()函数后，会判断是否是从用户态进入的内核，如果是则会将 Trapframe 的内容保存在 curenv->env_tf 中。

```
if ((tf->tf_cs & 3) == 3) {
    // Trapped from user mode.
    // Acquire the big kernel lock before doing any
    // serious kernel work.
    // LAB 4: Your code here.
    lock_kernel();

    assert(curenv);

    // Garbage collect if current enviroment is a zombie
    if (curenv->env_status == ENV_DYING) {
        env_free(curenv);
        curenv = NULL;
        sched_yield();
    }

    // Copy trap frame (which is currently on the stack)
    // into 'curenv->env_tf', so that running the environment
    // will restart at the trap point.
    curenv->env_tf = *tf;
    // The trapframe on the stack should be ignored from here on.
    tf = &curenv->env_tf;
}
```

Exercise 7

Part A 的最后一部分就是实现环境（Environment）的创建有关的系统调用。所谓环境，即 JOS 中的进程的特有名词，每个环境有一个 Env 结构体（即 PCB）. 这个结构体中存储了进程的 ID、状态、页表、寄存器状态等内容。

目前所有的进程都是由系统初始化时创建，如果要实现用户进程创建新进程的功能，则一定要通过系统调用。在 UNIX 中，系统提供 fork 调用来实现创建进

程。这个调用会创建一个和父进程完全相同的子进程，唯一的不同是在父进程中 fork 会返回子进程的 PID，而在子进程中则会返回 0。虽然父进程和子进程几乎相同，地址空间的内容也相同，但是他们的地址空间是独立的，任何一个进程对于内存的修改不会影响另外一个进程。

要实现 fork 这个库函数，我们需要以下几个库函数：

1) sys_exofork：创建一个新的（空白）进程，在用户空间内几乎为空，初始状态下新进程是无法运行的。这个系统调用最后返回新进程的 envid；
2) sys_env_set_status：修改进程的状态；
3) sys_page_alloc：将一个新的物理页映射到指定进程的指定虚拟地址；
4) sys_page_map：将一个进程的某一虚拟地址的物理页映射到另一进程的某一虚拟地址；
5) sys_page_unmap：将指定进程的某一虚拟地址的物理页解除映射。

这些系统调用有一个共同点，就是需要对传入的 envid 进行检查，首先是这个 envid 是否合法，其次就是当前进程是否对其操作的进程有权限（即只能操纵自己和子进程）。如果违反权限则需要报错。

另外就是在 sys_page_alloc 和 sys_page_map 时需要检查地址是否在 UTOP 一下，一个用户进程显然不应该修改 UTOP 以上的内核部分的。

系统调用 sys_exofork 子进程返回时需要得到 0，但是这并不是通过函数返回实现的，而是通过将 Trapframe 的 eax 设置为 0，这样子进程运行时就会以为之前的函数调用返回了 0.

```
struct Env *e;
int err;
if((err = env_alloc(&e, curenv->env_id)) != 0)
    return err;

e->env_status = ENV_NOT_RUNNABLE;
e->env_tf = curenv->env_tf;
e->env_tf.tf_regs.reg_eax = 0;
```

第二部分

这一部分的工作是实现 Copy-on-Write 的 fork 库函数。这个函数的功能已经在上面描述过了，但是 Copy-on-Write(COW)的特性确是我们需要完成的。

在早期 UNIX 中，fork 函数只是简单的将父进程的用户地址空间一页一页的复制到子进程的地址空间（例如目前的 dumpfork）。这样实现虽然简单明了但是却有致命的缺点，即过于浪费时间和空间。很多时候执行完 fork，子进程会立刻运行一个新的程序(比如 shell 程序)，这样的话之前复制的大量内容都要再次删除，相当于做了很多无用功，因此我们需要更好的解决方案。

COW 就是这样一种技术，即在 fork 时父进程创建子进程并不复制内容，而是父进程和子进程共享一些物理页，在任何一方需要修改时则通过将原内容复制到独立的新位置，随后解除原来的映射，改为映射到新的位置。这样就可以实现简便的 fork 函数了。在这一部分中我们就是要实现这样的 fork 函数。

Exercise 8

由于 COW 的设置，对写保护的页面发生写操作时会发生 Page Fault，而这些 Page Fault 需要交给 fork 用户程序自己处理。所以我们需要实现用户级缺页处理技术。

这个技术还可能有许多其他的应用。比如在用户栈溢出时，可以动态分配一个新页面；如果需要对原本没有分配的 BSS 段读写时，可以动态分配一个全 0 页面。用户态的缺页交给用户程序自己处理，可以增加许多灵活性。

在这个 Exercise 中，我们需要实现 sys_env_set_pgfault_upcall 这个系统调用，从而能够使用户程序可以设置自己的缺页处理函数。

Exercise 9

在用户程序正常运行时，栈为用户栈，在当前为 USTACKTOP-PGSIZE 开始的一页。但是在执行用户缺页处理程序时，显然不能使用这个栈，因为可能破坏已有的栈结构，所以我们需要使用一个特殊的栈来处理缺页。在 JOS 中规定栈顶为 UXSTACKTOP。设置用户缺页处理程序时可以使用系统调用 sys_page_alloc 来为这个栈分配空间。

和内核的中断处理程序相似，用户态的处理程序也需要将现场和一些其他信息传入。这些工作需要在内核中进行。所以在这个 Exercise 中我们需要完成 kern/trap.c 的 page_fault_handler 函数，来将信息传递并且调用用户缺页处理程序。传递现场和信息使用的是和 Trapframe 相似的 UTrapframe。内核需要将这个结构构造好放置在用户异常栈中。当然在用户缺页处理程序中也可能发生缺页，这样就会造成迭代，这种情况下我们需要在旧栈指针基础上在空出 4 字节后放置新的 UTrapframe。这 4 字节是为了之后用户处理程序恢复状态使用。

之后的主要代码如下：

```
user_mem_assert(curenv, (void *)(utf), sizeof(struct UTrapframe), PTE_U | PTE_P);
//user_mem_assert(curenv, utf, PGSIZE, PTE_U | 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;

tf->tf_eip = (uint32_t)curenv->env_pgfault_upcall;
tf->tf_esp = (uint32_t)utf;
//page_remove(kern_pgdir, (void *)(UXSTACKTOP-PGSIZE));
env_run(curenv);
```

可见这个函数首先需要检查用户异常栈中防止 UTrapframe 的位置是否可以使用，随后才可以构建这个结构，否则则会出现致命的内核态下的缺页。

Exercise 10

非常有趣的是，我们并不能直接将用户所写的 C 程序传入内核并且设置为缺页处理程序，因为从内核回到用户程序时还需要传入一个 UTrapframe 的指针作为处理程序的参数，其次 C 程序退出时无法恢复之前程序的状态。因此我们需要一端汇编程序来实现这两个功能。

这个 Exercise 的任务就是实现 lib/pfentry.S。其中进入用户处理程序的方

法非常简单，即推入 UTrapframe 的指针到栈内就行了，但是从用户 C 程序返回后，我们还需要恢复现场，这就非常复杂。因为我们无法直接使用 jmp 指令，因为这需要我们将目标地址存储到一个数据寄存器中，但是由于我们需要所有的寄存器都恢复原样，因此无法再使用数据寄存器，所以我们需要操纵栈指针，最后调用 ret 来返回程序原来的地址。还有一个难点是恢复 EFLAGS 寄存器后，不能再使用任何可能造成该寄存器发生变化的指令，比如 add，sub 等。因此也需要注意。最后的实现方法如下：

```
subl $4, %esp
subl $4, 52(%esp)
movl 48(%esp), %eax
pushl %eax
popfl
movl 52(%esp), %eax
movl 44(%esp), %ebx
movl %ebx, (%eax)
movl 40(%esp), %eax
movl 36(%esp), %ecx
movl 32(%esp), %edx
movl 28(%esp), %ebx
movl 20(%esp), %ebp
movl 16(%esp), %esi
movl 12(%esp), %edi
movl 52(%esp), %esp
ret
```

可以看出，这里发生了许多复杂的栈操作，并且在 popfl(恢复 eflags)后再没有计算操作。

Exercise 11

这个 Exercise 的工作就是完成 lib/pgfault.c 中的 set_pgfault_handler 函数，来包装之前所写的系统调用即汇编程序。

这个函数的功能首先是为之前没有设置过用户缺页处理程序的进程设置栈，随后使用系统调用。

Exercise 12

以上部分实现后，我们就可以继续完成有 COW 特性的 fork 库函数了。在程序部分，需要完成的工作有：

1) 设置缺页处理函数（调用 set_pgfault_handler）；
2) 调用 sys_exofork 创建一个新进程；
3) 将 UTOP 一下的页面映射到相同的物理页面，如果是只读页直接映射即可，如果是 COW 页则子进程页面也为 COW 页，如果父进程可写的页则父进程和子进程都需要设置为 COW 页。用户异常栈必须为子进程分配一个新页面；
4) 父进程为子进程设置用户缺页处理程序；
5) 父进程将子进程状态设置为 RUNNABLE。

在复制映射时，我们可以使用 JOS 一个非常好的特性，即所有的页面都被映射到了 UPAGES 开始的 4M 空间内，所以能够直接使用**地址/2^12** 来得到页表项。

```
uintptr_t va = pn * PGSIZE;

volatile pde_t *pde = &uvpd[PDX(va)];
if(!(*pde & PTE_P))
    return 0;

volatile pte_t *pte = &uvpt[pn];
```

我们还需要完成用户缺页处理程序，其流程如下：

1）检查缺页是否为 COW 造成，如果不是则说明出错；

2）将目标页面复制到一个新页面，将新页面映射到原位置，并且解除原页面的映射

值得一提的是，完成拷贝内容的工作需要如下步骤：

1）将新页面分配在虚拟地址 PFTEMP；

2）将内容拷贝到 PFTEMP；

3）将 PFTEMP 映射到原位置；

4）解除 PFTEMP 映射。

这样做的原因是我们不能直接将内容拷贝到一个物理页，所以需要做这样一种转换。

第三部分

这一部分需要实现抢占式调度和进程间通信两个功能。在之前实现的非抢占调度中，如果一个程序不结束或主动让出 CPU 使用权，这个程序会一直执行。如果一个程序出现了死循环，则会造成整个 CPU 不能被其他程序使用。因此我们需要实现抢占式调度来解决这一问题。

要实现抢占式调度，需要有时钟中断的支持，因此我们首先需要实现这个功能。

Exercise 13

在这个 Exercise 中我们需要添加对外部中断的支持。我们目前有 IRQ0 到 IRQ15 一共 16 个外部中断。我们需要为这些中断分配的中断号为 IRQ_OFFSET+0 到 IRQ_OFFSET+15。这个位置的选定需要保证不和其他中断冲突。在 JOS 中我们选定 IRQ_OFFSET 为 32.

理论上，我们为了实现在用户态相应时钟中断，需要将 EFLAGS 的 IF 位打开。但是在 JOS 中我们假定在内核态不响应中断，在用户态一定相应中断。所以我们就可以使用存储和恢复 EFLAGS 的方法自动打开或关闭中断，从而不需要经常显式的对 IF 位进行操纵。

接下来的工作就是修改 kern/trapentry.S 和 kern/trap.c 来设置这 16 个中断的中断处理程序和中断描述符。具体内容不再赘述。值得一提的是由于这些中断都是硬件触发，所以可以把 DPL 位都设置成 0，防止程序恶意调用。

随后我们需要修改 kern/env.c 的 env_alloc()，我们需要将每个 env 的 EFLAGS 的 IF 位打开，来实现上述功能。

Exercise 14

下来我们就需要处理时钟中断了。之前在 i386_init()函数中调用了 lapic_init()和 pic_init()来初始化中断控制器的时钟，使其能够定期发送时钟中断。

接下来我们需要处理这些中断。其实方法十分简单，在 kern/trap.c 的 trap_dispatch()中将时钟中断引导到 sched_yield()中就可以实现重新调度。

```
if (tf->tf_trapno == IRQ_OFFSET + IRQ_TIMER) {
    cprintf("Timer interrupt\n");
    lapic_eoi();
    sched_yield();
    return;
}
```

函数 lapic_eoi()的功能是通知 lapic 已经接收到了中断。

Exercise 15

最后就是实现进程间通信(Inter-Process Communication IPC)的工作了。在 JOS 中，IPC 的功能基于两个函数：sys_ipc_recv 和 sys_ipc_try_send。其功能为发送一个 4 字节数或一个页，接受 4 字节数或将页映射到自己的空间。

sys_ipc_recv 是阻塞的，如果没有收到任何信息不会返回。，实际上收到信息后这个函数也不会返回，而是被内核唤醒后继续执行。

```
if((uint32_t)dstva < UTOP) {
    if((uint32_t)dstva % PGSIZE != 0)
        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();
return 0;
```

这个系统调用函数同时会将 env_ipc_recving 设置为 1，以通知所有发送者这个进程正在接受消息。随后将进程的状态设置为 NOT_RUNNABLE，进入休眠状态等待消息。

sys_ipc_try_send 则是非阻塞的调用，如果目标进程不在接受状态，则会返回-E_IPC_NOT_RECV。

在设置完目标进程中所有需要的信息后，这个调用会返回，在返回之前会进行这些额外工作：

```
e->env_tf.tf_regs.reg_eax = 0;
e->env_status = ENV_RUNNABLE;
return 0;
```

这样做的目的是让 sys_ipc_recv“返回”时返回值为 0，表示成功，并且将目标进程设置为可运行状态，以便被再次调度。

这两个系统调用有一个共同的问题：如何判断进程是否希望传递页。因为如果传递指针为 NULL，即 0，系统可以认为是希望把页面映射到 0 的虚拟地址。这明显是不可行的，所以需要另想办法实现这个功能。由于用户不能操纵 UTOP 以上的内容，因此我的设计中将 UTOP 即以上理解为不希望传送页。这个问题得以解决。

这个 Exercise 进一步需要实现 lib/ipc.c 中的 ipc_recv 和 ipc_send 两个库函数来包装之前的系统调用。ipc_recv 几乎没有太多改动，而 ipc_send 则是把非阻塞的系统调用使用 while 循环改进为忙等待型阻塞的调用。

Challenge 2

这个 Challenge 的主要目的是修改目前的调度算法。JOS 在之前使用了最简单的轮盘算法(Round-Robin Scheduling)。在这个 Challenge 中，我选择将其修改为优先级调度算法中的阶梯算法(Staircase Scheduling)。

这个算法曾经被 UNIX 采用，其主要思想如下：

1) 进程初始化时分配优先级；
2) 每个优先级维护一个队列；
3) 优先调度优先级高的进程；
4) 进程时间片用完后进入低一级队列。

这个算法需要修改的地方较多，一共修改了 inc/env.h, kern/sched.h, kern/env.c, kern/sched.c, kern/trap.c, kern/syscall.c, kern/init.c 共 7 个文件。下面是修改的详细介绍：

1. inc/env.h

这一部分主要是对 Env 结构进行的修改：

```
// Lab 4 Challenge Scheduling
int env_sched_priority;
struct Env* env_sched_next;
```

Env 中新增变量 env_sched_priority 意为当前优先级，指针 env_sched_next 为当前优先级列表中的下一个 Env.

还有修改：

```
// Lab 4 Challenge Scheduling
#define SCHED_MAX_USER_PRIORITY 4
```

表明当前优先级最高位 4（最小固定为 1），这个数可以直接修改，来增加或减少优先级。

2. kern/sched.h

这一部分主要增加了一些函数接口的声明：

```
// Lab 4 Challenge Scheduling
void sched_init(void);
void sched_first_in_que(struct Env *e);
void sched_change_priority(struct Env *e, int p);
void sched_recycle(struct Env *e);
```

函数 sched_init：初始化
函数 sched_first_in_que：创建进程时调用
函数 sched_change_priority：修改进程优先级调用
函数 sched_recycle：销毁进程时调用

3. kern/env.c

这一部分主要修改为两处，第一处在 env_alloc 函数中：

```
// Lab 4 Challenge Scheduling
e->env_sched_priority = SCHED_MAX_USER_PRIORITY;
sched_first_in_que(e);
```

第二处在 env_free 中：

```
// Lab 4 Challenge Scheduling
sched_recycle(e);
```

4. kern/sched.c

这一部分主要实现了调度算法和一些工具函数。具体内容比较多在这里就不再赘述。主要逻辑就是优先调度高优先级队列的进程。记录每个队列上次执行的进程，从下一个开始查找。每个优先级内使用轮盘调度。

5. kern/trap.c

这一部分的功能是给时间片用完的进程降级，修改在 trap_dispatch()函数中：

```
// Handle spurious interrupts
// The hardware sometimes raises these because of noise on the
// IRQ line or other reasons. We don't care.
if (tf->tf_trapno == IRQ_OFFSET + IRQ_SPURIOUS) {
    cprintf("Spurious interrupt on irq 7\n");
    print_trapframe(tf);

    // Lab 4 Challenge Scheduling
    if(curenv->env_sched_priority != 1) {
        sched_change_priority(curenv, curenv->env_sched_priority - 1);
    }

    return;
}
```

6. kern/syscall.c

这里的修改实在 sys_exofork 中将子进程优先级修改到与父进程相同：

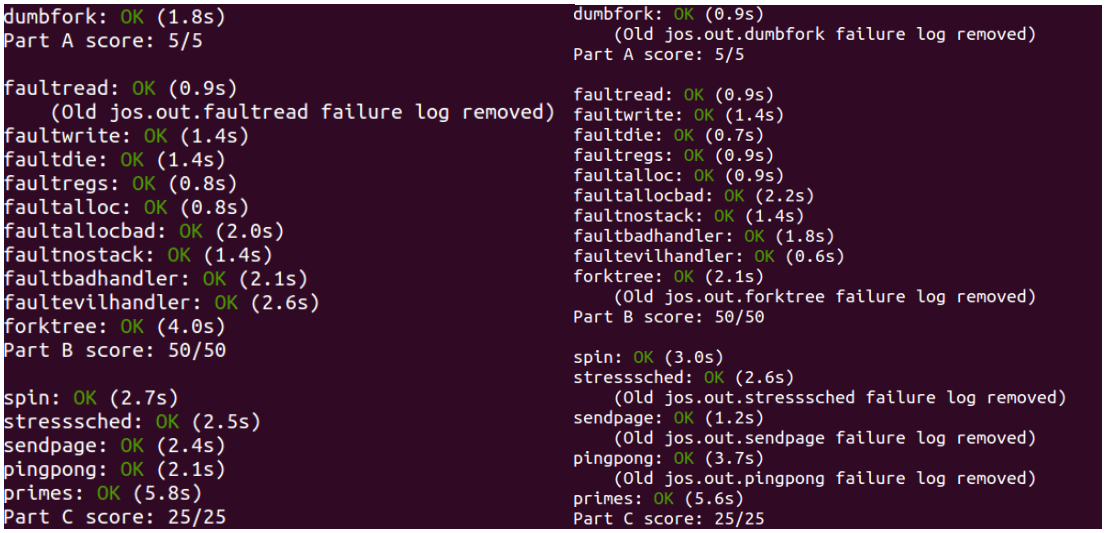
```
  // Lab 4 Challenge Scheduling
  sched_change_priority(e, curenv->env_sched_priority);
```

7. kern/init.c

这里的修改实在 i386_init 中调用 sched_init 进行初始化。

以下是算法评测。下图左侧为使用 Round-Robin 算法的结果，右图为使用阶梯算法的结果。可以发现这两种算法的差别并不算大。

经过分析，我认为出现这种情况的原因很有可能是因为时间片时间设置过长，导致大部分的程序都不会用完时间片就会回到内核或者程序已经结束。也可能是因为测试函数多会调用 sys_yield 主动出让控制权，导致大部分进程都一直处于最高优先级。这样两种算法其实是十分相似的，这就可以解释这个结果为何出现。



内容三：遇到的困难以及解决方法

困难 1

一开始我将 fork 函数中父进程为子进程设置缺页处理程序的部分写在了子进程，结果一直出错。仔细思考后发现这样做的确有问题，如下图：

```
set_pgfault_handler(pgfault);
envid_t envid = sys_exofork();
if(envid == 0) {
    cprintf("at child\n");
    thisenv = &envs[ENVX(sys_getenvid())];
    return 0;
}
```

很明显，调用 sys_exofork 后，返回值存储在 envid 中。这就会对内存造成读写，所以在这时立刻会发生缺页，而子进程根本来不及设置缺页处理程序。所以这个工作绝对不能放在子进程中。

困难 2

经过之前的困难，修改修改程序后，发现子进程缺页处理程序仍然不能正常工作。经过自己研究一下代码，我发现一些问题：

```
r = sys_page_alloc(thisenv->env_id, (void *)PFTEMP, PTE_P | PTE_U | PTE_W);
if(r < 0)
    panic("pgfault: %e", r);
memmove((void *)PFTEMP, ROUNDDOWN(addr, PGSIZE), PGSIZE);
r = sys_page_map(thisenv->env_id, (void *)PFTEMP, thisenv->env_id, ROUNDDOWN(addr, PGSIZE), PTE_P | PTE_U | PTE_W);
if(r < 0)
    panic("pgfault: %e", r);
r = sys_page_unmap(thisenv->env_id, (void *)PFTEMP);
if(r < 0)
    panic("pgfault: %e", r);
```

中断处理程序的这段代码多次使用了 thisenv 变量。但是正如之前所说的，子进程一旦开始运行会立刻发生缺页，根本没有时间修改 thisenv，并且修改 thisenv 还会发生缺页，因此必然不能在缺页处理程序修改。因此我使用了一下变量来代替 thisenv 的作用，错误得到解决。

```
int thisid = sys_getenvid();
```

内容四：收获及感想

这次 lab 做的十分艰难，除了这个 lab 本身的内容较多之外，还因为很多之前 lab 遗留下来的 bug。这些 bug 没有被之前的测试程序检测出来，但是却在这一次出了非常多的问题，这也为我造成了许多不便。

其实操作系统就是这样，非常难以调试并且错误很难重现，这就需要程序员认真编码并且仔细测试。

内容五：对课程的意见和建议

在 kern/sched.c 中，有如下代码片段：

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

我十分疑惑这个 idle 是怎么回事，因为在之前从来没有出现过和这个名字有关的内容。经过查找资料，发现这可能是老版本的遗留。原来在启动其他用户程序之前，需要启动 idle 程序。在调度时如果没有其他程序可以运行，则会执行 idle。在目前的 2014 版本中，这项功能被取消。如果没有进程可以调读，则调度函数会执行 halt，使 CPU 挂起。[ii]

内容六：参考文献

[i] https://en.wikipedia.org/wiki/Cooperative_multitasking
[ii] https://pdos.csail.mit.edu/6.828/2011/