MIT6.828 Lab4: Preemptive Multitasking

Zhuofan Zhang

April 2022

Contents

Chapter 1

Multiprocessor Support and Cooperative Multitasking

1.1 Multiprocessor Support

本次 Lab 的内容是对 JOS 进行补充以提供多处理器支持，并实现任务调度功能。

JOS 实现的多核支持属于对称多核支持 *(Symmertric Multiprocessing, SMP)*，即所有处理器对资源（内存、IO 总线等）的访问是平等的。SMP 的概念是在系统初始化完成后建立的，在 Boot 阶段，仍然需要选择一个 CPU 作为启动处理器 *(Bootstrap Processor, BSP)*，用来初始化资源并启动 OS，最后启动其他的 CPU（称为应用处理器 *(Application Processor, AP)*）。对 BSP 的选择是由 BIOS 完成的，在当前实验阶段，我们相当于完成了 BSP 的启动内容。

在 SMP 体系中，每个 CPU 都拥有一个称为 LAPIC(local Advanced-Programmable Interrupt Controller) 的可编程中断单元，用于在系统中传递中断信息，并提供 CPU 的唯一识别信息。因此，与多个 CPU 的通讯依赖于 LAPIC 单元。

在本次实验中我们使用了 LAPIC 的如下功能：

- 代码可以利用 APIC-id 确认自己运行于哪一个 cpu 上（见 cpunum() 的实现）
- 利用 BSP 来启动其他的 AP（见 lapic_startap() 的实现）
- 使用 LAPIC 中的时钟中断来实现抢断式调度（PartC，apic_init()）

CPU 对 LAPIC 单元的访问使用的是一段映射到 PAS 特定位置的区域：memory-mapped I/O (MMIO)。MMIO 是一段硬连接到部分 IO 设备寄存器的物理地址区域，常用于访问设备的寄存器。JOS 的 VAS 在 MMIOBASE 处留有 4MB 的空间用于映射这段物理地址区域。

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.

这一个 Exercise 要求我们实现 mmio_map_region()。我们查看该函数的注释可知，它就是用来实现上文提到的 MMIO 在 VAS 中映射的函数。JOS 的 VAS 中预留了 [MMIOBASE, MMIOLIM) 可供使用。根据提示使用 boot_map_region() 并补上禁用缓存的 PWT-flag，实现函数如下：

```
void *
mmio_map_region(physaddr_t pa, size_t size)
{
    static uintptr_t base = MMIOBASE;

    size = ROUNDUP(size, PGSIZE);
    if(base + size > MMIOLIM)
        panic("mmio_map_region: MMIOLIM overflow.\n");
    boot_map_region(kern_pgdir, base, size, pa, PTE_W | PTE_PCD | PTE_PWT);
    base += size;
    return (void *)(base - size);

}
```

1.2 Application Processor Bootstrap

在启动 APs 前，先启动的 BSP 必须能够获取 APs 的信息，如 CPU 数量、APIC IDs 等。这些信息被保留在 BIOS 中的 MP config table 中，kern/mpconfig.c/mp_init() 将其读出。

kern/init.c/boot_aps() 是整个流程的核心函数，它将 AP 的入口程序加载到内存，并发送启动信号。AP 的入口程序被加载到任意可用的低位 640k 地址（JOS 将其加载到 MPENTRY_PADDR），它与 BSP 的入口程序有一定的区别。

boot_aps() 函数通过给每个 AP 的 LAPIC 发送 STARTUP 信号的方式唤醒它们，同时设置它们的 entry 位置，即为 AP 准备的入口程序（位于 MPENTRY_PADDR），执行完入口程序后 AP 将运行 mp_main()；与此同时，BSP 唤醒每一个 AP 时等待它的 CPU 状态被设置为 CPU_STARTED——即该 AP 成功启动后，再开始唤醒下一个 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).

这一个 Exercise 的编码任务比较简单：修改我们在 Lab2 实现的 page_init 函数，将现在存放有 AP 初始代码的那个物理页从空闲列表中拿出去：

```
void
page_init(void)
{
    // LAB 4:
    // Change your code to mark the physical page at MPENTRY_PADDR
    // as in use

    ...

    // 2) Base-memory
    size_t i;
    for (i = 1; i < npages_basemem; i++) {
        // Add for Lab4
        if(i * PGSIZE == MPENTRY_PADDR)
        {
            pages[i].pp_ref = 1;
            continue;
        }

        pages[i].pp_ref = 0;
        pages[i].pp_link = page_free_list;
        page_free_list = &pages[i];
    }

    ...

}
```

代码之外的任务是要求我们阅读 boot_aps() 及 mp_main() 源码以及 APs 的 entry 汇编代码，捋清执行流。boot_aps() 做的事情就是将 mpentry.S 搬运到前文提到的 MPENTRY_PADDR 物理页上，再循环唤醒 APs（使用 lapic_startaps()）；APs 被唤醒后从 mpentry.S 开始执行，再跳转到 mp_main() 上。

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.

AP 的入口程序与 BSP 的入口程序差异主要表现在：

- 使用地址时需要使用 MPBOOTPHYS-macro 进行一步转换；
- 无需进行 A20 地址线的使能操作；
- 直接在入口处打开了分页功能，并使用已经被 BSP 设置好的内核页表目录

使用 MPBOOTPHYS 是因为 mpentry.S 被链接到了内核的高地址处（above KERNBASE）, 需要进行转换。

1.3 Per-CPU State and Initialization

每个 CPU 具有自己独立的状态和信息，JOS 将各个 CPU 独立的信息定义在 kern/cpu.h 中。其中比较关键的内容有：

- **CPU 内核栈**：支持多 CPU 同时陷入内核；
- **TSS 及 TSS 描述符**：每个 CPU 拥有独立的 TSS 指明内核栈位置；
- **当前运行 Env 指针**：用于记录每个 CPU 的当前运行进程；
- **系统寄存器**：每个 CPU 需独立加载页表、GDT、LDT 等内容

本节实验的内容是对这类 per-CPU 状态进行初始化。

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 Exercise 要求我们对各个内核的内核栈进行映射。BSP 在初始化 VAS 时，完成对各个 cpu 的内核栈的映射（JOS 默认支持 8 核，且不管每次实际启动的 CPU 数量，为 8 个内核栈都分配空间）。根据提示，我们知道分配给各个 cpu 内核栈的物理空间由 percpu_kstacks 数组描述（定义于 mpconfig.c 中），我们需要将 VAS 中的各个内核栈位置映射到这个数组描述的物理地址

处。同时也注意到，JOS 的 VAS 中每个内核栈也预留了一个 Guard-page，我们需要跨过这段区域进行映射。

实现内核栈映射的方法与之前 Lab2 中对 BSP 的内核栈映射方式相同，同时注意到：此处进行映射时也替换了原本 BSP 中的内核栈位置（从 bootstack 切换到了 &percpu_kstacks[0]）。

```
static void
mem_init_mp(void)
{
    ...
    // LAB 4: Your code here:
    for(int i = 0; i < NCPU; ++i)
    {
        // Map the kstk_i
        boot_map_region(
            kern_pgdir,
            KSTACKTOP - i*(KSTKSIZE + KSTKGAP) - KSTKSIZE,
            KSTKSIZE,
            PADDR(percpu_kstacks[i]),
            PTE_W
        );
    }
}
```

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 Exercise 要求我们修改 trap 初始化的部分，主要就是修改 TSS 及其 descriptor：TSS 内主要要修改的是对应的内核栈（每个 CPU 使用各自独立的内核栈），同时修改 GDT 表项（如前文所述，每个 CPU 拥有自己独立的 TSS-descriptor）。根据实验提示我们可以完成下列代码：

```
void
trap_init_percpu(void)
{
    ...
    // LAB 4: Your code here:

    // Setup a TSS so that we get the right stack
    // when we trap to the kernel.
    // ts.ts_esp0 = KSTACKTOP;
    // ts.ts_ss0 = GD_KD;
    // ts.ts_iomb = sizeof(struct Taskstate);
    uint8_t nowCpuId = thiscpu->cpu_id;
    struct Taskstate* nowTs = &(thiscpu->cpu_ts);

    nowTs->ts_esp0 = KSTACKTOP - nowCpuId*(KSTKSIZE + KSTKGAP);
    nowTs->ts_ss0 = GD_KD;
    nowTs->ts_iomb = sizeof(struct Taskstate);

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

    // Load the TSS selector (like other segment selectors, the
    // bottom three bits are special; we leave them 0)
    // ltr(GD_TSS0);
    ltr(GD_TSS0 + (nowCpuId << 3));

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

Locking

目前我们的 mp_main() 仍处于空转状态，但在有效使用 APs 来并行运算代码前，我们必须解决并发所带来的竞争问题。JOS 解决并发访问冲突的方法是使用 big-kernel-lock（也即早期 Linux 使用的方法），任意时刻仅能容纳 1 个 user-mode 进程陷入内核。JOS 已经提前为我们实现了两个 API：lock_kernel() 和 unlock_kernel()，其内部实现是一个全局自旋锁。

根据 Lab 页面的提示，我们需要在以下部分上锁/释放锁：

- **i386_init()**： BSP 在启动 APs 前需先持有锁（该锁在后续执行流的 sched_yield() 中释放），防止与 APs 同时调度进程时出现 race-condition；
- **mp_main()**： APs 与 BSP 同理，为防止进程调度时产生 race-condition，调度前需先持锁；
- **trap()**：如前文所述，当 user-mode 进程陷入内核时需持锁，保证同一时刻仅一个进程陷入内核中；
- **env_run()**：env_run 即从内核态返回 user-mode，此时释放内核锁，让其他等待陷入的进程可以进入内核

> **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上述内容，我们直接在对应函数中相应区域填入代码即可：

```
/**** i386_init() ****/
...
lock_kernel();
boot_aps();
...
/**** mp_main() ****/
...
    lock_kernel(); # Add in Lab4 Exercise 5
    sched_yield();
...
/**** trap() ****/
...
if ((tf->tf_cs & 3) == 3) {
    ...
    lock_kernel();
    assert(curenv);
...
/**** env_run() ****/
...
lcr3(PADDR(curenv->env_pgdir));
unlock_kernel(); # Add in Lab4 Exercise 5
env_pop_tf(&(curenv->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.

在进入内核前，实际上也有对内核栈的操作（trap 将寄存器信息压入内核栈中），此时栈处于无锁状态，有可能发生 race-condition。

1.4 Round-Robin Scheduling

本节要求我们实现最简单的 Round robin 进程调度。sched_yield() 函数负责从 envs 中挑选返回 user-mode 后运行的新进程。调度算法即 Round-robin：在循环队列中寻找当前进程（curenv）位置后方可执行的（ENV_RUNNABLE）第一个进程。

Exercise 6

Implement round-robin scheduling in sched_yield() as described above. Don't forget to modify syscall() to dispatch sys_yield().

Make sure to invoke sched_yield() in mp_main.

Modify kern/init.c to create three (or more!) environments that all run the program user/yield.c.

Run make qemu. You should see the environments switch back and forth between each other five times before terminating, like below.

Test also with several CPUS: *make qemu CPUS=2*.

```
...
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.

我们首先定位到 sched_yield() 函数。根据注释的提示，我们可以将函数实现如下（需要考虑三种调度的情况）：

```
void
sched_yield(void)
{
    struct Env *idle = NULL;
    ...
    // situation 1: no running env
    if(!curenv)
    {
        for(int i = 0; i < NENV; ++i)
        {
            if(envs[i].env_status == ENV_RUNNABLE)
            {
                idle = &envs[i];
                break;
            }
        }
    }
    // situation 2: search after not-null curenv
    else
    {
        int currentIdx = curenv - envs;
        for(int i = 1; i < NENV; ++i)
        {
            int idx = (currentIdx + i) % NENV;
            if(envs[idx].env_status == ENV_RUNNABLE)
            {
                idle = &envs[idx];
                break;
            }
        }
    }
    // situation 3: only the curenv can run
    if(!idle && curenv && curenv->env_status == ENV_RUNNING)
        idle = curenv;

    // find next-run success
    if(idle)
        env_run(idle);  // the env_run will unlock_kernel

    // sched_halt never returns
    sched_halt();
}
```

之后，我们实现提供给用户态的 sys_yield() 接口（此处从略）。为了测试结果，我们根据提示在 i386_init() 中添加创建用户进程的代码：

```
/**** i386_init() ****/
...
#else
   // Touch all you want.
   // ENV_CREATE(user_primes, ENV_TYPE_USER);
#endif // TEST*

   for(int i = 0; i < 3; ++i)
   {
      ENV_CREATE(user_yield, ENV_TYPE_USER);
   }

   // Schedule and run the first user environment!
   sched_yield();
```

在 *make qemu* 命令启用单个 CPU 运行，得到运行结果如下：

```
VNC server running on `127.0.0.1:5900'
Physical memory: 131072KB available, base = 640KB, extended = 130432KB
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2
[00000000] new env 00001000
[00000000] new env 00001001
[00000000] new env 00001002
Hello, I am environment 00001000.
Hello, I am environment 00001001.
Hello, I am environment 00001002.
Back in environment 00001000, iteration 0.
Back in environment 00001001, iteration 0.
Back in environment 00001002, iteration 0.
Back in environment 00001000, iteration 1.
Back in environment 00001001, iteration 1.
Back in environment 00001002, iteration 1.
Back in environment 00001000, iteration 2.
Back in environment 00001001, iteration 2.
Back in environment 00001002, iteration 2.
Back in environment 00001000, iteration 3.
Back in environment 00001001, iteration 3.
Back in environment 00001002, iteration 3.
Back in environment 00001000, iteration 4.
All done in environment 00001000.
[00001000] exiting gracefully
[00001000] free env 00001000
Back in environment 00001001, iteration 4.
All done in environment 00001001.
[00001001] exiting gracefully
```

若以 *make qemu CPUS=2* 命令启用双核系统，得到的运行结果如下：

```
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!
```

Question

3. 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?

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?

第一个问题的答案很简单：所有的进程都映射了完整的 VAS 的内核部分，而 envs 的映射也位于内核部分中，故所有进程对于 envs 数组任一进程的地址翻译都是相同的。

第二个问题：进程的现场保存发生在 _alltrap 过程中，现场恢复发生在 env_pop_tf() 时。

1.5 System Calls for Environment Creation

目前我们的系统已经可以完成用户态进程的调度了，但我们还未提供真正的用户态使用进程的接口。这一节我们将实现其中最基础的一部分。提供给用户的进程接口同样采用系统调用的实现方式。

> **Exercise 7**
>
> Implement the system calls described above in kern/syscall.c and make sure syscall() calls them. 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_INVAL in that case. Test your JOS kernel with user/dumbfork and make sure it works before proceeding.

sys_exofork

sys_exofork() 函数可以用于创建新的子进程，并且设置子进程的内存映射、寄存器状态与父进程完全一致（按照状态机复制理解）。对于父子进程而言，函数返回不同的值（父进程得到子进程的 env_id，子进程返回 0）：

（注：返回值的设置应回忆 syscall 的调用流与 env 状态保存逻辑；对于子进程而言，当它被调度时看起来“就像刚从系统调用返回”一样）

```
static envid_t
sys_exofork(void)
{
    // Create the new environment with env_alloc(), from kern/env.c.
    // It should be left as env_alloc created it, except that
    // status is set to ENV_NOT_RUNNABLE, and the register set is copied
    // from the current environment -- but tweaked so sys_exofork
    // will appear to return 0.

    // LAB 4: Your code here.
    struct Env *new_env;
    int r = env_alloc(&new_env, curenv->env_id);
    if(r  < 0)
        return r;

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

    return new_env->env_id;
}
```

sys_page_alloc

sys_page_alloc() 系统调用为用户提供申请物理页的接口，并映射到指定的虚拟地址位置。

```
static int
sys_page_alloc(envid_t envid, void *va, int perm)
{
    ...
    // LAB 4: Your code here.
    struct Env *e;
    int r = envid2env(envid, &e, 1);

    // bad-env
    if(r < 0)
        return r;

    // va is not page-aligned or va >= UTOP
    if((uintptr_t)va >= UTOP || (uintptr_t)va != ROUNDDOWN((uintptr_t)va, PGSIZE))
        return -E_INVAL;

    // perm inappropriate
    if(
        !(perm & PTE_U) ||
        !(perm & PTE_P) ||
        ((perm | PTE_SYSCALL) != PTE_SYSCALL)
        )
        return -E_INVAL;

    struct PageInfo *pp = page_alloc(0);
    if(!pp)
        return -E_NO_MEM;
    r = page_insert(e->env_pgdir, pp, va, perm);
    if(r < 0)
    {
        page_free(pp);
        return r;
    }

    return 0;

}
```

sys_page_map

sys_page_map() 系统调用用于将源进程 srcva 处映射的物理页映射到目的进程 dstva 处。需要注意的是若源进程中该物理页为 read-only 权限，在目的进程中也应保持其只读性。

```
static int
sys_page_map(envid_t srcenvid, void *srcva,
             envid_t dstenvid, void *dstva, int perm)
{
    // LAB 4: Your code here.
    struct Env *srcE, *dstE;
    // bad-env
    if(envid2env(srcenvid, &srcE, 1) < 0 ||
       envid2env(dstenvid, &dstE, 1) < 0)
    { return -E_BAD_ENV; }
    // va-error
    if((uintptr_t)srcva >= UTOP ||
       (uintptr_t)srcva != ROUNDDOWN((uintptr_t)srcva, PGSIZE) ||
       (uintptr_t)dstva >= UTOP || (uintptr_t)dstva != ROUNDDOWN((uintptr_t)dstva, PGSIZE)
    )
    { return -E_INVAL; }
    // perm inappropriate
    if(
        !(perm & PTE_U) || !(perm & PTE_P) ||
        ((perm | PTE_SYSCALL) != PTE_SYSCALL)
    )
    { return -E_INVAL; }

    pte_t *srcPTE;
    struct PageInfo *srcpp = page_lookup(srcE->env_pgdir, srcva, &srcPTE);
    struct PageInfo *dstpp = page_alloc(0);

    // write on read-only
    if((perm & PTE_W) && !(*srcPTE & PTE_W))
        return -E_INVAL;

    // srcva not mapped
    if(!srcpp)
        return -E_INVAL;

    // no memory
    if(page_insert(dstE->env_pgdir, srcpp, dstva, perm) < 0)
        return -E_NO_MEM;
    return 0;
}
```

sys_page_unmap

sys_page_unmap() 系统调用是解除 va 处的物理页映射，相当于对内核中 page_remove() 接口的用户态封装。

```
static int
sys_page_unmap(envid_t envid, void *va)
{
    // Hint: This function is a wrapper around page_remove().

    // LAB 4: Your code here.
    struct Env *e;
    if(envid2env(envid, &e, 1) < 0)
        return -E_BAD_ENV;
    if((uintptr_t)va >= UTOP || (uintptr_t)va != ROUNDDOWN((uintptr_t)va, PGSIZE))
        return -E_INVAL;

    page_remove(e->env_pgdir, va);
    return 0;

}
```

sys_env_set_status

sys_env_set_status() 用于设置目标进程的运行状态（可运行/不可运行）：

```
static int
sys_env_set_status(envid_t envid, int status)
{
    // LAB 4: Your code here.
    if(status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE)
        return -E_INVAL;
    struct Env *e;
    int r = envid2env(envid, &e, 1);
    if(r < 0)
        // error happened
        return r;
    e->env_status = status;
    // when go back to trap() who called this function,
    // the sched_yield will be called
    return 0;

}
```

完成上述函数实现后，修改 syscall() 函数的 dispatch 部分：

```
int32_t
syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t a4, uint32_t
    a5)
{
    // Call the function corresponding to the 'syscallno' parameter.
    // Return any appropriate return value.
    // LAB 3: Your code here.

    // panic("syscall not implemented");

    switch (syscallno)
    {
        case SYS_cputs:
            sys_cputs((char *)a1, a2);
            return 0;
        case SYS_cgetc:
            return sys_cgetc();
        case SYS_getenvid:
            return sys_getenvid();
        case SYS_env_destroy:
            return sys_env_destroy(a1);
        case SYS_yield:
            sys_yield();
            return 0;
        case SYS_exofork:
            return sys_exofork();
        case SYS_env_set_status:
            return sys_env_set_status(a1, a2);
        case SYS_page_alloc:
            return sys_page_alloc(a1, (void *)a2, a3);
        case SYS_page_map:
            return sys_page_map(a1, (void *)a2, a3, (void *)a4, a5);
        case SYS_page_unmap:
            return sys_page_unmap(a1, (void *)a2);
        default:
            return -E_INVAL;
    }
}
```

Chapter 2

Copy-on-Write Fork

本节实现 fork() 等用户态进程创建的相关调用。JOS 在实现 fork() 调用的时候采用了与 xv6 不同的方法：对创建的子进程的地址空间复制采用 copy-on-write 的方法，即创建时子进程直接复制使用父进程的地址映射（而不复制内存空间内容本身），等到父子进程任一一方执行写操作引发 page fault，再进行内存空间的复制。这样做的好处是在很多 workload 中，fork 得到的子进程很快就会进行 excute 操作，而前面耗时复制的父进程地址空间内容并不起到什么作用；采用 copy-on-write 可以有效解决这里的性能问题。

2.1 User-level page fault handling

JOS 将 page fault 的处理逻辑交给用户态程序定义。用户态进程定义的 handler 可以使用 JOS 系统调用的方法来处理可能出现在该进程上的 page fault。

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.

这个练习要求我们实现设置 pgfault_upcall 的函数。pgfault_upcall 是对 page fault handler 的封装（详见后文 Exercise 10），在执行真正的 handler 前完成必要的准备工作与善后工作。JOS 为 env 结构体添加了存放 upcall 函数的指针，我们只需将用户态传递来的指针设置即可。

（注：完成该系统调用同样需要修改 syscall 的 dispatch 部分，此处省略）

```
static int
sys_env_set_pgfault_upcall(envid_t envid, void *func)
{
    // LAB 4: Your code here.
    struct Env *e;
    if(envid2env(envid, &e, 1) < 0)
        return -E_BAD_ENV;
    e->env_pgfault_upcall = func;

    return 0;
}
```

Normal and Exception Stacks in User Environments

在调用 page fault handler 时，JOS 还涉及一次栈的切换。当用户态程序正常运行时，它会使用用户栈（即 [USTACKTOP-PGSIZE, STACKTOP) 区域）。当 page fault 发生时，JOS 将栈切换到地址空间中的用户异常栈（[UXSTACKTOP-PGSIZE, UXSTACKTOP)）。Hanlder 在这个栈上处理 page fault，通过 JOS 的系统调用执行为缺页提供新的物理页等方式解决 page fault。最后 Handler 退出，用户态程序回到原有的用户栈处。

Invoking the User Page Fault Handler

与陷入 trap 时我们在内核栈上维护了 Trapframe 结构同理，当 page fault 发生时我们也在上述用户异常栈上维护一个 UTrapframe 结构保存发生 page fault 时的用户态程序现场。当 handler 结束调用后，我们可以利用保存的现场恢复原有用户态进程的运行。这一过程是由接管所有 page fault trap 处理的 page_fault_handler() 函数完成的：我们在 Lab3 中实现了对内核态 page fault 的处理（crash），本节我们需要按照上述想法实现用户态的处理。

Setting the Page Fault Handler

Exercise 9

Implement the code in page_fault_handler in kern/trap.c required to dispatch page faults to the user-mode handler. Be sure to take appropriate precautions when writing into the exception stack. (What happens if the user environment runs out of space on the exception stack?)

根据实验要求，我们完成代码如下：

```
void
page_fault_handler(struct Trapframe *tf)
{
    ...
    // LAB 4: Your code here.
    if(curenv->env_pgfault_upcall)
   {
      struct UTrapframe *utf;
      // stacked page fault
      if(UXSTACKTOP - PGSIZE <= tf->tf_esp && tf->tf_esp < UXSTACKTOP)
         // remain a sratch space at the top
         utf = (struct UTrapframe*)(tf->tf_esp - sizeof(struct UTrapframe) - 4);
      // first page fault
      else
         utf = (struct UTrapframe*)(UXSTACKTOP - sizeof(struct UTrapframe));

      // (1) didn't allocate a page for exception stack
      // (2) exception stack overflows
      // (3) can't write it
      user_mem_assert(curenv, (void *)utf, sizeof(struct UTrapframe), PTE_W);

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

      tf->tf_esp = (uintptr_t)utf;
      tf->tf_eip = (uintptr_t)(curenv->env_pgfault_upcall);
      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);
}
```

关于这部分代码，有几点需要说明：

首先是判断 upcall 函数是否存在（用户态程序是否正确设置 handler），如不存在说明用户态程序没有准备处理 page fault 的程序，因此我们直接回收进程（根据实验提示，我们在进程回收前打印信息方便调试）。

第二，若 upcall 函数存在，则第一个要考虑的问题是构造 UTrapframe 的位置。我们检查 Trapframe 为我们提供的发生 page fault 时 ESP 的值：如果它已经位于用户异常栈中，则我们在当前栈中继续生长（根据实验提示，在栈上添加一个字长的空白区域）；否则，我们需要切换到用户异常栈栈顶处。需要注意的是，page_fault_handler 并不负责用户异常栈内存的分配（从后续实验可知，这一步交给了用户自定义的 handler 完成）。

在确定 UTrapframe 放置位置后，我们填充该结构，并将 Trapframe 的 EIP 切换到 upcall 函数上再调度，返回用户态后，用户进程开始执行 upcall 函数。

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.

在上一个 Exercise 中，我们在内核态中设置了用户异常栈、将执行流位置设置到 upcall 函数后返回到发生了 page fault 的用户态进程。因此，整个 upcall 函数都是运行在用户态下的。如前文所述，upcall 函数是对用户自定义 page fault handler 的一个封装：它的主要任务是在调用 handler 以后恢复进程现场，使进程重新恢复到触发 page fault 时的状态。根据实验提示，我们对它的实现如下：

```
.text
.globl _pgfault_upcall
_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 argument

    // LAB 4: Your code here.
    movl 0x28(%esp), %eax  // take out the trap-time eip
    subl $4, 0x30(%esp)     // page fault address
    movl 0x30(%esp), %edx  // take out the trap-time esp
    movl %eax, (%edx)       // push EIP into USER-STACK
    addl $8, %esp
    popal
    addl $4, %esp
    popfl
    popl %esp
    ret
```

我们同样来理解一下 _pgfault_upcall 的实现。第一部分非我们实现的内容是对用户定义的 handler 的调用。我们在这个 Exercise 中实现的内容是现场的恢复。根据实验的提示，直接使用 jmp 指令进行跳转是不可行的：我们需要恢复所有的寄存器现场，如果使用 jmp 的话我们总是会需要再一个额外的、不需要恢复的寄存器来保存跳转地址，这会导致恢复的现场与实际现场不一致。根据实验的提示，我们采用 ret 指令来回到原来的状态。在 Lab3 时我们已经接触过，ret 指令会将在栈上的 EIP 弹出并载入，从而使我们能够回到原初的状态。

我们的代码首先将用户栈和发生 page fault 时的 EIP 取出（均存在 UTrapframe 中），并将 EIP 放入用户栈中（注意需先将 EIP 地址减去 4，才是发生 page fault 的正确地址）；之后我们跳过栈顶两个与寄存器现场恢复无关的值后，popal/popfl 恢复现场。最后我们用 popl 将存放在异常栈栈顶的用户栈地址弹出，进程即成功切换到用户栈。此时用户栈栈顶存放着 page fault 发生地址，我们只需 ret 即可回到现场。

Exercise 11

Finish set_pgfault_handler() in lib/pgfault.c.

最后我们要实现注册 handler 的用户 library 函数。该函数本身是用户态函数，因此所有与 JOS 的交互通过系统调用来执行。在函数第一次被调用时，我们使用系统调用为用户异常栈分配一个物理页并完成映射，同时将 upcall 函数设置为我们在上一个 Exercise 中实现的 _pgfault_upcall。upcall 函数会根据函数指针名找到 _pgfault_handler 并执行。

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

    if (_pgfault_handler == 0)
    {
        // First time through!
        // LAB 4: Your code here.
        envid_t envid = sys_getenvid();
        if(sys_page_alloc(envid, (void *)(UXSTACKTOP-PGSIZE), PTE_W | PTE_U | PTE_P) < 0)
            panic("sys_page_alloc failed.");
        if(sys_env_set_pgfault_upcall(envid, _pgfault_upcall) < 0)
            panic("sys_env_set_pgfault_upcall failed.");
    }

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

2.2 Implementing Copy-on-Write Fork

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.

这一节中我们需要实现 duppage，pgfault 以及完整的 fork 函数，完成这一个 Exercise 后我们就可以在用户态创建新进程了。我们按照 duppage，fork 到 pgfault 的顺序实现上述函数。

duppage

duppage() 函数用于在指定的进程上完成对某个物理页的映射复制。其使用场景是父进程 fork 时调用该函数，将目标页映射复制到子进程 VAS 中，同时若该页原本为可写页，将父子进程的该映射都设置为 copy-on-write 模式。

```
static int
duppage(envid_t envid, unsigned pn)
{
    // LAB 4: Your code here.
    // map for child
    if((uvpt[pn] & PTE_W) || (uvpt[pn] & PTE_COW))
    {
        int r = sys_page_map(0, (void *)(pn*PGSIZE),envid, (void *)(pn*PGSIZE),
                             PTE_COW | PTE_U | PTE_P);
        if(r < 0)
            return -1;

        // remap for thisenv
        r = sys_page_map(0, (void *)(pn*PGSIZE), 0, (void *)(pn*PGSIZE),
                         PTE_COW | PTE_U | PTE_P);
        if(r < 0)
            return -1;
    }
    else
    {
        int r = sys_page_map(0, (void *)(pn*PGSIZE), envid, (void *)(pn*PGSIZE),
                             PTE_U | PTE_P);
        if(r < 0)
            return -1;
    }
    return 0;
}
```

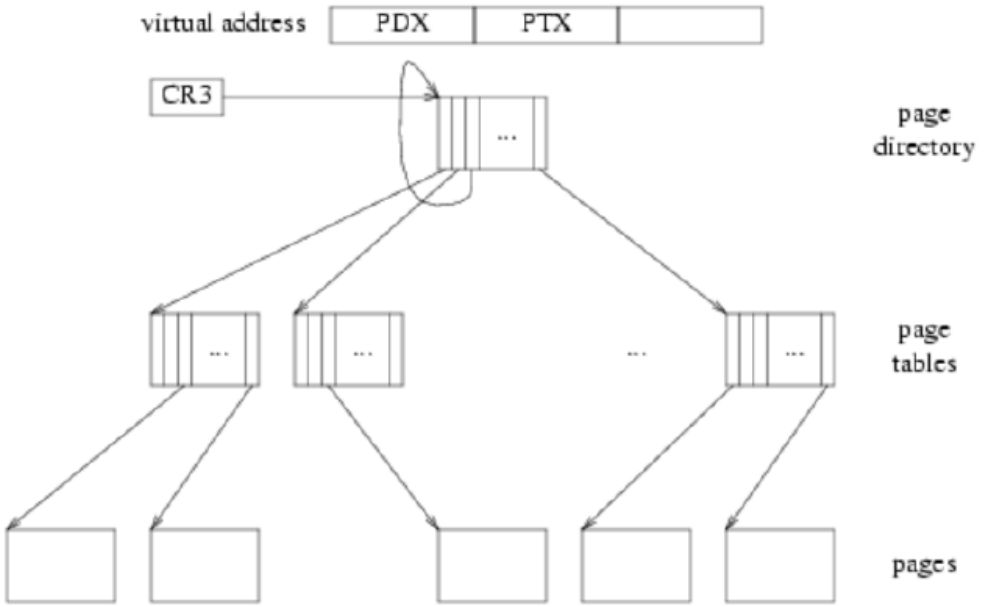
函数实现上有一个要注意的点：检查需要被 dup 的页的用户态权限时，用到了在 env_alloc 阶段映射在 VAS 中的 [UVPT, UVPT+PTSIZE) 段：这个段映射了用户进程的页目录（见 env_setup_vm() 的实现）。JOS 在 entry.S 中提供了两个变量：uvpt 和 uvpd。

```
.globl uvpt
.set uvpt, UVPT
.globl uvpd
.set uvpd, (UVPT+(UVPT>>12)*4)
```

这里 JOS 使用了一个相当 trick 的方法：将页目录作为页表进行查询，从而得到页目录/页表自身。将 UVPT 段映射到页目录自身的物理地址后，x86 对 PDX 与 UVPT 相同的段进行地址翻译时，根据对应的 PDE 找到自身，在第二步翻译时实际上是根据 PTX 对页目录进行搜索，根据 PTX 偏移量得到的正是对应页表的 PTE。JOS 定义了一个虚拟地址为 UVPT 的数组 uvpt，作为页表访问入口。

如果需要访问页目录呢？在访问页表时首先需要确定页表是否存在，这是记录在 PDE 中的。实际访问逻辑与上述访问页表逻辑是相同的，只需将地址的 PDX 与 PTX 都设置为与 UVPT 的 PDX 段相同即可。JOS 定义了数组 uvpd，虚拟地址为 (UVPT+(UVPT » PGSHIFT)*4)。

（注：页目录/页表与页的寻址方式不同：页目录与页表按照字（JOS 中即 4 字节寻址），而物理页按照字节寻址。在定义 uvpt 和 uvpd 变量时将它们定义为了 pte_t 和 pde_t 数组，因此索引时得到实际地址应为 uvpt/uvpd 基址加上 4* 下标偏移量。对于 uvpt，下标偏移量计算使用 PGNUM(addr)，而对于 uvpd 则为 PDX(addr)。）



fork

```
envid_t
fork(void)
{
    // LAB 4: Your code here.
    set_pgfault_handler(pgfault);
    envid_t newChildId = sys_exofork();
    if(newChildId < 0)
        return -1;

    // for child-process
    if(newChildId == 0)
    {
        thisenv = envs + ENVX(sys_getenvid());
        return 0;
    }

    // allocate a page for child-exception-stack
    if(sys_page_alloc(newChildId, (void *)(UXSTACKTOP-PGSIZE), PTE_W | PTE_U | PTE_P) < 0)
        return -1;

    // set child-pgfault-upcall
    sys_env_set_pgfault_upcall(newChildId, (void *)_pgfault_upcall);

    // copy the map into child
    uintptr_t addr;
    for(addr = 0; addr < USTACKTOP; addr += PGSIZE)
        if((uvpd[PDX(addr)] & PTE_P) && (uvpt[PGNUM(addr)] & PTE_P))
            duppage(newChildId, PGNUM(addr));

    // set child to be runnable
    if(sys_env_set_status(newChildId, ENV_RUNNABLE) < 0)
        return -1;

    return newChildId;
}
```

fork 实现时依次完成几件事：(1) 创建子进程（此时子进程仍未 NOT_RUNNABLE）；(2) 从子进程返回时修改 thisenv 后直接返回（这一步会在父进程昨晚所有处理、将子进程设置为 RUNNABLE 后才可能被调度执行）；(3) 为子进程分配独立的用户异常栈（不使用 cow）；(4) 对所有 VAS 中的用户态内存进行 cow-dup；(5) 将子进程设置为可被调度运行的状态后返回。

pgfault

```
static void
pgfault(struct UTrapframe *utf)
{
    void *addr = (void *) utf->utf_fault_va;
    uint32_t err = utf->utf_err;
    int r;

    // Check that the faulting access was (1) a write, and (2) to a
    // copy-on-write page.  If not, panic.
    // Hint:
    //   Use the read-only page table mappings at uvpt
    //   (see <inc/memlayout.h>).

    // LAB 4: Your code here.
    if(!((uvpt[PGNUM(addr)] & PTE_COW) && (err & FEC_WR))
    )
    { panic("pgfault panic!"); }

    // Allocate a new page, map it at a temporary location (PFTEMP),
    // copy the data from the old page to the new page, then move the new
    // page to the old page's address.
    // Hint:
    //   You should make three system calls.

    // LAB 4: Your code here.
    envid_t envId = sys_getenvid(); // Don't use curenv->env_id!
    if(sys_page_alloc(envId, (void *)(PFTEMP), PTE_P | PTE_U | PTE_W) < 0)
        panic("sys_page_alloc failed in pgfault().");

    // memory-move
    memmove((void *)PFTEMP, (void *)(ROUNDDOWN(addr, PGSIZE)), PGSIZE);

    if(sys_page_map(
        envId, (void *)(PFTEMP),
        envId, (void *)(ROUNDDOWN(addr, PGSIZE)),
        PTE_P | PTE_W | PTE_U) < 0
    )
    { panic("sys_page_map failed in pgfault().");; }

    if(sys_page_unmap(envId, (void *)(PFTEMP)) < 0)
        panic("sys_page_unmap failed in pgfault().");
}
```

该函数即为默认的 page fault 处理函数。当发生的错误是对 cow 页进行写操作时，创建新的物理页，对原始页进行复制操作。这里借用了 VAS 中的临时区域 PFTEMP 作为临时位置，协助进行物理页内容的移动。

Chapter 3

Preemptive Multitasking and Inter-Process communication (IPC)

3.1 Clock Interrupts and Preemption

本节要求我们利用时钟中断实现抢断式的任务调度。在目前 JOS 的完成进度上，OS 并没有任何主动手段可以从正在运行中的用户进程手中重新获取执行权，只能等待用户进程执行完后回收或调用 sys_yield() 主动让渡出 CPU。我们在这节实验中要实现抢占式调度，也即在每次时钟中断发生时，OS 重新获得执行权并进行新一轮的进程调度（根据 PartA 内容，JOS 执行 Round-robin 调度）。

在 JOS 中，处于内核态时会关闭外部中断，仅在用户进程执行时开启中断。外部中断由 IRQs 表示，其中 IRQ 0 为时钟中断。

Interrupt discipline

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.

Also uncomment the sti instruction in sched_halt() so that idle CPUs unmask interrupts.

The processor never pushes an error code 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.

这节要为 IRQs 设置 trap 入口，与我们在 Lab3 中设置其他 trap 入口的方式是一样的。根据提示，IRQs 均为无 ErrorCode 的模式，我们修改 trapentry.S，增加对应的接口函数：

```
/* Add in Lab4 */
TRAPHANDLER_NOEC(IRQ_TIMER_HANDLER,    IRQ_OFFSET + IRQ_TIMER)
TRAPHANDLER_NOEC(IRQ_KBD_HANDLER,      IRQ_OFFSET + IRQ_KBD)
TRAPHANDLER_NOEC(IRQ_SERIAL_HANDLER,   IRQ_OFFSET + IRQ_SERIAL)
TRAPHANDLER_NOEC(IRQ_SPURIOUS_HANDLER, IRQ_OFFSET + IRQ_SPURIOUS)
TRAPHANDLER_NOEC(IRQ_IDE_HANDLER,      IRQ_OFFSET + IRQ_IDE)
TRAPHANDLER_NOEC(IRQ_ERROR_HANDLER,    IRQ_OFFSET + IRQ_ERROR)
```

修改 trap 中设置 IDT 的部分的代码：

```
void
trap_init(void)
{
    extern struct Segdesc gdt[];
    ...
    // Add in Lab4
    extern void IRQ_TIMER_HANDLER();
    extern void IRQ_KBD_HANDLER();
    extern void IRQ_SERIAL_HANDLER();
    extern void IRQ_SPURIOUS_HANDLER();
    extern void IRQ_IDE_HANDLER();
    extern void IRQ_ERROR_HANDLER();
    ...
    // Add in Lab4
    SETGATE(idt[IRQ_OFFSET + IRQ_TIMER], 0, GD_KT, IRQ_TIMER_HANDLER, 0);
    SETGATE(idt[IRQ_OFFSET + IRQ_KBD], 0, GD_KT, IRQ_KBD_HANDLER, 0);
    SETGATE(idt[IRQ_OFFSET + IRQ_SERIAL], 0, GD_KT, IRQ_SERIAL_HANDLER, 0);
    SETGATE(idt[IRQ_OFFSET + IRQ_SPURIOUS], 0, GD_KT, IRQ_SPURIOUS_HANDLER, 0);
    SETGATE(idt[IRQ_OFFSET + IRQ_IDE], 0, GD_KT, IRQ_IDE_HANDLER, 0);
    SETGATE(idt[IRQ_OFFSET + IRQ_ERROR], 0, GD_KT, IRQ_ERROR_HANDLER, 0);

    ...
}
```

最后再根据提示修改 sched_halt() 函数，增加打开中断的指令即可。

Handling Clock Interrupts

完成入口设置后，我们再实现时钟中断的 handler，实现也比较简单：每次触发 IRQ Timer 时使 JOS 进行一次系统调度即可。

> **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.

```
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();
    }

    ...
}
```

3.2 Inter-Process communication (IPC)

这一节要求我们完成 JOS 的进程间通信功能。JOS 为用户进程提供的进程通信方法由两个系统调用来实现：sys_ipc_try_send 与 sys_ipc_recv。前者用于向某一 envid 指定的进程发送消息（方式是传递一个在当前进程中映射的虚拟地址，JOS 找到它对应的物理页，并将其映射到指定进程的 VAS 中），后者用于将当前进程设置为接收信息的状态并挂起（挂起前指定接收到的物理页应当映射的 VA），直到有任一进程向其通讯后再重新变为可调度状态。

JOS 在 user-lib 中进一步封装了这两个 API，得到：ipc_send() 与 ipc_recv()。用户可以直接使用这两个 API 进行通信（实例：user/sendpage.c）。

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. user/primes will generate for each prime number a new environment until JOS runs out of environments. You might find it interesting to read user/primes.c to see all the forking and IPC going on behind the scenes.

sys_ipc_recv

该函数负责设置接受新页映射的 VA。当设置完成后，进程将被挂起（即暂时不被调度），直到有某一进程向其通信并设置 VA 映射后，该进程才重新可调度。

```
static int
sys_ipc_recv(void *dstva)
{
    // LAB 4: Your code here.
    if((uintptr_t)dstva < UTOP && (uintptr_t)dstva != ROUNDDOWN((uintptr_t)dstva, PGSIZE))
        return -E_INVAL;

    curenv->env_status = ENV_NOT_RUNNABLE;
    curenv->env_ipc_dstva = dstva;
    curenv->env_ipc_recving = true;
    sched_yield();
    // should not come here
    return 0;
}
```

这里有一点需要注意：当调用 sched_yield() 后，实际上进程再次被调用时不会返回该执行点处（直接自系统调用处返回），因此在后续我们实现 recv 的恢复时，应当注意对返回状态的设置问题（见下文 send 函数的实现）。

sys_ipc_try_send

send 函数首先找到 srcva 处映射的物理页，检查其各 flag 位符合要求后，将该物理页映射到通信目标进程指定的 dstva 处，并将目标进程重新设置为可调度的。需要注意的是：此处有对目标进程 trapFrame-eax 寄存器设置的语句（EAX 存放 trap 的返回值）。上文我们提到，当等待通信的进程再次被调度时，它会直接从系统调用返回，而在 recv 中我们还未能正确设置其返回值，因此我们需要在这一步完成对目标进程系统调用返回值的设置。

```
static int
sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned perm)
{
    // LAB 4: Your code here.
    struct Env *dstEnv;
    // Error1: No dst-env
    if(envid2env(envid, &dstEnv, 0) < 0)
        return -E_BAD_ENV;
    // Error2: dst-env is not recving
    if(!dstEnv->env_ipc_recving)
        return -E_IPC_NOT_RECV;
    // Error3: not page-aligned / not mapped / PTE_W for read-only page
    struct PageInfo *pp;
    pte_t *srcPte;
    pp = page_lookup(curenv->env_pgdir, srcva, &srcPte);
    if(
        (uintptr_t)srcva < UTOP &&
        ((uintptr_t)srcva != ROUNDDOWN((uintptr_t)srcva, PGSIZE) ||
        !pp || ((perm & PTE_W) && !(*srcPte & PTE_W)) ||
        !(perm & PTE_U) || !(perm & PTE_P) || (perm & (~PTE_SYSCALL)))
    ){ return -E_INVAL; }

    int r;
    if(
        (uintptr_t)srcva < UTOP &&
        (r = page_insert(dstEnv->env_pgdir, pp, dstEnv->env_ipc_dstva, perm) < 0)
    ){ return r; }

    dstEnv->env_ipc_recving = false;
    dstEnv->env_ipc_from = curenv->env_id;
    dstEnv->env_ipc_value = value;
    dstEnv->env_ipc_perm = perm;
    dstEnv->env_status = ENV_RUNNABLE;
    dstEnv->env_tf.tf_regs.reg_eax = 0;  // !
    return 0;

}
```

ipc_recv

该 lib 函数是对系统调用 recv 的一层封装。需要注意的点是：当指定的地址 pg 为 0 的时候，我们无法完成正确的通信，而且我们需要将这一点传达给 send 函数（忽略此次通信）。设置为 0 的地址并不能完成这一点。根据我们上面对两个系统调用的实现，我们可以将 pg 设置为边界值 UTOP，这样系统调用将自然返回。

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

    int r = sys_ipc_recv(pg);
    if(r < 0)
    {
        if(from_env_store)
            *from_env_store = 0;
        if(perm_store)
            *perm_store = 0;
        return r;
    }
    if(from_env_store)
        *from_env_store = thisenv->env_ipc_from;
    if(perm_store)
        *perm_store = thisenv->env_ipc_perm;

    return thisenv->env_ipc_value;
}
```

ipc_send

send 函数的 lib-version 实现也是同理。（未解决：为什么使用 while）

```
void
ipc_send(envid_t to_env, uint32_t val, void *pg, int perm)
{
    // LAB 4: Your code here.
    if(pg == NULL)
        pg = (void *)UTOP;

    int r;
    while((r = sys_ipc_try_send(to_env, val, pg, perm)) < 0)
    {
        if(r != -E_IPC_NOT_RECV)
            panic("ipc_send error: sys_ipc_try_send failed.");
        sys_yield();
    }
}
```