操作系统实习报告

曹胜操 - 1500012838

Lab 4: Preemptive Multitasking

在这一Lab中，我们将在用户模式运行环境的基础之上实现多任务的并发。首先我们需要添加多处理器的支持，设计一个基本的调度算法，以及提供用户环境相关的系统调用接口。更进一步的内容包括类似于Unix中的`fork()`函数以及进程间通信。

Part A: Multipr ocessor Support and Cooper ative Multitasking

我们将在JOS上支持SMP (Symmetric Multiprocessing)。在启动时，不同的处理器还是有区别的，因为首先会有一个BSP (Bootstrap Processor)来将系统初始化，并激活其它的APs (Application Processors)。我们之前已有的JOS代码都是在BSP上运行的。

每个CPU都有一个LAPIC单元，用以传递中断信号。我们将会使用LAPIC的以下功能：

- 确认当前代码运行在哪个CPU上：`cpunum()`
- 由BSP发送`STARTUP`中断给APs来启动其它CPU：`lapic_startup()`
- 触发时钟中断以支持抢占式调度：`apic_init()`

我们可以看一下`kern/lapic.c`中的这些函数。例如`cpunum()`：

```
int
cpunum(void)
{
    if (lapic)
        return lapic[ID] >> 24;
    return 0;
}
```

这里的`lapic`是如何得来的呢？处理器通过MMIO (Memory-Mapped I/O)来访问自己的LAPIC，这样要访问其中的寄存器就可以直接通过内存访问来实现。LAPIC映射到的物理内存地址从`0xFE000000`开始，但是这个地址太大以至于我们不能通过原有的加上`KERNBASE`来得到虚拟地址。我们需要自己建立从`MMIOBASE`的映射。

> Exercise 1. 实现`kern/pmap.c`中的`mmio_map_region`。可以参考`kern/lapic.c`中的`lapic_init`看看这个函数是怎么被用到的。

在`lapic_init()`中：

```
    // lapicaddr is the physical address of the LAPIC's 4K MMIO
    // region.  Map it in to virtual memory so we can access it.
    lapic = mmio_map_region(lapicaddr, 4096);
```

可见，`mmio_map_region()`返回了`lapicaddr`实际被映射到的虚拟地址，后面将用这个地址来对LAPIC进行访问。类似于`mem_init()`中，利用`boot_map_region()`我们可以写出：

```
//
// Reserve size bytes in the MMIO region and map [pa,pa+size) at this
// location.  Return the base of the reserved region.  size does *not*
// have to be multiple of PGSIZE.
//
void *
mmio_map_region(physaddr_t pa, size_t size)
{
    // Where to start the next region.  Initially, this is the
    // beginning of the MMIO region.  Because this is static, its
    // value will be preserved between calls to mmio_map_region
    // (just like nextfree in boot_alloc).
    static uintptr_t base = MMIOBASE;

    // Reserve size bytes of virtual memory starting at base and
    // map physical pages [pa,pa+size) to virtual addresses
    // [base,base+size).  Since this is device memory and not
    // regular DRAM, you'll have to tell the CPU that it isn't
    // safe to cache access to this memory.  Luckily, the page
    // tables provide bits for this purpose; simply create the
    // mapping with PTE_PCD|PTE_PWT (cache-disable and
    // write-through) in addition to PTE_W.  (If you're interested
    // in more details on this, see section 10.5 of IA32 volume
    // 3A.)
    //
    // Be sure to round size up to a multiple of PGSIZE and to
    // handle if this reservation would overflow MMIOLIM (it's
    // okay to simply panic if this happens).
    //
    // Hint: The staff solution uses boot_map_region.
    //
    // Your code here:
    size = ROUNDUP(size, PGSIZE);
    if (base + size > MMIOLIM)
        panic("mmio_map_region: MMIO overflows!\n" );
    boot_map_region(kern_pgdir, base, size, pa, PTE_PCD | PTE_PWT | PTE_W);
    void *ret = (void*)base;
    base += size;
    return ret;
}
```

注释中已经提示了，我们需要注意页对齐、权限和边界的问题。

SMP启动的过程是这样的：

- 在APs启动之前，BSP需要调用`kern/mpconfig.c`中的`mp_init()`，通过BIOS中的内容收集必要的信息。
- APs在实模式下启动，其代码`kern/mpentry.S`就类似于BSP的启动代码`boot/boot.S`。BSP要将`kern/mpentry.S`复制到`MPENTRY_PADDR = 0x7000`的位置，以便APs启动。
- 复制完成后，`boot_aps()`通过逐个发送`STARTUP`这个中断给每个LAPIC来启动每个APs。
- 每个AP执行完`kern/mpentry.S`的代码后，即进入了带分页的保护模式，然后调用`mp_main()`，最后会设置`struct CpuInfo`中的`cpu_status`为`CPU_STARTED`，这样BSP就知道AP已经启动完毕，可以继续唤醒下一个AP了。

> Exercise 2. 阅读 `kern/init.c` 中的 `boot_aps()` `mp_main()` 以及 `kern/mpentry.S` 。理解启动过程中的控制流转换。然后修改 `kern/pmap.c` 中的 `page_init()` 避免把 `MPENTRY_PADDR` 这一页加入空闲页，这样我们才能安全地使用这一地址来运行AP的启动代码。

在 `boot_aps()` 中：

```
// Start the non-boot (AP) processors.
static void
boot_aps(void)
{
    extern unsigned char mpentry_start[], mpentry_end[];
    void *code;
    struct CpuInfo *c;

    // Write entry code to unused memory at MPENTRY_PADDR
    code = KADDR(MPENTRY_PADDR);
    memmove(code, mpentry_start, mpentry_end - mpentry_start);

    // Boot each AP one at a time
    for (c = cpus; c < cpus + ncpu; c++) {
        if (c == cpus + cpunum())  // We've started already.
            continue;

        // Tell mpentry.S what stack to use
        mpentry_kstack = percpu_kstacks[c - cpus] + KSTKSIZE;
        // Start the CPU at mpentry_start
        lapic_startap(c->cpu_id, PADDR(code));
        // Wait for the CPU to finish some basic setup in mp_main()
        while(c->cpu_status != CPU_STARTED)
            ;
    }
}
```

这里BSP把 `kern/mpentry.S` 代码复制到了 `MPENTRY_PADDR` ，然后一个个地启动AP。它通过 `lapic_startap(c->cpu_id, PADDR(code))` 发送启动的中断信号，然后忙等待直到AP启动完成。中断发送后，AP就从 `MPENTRY_PADDR` 开始执行 `kern/mpentry.S` 中的代码。在 `kern/mpentry.S` 中：

```
...
.code16
.globl mpentry_start
mpentry_start:
    cli

    xorw    %ax, %ax
    movw    %ax, %ds
    movw    %ax, %es
    movw    %ax, %ss

    lgdt    MPBOOTPHYS(gdtdesc)
    movl    %cr0, %eax
    orl     $CR0_PE, %eax
    movl    %eax, %cr0

    ljmpl   $(PROT_MODE_CSEG), $(MPBOOTPHYS(start32))

.code32
start32:
    movw    $(PROT_MODE_DSEG), %ax
    movw    %ax, %ds
    movw    %ax, %es
    movw    %ax, %ss
    movw    $0, %ax
    movw    %ax, %fs
    movw    %ax, %gs

    # Set up initial page table. We cannot use kern_pgdir yet because
    # we are still running at a low EIP.
    movl    $(RELOC(entry_pgdir)), %eax
    movl    %eax, %cr3
    # Turn on paging.
    movl    %cr0, %eax
    orl     $(CR0_PE|CR0_PG|CR0_WP), %eax
    movl    %eax, %cr0

    # Switch to the per-cpu stack allocated in boot_aps()
    movl    mpentry_kstack, %esp
    movl    $0x0, %ebp       # nuke frame pointer

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

类似的过程，初始化一些寄存器，设置GDT，进入保护模式，打开分页，最后调用`mp_main()`。在`mp_main()`中：

```
...
    // We are in high EIP now, safe to switch to kern_pgdir
    lcr3(PADDR(kern_pgdir));
    cprintf("SMP: CPU %d starting\n", cpunum());

    lapic_init();
    env_init_percpu();
    trap_init_percpu();
    xchg(&thiscpu->cpu_status, CPU_STARTED); // tell boot_aps() we're up
...
```

在完成一些其它的初始化之后，将`thiscpu->cpu_status`设置为`CPU_STARTED`，告诉BSP启动已完成。之后BSP将继续启动下一个AP。

回到这个Exercise的正题，我们修改`page_init()`只需要加入一个特判：

```
...
    size_t t5 = PGNUM(MPENTRY_PADDR);
    size_t i;
    for (i = 0; i < npages; ++i)
    {
        if (i == t5)
        {
            pages[i].pp_ref = 1;
            pages[i].pp_link = NULL;
            continue;
        }
...
```

> Question 1. 比较`kern/mpentry.S`和`boot/boot.S`。前者是被编译链接到`KERNBASE`地址以上的。宏`MPBOOTPHYS`的目的是什么？为什么在前者中才需要用这个宏？

最主要的区别就是前者中需要用到地址时，都使用了`MPBOOTPHYS`宏，例如`lgdt MPBOOTPHYS(gdtdesc)`（后者中直接是`lgdt gdtdesc`）。宏的定义是：

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

可见，其目的是计算出一个绝对的物理地址。AP是启动于实模式的，只能使用这样的低地址。如果不加这个宏，这里的地址都是链接器算出来的地址，是在`KERNBASE`以上的，无法为AP所用。`boot/boot.S`就不需要了，因为它本身就在低地址上。

在多处理器OS上，区分每个CPU的私有状态是很重要的。`kern/cpu.h`中定义了大多数CPU状态。`struct CpuInfo`存储了CPU相关的变量，`cpunum()`返回当前CPU的ID，`thiscpu`宏指向当前CPU的`struct CpuInfo`。需要注意的CPU状态包括：

- 内核栈：多CPU可能同时陷入内核，因此需要各自独立的内核栈。`percpu_kstacks[NCPU][KSTKSIZE]`保留了这一空间。`inc/memlayout.h`中显示了映射的布局状态。还需要注意的是不同CPU内核栈之间还有`KSTKGAP`大小的额外空间，这样栈溢出时也不会破坏其它CPU的内核栈。
- TSS：同样需要各自独立的TSS来指定不同CPU的内核栈所在位置。TSS位于`cpus[i].cpu_ts`，TSS描述符位于`gdt[(GD_TSS0 >> 3) + i]`。之前所用的`ts`将不再有效。

- 当前环境指针：每个CPU可以并行运行不同环境，所以`curenv`被重新定义为`cpus[cpunum()].cpu_env`。
- 系统寄存器：所有的寄存器对于CPU都是私有的，包括`lcr3()` `ltr()` `lgdt()` `lidt()`等。`env_init_percpu()`和`trap_init_percpu()`会初始化这些私有的寄存器。

> Exercise 3. 修改`kern/pmap.c`中的`mem_init_mp()`来映射每个CPU各自的内核栈，其布局显示在`inc/memlayout.h`中。

通过`inc/memlayout.h`可以看到，在`KSTACKTOP`之下依次是`KSTKSIZE`大小的CPU0的内核栈，`KSTKGAP`大小的保护性无效内存，`KSTKSIZE`大小的CPU1的内核栈，`KSTKGAP`大小的保护性无效内存……总的大小不超过`PTSIZE`。于是得到`mem_init_mp()`：

```
// Modify mappings in kern_pgdir to support SMP
//   - Map the per-CPU stacks in the region [KSTACKTOP-PTSIZE, KSTACKTOP)
//
static void
mem_init_mp(void)
{
    // Map per-CPU stacks starting at KSTACKTOP, for up to 'NCPU' CPUs.
    //
    // For CPU i, use the physical memory that 'percpu_kstacks[i]' refers
    // to as its kernel stack. CPU i's kernel stack grows down from virtual
    // address kstacktop_i = KSTACKTOP - i * (KSTKSIZE + KSTKGAP), and is
    // divided into two pieces, just like the single stack you set up in
    // mem_init:
    //     * [kstacktop_i - KSTKSIZE, kstacktop_i)
    //          -- backed by physical memory
    //     * [kstacktop_i - (KSTKSIZE + KSTKGAP), kstacktop_i - KSTKSIZE)
    //          -- not backed; so if the kernel overflows its stack,
    //             it will fault rather than overwrite another CPU's stack.
    //             Known as a "guard page".
    //     Permissions: kernel RW, user NONE
    //
    // LAB 4: Your code here:
    int i;
    uintptr_t kstacktop_i;
    for (i = 0; i < NCPU; ++i)
    {
        kstacktop_i = KSTACKTOP - i * (KSTKSIZE + KSTKGAP);
        boot_map_region(kern_pgdir, kstacktop_i - KSTKSIZE, KSTKSIZE,
                PADDR(&percpu_kstacks[i]), PTE_W);
    }

}
```

> Exercise 4. 修改`kern/trap.c`中的`trap_init_percpu()`来初始化所有CPU的TSS及其描述符。

在Lab 3里这个函数正确地初始化了BSP的TSS及其描述符，我们只需改成找到当前CPU进行初始化就可以了：

```
// Initialize and load the per-CPU TSS and IDT
void
trap_init_percpu(void)
{
    // The example code here sets up the Task State Segment (TSS) and
    // the TSS descriptor for CPU 0. But it is incorrect if we are
    // running on other CPUs because each CPU has its own kernel stack.
    // Fix the code so that it works for all CPUs.
    //
    // Hints:
    //   - The macro "thiscpu" always refers to the current CPU's
    //     struct CpuInfo;
    //   - The ID of the current CPU is given by cpunum() or
    //     thiscpu->cpu_id;
    //   - Use "thiscpu->cpu_ts" as the TSS for the current CPU,
    //     rather than the global "ts" variable;
    //   - Use gdt[(GD_TSS0 >> 3) + i] for CPU i's TSS descriptor;
    //   - You mapped the per-CPU kernel stacks in mem_init_mp()
    //
    // ltr sets a 'busy' flag in the TSS selector, so if you
    // accidentally load the same TSS on more than one CPU, you'll
    // get a triple fault.  If you set up an individual CPU's TSS
    // wrong, you may not get a fault until you try to return from
    // user space on that CPU.
    //
    // LAB 4: Your code here:

    // Setup a TSS so that we get the right stack
    // when we trap to the kernel.
    int cpu_id = cpunum();
    thiscpu->cpu_ts.ts_esp0 = KSTACKTOP - cpu_id * (KSTKSIZE + KSTKGAP);
    thiscpu->cpu_ts.ts_ss0 = GD_KD;
    thiscpu->cpu_ts.ts_iomb = sizeof(struct Taskstate);

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

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

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

我们还需要解决运行内核代码时不同CPU之间的竞争问题。最简单的办法是使用大内核锁，这是一种全局的锁，只要进入内核模式就必须获得该锁，而返回用户模式时释放该锁。这样，任何时刻只有一个CPU可以运行内核代码，其它的CPU只能等待。

`kern/spinlock.h`声明了这个大内核锁`kernel_lock`，提供了`lock_kernel()` `unlock_kernel()`来获得或释放锁。我们可以查看其代码。JOS中的锁是通过自旋锁实现的，结构体中有意义的只是一个`unsigned locked`。具体实现在`kern/spinlock.c`中：

```
// Acquire the lock.
// Loops (spins) until the lock is acquired.
// Holding a lock for a long time may cause
// other CPUs to waste time spinning to acquire it.
void
spin_lock(struct spinlock *lk)
{
#ifdef DEBUG_SPINLOCK
    if (holding(lk))
        panic("CPU %d cannot acquire %s: already holding" , cpunum(), lk->name);
#endif

    // The xchg is atomic.
    // It also serializes, so that reads after acquire are not
    // reordered before it.
    while (xchg(&lk->locked, 1) != 0)
        asm volatile ("pause");

    // Record info about lock acquisition for debugging.
#ifdef DEBUG_SPINLOCK
    lk->cpu = thiscpu;
    get_caller_pcs(lk->pcs);
#endif
}

// Release the lock.
void
spin_unlock(struct spinlock *lk)
{
#ifdef DEBUG_SPINLOCK
    if (!holding(lk)) {
        int i;
        uint32_t pcs[10];
        // Nab the acquiring EIP chain before it gets released
        memmove(pcs, lk->pcs, sizeof pcs);
        cprintf("CPU %d cannot release %s: held by CPU %d\nAcquired at:" ,
            cpunum(), lk->name, lk->cpu->cpu_id);
        for (i = 0; i < 10 && pcs[i]; i++) {
            struct Eipdebuginfo info;
            if (debuginfo_eip(pcs[i], &info) >= 0)
                cprintf("  %08x %s:%d: %.*s+%x\n" , pcs[i],
                    info.eip_file, info.eip_line,
                    info.eip_fn_namelen, info.eip_fn_name,
                    pcs[i] - info.eip_fn_addr);
            else
                cprintf("  %08x\n", pcs[i]);
        }
        panic("spin_unlock");
    }

    lk->pcs[0] = 0;
    lk->cpu = 0;
#endif
```

```
    // The xchg instruction is atomic (i.e. uses the "lock" prefix) with
    // respect to any other instruction which references the same memory.
    // x86 CPUs will not reorder loads/stores across locked instructions
    // (vol 3, 8.2.2). Because xchg() is implemented using asm volatile,
    // gcc will not reorder C statements across the xchg.
    xchg(&lk->locked, 0);
}
```

除掉调试信息之外，对锁所做的只是通过原子的`xchg()`来修改锁中的`locked`，获得锁时如果暂时不能就暂停等待锁被释放。

我们需要在四处使用大内核锁：

- `i386_init()`中BSP唤醒APs之前获得锁
- `mp_main()`中AP初始化完之后，`sched_yield()`之前获得锁
- `trap()`中从用户模式陷入之后获得锁（但是如果之前已经在内核模式则不用）
- `env_run()`中切换到用户模式之前释放锁

> Exercise 5. 按上面所描述的使用大内核锁。

`i386_init()`中：

```
...
    // Lab 4 multitasking initialization functions
    pic_init();

    // Acquire the big kernel lock before waking up APs
    // Your code here:
    lock_kernel();

    // Starting non-boot CPUs
    boot_aps();
...
```

`mp_main()`中：

```
...
    xchg(&thiscpu->cpu_status, CPU_STARTED); // tell boot_aps() we're up

    // Now that we have finished some basic setup, call sched_yield()
    // to start running processes on this CPU.  But make sure that
    // only one CPU can enter the scheduler at a time!
    //
    // Your code here:
    lock_kernel();
    sched_yield();
...
```

`trap()`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);
...
```

`env_run()`中：

```
...
    lcr3(PADDR(curenv->env_pgdir));
    unlock_kernel();
    env_pop_tf(&curenv->env_tf);
...
```

> Question 2. 似乎使用大内核锁可以保证只有一个CPU运行内核代码，那么为什么还需要各自独立的内核栈？

在实现之后，发现其实前面所说的似乎有些不准确。比如说其实从用户模式trap进入内核时，已经在内核模式下压入了`struct Trapframe`等信息，调用了`trap()`函数，直到在这里`lock_kernel()`完成后才算是获得了大内核锁，因此其实是在内核模式下进行的等待。如果只用同一个内核栈，那么`struct Trapframe`有可能会因为不同CPU同时的处理造成混乱。

接下来我们在JOS中实现RR (Round-Robin)调度，具体如下：

- `kern/sched.c`中的`sched_yield()`负责选出一个新的环境来运行，它会顺序地、循环地在`envs[]`中寻找一个状态为`ENV_RUNNABLE`的环境，通过`env_run()`来运行它。
- `sched_yield()`不能在两个CPU上同时运行一个环境，环境状态为`ENV_RUNNING`时，其它的CPU不能运行。
- 用户程序可以通过系统调用`sys_yield()`来调用`sched_yield()`，主动让出CPU运行其它环境。

> Exercise 6. 在`sched_yield()`中实现RR调度，并且修改`syscall()`。
>
> 修改`kern/init.c`来创建三个运行`user/yield.c`的环境，并用`make qemu` `make qemu CPUS=2`来测试。最终没有可运行环境时，调度器将会调用内核监视器。

在`sched_yield()`中，我们需要做的就是从上一个运行的环境之后开始寻找一个可运行的环境。如果上一个环境并不存在，就从头开始找。如果所有其它环境都不是可运行的，则还是运行上一个运行的环境。第一次写这个函数时出了一个bug，后面才遇到问题，花了不少时间来调试才发现问题。在后面的部分会回到这里的代码做一个解释。

```
// Choose a user environment to run and run it.
void
sched_yield(void)
{
    struct Env *idle;

    // Implement simple round-robin scheduling.
    //
    // Search through 'envs' for an ENV_RUNNABLE environment in
    // circular fashion starting just after the env this CPU was
    // last running.  Switch to the first such environment found.
    //
    // If no envs are runnable, but the environment previously
    // running on this CPU is still ENV_RUNNING, it's okay to
    // choose that environment.
    //
    // Never choose an environment that's currently running on
    // another CPU (env_status == ENV_RUNNING). If there are
    // no runnable environments, simply drop through to the code
    // below to halt the cpu.

    // LAB 4: Your code here.
    struct Env *cur = thiscpu->cpu_env;
    int start = 1, end = 0, i;
    if (cur)
    {
        if (cur->env_status == ENV_RUNNING)
            cur->env_status = ENV_RUNNABLE;
        start = (cur - envs + 1) % NENV;
        end = cur - envs;
    }
    else
    {
        if (envs[0].env_status == ENV_RUNNABLE)
            env_run(&envs[0]);
    }
    for (i = start; i != end; i = (i + 1) % NENV)
        if (envs[i].env_status == ENV_RUNNABLE)
            env_run(&envs[i]);
    if (cur && cur->env_status == ENV_RUNNABLE)
        env_run(cur);

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

同样要记得在系统调用相关源代码中添加`sys_yield()`。修改`kern/init.c`之后，如果用`make qemu CPUS=1`测试，大概会看到这样的结果：

```
...
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.
...
```

但是，如果在这里是用更多的CPU测试，则会产生一般保护错误。Exercise中的解释是我们还没有处理时钟中断。

```
...
[00001000] free env 00001000
TRAP frame at 0xf02b407c from CPU 1
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfdfe0
  oesp 0xeffeffdc
  ebx  0x00000000
  edx  0x00000000
  ecx  0x00000000
  eax  0x00001001
  es   0x----0023
  ds   0x----0023
  trap 0x0000000d General Protection
  err  0x00000102
  eip  0x00800ae9
  cs   0x----001b
  flag 0x00000246
  esp  0xeebfdfd4
  ss   0x----0023
...
```

> Question 3. 在`env_run()`中应该调用了`lcr3()`，在调用的前后，代码可能会使用变量`e`，即`env_run`的参数。载入`%cr3`时，地址空间被立即切换了。为什么`e`在切换前后都可以指向正确的位置?

注意到`env_run()`运行于内核态，`e`也是属于内核地址空间的。我们在JOS中实现的任何页表，至少在内核区域的映射都是一样的。所以即使地址空间切换了，我们还是可以正确地使用这些内容。

> Question 4. 无论何时内核进行进程切换时，它都必须保证旧环境的寄存器都被保存了，以便将来能够恢复。为什么？这是在哪里发生的?

进程上下文切换当然要保存上下文。在用户环境被trap时，`trapentry.S`中的代码会压入用户环境原来的寄存器，在后续的处理中则会保存到环境中。这样任何CPU都可以通过环境中保存的上下文恢复这个环境。

我们现在来实现一些系统调用，允许用户环境来创建、启动新的用户环境。Unix中使用`fork()`来创建新的进程，它会复制调用的父进程的整个地址空间来创建一个子进程，唯一可见的区别是不同的ID。默认情况下，每个进程的地址空间是独立的。

接下来我们提供更加初级的系统调用来创建新环境，有了它们也可以在用户态下实现类似Unix中的`fork()`。这些系统调用包括：

- `sys_exofork`创建一个几乎空白的新环境，没有用户区域的内存地址映射，也不可运行。而寄存器状态和调用环境是一样的。在父环境中返回子环境的`envid_t`，在子环境中返回`0`。
- `sys_env_set_status`可以设置一个指定环境的状态。
- `sys_page_alloc`分配一个物理页并建立映射。
- `sys_page_map`复制一个环境的地址空间中的某一页的映射给另一个环境。两个环境可以共享内存。
- `sys_page_unmap`取消一个物理页的映射。

这些系统调用中，如果环境的ID设置为`0`则表示当前环境。这是由`kern/env.c`中的`envid2env()`完成的。`user/dumbfork.c`中有一个非常初级的`fork()`的实现，它只是直接地将父环境的整个地址空间都复制过去。不过我们可以用它来测试。

> Exercise 7. 实现上面所述的系统调用。调用`envid2env()`时，`checkperm`设置为`1`。

`envid2env()`的代码中，关于`checkperm`的部分：

```
...
    // Check that the calling environment has legitimate permission
    // to manipulate the specified environment.
    // If checkperm is set, the specified environment
    // must be either the current environment
    // or an immediate child of the current environment.
    if (checkperm && e != curenv && e->env_parent_id != curenv->env_id) {
        *env_store = 0;
        return -E_BAD_ENV;
    }
...
```

其作用是检查调用的环境是否对这个环境有权限，即是不是同一个环境或者是不是父环境。

回到`kern/syscall.c`中，我们需要实现的函数如下：

```
// Allocate a new environment.
// Returns envid of new environment, or < 0 on error.  Errors are:
//  -E_NO_FREE_ENV if no free environment is available.
//  -E_NO_MEM on memory exhaustion.
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 *child = NULL, *parent = curenv;
    int r = env_alloc(&child, parent->env_id);
    if (r < 0) return r;
    child->env_status = ENV_NOT_RUNNABLE;
    child->env_tf = parent->env_tf;
    child->env_tf.tf_regs.reg_eax =  0;
    return child->env_id;
}

// Set envid's env_status to status, which must be ENV_RUNNABLE
// or ENV_NOT_RUNNABLE.
//
// Returns 0 on success, < 0 on error.  Errors are:
//  -E_BAD_ENV if environment envid doesn't currently exist,
//      or the caller doesn't have permission to change envid.
//  -E_INVAL if status is not a valid status for an environment.
static int
sys_env_set_status(envid_t envid,  int status)
{
    // Hint: Use the 'envid2env' function from kern/env.c to translate an
    // envid to a struct Env.
    // You should set envid2env's third argument to 1, which will
    // check whether the current environment has permission to set
    // envid's status.

    // LAB 4: Your code here.
    struct Env *e = NULL;
    int r = envid2env(envid, &e,  1);
    if (r < 0) return r;
    if (status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE)
        return -E_INVAL;
    e->env_status = status;
    return 0;
}

// Allocate a page of memory and map it at 'va' with permission
// 'perm' in the address space of 'envid'.
// The page's contents are set to 0.

// If a page is already mapped at 'va', that page is unmapped as a
```

```
// side effect.
//
// perm -- PTE_U | PTE_P must be set, PTE_AVAIL | PTE_W may or may not be set,
//         but no other bits may be set.  See PTE_SYSCALL in inc/mmu.h.
//
// Return 0 on success, < 0 on error.  Errors are:
//  -E_BAD_ENV if environment envid doesn't currently exist,
//      or the caller doesn't have permission to change envid.
//  -E_INVAL if va >= UTOP, or va is not page-aligned.
//  -E_INVAL if perm is inappropriate (see above).
//  -E_NO_MEM if there's no memory to allocate the new page,
//      or to allocate any necessary page tables.
static int
sys_page_alloc(envid_t envid, void *va,  int perm)
{
    // Hint: This function is a wrapper around page_alloc() and
    //   page_insert() from kern/pmap.c.
    //   Most of the new code you write should be to check the
    //   parameters for correctness.
    //   If page_insert() fails, remember to free the page you
    //   allocated!

    // LAB 4: Your code here.
    struct Env *e = NULL;
    int r = envid2env(envid, &e,  1);
    if (r < 0) return r;
    if ((uint32_t)va >= UTOP || (uint32_t)va % PGSIZE) return -E_INVAL;
    if (!(perm & PTE_U) || !(perm & PTE_P) || perm & ~PTE_SYSCALL) return -E_INVAL;
    struct PageInfo *pp = page_alloc( 1);
    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;
}

// Map the page of memory at 'srcva' in srcenvid's address space
// at 'dstva' in dstenvid's address space with permission 'perm'.
// Perm has the same restrictions as in sys_page_alloc, except
// that it also must not grant write access to a read-only
// page.
//
// Return 0 on success, < 0 on error.  Errors are:
//  -E_BAD_ENV if srcenvid and/or dstenvid doesn't currently exist,
//      or the caller doesn't have permission to change one of them.
//  -E_INVAL if srcva >= UTOP or srcva is not page-aligned,
//      or dstva >= UTOP or dstva is not page-aligned.
//  -E_INVAL is srcva is not mapped in srcenvid's address space.
//  -E_INVAL if perm is inappropriate (see sys_page_alloc).

//  -E_INVAL if (perm & PTE_W), but srcva is read-only in srcenvid's
```

```
//      address space.
//  -E_NO_MEM if there's no memory to allocate any necessary page tables.
static int
sys_page_map(envid_t srcenvid, void *srcva,
         envid_t dstenvid, void *dstva,  int perm)
{
    // Hint: This function is a wrapper around page_lookup() and
    //   page_insert() from kern/pmap.c.
    //   Again, most of the new code you write should be to check the
    //   parameters for correctness.
    //   Use the third argument to page_lookup() to
    //   check the current permissions on the page.

    // LAB 4: Your code here.
    struct Env *srcE = NULL, *dstE = NULL;
    int r = 0;
    r = envid2env(srcenvid, &srcE,  1);
    if (r < 0) return r;
    r = envid2env(dstenvid, &dstE,  1);
    if (r < 0) return r;
    if ((uint32_t)srcva >= UTOP || (uint32_t)srcva % PGSIZE) return -E_INVAL;
    if ((uint32_t)dstva >= UTOP || (uint32_t)dstva % PGSIZE) return -E_INVAL;
    pte_t *ptep = NULL;
    struct PageInfo *pp = page_lookup(srcE->env_pgdir, srcva, &ptep);
    if (!pp) return -E_INVAL;
    if (!(perm & PTE_U) || !(perm & PTE_P) || perm & ~PTE_SYSCALL) return -E_INVAL;
    if (perm & PTE_W && !(*ptep & PTE_W)) return -E_INVAL;
    r = page_insert(dstE->env_pgdir, pp, dstva, perm);
    if (r < 0) return r;
    return 0;
}

// Unmap the page of memory at 'va' in the address space of 'envid'.
// If no page is mapped, the function silently succeeds.
//
// Return 0 on success, < 0 on error.  Errors are:
//  -E_BAD_ENV if environment envid doesn't currently exist,
//      or the caller doesn't have permission to change envid.
//  -E_INVAL if va >= UTOP, or va is not page-aligned.
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 = NULL;
    int r = envid2env(envid, &e,  1);
    if (r < 0) return r;
    if ((uint32_t)va >= UTOP || (uint32_t)va % PGSIZE) return -E_INVAL;
    page_remove(e->env_pgdir, va);
    return 0;
}
```

其实并不困难，主要是进行一些必要的检查。用`dumbfork`来测试的结果如下：

```
...
[00000000] new env 00001000
[00001000] new env 00001001
0: I am the parent!
0: I am the child!
1: I am the parent!
1: I am the child!
2: I am the parent!
2: I am the child!
3: I am the child!
3: I am the parent!
4: I am the child!
4: I am the parent!
5: I am the child!
5: I am the parent!
6: I am the child!
6: I am the parent!
7: I am the child!
7: I am the parent!
8: I am the child!
8: I am the parent!
9: I am the child!
9: I am the parent!
10: I am the child!
[00001000] exiting gracefully
[00001000] free env 00001000
11: I am the child!
12: I am the child!
13: I am the child!
14: I am the child!
15: I am the child!
16: I am the child!
17: I am the child!
18: I am the child!
19: I am the child!
[00001001] exiting gracefully
[00001001] free env 00001001
No runnable environments in the system!
...
```

Part B: Copy-on-Write Fork

Unix通过`fork()`来创建进程，它会复制整个地址空间。但是我们也知道`fork()`之后子进程往往立刻就会调用`exec()`来运行一个新的程序，这样花在复制父进程地址空间上的时间被极大地浪费了，因为其实只会用到很少一部分的内存。因此，Unix充分利用虚拟内存机制来允许两个进程共享内存，直到其中一个实际进行了修改，这被称为COW (Copy On Write)。这其实是通过在复制内存映射时进行特殊的标记来实现的。接下来我们将实现这样的`fork()`。

由于JOS中是在用户级实现的`fork()`，首先我们也需要实现在用户级处理缺页异常。用户环境需要通过一个新的系统调用`sys_env_set_pgfault_upcall`向内核注册一个处理函数入口，在`struct Env`中已经有了一个新的域`env_pgfault_upcall`来记录这个信息。

> Exercise 8. 实现`sys_env_set_pgfault_upcall`系统调用。需要检查环境的权限。

首先利用`envid2env()`来转换得到环境，并且保证调用环境权限足够。然后给其中的`env_pgfault_upcall`赋值即可。

```
// Set the page fault upcall for 'envid' by modifying the corresponding struct
// Env's 'env_pgfault_upcall' field.  When 'envid' causes a page fault, the
// kernel will push a fault record onto the exception stack, then branch to
// 'func'.
//
// Returns 0 on success, < 0 on error.  Errors are:
//  -E_BAD_ENV if environment envid doesn't currently exist,
//      or the caller doesn't have permission to change envid.
static int
sys_env_set_pgfault_upcall (envid_t envid, void *func)
{
    // LAB 4: Your code here.
    struct Env *e = NULL;
    int r = envid2env(envid, &e, 1);
    if (r < 0) return r;
    e->env_pgfault_upcall  = func;
    return 0;
}
```

当然，还是需要在`syscall()`中添加对应的条目：

```
    case SYS_env_set_pgfault_upcall :
        return sys_env_set_pgfault_upcall (a1, (void*)a2);
```

以及`lib/syscall.c`中：

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

当在用户模式下发生缺页时，内核会重启用户环境，在与普通栈所不同的用户异常栈上运行用户级的异常处理函数。这个用户异常栈位于从`UXSTACKTOP`向下的一个页中。

我们还需要修改`kern/trap.c`以通过用户模式处理缺页异常。如果没有已注册的处理函数，JOS仍会和以前一样直接终止用户环境。否则内核会在用户异常栈中设置好触发异常时的trap frame，其结构由`struct UTrapframe`定义，主要有各种寄存器的值，以及触发异常的虚拟地址。然后内核会安排用户环境在用户异常栈上利用这个trap frame来运行异常处理函数，如果之前已经在异常栈上了（即缺页异常处理函数本身又触发了异常，我们可以通过`tf->tf_esp`所在的范围来判断），则就接着在下方新建栈帧。注意，这里需要先压入一个32位的字，再是`struct UTrapframe`，其原因我们将在之后讨论。

我们也可以更仔细地在`inc/trap.h`中对比一下`struct UTrapframe`和`struct Trapframe`：

```
struct Trapframe {
    struct PushRegs tf_regs;
    uint16_t tf_es;
    uint16_t tf_padding1;
    uint16_t tf_ds;
    uint16_t tf_padding2;
    uint32_t tf_trapno;
    /* below here defined by x86 hardware */
    uint32_t tf_err;
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding3;
    uint32_t tf_eflags;
    /* below here only when crossing rings, such as from user to kernel */
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding4;
} __attribute__((packed));

struct UTrapframe {
    /* information about the fault */
    uint32_t utf_fault_va;  /* va for T_PGFLT, 0 otherwise */
    uint32_t utf_err;
    /* trap-time return state */
    struct PushRegs utf_regs;
    uintptr_t utf_eip;
    uint32_t utf_eflags;
    /* the trap-time stack to return to */
    uintptr_t utf_esp;
} __attribute__((packed));
```

可见，少的是各种段寄存器的值、trap编号、错误码，多的是触发异常的虚拟地址。这是为处理缺页异常而特别设计的，因为在用户级处理缺页，不存在段的切换，所以不需要保存段寄存器，而需要确定是哪个虚拟地址造成了异常。整理一下缺页异常处理的流程如下：

- 用户模式下试图使用某个虚拟地址，触发了缺页异常
- 通过trap机制进入内核模式下的`trap()`，其trap frame位于内核异常栈中
- 进入到`page_fault_handler()`中，如果环境没有注册处理函数的话就直接终止，否则
- 如果触发时不在用户异常栈中，则在`UXSTACKTOP`以下建立`struct UTrapframe`，否则在原来的`tf->tf_esp - 4`以下建立，需要留出一个字的空白
- 设置该环境所应跳转到的处理函数入口以及栈指针，并运行环境切换到用户模式
- 处理函数完成后，返回用户环境中继续执行触发缺页的那一条语句

> Exercise 9. 实现`kern/trap.c`中的`page_fault_handler`。在写入用户异常栈之前，需要进行检查。如果用户异常栈的空间用完了会怎么样？

在用户级处理异常时，的确有可能会是递归的。但是我们给用户异常栈分配的空间是固定的一页，如果超出了这个空间，说明更可能是用户程序的问题，不能允许用户异常栈的无限扩增，只能终止该用户环境。我们可以用`user_mem_assert()`来检查是否可以在用户异常栈中写入。具体实现如下：

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

    // Read processor's CR2 register to find the faulting address
    fault_va = rcr2();

    // Handle kernel-mode page faults.

    // LAB 3: Your code here.
    if ((tf->tf_cs & 3) == 0)
        panic("page_fault_handler: Kernel mode page fault!\n" );

    // We've already handled kernel-mode exceptions, so if we get here,
    // the page fault happened in user mode.

    // Call the environment's page fault upcall, if one exists.  Set up a
    // page fault stack frame on the user exception stack (below
    // UXSTACKTOP), then branch to curenv->env_pgfault_upcall.
    //
    // The page fault upcall might cause another page fault, in which case
    // we branch to the page fault upcall recursively, pushing another
    // page fault stack frame on top of the user exception stack.
    //
    // The trap handler needs one word of scratch space at the top of the
    // trap-time stack in order to return.  In the non-recursive case, we
    // don't have to worry about this because the top of the regular user
    // stack is free.  In the recursive case, this means we have to leave
    // an extra word between the current top of the exception stack and
    // the new stack frame because the exception stack _is_ the trap-time
    // stack.
    //
    // If there's no page fault upcall, the environment didn't allocate a
    // page for its exception stack or can't write to it, or the exception
    // stack overflows, then destroy the environment that caused the fault.
    // Note that the grade script assumes you will first check for the page
    // fault upcall and print the "user fault va" message below if there is
    // none.  The remaining three checks can be combined into a single test.
    //
    // Hints:
    //   user_mem_assert() and env_run() are useful here.
    //   To change what the user environment runs, modify 'curenv->env_tf'
    //   (the 'tf' variable points at 'curenv->env_tf').

    // LAB 4: Your code here.
    if (curenv->env_pgfault_upcall)
    {
        struct UTrapframe *utf;
        if (UXSTACKTOP - PGSIZE <= tf->tf_esp && tf->tf_esp < UXSTACKTOP)
            utf = (struct UTrapframe*)(tf->tf_esp - sizeof(struct UTrapframe) - 4);
        else
            utf = (struct UTrapframe*)(UXSTACKTOP - sizeof(struct UTrapframe));
```

```
        user_mem_assert(curenv, (void*)utf, sizeof(struct UTrapframe), PTE_W);
        utf->utf_fault_va = fault_va;
        utf->utf_err = tf->tf_err;
        utf->utf_regs = tf->tf_regs;
        utf->utf_eip = tf->tf_eip;
        utf->utf_eflags = tf->tf_eflags;
        utf->utf_esp = tf->tf_esp;
        curenv->env_tf.tf_eip = (uintptr_t)curenv->env_pgfault_upcall;
        curenv->env_tf.tf_esp = (uintptr_t)utf;
        env_run(curenv);
    }

    // Destroy the environment that caused the fault.
    cprintf("[%08x] user fault va %08x ip %08x\n",
        curenv->env_id, fault_va, tf->tf_eip);
    print_trapframe(tf);
    env_destroy(curenv);
}
```

接下来我们实现汇编代码的部分，负责调用处理缺页异常的函数，并且继续执行触发异常的语句。这个汇编代码才是将来注册的入口。

> Exercise 10. 实现`lib/pfentry.S`中的`_pgfault_upcall`。返回触发异常的原点是比较有趣的部分。返回并不是通过内核中转的，难点在于同时切换栈帧和载入EIP。

这部分也是参考了一些已有的资料，其设计的确十分巧妙。根据我们在上面一个Exercise中的实现，在`_pgfault_upcall`的入口处，栈指针指向这个`struct UTrapframe`，从低到高依次为`uint32_t utf_fault_va` `uint32_t utf_err` `struct PushRegs utf_regs` `uintptr_t utf_eip` `uint32_t utf_eflags` `uintptr_t utf_esp`以及一个字的空白。

我们需要先调用C代码的那个处理函数：

```
.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
```

接下来就比较困难了。我们需要根据trap frame中的信息，正确地恢复当时的寄存器状态，特别是切换栈帧和载入EIP。以下是给出的注释中的提示：

```
    // Now the C page fault handler has returned and you must return
    // to the trap time state.
    // Push trap-time %eip onto the trap-time stack.
    //
    // Explanation:
    //   We must prepare the trap-time stack for our eventual return to
    //   re-execute the instruction that faulted.
    //   Unfortunately, we can't return directly from the exception   stack:
    //   We can't call 'jmp', since that requires that we load the address
    //   into a register, and all registers must have their trap-time
    //   values after the return.
    //   We can't call 'ret' from the exception stack either, since if we
    //   did, %esp would have the wrong value.
    //   So instead, we push the trap-time % eip onto the *trap-time* stack!
    //   Below we'll switch to that stack and call 'ret', which will
    //   restore %eip to its pre-fault value.
    //
    //   In the case of a recursive fault on the exception stack,
    //   note that the word we're pushing now will fit in the
    //   blank word that the kernel reserved for us.
    //
    // Throughout the remaining code, think carefully about what
    // registers are available for intermediate calculations.  You
    // may find that you have to rearrange your code in non-obvious
    // ways as registers become unavailable as scratch space.
```

也就是说，需要把trap-time的EIP压入到trap-time的栈，切换到trap-time的栈再从中弹出EIP，从而完全恢复当时的状态。之前所提到的在用户异常栈中留出一个字的空白，就是为了这个EIP。如果触发异常时不在异常栈，这个EIP则会被压入普通栈。

```
    movl 0x30(%esp), %eax
    subl $4, %eax
    movl %eax, 0x30(%esp)
    movl 0x28(%esp), %ebx
    movl %ebx, (%eax)
```

这里的`0x30(%esp)`代表的是trap frame中的`utf_esp`，`0x28(%esp)`代表的是`utf_eip`。我们把`utf_esp`减去`4`，找到留出的那个空白以放入`utf_eip`。

```
    // Restore the trap-time registers.  After you do this, you
    // can no longer modify any general-purpose registers.
    // LAB 4: Your code here.
    addl $8, %esp
    popal
```

跳过`uint32_t utf_fault_va` `uint32_t utf_err`，恢复`struct PushRegs utf_regs`中通用寄存器的内容，此后便不能使用通用寄存器了。

```
    // Restore eflags from the stack.  After you do this, you can
    // no longer use arithmetic operations or anything else that
    // modifies eflags.
    // LAB 4: Your code here.
    addl $4, %esp
    popfl
```

跳过`uintptr_t utf_eip`，恢复`uint32_t utf_eflags`，此后便不能使用算术运算等改变EFLAGS的操作了。

```
    // Switch back to the adjusted trap-time stack.
    // LAB 4: Your code here.
    popl %esp
```

此时恰好指向`uintptr_t utf_esp`，弹出即跳转到trap-time的栈，之后栈指针指向的是所需要恢复的EIP。

```
    // Return to re-execute the instruction that faulted.
    // LAB 4: Your code here.
    ret
```

最后用一个`ret`来同时使ESP加上`4`，使EIP恢复到当时，完成所有的恢复。可见，整个流程的设计十分巧妙，利用了各结构体、栈帧的结构特点，实现了完美的上下文恢复。`struct UTrapframe`中的成员顺序并非随意设定，而是根据恢复的顺序来特别指定的。

> Exercise 11. 完成`lib/pgfault.c`中的`set_pgfault_handler()`。

这是用户级缺页异常处理机制的最后一部分。所需要做的其实就是设置`_pgfault_handler`这个全局变量以供`_pgfault_upcall`调用，在第一次设置时还需要分配一页内存给用户异常栈和注册处理函数。

```
//
// Set the page fault handler function.
// If there isn't one yet, _pgfault_handler will be 0.
// The first time we register a handler, we need to
// allocate an exception stack (one page of memory with its top
// at UXSTACKTOP), and tell the kernel to call the assembly-language
// _pgfault_upcall routine when a page fault occurs.
//
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(0, (void*)(UXSTACKTOP - PGSIZE),
                PTE_U | PTE_W | PTE_P);
        if (r < 0)
            panic("set_pgfault_handler: %e!\n", r);
        r = sys_env_set_pgfault_upcall(0, _pgfault_upcall);
        if (r < 0)
            panic("set_pgfault_handler: %e!\n", r);
    }

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

有了各类系统调用的基础，我们就可以在用户空间开始实现COW的`fork()`了。与之前的`dumbfork()`不同的是，COW机制下一开始只会复制页的映射，直到父子环境其中的一个试图进行写操作时才会真正复制页的内容。

整个控制流大概如下：

- 父环境将`pgfault()`设置为缺页异常的处理函数
- 调用`sys_exofor()`创建空白的子环境
- 对于父环境中低于`UTOP`的所有可写的或者COW的页，父环境调用`duppage`，它会把这些页在子环境中映射为COW，在父环境中也重新映射为COW。具体而言它会将两者中的PTE都设为不可写，并标记上`PTE_COW`位以区分于纯粹只读的页。不同的是异常栈，子环境需要直接单独分配新的页。当然，其它的页也是需要被映射的。

  这里有两个需要思考的问题：

  - 为什么先标记子环境中的映射再标记父环境中的映射？

    调用`duppage()`时，父环境必须对运行栈进行写操作。如果先标记了父环境中的映射，在处理到栈时会发生这样的事：父环境中的栈被标为COW，调用`duppage()`时发生了写操作，栈中的内容被复制到新的页中，并且此时父环境中的COW标记不复存在，再去标记子环境。之后父环境对这部分栈的写操作子环境都是可见的，显然不符合COW的初衷。

  - 为什么异常栈需要直接分配？

    处理缺页异常需要异常栈。如果异常栈也被设为COW，它本身的异常处理就无法进行了。

- 父环境设置子环境的缺页异常处理函数
- 子环境被标记为可运行

每当有一个环境试图写一个COW的页时，就会触发异常，这里的控制流大概如下：

- 内核将异常推给`_pgfault_upcall`，它会调用`pgfault()`
- `pgfault()`检查触发条件和标记位
- 分配新的一页，将原有的内容复制过去并建立映射，此时是权限是可写的

`lib/fork.c`需要查询环境的页目录和页表中的项，这是通过`UVPT`来实现的。例如，`uvpd[PDX(va)]`就恰好是`va`在页目录中的项，`uvpt[PGNUM(va)]`就恰好是`va`在页表中的项。还记得我们在设置内存布局的时候，页目录中的`0x3BD`这一项指向的是页目录本身，而`uvpt`的虚拟地址恰好为`0x3BD << 22`。这样，`uvpt[PGNUM(va)]`的地址就是`(0x3BD << 22) | (PDX(va) << 12) | (PTX(va) << 2)`，通过两级页表的查找，先找到页目录本身，再找到页目录项，再找到页表项。`uvpd[PDX(va)]`也是同样的道理。

> Exercise 12. 实现`lib/fork.c`中的`fork` `duppage` `pgfault`。通过`forktree`程序来测试。

按照前面已经说明的信息，在`fork()`中需要做的就是设置缺页异常处理函数，创建子环境，然后逐一处理映射。还有两个小问题，一是在子环境中重设`thisenv`这个变量，二是父环境要为子环境注册异常处理函数。

```
//
// User-level fork with copy-on-write.
// Set up our page fault handler appropriately.
// Create a child.
// Copy our address space and page fault handler setup to the child.
// Then mark the child as runnable and return.
//
// Returns: child's envid to the parent, 0 to the child, < 0 on error.
// It is also OK to panic on error.
//
// Hint:
//   Use uvpd, uvpt, and duppage.
//   Remember to fix "thisenv" in the child process.
//   Neither user exception stack should ever be marked copy-on-write,
//   so you must allocate a new page for the child's user exception stack.
//
envid_t
fork(void)
{
    // LAB 4: Your code here.
    int r;
    set_pgfault_handler(pgfault);
    envid_t envid = sys_exofork();
    if (envid < 0) panic("fork: %e!\n", envid);
    if (envid == 0)
    {
        thisenv = &envs[ENVX(sys_getenvid())];
        return 0;
    }
    uint32_t va;
    for (va = 0; va < USTACKTOP; va += PGSIZE)
        if ((uvpd[PDX(va)] & PTE_P) && (uvpt[PGNUM(va)] & PTE_P))
            duppage(envid, PGNUM(va));
    r = sys_page_alloc(envid, (void*)(UXSTACKTOP - PGSIZE), PTE_U | PTE_W | PTE_P);
    if (r < 0) panic("fork: %e!\n", r);
    extern void _pgfault_upcall(void);
    sys_env_set_pgfault_upcall(envid, _pgfault_upcall);
    r = sys_env_set_status(envid, ENV_RUNNABLE);
    if (r < 0) panic("fork: %e!\n", r);
    return envid;
}
```

`duppage()`中负责复制地址映射，根据情况来区分。如果原页面可写或者COW，则依次在子环境中和父环境中设为COW。否则权限是不变的。

```
//
// Map our virtual page pn (address pn*PGSIZE) into the target envid
// at the same virtual address.  If the page is writable or copy-on-write,
// the new mapping must be created copy-on-write, and then our mapping must be
// marked copy-on-write as well.  (Exercise: Why do we need to mark ours
// copy-on-write again if it was already copy-on-write at the beginning of
// this function?)
//
// Returns: 0 on success, < 0 on error.
// It is also OK to panic on error.
//
static int
duppage(envid_t envid, unsigned pn)
{
    int r;

    // LAB 4: Your code here.
    void *va = (void*)(pn * PGSIZE);
    int perm = uvpt[pn] & PTE_SYSCALL;
    if (perm & (PTE_W | PTE_COW))
    {
        perm &= ~PTE_W;
        perm |= PTE_COW;
    }
    r = sys_page_map(0, va, envid, va, perm);
    if (r < 0) panic("duppage: %e!\n");
    r = sys_page_map(0, va, 0, va, perm);
    if (r < 0) panic("duppage: %e!\n");
    return 0;
}
```

在`pgfault()`中，主要是检查好触发条件是不是我们所期望的（写一个COW的页），然后重新分配页面，复制内容，重设映射即可。

```
//
// Custom page fault handler - if faulting page is copy-on-write,
// map in our own private writable copy.
//
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 (!(err & FEC_WR) || !(uvpt[PGNUM(addr)] & PTE_COW))
        panic("pgfault: Not copy-on-write!\n" );

    // 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.
    addr = ROUNDDOWN(addr, PGSIZE);
    r = sys_page_alloc(0, (void*)PFTEMP, PTE_U | PTE_W | PTE_P);
    if (r < 0) panic("pgfault: %e!\n" );
    memcpy((void*)PFTEMP, addr, PGSIZE);
    r = sys_page_map(0, (void*)PFTEMP, 0, addr, PTE_U | PTE_W | PTE_P);
    if (r < 0) panic("pgfault: %e!\n" );
    r = sys_page_unmap(0, (void*)PFTEMP);
    if (r < 0) panic("pgfault: %e!\n" );
}
```

用`forktree`来测试，内容大概是一个环境`fork`形成一棵4层的二叉树，结果如下：

```
[00000000] new env 00001000
1000: I am ''
[00001000] new env 00001001
[00001000] new env 00001002
1001: I am '0'
[00001001] new env 00001003
[00001001] new env 00001004
1003: I am '00'
[00001003] new env 00001005
1005: I am '000'
[00001005] exiting gracefully
[00001005] free env 00001005
[00001000] exiting gracefully
[00001000] free env 00001000
1002: I am '1'
[00001002] new env 00002000
[00001002] new env 00002005
[00001003] new env 00001006
[00001003] exiting gracefully
[00001003] free env 00001003
2000: I am '10'
[00001001] exiting gracefully
[00001001] free env 00001001
[00002000] new env 00002001
[00001002] exiting gracefully
[00001002] free env 00001002
[00002000] new env 00002002
2001: I am '100'
[00002001] exiting gracefully
[00002001] free env 00002001
[00002000] exiting gracefully
[00002000] free env 00002000
2002: I am '101'
[00002002] exiting gracefully
[00002002] free env 00002002
1004: I am '01'
[00001004] new env 00003002
2005: I am '11'
[00002005] new env 00003000
[00002005] new env 00003001
1006: I am '001'
[00001006] exiting gracefully
[00001006] free env 00001006
3000: I am '110'
[00003000] exiting gracefully
[00003000] free env 00003000
[00001004] new env 00004000
[00002005] exiting gracefully
[00002005] free env 00002005
3001: I am '111'
[00003001] exiting gracefully
[00003001] free env 00003001
3002: I am '010'
```

```
[00003002] exiting gracefully
[00003002] free env 00003002
[00001004] exiting gracefully
[00001004] free env 00001004
4000: I am '011'
[00004000] exiting gracefully
[00004000] free env 00004000
No runnable environments in the system!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
```

Part C: Preemptive Multitasking and Inter-Pr ocess communication (IPC)

最后，我们将修改内核以支持抢占式调度以及进程间的通信机制。

在现实中存在一些不完善的程序会掉入死循环，或者因其他方式而不会主动让出CPU。内核需要通过抢占来处理这种情况，强行获取CPU的控制。这是基于时钟中断实现的。

外部的中断被称为IRQ (Interrupt ReQuest)，共有16种，JOS中映射到`IDT[IRQ_OFFSET]`至`IDT[IRQ_OFFSET + 15]`，其中`IRQ_OFFSET = 32`。简化起见，在内核中我们禁止外部中断，而在用户模式下必须打开中断，这是通过设置`%eflags`中的`FL_IF`来决定是否允许中断的。其实，在Bootloader的一开始我们就已经关闭了中断。

> Exercise 13. 修改`kern/trapentry.S`和`kern/trap.c`来初始化IDT中的IRQ处理函数，修改`env_alloc()`来保证用户环境中的中断是打开的。硬件中断时处理器不会压入错误码。完成后，如果允许一个时间足够长的程序，将可以看到硬件中断的trap frame，但由于目前内核还没有正确的处理函数，当前环境会被终止。

和之前异常的trap一样，我们在`kern/trapentry.S`中添加：

```
TRAPHANDLER_NOEC(t_irq0, IRQ_OFFSET + 0)
TRAPHANDLER_NOEC(t_irq1, IRQ_OFFSET + 1)
TRAPHANDLER_NOEC(t_irq2, IRQ_OFFSET + 2)
TRAPHANDLER_NOEC(t_irq3, IRQ_OFFSET + 3)
TRAPHANDLER_NOEC(t_irq4, IRQ_OFFSET + 4)
TRAPHANDLER_NOEC(t_irq5, IRQ_OFFSET + 5)
TRAPHANDLER_NOEC(t_irq6, IRQ_OFFSET + 6)
TRAPHANDLER_NOEC(t_irq7, IRQ_OFFSET + 7)
TRAPHANDLER_NOEC(t_irq8, IRQ_OFFSET + 8)
TRAPHANDLER_NOEC(t_irq9, IRQ_OFFSET + 9)
TRAPHANDLER_NOEC(t_irq10, IRQ_OFFSET + 10)
TRAPHANDLER_NOEC(t_irq11, IRQ_OFFSET + 11)
TRAPHANDLER_NOEC(t_irq12, IRQ_OFFSET + 12)
TRAPHANDLER_NOEC(t_irq13, IRQ_OFFSET + 13)
TRAPHANDLER_NOEC(t_irq14, IRQ_OFFSET + 14)
TRAPHANDLER_NOEC(t_irq15, IRQ_OFFSET + 15)
```

在`kern/trap.c`中的`trap_init()`中添加：

```
    extern void t_irq0();
    extern void t_irq1();
    extern void t_irq2();
    extern void t_irq3();
    extern void t_irq4();
    extern void t_irq5();
    extern void t_irq6();
    extern void t_irq7();
    extern void t_irq8();
    extern void t_irq9();
    extern void t_irq10();
    extern void t_irq11();
    extern void t_irq12();
    extern void t_irq13();
    extern void t_irq14();
    extern void t_irq15();

    SETGATE(idt[IRQ_OFFSET + 0], 0, GD_KT, t_irq0, 0);
    SETGATE(idt[IRQ_OFFSET + 1], 0, GD_KT, t_irq1, 0);
    SETGATE(idt[IRQ_OFFSET + 2], 0, GD_KT, t_irq2, 0);
    SETGATE(idt[IRQ_OFFSET + 3], 0, GD_KT, t_irq3, 0);
    SETGATE(idt[IRQ_OFFSET + 4], 0, GD_KT, t_irq4, 0);
    SETGATE(idt[IRQ_OFFSET + 5], 0, GD_KT, t_irq5, 0);
    SETGATE(idt[IRQ_OFFSET + 6], 0, GD_KT, t_irq6, 0);
    SETGATE(idt[IRQ_OFFSET + 7], 0, GD_KT, t_irq7, 0);
    SETGATE(idt[IRQ_OFFSET + 8], 0, GD_KT, t_irq8, 0);
    SETGATE(idt[IRQ_OFFSET + 9], 0, GD_KT, t_irq9, 0);
    SETGATE(idt[IRQ_OFFSET + 10], 0, GD_KT, t_irq10, 0);
    SETGATE(idt[IRQ_OFFSET + 11], 0, GD_KT, t_irq11, 0);
    SETGATE(idt[IRQ_OFFSET + 12], 0, GD_KT, t_irq12, 0);
    SETGATE(idt[IRQ_OFFSET + 13], 0, GD_KT, t_irq13, 0);
    SETGATE(idt[IRQ_OFFSET + 14], 0, GD_KT, t_irq14, 0);
    SETGATE(idt[IRQ_OFFSET + 15], 0, GD_KT, t_irq15, 0);
```

这样就完成了一个初始的中断处理。接下来我们再详细看看时钟中断。在`i386_init`调用`lapic_init`和`pic_init`中，已经设置了时钟和中断控制器来产生时钟中断。比如`lapic_init()`中有代码：

```
    // The timer repeatedly counts down at bus frequency
    // from lapic[TICR] and then issues an interrupt.
    // If we cared more about precise timekeeping,
    // TICR would be calibrated using an external time source.
    lapicw(TDCR, X1);
    lapicw(TIMER, PERIODIC | (IRQ_OFFSET + IRQ_TIMER));
    lapicw(TICR, 10000000);
```

> Exercise 14. 修改内核的`trap_dispatch()`，处理时钟中断时调用`sched_yield()`来调度另一个环境。完成后可以用`user/spin`来测试。

在`trap_dispatch()`的`switch`中添加项：

```
    // Handle clock interrupts. Don't forget to acknowledge the
    // interrupt using lapic_eoi() before calling the scheduler!
    // LAB 4: Your code here.
    case IRQ_OFFSET + IRQ_TIMER:
        lapic_eoi();
        sched_yield();
        break;
```

测试结果如下：

```
[00000000] new env 00001000
I am the parent.  Forking the child...
[00001000] new env 00001001
I am the parent.  Running the child...
I am the child.  Spinning...
I am the parent.  Killing the child...
[00001000] destroying 00001001
[00001000] free env 00001001
[00001000] exiting gracefully
[00001000] free env 00001000
No runnable environments in the system!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
```

可见，尽管子环境陷入死循环，但时钟中断使得父环境可以被调度，从而能够终止子环境。我们甚至可以用多个CPU来测试，也可得到正确结果。

进程间通信有很多种模型可以实现。在JOS中我们将实现一个简单的IPC (Inter-Process Communication)机制。在系统调用级将会有两个函数`sys_ipc_recv` `sys_ipc_try_send`，在用户库级将会有两个包装函数`ipc_recv` `ipc_send`。用户环境之间传递的消息由两部分组成：一个32位的字和一个可选的页面映射。后者可以传递更多的信息，实际上也是一种内存共享。

接收消息时，环境调用`sys_ipc_recv`，它会阻塞当前环境，直到消息被真正接收才能继续运行。当环境等待接收时，任何一个环境都可以向它发送。也就是说，不需要检查环境的权限。发送消息时，环境调用`sys_ipc_try_send`，既需要指定所发消息也需要指定接收的目标环境。如果目标环境确实在等待接收，那么就可以成功发送，返回值为`0`，否则返回`-E_IPC_NOT_RECV`。包装函数`ipc_recv` `ipc_send`会处理好调用和返回的值。后者会循环调用`sys_ipc_try_send`直到成功发送。

此外，调用两个系统调用时还可以指定一个`UTOP`以下的虚拟地址，表示需要传递的页面映射的地址，以及一个权限位。如果成功传递，这个页就可以被两个环境所共享。

> Exercise 15. 实现`kern/syscall.c`中的`sys_ipc_recv` `sys_ipc_try_send`。调用`envid2env`时，无需检查权限。然后实现`lib/ipc.c`中的`ipc_recv`和`ipc_send`。用`user/pingpong`和`user/primes`来测试IPC。

在`sys_ipc_try_send`中，我们要找到目标环境，检查其是否正在等待接收。如果不是，则返回一个错误信息。然后设置好目标环境中的域，特别是其中的返回值和环境状态。可能的话需要建立映射。注意，这里不能使用先前实现的`sys_page_map`函数来建立映射，尽管它们看起来功能是一样的。原因是这个函数会检查环境是否具有权限，但是IPC并不要求环境之间有父子关系或者相同。此外，由于这个系统调用返回非零值并不是不可接受的，在`lib/syscall.c`中要把`check`这个参数设为`0`，即`syscall(SYS_ipc_try_send, 0, envid, value, (uint32_t) srcva, perm, 0)`。具体的实现如下：

```
// Try to send 'value' to the target env 'envid'.
// If srcva < UTOP, then also send page currently mapped at 'srcva',
// so that receiver gets a duplicate mapping of the same page.
//
// The send fails with a return value of -E_IPC_NOT_RECV if the
// target is not blocked, waiting for an IPC.
//
// The send also can fail for the other reasons listed below.
//
// Otherwise, the send succeeds, and the target's ipc fields are
// updated as follows:
//    env_ipc_recving is set to 0 to block future sends;
//    env_ipc_from is set to the sending envid;
//    env_ipc_value is set to the 'value' parameter;
//    env_ipc_perm is set to 'perm' if a page was transferred, 0 otherwise.
// The target environment is marked runnable again, returning 0
// from the paused sys_ipc_recv system call.  (Hint: does the
// sys_ipc_recv function ever actually return?)
//
// If the sender wants to send a page but the receiver isn't asking for one,
// then no page mapping is transferred, but no error occurs.
// The ipc only happens when no errors occur.
//
// Returns 0 on success, < 0 on error.
// Errors are:
//  -E_BAD_ENV if environment envid doesn't currently exist.
//      (No need to check permissions.)
//  -E_IPC_NOT_RECV if envid is not currently blocked in sys_ipc_recv,
//      or another environment managed to send first.
//  -E_INVAL if srcva < UTOP but srcva is not page-aligned.
//  -E_INVAL if srcva < UTOP and perm is inappropriate
//      (see sys_page_alloc).
//  -E_INVAL if srcva < UTOP but srcva is not mapped in the caller's
//      address space.
//  -E_INVAL if (perm & PTE_W), but srcva is read-only in the
//      current environment's address space.
//  -E_NO_MEM if there's not enough memory to map srcva in envid's
//      address space.
static int
sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned perm)
{
    // LAB 4: Your code here.
    struct Env *e;
    int r;
    r = envid2env(envid, &e, 0);
    if (r < 0) return r;
    if (!(e->env_ipc_recving) || e->env_ipc_from)
    {
        return -E_IPC_NOT_RECV;
    }
    if (e->env_ipc_dstva < (void*)UTOP && srcva < (void*)UTOP)
    {

        if (srcva != ROUNDDOWN(srcva, PGSIZE)) return -E_INVAL;
```

```
        pte_t *ptep = NULL;
        struct PageInfo *pp = page_lookup(curenv->env_pgdir, srcva, &ptep);
        if (!pp) return -E_INVAL;
        if (!(perm & PTE_U) || !(perm & PTE_P) || perm & ~PTE_SYSCALL) return -
E_INVAL;
        if (perm & PTE_W && !(*ptep & PTE_W)) return -E_INVAL;
        r = page_insert(e->env_pgdir, pp, e->env_ipc_dstva, perm);
        if (r < 0) return r;
        e->env_ipc_perm = perm;
    }
    else
        e->env_ipc_perm = 0;
    e->env_ipc_recving = 0;
    e->env_ipc_from = curenv->env_id;
    e->env_ipc_value = value;
    e->env_tf.tf_regs.reg_eax = 0;
    e->env_status = ENV_RUNNABLE;
    return 0;
}
```

在`sys_ipc_recv`中，事情要简单一些，主要是设置当前环境中的一些域，然后阻塞调度其它进程即可。实际的信息传递是在发送的过程中完成的，由发送者把这些信息写入。

```
// Block until a value is ready.  Record that you want to receive
// using the env_ipc_recving and env_ipc_dstva fields of struct Env,
// mark yourself not runnable, and then give up the CPU.
//
// If 'dstva' is < UTOP, then you are willing to receive a page of data.
// 'dstva' is the virtual address at which the sent page should be mapped.
//
// This function only returns on error, but the system call will eventually
// return 0 on success.
// Return < 0 on error.  Errors are:
//  -E_INVAL if dstva < UTOP but dstva is not page-aligned.
static int
sys_ipc_recv(void *dstva)
{
    // LAB 4: Your code here.
    if (dstva < (void*)UTOP && dstva != ROUNDDOWN(dstva, PGSIZE))
        return -E_INVAL;
    curenv->env_ipc_recving = 1;
    curenv->env_ipc_from = 0;
    curenv->env_ipc_dstva = dstva;
    curenv->env_status = ENV_NOT_RUNNABLE;
    sched_yield();
}
```

再来看两个包装函数。如果参数中的虚拟地址为空，进行系统调用时用`UTOP`这个地址就可以了，因为后续的检查会发现这个地址无效（但是直接用`NULL`是不行的）。在`ipc_send`中，需要通过一个循环来不断尝试发送。为了提高系统效率，每次发送失败之后还需要让出CPU，不能总是忙等待。

```
// Receive a value via IPC and return it.
// If 'pg' is nonnull, then any page sent by the sender will be mapped at
//  that address.
// If 'from_env_store' is nonnull, then store the IPC sender's envid in
//  *from_env_store.
// If 'perm_store' is nonnull, then store the IPC sender's page permission
//  in *perm_store (this is nonzero iff a page was successfully
//  transferred to 'pg').
// If the system call fails, then store 0 in *fromenv and *perm (if
//  they're nonnull) and return the error.
// Otherwise, return the value sent by the sender
//
// Hint:
//   Use 'thisenv' to discover the value and who sent it.
//   If 'pg' is null, pass sys_ipc_recv a value that it will understand
//   as meaning "no page".  (Zero is not the right value, since that's
//   a perfectly valid place to map a page.)
int32_t
ipc_recv(envid_t *from_env_store, void *pg, int *perm_store)
{
    // LAB 4: Your code here.
    int r;
    if (pg)
        r = sys_ipc_recv(pg);
    else
        r = sys_ipc_recv((void*)UTOP);
    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;
    if (r < 0) return r;
    return thisenv->env_ipc_value;
}

// Send 'val' (and 'pg' with 'perm', if 'pg' is nonnull) to 'toenv'.
// This function keeps trying until it succeeds.
// It should panic() on any error other than -E_IPC_NOT_RECV.
//
// Hint:
//   Use sys_yield() to be CPU-friendly.
//   If 'pg' is null, pass sys_ipc_try_send a value that it will understand
//   as meaning "no page".  (Zero is not the right value.)
void
ipc_send(envid_t to_env, uint32_t val, void *pg, int perm)
{
    // LAB 4: Your code here.
    if (!pg) pg = (void*)UTOP;
    while (1)
    {
        int r;
        r = sys_ipc_try_send(to_env, val, pg, perm);
        if (r < 0 && r != -E_IPC_NOT_RECV)
            panic("ipc_send: %e!\n", r);
```

```
        sys_yield();
        if (r == 0) break;
    }
}
```

总结一下JOS中的IPC机制，假设A要向B发送一个消息，包括一个页面中的字符串`str`及其长度`len`。过程如下：

- A准备好`len` `str`
- A调用库函数`ipc_send()`，参数包括`len` `str`所在的页地址和权限位
- `ipc_send()`产生系统调用`sys_ipc_try_send()`
- 然而B还没有准备接收，系统调用发现这个情况后返回`-E_IPC_NOT_RECV`，A会让出CPU并且在之后循环尝试发送
- B准备接收了，调用`ipc_recv()`
- `ipc_recv()`产生系统调用`sys_ipc_recv()`
- 系统调用会设置B环境中的一些变量，最后会设B为`ENV_NOT_RUNNABLE`，让出CPU
- `ipc_send()`再次产生系统调用`sys_ipc_try_send()`，发现B准备接收
- 填写B环境中的一些变量，设置页面映射，实现`len` `str`传递到B环境中，最后B的`%eax`设为`0`，状态设为`ENV_RUNNABLE`
- `sys_ipc_try_send()`成功返回，A也可以继续运行
- `sys_ipc_recv()`被调度，然后返回，B继续运行

初次完成时，测试发现过不了`primes`这个测试（事实上前面的`pingpong`在多核也过不了）。并且如果CPU只有1个不会有问题，超过1个之后就开始出错。大概的输出情况如下：

```
[00000000] new env 00001000
[00001000] new env 00001001
CPU 0: 2 [00001001] new env 00001002
CPU 3: 3 [00001002] new env 00001003
[00000000] user panic in <unknown> at lib/syscall.c:35: syscall 12 returned 12 (> 0)
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
TRAP frame at 0xf02b407c from CPU 2
  edi  0x00000000
  esi  0x00801423
  ebp  0xeebfdf20
  oesp 0xeffdffdc
  ebx  0xeebfdf34
  edx  0xeebfddd8
  ecx  0x00000001
  eax  0x00000001
  es   0x----0023
  ds   0x----0023
  trap 0x00000003 Breakpoint
  err  0x00000000
  eip  0x00800194
  cs   0x----001b
  flag 0x00000286
  esp  0xeebfdf18
  ss   0x----0023
```

调试这个bug花了不少时间。从输出信息来看，系统调用`sys_ipc_recv`是直接返回了（调用时`%eax`为调用号`12`，返回值也是它）。按理说，`%eax`应当由发送环境的系统调用进行设置，最终会返回`0`。最终查明的原因是在写调度函数`sched_yield()`时出现的问题。某个环境调用了`sched_yield()`，会检查是否有其它环境可以运行，如果都没有的话则运行原来的环境。当时代码最后是这样的：

```
    if (cur)
        env_run(cur);
```

但是，这里就没有考虑到调用的环境本身是否可以运行，当时没有想到会有`sys_ipc_recv`这样的系统调用，将自己设为不可运行再进行调度，于是默认了原调用环境是可以运行的。这样在系统调用的最后：

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

在多核情况下，很可能这个接收环境调用调度函数后发现没别的环境可运行（都在其它CPU上运行着），于是就继续运行了当前的接收环境，并没有起到阻塞的作用，进而可以观察到上面奇怪的现象。因此，在`sched_yield()`中必须加上条件判断：

```
    if (cur && cur->env_status == ENV_RUNNABLE)
        env_run(cur);
```