Lab 4: Preemptive Multitasking

Introduction

在这个lab中，你将会实现在多个同时激活的*用户模式*环境中的实现*preemptive multitasking(抢占式多任务处理)*。

- Part A

  向`JOS`中加入*多处理器支持*，实现*循环调度*，加入基本的*环境管理系统调用*（也就是创建/销毁环境，分配、映射内存）。

- Part B

  你将会实现一个*类Unix的*`fork()`，该功能允许一个*用户模式环境*创建自身的副本。

- Part C

  将加入对*inter-process communication(IPC)*的支持，允许不同*用户模式环境*互相之间显示地*通信*和*同步*。也会加入对硬件*时钟中断*和*抢占式*的支持。

Getting Started

```
cd ~/6.828/lab
add git
git pull      # 如果这里出错，要导入环境变量 export GIT_SSL_NO_VERIFY=1
#Already up-to-date
git checkout -b lab4 origin/lab4 # 若出现conflict，需删掉conflict文件中的`═══HEAD═══lab=`
                                 # 然后再 git add . & git commit -am "..."
                                 # & git merge lab3
git merge lab3
# Already up to date.
```

Lab 4 包含了一些新的源文件，在开始之前应该先浏览：

```
kern/cpu.h          Kernel-private definitions for multiprocessor
support

kern/mpconfig.c     Code to read the multiprocessor configuration

kern/lapic.c        Kernel code driving the local APIC unit in each
processor

kern/mpentry.S      Assembly-language entry code for non-boot CPUs

kern/spinlock.h     Kernel-private definitions for spin locks,
including the big kernel
                    lock
kern/spinlock.c     Kernel code implementing spin locks

kern/sched.c        Code skeleton of the scheduler that you are about
to implement
```

Lab Requirements

这个lab被分为三个部分，A，B，C。每个部分有一周的时间。

As before, you will need to do all of the regular exercises described in the lab and at least one challenge problem. (You do not need to do one challenge problem per part, just one for the whole lab.) Additionally, you will need to write up a brief description of the challenge problem that you implemented. If you implement more than one challenge problem, you only need to describe one of them in the write-up, though of course you are welcome to do more. Place the write-up in a file called answers-lab4.txt in the top level of your lab directory before handing in your work.

Part A: Multiprocessor Support and Cooperative Multitasking

在这个lab的第一部分，你将扩展 `JOS` 使其在一个多处理器系统上运行，然后实现一些新的 `JOS` kernel *系统调用* ，来允许 *用户环境* 创建额外新的环境；你将会实现 *协作循环调度* ，当当前环境放弃或退出CPU时，允许kernel切换到另一个环境。之后，在Part C，你将实现 *抢占式调度* ，允许kernel在一段时间之后从一个环境中重新拿回CPU的控制，尽管环境没有协作。

Multiprocessor Support

我们将使 `JOS` 支持 *对称多处理(symmetric multiprocessing, SMP)*，一个对称多处理模型中的所有CPUs都有对系统资源平等的访问权利，如内存和输入输出。虽然在 *SMP* 中所有CPUs功能上都是相同的，但在开机过程它们被分为两种类型：

- the bootstrap processor(BSP)：负责初始化系统和启动操作系统
- the application processors(APs)：仅当操作系统打开并运行之后才能被 *BSP* 激活

哪个处理器是 *BSP* 由硬件和BIOS决定。直到现在，所有存在的代码都是运行在 *BSP* 上。

在 *SMP* 系统中，每个CPU都有一个陪伴的 *local APIC(LAPIC) unit*，LAPIC单元负责在系统中传递中断，提供其连接的带有独特标识符的CPU。在这个lab，我们将使用LAPIC单元下面这几个基本的功能：

- 读取 *LAPIC identifier(APIC ID)*。告诉我们的代码当前正在哪个CPU上运行。(see `cpunum()`)
- 发送 *STARTUP* 处理器间中断(interprocessor interrupt, IPI)。从BSP发送到APs以启动其它CPUs。(see `lapic_startup()`)
- 在Part C，我们编程LAPIC的内置计时器来触发 *时钟中断* 来支持 *抢占式的多任务处理*。(see `apic_init()`)

一个进程使用 *memory-mapped I/O(MMIO)* 来访问它的LAPIC。在MMIO中，*物理内存* 的一部分被硬连线到一些I/O设备的寄存器，所以一般用于访问内存的相同 `load/store` 指令可以被用于访问 *设备寄存器*。

在物理内存 *0xA0000* 有个IO洞（我们用它来写VGA显示缓冲区）。*LAPIC* 位于物理内存 *0xFE000000* 的洞（32MB short of 4GB），所以这个地址 *过于高而不能* 使用我们通常在KERNBASE的直接映射来访问。

`JOS` 虚拟内存映射在MMIOBASE留了一个4MB的gap，所以我们有像这样的地方来映射设备。因为之后的lab会引入更多的MMIO区域，你将会写一个简单的函数来从该区域 *分配空间* 并为其 *映射设备内存*。

> **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.
>
> `mmio_map_region()` in `kern/pmap.c`

```
// 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)
{
   static uintptr_t base;
   // WARNING: maybe here is wrong
   if(!base){
       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:
uintptr_t result;
   result = base;
   if(0 == size){
       cprintf("mmio_map_region size parameter is 0\n");
       return (void*)result;
   }
```

```
    size = ROUNDUP(size, PGSIZE);
    if(0 < size){
        if((base + size) ≥ MMIOLIM){
            panic("mmio_map_region out of MMIOLIM\n");
        }
           boot_map_region(kern_pgdir, base, size, pa, PTE_PCD |
PTE_PWT);
        base += size;
        cprintf("mmio_map_region alloc memory at %x, next base is
%x\n", result,
        return (void*)result;
    }
    panic("mmio_map_region's size < 0");

    return NULL;
    // panic("mmio_map_region not implemented");

}
```

Application Processor Bootstrap

在启动*APs*之前，*BSP*应该先收集*多处理器系统*的信息，比如CPUs的数量，他们的*APIC IDs*以及*LAPIC*单元的MMIO地址。在`kern/mpconfig.c`中的`mp_init()`函数通过读取存储于内存BIOS区域的(*MP*配置表)取回信息。

`kern/init.c`中的`boot_aps()`驱动*AP bootstrap process*。*APs*在实模式启动，和在`boot/boot.S`中启动的bootloader很相似，所以`boot_aps()`将*AP*的entry code（`kern/mpentry.S`）复制到一个内存位置，这个内存位置在实模式是可寻址的。和bootloader不一样的是，我们**可以控制AP从哪开始执行代码**；我们复制entry code到`0x7000(MPENTRY_PADDR)`，但是任何低于640KB未使用，页对齐的物理地址也会起作用。

然后，`boot_aps()`依次激活APs，通过向对应AP的LAPIC单元发送`STARTUP IPIs`，以及一个初始的`CS:IP`地址，这个地址是AP应该开始运行其entry code的地方（`MPENTRY_PADDR` in our case)。`kern/mpentry.S`中的entry code和`boot/boot.S`中的非常相似。

一些简短的设置之后，它将AP置于开启页映射的保护模式，然后调用C常规设置函数`mp_main()`(in `kern/init.c`)。`boot_aps()`在继续唤醒下一个AP之前，等待当前AP发送一个`CPU_STARTED`标志，这个标志位于`struct CpuInfo`的的`cpu_status`字段。

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).

`boot_aps()` in `kern/init.c`

```
// While boot_aps is booting a given CPU, it communicates the per-
core
// stack pointer that should be loaded by mpentry.S to that CPU in
// this variable.
void *mpentry_kstack;

// Start the non-boot (AP) processors.
static void
boot_aps(void)
{
  // 在 `mpentry.S` 中用.global 定义
   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);
  // 这里的代码应该就是 `mpentry.S` 中的汇编代码
  // 该代码中会调用 `mp_main()`
      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)
            ;
    }
}

struct CpuInfo {
      uint8_t cpu_id;                 // Local APIC ID; index into
cpus[] below
    volatile unsigned cpu_status;   // The status of the CPU
    struct Env *cpu_env;            // The currently-running
environment.
      struct Taskstate cpu_ts;        // Used by x86 to find stack
for interrupt
};
```

`mp_main()` in `kern/init.c`

```
// Setup code for APs
void
mp_main(void)
{
    // 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

    // 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:

    // Remove this after you finish Exercise 6
    for (;;);
}
```

`lapic_init()` in `kern/lapic.c`

```
void
lapic_init(void)
{
  if (!lapicaddr)
    return;

  // lapicaddr is the physical address of the LAPIC's 4K MMIO
  // region.  Map it in to virtual memory so we can access it.
// 所有的物理地址都要映射到虚拟地址才能访问
  lapic = mmio_map_region(lapicaddr, 4096);

  // Enable local APIC; set spurious interrupt vector.
  lapicw(SVR, ENABLE | (IRQ_OFFSET + IRQ_SPURIOUS));

  // 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);       // TDCR-timer divide configuration, X1-
divide counts by 1
  lapicw(TIMER, PERIODIC | (IRQ_OFFSET + IRQ_TIMER));     //
TIMER-local vector table 0
                                                        //
PERIODIC-periodic
  lapicw(TICR, 10000000);                                 // TICR-
timer initial count

  // Leave LINT0 of the BSP enabled so that it can get
  // interrupts from the 8259A chip.
```

```
  //
  // According to Intel MP Specification, the BIOS should
initialize
  // BSP's local APIC in Virtual Wire Mode, in which 8259A's
  // INTR is virtually connected to BSP's LINTIN0. In this mode,
  // we do not need to program the IOAPIC.

// 只保留bootcpu的LINT0字段
  if (thiscpu ≠ bootcpu)
    lapicw(LINT0, MASKED);

  // Disable NMI (LINT1) on all CPUs
  lapicw(LINT1, MASKED);

  // Disable performance counter overflow interrupts
  // on machines that provide that interrupt entry.
  if (((lapic[VER]>>16) & 0xFF) ⩾ 4)
    lapicw(PCINT, MASKED);

  // Map error interrupt to IRQ_ERROR.
  lapicw(ERROR, IRQ_OFFSET + IRQ_ERROR);

  // Clear error status register (requires back-to-back writes).
  lapicw(ESR, 0);
  lapicw(ESR, 0);

  // Ack any outstanding interrupts.
  lapicw(EOI, 0);

  // Send an Init Level De-Assert to synchronize arbitration ID's.
  lapicw(ICRHI, 0);
  lapicw(ICRLO, BCAST | INIT | LEVEL);
  while(lapic[ICRLO] & DELIVS)
    ;

  // Enable interrupts on the APIC (but not on the processor).
  lapicw(TPR, 0);
}
```

kern/mpentry.S

```
###################################################################
#
# entry point for APs
###################################################################
#

# Each non-boot CPU ("AP") is started up in response to a STARTUP
# IPI from the boot CPU.  Section B.4.2 of the Multi-Processor
# Specification says that the AP will start in real mode with
CS:IP
# set to XY00:0000, where XY is an 8-bit value sent with the
# STARTUP. Thus this code must start at a 4096-byte boundary.
#
# Because this code sets DS to zero, it must run from an address
in
# the low 2^16 bytes of physical memory.
#
# boot_aps() (in init.c) copies this code to MPENTRY_PADDR (which
# satisfies the above restrictions).  Then, for each AP, it stores
the
# address of the pre-allocated per-core stack in mpentry_kstack,
sends
# the STARTUP IPI, and waits for this code to acknowledge that it
has
# started (which happens in mp_main in init.c).
#
# This code is similar to boot/boot.S except that
#    - it does not need to enable A20
#    - it uses MPBOOTPHYS to calculate absolute addresses of its
#      symbols, rather than relying on the linker to fill them

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

.set PROT_MODE_CSEG, 0x8    # kernel code segment selector
.set PROT_MODE_DSEG, 0x10   # kernel data segment selector
```

```
.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

  # If mp_main returns (it shouldn't), loop.
spin:
  jmp     spin

# Bootstrap GDT
.p2align 2                # force 4 byte alignment

  SEG_NULL                # null seg
  SEG(STA_X|STA_R, 0x0, 0xffffffff)   # code seg
  SEG(STA_W, 0x0, 0xffffffff)     # data seg

gdtdesc:
  .word   0x17                # sizeof(gdt) - 1
  .long   MPBOOTPHYS(gdt)         # address gdt

.globl mpentry_end
mpentry_end:
  nop
```

summary

https://img2020.cnblogs.com/blog/1346871/202007/1346871-20200713071810987-474318476.png

在 `kern/init.c` 中 `i386_init()` 中的调用顺序为 `cons_init()`，`mem_init()`，`env_init()`，`trap_init()`，`mp_init()`，`lapic_init()`，`pic_init()`。

- 首先调用 `mp_init()`

  `mp_init()` 用于获取*MP配置表*的信息
- `lapic_init()`

  用于初始化 `local APIC register`，函数中会调用 `mmio_map_region()`。
- 最后调用 `boot_aps()`

Per-CPU State and Initialization

当实现一个 *多处理器操作系统* 时，区分每个CPU状态（对每个处理器都是 *私有的* ）和全局的状态（在整个系统分享）很重要。

`kern/cpu.h` 定义了大多数的 `per-CPU state` ，包括 `struct CpuInfo` ，用于存储per-CPU变量。 `cpunum()` 总是返回调用它的CPU的ID，可用于下标访问 `cpus` 这样的数组。或者，宏 `thiscpu` 是对于当前CPU's `struct CpuInfo` 的简单表示。

下面是你应该注意的 `per-CPU state`

- **Per-CPU kernel stack**

  因为多个CPU能同时trap into内核，对于每个处理器我们需要一个分隔的内核栈来阻止他们影响各自的执行。数组 `percpu_kstacks[NCPU][KSTKSIZE]` 为NCPU的内核堆栈保留空间。

  在lab2中，将物理地址 `bootstack` 映射为BSP的内核栈， `bootstack` 就位于 `KSTACKTOP` 之下。相似的，在这个lab，你将映射每个CPU内核栈到该区域，并使用 *保护页* 作为作为它们之间的缓冲区。CPU0的栈仍将从KSTACKTOP向下增长；CPU1的栈将从CPU0的栈底往下KSTKGAP个字节开始，以此类推。 `inc/memlayout.h` 展示了映射分布。

- **Per-CPU TSS and TSS descriptor**

  Per-CPU task state segment(TSS)也被需要为了指定每个CPU内核栈的位置。CPU*i*的TSS被存储于 `cpus[i].cpu_ts` ，对应的TSS描述符定义于GDT条目 `get[(GD_TSS0 >> 3) + i]` 。定义于 `kern/trap.c` 中的全局 `ts` 变量不再有用。

- **Per-CPU current environment pointer**

  因为每个CPU能同时运行不同用户进程，我们重新定义标志 `curenv` 指向 `cpus[cpunum()]` 。 `cpu_env(or thiscpu->cpu_env)` 指向在当前CPU上当前正在执行的环境。

- **Per-CPU system registers**

  所有寄存器，包括系统寄存器，对CPU都是私有的 。因此，初始化寄存器的指令，如 `lcr3()` ， `ltr()` ， `lgdt()` ， `lidt()` 等，必须在每个CPU执行一次。函数 `env_init_percpu()` 和 `trap_init_percpu()` 被定义用于该目的。

  In addition to this, if you have added any extra per-CPU state or performed any additional CPUspecific initialization (by say, setting new bits in the CPU registers) in your solutions to challenge problems in earlier labs, be sure to replicate them on each CPU here!

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()`.

`mem_init_mp()` in `kern/pmap.c`

```
// 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:
```

```
  for(uint8_t i = 0; i < NCPU; ++i){
uintptr_t va = KSTACKTOP - i * (KSTKSIZE + KSTKGAP) - KSTKSIZE;
size_t size = KSTKSIZE;
    boot_map_region(kern_pgdir, va, size,
PADDR(percpu_kstacks[i]), PTE_P | 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.)

`trap_init_percpu()` in `kern/trap.c`

```
// 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()
```

```
	//   - Initialize cpu_ts.ts_iomb to prevent unauthorized
environments
	//     from doing IO (0 is not the correct value!)
	//
	// 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:
// 注意这里的地址需要再加上KSTKSIZE，需要根据 `mem_init_mp()` 的映射规则
	thiscpu->cpu_ts.ts_esp0 = (uintptr_t)percpu_kstacks[thiscpu-
>cpu_id] + KSTKSIZE;
	thiscpu->cpu_ts.ts_ss0 = GD_KD;
	thiscpu->cpu_ts.ts_iomb = sizeof(struct Taskstate);

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

	// 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);

	// 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;

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

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

```
}
```

Locking

我们当前代码在初始化完 `mp_main()` 中的 `AP` 后 *旋转* 。

在 `AP` 进一步讨论之前，我们首先需要解决多个CPUs同时运行内核代码时的 *竞争条件* 。解决这个问题最简单的方式时使用一个 ***big kernel lock*** 。这个big kernel lock是单个全局锁，任何时候当环境进入 *内核模式* ，这个锁被触发；当环境返回 *用户模式* ，这个锁被释放。在该模型下，*用户模式* 的环境能在任何可获得的CPUs上正确运行，但是 *内核模式* 中最多只能有一个环境；任何别的想进入 *内核模式* 的环境都会被强制等待。

`kern/spinlock.h` 声明了 *big kernel lock* ，叫做 `kernel_lock` 。它也提供了 `lock_kernel()` 和 `unlock_kernel()` ，用于获取和释放锁的快捷操作。你应该将 *big kernel lock* 应用在四个位置：

- `i386_init()` 中，在BSP唤醒其它CPUs之前 *获取* 锁
- `mp_main()` 中，初始化完AP之后 *获取* 锁，然后调用 `sched_yield()` 在该AP上开始运行环境
- `trap()` 中，当从 *用户模式* trap时， *获取* 锁。**检查 `tf_cs` 的低位**来查看trap发生在 *用户模式* 还是 *内核模式* 。
- `env_run()` 中，在切换到 *用户模式* 之前 *释放* 锁。不要做这个操作过早或过迟，否则，你将体验到 *竞争* 或者 *死锁* 。

> Exercise 5.
>
> Apply the big kernel lock as described above, by calling `lock_kernel()` and `unlock_kernel()` at the proper locations.

Round-Robin Scheduling

这个lab的下一个任务就是修改 `JOS` 内核，使得其可以在多个环境之间以 *轮转方式* 切换。 `JOS` 中的循环调度如下所示：

- `kern/sched.c` 中的 `sched_yield()` 负责选择一个新环境来运行

  它从上一个运行环境的后面，以循环的方式在 `envs[]` 数组中顺序查找，选择第一个状态为ENV_RUNNABLE(`inc/env.h`)的环境，然后调用 `env_run()` 来跳入那个环境。
- `sched_yield()` 禁止同一时间在两个CPU上运行同一个环境

  它能告知一个环境正在某个CPU上运行，因为该环境的状态是ENV_RUNNING。

- 我们已经实现了一个新的系统调用，`sys_field()`

  用户环境可以调用来唤醒内核的`sched_yield()`函数，因此自愿放弃CPU给一个不同的环境。

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.

`sched_yield()` in `kern/sched.c`

```
// 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.
  idle = thiscpu->cpu_env;
int cur_env_id = NULL == idle ? -1 : ENVX(idle->env_id);
int i;
  for(i = cur_env_id; i < NENV; ++i){
    if(ENV_RUNNABLE == envs[i].env_status){
      env_run(&envs[i]);
    }
  }
  for(i = 0; i < cur_env_id; ++i){
    if(ENV_RUNNABLE == envs[i].env_status){
      env_run(&envs[i]);
    }
  }
  if(NULL != idle && ENV_RUNNING == idle->env_status){
    env_run(idle);
  }
  // sched_halt never returns
```

```
  sched_halt();
}
```

`trap_dispatch()` in `kern/trap.c`

```
// this file should be changed, but kern/syscall.c
// case SYS_yield:
//      sched_yield();
//      return;
```

`syscall()` in `kern/syscall.c`

```
// Dispatches to the correct kernel function, passing the
arguments.
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:
     // user_mem_assert(thisenv, (const void*)a1, a2, PTE_U);
     sys_cputs((const 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;
    default:
        return -E_INVAL;
```

```
  }
}
```

`i386_init()` in `kern/init.c`

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

Output:

```
SMP: CPU 0 found 1 CPU(s)
mmio_map_region alloc memory at ef803000, next base is ef804000
enabled interrupts: 1 2
[00000000] new env 00001000
[00000000] new env 00001001
[00000000] new env 00001002
[00000000] new env 00001003
Hello, I am environment 00001000.
Back in environment 00001000, iteration 0.
Back in environment 00001000, iteration 1.
Back in environment 00001000, iteration 2.
Back in environment 00001000, iteration 3.
Back in environment 00001000, iteration 4.
All done in environment 00001000.
[00001000] exiting gracefully
[00001000] free env 00001000
Hello, I am environment 00001001.
Back in environment 00001001, iteration 0.
Back in environment 00001001, iteration 1.
Back in environment 00001001, iteration 2.
```

```
Back in environment 00001001, iteration 3.
Back in environment 00001001, iteration 4.
All done in environment 00001001.
[00001001] exiting gracefully
[00001001] free env 00001001
Hello, I am environment 00001002.
Back in environment 00001002, iteration 0.
Back in environment 00001002, iteration 1.
Back in environment 00001002, iteration 2.
Back in environment 00001002, iteration 3.
Back in environment 00001002, iteration 4.
All done in environment 00001002.
[00001002] exiting gracefully
[00001002] free env 00001002
Hello, I am environment 00001003.
Back in environment 00001003, iteration 0.
Back in environment 00001003, iteration 1.
Back in environment 00001003, iteration 2.
Back in environment 00001003, iteration 3.
Back in environment 00001003, iteration 4.
All done in environment 00001003.
[00001003] exiting gracefully
[00001003] free env 00001003
No runnable environments in the system!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

System Calls for Environment Creation

尽管你的内核现在可以在多个 *用户级别* 的环境运行和切换，但运行内核初始设置的环境仍是受限的。你现在将实现必须的 `JOS` 系统调用来允许 *用户环境* 创建和开始其它新的 *用户环境* 。

`Unix` 提供系统调用 `fork()` 作为其原始的进程创建。Unix `fork()` 复制整个调用进程（父进程）的地址空间来创建一个新的进程（子进程）。两个用户空间唯一可观察到的区别就是他们的 `process IDs` 和 `parent process IDs` （分别由 `getpid` 和 `getppid` ）得到。

在父进程，`fork()` 返回子进程的 `process ID`，而在子进程，`fork()` 返回0。默认的，每个进程有自己的私密地址空间，并且 *进程对内存的修改对其它是不可见的* (neither process's modifications to memory are visible to the other.)。

你将提供一个不同的，更原始的 `JOS` 系统调用的设置来创建新 *用户模式环境* 。有了这些系统调用你将能够完全实现一个用户空间的类Unix的 `fork()`，作为其它形式环境创建的补充。你将编写的新系统调用如下：

`sys_exofork`

这个系统调用用一个几乎白板创建一个新环境：其地址空间的用户部分没有映射任何东西，也不能运行。新环境拥有和父环境调用 `sys_exofork` 时**相同的寄存器状态**。在父进程，`sys_exofork` 返回新创建环境的 `envid_t` （或者一个负的错误代码，如果环境分配失败）。然而，在子进程，它返回0。（由于子进程刚开始被标记为不可运行，`sys_exofork` 实际上将不会在子进程返回，直到父进程显式地允许返回，通过标记子进程 *可运行* 。

`sys_env_set_status`

设置一个指定的环境状态为 `ENV_RUNNABLE` 或者 `ENV_NOT_RUNNABLE` 。这个系统调用通常用于标记一个新环境准备好去运行，一旦其 *地址空间* 和 *寄存器状态* 已被全部初始化。

`sys_page_alloc`

分配一页 *物理内存* ，映射其为给定环境地址空间的给定虚拟地址。

`sys_page_map`

复制 (页映射) （**不是**页内容），从一个环境的地址空间到另一个。保留 *内存共享* 功能，以至于新映射和旧映射都指向相同的物理内存页。

`sys_page_unmap`

取消给定环境的给定虚拟地址的映射。

对于上面所有的系统调用，接收 *环境IDs* 作为参数，`JOS` 支持将0转化为“当前环境”的意思。这个转换由 `kern/env.c` 中的 `envid2env()` 实现。

我们已经提供了一个非常原始的类Unix `fork()` 的实现，在测试程序 `user/dumbfork.c` 中。这个测试程序使用上面的 *系统调用* 来创建并运行一个 *子环境* ，这个子环境拥有其自身地址空间的副本。然后这两个环境会使用 `sys_yield` 来switch back and forth。父进程在10次迭代后退出，而子进程在20次迭代后退出。

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()`

```
// 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* e;
// 注意这里的bug，困扰了我很久。竟然调用了两次env_alloc()!!
// 非常低级的错误，但是也帮我更加理解了进程的fork
// int env_alloc_return = env_alloc(&e, curenv->env_id);

int r;
  if((r = env_alloc(&e, curenv->env_id)) != 0){
    return r;
  }
  e->env_status = ENV_NOT_RUNNABLE;
  // memmove(&e->env_tf, &curenv->env_tf, sizeof(curenv->env_tf));
  e->env_tf = curenv->env_tf;
```

```
  e->env_tf.tf_regs.reg_eax = 0;
  return e->env_id;
  // panic("sys_exofork not implemented");
}
```

`sys_env_set_status()`

```
// 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;
int r;
  if((r = envid2env(envid, &e, 1)) != 0){
    return r;
  }
  if(ENV_RUNNABLE != status && ENV_NOT_RUNNABLE != status){
    return -E_INVAL;
  }
  e->env_status = status;
  return 0;
  // panic("sys_env_set_status not implemented");
}
```

`sys_env_set_pgfault_upcall()`

```
// 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.
  panic("sys_env_set_pgfault_upcall not implemented");
}
```

`sys_page_alloc`

```
// 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;
int r;
  if((r = envid2env(envid, &e, 1)) != 0){
    return r;
  }
  if(UTOP <= (pte_t)va || (pte_t)va % PGSIZE != 0 || (perm &
(~PTE_SYSCALL)) != 0){
    return -E_INVAL;
  }
struct PageInfo* pp = page_alloc(ALLOC_ZERO);
  if(NULL == pp){
    return -E_NO_MEM;
  }
  if((r = page_insert(e->env_pgdir, pp, va, perm)) != 0){
    page_free(pp);
    return -E_NO_MEM;
  }
  return 0;
  // panic("sys_page_alloc not implemented");
}
```

`sys_page_map()`

```
// 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.
  // panic("sys_page_map not implemented");
struct Env* srce;
struct Env* dste;
int r;
  if(envid2env(srcenvid, &srce, 1) < 0 || envid2env(dstenvid,
&dste, 1) < 0){
    return -E_BAD_ENV;
  }
```

```
  if(UTOP ⩽ (pte_t)srcva || (pte_t)srcva % PGSIZE ≠ 0 || UTOP ⩽
(pte_t)dstva || (pte_t)dstva % PGSIZE ≠ 0){
    return -E_INVAL;
  }
pte_t* src_store;
  if(NULL == page_lookup(srce→env_pgdir, srcva, &src_store)){
    return -E_INVAL;
  }
  if((perm & (~PTE_SYSCALL)) ≠ 0){
    return -E_INVAL;
  }
  if((perm & (PTE_U | PTE_P)) ≠ (PTE_U | PTE_P)){
    return -E_INVAL;
  }
  if((PTE_W & *src_store) == 0 && (PTE_W & perm) == PTE_W){
    return -E_INVAL;
  }
struct PageInfo* dstpp = page_lookup(srce→env_pgdir, srcva, 0);
  if((r = page_insert(dste→env_pgdir, dstpp, dstva, perm)) ≠ 0){
    return r;
  }
  return 0;
}
```

`sys_page_unmap()`

```
// 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.
    // panic("sys_page_unmap not implemented");
  struct Env* e;
  int r;
    if((r = envid2env(envid, &e, 1)) ≠ 0){
      return r;
    }
    if(UTOP ⩽ (pte_t)va || (pte_t)va % PGSIZE ≠ 0){
      return -E_INVAL;
    }
    page_remove(e→env_pgdir, va);
    return 0;
  }
```

modification in `syscall()`

```
    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);
```

这就完成了lab的Part A的部分；确保运行 `make grade` 能通过所有Part A测试，然后 `make handin` 。如果你想知道一个确切的测试案例为什么会出错，运行 `./grade-lab4 -v` ，会向你展示内核build和QEMU在每个测试上运行的的输出，知道测试失败。当一个测试失败，脚本将会停止，然后你可以查看 `jos.out` 来看内核实际上打印了什么。

Summary

Part A 实现了多处理器和最原始的 `fork()` 调用。

1. 多处理器
    - 首先调用 `mp_init()` 来在 `BIOS` 中查找 *MP configure table*，获取多处理器系统的信息，如 *LAPIC* 的地址等。
    - 调用 `lapic_init()` 初始化 *Local APIC unit*。
    - 调用 `boot_aps()`

      `boot_aps()` 将 `mpentry.S` 中的代码复制到合适的地址并运行，`mpentry.S` 中的代码又会调用 `mp_main()`。

      `mp_main()` 中会对每个 *AP* 进行初始化，如环境，中断，LAPIC。
2. 创建子环境
    - 调用 `sys_exofork()`

      该函数使用 `env_alloc()` 来创建一个新的环境，并将新环境设置为 `ENV_NOT_RUNNABLE`，返回新创建环境的 *env_id*。
    - 关于如何理解子环境的返回值为0?

      在使用 `env_alloc()` 得到新的环境之后，修改新环境的 `e->env_tf.tf_regs.reg_eax = 0`。所以在父环境中会直接返回新创建环境的 *env_id*，子环境中由于寄存器 `eax = 0`,故返回值为0。

Part B: Copy-on-Write Fork

之前提到过，`Unix` 提供 `fork()` 系统调用作为其主要进程的创建原语。`fork()` 系统调用复制调用进程（父进程）的地址空间来创建一个新的进程（子进程）。

xv6 通过复制父进程页中的所有数据到为子进程创建的页中来实现 `fork()`。这本质上就是 `dumbfork()` 使用的方法，`fork()` 操作中开销最大的部分就是 *将父进程的地址空间复制进入子进程*。

然而，一个对 `fork()` 的调用，经常立即跟着子进程中对 `exec()` 的调用，这个调用将子进程的内存替换为一个新的程序。This is what the shell typically does, for example. 这种情况下，*复制父进程的内存空间* 很**浪费时间**，因为子进程在调用 `exec()` 之前讲几乎不会使用它的内存。

因此，之后的 `Unix` 版本利用虚拟内存硬件来实现 *父进程* 和 *子进程* **共享** 内存，该内存映射到它们各自的地址空间**直到其中一个进程修改了该内存**。这个技术称为 ***copy-on-write*** 。为了实现这个，内核将从父进程复制 *地址空间映射* 到子进程而不是映射页的内容，同时标记刚分享的页为 *read-only* 。当两进程中的其中一个尝试去写其中一个分享页，进程就会产生一个 `page fault` 。这时， `Unix` 内核就会意识到该页是一个"virtual" 或 "copy-on-write"的副本，所以它会 *创建一个新的，私有的，可写的页副本* 针对该faulting process。

这种机制下，私有的 *页内容* 不会真正地被复制，除非它们被真正地写入。这个优化使得子进程中紧跟着 `exec()` 的 `fork()` 开销要小的多：子进程可能只需要复制一页（它栈上的当前页）在它调用 `exec()` 之前。

在这个lab的下个部分，你将实现一个“合适的”类Unix `fork()` ，该函数使用 `copy-on-write` 机制，作为一个 *用户空间库例程* 。在用户空间实现支持 `copy-on-write` 的 `fork()` 是有好处的：内核会更加简单也因此更大可能性的正确。同时，也允许各个 *用户模式* 程序定义各自 `fork()` 的语义。如果一个程序想要一个稍微不同的实现，就可以很容易地提供自己的。比如，the expensive always-copy version like dumbfork(), or one in which the parent and child actually share memory afterward。

User-level page fault handling

一个用户级的 *copy-on-write* `fork()` 需要了解 *写保护* 页的 `page fault` ，所以你需要首先实现 `page fault` 。*Copy-on-write* 只是 *用户级页错误处理* 许多可能的用途之一。

设置一个地址空间用于 *page faults* 指示何时需要执行某些操作。比如，大多数的Unix核初始只在新进程栈的区域映射单个页，当进程栈的需求增长时会按照需求分配和映射额外的 *栈页* ，并且在没有映射的栈地址会导致 `page fault` 。一般来说，Unix内核必须跟踪 `page fault` 发生时采取了什么动作，这个 *page fault* 发生在进程空间的每个区域。比如， *栈区域* 的错误通常会分配和映射新的物理页；程序 *BSS区域* 的错误通常会分配新的页，初始化为0，然后映射它。在具有按需分页的可执行文件系统中， *text区域* 发生的错误会从磁盘上读取对应的二进制页，然后映射它。

对于 `kernel` 来说会有很多的信息去跟踪。你将在 *用户空间* 决定每个 `page fault` 发生时去做什么，而不是采取传统的Unix做法，另外 *用户空间* 产生的 *bugs* 危害性更低。这个设计还有额外的优点，允许程序在定义它们内存区域时有 *很大的灵活性* ；之后，在基于磁盘的文件系统，你将使用 *用户级的* 页错误处理程序来映射和访问文件。

Setting the Page Fault Handler

为了处理它自己的 `page fault` ，一个用户环境将需要在 `JOS` 内核注册一个 *page fault handler entrypoint* 。*用户环境* 通过新的 `sys_env_set_pgfault_upcall` 系统调用来注册它的 *page fault entrypoint* 。我们将向 `Env` 结构体加入一个新的成员， `env_pgfault_upcall` 来记录这个信息。

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.

`sys_env_set_pgfault_upcall()` in `kern/syscall.c`

```
// 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.
  // panic("sys_env_set_pgfault_upcall not implemented");
struct Env* e;
int r;
  if((r = envid2env(envid, &e, 1)) != 0){
    return r;
  }
  e->env_pgfault_upcall = func;
  return 0;
}
```

Normal and Exception Stacks in User Environments

一般的执行过程中，`JOS` 中的 *用户环境* 将运行在 *一般的* 用户栈上：它的 `ESP` 寄存器开始指向 `USTACKTOP`，它推送的栈数据驻留在 `USTACKTOP-PGSIZE` 和 `USTACKTOP-1` 之间，包括 `USTACKTOP-PGSIZE` 和 `USTACKTOP-1`。但是，当一个 `page fault` 发生在 *用户模式* 时，内核将在 *不同的栈* 上重新开始一个运行 *指定用户级页错误处理程序* 的 *用户环境*，这个栈被称为 (*用户异常栈*)。**本质上**，我们将使 `JOS` 内核 (代表) *用户环境* 实现自动的“栈切换”。这种方式和x86处理器从 *用户模式* 转向 *内核模式* 时 *代表* `JOS` 实现栈切换的方式大致相同。

`JOS` 用户异常栈一个页的大小，其顶部被定义为虚拟地址 `UXSTACKTOP`，所以 *用户异常栈* 的有效字节是从 `UXSTACKTOP-PGSIZE` 到 `UXSTACKTOP-1` （闭区间）。当在该 *异常栈* 运行时，用户级页错误处理程序可以使用 `JOS` 的常规系统调用来 *映射新的页* 或者 *调整映射* 来解决任何由于 `page fault` 导致的问题。然后，*用户级页错误处理程序返回*，通过一个 *汇编语言存根*，返回到 *原始栈上的错误代码*。

每个想要去支持用户级页错误处理程序的用户环境将需要去为其自己的 *异常栈* 来分配内存（使用part A中的 `sys_page_alloc()` 系统调用）。

Invoking the User Page Fault Handler

你现在将需要去修改 `kern/trap.c` 中的 *页错误处理代码* 来从 *用户模式* 处理页错误，如下所示。我们将用户环境发生错误的状态称为 `陷阱时间` 状态。

如果没有 *页错误处理程序* 注册，`JOS` 内核会摧毁用户环境并打印和之前一样的信息。否则，内核会设置一个 `trap frame` 在 *异常栈* 上，看起来像是 `inc/trap.h` 中的 `struct UTrapframe`：

```
                    <-- UXSTACKTOP
trap-time esp
trap-time eflags
trap-time eip
trap-time eax       start of struct PushRegs
trap-time ecx
trap-time edx
trap-time ebx
trap-time esp
trap-time ebp
trap-time esi
trap-time edi       end of struct PushRegs
tf_err (error code)
fault_va            <-- %esp when handler is run
```

内核然后安排 *用户环境* 用页错误处理程序来重新开始执行，该页错误处理程序运行于有着上面 `stack frame` 的异常栈；你必须**理解这件事如何发生**。 `fault_va` 是导致该 *页错误* 的虚拟地址。

如果当一个异常发生时， *用户环境* **已经** 运行在了 *用户异常栈* ，那么该页错误处理程序自身就发出 *fault* 。这种情况下，你应该重新开始一个新的 `stack frame` 就在当前 `tf->tf_esp` 的下面，而不是在 `UXSTACKTOP` 。你应该**首先**推入一个空的 *32-bit* 的字，然后是 `struct UTrapframe` 。

为了测试 `tf->tf_esp` 是否已经在 *用户异常栈* ，检查其是否在 `UXSTACKTOP-PGSIZE` 和 `UXSTACKTOP-1` （闭区间）范围内。

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?)

`page_fault_handler` in `kern/trap.c`

```
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(0 == (tf->tf_cs & 0x3)){
     panic("It's a kernel 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.
  //
  // It is convenient for our code which returns from a page fault
  // (lib/pfentry.S) to have one word of scratch space at the top
of the
  // trap-time stack; it allows us to more easily restore the
eip/esp. 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.
  // test whether there is env_pgfault_upcall
  if(curenv->env_pgfault_upcall){
struct UTrapframe *utf;
    // test whether tf->tf_esp is already on the user exception stack
    if(UXSTACKTOP - PGSIZE <= tf->tf_esp && UXSTACKTOP - 1 >= tf->tf_esp){
      *(pte_t*)(tf->tf_esp - 4) = 0;
      utf = (struct UTrapframe*)(tf->tf_esp - 4);
    }else{
      utf = (struct UTrapframe*)(tf->tf_esp);
    }
    user_mem_assert(curenv, UXSTACKTOP, PGSIZE, 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 = (uint32_t)(curenv->env_pgfault_upcall);
    curenv->env_tf.tf_esp = (uint32_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);
```

```
}
```

该函数的主要目的是要在 `exception stack` 保存一个 `struct UTrapframe`，该结构保存了当前进程的相关数据，也就是 `curenv->env_tf` 的某些字段。然后再修改 `curenv->env_tf` 的 `eip` 和 `esp` 字段来在 `exception stack` 上运行进程。

User-mode Page Fault Entrypoint

接下来，你需要实现 *汇编例程* 来调用C语言的页错误处理程序并且重新执行 *导致错误的指令* 。这个汇编例程是一个处理程序，这个处理程序将会用 `sys_env_set_pgfault_upcall()` 来在内核注册。

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.

`_pgfault_upcall` in `lib/pfentry.S`

```
#include <inc/mmu.h>
#include <inc/memlayout.h>

// Page fault upcall entrypoint.

// This is where we ask the kernel to redirect us to whenever we
cause
// a page fault in user space (see the call to
sys_set_pgfault_handler
// in pgfault.c).
//
// When a page fault actually occurs, the kernel switches our ESP
to
// point to the user exception stack if we're not already on the
user
// exception stack, and then it pushes a UTrapframe onto our user
```

```
// exception stack:
//
//  trap-time esp
//  trap-time eflags
//  trap-time eip
//  utf_regs.reg_eax
//  ...
//  utf_regs.reg_esi
//  utf_regs.reg_edi
//  utf_err (error code)
//  utf_fault_va            <-- %esp
//
// If this is a recursive fault, the kernel will reserve for us a
// blank word above the trap-time esp for scratch work when we
unwind
// the recursive call.
//
// We then have call up to the appropriate page fault handler in C
// code, pointed to by the global variable '_pgfault_handler'.

.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

  // 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.
  //
  // LAB 4: Your code here.
  movl 0x30(%esp), %edi
  subl $4, %edi
  movl %edi, 0x30(%esp)
  movl 0x28(%esp), %edx
  movl %edx, (%edi)


  // 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

  // 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

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

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

最后，你需要实现C语言的用户库端，该用户库实现页错误处理机制。

Exercise 11.

Finish `set_pgfault_handler()` in `lib/pgfault.c`.

`set_pgfault_handler()`

```
// 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.
```

```
    // panic("set_pgfault_handler not implemented");
    if((r = sys_page_alloc(thisenv->env_id, (void*)(UXSTACKTOP -
PGSIZE), PTE_W | PTE_U | PTE_P)) != 0){
      panic("set_pgfault_handler: %e", r);
    }
    if((r = sys_env_set_pgfault_upcall(thisenv->env_id,
_pgfault_upcall)) != 0){
      panic("set_pgfault_handler: %e", r);
    }
  }

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

Implementing Copy-on-Write Fork

你现在有了可以在*用户空间*完全实现 `copy-on-write` 的 `fork()` 的完整内核设施。

我们已经提供了 `fork()` 的骨架，位于 `lib/fork.c` 中。和 `dumbfork()` 一样，`fork()` 应该创建一个新环境，然后扫描*父环境*的整个地址空间并在*子环境*设置对应的页映射。**关键不同是**：`dumbfork()` 复制页，`fork()` 仅仅初始化*页映射*。`fork()` 只有当一个环境尝试去写它时，才*复制整个页*。

`fork()` 基本的控制流如下：

1. 父进程install `pgfault()` 作为C级别的页错误处理程序，使用之前实现的 `set_pgfault_handler()`
2. 父进程调用 `sys_exofork()` 来创建一个*子环境*
3. 对于每个位于其地址空间 `UTOP` 之下的*可写*或*copy-on-write* 页，父进程调用 `duppage`，该方法应将*page copy-on-write* 映射进入*子进程*的地址空间，然后重新将*page copy-on-write* 映射到其自己的地址空间。（**注意**：这里的*顺序*很重要！试着去想一想特殊例子，颠倒顺序会导致错误） `duppage` set boths `PTEs`，使得页不可读；在 `avail` 字段包含 `PTE_COW` 来区分*copy_on_write page* 和*read-only page*。

但是，*异常栈* 不使用这种方式重新映射。你需要分配一个 *新的页* 给子进程中的 *异常栈* 。因为 *页错误处理程序* 将会做真正的复制，而且 *页错误处理程序* 运行在 *异常栈* 上。

The exception stack cannot be made copy-on-write: who would copy it?

`fork()` 也需要处理页（with present bit），但不是writable 或 copy-on-write

4. 父进程为子进程设置 *用户页错误进入点* ，来使得其看起来像自己一样
5. 子进程现在准备好去运行了，所以父进程标记它为 `runnable`

每当有进程（环境）写一个 *copy-on-write* 页时（还没有完全写结束），它将有一个 *页错误* 。下面是 *用户页错误处理程序* 的控制流：

1. 内核将 *页错误* 传到 `_pgfault_upcall` ，它会调用 `fork()` 的 `pgfault()` 的处理程序
2. `pgfault()` 检查错误是否为 *写错误* （检查error code的 `FEC_WR` 字段）和该页的PTE是否被标记为 `PTE_COW` 。如果没有，则 `painc`
3. `pgfault()` 分配一个新的页（映射于一个临时位置），将 *错误页面* 的内容复制到新的页。然后 *错误处理程序* 将新的页映射到合适的地址，权限为 `read/write` ，代替旧的 *只读* 映射

用户级的 `lib/fork.c` 代码必须向环境的页表询问上述的几个操作（例如，PTE for a page is marked `PTE_COW` ）。内核将进程的页表映射到 `UVPT` ,就是为了这个目的。它使用一个 *clever mapping trick* 来实现使得查询用户代码的PTEs很简单。 `lib/entry.S` 设置 `uvpt` 和 `uvpd` 来让你简单地查询 `lib/fork.c` 中的页表信息。

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.

```
1000: I am ''
1001: I am '0'
```

```
2000: I am '00'
2001: I am '000'
1002: I am '1'
3000: I am '11'
3001: I am '10'
4000: I am '100'
1003: I am '01'
5000: I am '010'
4001: I am '011'
2002: I am '110'
1004: I am '001'
1005: I am '111'
1006: I am '101'
```

`fork()`

```
// 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;
```

```
envid_t child_envid;
  set_pgfault_handler(pgfault);

  child_envid = sys_exofork();
  if(child_envid == 0){
    // we are child
    thisenv = &envs[ENVX(sys_getenvid())];
    return 0;
  }
uint32_t start = PGNUM(UTEXT);
uint32_t end = PGNUM(USTACKTOP);
  for(uint32_t i = start; i < end; ++i){
    if((uvpd[i >> 10] & PTE_P) && (uvpt[i] & PTE_P)){
      if((r = duppage(child_envid, i)) != 0){
        return r;
      }
    }
  }
  if((r = sys_page_alloc(child_envid, (void*)(UXSTACKTOP -
PGSIZE), PTE_P | PTE_U | PTE_W)) != 0){
    panic("fork: %e", r);
  }
extern void _pgfault_upcall(void);
  if((r = sys_env_set_pgfault_upcall(child_envid,
_pgfault_upcall)) != 0){
    panic("fork, sys_env_set_pgfault_upcall: %e", r);
  }
  if((r = sys_env_set_status(child_envid, ENV_RUNNABLE)) != 0){
    return r;
  }
  return child_envid;
  // panic("fork not implemented");
}

// Challenge!
int
sfork(void)
{
  panic("sfork not implemented");
```

```
  return -E_INVAL;
}
```

`duppage()`

```
// 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* addr = (void*)(pn * PGSIZE);
pte_t pte = uvpt[pn];
  if((pte & PTE_W) == PTE_W || (pte & PTE_COW) == PTE_COW){
    if((r = sys_page_map(0, addr, envid, addr, PTE_COW | PTE_P |
PTE_U)) < 0){
      panic("duppage: %e", r);
    }
    if((r = sys_page_map(0, addr, 0, addr, PTE_COW | PTE_P |
PTE_U))){
      panic("duppage: %e", r);
    }
  }else{
    if((r = sys_page_map(0, addr, envid, addr, PTE_P | PTE_W |
PTE_U)) < 0){
```

```
      panic("duppage: %e", r);
    }
  }
  // panic("duppage not implemented");
  return 0;
}
```

pgfault()

```
// 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.
pte_t pte = uvpt[PGNUM(addr)];
  if(((err & FEC_WR) == 0) || ((pte & PTE_COW) == 0)){
    panic("faulting access is not a write or a copy-on-write
page.");
  }

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

    // panic("pgfault not implemented");
  }
```

Summary

`fork` *工作流程*

- 设置页错误处理程序

  ```
  set_pgfault_handler(pgfault);
  ```

  上述函数中设置 `e->env_pgfault_upcall` 为 `_pgfault_upcall`，而 `_pgfault_upcall` 为一个汇编例程。`_pgfault_upcall` 中首先会调用上述代码中设置的*pgfault handler*，也就是函数 `pgfault`，然后会将处理器上下文恢复为发生错误时的状态。

- 复制页映射

  当发生页错误时，控制首先跳转到 `trap.c` 中的 `page_fault_handler()`，该函数会判断*是否设置* `e->env_pgfault_upcall`，若设置，则在内核环境设置 `struct UTrapframe`，并重新跳回用户环境。

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

在lab4的最后部分，你将修改内核实现 *抢占* 不合作的环境并允许环境之间显式地互相传递信息。

Clock Interrupts and Preemption

运行 `user/spin` 这个测试程序分叉一个 *子环境* ，该环境一旦得到CPU的控制权就会在一个紧密的循环中永远旋转。父进程和内核都无法重新获取CPU。这显然不是一个理想的状态，对于保护系统不受 *用户环境* 中的漏洞或恶意代码所影响，因为任何 *用户模式* 环境都可以简单的通过一个 *无限* 的循环将整个系统暂停并且不交还CPU。为了允许内核抢占一个运行环境并强制重新获取CPU的控制权，我们必须扩展 `JOS` 内核来支持来自时钟硬件的 *外部硬件中断* 。

Interrupt discipline

外部中断（i.e., device interrupts)指的是 `IRQs` 。一共有16个可能的 `IRQs` ，从0到15。从 `IRQ` 编号到 `IDT` 条目的映射不是固定的。 `pic_init` in `picirq.c` 将 `IRQs` 0-15映射到 `IDT` 条目从 `IRQ_OFFSET` 到 `IRQ_OFFSET+15` 。

在 `inc/trap.h` 中， `IRQ_OFFSET` 被定义为十进制32.因此IDT条目的32-47对应于 `IRQs` 0-15。举个例子，*时钟中断* 是 `IRQ 0` 。因此， `IDT[IRQ_OFFSET+0]` 包含了位于内核的 *时钟中断处理程序例程* 的地址。选用 `IRQ_OFFSET` 所以 *设备中断* 和 *处理器异常* 不会重叠，否则会导致歧义。(In fact, in the early days of PCs running MS-DOS, the `IRQ_OFFSET` effectively was zero, which indeed caused massive confusion between handling hardware interrupts and handling processor exceptions!)

在 `JOS` 中，相比于 `xv6 Unix` 我们做了一个关键的简化。*外部设备中断* 总是被禁止的当处于内核态时。*外部中断* 被 `%eflags` 寄存器中的 `FL_IF` 标志位所控制（see `inc/mmu.h` ）。当这个位被设置，*外部中断* 被启用。尽管该位可以使用好几种方式来修改，但由于我们的简化，我们完全可以通过 *保存* 和 *恢复* 寄存器的过程来处理它，当我们进入和离开 *用户模式* 时。

你将不得不确保 `FL_IF` 标志在 *用户环境* 中设置，当它们在运行时，所以当一个中断到达时，它被传入处理器和被你的 *中断代码* 处理。**否则**，中断被掩蔽或 *忽略* 直到中断被重新启用。我们使用 `bootloader` 中的第一条指令来掩蔽中断，目前为止，我们还没有抽出时间来重新启用它们。

> **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.

```
+++ b/kern/env.c
@@ -270,6 +270,7 @@ env_alloc(struct Env **newenv_store, envid_t
parent_id)

     // Enable interrupts while in user mode.
     // LAB 4: Your code here.
+       e->env_tf.tf_eflags |= FL_IF;

+++ b/kern/sched.c
@@ -90,7 +90,7 @@ sched_halt(void)
                "pushl $0\n"
                "pushl $0\n"
                // Uncomment the following line after completing
exercise 13
-               //"sti\n"
+               "sti\n"
                "1:\n"
                "hlt\n"
                "jmp 1b\n"

+++ b/kern/trap.c
```

```
@@ -94,6 +94,13 @@ trap_init(void)
     void t_syscall();
     void t_default();

+       void irq_timer();
+       void irq_kbd();
+       void irq_serial();
+       void irq_spurious();
+       void irq_ide();
+       void irq_error();
+
     SETGATE(idt[T_DIVIDE], 0, GD_KT, &t_divide, 0);
     SETGATE(idt[T_DEBUG], 0, GD_KT, &t_debug, 0);
     SETGATE(idt[T_NMI], 0, GD_KT, &t_nmi, 0);
@@ -115,6 +122,13 @@ trap_init(void)
     SETGATE(idt[T_SYSCALL], 0, GD_KT, &t_syscall, 3);
     SETGATE(idt[T_DEFAULT], 0, GD_KT, &t_default, 0);

+       SETGATE(idt[IRQ_OFFSET + IRQ_TIMER], 0, GD_KT, &irq_timer,
0);
+       SETGATE(idt[IRQ_OFFSET + IRQ_KBD], 0, GD_KT, &irq_kbd, 0);
+       SETGATE(idt[IRQ_OFFSET + IRQ_SERIAL], 0, GD_KT,
&irq_serial, 0);
+       SETGATE(idt[IRQ_OFFSET + IRQ_SPURIOUS], 0, GD_KT,
&irq_spurious, 0);
+       SETGATE(idt[IRQ_OFFSET + IRQ_IDE], 0, GD_KT, &irq_ide, 0);
+       SETGATE(idt[IRQ_OFFSET + IRQ_ERROR], 0, GD_KT, &irq_error,
0);
```

Handling Clock Interrupts

在 `user/spin` 程序中，*在 子环境* 首先被运行之后，它仅仅在循环中不断执行，内核无法拿回控制权。我们需要给硬件编程来 *周期性地* 生成 *时钟中断* ，以便能够强制将控制转移给内核，然后我们可以将控制给不同的用户环境。

对 `lapic_init` 和 `pic_init` 的调用（from `i386_init` in `init.c`）设置了 *时钟* 和 *中断控制器* 来生成中断。你现在需要编写代码来处理这些中断。

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.

`trap_dispatch()` in `kern/trap.c`

```
@@ -255,6 +268,10 @@ trap_dispatch(struct Trapframe *tf)
                                tf->tf_regs.reg_ebx, tf-
>tf_regs.reg_edi,
                                tf->tf_regs.reg_esi);
          return;
+               case IRQ_OFFSET + IRQ_TIMER:
+                       lapic_eoi();
+                       sched_yield();
+                       return;
```

这是一个好的时间点来做一些回归测试。确保没有由于 *enabling interrupts* 而破坏之前的部分(e.g. forktree)。同时，尝试使用 `make CPUS=2 target` 运行多个CPUs。你现在应该可以通过 `stressshed` 。运行 `make grade` 来验证。你现在应该得到这个实验的65/80分。

Inter-Process communication(IPC)

（在 `JOS` 的专业术语中，这被称为 *inter-environment communication* 或 *IEC* ，但是其他人都称它为 `IPC` ，所以我们也这么叫。）

我们一直关注于操作系统的 *隔离* 方面，它提供了一种幻想--每个程序都自己拥有一整台机器。而操作系统另一个重要的服务就是，**允许程序之间互相通信**。让程序和其它程序进行交互可以非常强大。 `Unix pipe model` 就是一个权威性的例子。

有很多关于 *interprocess communication* 的模型，甚至现在仍然有关于哪个模型是最好的辩论，我们不会参与到争辩。相反，我么将实现一个简单的 `IPC` 机制然后试试看。

IPC in JOS

你将实现一些额外的 `JOS` 内核系统调用来共同提供一个简单的 *进程间通信机制* 。你将实现两个 *系统调用* ， `sys_ipc_recy` 和 `sys_ipc_try_send` 。然后你将实现两个库包装 `ipc_recv` 和 `ipc_send` 。

用户环境可以使用 `JOS` 的 `IPC` 机制来互相发送的“消息”包括两个部分： *单个32-bit值* 和可选择的 *单个页映射* 。允许环境在“消息”中传递 *页映射* 提供了一个比填入单个32-bit整数值更高效的转移数据的方式；也允许环境能简单地设置共享 *内存安排* 。

Sending and Receiving Messages

为了收到“消息”，一个环境调用 `sys_ipc_recv` 。这个系统调用取消当前环境的调度，并且不再运行直到一个消息被收到。当一个环境正在等待接收一个“消息”， *任何其它环境* 可以向它发送消息--不只是特定的环境，也不只是和 *接收环境* 有父/子安排。换句话说，在Part A中实现的权限检查不会应用到 `IPC` ，因为 `IPC` 系统调用被细心设计来保证”安全“：一个环境不能通过发送信息导致另一个环境发生故障（除非目标环境本身就是错误的）。

为了尝试去发送一个值，一个环境调用 `sys_ipc_try_send` ，带有接收者的 *环境id* 和 *被发送的值* 。如果指定的环境正在接收（它已经调用了 `sys_ipc_recv` 但还没有得到一个值），然后发送“消息”的函数传递出“消息”然后返回0。否则， `the send` 返回 `-E_IPC_NOT_RECV` 来表明目标环境当前没有收到一个值。

用户空间的一个库函数 `ipc_recv` 负责调用 `sys_ipc_recv` ，然后在当前环境的 `struct Env` 寻找关于接收值的信息。

类似的，库函数 `ipc_send` 负责重复调用 `sys_ipc_try_send` 知道发送成功。

Transfer Pages

当一个环境调用 `sys_ipc_recv` ，参数为有效的 `srcva` （below UTOP），它表示the sender 想发送当前映射在 `srcva` 的页给接收者，权限为 `perm` 。在成功的 `IPC` 之后，发送者保持着在其地址空间 `srcva` 位置页的原始映射，但是接收者也获得了相同物理页的映射，该映射位置 `dstva` 由接收者原始指定，在接收者的地址空间。最后，这个页就在发送者和接收者之间共享。

如果发送者和接收者都没有表明一个页应该被转移，就**没有页**被转移。在任何 `IPC` 之后，内核在接收者的 `Env` 结构设置一个新的字段 `env_ipc_perm` 给接收页的权限，或是 `0` 如果没有页被接收。

Implementing IPC

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` in `kern/syscall.c`

```
// 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(UTOP > (uint32_t)dstva){
    if(PGOFF(dstva) != 0){
      return -E_INVAL;
    }
  }
  curenv->env_ipc_recving = 1;
  curenv->env_ipc_dstva = dstva;
  curenv->env_status = ENV_NOT_RUNNABLE;
  // panic("sys_ipc_recv not implemented");
  return 0;
}
```

`sys_ipc_try_send()` in `kern/syscall.c`

```
// 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.
int r;
struct Env* e;
struct PageInfo* pp;
pte_t* pte;
  if((r = envid2env(envid, &e, 0)) != 0){
    return r;
  }
// 这里的判断条件不能加上 || e->env_ipc_from != 0 !!!
  if(e->env_ipc_recving == 0){
    return -E_IPC_NOT_RECV;
  }
  if((uint32_t)srcva < UTOP){
```

```
    if(PGOFF(srcva) ≠ 0){
      return -E_INVAL;
    }
    if(((perm & (PTE_P | PTE_U)) ≠ (PTE_P | PTE_U))){
      return -E_INVAL;
    }
    if((perm & (~PTE_SYSCALL)) ≠ 0){
      return -E_INVAL;
    }
    if((pp = page_lookup(curenv→env_pgdir, srcva, &pte)) == NULL)
{
      return -E_INVAL;
    }
    if((perm & PTE_W) == PTE_W && (*pte & PTE_W) == 0){
      return -E_INVAL;
    }
    if((r = page_insert(e→env_pgdir, pp, e→env_ipc_dstva, perm))
≠ 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_status = ENV_RUNNABLE;
  return 0;
//  if((uint32_t)srcva < UTOP && PGOFF(srcva) ≠ 0){
//    return -E_INVAL;
//  }
//  if(UTOP > (uint32_t)srcva && ((perm & (PTE_P | PTE_U)) ≠
PTE_P | PTE_U)){
//    return -E_INVAL;
//  }
//  if(UTOP > (uint32_t)srcva && (perm & (~PTE_SYSCALL)) ≠ 0){
//    return -E_INVAL;
//  }
```

```
//  if(UTOP > (uint32_t)srcva){
//    if((pp = page_lookup(curenv->env_pgdir, srcva, &pte)) ==
NULL){
//      return -E_INVAL;
//    }
//  }
//  if((perm & PTE_W) && !(*pte & PTE_W)){
//    return -E_INVAL;
//  }
//  if((r = page_insert(e->env_pgdir, pp, e->env_ipc_dstva, perm))
!= 0){
//    return r;
//  }
//

  // panic("sys_ipc_try_send not implemented");

}
```

`ipc_recv()` in `lib/ipc.c`

```
// 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 == NULL){
    pg = (void*)UTOP;
  }
  r = sys_ipc_recv(pg);
  if(r == 0){
    if(NULL != from_env_store){
      *from_env_store = thisenv->env_ipc_from;
    }
    if(NULL != perm_store){
      *perm_store = thisenv->env_ipc_perm;
    }
    return thisenv->env_ipc_value;
  }else{
    if(NULL != from_env_store){
      *from_env_store = 0;
    }
    if(NULL != perm_store){
      *perm_store = 0;
    }
    return r;
  }
  // panic("ipc_recv not implemented");
  // return 0;
}
```

`ipc_send()` in `lib/ipc.c`

```
// 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.
int r = 1;
if(pg == NULL){
  pg = (void*)UTOP;
}
  while(r != 0){
    r = sys_ipc_try_send(to_env, val, pg, perm);
    if(r < 0 && r != -E_IPC_NOT_RECV){
      panic("ipc_send: %e", r);
    }
    sys_yield();
  }
  // panic("ipc_send not implemented");
}
```

Before handing in, use `git status` and `git diff` to examine your changes and don't forget to `git add answers-lab4.txt`. When you're ready, commit your changes with `git commit -am 'my solutions to lab 4'`, then `make handin` and follow the directions.