

操作系统上机实验

Lab 4：抢占式多任务处理

实验报告

December 23, 2019

曹元议 1711425
黄艺璇 1711429
王雨奇 1711299
张瑞宁 1711302
阔坤柔 1611381
冯晓妍 1513408

Abstract

完成了前三个 Lab 实验以后，我们将在 Lab 3 的基础上上进一步探究多任务处理、IPC 等机制。本次实验主要分为三个部分：第一部分（Part A）是多处理器支持与协作式多任务，在这一部分将会使 JOS 支持多处理器和轮转调度，并且增加一些系统调用。第二部分（Part B）是写时复制的 Fork，这一部分需要实现一个与 Unix 类似`fork()`函数，支持写时复制技术。第三部分（Part C）是抢占式多任务和进程间通信，这一部分是在前两部分基础上实现不同进程间相互传递消息以及时钟中断。

Keywords: 操作系统；QEMU；JOS；进程；多处理器；`fork`；IPC；缺页；系统调用；时钟中断

Contents

1 小组分工情况	1
2 Part A: 多处理器支持与协作式多任务	1
2.1 作业 1	2
2.2 问题 1	6
2.3 作业 2	7
2.4 作业 3	9
2.5 作业 4	12
2.6 问题 2	17
2.7 作业 5A	18
2.8 问题 3	22
2.9 作业 6	23
3 Part B: 写时复制的 Fork	35
3.1 背景知识	35
3.2 作业 7	36
3.3 作业 8	39
3.4 作业 5B	41
3.5 作业 9	45
3.6 Testing	46
3.7 作业 10	49
4 Part C: 抢占式多任务和进程间通信	57
4.1 作业 11	57
4.2 作业 12	63
4.3 作业 13	65
5 总结	75

1 小组分工情况

	问题解决	报告整理	报告检查	小组讨论	Lab 3 任务
曹元议	√	√		√	作业 6、作业 10、作业 13
黄艺璇	√		√	√	作业 4、问题 2
王雨奇	√		√	√	作业 7、作业 8、作业 5B、作业 9
张瑞宁	√		√	√	作业 5A、问题 3
阔坤柔	√		√	√	作业 1、问题 1、作业 2、作业 3
冯晓妍	√		√	√	作业 11、作业 12

Table 1: 小组分工

曹元议	黄艺璇	王雨奇	张瑞宁	阔坤柔	冯晓妍
1/6	1/6	1/6	1/6	1/6	1/6

Table 2: 贡献比

2 Part A: 多处理器支持与协作式多任务

在本次实验中，我们使用前三个实验配置好的环境。根据实验指导取得 Lab 4 的代码，并把分支切换到 Lab 4 的分支。

我们取得了这些新的的实验文件：

文件	说明
kern/cpu.h	内核私有的关于多处理器支持的定义
kern/mpconfig.c	用于读取多处理器配置的代码
kern/lapic.c	驱动每个处理器 Local APIC(LAPIC) 单元的内核代码
kern/mpentry.S	非启动 CPU 的汇编入口
kern/spinlock.h	内核私有的自旋锁的定义，包括全局内核锁 (big kernel lock)
kern/spinlock.c	实现自旋锁的内核代码
kern/sched.c	将要实现的调度方法的代码框架

Table 3: Lab 4 的新文件

其中遇到了输入 `git merge lab3` 不能合并分支的问题，解决这个问题可以先输入 `git diff --check` 来检查冲突，再按照输出结果解决冲突。如果再输入 `git diff --check` 没有冲突，则再用 `git commit` 提交一遍就可以合并分支了。

```

user@pc: ~/oslab/lab1/src/lab3
user@pc:~/oslab/lab1/src/lab3$ git diff --check
kern/pmap.c:109: trailing whitespace.
+ [REDACTED]
kern/pmap.c:225: leftover conflict marker
kern/pmap.c:229: leftover conflict marker
kern/pmap.c:230: trailing whitespace.
+ [REDACTED]
kern/pmap.c:231: leftover conflict marker
kern/pmap.c:320: trailing whitespace.
+ // This way we preserve the real-mode IDT and BIOS structures [REDACTED]
kern/pmap.c:321: trailing whitespace.
+ // in case we ever need them. (Currently we don't, but...) [REDACTED]
kern/pmap.c:323: trailing whitespace.
+ // 2) The rest of base memory, [PGSIZE, npages_basemem * PGSIZE] [REDACTED]
kern/pmap.c:325: trailing whitespace.
+ for (i = 1; i < npages_basemem; i++) { [REDACTED]
kern/pmap.c:330: trailing whitespace.
+ // 3) Then comes the IO hole [IOPHYSMEM, EXTPHYSMEM], which must [REDACTED]
kern/pmap.c:340: trailing whitespace.
+ for (i = EXTPHYSMEM/PGSIZE; i < first_free_address/PGSIZE; i++) { [REDACTED]
kern/pmap.c:342: trailing whitespace.
+ } [REDACTED]
kern/pmap.c:343: trailing whitespace.
+ for (i = first_free_address/PGSIZE; i < npages; i++) { [REDACTED]

```

Figure 1: git diff –check

2.1 作业 1

2.1.1 题目原文

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).

2.1.2 题目大意

阅读 `kern/init.c` 中的 `boot_aps()` 和 `mp_main()` 方法，和 `kern/mpentry.S` 中的汇编代码。确保已经明白了引导 AP 启动的控制流执行过程。接着，修改你在 `kern/pmap.c` 中实现过的 `page_init()` 以避免将 `MPENTRY_PADDR` 加入到 free list 中，以使得我们可以安全地将 AP 的引导代码拷贝于这个物理地址并运行。你的代码应当通过我们更新过的 `check_page_free_list()` 测试，不过可能仍会在更新过的 `check_kern_pgdir()` 测试中失败，后续将会解决这个问题。

2.1.3 修改 page_init()

由实验文档可大致了解 AP 引导和启动的过程。在 SMP 体系结构中，引导处理器 (BSP) 负责初始化系统以引导操作系统，当操作系统启动时，应用处理器 (AP) 由 BSP 激活。系统通过 BSP 引导之后，剩下的被激活的 AP 处理器就不用引导系统了，而是直接把地址 0x7000 作为入口地址，开始执行系统。每个 AP 根据输入地址执行相应的代码。因此，为了防止该页被其他 AP 映射并更改其中的内容，需要设置 0x7000 这个页面为已使用。`kern/mpentry.S` 是 AP 的入口代码，它在 `kern/init.c` 的 `boot_aps()` 中被拷贝到了入口地址 0x7c00，然后 `boot_aps()` 依次激活每个 AP。然后 `kern/init.c` 的 `mp_main()` 对 AP 进行一些设置。宏定义 `MPENTRY_PADDR` 对应这个入口地址 0x7000，如下图所示：

```
// Physical address of startup code for non-boot CPUs (APs)
#define MPENTRY_PADDR 0x7000
```

Figure 2: inc/memlayout.h

因此修改 `kern/pmap.c` 中的 `page_init()` 函数，避免将 `MPENTRY_PADDR` 处的页添加到空闲列表中。实际上就是标记 `MPENTRY_PADDR` 开始的一个物理页为已使用，在 `page_init()` 中做一个特例处理，不将这一页的空闲链表分配出去。

因此作如下修改：

```
1 void
2 page_init(void)
3 {
4     // Change the code to reflect this.
5     // NB: DO NOT actually touch the physical memory corresponding to
6     // free pages!
7     size_t i;
8     .....
9     pages[0].pp_ref = 1;
10    // 2) The rest of base memory, [PGSIZE, npages_basemem * PGSIZE)
11    //     is free.
12    size_t mp_page = MPENTRY_PADDR/PGSIZE;
13    for (i = 1; i < npages_basemem; i++) {
14        if(i == mp_page){
15            pages[i].pp_ref = 1;
16            pages[i].pp_link = NULL;
17            continue;
18        }
19        pages[i].pp_ref = 0;
20        pages[i].pp_link = page_free_list;
21        page_free_list = &pages[i];
```

```

22 }
23 .....
24 }
```

输入make qemu的测试结果:

```

6828 decimal is 15254 octal!
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
kernel panic on CPU 0 at kern/pmap.c:629: mmio_map_region not implemented
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> 
```

Figure 3: make qemu

函数mmio_map_region()没有实现，这道题虽然不在实验文档上，但是不实现本次Lab无法完成。

2.1.4 mmio_map_region() 的实现

JOS 在 MMIOBASE (0xef800000) 到MMIOLIM预留了 4MB (一个PTSIZE) 来映射 LAPIC 设备，它用于支持 JOS 的抢占式多任务处理。我们需要一个函数来分配这个空间并在其中映射 LAPIC 设备内存。

```

*   MMIOLIM -----> +-----+ 0xeefc00000
*           |       Memory-mapped I/O      | RW/- PTSIZE
* ULIM, MMIOBASE --> +-----+ 0xef800000
```

Figure 4: inc/memlayout.h

先看kern/lapic.c的lapic_init()函数，它将从 lapicaddr (MMIOBASE) 开始的 4kB 物理地址映射到虚拟地址，并返回其起始地址。

```

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.
    lapic = mmio_map_region(lapicaddr, 4096);
```

Figure 5: kern/lapic.c 的 lapic_init() 函数

由注释可知，它是以页为单位进行映射的，可以使用boot_map_region() 来建立所需要的映射。需要注意的是，每次需要更改base的值，使得每次都是映射到一个新的页面，不要让映射超过了MMIOBASE和MMIOLIM (否则需要panic)。并且，需要设置的权限位为PCD (禁用 cache)、PWT (写直达)、W。

```

// 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);
    pa = ROUNDDOWN(pa, PGSIZE);
    if(size + base >= MMIOLIM || base + size < base)
        panic("mmio_map_region:overflow");
    boot_map_region(kern_pgdir, base, size, pa, PTE_PCD|PTE_PWT|PTE_W);
    uintptr_t ret = base;
    base = base +size;
    return (void*) ret;
}

```

Figure 6: kern/pmap.c 的 mmio_map_region() 函数

输入make qemu的测试结果：

```

6828 decimal is 15254 octal!
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
map region size = 8192, 2 pages
map region size = 4096, 1 pages
check_page() succeeded!
map region size = 4194304, 1024 pages
map region size = 126976, 31 pages
map region size = 32768, 8 pages
map region size = 268435455, 65535 pages
kernel panic on CPU 0 at kern/pmap.c:890: assertion failed: check_va2pa(pgdir, b
ase + KSTKGAP + i) == PADDR(percpu_kstacks[n]) + i
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.

```

Figure 7: make qemu

没有通过check_kern_pgdir()测试。

2.2 问题 1

2.2.1 题目原文

Compare kern/mpentry.S side by side with boot/boot.S. Bearing in mind that kern/mpentry.S is compiled and linked to run above KERNBASE just like everything else in the kernel, what is the purpose of macro MPBOOTPHYS? Why is it necessary in kern/mpentry.S but not in boot/boot.S? In other words, what could go wrong if it were omitted in kern/mpentry.S?
 Hint: recall the differences between the link address and the load address that we have discussed in Lab 1.

2.2.2 题目大意

逐行比较 `kern/mpentry.S` 和 `boot/boot.S`。牢记 `kern/mpentry.S` 和其他内核代码一样也是被编译和链接在 `KERNBASE` 之上运行的。那么，`MPBOOTPHYS` 这个宏定义的目的是什么呢？为什么它在 `kern/mpentry.S` 中是必要的，但在 `boot/boot.S` 却不用？换句话说，如果我们忽略掉 `kern/mpentry.S` 哪里会出现问题呢？提示：在 Lab 1 讨论的链接地址和装载地址的不同之处。

2.2.3 回答

```
# 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
```

Figure 8: kern/mpentry.S

在 `kern/mpentry.S` 中说明了其与 `boot/boot.S` 的区别：

1. `kern/mpentry.S` 不需要启用 A20。如果 A20 Gate 被打开了，则在实模式下，程序员可以直接访问 `100000H~10FFEFH` 之间的内存，如果 A20 Gate 被禁止，则在实模式下，若程序员访问 `100000H~10FFEFH` 之间的内存，则会被硬件自动转换为 `0H 0FFEFH` 之间的内存。
2. `kern/mpentry.S` 使用 `MPBOOTPHYS` 计算其符号的绝对地址，而不是依靠链接器来填充。

在kern/mpentry.S中，宏MPBOOTPHYS的定义如下：

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

Figure 9: kern/mpentry.S 中宏 MPBOOTPHYS 的定义

意为，从mpentry_start到MPENTRY_PADDR建立映射，将mpentry_start+offset地址转为MPENTRY_PADDR+offset地址。

在boot/boot.S中，没有启用分页机制，因此我们可以指定程序的启动位置和加载位置，代码最初加载在可寻址实模式的位置。但在kern/mpentry.S中，BSP一直处于保护模式，我们不能直接指定物理地址，所以应该给出线性地址并映射到相应的物理地址。

2.3 作业 2

2.3.1 题目原文

```
Modify mem_init_mp() (in kern/pmap.c) to map per-CPU stacks starting at KSTACKTOP, as shown in our revised 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().
```

2.3.2 题目大意

修改位于kern/pmap.c中的mem_init_mp()，将每个CPU堆栈映射在KSTACKTOP开始的区域，就像inc/memlayout.h中描述的那样。每个堆栈的大小都是KSTKSIZE字节，加上KSTKGAP字节没有被映射的守护页。现在，代码应当能够通过新的check_kern_pgdir()测试了。

2.3.3 回答

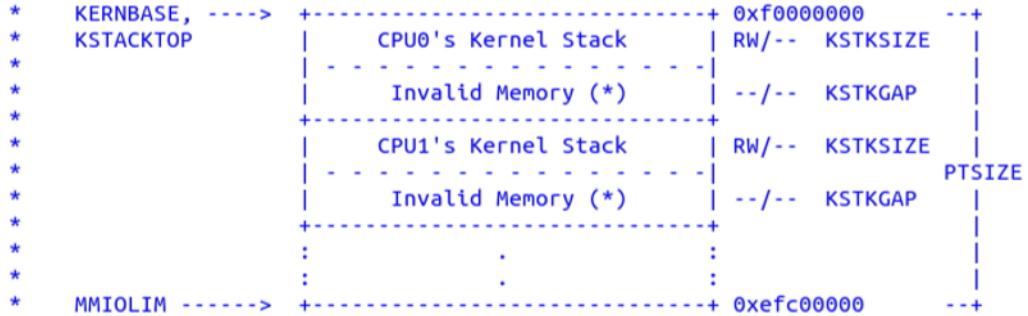


Figure 10: inc/memlayout.h

`mem_init_mp()`函数设置每个CPU的内核栈。各CPU从`KSTACKTOP`地址开始，依次向下进行地址的映射，把相对应的地址作为CPU的内核栈。每个内核栈的大小是`KSTKSIZE`，而内核栈之间的间距是`KSTKGAP`，起到保护作用。

因此，根据注释内的提示可在`kern/pmap.c`中修改`mem_init_mp()`函数：

```

// 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 = 0;
    uintptr_t kstacktop_i;

    for (i = 0; i < NCPU; i++)
    {
        kstacktop_i = KSTACKTOP - i * (KSTKSIZE + KSTKGAP);
        boot_map_region(kern_pgdir, kstacktop_i - KSTKSIZE, ROUNDUP(KSTKSIZE, PGSIZE), PADDR(&percpu_kstacks[i]), PTE_W);
    }
}

```

Figure 11: kern/pmap.c 中修改 mem_init_mp() 函数

`percpu_kstacks[i]`物理内存地址是 CPU*i* 内核栈的物理地址。`kstacktop_i`为 CPU*i* 的内核栈的虚拟地址。NCPU 为 CPU 数量。`KSTKSIZE` 为每个内核栈的大小。`KSTKGAP` 为

内核栈之间的距离。其中NCPU的定义都可以在kern/cpu.h中找到，percpu_kstacks的定义在kern/mpconfig.c中可以找到。

```
// Per-CPU kernel stacks  
unsigned char percpu_kstacks[NCPUS][KSTKSIZEx  
__attribute__((aligned(PGSIZE)));
```

Figure 12: percpu_kstacks 的定义

```
// Maximum number of CPUs  
#define NCPU 8
```

Figure 13: NCPU 的定义

输入make qemu的测试结果如下,这次解决了作业1的断言错误,通过了check_kern_pgdir()测试:

```
user@pc: ~/oslab/lab1/src/lab4
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
map region size = 4194304, 1024 pages
map region size = 126976, 31 pages
map region size = 32768, 8 pages
map region size = 268435455, 65535 pages
map region size = 32768, 8 pages
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 1 CPU(s)
map region size = 4096, 1 pages
enabled interrupts: 1 2
[00000000] new env 000001000
kernel panic on CPU 0 at kern/trap.c:331: page_fault in kernel mode, fault address 0x00000000

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

Figure 14: make qemu

2.4 作业 3

2.4.1 题目原文

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.)

2.4.2 题目大意

位于 `kern/trap.c` 中的 `trap_init_percpu()` 为 BSP 初始化了 TSS 和 TSS 描述符，它在 Lab 3 中可以工作，但是在其他 CPU 上运行时，它是不正确的。修改这段代码使得它能够在所有 CPU 上正确执行。（注意：你的代码不应该再使用全局变量 `ts`。）

2.4.3 回答

内核栈的栈地址存储在 TSS 中，而每个 CPU 都有一个独立的 TSS。初始化每个 CPU 的 TSS，并向 GDT 中加入相应条目。进程在进入中断时，栈指针能够通过对称的 CPU 的 TSS 寄存器把栈转换为内核栈。

在 `inc/memlayout.h` 中可以找到 `GD_TSS0` 的定义为 `0x28`。需要确定其他 CPU 的 TSS，偏移为 `cpu_id << 3` 即可。

根据注释中的提示并仿照原来的代码可在 `kern/trap.c` 中修改 `trap_init_percpu()` 函数：

```

// 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:
    struct Taskstate* this_ts = &thiscpu->cpu_ts;

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

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

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

}

```

Figure 15: kern/trap.c 中的 trap_init_percpu() 函数

其中 `cpunum()` 与 `thiscpu->cpu_id` 均为获取当前 CPU 的 ID (在 `kern/cpu.h` 中可以找到)。`thiscpu->cpu_ts` 为当前 CPU 的 TSS。`gdt[(GD_TSS0 » 3) + i]` 为 CPU*i* 的 TSS 描述符。`ts_esp0` 指向当前 CPU 的内核堆栈。`ts_ss0` 指向当前 CPU 的堆栈段寄存器。

```

int cpunum(void);
#define thiscpu (&cpus[cpunum()])

```

Figure 16: kern/cpu.h

```
// Per-CPU state
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
};
```

Figure 17: kern/cpu.h 的 CpuInfo 的定义

输入`make qemu CPUS=4`的结果如下，可以看到 4 个 CPU 都被启动了：

```
user@pc: ~/oslab/lab1/src/lab4
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
map region size = 32768, 8 pages
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 4 CPU(s)
map region size = 4096, 1 pages
enabled interrupts: 1 2
SMP: CPU 1 starting
map region size = 4096, 1 pages
SMP: CPU 2 starting
map region size = 4096, 1 pages
SMP: CPU 3 starting
map region size = 4096, 1 pages
[00000000] new env 00001000
kernel panic on CPU 0 at kern/trap.c:334: page_fault in kernel mode, fault address 0x00000000

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

Figure 18: make qemu CPUS=4

2.5 作业 4

2.5.1 题目原文

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

2.5.2 题目大意

在上述提到的位置使用内核锁，加锁时使用 `lock_kernel()`，释放锁时使用 `unlock_kernel()`。

2.5.3 JOS 的大内核锁

一般在单处理器操作系统中会采用大内核锁，就是当一个 CPU 需要进入内核的时候，必须获取整个内核的锁。这样的话，所有的 CPU 可以并行的跑用户程序，但是内核程序最多只能有一个在跑。更好的设计是给进程表的每个条目以及其他可能出现竞争的变量单独的加锁，这样可以在内核上实现更高的并行度，但是会大幅度增加设计的复杂性，比如说 XV6 就使用了不止一个锁。

在 `kern/spinlock.*` 中实现了 `lock_kernel()` 和 `unlock_kernel()` 函数，但凡是进入到内核临界区之前都需要加锁，离开内核临界区之后需要尽快释放锁。

`lock_kernel()` 和 `unlock_kernel()` 的调用关系图如下：

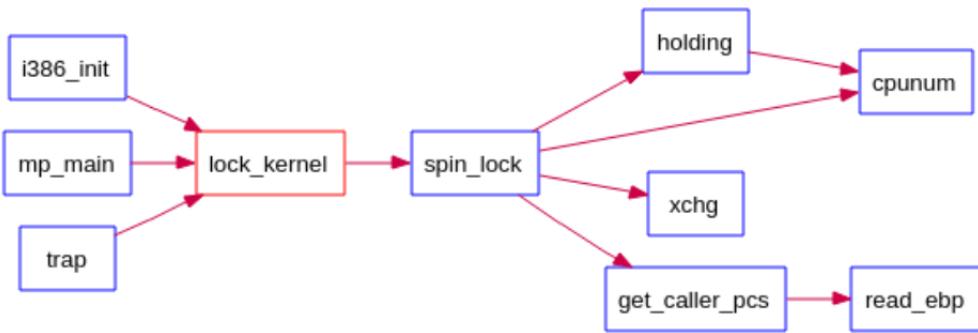


Figure 19: `lock_kernel()` 的调用关系图

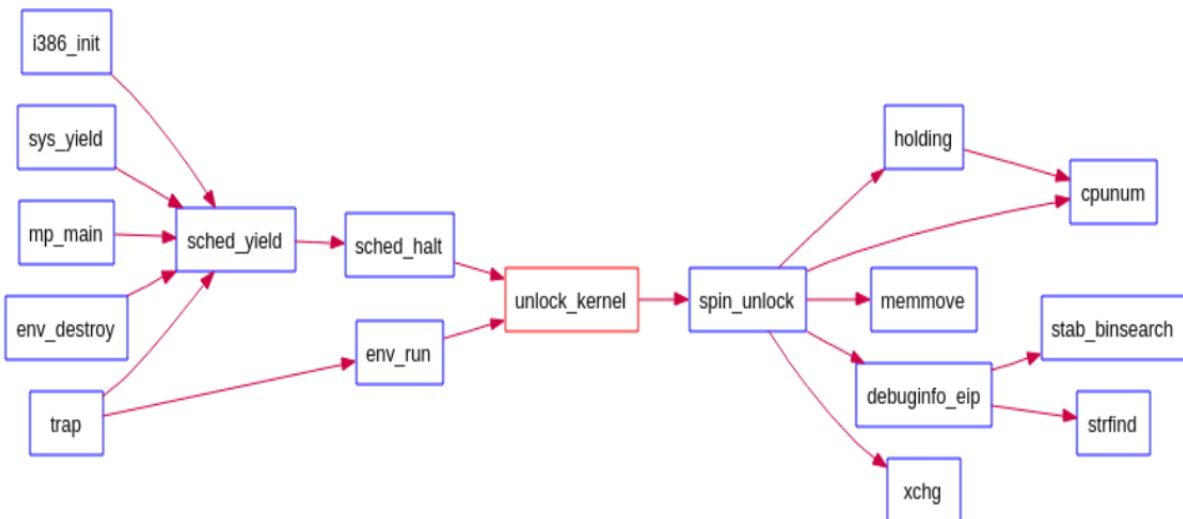


Figure 20: `unlock_kernel()` 的调用关系图

`lock_kernel()` 和 `unlock_kernel()` 的函数定义如下，这两个都是静态内联函数，本质上都是一对自旋锁：

```

#define spin_initlock(lock)    __spin_initlock(lock, #lock)
extern struct spinlock kernel_lock;

static inline void
lock_kernel(void)
{
    spin_lock(&kernel_lock);
}

static inline void
unlock_kernel(void)
{
    spin_unlock(&kernel_lock);

    // Normally we wouldn't need to do this, but QEMU only runs
    // one CPU at a time and has a long time-slice. Without the
    // pause, this CPU is likely to reacquire the lock before
    // another CPU has even been given a chance to acquire it.
    asm volatile("pause");
}

#endif

```

Figure 21: kern/spinlock.h 的 lock_kernel() 和 unlock_kernel()

在kern/spinlock.c中，自旋锁spin_lock()和spin_unlock()函数的定义如下。可以看出spin_lock()是通过循环来测试锁的状态、实现锁的机制（而不是阻塞），会造成忙等待，是会耗CPU资源的；spin_lock()和spin_unlock()都是通过xchg()函数完成的。

```

// 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
}

```

Figure 22: kern/spinlock.c 的 spin_lock() 函数

```

// 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 serializes, so that reads before release are
// not reordered after it. The 1996 PentiumPro manual (Volume 3,
// 7.2) says reads can be carried out speculatively and in
// any order, which implies we need to serialize here.
// But the 2007 Intel 64 Architecture Memory Ordering White
// Paper says that Intel 64 and IA-32 will not move a load
// after a store. So lock->locked = 0 would work here.
// The xchg being asm volatile ensures gcc emits it after
// the above assignments (and after the critical section).
xchg(&lk->locked, 0);
}

```

Figure 23: kern/spinlock.c 的 spin_unlock() 函数

其中自旋锁结构体 `spinlock` 的定义如下，变量 `locked` 表明是否持有锁（1 为持有锁，0 为不持有锁）。

```

// Mutual exclusion lock.
struct spinlock {
    unsigned locked;           // Is the lock held?

#ifdef DEBUG_SPINLOCK
    // For debugging:
    char *name;               // Name of lock.
    struct CpuInfo *cpu;      // The CPU holding the lock.
    uintptr_t pcs[10];         // The call stack (an array of program counters)
                               // that locked the lock.
#endif
};

```

Figure 24: kern/spinlock.h 的 spinlock 结构体定义

`xchg()` 函数在 `inc/x86.h` 中定义，这是一个原子操作。它可以原子地设置新值并返回原值，以此来判断获取或释放锁是否成功：

```

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

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

```

Figure 25: inc/x86.h 的 xchg() 函数

2.5.4 第一处

在启动的时候，BSP 启动其余的 CPU 之前，BSP 需要取得内核锁。

所以在 `kern/init.c` 下的 `i386_init()` 中添加代码如下：

```

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

```

Figure 26: kern/init.c 的 i386_init() 函数

2.5.5 第二处

在 `mp_main()`，也就是 CPU 被启动之后执行的第一个函数中，这里调用了调度函数，选择一个进程来执行的，但是在执行调度函数之前，也就是进入内核临界区的时候必须获取锁。

在 `kern/init.c` 下的 `mp_main()` 中添加代码如下：

```

// 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:
    lock_kernel();
    sched_yield();

    // Remove this after you finish Exercise 4
    //for (;;);
}

```

Figure 27: kern/init.c 的 mp_main() 函数

2.5.6 第三处

在`trap()`函数中，因为可以访问临界区的CPU只能有一个，所以从用户态陷入到内核态的话，要加锁，因为可能多个CPU同时陷入内核态。

在`kern/trap.c`下的`trap()`中添加代码如下：

```
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);
```

Figure 28: kern/trap.c 的 trap() 函数

2.5.7 第四处

在`env_run()`函数中，也就是启动进程的函数，在`env_pop_tf()`函数执行结束之后，就将跳回到用户态，此时离开内核，需要将内核锁释放。

在`kern/env.c`下的`env_run()`中添加代码如下：

```
// Hint: This function loads the new environment's state from
//        e->env_tf. Go back through the code you wrote above
//        and make sure you have set the relevant parts of
//        e->env_tf to sensible values.

// LAB 3: Your code here.
if(curenv != NULL && curenv->env_status == ENV_RUNNING)
    curenv->env_status = ENV_RUNNABLE;

curenv = e;
curenv->env_status = ENV_RUNNING;
curenv->env_runs++;
lcr3(PADDR(curenv->env_pgdir));

unlock_kernel();
env_pop_tf(&curenv->env_tf);

panic("env_run not yet implemented");
```

Figure 29: kern/env.c 的 env_run() 函数

2.6 问题 2

2.6.1 题目原文

It seems that using the big kernel lock guarantees that only one CPU can run the kernel code at a time. Why do we still need separate kernel stacks for each CPU? Describe a scenario in which using a shared kernel stack will go wrong, even with the protection of the big kernel lock.

2.6.2 题目大意

看起来使用全局内核锁能够保证同一时段内只有一个 CPU 能够运行内核代码。既然这样，为什么还需要为每个 CPU 分配不同的内核堆栈呢？请描述一个即使使用了全局内核锁，共享内核堆栈仍会导致错误的情形。

2.6.3 回答

因为不同的内核栈上可能保存有不同的信息，在一个 CPU 从内核退出来之后，有可能在内核栈中留下了一些将来还有用的数据，所以一定要有单独的栈。比如在 `_alltraps` 到 `lock_kernel()` 的过程中，进程已经切换到了内核态，但并没有上内核锁，如果此时有其他 CPU 进入内核，如果使用同一个内核栈，则 `_alltraps` 中保存的上下文信息会被破坏，所以即使有大内核栈，CPU 也不能用同一个内核栈。同样的，解锁也是在内核态解锁，在解锁到真正返回用户态的过程中，也存在上述问题。

2.7 作业 5A

2.7.1 题目原文

```
Implement round-robin scheduling in sched_yield() as described above. Don't
forget to modify syscall() to dispatch sys_yield().
Modify kern/init.c to create three (or more!) environments that all run the
program user/yield.c. You should see the environments switch back and
forth between each other five times before terminating, like this:

...
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, when only idle environments are runnable,
the scheduler should invoke the JOS kernel monitor. If any of this does
not happen, then fix your code before proceeding.
```

2.7.2 题目大意

按照以上描述，实现 `sched_yield()` 轮转算法。不要忘记修改 `syscall()` 将相应的系统调用分发至 `sys_yield()`（以后还要添加新的系统调用，同样不要忘记修改 `sys_yield()`）。

确保在 `mp_main` 中调用了 `sched_yield()`。

修改 `kern/init.c` 创建三个或更多进程，运行 `user/yield.c`。运行 `make qemu`。你应当看到进程在退出之前会在彼此之间来回切换 5 次，就像下面这样：

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

在 `yield` 测试程序退出后，系统中没有其他运行的进程了，调度器应当调用 JOS 的内核监视器 (kernel monitor)。如果这些没有发生，你应当在继续之前检查你的代码。

2.7.3 回答

首先了解 JOS 的轮转调度算法工作机制。

`kern/sched.c` 中的 `sched_yield()` 函数负责挑选一个进程运行。它从刚刚在运行的进程开始，按顺序循环搜索 `envs[]` 数组（如果从来没有运行过进程，那么就从数组起点开始搜索），选择它遇到的第一个处于 `ENV_RUNNABLE`（参考 `inc/env.h`）状态的进程，并调用 `env_run()` 来运行它。

`sched_yield()` 绝不应当在两个 CPU 上同时运行同一进程。它可以分辨出一个进程正在其他 CPU（或者就在当前 CPU）上运行，因为这样的进程处于 `ENV_RUNNING` 状态。用户进程可以使用这个系统调用 `sys_yield()` 来触发内核的 `sched_yield()` 方法，自愿放弃 CPU，给其他进程运行。

根据注释中的提示在kern/sched.c中的sched_yield()函数中实现轮询调度如下：

```

// 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 = curenv;
    size_t idx = idle!=NULL ? ENVX(idle->env_id):-1;
    size_t i;
    for (i=0; i<NENV; i++) {
        idx = (idx+1 == NENV) ? 0:idx+1;
        if (envs[idx].env_status == ENV_RUNNABLE) {
            env_run(&envs[idx]);
            return;
        }
    }
    if (idle && idle->env_status == ENV_RUNNING) {
        env_run(idle);
        return;
    }

    // sched_halt never returns
    sched_halt();
}

```

Figure 30: kern/sched.c 的 sched_yield() 函数

然后需要在kern/syscall.c中增加这个系统调用的分支：

```

1 // Dispatches to the correct kernel function, passing the arguments.
2 int32_t
3 syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t
4         a4, uint32_t a5)
5 {
6     // Call the function corresponding to the 'syscallno' parameter.
7     // Return any appropriate return value.
8     // LAB 3: Your code here.
9
10    //panic("syscall not implemented");
11    switch (syscallno) {
12        case SYS_cputs:
13            sys_cputs((const char *)a1, a2);

```

```

13         return 0;
14     case SYS_cgetc:
15         return sys_cgetc();
16     case SYS_getenvid:
17         return sys_getenvid();
18     case SYS_env_destroy:
19         return sys_env_destroy(a1);
20     case SYS_yield:
21         sys_yield();
22         return 0;
23     default:
24         return -E_INVAL;
25     }
26 }
```

最后在kern/init.c中的i386_init()的用户进程修改如下（创建3个yield的用户进程，这样输入命令make qemu就可以运行在这里声明的用户进程）：

```

25 #if defined(TEST)
26 // Don't touch -- used by grading script!
27 ENV_CREATE(TEST, ENV_TYPE_USER);
28 #else
29 // Touch all you want.
30 ENV_CREATE(user_yield, ENV_TYPE_USER);
31 ENV_CREATE(user_yield, ENV_TYPE_USER);
32 ENV_CREATE(user_yield, ENV_TYPE_USER);
33 //ENV_CREATE(user_primes, ENV_TYPE_USER);
34 #endif // TEST*
35 // Schedule and run the first user environment!
36 sched_yield();
```

Figure 31: kern/sched.c 的 i386_init() 函数

输入命令make qemu CPUS=2可以看到进程通过调用sys_yield()切换了5次。其中的断言错误稍后会解决。

```

user@pc: ~/oslab/lab1/src/lab4
[00000000] new env 00001000
[00000000] new env 00001001
[00000000] new env 00001002
Hello, I am environment 00001000.
Hello, I am environment 00001001.
Back in environment 00001000, iteration 0.
Back in environment 00001001, iteration 0.
Back in environment 00001000, iteration 1.
Back in environment 00001001, iteration 1.
Back in environment 00001000, iteration 2.
Back in environment 00001001, iteration 2.
Back in environment 00001000, iteration 3.
Back in environment 00001001, iteration 3.
Back in environment 00001000, iteration 4.
Back in environment 00001001, iteration 4.
All done in environment 00001001.
All done in environment 00001000.
[00001001] exiting gracefully
[00001001] free env 00001001
[00001000] exiting gracefully
[00001000] free env 00001000
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.
kernel panic on CPU 0 at kern/trap.c:281: assertion failed: !(read_eflags() & FL_IF)
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.

```

Figure 32: make qemu CPUS=2

2.8 问题 3

2.8.1 题目原文

In your implementation of `env_run()` you should have called `lcr3()`. Before and after the call to `lcr3()`, your code makes references (at least it should) to the variable `e`, the argument to `env_run`. Upon loading the %cr3 register, the addressing context used by the MMU is instantly changed. But a virtual address (namely `e`) has meaning relative to a given address context--the address context specifies the physical address to which the virtual address maps. Why can the pointer `e` be dereferenced both before and after the addressing switch?

2.8.2 题目大意

`lcr3()`在 `env_run()` 中会被调用。在调用 `lcr3()` 之前和之后，你的代码应当都在引用变量 `e`，就是 `env_run()` 所需要的参数。在装载 `%cr3` 寄存器之后，MMU 使用

的地址上下文立刻发生改变，但是处在之前地址上下文的虚拟地址（比如说 **e**）却还能够正常工作，为什么 **e** 在地址切换前后都可以被正确地解引用呢？

2.8.3 回答

%cr3寄存器存储了一个进程的页目录。`lcr3(next->cr3)`表示加载下一个调度的进程**next**的页目录的首地址。

在 Lab 3 中，曾经写过的函数 `env_setup_vm()`。进程的页目录是直接拷贝了内核的页目录 (`memmove`)，这确保了两个页目录的 **e** 在内核部分是映射到了同一块物理内存，所以**e** 在地址切换前后都可以被正确地解引用。至于做这样处理的原因已在 Lab 3 中解释过，不再重复解释。

```
// LAB 3: Your code here.
e->env_pgdir = (pde_t*)page2kva(p);
p->pp_ref++;
memmove(e->env_pgdir, kern_pgdir, PGSIZE);

// UVPT maps the env's own page table read-only.
// Permissions: kernel R, user R
e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_P | PTE_U;
```

Figure 33: kern/env.c 的 `env_setup_vm()`

2.9 作业 6

2.9.1 题目原文

Implement the system calls described above in kern/syscall.c. 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.

2.9.2 题目大意

在 `kern/syscall.c` 中实现上面描述的系统调用。`kern/pmap.c` 和 `kern/env.c` 中定义的多个函数将会被用到，尤其是 `envid2env()`。此时，无论何时你调用 `envid2env()`，都应该传递 1 给 `checkperm` 参数。确定你检查了每个系统调用参数均合法，否则返回 `-E_INVAL`。用 `user/dumbfork` 来测试你的 JOS 内核，在继续前确定它正常的工作。（`make run-dumbfork`）

2.9.3 需要实现的系统调用

根据实验文档,作业6需要实现类似`fork()`函数的系统调用(均在`kern/syscall.c`中),这些系统调用将`fork()`的功能进行了细分。这些系统调用分别如下:

- `sys_exofork`: 创建一个空白的新进程,它的用户地址空间中没有物理内存建立任何映射关系,也不能运行。该函数的调用者为父进程,父进程调用此函数返回子进程的`envid_t`(也可能是负值错误码)。该函数产生的新进程为子进程,子进程调用此函数将返回0。在开始时父进程和子进程具有相同的寄存器状态。
- `sys_env_set_status`: 为指定的进程设置状态,可以是`ENV_RUNNABLE`或`ENV_NO_T_RUNNABLE`。这个系统调用通常用来在新创建的进程的地址空间和寄存器状态已经初始化完毕后将它标记为就绪态。
- `sys_page_alloc`: 分配一个物理页,并将其映射到指定进程的指定虚拟地址上。
- `sys_page_map`: 从一个进程的地址空间拷贝一个页的映射(不是页的内容)到另一个进程的地址空间。这样新进程和旧进程的映射指向同一块物理内存区域,即实现两个进程共享内存。
- `sys_page_unmap`: 取消给定进程在给定虚拟地址的页映射。

`user/dumbfork.c`给出了一个原始的`fork()`实现,可作为实现这些系统调用的参考。

在`user/dumbfork.c`中, `duppage()`函数使用了这些系统调用, `dumbfork()`实现了与`fork()`类似的过程, `umain()`是主函数。

`duppage()`函数如下图所示:

```
void
duppage(envid_t dstenv, void *addr)
{
    int r;

    // This is NOT what you should do in your fork.
    if ((r = sys_page_alloc(dstenv, addr, PTE_P|PTE_U|PTE_W)) < 0)
        panic("sys_page_alloc: %e", r);
    if ((r = sys_page_map(dstenv, addr, 0, UTEMP, PTE_P|PTE_U|PTE_W)) < 0)
        panic("sys_page_map: %e", r);
    memmove(UTEMP, addr, PGSIZE);
    if ((r = sys_page_unmap(0, UTEMP)) < 0)
        panic("sys_page_unmap: %e", r);
}
```

Figure 34: `user/dumbfork.c` 的 `duppage()` 函数

该函数主要分为四个步骤:为子进程分配空闲物理页(`sys_page_alloc()`)、将被分配的新的物理页映射到`UTEMP`中(`sys_page_map()`)、向`UTEMP`拷贝一个物理页的内容、删除这个页映射(`sys_page_unmap()`)。

2.9.4 sys_exofork()

该函数的注释如下图:

```
// 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.
```

Figure 35: kern/syscall.c 的 sys_exofork() 函数的注释

该函数用来分配新的进程。如果函数执行成功则返回该进程的envid，否则返回小于0的值: -E_NO_FREE_ENV表示没有足够的进程可分配，-E_NO_MEM表示没有足够空间可用。

可以使用kern/env.c中的env_alloc()函数来分配新的进程。新进程的状态要设置为ENV_NOT_RUNNABLE，调用者的寄存器要拷贝给这个新的进程，唯一的例外是eax寄存器要传入0值，代表子进程返回0(类似于函数返回值的传递)。

根据上述提示，sys_exofork()函数可实现如下：

```
1 static envid_t
2 sys_exofork(void)
3 {
4     // LAB 4: Your code here.
5     // panic("sys_exofork not implemented");
6     struct Env *e;
7     int r = env_alloc(&e, curenv->env_id); // 分配一个新进程
8     if (r < 0) return r;
9     e->env_status = ENV_NOT_RUNNABLE; // 未就绪
10    e->env_tf = curenv->env_tf; // 子进程复制父进程的 trapframe
11    // 除了eax以外，子进程的寄存器值是拷贝父进程的
12    e->env_tf.tf_regs.reg_eax = 0; // 将子进程的eax置0，相当于让子进程返回0
13    return e->env_id; // 返回子进程的id
14 }
```

2.9.5 sys_page_set_status()

该函数的注释如下图:

```
// 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.
```

Figure 36: kern/syscall.c 的 sys_page_set_status() 函数的注释

注释的意思是,给由`envid`指定的进程的状态只能是`ENV_RUNNABLE`和`ENV_NOT_RUNNABLE`其中一个。如果函数执行成功则返回 0, 否则返回小于 0 的值: 如果该`envid`不存在或者调用者没有权限去修改该`envid`所对应的进程时则返回`-E_BAD_ENV`。如果要设置的状态`status`是非法状态 (不是那两个状态的其中一个), 则返回`-E_INVAL`。

定义在kern/env.c的envid2env()函数可以实现envid到Env结构体变量的转换(定义如下图), 在调用这个函数的时候必须要将传的第 3 个参数置为 1, 这样就可以检查当前进程是否有权限去设置`envid`指定进程的状态。

```

/*
// Converts an envid to an env pointer.
// If checkperm is set, the specified environment must be either the
// current environment or an immediate child of the current environment.
//
// RETURNS
//   0 on success, -E_BAD_ENV on error.
//   On success, sets *env_store to the environment.
//   On error, sets *env_store to NULL.
//
int
envid2env(envid_t envid, struct Env **env_store, bool checkperm)
{
    struct Env *e;

    // If envid is zero, return the current environment.
    if (envid == 0) {
        *env_store = curenv;
        return 0;
    }

    // Look up the Env structure via the index part of the envid,
    // then check the env_id field in that struct Env
    // to ensure that the envid is not stale
    // (i.e., does not refer to a _previous_ environment
    // that used the same slot in the envs[] array).
    e = &envs[ENVX(envid)];
    if (e->env_status == ENV_FREE || e->env_id != envid) {
        *env_store = 0;
        return -E_BAD_ENV;
    }

    // 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;
    }

    *env_store = e;
    return 0;
}

```

Figure 37: kern/env.c 的 envid2env() 函数

根据上述提示, sys_page_set_status()函数可实现如下:

```

1 static int
2 sys_env_set_status(envid_t envid, int status)
3 {
4     // LAB 4: Your code here.
5     // panic("sys_env_set_status not implemented");
6
7     if (status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE)
8         return -E_INVAL; //非法状态
9     struct Env *e;
10    if (envid2env(envid, &e, 1) < 0) return -E_BAD_ENV; //第二个参数为返回
11    e->env_status = status; //设置状态
12    return 0;
13 }

```

2.9.6 sys_page_alloc()

该函数的注释如下图:

```
// 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/mmuh.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.
}
```

Figure 38: kern/syscall.c 的 sys_page_alloc() 函数的注释

该函数要实现的功能是分配一个物理页，并将其分配到va指定的虚拟地址上，并设置相应的权限位。如果已经存在物理页与va的映射关系。如果函数执行成功则返回 0，不成功则返回负值：-E_BAD_ENV含义同上，-E_INVAL表示虚拟地址的范围不合法（达到UTOP）或该虚拟地址不能被映射或者设置的权限不合法（规则如上图），-E_NO_MEM表示没有足够的空间去分配新的物理页。

该函数主要实现思路是调用page_alloc()和page_insert(),如果page_insert()调用失败（映射关系建立失败），则要释放这个物理页（使用page_free()）。除此之外，该函数主要部分都是为了检查参数的合法性。

根据上述提示，sys_page_alloc()函数可实现如下：

```
1 static int
2 sys_page_alloc(envid_t envid, void *va, int perm)
3 {
4     // LAB 4: Your code here.
5     // panic("sys_page_alloc not implemented");
6     // 检查权限的合法性
7     if ((~perm & (PTE_U|PTE_P)) != 0)
```

```

8     return -E_INVAL;
9     if ((perm & (~(PTE_U|PTE_P|PTE_AVAIL|PTE_W))) != 0)
10    return -E_INVAL;
11 //检查给出的虚拟地址是否超界、是否页对齐
12 if ((uintptr_t)va >= UTOP || PGOFF(va) != 0)
13    return -E_INVAL;
14
15 struct PageInfo *pginfo = page_alloc(ALLOC_ZERO); //分配新的物理页
16 if (!pginfo) return -E_NO_MEM; //如果物理页没分配成功，则返回小于0的值-E_NO_MEM
17 struct Env *e;
18 int r = envid2env(envid, &e, 1);
19 if (r < 0) return -E_BAD_ENV;
20 r = page_insert(e->env_pgd, pginfo, va, perm); //建立物理页与虚拟地址的映射
21 if (r < 0) { //如果映射没建立成功，就释放这个物理页
22     page_free(pginfo);
23     return -E_NO_MEM;
24 }
25 return 0;
26 }

```

2.9.7 sys_page_map()

该函数的注释如下图：

```

// 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 if 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.

```

Figure 39: kern/syscall.c 的 sys_page_map() 函数的注释

该函数要实现的功能是将源虚拟地址的映射拷贝给目的虚拟地址，示意图如下。并设置相应的权限。如果函数执行成功，则返回 0，否则返回小于 0 的值：-E_BAD_ENV 和 -E_NO_MEM 的意义同上，如果源或目的虚拟地址超过范围或没有页对齐，或要设置的权限不合法（见 sys_page_alloc() 的注释），或欲给只读的虚拟地址设置写权限，或者源虚拟地址没有被映射到源进程（即调用者）的 envid 的地址空间，都会返回 -E_INVAL。

该函数主要实现思路是调用 page_lookup() 和 page_insert()。除此之外，该函数主要部分都是为了检查参数的合法性。调用 page_lookup() 的第三个参数需要用来检查要查找的物理页的权限。

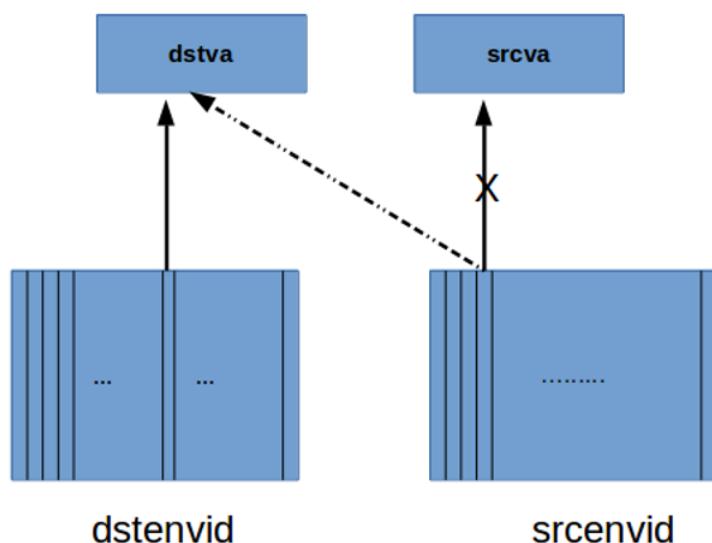


Figure 40: sys_page_map() 的示意图

根据上述提示，sys_page_map() 函数可实现如下：

```

1 static int
2 sys_page_map(envid_t srcenvid, void *srcvva,
3               envid_t dstenvid, void *dstvva, int perm)
4 {
5     // LAB 4: Your code here.
6     // panic("sys_page_map not implemented");
7
8     // 先检查由参数给出的虚拟地址是否超界、是否页对齐
9     if ((uintptr_t)srcvva >= UTOP || PGOFF(srcvva) != 0)
10        return -E_INVAL;
11     if ((uintptr_t)dstvva >= UTOP || PGOFF(dstvva) != 0)
12        return -E_INVAL;
13     // 权限合法性检查

```

```

14     if (((perm & PTE_U) == 0 || (perm & PTE_P) == 0 || (perm & ~PTE_SYSCALL)
15     != 0)
16         return -EINVAL;
17     struct Env *src_e, *dst_e;
18     //envid是否有对应的进程
19     if (envid2env(srcenvid, &src_e, 1) < 0 || envid2env(dstenvid, &dst_e, 1)
20     < 0)
21         return -E_BAD_ENV;
22     pte_t *src_ptab;
23     //查找源进程和源虚拟地址的映射（页表项）
24     struct PageInfo *pp = page_lookup(src_e->env_pgdir, srcva, &src_ptab);
25     //检查查找到的页表项的权限位（是否给只读物理页赋予写权限）
26     if ((*src_ptab & PTE_W) == 0 && (perm & PTE_W) == 1)
27         return -EINVAL;
28     //将查找到的映射加入（拷贝）到目的进程的页目录中
29     if (page_insert(dst_e->env_pgdir, pp, dstva, perm) < 0)
30         return -E_NO_MEM;
31     return 0;
32 }
```

2.9.8 sys_page_unmap()

该函数的注释如下图：

```

// 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().
```

Figure 41: kern/syscall.c 的 sys_page_unmap() 函数的注释

该函数要实现的功能是解除虚拟地址 **va** 所对应的物理页与 **envid** 所指定的进程的地址空间的映射。如果没有相应的物理页映射，则该函数也会执行成功。如果函数执行成功，则返回 0，否则返回小于 0 的值：-E_BAD_ENV 和 -E_INVAL 的意义同上。该函数的主要功能可由 **page_free()** 实现。

根据上述提示，**sys_page_unmap()** 函数可实现如下：

```

1 static int
2 sys_page_unmap(envid_t envid, void *va)
```

```

3 {
4     // LAB 4: Your code here.
5     // panic("sys_page_unmap not implemented");
6     if ((uintptr_t)va >= UTOP || PGOFF(va) != 0) return -E_INVAL; //如果给出的虚拟地址超界或没有页对齐则返回-EINVAL
7     struct Env *e;
8     if (envid2env(envid, &e, 1) < 0) return -E_BAD_ENV; //如果找不到envid所对应的进程则返回-E_BAD_ENV
9     page_remove(e->env_pgd, va); //从该进程的页目录中删除映射关系
10    return 0;
11 }

```

2.9.9 增加系统调用分支

在kern/syscall.c中添加以上的系统调用分支:

```

1 // Dispatches to the correct kernel function, passing the arguments.
2 int32_t
3 syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t
4         a4, uint32_t a5)
5 {
6     // Call the function corresponding to the 'syscallno' parameter.
7     // Return any appropriate return value.
8     // LAB 3: Your code here.
9
10    //panic("syscall not implemented");
11    switch (syscallno) {
12        case SYS_cputs:
13            sys_cputs((const char *)a1, a2);
14            return 0;
15        case SYS_cgetc:
16            return sys_cgetc();
17        case SYS_getenvid:
18            return sys_getenvid();
19        case SYS_env_destroy:
20            return sys_env_destroy(a1);
21        case SYS_yield:
22            sys_yield();
23            return 0;
24        case SYS_exofork:
25            return (int32_t)sys_exofork();
26        case SYS_env_set_status:

```

```

26         return sys_env_set_status(a1, a2);
27     case SYS_page_alloc:
28         return sys_page_alloc(a1, (void *)a2, (int)a3);
29     case SYS_page_map:
30         return sys_page_map(a1, (void *)a2, a3, (void*)a4, (int)a5);
31     case SYS_page_unmap:
32         return sys_page_unmap(a1, (void *)a2);
33     default:
34         return -E_INVAL;
35     }
36 }
```

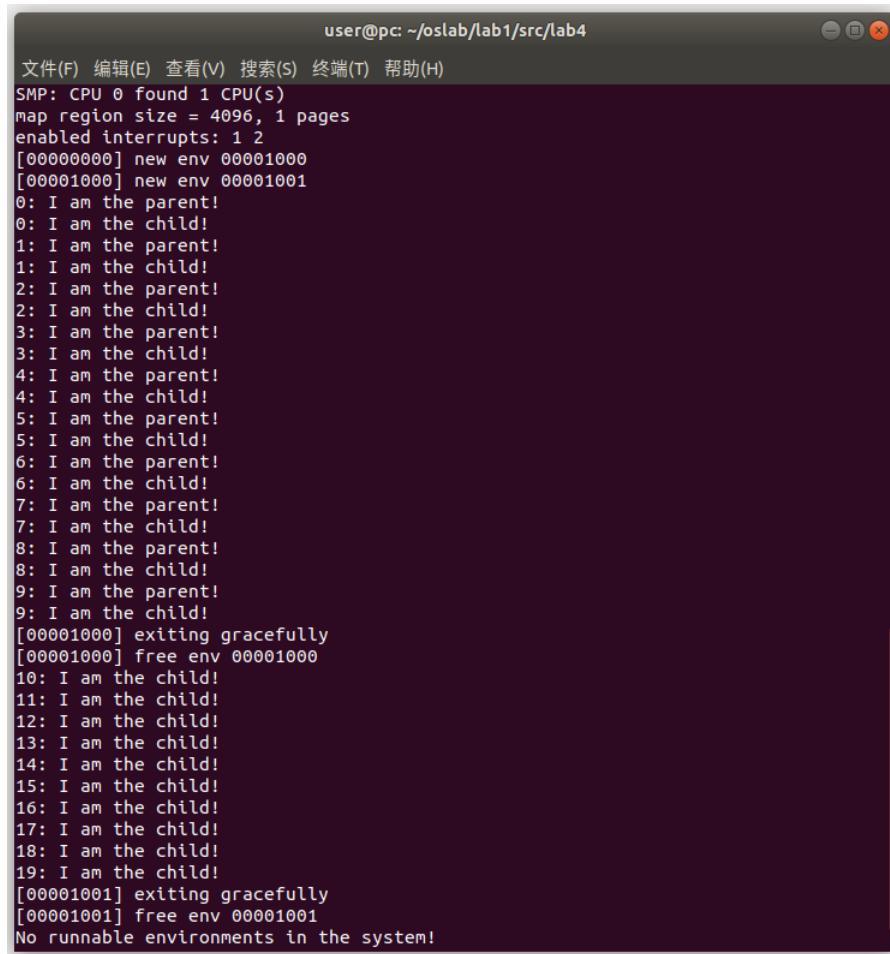
现在输入命令`make qemu CPUS=2`, 就可以看到作业 5A 中预期的结果了。

```

user@pc: ~/oslab/lab1/src/lab4
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
SMP: CPU 0 found 2 CPU(s)
map region size = 4096, 1 pages
enabled interrupts: 1 2
SMP: CPU 1 starting
map region size = 4096, 1 pages
[00000000] new env 00001000
[00000000] new env 00001001
[00000000] new env 00001002
Hello, I am environment 00001000.
Hello, I am environment 00001001.
Back in environment 00001000, iteration 0.
Hello, I am environment 00001002.
Back in environment 00001001, iteration 0.
Back in environment 00001000, iteration 1.
Back in environment 00001002, iteration 0.
Back in environment 00001001, iteration 1.
Back in environment 00001000, iteration 2.
Back in environment 00001002, iteration 1.
Back in environment 00001001, iteration 2.
Back in environment 00001000, iteration 3.
Back in environment 00001002, iteration 2.
Back in environment 00001001, iteration 3.
Back in environment 00001000, iteration 4.
Back in environment 00001002, iteration 3.
All done in environment 00001000.
[00001000] exiting gracefully
[00001000] free env 00001000
Back in environment 00001001, iteration 4.
Back in environment 00001002, iteration 4.
All done in environment 00001001.
All done in environment 00001002.
[00001001] exiting gracefully
[00001001] free env 00001001
[00001002] exiting gracefully
[00001002] free env 00001002
No runnable environments in the system!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> 
```

Figure 42: make qemu CPUS=2

输入命令`make run-dumbfork`也可以正常运行:

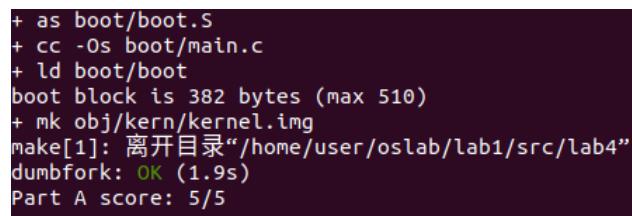


The terminal window shows the following output:

```
user@pc: ~/oslab/lab1/src/lab4
SMP: CPU 0 found 1 CPU(s)
map region size = 4096, 1 pages
enabled interrupts: 1 2
[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 parent!
3: I am the child!
4: I am the parent!
4: I am the child!
5: I am the parent!
5: I am the child!
6: I am the parent!
6: I am the child!
7: I am the parent!
7: I am the child!
8: I am the parent!
8: I am the child!
9: I am the parent!
9: I am the child!
[00001000] exiting gracefully
[00001000] free env 00001000
10: I am the child!
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!
```

Figure 43: make run-dumbfork

至此, Lab 4 的 Part A 已完成, 以下是在终端输入命令`make grade`得到的打分情况:



```
+ as boot/boot.s
+ cc -Os boot/main.c
+ ld boot/boot
boot block is 382 bytes (max 510)
+ mk obj/kern/kernel.img
make[1]: 离开目录“/home/user/oslab/lab1/src/lab4”
dumbfork: OK (1.9s)
Part A score: 5/5
```

Figure 44: Part A 打分情况

3 Part B: 写时复制的 Fork

3.1 背景知识

在做本部分之前需要先了解写入时复制（copy-on-write）机制。根据实验文档中的介绍，我们可以了解到，Unix 中父线程创建子进程会使用 `fork()` 函数，通常来说，子进程会调用 `exec()` 函数，使得子进程和父进程之间共享代码段的同时，会复制父进程的数据段，堆，栈到分配给子进程的新的内存区域。如下图所示：

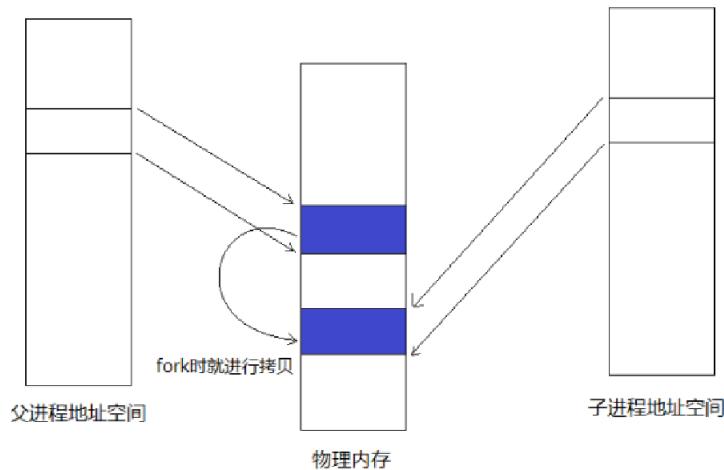


Figure 45: 非写时复制 fork

然而，上述复制过程会耗费大量的资源，因此引入写入时复制机制：`fork()` 函数执行时，子进程只会复制父进程的地址映射，即指向与父进程相同的空间，直到某个进程试图修改资源的内容时，程序会发生一个缺页错误（page fault），此时系统才会真正复制一份专用副本（private copy）给该进程，而其他调用者所见到的最初的资源仍然保持不变。此作法主要的优点是如果进程没有修改该内存页，就不会有新的内存页副本被创建，因此在多个进程只有读取操作时可以共享同一份资源，因此降低了子进程调用 `exec()` 产生的开销。如下图所示：

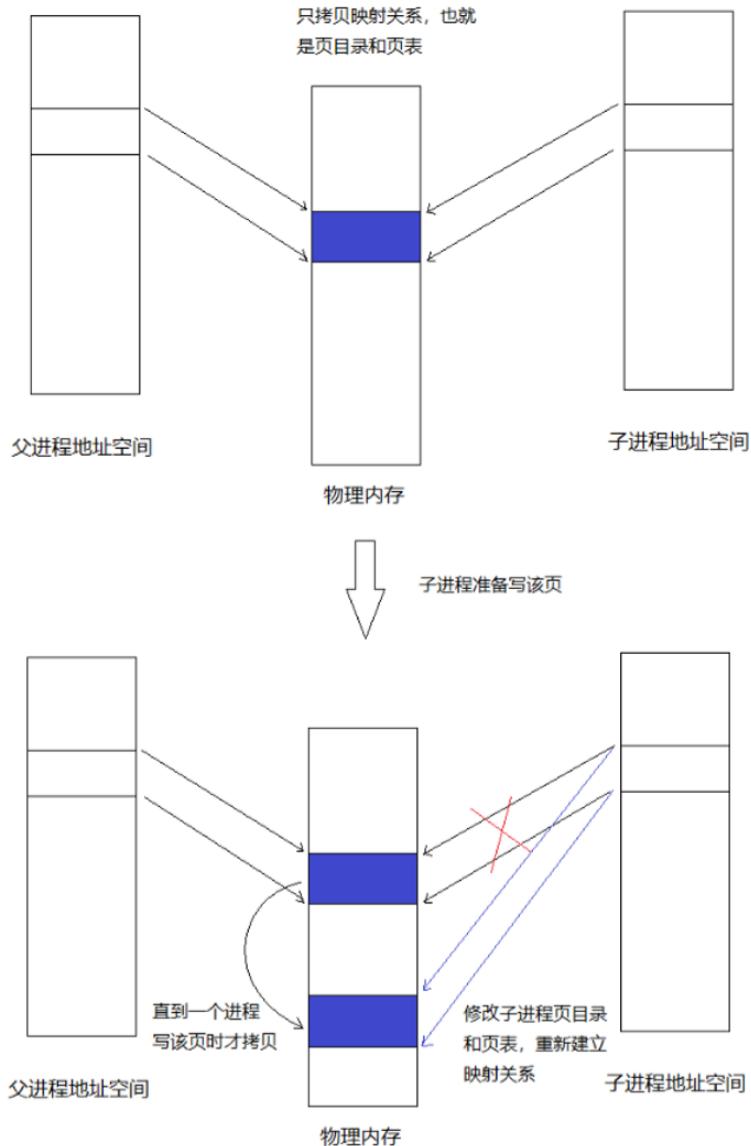


Figure 46: 写时复制 fork

3.2 作业 7

3.2.1 题目原文

实现系统调用 `sys_env_set_pgfault_upcall`。注意在寻找目标进程的 ID 时进行权限检查，因为这个系统调用是比较“危险”的。

3.2.2 函数的实现

该函数kern/syscall.c中。首先作业要求中提到，为了使用户自行处理缺页错误，进程需要在JOS内核中注册一个缺页处理函数的入口点。用户进程可以使用新的系统调用sys_env_set_pgfault_upcall来设置这个入口点。在Env结构中，已经添加了一个成员env_pgfault_upcall用来记录这个入口信息。

```
struct Env {
    struct Trapframe env_tf;           // Saved registers
    struct Env *env_link;              // Next free Env
    envid_t env_id;                   // Unique environment identifier
    envid_t env_parent_id;             // env_id of this env's parent
    enum EnvType env_type;             // Indicates special system environments
    unsigned env_status;               // Status of the environment
    uint32_t env_runs;                // Number of times environment has run
    int env_cpunum;                  // The CPU that the env is running on

    // Address space
    pde_t *env_pgdir;                 // Kernel virtual address of page dir

    // Exception handling
    void *env_pgfault_upcall;          // Page fault upcall entry point
```

Figure 47: inc/env.h 中 Env 结构体定义（部分）

下面给出了该函数的注释：

```
// 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.
```

Figure 48: kern/syscall.c 中函数 sys_env_set_pgfault_upcall() 的注释

可以看出，该系统调用主要是为了添加一个用户态的缺页处理函数，根据该函数的注释可以了解到，我们需要通过更改结构体Env中相应的env_pgfault_upcall区域，来为envid设置一个缺页处理系统调用，当envid发生缺页时，内核将调用函数func。因此函数补充如下：

```
1 static int
2 sys_env_set_pgfault_upcall(envid_t envid, void *func)
3 {
4     // LAB 4: Your code here.
5     //panic("sys_env_set_pgfault_upcall not implemented");
6     struct Env* e;
7     if(envid2env(envid,&e,1)<0){
8         return -E_BAD_ENV;
9     }
10    e->env_pgfault_upcall=func;
```

```

11     return 0;
12 }
```

3.2.3 增加系统调用分支

完成了该函数之后需要在 `kern/syscall.c` 增加新的系统调用分支:

```

1 // Dispatches to the correct kernel function, passing the arguments.
2 int32_t
3 syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t
4         a4, uint32_t a5)
5 {
6     // Call the function corresponding to the 'syscallno' parameter.
7     // Return any appropriate return value.
8     // LAB 3: Your code here.
9
10    //panic("syscall not implemented");
11    switch (syscallno) {
12        case SYS_cputs:
13            sys_cputs((const char *)a1, a2);
14            return 0;
15        case SYS_cgetc:
16            return sys_cgetc();
17        case SYS_getenvid:
18            return sys_getenvid();
19        case SYS_env_destroy:
20            return sys_env_destroy(a1);
21        case SYS_yield:
22            sys_yield();
23            return 0;
24        case SYS_exofork:
25            return (int32_t)sys_exofork();
26        case SYS_env_set_status:
27            return sys_env_set_status(a1, a2);
28        case SYS_page_alloc:
29            return sys_page_alloc(a1, (void *)a2, (int)a3);
30        case SYS_page_map:
31            return sys_page_map(a1, (void *)a2, a3, (void*)a4, (int)a5);
32        case SYS_page_unmap:
33            return sys_page_unmap(a1, (void *)a2);
34        case SYS_env_set_pgfault_upcall:
35            return sys_env_set_pgfault_upcall(a1, (void*)a2);
```

```
35     default:  
36         return -E_INVAL;  
37     }  
38 }
```

3.3 作业 8

3.3.1 题目原文

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?)

3.3.2 题目大意

实现在 `kern/trap.c` 中的 `page_fault_handler` 方法，使其能够将缺页分发给用户模式缺页处理函数。确认你在写入异常堆栈时已经采取足够的预防措施了。（如果用户进程的异常堆栈已经没有空间了会发生什么？）

3.3.3 回答

该函数在 `kern/trap.c` 中。

首先需要了解用户的正常栈和异常栈的区别：正常栈位于 `[USTACKTOP-PGSIZE , USTACKTOP-1]`，是用户进程正常运行情况下使用的栈，异常栈位于 `[UXSTACKTOP-PGSIZE , UXSTACKTOP-1]`，在用户模式下出现缺页错误时，内核就会在一个被称为用户异常栈（user exception stack）的堆栈上运行指定的用户级缺页处理函数。

下图显示了用户正常栈和异常栈在内存地址中所处的位置。而该图片也可以回答作业中的问题（What happens if the user environment runs out of space on the exception stack?），可以看到，用户异常栈的低地址部分为 `Empty Memory`，且该部分对于用户和内核来说都没有读写权限，因此会造成程序执行异常。

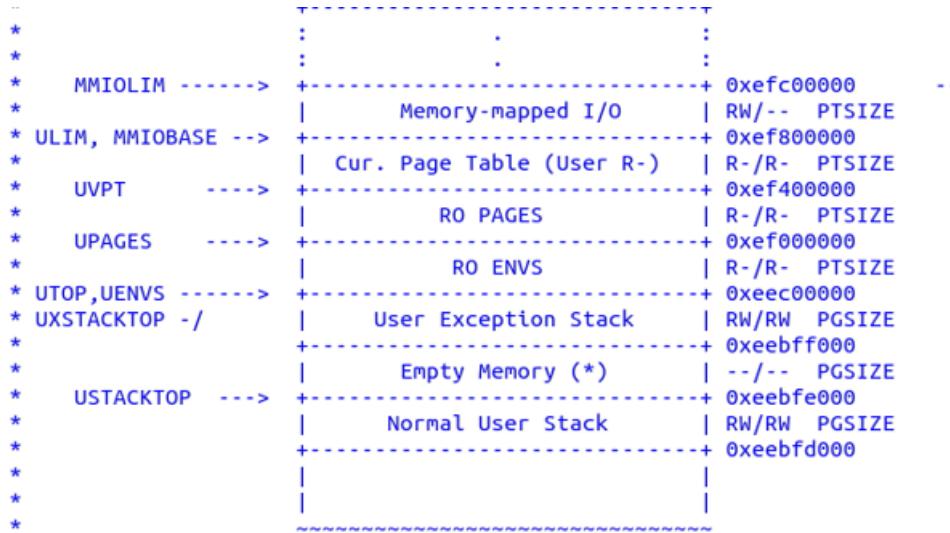


Figure 49: 用户正常栈和异常栈 (inc/memlayout.h)

从本质上说，我们需要让 JOS 内核实现自动的用户进程上的栈切换，就好比 x86 处理器为 JOS 从用户模式转向内核模式时所做的一样。在缺页错误处理之后，用户级的处理程序会返回到在原先堆栈上出错代码的位置。每一个想要支持用户级缺页错误处理的进程都必须为自己的异常堆栈分配内存空间，否则，在发生缺页错误时 JOS 内核将会像前面一样销毁用户进程。综合上述知识，我们可以总结出：一个支持用户级缺页错误处理的进程发生页错误处理的步骤是：进程 A 的正常栈 → 内核 → 进程 A 的异常栈。在此过程中，内核的工作就是按照 inc/trap.h 中 struct UTrapframe 的格式设置异常堆栈，确保进程能顺利切换到异常栈运行。UTrapframe 的格式如下：

```

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));

```

Figure 50: inc/trap.h 的 UTrapframe 结构体定义

此外，作业要求中还提到，如果用户程序在发生缺页错误时已经运行在异常堆栈上，那么可以知道缺页处理函数发生了缺页错误。在这样的情况下，就需要在当前的 **tf->tf_esp** 之下而不是 **UXSTACKTOP** 设置新的堆栈。因此，在处理这种情况时，需要首先压入一个 32 位空字 (word)，然后是一个 UTrapframe。为了识别这种情况，我们需要检验 **tf->tf_sep** 是否已经在用户的异常堆栈上，即检查它是否是在 [**UXSTACKTOP-PGSIZE** , **UXSTACKTOP-1**] 的区域。

代码补充如下：

```

1 void
2 page_fault_handler(struct Trapframe *tf)
3 {
4     .....
5     // LAB 4: Your code here.
6     struct UTrapframe *utf;
7
8     if (curenv->env_pgfault_upcall) {
9         //判断当前是否运行在用户异常栈
10        if (UXSTACKTOP - PGSIZE <= tf->tf_esp && tf->tf_esp <= UXSTACKTOP -
11            1)
12            utf = (struct UTrapframe *) (tf->tf_esp - sizeof(struct
13            UTrapframe) - 4);
14        else
15            utf = (struct UTrapframe *) (UXSTACKTOP - sizeof(struct
16            UTrapframe));
17        user_mem_assert(curenv, (void *)utf, sizeof(struct UTrapframe),
18            PTE_U | PTE_W);
19
20        utf->utf_fault_va = fault_va;
21        utf->utf_err = tf->tf_trapno;
22        utf->utf_eip = tf->tf_eip;
23        utf->utf_eflags = tf->tf_eflags;
24        utf->utf_esp = tf->tf_esp;
25        utf->utf_regs = tf->tf_regs;
26        //开始切换栈帧
27        tf->tf_eip = (uint32_t)curenv->env_pgfault_upcall;
28        tf->tf_esp = (uint32_t)utf;
29        env_run(curenv);
30    }
31 }

```

3.4 作业 5B

3.4.1 题目原文

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.
```

3.4.2 题目大意

实现在 `lib/pfentry.S` 中的 `_pgfault_upcall` 例程。返回到一开始运行造成缺页的用户代码这一部分很有趣。你在这里将会直接返回，而不是通过内核。最难的部分是同时调整堆栈并重新装载 EIP。

3.4.3 回答

根据题目要求，该部分代码需要实现在缺页处理函数执行完毕后，直接返回此前用户程序发生缺页错误的地方的功能，且此过程不需要经过内核。

```
// 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'.
```

Figure 51: lib/pfentry.S 的注释

```

// 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.
//

```

Figure 52: lib/pfentry.S 的注释

根据注释所指出的内容，在程序运行在嵌套的用户异常栈时，其结构为：

<code>Utf_esp</code>	
	Reserved 32bit
<code>48(%esp)</code>	<code>trap-time esp</code>
<code>44(%esp)</code>	<code>trap-time eflags</code>
<code>40(%esp)</code>	<code>trap-time eip</code>
<code>36(%esp)</code>	<code>utf_regs.reg_eax</code>
...	...
<code>12(%esp)</code>	<code>utf_regs.reg_esi</code>
<code>8(%esp)</code>	<code>utf_regs.reg_edi</code>
<code>4(%esp)</code>	<code>utf_err (error code)</code>
<code>%esp</code>	<code>utf_fault_va</code>

Figure 53: 目前的栈结构

首先第一段：

```

// LAB 4: Your code here.
movl 48(%esp), %ebp//将ebp指向utf_esp
subl $4, %ebp//utf_esp-4

```

```

movl %ebp, 48(%esp)// utf_esp变为utf_esp-4
movl 40(%esp), %eax//将eax指向utf_eip
movl %eax, (%ebp)//将eip存入地址utf_esp-4

```

此时，栈结构变为：

Utf_esp	
	trap-time eip
48(%esp)	trap-time esp-4
44(%esp)	trap-time eflags
40(%esp)	trap-time eip
36(%esp)	utf_regs.reg_eax
...	...
12(%esp)	utf_regs.reg_esi
8(%esp)	utf_regs.reg_edi
4(%esp)	utf_err (error code)
%esp	utf_fault_va

Figure 54: 目前的栈结构

其他代码更改如下：

```

1 .text
2 .globl _pgfault_upcall
3 _pgfault_upcall:
4     // Call the C page fault handler.
5     pushl %esp      // function argument: pointer to UTF
6     movl _pgfault_handler, %eax
7     call *%eax
8     addl $4, %esp    // pop function argument
9
10 .....
11 // LAB 4: Your code here.
12     movl 48(%esp), %ebp//将ebp指向utf_esp
13     subl $4, %ebp//utf_esp-4
14     movl %ebp, 48(%esp)// utf_esp变为utf_esp-4
15     movl 40(%esp), %eax//将eax指向utf_eip
16     movl %eax, (%ebp)//将eip存入地址utf_esp-4

```

```
17 // Restore the trap-time registers. After you do this, you
18 // can no longer modify any general-purpose registers.
19 // LAB 4: Your code here.
20     addl $8, %esp//跳过utf_fault_va和utf_err
21     popal//重新装载各寄存器
22 // Restore eflags from the stack. After you do this, you can
23 // no longer use arithmetic operations or anything else that
24 // modifies eflags.
25 // LAB 4: Your code here.
26     addl $4, %esp//跳过eip
27     popfl//pop eflags
28 // Switch back to the adjusted trap-time stack.
29 // LAB 4: Your code here.
30     popl %esp
31 // Return to re-execute the instruction that faulted.
32 // LAB 4: Your code here.
33     ret
```

3.5 作业 9

3.5.1 题目原文

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

3.5.2 题目大意

完成在 lib/pgfault.c 中的 set_pgfault_handler()。

3.5.3 回答

在发生缺页时,_pgfault_handler会被赋值为handler,且此函数在_pgfault_upcall中被调用。根据注释(如下图),可以看到当进程是第一次调用缺页处理函数时,_pgfault_handler会为0,那么我们需要为该进程分配一个页面作为异常栈,并且为该进程设置_pgfault_upcall(通过sys_env_set_pgfault_upcall()实现)。

其具体代码是:

```

// 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");
        envid_t e_id = sys_getenvid();
        r = sys_page_alloc(e_id, (void *) (UXSTACKTOP - PGSIZE), PTE_U | PTE_W | PTE_P);
        if (r < 0) {
            panic("pgfault_handler: %e", r);
        }
        // r = sys_env_set_pgfault_upcall(e_id, handler);
        r = sys_env_set_pgfault_upcall(e_id, _pgfault_upcall);
        if (r < 0) {
            panic("pgfault_handler: %e", r);
        }
    }

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

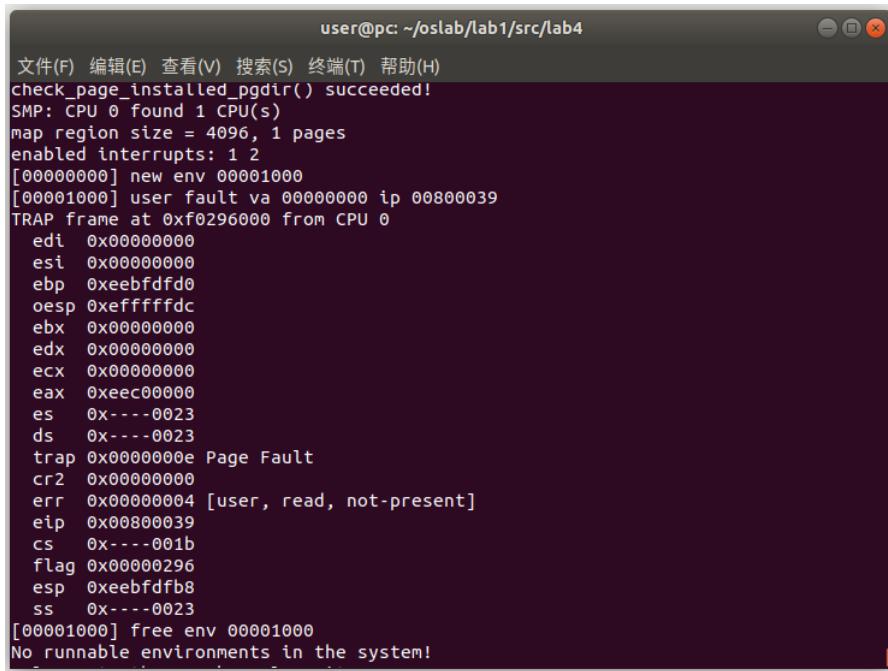
```

Figure 55: lib/pgfault.c 的 set_pgfault_handler()

3.6 Testing

3.6.1 运行 user/faultread

输入命令 `make run-faultread` 的运行结果如下：



```

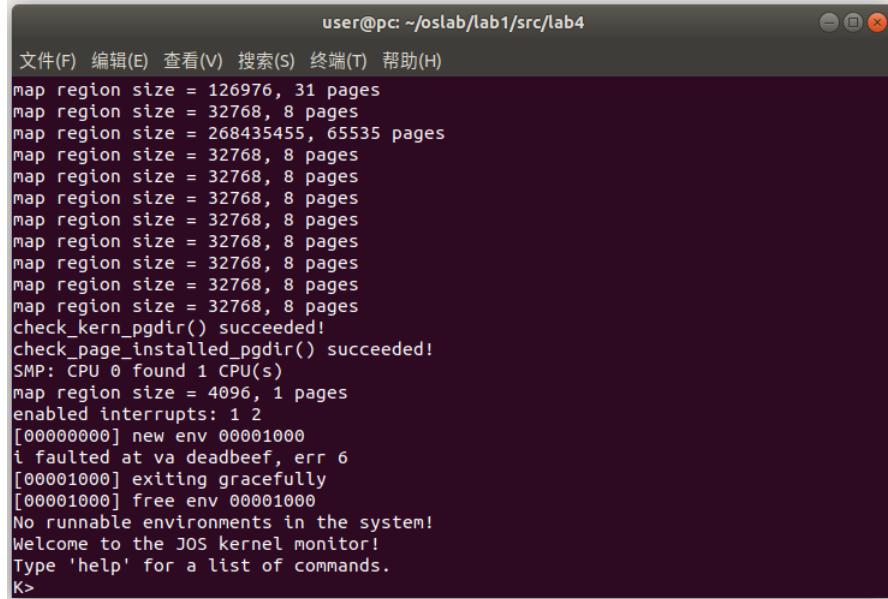
user@pc: ~/oslab/lab1/src/lab4
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 1 CPU(s)
map region size = 4096, 1 pages
enabled interrupts: 1 2
[00000000] new env 00001000
[00001000] user fault va 00000000 ip 00800039
TRAP frame at 0xf0296000 from CPU 0
    edi 0x00000000
    esi 0x00000000
    ebp 0xeebfdf0
    oesp 0xfffffd0
    ebx 0x00000000
    edx 0x00000000
    ecx 0x00000000
    eax 0xec00000
    es 0x----0023
    ds 0x----0023
    trap 0x0000000e Page Fault
    cr2 0x00000000
    err 0x00000004 [user, read, not-present]
    eip 0x00800039
    cs 0x----001b
    flag 0x00000296
    esp 0xeebfdfb8
    ss 0x----0023
[00001000] free env 00001000
No runnable environments in the system!

```

Figure 56: make run-faultread

3.6.2 运行 user/faultdie

输入命令`make run-faultdie`的运行结果如下：



```

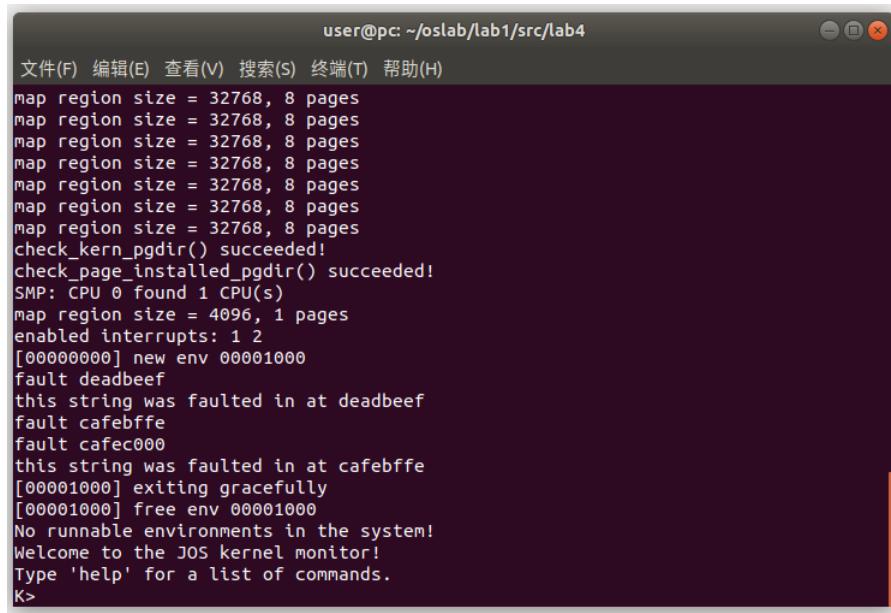
user@pc: ~/oslab/lab1/src/lab4
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
map region size = 126976, 31 pages
map region size = 32768, 8 pages
map region size = 268435455, 65535 pages
map region size = 32768, 8 pages
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 1 CPU(s)
map region size = 4096, 1 pages
enabled interrupts: 1 2
[00000000] new env 00001000
i faulted at va deadbeef, err 6
[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.
K>

```

Figure 57: make run-faultdie

3.6.3 运行 user/faultalloc

输入命令`make run-faultalloc`的运行结果如下:

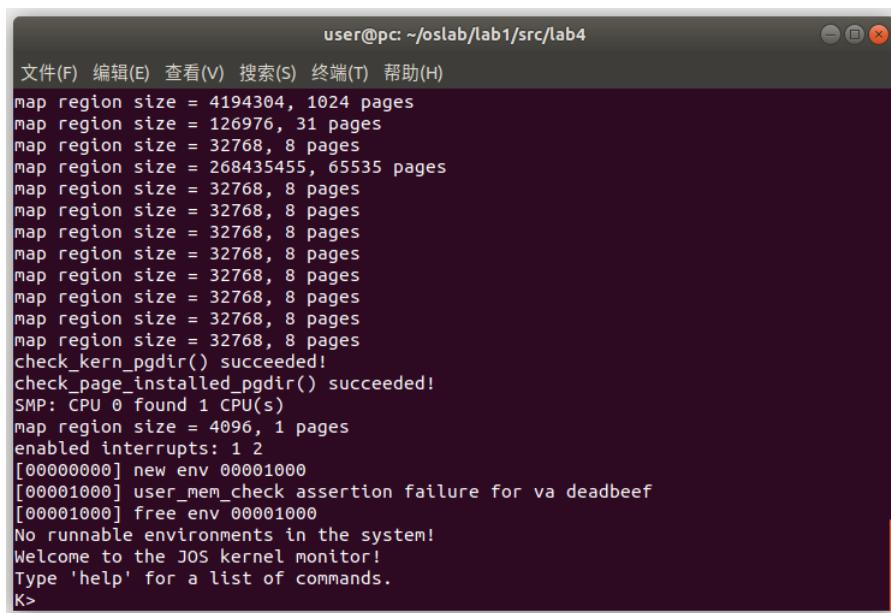


```
user@pc: ~/oslab/lab1/src/lab4
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
map region size = 32768, 8 pages
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 1 CPU(s)
map region size = 4096, 1 pages
enabled interrupts: 1 2
[00000000] new env 00001000
fault deadbeef
this string was faulted in at deadbeef
fault cafebff
fault cafec000
this string was faulted in at cafebff
[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.
K>
```

Figure 58: make run-faultalloc

3.6.4 运行 user/faultallocbad

输入命令`make run-faultallocbad`的运行结果如下:



```
user@pc: ~/oslab/lab1/src/lab4
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
map region size = 4194304, 1024 pages
map region size = 126976, 31 pages
map region size = 32768, 8 pages
map region size = 268435455, 65535 pages
map region size = 32768, 8 pages
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 1 CPU(s)
map region size = 4096, 1 pages
enabled interrupts: 1 2
[00000000] new env 00001000
[00001000] user_mem_check assertion failure for va deadbeef
[00001000] free env 00001000
No runnable environments in the system!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

Figure 59: make run-faultallocbad

`user/faultalloc` 和 `user/faultallocbad` 的运行结果不同的原因是：前者是直接通过用户态的 `cprintf()` 向控制台输出字符串；后者是通过系统调用 `sys_cputs()` 向控制台输出字符串，没有通过 `user_mem_assert()` 的断言（用户态的进程对地址 `0xdeadbeef` 没有任何权限），导致系统调用不成功。

```
void
umain(int argc, char **argv)
{
    set_pgfault_handler(handler);
    cprintf("%s\n", (char*)0xDeadBeef);
    cprintf("%s\n", (char*)0xCafeBffe);
}

void
umain(int argc, char **argv)
{
    set_pgfault_handler(handler);
    sys_cputs((char*)0xDEADBEEF, 4);
}
```

(a) user/faultalloc.c
(b) user/faultallocbad.c

Figure 60: 程序对比

```
// Print a string to the system console.
// The string is exactly 'len' characters long.
// Destroys the environment on memory errors.
static void
sys_cputs(const char *s, size_t len)
{
    // Check that the user has permission to read memory [s, s+len].
    // Destroy the environment if not.

    // LAB 3: Your code here.
    user_mem_assert(curenv, s, len, 0);
    // Print the string supplied by the user.
    cprintf("%.*s", len, s);
}
```

Figure 61: kern/syscall.c 的 sys_puts() 函数

3.7 作业 10

3.7.1 题目原文

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'
```

3.7.2 题目大意

实现在 `lib/fork.c` 中的 `fork`, `duppage` 和 `pgfault`。用 `forktree` 程序来测试你的代码 (`make run-forktree`)。它应当产生下面的输出, 其中夹杂着 **new env**, **free env** 和 **exiting gracefully** 这样的消息。下面的这些输出可能不是按照顺序的, 进程 ID 也可能有所不同:

```
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'
```

3.7.3 fork 与 dumbfork

在作业 6 中提到的 `dumbfork` 是对 `fork()` 的原始实现 (非写时复制), 而现在是真正 在用户空间中实现具有写时复制机制的 `fork`。两者最关键的区别在于: `dumbfork()` 会

复制页；而**fork()**则只是复制页的映射关系，它只有当一个进程需要对某个页执行写操作时才会对这个页进行复制，这就是所谓的写时复制。

3.7.4 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.
//
...

```

Figure 62: lib/fork.c 的 fork() 函数的注释

由实验文档以及**fork**的注释可知其执行流程：

1. 父进程使用 **set_pgfault_handler** 将 **pgfault()** 设为缺页处理函数。
2. 父进程调用 **sys_exofork()** 创建一个子进程。
3. 父进程为子进程设置与其相同的用户缺页处理函数入口。
4. 父进程将其子进程的状态设置为就绪态（可运行）。

对于父进程中每一个标记为可写的或者写时复制的虚拟地址位于 **UTOP** 以下的页，父进程将子进程地址空间中的对应页映射为写时复制属性，并且重新将自身地址空间中的该页映射为写时复制属性。父进程将两个进程的 PTE 都设置为只读，并且在 **avail** 区域中设置了 **PET_COW**（写时复制）属性，用以区别写时复制页和只读页。

对于异常栈，应该要在子进程为其重新分配一个新页。由于缺页处理函数将运行在异常栈上去复制实际的缺页内容，因此异常栈不能设置为写时复制的：如果将其设置为写时复制的，那么将没有复制异常栈页的载体。

```

1 envid_t
2 fork(void)
3 {
4     // LAB 4: Your code here.
5     // panic("fork not implemented");
6
7     set_pgfault_handler(pgfault); // 设置缺页处理函数为 pgfault

```

```

8  envid_t e_id = sys_exofork(); //产生新的子进程
9  if (e_id < 0)
10     panic("fork: %e", e_id);
11  if (e_id == 0) {
12      // child
13      thisenv = &envs[ENVX(sys_getenvid())]; //子进程作为当前进程（获取子
14      //进程的envid所对应的Env结构体的地址）
15      return 0;//返回0
16  }
17
18  uintptr_t addr;
19  //设置子进程地址空间中页映射，设置页映射的属性为写时复制属性
20  for (addr = UTEXT; addr < USTACKTOP; addr += PGSIZE) {
21      if ( (uvpd[PDX(addr)] & PTE_P) && (uvpt[PGNUM(addr)] & PTE_P) ) {
22          // dup page to child
23          duppage(e_id, PGNUM(addr));
24      }
25  }
26  //为异常栈单独分配新页
27  int r = sys_page_alloc(e_id, (void *) (UXSTACKTOP-PGSIZE), PTE_U | PTE_W
28  | PTE_P);
29  if (r < 0) panic("fork: %e", r);
30
31  //由于sys_exofork()并不会复制父进程的 e->env_pgfault_upcall 给子进程,
32  //故要给子进程单独设置
33  extern void _pgfault_upcall();
34  r = sys_env_set_pgfault_upcall(e_id, _pgfault_upcall);
35  if (r < 0) panic("fork: set upcall for child fail, %e", r);
36
37
38  return e_id;
39 }

```

其中uvpd和uvpt的声明和定义如下，是当前用户进程的页目录项和页表项：

```

extern volatile pte_t uvpt[];           // VA of "virtual page table"
extern volatile pde_t uvpd[];           // VA of current page directory

```

Figure 63: inc/memlayout.h

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

Figure 64: lib/entry.S

3.7.5 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.
//
```

Figure 65: lib/fork.c 的 duppage() 函数的注释

此函数在 `fork()` 中被调用，由父进程派生出子进程。因此在这里是将父进程的内存映射拷贝给子进程，实现父子进程内存空间的共享。

```
1 static int
2 duppage(envid_t envid, unsigned pn)
3 {
4     int r;
5
6     // LAB 4: Your code here.
7     // panic("duppage not implemented");
8
9     envid_t this_env_id = sys_getenvid(); // 获取当前进程（父进程）的envid
10    void * va = (void * )(pn * PGSIZE); // 页首地址
11
12    int perm = uvpt[pn] & 0FFF;
13    if ( (perm & PTE_W) || (perm & PTE_COW) ) {
14        // 将权限位设为只读和写时复制
15        perm |= PTE_COW;
16        perm &= ~PTE_W;
17    }
18    // 需要设置权限位以适应函数sys_page_map()调用的要求
19    perm &= PTE_SYSCALL;
20
21    // 将父进程的内存映射拷贝会给予进程，实现父子进程内存的共享
22    if((r = sys_page_map(this_env_id, va, envid, va, perm)) < 0)
23        panic("duppage: %e", r);
```

```

24 // 父进程的映射需要重新建立
25 if((r = sys_page_map(this_env_id, va, this_env_id, va, perm)) < 0)
26     panic("duppage: %e", r);
27 return 0;
28 }

```

3.7.6 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)

```

Figure 66: lib/fork.c 的 pgfault() 函数的注释

```

// 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>).

```

Figure 67: lib/fork.c 的 pgfault() 函数的注释

```

// 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.

```

Figure 68: lib/fork.c 的 pgfault() 函数的注释

每当一个进程要修改它之前没有修改过的写时复制页时，就会出现缺页错误。由实验文档以及 pgfault 的注释可知其执行流程：

1. 内核将缺页错误传递给 `_pgfault_upcall`，后者调用 `fork()` 的 `pgfault()` 函数。
2. `pgfault()` 首先检查这个错误是写操作 (`FEC_WR`) 并且缺页的 PTE 是含有写时复制 (`PTE_COW`) 属性的。如果不是则 `panic`。
3. 然后 `pgfault()` 会分配一个新物理页并将缺页暂存区 `PFTEMP` 和这个物理页映射。
4. 然后将出错地址对应物理页的数据拷贝到 `PFTEMP` 对应的物理页。

5. 然后解除出错地址与原先物理页的映射，并将出错地址与PFTEMP的物理页进行映射。
6. 最后解除PFTEMP与其原先物理页的映射，这样就只有出错地址映射到这个物理页上了。

注释还说在此期间需要进行三次系统调用。

PTE_COW的宏定义也在lib/fork.c中，如下图所示：

```
// PTE_COW marks copy-on-write page table entries.
// It is one of the bits explicitly allocated to user processes (PTE_AVAIL).
#define PTE_COW      0x800
```

Figure 69: PTE_COW

因此函数pgfault()的实现如下：

```
1 static void
2 pgfault(struct UTrapframe *utf)
3 {
4     void *addr = (void *) utf->utf_fault_va;
5     uint32_t err = utf->utf_err;
6     int r;
7
8     // Check that the faulting access was (1) a write, and (2) to a
9     // copy-on-write page. If not, panic.
10    // Hint:
11    // Use the read-only page table mappings at uvpt
12    // (see <inc/memlayout.h>).
13
14    // LAB 4: Your code here.
15    // 检查引发缺页错误的操作是不是写操作，且pte是否带有写时复制属性，否则
16    // panic
17    if ((err & FEC_WR)==0 || (uvpt[PGNUM(addr)] & PTE_COW)==0) {
18        panic("pgfault: it's not writable or attempt to access a non-cow
page!");
19    }
20    // Allocate a new page, map it at a temporary location (PFTEMP),
21    // copy the data from the old page to the new page, then move the new
22    // page to the old page's address.
23    // Hint:
24    // You should make three system calls.
25    // LAB 4: Your code here.
```

```
26 // panic("pgfault not implemented");
27 envid_t envid = sys_getenvid(); // 获取当前进程的envid
28 // 在暂存区PFTEMP (见inc/memlayout.h) 中分配一个物理页
29 if ((r = sys_page_alloc(envid, (void *)PFTEMP, PTE_P | PTE_W | PTE_U)) < 0)
30     panic("pgfault: page allocation failed %e", r);
31
32 // 将数据从addr的物理页拷贝到新分配的页上
33 addr = ROUNDDOWN(addr, PGSIZE);
34 memmove(PFTEMP, addr, PGSIZE);
35 // 解除出错地址addr与其原来物理页的映射
36 if ((r = sys_page_unmap(envid, addr)) < 0)
37     panic("pgfault: page unmap failed (%e)", r);
38 // 将addr与PFTEMP对应的物理页映射
39 if ((r = sys_page_map(envid, PFTEMP, envid, addr, PTE_P | PTE_W | PTE_U)) < 0)
40     panic("pgfault: page map failed (%e)", r);
41 // 解除PFTEMP与其原来物理页的映射
42 if ((r = sys_page_unmap(envid, PFTEMP)) < 0)
43     panic("pgfault: page unmap failed (%e)", r);
44 }
```

在终端输入命令make run-forktree得到的输出如下：

```
user@pc: ~/oslab/lab1/src/lab4
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
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
[00001003] new env 00001006
1005: I am '000'
[00001005] exiting gracefully
[00001005] free env 00001005
[00001000] exiting gracefully
[00001000] free env 00001000
[00001001] exiting gracefully
[00001001] free env 00001001
1002: I am '1'
[00001002] new env 00002001
[00001002] new env 00002000
[00001002] exiting gracefully
[00001002] free env 00001002
2000: I am '11'
[00002000] new env 00002002
2001: I am '10'
[00002001] new env 00002005
[00001003] exiting gracefully
[00001003] free env 00001003
[00002000] new env 00002003
[00002001] new env 00001007
[00002001] exiting gracefully
```

Figure 70: make run-forktree

至此，Lab 4 的 Part B 已完成，以下是在终端输入命令 `make grade` 得到的打分情况：

```
faultread: OK (1.2s)
faultwrite: OK (1.7s)
faultdie: OK (1.8s)
faultregs: OK (1.7s)
faultalloc: OK (0.7s)
faultallocbad: OK (2.1s)
faultnystack: OK (2.9s)
faultbadhandler: OK (2.1s)
faulterrors: OK (2.5s)
forktree: OK (1.5s)
Part B score: 50/50
```

Figure 71: Part B 打分情况

4 Part C: 抢占式多任务和进程间通信

4.1 作业 11

4.1.1 题目原文

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.

The processor never pushes an error code or checks the Descriptor Privilege Level (DPL) of the IDT entry when invoking a hardware interrupt handler. You might want to re-read section 9.2 of the 80386 Reference Manual (<http://pdos.csail.mit.edu/6.828/2011/readings/i386/toc.htm>), or section 5.8 of the IA-32 Intel Architecture Software Developer's Manual, Volume 3 (<http://pdos.csail.mit.edu/6.828/2011/readings/ia32/IA32-3A.pdf>), 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.

4.1.2 题目大意

修改 `kern/trapentry.S` 和 `kern/trap.c` 来初始化一个合适的 IDT 入口，并为 IRQ 0-15 提供处理函数。接着，修改 `kern/env.c` 中的 `env_alloc()` 以确保用户进程总是在中断被打开的情况下运行。

当调用用户中断处理函数时，处理器从来不会将 error code 压栈，也不会检查 IDT 入口的描述符特权等级 (Descriptor Privilege Level, DPL)。此时你可能需要重新阅读一下 80386 手册中 9.2 这一部分，或者 IA-32 Intel Architecture Software Developer's Manual, Volume 3 的 5.8 章节。

完成这个练习后，当你运行任何一个运行时间较长的测试程序时（比如 `make run-spin`），你应当看见内核打印硬件中断的 trap frame。因为到目前为止，虽然处理器的硬件中断已经被打开了，但 JOS 还没有处理它，所以你应该注意到，它以为这个中断发生在正在运行的用户进程，并将其销毁。最终当没有进程可以销毁的时候，JOS 会陷入内核监视器。

4.1.3 回答

IRQ 即外部中断，IRQ 到中断描述符表的入口不是固定的。但在 `pic_init` 中我们已经将 IRQ 的 0-15 映射到了 IDT 的 `[IRQ_OFFSET, IRQ_OFFSET+15]`。其中 `IRQ_OFFSET` 为

32，所以 IRQ 在 IDT 中的范围是 [32, 47]，一共 16 个。JOS 对中断做了简化处理，在内核态时外部中断时禁止的，在用户态时外部中断是允许的。外部中断的开启和禁止通过`eflags`寄存器的`FL_IF`位控制，1 表示开中断，0 表示关中断。

接下来需要像 Lab 3 一样，设置中断号和中断处理程序。在实验 3 中将`istrap`设置为 1 不影响实验结果，在实验 4 中`istrap`需要全部设置为 0。如果`istrap`为 1 的话，JOS 会在开始处理中断时将标志位寄存器`%eflag`的`FL_IF`位置为 1，而`istrap`为 0 时则保持`FL_IF`不变。因此只有将`istrap`置 0 才能通过`trap()`中对`FL_IF`的检查。另外，从题干可以知道，这些中断都是不产生错误码的，因此要使用宏定义`TRAPHANDLER_NOEC`。

4.1.4 kern/trapentry.S

```
1 .text
2
3 /*
4  * Lab 3: Your code here for generating entry points for the different
5  * traps.
6 */
7 TRAPHANDLER_NOEC(divide_handler, T_DIVIDE);
8 TRAPHANDLER_NOEC(debug_handler, T_DEBUG);
9 TRAPHANDLER_NOEC(nmi_handler, T_NMI);
10 TRAPHANDLER_NOEC(brkpt_handler, T_BRKPT);
11 TRAPHANDLER_NOEC(oflow_handler, T_OFLOW);
12 TRAPHANDLER_NOEC(bound_handler, T_BOUND);
13 TRAPHANDLER_NOEC(illop_handler, T_ILLOP);
14 TRAPHANDLER_NOEC(device_handler, T_DEVICE);
15 TRAPHANDLER(dblflt_handler, T_DBLFLT);
16 // 9 deprecated since 386
17 TRAPHANDLER(tss_handler, T_TSS);
18 TRAPHANDLER(segnp_handler, T_SEGNP);
19 TRAPHANDLER(stack_handler, T_STACK);
20 TRAPHANDLER(gpflt_handler, T_GPFLT);
21 TRAPHANDLER(pgflt_handler, T_PGFLT);
22 // 15 reserved by intel
23 TRAPHANDLER_NOEC(fperr_handler, T_FPERR);
24 TRAPHANDLER(align_handler, T_ALIGN);
25 TRAPHANDLER_NOEC(mchk_handler, T_MCHK);
26 TRAPHANDLER_NOEC(simderr_handler, T_SIMDERR);
27 // system call (interrupt)
28 TRAPHANDLER_NOEC(syscall_handler, T_SYSCALL);
```

```

28 // Lab 4
29 TRAPHANDLER_NOEC(irq_timer, IRQ_OFFSET + IRQ_TIMER);
30 TRAPHANDLER_NOEC(irq_kbd, IRQ_OFFSET + IRQ_KBD);
31 TRAPHANDLER_NOEC(irq_serial, IRQ_OFFSET + IRQ_SERIAL);
32 TRAPHANDLER_NOEC(irq_spurious, IRQ_OFFSET + IRQ_SPURIOUS);
33 TRAPHANDLER_NOEC(irq_ide, IRQ_OFFSET + IRQ_IDE);
34 TRAPHANDLER_NOEC(irq_error, IRQ_OFFSET + IRQ_ERROR);
35
36 /*
37 * Lab 3: Your code here for _alltraps
38 */
39 _alltraps:
40     //第一步: 将被中断的程序的%ds, %es和通用寄存器内容压栈保存
41     pushl %ds
42     pushl %es
43     pushal //压入8个通用寄存器
44
45     //第二步: 将GD_KD的值存入%ds和%es
46     //注意段寄存器不能直接传入立即数, 因此需要%eax中介
47     movl $GD_KD, %eax
48     movl %eax, %ds
49     movl %eax, %es
50
51     //第三步: %esp入栈
52     pushl %esp
53     //第四步: 调用trap()函数
54     call trap

```

IRQ 的相关参数可在inc/trap.h中找到。

```

#define IRQ_OFFSET      32      // IRQ 0 corresponds to int IRQ_OFFSET

// Hardware IRQ numbers. We receive these as (IRQ_OFFSET+IRQ_WHATEVER)
#define IRQ_TIMER      0
#define IRQ_KBD        1
#define IRQ_SERIAL     4
#define IRQ_SPURIOUS   7
#define IRQ_IDE        14
#define IRQ_ERROR      19

```

Figure 72: inc/trap.h 的 IRQ 参数

FL_IF在inc/mmu.h可以找到。

```
#define FL_IF          0x000000200    // Interrupt Flag
```

Figure 73: FL_IF

4.1.5 kern/trap.c

```
1 void
2 trap_init(void)
3 {
4     extern struct Segdesc gdt[];
5
6     // LAB 3: Your code here.
7     // 声明后才能使用
8     void divide_handler();
9     void debug_handler();
10    void nmi_handler();
11    void brkpt_handler();
12    void oflow_handler();
13    void bound_handler();
14    void illop_handler();
15    void device_handler();
16    void dblflt_handler();
17    void tss_handler();
18    void segnp_handler();
19    void stack_handler();
20    void gpflt_handler();
21    void pgflt_handler();
22    void fperr_handler();
23    void align_handler();
24    void mchk_handler();
25    void simderr_handler();
26    void syscall_handler();
27    //Lab 4
28    void irq_timer();
29    void irq_kbd();
30    void irq_serial();
31    void irq_spurious();
32    void irq_ide();
33    void irq_error();
34
35    //Exception
36    SETGATE(idt[T_DIVIDE], 1, GD_KT, divide_handler, 0);
37    SETGATE(idt[T_DEBUG], 1, GD_KT, debug_handler, 0);
38    SETGATE(idt[T_NMI], 0, GD_KT, nmi_handler, 0); //non-maskable interrupt
39    SETGATE(idt[T_BRKPT], 1, GD_KT, brkpt_handler, 3); //需要调用monitor, 所以
```

```

权限为3
40 SETGATE(idt[T_OFLOW], 1, GD_KT, oflow_handler, 0);
41 SETGATE(idt[T_BOUND], 1, GD_KT, bound_handler, 0);
42 SETGATE(idt[T_ILOP], 1, GD_KT, illop_handler, 0);
43 SETGATE(idt[T_DEVICE], 1, GD_KT, device_handler, 0);
44 SETGATE(idt[T_DBLFLT], 1, GD_KT, dblflt_handler, 0);
45 SETGATE(idt[T_TSS], 1, GD_KT, tss_handler, 0);
46 SETGATE(idt[T_SEGNP], 1, GD_KT, segnp_handler, 0);
47 SETGATE(idt[T_STACK], 1, GD_KT, stack_handler, 0);
48 SETGATE(idt[T_GPFILT], 1, GD_KT, gpflt_handler, 0);
49 SETGATE(idt[T_PGFILT], 0, GD_KT, pgflt_handler, 0); // 需要修改缺页错误的第
    二个参数为0
50 SETGATE(idt[T_FPEERR], 1, GD_KT, fperr_handler, 0);
51 SETGATE(idt[T_ALIGN], 1, GD_KT, align_handler, 0);
52 SETGATE(idt[T_MCHK], 1, GD_KT, mchk_handler, 0);
53 SETGATE(idt[T_SIMDERR], 1, GD_KT, simderr_handler, 0);
54
//Interrupt
55 SETGATE(idt[T_SYSCALL], 0, GD_KT, syscall_handler, 3); // 由于是用户程序请
    求系统调用，因此权限也为3
56
57 //Lab 4
58 SETGATE(idt[IRQ_OFFSET + IRQ_TIMER], 0, GD_KT, irq_timer, 0);
59 SETGATE(idt[IRQ_OFFSET + IRQ_KBD], 0, GD_KT, irq_kbd, 0);
60 SETGATE(idt[IRQ_OFFSET + IRQ_SERIAL], 0, GD_KT, irq_serial, 0);
61 SETGATE(idt[IRQ_OFFSET + IRQ_SPURIOUS], 0, GD_KT, irq_spurious, 0);
62 SETGATE(idt[IRQ_OFFSET + IRQ_IDE], 0, GD_KT, irq_ide, 0);
63 SETGATE(idt[IRQ_OFFSET + IRQ_ERROR], 0, GD_KT, irq_error, 0);
64
65
66
67 // Per-CPU setup
68 trap_init_percpu();
69 }

```

注意在SETGATE调用缺页错误的第二个参数必须为0，修正了Lab 3的错误。否则会不能通过kern/trap.c的trap()的assert(!(read_eflags() & FL_IF));断言。

4.1.6 kern/env.c

这一部分要实现进程的抢占式调度。前面的调度是进程资源放弃CPU，但是实际中没有进程会这样做。为了将CPU资源公平地分配给各个进程，保证CPU对用户请求及时响应，需要抢占式调度。

JOS 采取的策略是，在内核态中，外部中断是始终关闭的，而在用户态时，需要开中断。为保证进入用户态时外部中断是使能的，可以在每个进程初始化的时候就将外部中断使能位置位。为此，要修改kern/env.c的env_alloc()。

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

Figure 74: env_alloc()

输入命令make run-spin的运行结果如下：

```
user@pc: ~/oslab/lab1/src/lab4
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 1 CPU(s)
map region size = 4096, 1 pages
enabled interrupts: 1 2
[00000000] new env 00001000
I am the parent. Forking the child...
[00001000] new env 00001001
TRAP frame at 0xf0296000 from CPU 0
  edi 0x2cf79000
  esi 0x00802000
  ebp 0xeebfdfb0
  oesp 0xfffffdcc
  ebx 0x00000bff
  edx 0x00001000
  ecx 0x00802000
  eax 0x00000000
  es 0x----0023
  ds 0x----0023
  trap 0x00000020 Hardware Interrupt
  err 0x00000000
  eip 0x00801187
  cs 0x----001b
  flag 0x00000246
  esp 0xeebfdf68
  ss 0x----0023
[00001000] free env 00001000
No runnable environments in the system!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> |
```

Figure 75: make run-spin

中断号 32，对应irq_timer，说明还没有实现时钟中断，内核得不到控制权。

4.2 作业 12

4.2.1 题目原文

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.

4.2.2 题目大意

修改内核的 `trap_dispatch()` 函数，使得其每当收到时钟中断的时候，它会调用 `sched_yield()` 寻找另一个进程并运行。

你现在应当能让 `user/spin` 测试程序正常工作了：父进程会创建子进程，`sys_yield()` 会切换到子进程几次，但在时间片过后父进程会重新占据 CPU，并最终杀死子进程并正常退出。

4.2.3 回答

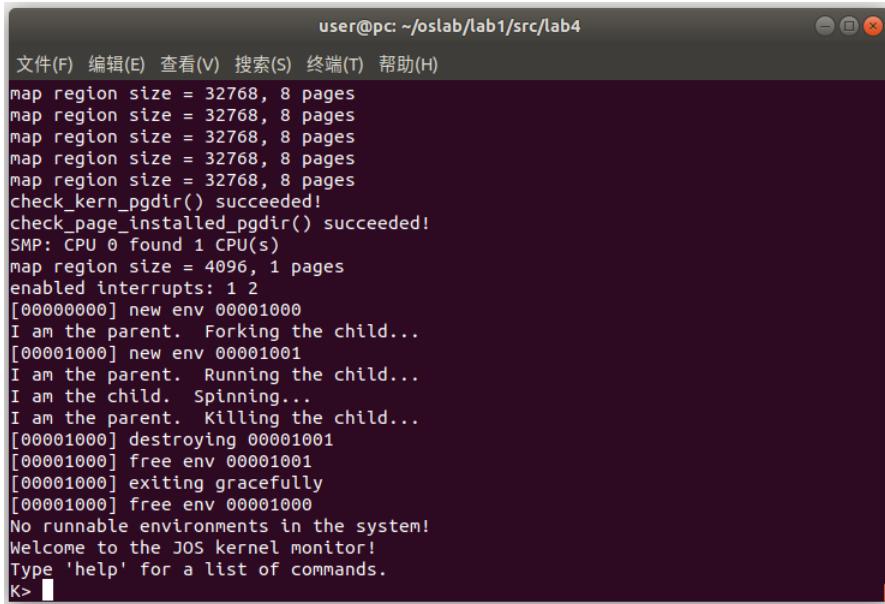
现在虽然中断使能已经打开，在用户态进程运行的时候，外部中断会产生并进入内核，但是现在还没有能处理这类中断。所以需要修改 `kern/trap.c` 的 `trap_dispatch()`，在发生外部定时中断的时候，调用调度器，调度另外一个可运行的进程。根据注释中的提示增加的代码如下：

```
// Handle clock interrupts. Don't forget to acknowledge the
// interrupt using lpic_eoi() before calling the scheduler!
// LAB 4: Your code here.
if (tf->tf_trapno == IRQ_OFFSET + IRQ_TIMER) {
    lpic_eoi();
    sched_yield();
    return;
}
```

Figure 76: `trap_dispatch()`

其中，`lpic_eoi()` 是 lpic 设备接收并处理时钟中断，定义在 `kern/lpic.c`。

输入命令 `make run-spin` 的运行结果如下，可正常运行：



```

user@pc: ~/oslab/lab1/src/lab4
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
map region size = 32768, 8 pages
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 1 CPU(s)
map region size = 4096, 1 pages
enabled interrupts: 1 2
[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.
K> 

```

Figure 77: make run-spin

4.3 作业 13

4.3.1 题目原文

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. You might find it interesting to read `user/primes.c` to see all the forking and IPC going on behind the scenes.

4.3.2 题目大意

实现 `kern/syscall.c` 中的 `sys_ipc_recv` 和 `sys_ipc_try_send`。这两个函数需要协同工作。当调用 `envid2env` 时，应当将 `checkperm` 设置为 0，这意味着进程可以与任何其他进程通信，内核除了确保目标进程 ID 有效之外，不会做其他任何检查。

接下来在 `lib/ipc.c` 中实现 `ipc_recv` 和 `ipc_send`。

用 `user/pingpong` 和 `user/primes` 来测试 IPC 机制。`user/primes` 会为每一个素数生成一个新的进程，直到 JOS 已经没有新的进程页可以分配了。

`user/primes.c` 用来创建子进程和通信的代码读起来可能很有趣。

4.3.3 回答

在 `inc/env.h` 的结构体 `Env` 定义中，增加了 IPC 用于传递消息的成员变量，如下图所示：

```
// Lab 4 IPC
bool env_ipc_recving;           // Env is blocked receiving
void *env_ipc_dstva;            // VA at which to map received page
uint32_t env_ipc_value;          // Data value sent to us
envid_t env_ipc_from;           // envid of the sender
int env_ipc_perm;               // Perm of page mapping received
```

Figure 78: IPC 用于传递消息的成员变量

各成员变量的意义如下：

- `env_ipc_recving`: 当进程使用 `sys_ipc_recv()` 等待信息时，此成员变量的值被设为 1，然后被阻塞等待；当进程向其发送消息以解除阻塞时，发送进程将此成员变量的值修改为 0。
- `env_ipc_dstva`: 接收页的虚拟地址，不能超过 UTOP。
- `env_ipc_value`: 若等待消息的进程接收的消息的消息值。
- `env_ipc_from`: 发送方的 `envid`。
- `env_ipc_perm`: 接收页的权限。

4.3.4 `sys_ipc_recv()`

以下是该函数的注释：

```
// 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.
```

Figure 79: kern/syscall.c 的 `sys_ipc_recv()` 函数的注释

该函数会让进程进入等待接收的阻塞状态，同时放弃自己的CPU。这个进程要接收的页一定是和`dstva`建立好映射的。这里只需要检查的错误就是`dstva`有没有超过UTOP，是不是页对齐的（因为要传送的是页）。

因此`sys_ipc_recv()`函数实现如下：

```
1 static int
2 sys_ipc_recv(void *dstva)
3 {
4     // LAB 4: Your code here.
5     // panic("sys_ipc_recv not implemented");
6
7     // 检查接收地址的范围和是否页对齐
8     if ((uintptr_t) dstva < UTOP && PGOFF(dstva) != 0)
9         return -EINVAL;
10
11    envid_t envid = sys_getenvid();
12    struct Env *e;
13    if (envid2env(envid, &e, 0) < 0) { // 检查当前envid(源进程)的Env是否存在，不需要检查权限
14        return -E_BAD_ENV;
15    }
16
17    e->env_ipc_recving = 1; // 目的进程等待接收消息
18    e->env_ipc_dstva = dstva;
19    e->env_status = ENV_NOT_RUNNABLE; // 等待消息时，该进程会被阻塞
20    sys_yield(); // 进程被阻塞，就释放CPU给其他进程
21
22    return 0;
23 }
```

4.3.5 sys_ipc_try_send()

以下是该函数的注释：

```

// 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.

```

Figure 80: kern/syscall.c 的 sys_ipc_try_send() 函数的注释

由注释可知,该函数的作用是尝试发送消息给目标进程的envid。如果源虚拟地址小于UTOP,则需要发送源进程的页映射(用于共享内存)给目的进程。如果发送成功,则更新上述提到的IPC域,同时取消目的进程的阻塞状态(设为可执行,ENV_RUNNABLE),再从寄存器eax返回0(成功返回值)。sys_ipc_recv没有这个返回值的原因是进程会释放其CPU进入阻塞状态。如果目的进程并不处于等待消息的阻塞状态(env_ipc_recving为0),则发送失败,返回-E_IPC_NOT_RECV。函数还有其他执行失败的原因,大体上和作业6中提到的失败原因类似,就不再多解释了。只是需要注意的一点是,在使用函数envid2env()去获取该envid对应的结构体Env时不需要检查权限,即第三个参数置为0。

因此函数sys_ipc_try_send()的实现如下:

```

1 static int
2 sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned perm)
3 {
4     // LAB 4: Your code here.
5     //panic("sys_ipc_try_send not implemented");
6     struct Env *env=0;
7     int r =0;

```

```

8 pte_t * pte =0;
9 if((r = envid2env(envid, &env, 0)) < 0) //不需要检查权限
10   return -E_BAD_ENV;
11
12
13 if(env->env_ipc_recving == 0) //如果目的进程不处于等待接收状态，则发送失败
14   return -E_IPC_NOT_RECV;
15
16 if((int)srcva < UTOP) {
17   //其他错误检查，不多解释
18   if( (int)srcva < UTOP && ((int)srcva % PGSIZE != 0) )
19     return -E_INVAL;
20
21   if( (perm & (~PTE_SYSCALL)) !=0 )
22     return -E_INVAL;
23
24   if( (perm & PTE_P) ==0 )
25     return -E_INVAL;
26
27   struct PageInfo *page = page_lookup(curenv->env_pgdir, srcva, &pte);
28   if( (perm & PTE_W) && ( (*pte & PTE_W) == 0) )
29     return -E_INVAL;
30
31   if((int)env->env_ipc_dstva >= UTOP)
32     return 0;
33   r = page_insert(env->env_pgdir, page, env->env_ipc_dstva ,perm); //将当前进程的物理页和目的进程地址空间建立映射，以实现共享内存
34   if(r < 0)
35     return -E_NO_MEM;
36 }
37
38 env->env_ipc_value = value; //将env_ipc_value设置参数value指定的消息值
39 env->env_ipc_from = curenv->env_id; //env_ipc_from设为当前（发送）进程的envid
40 env->env_ipc_perm = perm;
41 env->env_ipc_recving = 0; //取消等待接收状态
42 env->env_status = ENV_RUNNABLE; //可执行
43 env->env_tf.tf_regs.reg_eax = 0; //执行成功则返回0
44
45 return 0;
46 }

```

4.3.6 增加系统调用分支

完成了函数`sys_ipc_recv()`和`sys_ipc_try_send()`需要在`kern/syscall.c`增加新的系统调用分支:

```
1 // Dispatches to the correct kernel function, passing the arguments.
2 int32_t
3 syscall(uint32_t syscallno, uint32_t a1, uint32_t a2, uint32_t a3, uint32_t
4         a4, uint32_t a5)
5 {
6     // Call the function corresponding to the 'syscallno' parameter.
7     // Return any appropriate return value.
8     // LAB 3: Your code here.
9
10    //panic("syscall not implemented");
11    switch (syscallno) {
12        case SYS_cputs:
13            sys_cputs((const char *)a1, a2);
14            return 0;
15        case SYS_cgetc:
16            return sys_cgetc();
17        case SYS_getenvid:
18            return sys_getenvid();
19        case SYS_env_destroy:
20            return sys_env_destroy(a1);
21        case SYS_yield:
22            sys_yield();
23            return 0;
24        case SYS_exofork:
25            return (int32_t)sys_exofork();
26        case SYS_env_set_status:
27            return sys_env_set_status(a1, a2);
28        case SYS_page_alloc:
29            return sys_page_alloc(a1, (void *)a2, (int)a3);
30        case SYS_page_map:
31            return sys_page_map(a1, (void *)a2, a3, (void*)a4, (int)a5);
32        case SYS_page_unmap:
33            return sys_page_unmap(a1, (void *)a2);
34        case SYS_env_set_pgfault_upcall:
35            return sys_env_set_pgfault_upcall(a1, (void*)a2);
36        case SYS_ipc_try_send:
37            return sys_ipc_try_send(a1, a2, (void *)a3, a4);
38        case SYS_ipc_recv:
```

```

38         return sys_ipc_recv((void *)a1);
39     default:
40         return -E_INVAL;
41     }
42 }
```

4.3.7 ipc_recv()

以下是该函数的注释:

```

// 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.)
```

Figure 81: lib/ipc.c 的 ipc_recv() 函数的注释

该函数的作用是通过 IPC 接收一个值并返回。如果参数`pg`不为空，则通过系统调用`sys_ipc_recv()`接收一个页映射。如果参数`from_env_store`不为空，则用该指针指向的变量保存发送者进程的`envid`。如果参数`perm_store`不为空，则用该指针指向的变量保存发送者进程的页权限（当且仅当`pg`被成功映射时该值不为 0）。`thisenv`可以代表这个发送者进程。如果系统调用失败，则往参数的两个指针（如果不为空）指向的两个变量存入 0 值并返回错误值。成功就返回这个从发送者收到的值。如果`pg`为空，则应该往`sys_ipc_recv()`传入一个非 0 的非法参数，可以传入 UTOP 值，这样`sys_ipc_recv()`会检测到超界然后执行失败。

因此函数`ipc_recv()`的实现如下:

```

1 int32_t
2 ipc_recv(envid_t *from_env_store, void *pg, int *perm_store)
3 {
4     // LAB 4: Your code here.
5     //panic("ipc_recv not implemented");
6     int r =0;
7     int a;
8     if(pg == 0)
```

```

9   r= sys_ipc_recv( (void *)UTOP); //传入非法虚拟地址值以表示没有物理页
10  else
11    r = sys_ipc_recv(pg); //使用pg来接收映射
12  if(r == 0){
13    if( from_env_store != 0 )
14      *from_env_store = thisenv->env_ipc_from; //存入发送者进程的envid
15
16    if(perm_store != 0 )
17      *perm_store = thisenv->env_ipc_perm; //存入发送者进程的页权限
18  }
19 else{ //系统调用失败
20  panic("The ipc_recv is not right, and the errno is %d\n",r);
21 //存入0值
22  if(from_env_store != 0 )
23    *from_env_store = 0;
24
25  if(perm_store != 0 )
26    *perm_store = 0;
27  return r; //返回错误值
28 }
29 return thisenv->env_ipc_value;//成功就返回从发送者收到的值
30 }
```

4.3.8 ipc_send()

以下是该函数的注释：

```

// 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.)
```

Figure 82: lib/ipc.c 的 ipc_send() 函数的注释

该函数的作用是进程不断尝试发送一个值，直到成功为止。发送值的操作通过系统调用 `sys_ipc_try_send()` 来完成。如果系统调用失败，除了 `-E_IPC_NOT_RECV` 之外都要 `panic`。特别地，如果遇到了错误 `-E_IPC_NOT_RECV` 暂时先放弃 CPU（等到被调度之后再执行）。

因此函数 `ipc_send()` 的实现如下：

```
1 void
```

```

2 ipc_send(envid_t to_env, uint32_t val, void *pg, int perm)
3 {
4     // LAB 4: Your code here.
5     //panic("ipc_send not implemented");
6     int r = 0;
7     while(1){ //直到成功才停止尝试发送（退出循环）
8         if(pg == 0)
9             r=sys_ipc_try_send(to_env, val, (void*) UTOP, perm);
10        else
11            r = sys_ipc_try_send(to_env, val, pg, perm);
12
13        if(r <0 && r != -E_IPC_NOT_RECV){ //遇到其他错误则panic
14            cprintf("the envid is %x\n", sys_getenvid());
15            panic("ipc_send is error, and the errno is %d\n", r);
16        }
17        else if(r == -E_IPC_NOT_RECV) //如果未接收，则需要放弃CPU给其他进程
18            sys_yield();
19        else break; //执行成功则退出循环
20    }
21 }

```

输入命令`make run-pingpong`的结果如下：

```

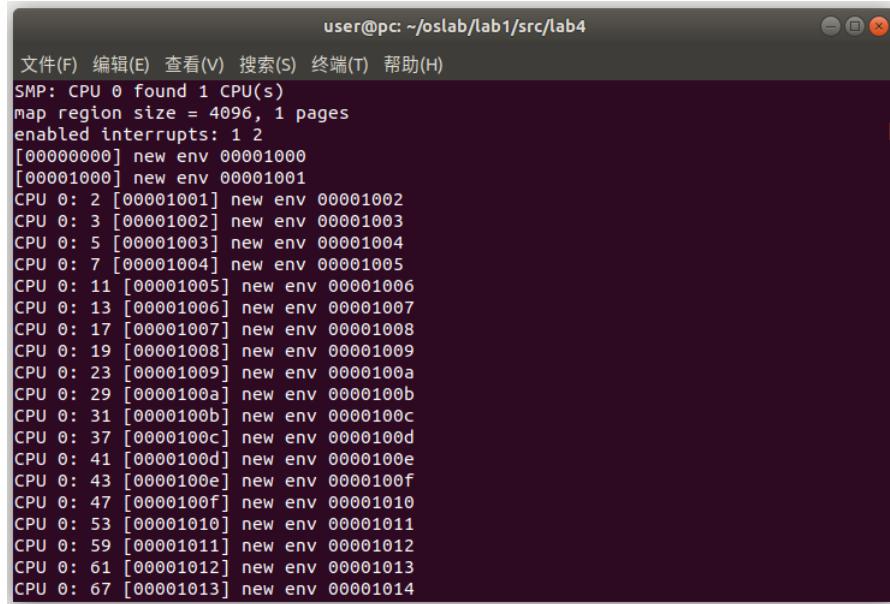
user@pc: ~/oslab/lab1/src/lab4
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
SMP: CPU 0 found 1 CPU(s)
map region size = 4096, 1 pages
enabled interrupts: 1 2
[00000000] new env 00001000
[00001000] new env 00001001
send 0 from 1000 to 1001
1001 got 0 from 1000
1000 got 1 from 1001
1001 got 2 from 1000
1000 got 3 from 1001
1001 got 4 from 1000
1000 got 5 from 1001
1001 got 6 from 1000
1000 got 7 from 1001
1001 got 8 from 1000
1000 got 9 from 1001
[00001000] exiting gracefully
[00001000] free env 00001000
1001 got 10 from 1000
[00001001] exiting gracefully
[00001001] free env 00001001
No runnable environments in the system!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.

```

Figure 83: make run-pingpong

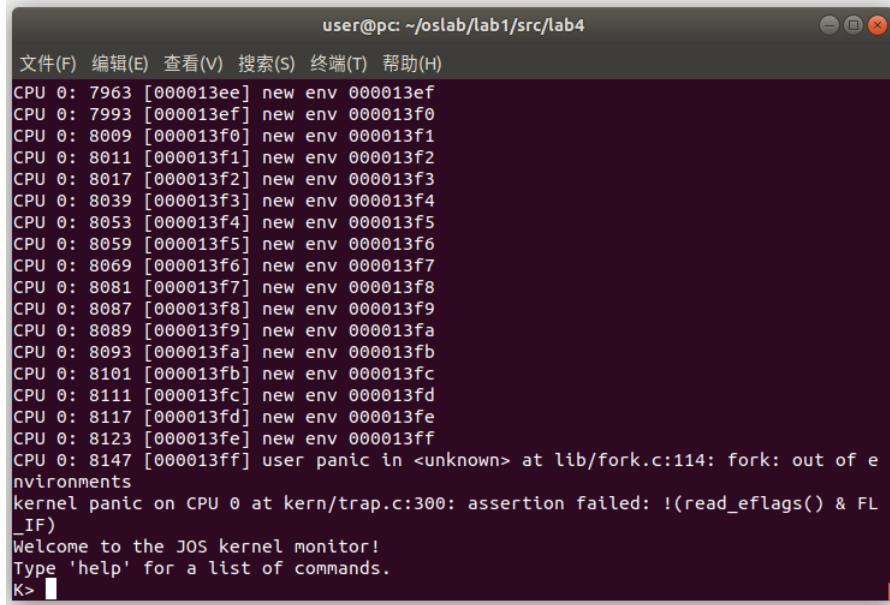
输入命令`make run-primes`的结果如下。可以看到前两个进程是内核进程，后面

的进程都是用户进程，每一个素数都对应一个用户进程，直到 8147 时没有新的进程页可分配了：



```
user@pc: ~/oslab/lab1/src/lab4
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
SMP: CPU 0 found 1 CPU(s)
map region size = 4096, 1 pages
enabled interrupts: 1 2
[00000000] new env 00001000
[00001000] new env 00001001
CPU 0: 2 [00001001] new env 00001002
CPU 0: 3 [00001002] new env 00001003
CPU 0: 5 [00001003] new env 00001004
CPU 0: 7 [00001004] new env 00001005
CPU 0: 11 [00001005] new env 00001006
CPU 0: 13 [00001006] new env 00001007
CPU 0: 17 [00001007] new env 00001008
CPU 0: 19 [00001008] new env 00001009
CPU 0: 23 [00001009] new env 0000100a
CPU 0: 29 [0000100a] new env 0000100b
CPU 0: 31 [0000100b] new env 0000100c
CPU 0: 37 [0000100c] new env 0000100d
CPU 0: 41 [0000100d] new env 0000100e
CPU 0: 43 [0000100e] new env 0000100f
CPU 0: 47 [0000100f] new env 00001010
CPU 0: 53 [00001010] new env 00001011
CPU 0: 59 [00001011] new env 00001012
CPU 0: 61 [00001012] new env 00001013
CPU 0: 67 [00001013] new env 00001014
```

Figure 84: make run-primes



```
user@pc: ~/oslab/lab1/src/lab4
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
CPU 0: 7963 [000013ee] new env 000013ef
CPU 0: 7993 [000013ef] new env 000013f0
CPU 0: 8009 [000013f0] new env 000013f1
CPU 0: 8011 [000013f1] new env 000013f2
CPU 0: 8017 [000013f2] new env 000013f3
CPU 0: 8039 [000013f3] new env 000013f4
CPU 0: 8053 [000013f4] new env 000013f5
CPU 0: 8059 [000013f5] new env 000013f6
CPU 0: 8069 [000013f6] new env 000013f7
CPU 0: 8081 [000013f7] new env 000013f8
CPU 0: 8087 [000013f8] new env 000013f9
CPU 0: 8089 [000013f9] new env 000013fa
CPU 0: 8093 [000013fa] new env 000013fb
CPU 0: 8101 [000013fb] new env 000013fc
CPU 0: 8111 [000013fc] new env 000013fd
CPU 0: 8117 [000013fd] new env 000013fe
CPU 0: 8123 [000013fe] new env 000013ff
CPU 0: 8147 [000013ff] user panic in <unknown> at lib/fork.c:114: fork: out of environments
kernel panic on CPU 0 at kern/trap.c:300: assertion failed: !(read_eflags() & FL_IF)
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> 
```

Figure 85: make run-primes

至此，Lab 4 的 Part C 已完成。以下是在终端输入命令 `make grade` 之后得到的打分情况：

```
user@pc: ~/oslab/lab1/src/lab4
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
dumbfork: OK (1.9s)
Part A score: 5/5

faultread: OK (0.7s)
faultwrite: OK (0.7s)
faultdie: OK (1.2s)
faultregs: OK (1.6s)
faultalloc: OK (2.3s)
faultallocbad: OK (1.9s)
faultnystack: OK (1.9s)
faultbadhandler: OK (2.0s)
faultevilhandler: OK (1.2s)
forktree: OK (1.8s)
Part B score: 50/50

spin: OK (1.3s)
stresssched: OK (2.0s)
sendpage: OK (1.8s)
pingpong: OK (2.0s)
primes: OK (5.2s)
Part C score: 25/25

Score: 80/80
user@pc:~/oslab/lab1/src/lab4$
```

Figure 86: Part C 打分情况

5 总结

Lab 4 真的很难做，我得到这个满分的`make grade`比前面的任何一个实验都要难。马上就要考试了，而且这也是最后一个实验，大家的积极性似乎都不高。做完 Lab 4 以后，深入理解了操作系统进程调度以及 IPC、共享内存机制，但是要真正懂还是要看书。