

# 操作系统第三次上机作业实验报告

成员 1:

专业:计算机科学与技术

学号:1310617

姓名:刘丹

成员 2:

专业:计算机科学与技术

学号:1310636

姓名:赵婧涵

# 一 多处理器支持与协作式多任务

## 1.1.多处理器支持(Multiprocessor Support)

为 JOS 添加"symmetric multiprocessing" (SMP)支持,这种多处理器模式允许 CPUs 有同等的机会获得系统资源比如内存、IO 总线等。虽然在 SMP 中所有 CPUs 功能相同,但是启动过程可分为两类:the bootstrap processor (BSP) 是负责系统初始化和启动操作系统,the application processors (APs)只有在操作系统启动完成后才由 BSP 激活。哪个处理器是 BSP 由硬件和 BIOS 决定,到目前为止,所有的 JOS 代码都是运行在 BSP 上。在一个 SMP 系统中,每个 CPU 有一个伴随的局部 APIC (local APIC, LAPIC)单元。The LAPIC units 负责在系统中传输中断,也为它连接的 CPU 提供唯一标识符。在本次试验中,我们要利用 LAPIC units 的功能(in kern/lapic.c):

- 读取 LAPIC ID(LAPIC identifier)来获得我们的 code 当前运行在哪一个 CPU(see cpunum())上.

- 在第三部分实验中,我们对 LAPIC 内置计时器编程来触发时钟中断以支持抢占式多任务(see apic\_init())一个处理器通过 memory-mapped I/O (MMIO)访问它的 LAPIC。在 MMIO 上,物理存储器的一部分是天生的在一些 I/O 设备的寄存器(register)上,因此用于访问内存的加载/存储指令(load/store instructions)通常也可用于访问设备寄存器(register)。物理地址是 0xA0000 的 IO hole(we use this to write to the CGA display buffer)。the LAPIC lives in a hole starting at physical address 0xFE000000 (32MB short of 4GB),所以它对我们通常在 KERNBASE 的直接映射来说太高了。所以在本次实验中,调整 JOS 的 memory layout ,映射内核虚拟空间的 top 32MB,即从 IOMEMBASE (0xFE000000)开始,到包含 the LAPIC 的 theIO hole. Since this region starts at physical address 0xFE000000, this is an identity mapping.我们已经在 kern/pmap.c 的 mem\_init\_mp()完成了新的映射,并更新了 inc/memlayout.h。

## 1.2. Application Processor Bootstrap

启动 APS 之前,BSP 首先应该收集有关多处理机系统信息,如总的 CPU 数量,它们的 APIC ID 和 LAPIC 单元的 MMIO 地址。

在 kern/mpconfig.c 的 mp\_init()函数通过读取驻存在 BIOS 的 MP configuration table 获得这些信息。

函数 `boot_aps()`(in `kern/init.c`)引导 APS 启动。APS 在实模式启动,和 `bootloader` 引导过程非常相像(`boot/boot.S`),`boot_aps()`复制 AP entry code(`kern/mpentry.S`)到实模式可寻址到的一处内存地址。但是和 `bootloader` 不同的是,我们对 AP 在哪执行代码有一些控制,我们 copy the entry code to `0x7000` (`MPENTRY_PADDR`), but any unused, page-aligned physical address below 640KB would work.

之后,`boot_aps()`通过发送 `STARTUP IPS` 到 AP 相对应的 `LAPIC` 单元激活 AP,同时初始化该 AP 运行它 entry code (`MPENTRY_PADDR` in our case)的 `CS:IP` 地址。The entry code (in `kern/mpentry.S`)与 `boot/boot.S` 非常相似。

一些简单的启动步骤之后,它将 AP 设置为保护模式,然后调用 C setup routine `mp_main()`(also in `kern/init.c`)。

`boot_aps()`收到 the AP 发送 `CPU_STARTED` flag(in `cpu_status` field of its struct `Cpu`)后再去唤醒下一个 AP。

## 作业 1

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).

## 代码阅读及理论准备:

多核 CPU 启动流程:

在内核加载和进行初始化时,即便是有一个多核 CPU,也只能使用一个核,为了和 JOS 实验中的描述一致,我们这里暂且把多个核 (Core) 称之为多个 CPU。

内核启动过程中用于执行代码的 CPU 叫做 `bootstrap processor` (BSP), 其它还未被使用的 CPU 叫做 `application processors` (APs), 所以在内核执行的时候必然会有有一个过程就是使用 BSP 来激活 AP 的过程。

对应 JOS 代码是 init.c 中的 init 主函数，其关键片段如下：

```
// Lab 2 memory management initialization functions
mem_init();

// Lab 3 user environment initialization functions
env_init();
trap_init();

// Lab 4 multiprocessor initialization functions
mp_init();
lapic_init();

// Lab 4 multitasking initialization functions
pic_init();

// Acquire the big kernel lock before waking up APs
// Your code here:

// Starting non-boot CPUs
boot_aps();

// Should always have idle processes at first.
int i;
for (i = 0; i < NCPU; i++)
    ENV_CREATE(user_idle, ENV_TYPE_IDLE);
```

lab2 的 mem\_init，lab3 的 env\_init 和 trap\_init，lab4 的 mp\_init 和 lapic\_init 在这之后需要补充一个 Big kernel Lock 暂时不用管，然后 boot\_ap 函数启动所有的 CPU，接着有多少个 CPU 就建立多少个 ENV。

在启动过程中，mp\_init 和 lapic\_init 是和硬件以及体系架构紧密相关的，通过读取某个特殊内存地址（当然前提是能读取到的，所以在 mem\_init 中需要修改进行相应映射），来获取 CPU 的信息，根据这些信息初始化 CPU 结构。

在函数 boot\_ap() 中首先找到一段用于启动的汇编代码，该代码和 上一章实验一样是嵌入在内核代码段之上的一部分，其中 mpendry\_start 和 mpendry\_end 是编译器导出符号，代表这段代码在内存（虚拟地址）中的起止位置，接着把代码复制到 MPENTRY\_PADDR 处。随后调用 lapic\_startap 来命令特定的 AP 去执行这段代码。

这段汇编（mpentry.S）中所做的工作和 entry.S 所做的工作基本相同。需要注意

的一点是每个 CPU 都有自己的寄存器和栈，需要启动的 AP 目前还处于实模式，并且内存也没有开启分页，因此所有和符号地址相关的操作都要非常谨慎的转换为物理地址。

之后控制流跳转到 mp\_main，值得注意的是从 mpendtry.S 开始这些代码都是 执行在 AP 上的，此时 BSP 正在等待（while 循环）AP 启动成功。在 mp\_main 里可以看到 AP 初始化自己的 env, trap 等，接着改变自己的 cpu 数据结构中的 cpu 状态标志为启动成功，然后进入死循环空转。BSP 得到 AP 启动成功的信息后接着尝试启动下一个 AP。

设计思路:

因为我们把代码拷贝到了 MPENTRY\_PADDR 处，所以要在初始化 Page 的时候把这一页从 Page\_Free\_List 中取出来，以防止该页被分配出去导致代码拷贝的失败进而不能正确启动 AP。只需要在 pmap.c 的 page\_init 函数单独处理 MPENTRY\_PADDR 即可。

代码实现:

```
for (i = 1; i < npages_basemem; i++)
{
    if( i == MPENTRY_PADDR / PGSIZE)
    {
        pages[i].pp_ref = 1;
        pages[i].pp_link = NULL;
        continue;
    }
    pages[i].pp_ref = 0;
    pages[i].pp_link = page_free_list;
    page_free_list = &pages[i];
}
int med = (int)ROUNDUP(((char*)envs) + (sizeof(struct Env) * NENV) - 0xf0000000, PGSIZE)/PGSIZE;
for (i = med; i < npages; i++)
{
    pages[i].pp_ref = 0;
    pages[i].pp_link = page_free_list;
    page_free_list = &pages[i];
}
```

检测结果:

```
diana@ubuntu: ~/lab3
diana@ubuntu:~$ cd lab3/
diana@ubuntu:~/lab3$ make
+ cc kern/pmap.c
+ ld obj/kern/kernel
+ mk obj/kern/kernel.img
diana@ubuntu:~/lab3$ make qemu

(process:2329): GLib-WARNING **: /build/glib2.0-ajuDY6/glib2.0-2.46.1/./glib/gmem.c:482: custom memory allocation vtable not supported
/home/diana/os/qemu-6.828-1.7.0/i386-softmmu/qemu-system-i386 -hda obj/kern/kernel.img -serial mon:stdio -gdb tcp::26000 -D qemu.log -smp 1

(process:2331): GLib-WARNING **: /build/glib2.0-ajuDY6/glib2.0-2.46.1/./glib/gmem.c:482: custom memory allocation vtable not supported
VNC server running on `127.0.0.1:5900'
6828 decimal is 15254 octal!
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
check_page() succeeded!
kernel panic on CPU 0 at kern/pmap.c:862: assertion failed: check_va2pa(pgdir, i) == i
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> 
```

pmap.c mem\_init()中:

```
page_init();

check_page_free_list(1);
check_page_alloc();
check_page();
```

检测显示:

check\_page\_alloc() succeeded!

check\_page() succeeded!

故前面的 check\_page\_free\_list(1)成功(未输出错误).

## 问题 1

Compare kern/mpentry.S side by side with boot/boot.S. Bearing in mind that kern/mpentry.S is compiled and linked to run above KERNBASE just like everything else in the kernel, what is the purpose of macro MPBOOTPHYS? Why is it necessary in kern/mpentry.S but not in boot/boot.S? In other words, what could go wrong if it were omitted in kern/mpentry.S?

Hint: recall the differences between the link address and the load address that we have discussed in Lab 1.

## Answer:

在 AP 的保护模式打开之前，是没有办法寻址到 3G 以上的空间的，因此用 MPBOOTPHYS 是用来计算相应的物理地址的。

但是在 boot.S 中，由于尚没有启用分页机制，所以我们能够指定程序开始执行的地方以及程序加载的地址；而在 mpentry.S 的时候，由于主 CPU 已经处于保护模式下了，因此是不能直接指定物理地址的，给定线性地址，映射到相应的物理地址是允许的。

## 1.3. Per-CPU State and Initialization

在写多处理器的 OS 时,能够区分每个处理器的私有 per-CPU state 和整个系统共享的 global state 是很重要的。kern/cpu.h 定义了大部分的 per-CPU state,包括 struct Cpu(存储 per-CPU variables)。cpunum()返回调用它的 CPU 的 ID,这可以用来作为如 cpus array 的索引。另外,宏 thiscpu 是当前 CPU 的 cpu struct 的简写。

对于 per-CPU state 要清楚的包括:

- Per-CPU kernel stack

因为多 CPUs 可同时捕捉内核,因此我们需要为每个处理器分配独立执行 kernel stack,以防止它们互相干扰。

在之前作业中,已经将 bootstack 映射的物理内存(just below KSTACKTOP)作为 BSP 的 kernel stack。相似的,在本次试验中将每个 CPU 的内核堆栈映射到这个区域,其

中保护页(guard pages)充当它们之间的缓冲区。 CPU 0's stack 依然从 KSTACKTOP 向下增长, CPU 1's stack 从 CPU 0's stack 底部的 KSTK GAP bytes 之后开始,以此类推。 inc/memlayout.h 包含了这一新的映射。

#### □ Per-CPU TSS and TSS descriptor

每一个 cpu 的 task state segment (TSS)被用来指定每一个 CPU 的内核栈存在的地方, The TSS for CPU i is stored in cpus[i].cpu\_ts,相应的 TSS descriptor 定义在 gdt[(GD\_TSS0 >> 3) + i]。 kern/trap.c 的全局变量 ts 将不再使用。

#### □ Per-CPU current environment pointer

因为每一个 cpu 同步的运行不同的用户环境,所以我们重新定义 curenv 为 cpus[cpunum()].cpu\_env (or thiscpu->cpu\_env),这个变量指向了正运行在当前 CPU 的环境。

#### □ Per-CPU system registers

所有 register 包括 system register 对 cpu 都是私有的,因此初始化 register 的指令,如 lcr3(),ltr(), lgdt(), lidt() 等等都需要对每个 CPU 都执行一次, Functions env\_init\_percpu() and trap\_init\_percpu() are defined for this purpose.

#### □ Per-CPU idle environment

当没有足够的常规环境运行时,JOS 使用 idle environment 作为备用(fallback)运行。一个环境只能在同一时间在一个 CPU 上运行。由于多个 CPUS 可在同一时间空闲,因此要为每个 CPU 创建一个空闲环境(idle environment)。 envs[cpunum()]表示当前 CPU 的空闲环境。



## 作业 2

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()`.

### `mem_init_mp()`:

函数作用:为每一个核映射其内核栈

设计思路:对于编号为 `i` 的 `cpu` , 需要映射 `KSTACKTOP-i*(KSTKSIZE+KSTKGAP)-KSTKSIZE` 到 `KSTACKTOP-i*(KSTKSIZE+KSTKGAP)` 这块虚拟地址空间到符号 `percpu_kstacks[i]` 所对应的物理地址处.

代码实现:

```
// LAB 4: Your code here:
//为每一个核映射其内核栈
uint32_t top_of_i, end_of_i;

for(int i = 0; i < NCPU; i++){
    top_of_i = KSTACKTOP - i * (KSTKSIZE + KSTKGAP);    // 栈顶位置
    end_of_i = top_of_i - KSTKSIZE;                      // 栈底位置
    boot_map_region(kern_pgdir, end_of_i, KSTKSIZE, PADDR(&percpu_kstacks[i]), PTE_W); //映射
}
```

检测结果:

```
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2
[00000000] new env 00001000
[00000000] new env 00001001
[00000000] new env 00001002
[00000000] new env 00001003
[00000000] new env 00001004
[00000000] new env 00001005
[00000000] new env 00001006
[00000000] new env 00001007
[00000000] new env 00001008
```

## 作业 3

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.)

设计思路:每一个 cpu 的 task state segment (TSS)被用来指定每一个 CPU 的内核栈存在的地方, The TSS for CPU *i* is stored in `cpus[i].cpu_ts`,相应的 TSS descriptor 定义在 `gdt[(GD_TSS0 >> 3) + i]`。 `kern/trap.c` 的全局变量 `ts` 将不再使用。

代码实现:

```
// LAB 4: Your code here:

// Setup a TSS so that we get the right stack
// when we trap to the kernel.

thiscpu->cpu_ts.ts_esp0 = KSTACKTOP - (thiscpu->cpu_id) * (KSTKSIZE + KSTKGAP); //当前CPU
thiscpu->cpu_ts.ts_ss0 = GD_KD;

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

检测结果: make qemu CPUS=4

```
diana@ubuntu: ~/lab4
(process:2759): Glib-WARNING **: /build/glib2.0-ajuDY6/glib2.0-2.46.1/./glib/gme
m.c:482: custom memory allocation vtable not supported
VNC server running on '127.0.0.1:5900'
6828 decimal is 15254 octal!
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 4 CPU(s)
enabled interrupts: 1 2
SMP: CPU 1 starting
SMP: CPU 2 starting
SMP: CPU 3 starting
[00000000] new env 00001000
[00000000] new env 00001001
[00000000] new env 00001002
[00000000] new env 00001003
[00000000] new env 00001004
[00000000] new env 00001005
[00000000] new env 00001006
[00000000] new env 00001007
[00000000] new env 00001008
[00000000] new env 00001009
```

## 1.4. Locking

我们目前的代码停在初始化 AP(in mp\_main()),在让 AP 做其他事情之前,我们需要先解决多 CPUS 同步运行 kernel code 时的 race conditions。最简单的实现方式是使用 big kernel lock。

The big kernel lock 是唯一的全局锁,当一个环境进入内核态(kernel mode)时上锁,返回到用户态(user mode)时释放锁。在这种模式下,用户态的环境可以在任何可用的 CPU 上并发(concurrently)进行,但是最多只有一个环境可以进入内核态,其他也需要进入内核态的环境被迫等待。kern/spinlock.h 声明了 the big kernel lock,命名为 kernel\_lock,它提供了 lock\_kernel()和 unlock\_kernel()用来加锁和释放锁。你可以在四个位置申请锁资源:

- In i386\_init(),在 BSP 唤醒其他 CPUS 之前取得锁。

- In mp\_main(),在 AP 初始化后获得锁,然后调用 sched\_yield()运行在该 AP 上的环境。

□ In trap(),当在用户态时发生 trap 时获得锁,为了判断 trap 发生在用户态还是内核态,检查 tf\_cs 的低位。

□ In env\_run(),切换用户模式之前释放锁(release the lock right before switching to user mode),不要太早也不要太晚,否则会造成不必要竞争或死锁。

## 作业 4

Apply the big kernel lock as described above, by calling lock\_kernel() and unlock\_kernel() at the proper locations.

### 理论准备(内核锁):

考虑到当多个 CPU 同时陷入内核的场景,若对于关键数据结构不加锁必然就会导致重入错误(如 cprintf 不加锁会在屏幕上输出奇怪的结果),因此使用锁来保证内核函数内部的逻辑正确性是很有必要的。

内核锁相关代码都在 spinlock.c 和 spinlock.h 中,关键代码如下:

```
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
}
```

以及 x86.h 里面的 xchg 函数:

```

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;
}

```

可以看到，xchg 函数首先使用 lock 指令使 xchgl 变为原子操作，然后尝试将 xchgl 两个操作数互换，并把原先第一个操作数的结果放入 result 中返回。

若此时锁是空闲的，则 xchg 返回 0，spin\_lock 函数执行完成，否则继续执行 pause 指令，然后接着执行 xchg 函数直到其返回值为 1。

关于 lock 引用一段汇编手册的资料：

总线加锁前缀“lock”，它是为了在多处理器环境中，保证在当前指令执行期间禁止一切中断。这个前缀仅仅对 ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, XCHG 指令有效，如果将 Lock 前缀用在其它指令之前，将会引起异常。

关于 pause：

提升 spin-wait-loop 的性能，当执行 spin-wait 循环的时候，笨死和小强处理器会因为在退出循环的时候检测到 memory order violation 而导致严重的性能损失，pause 指令就相当于提示处理器目前处于 spin-wait 中。在绝大多数情况下，处理器根据这个提示来避免 violation，藉此大幅提高性能，由于这个原因，我们建议在 spin-wait 中加上一个 pause 指令。（出自于 intel 汇编手册）

在以下 4 个地方加锁：

- (1) i386\_init 中，启动多个 ap 之前。
- (2) mp\_main 中，开始把任务调度到 cpu 上之前。

(3) trap 中，若从用户态陷入内核则加锁。

(4) env\_run 中，从内核态返回用户态需要释放锁。

加锁后，将原有的并行执行过程在关键位置变为串行执行过程，整个启动过程大概如下：

i386\_init-->BSP 获得锁-->boot\_ap--> (BSP 建立为每个 cpu 建立 idle 任务、建立用户任务，mp\_main)--->BSP 的 sched\_yield-->其中的 env\_run 释放锁-->AP1 获得锁-->执行 sched\_yield-->释放锁-->AP2 获得锁-->执行 sched\_yield-->释放锁.....其中括号表示并行执行。

代码实现:

i386\_init()(kern/init.c):

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

mp\_main()(kern/init.c):

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

trap()(kern/trap.c):

```
// LAB 4: Your code here.
if(tf->tf_cs!=GD_KT)
    lock_kernel();
```

```

env_run()(kern/env.c):

// LAB 3: Your code here.

if(curenv!=NULL)
    if(curenv->env_status==ENV_RUNNING)
        curenv->env_status=ENV_RUNNABLE;

e->env_status=ENV_RUNNING; //设置当前进程的一些信息
e->env_runs++;
curenv=e; //更改当前进程指针指向要运行的进程

unlock_kernel();

lcr3(PADDR(e->env_pgdir));
env_pop_tf(&e->env_tf); //加载进程页目录跳转到使用env_pop_tf真正使进程执行

```

现在还不能检测锁是否是正确的,完成下一个作业的调度之后才可以检测.

## 问题 2

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

## Answer:

我们描述这样一种情况，考虑 CPU0 和 CPU1

- (1) CPU0 运行中，遇到持续等待某个中断发生
- (2) CPU1 开始运行，CPU1 将保存在 task state 中的寄存器的值恢复
- (3) CPU1 持续等待某个中断发生
- (4) CPU0 等待的中断发生，切换到 CPU0

问题出现，因为 CPU1 保存在堆栈中的内容并没有弹出，因此此时转向 CPU0 时，共享栈的内容会出错。

## 1.5.Round-Robin Scheduling

下一个任务是修改 JOS 内核,使得它并不总是运行 idle environments,而是以轮转调度的方式在多个环境中选择。轮转调度工作方式如下:

如前所述,the first NCPU environments 总是运行 special idle environments(run the program user/idle.c)。这个程序的目的仅仅是当处理器无事可做时“浪费时间”。阅读 user/idle.c 的代码和注释了解细节。我们已经通过修改 kern/init.c 创建了那些 special idle environments(in envs[0] though envs[NCPU-1])。

函数 sched\_yield()(in kern/sched.c)负责选择一个新的环境来运行。它以循环的方式顺序搜索 the envs[] array,从上一个已经 running environment 之后开始(如果之前没有 running environment,从 array 头部开始),选择第一个状态为 ENV\_RUNNABLE(see inc/env.h)的 environment,然后调用 env\_run()进入该 environment。不过,sched\_yield()知道哪些是 the special idle environments,除非没有其他 runnable environments,否则不会选择 the special idle environments。sched\_yield()决不在两个处理器同时运行同一环境,它可以分辨一个环境当前是否正在运行在某个 CPU 上,因为如果运行那么这个环境的状态是 ENV\_RUNNING。

我们已经为你实现了一个新的系统调用 sys\_yield(),用户环境可以调用它从而调用 (invoke)内核的 sched\_yield()函数和自愿放弃 CPU 给其他环境。如你在 user/idle.c 所见,the idle environment does this routinely。

当内核从一个环境切换到另一个,它都保证 the old environment's registers 被保存以便以后恢复。



## 作业 5

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 00001008.
Hello, I am environment 00001009.
Hello, I am environment 0000100a.
Back in environment 00001008, iteration 0.
Back in environment 00001009, iteration 0.
Back in environment 0000100a, iteration 0.
Back in environment 00001008, iteration 1.
Back in environment 00001009, iteration 1.
Back in environment 0000100a, 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.

### 理论准备:

在 JOS 中，任务调度在内核中是函数 `sched_yield`，同时在用户态有相应的系统调用 `sys_yield` 也可以调用内核中的这个函数。

`i386_init` 启动时，在 `boot_ap` 函数调用后，为每个 CPU 创建一个 idle 任务，相应代码在 `user/idle.c`，通过代码可以看到，这些任务使用一个死循环，不断调用 `sys_yield` 尝试切换任务。

所以在这种机制下我们需要实现 `sched_yield` 函数，具有以下几个要求：

(1) 找到状态为 `runnable` 的任务，并切换执行。

(2) 如果找到一个 `running` 状态的任务，且此任务执行的 CPU 为当前 CPU，也可将此任务切换执行。

(3) 若没有 runnable 任务，则执行 idle 任务。

(4) 从当前 CPU 执行的任务处开始遍历链表（为了保证公平性）

代码实现:

```
// LAB 4: Your code here.
struct Env *new_env = 0;

// 设置 next_env 为当前 env 的下一个
size_t next_env = 0;
if(curenv)
    next_env = ENVX(ENVX(curenv->env_id));
|
// 从 next_env 循环寻找一个可运行的 env
for(i = 0; i < NENV; i++)
{
    next_env = (next_env + 1) % NENV;
    // 必须是可运行的 非 IDLE 的 env
    if(envs[next_env].env_status == ENV_RUNNABLE
        && envs[next_env].env_type != ENV_TYPE_IDLE)
    {
        new_env = &envs[next_env];
        break;
    }
}
// 如果找到了，则运行
if(new_env)
    env_run(new_env);

// 如果没找到，且当前env可运行，则继续运行当前 env
else
    if(curenv && curenv->env_status == ENV_RUNNING)
        env_run(curenv);
```

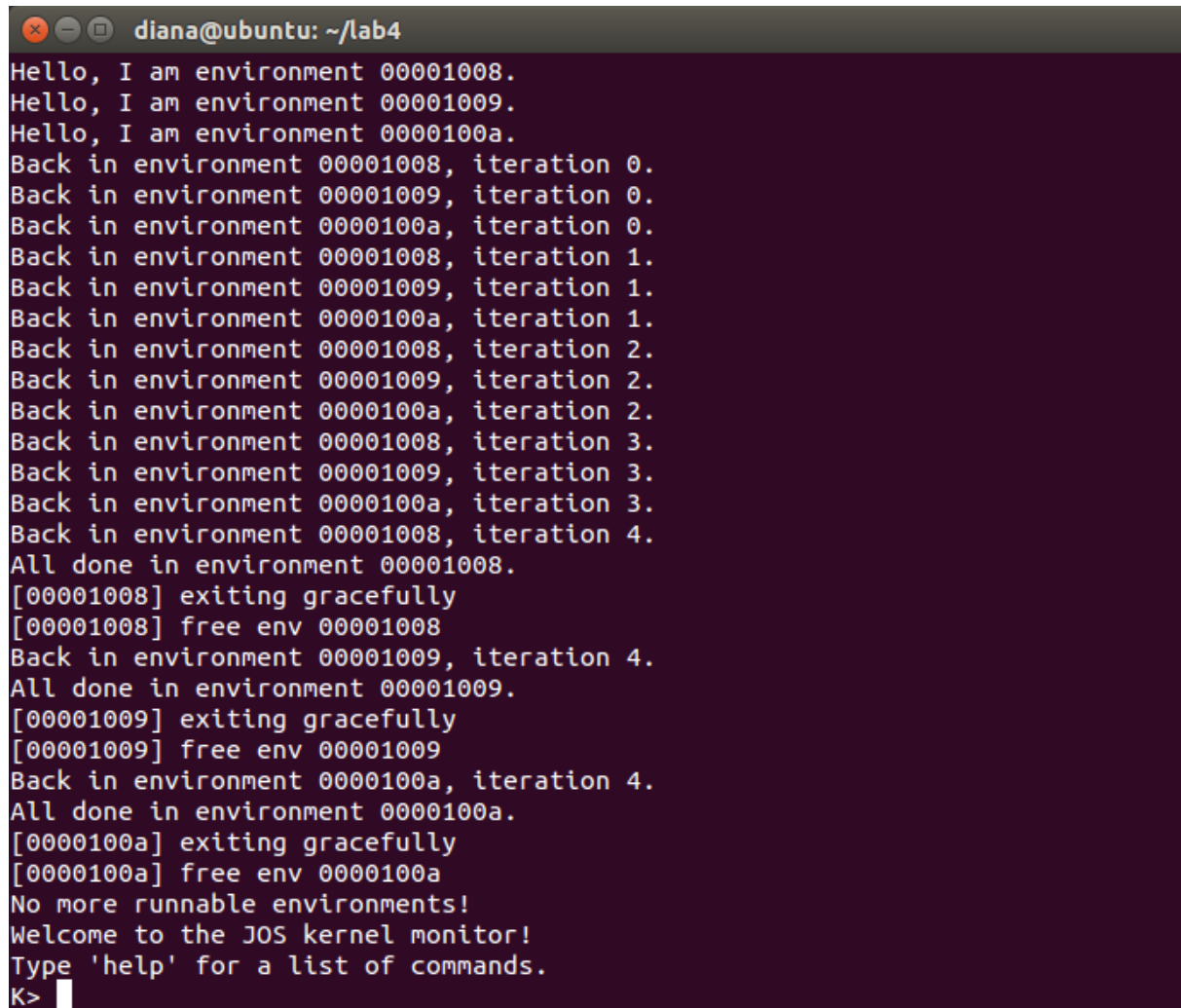
在 syscall 里处理 SYS\_yield 情况:

```
case:SYS_yield:
    SYS_yield();
    ret = 0;
    break;
```

在 init.c 中创建 3 个 user/yield 环境:

```
// Touch all you want.  
for(i=0;i<3;i++)  
    ENV_CREATE(user_yield, ENV_TYPE_USER);
```

检测结果:



```
diana@ubuntu: ~/lab4  
Hello, I am environment 00001008.  
Hello, I am environment 00001009.  
Hello, I am environment 0000100a.  
Back in environment 00001008, iteration 0.  
Back in environment 00001009, iteration 0.  
Back in environment 0000100a, iteration 0.  
Back in environment 00001008, iteration 1.  
Back in environment 00001009, iteration 1.  
Back in environment 0000100a, iteration 1.  
Back in environment 00001008, iteration 2.  
Back in environment 00001009, iteration 2.  
Back in environment 0000100a, iteration 2.  
Back in environment 00001008, iteration 3.  
Back in environment 00001009, iteration 3.  
Back in environment 0000100a, iteration 3.  
Back in environment 00001008, iteration 4.  
All done in environment 00001008.  
[00001008] exiting gracefully  
[00001008] free env 00001008  
Back in environment 00001009, iteration 4.  
All done in environment 00001009.  
[00001009] exiting gracefully  
[00001009] free env 00001009  
Back in environment 0000100a, iteration 4.  
All done in environment 0000100a.  
[0000100a] exiting gracefully  
[0000100a] free env 0000100a  
No more runnable environments!  
Welcome to the JOS kernel monitor!  
Type 'help' for a list of commands.  
K>
```

### 问题 3

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

### Answer:

因为进程的切换是在内核中的，而内核的地址是固定的，也就是无论是哪个进程的页目录，内核都映射到相同的地址，因此，不会出现问题。

## 1.6. System Calls for Environment Creation

尽管你的内核目前能够运行和切换多个用户态环境,但是受限于只能运行那些由内核创建的环境。现在你要实现必要的系统调用以使用户环境也可以创建和启动其他新的用户环境。

UNIX 提供了 `fork()`函数作为创建进程的原型, UNIX 的 `fork` 函数 copy 整个 calling process(the parent )的地址空间来创建新的进程((the child),唯一可观察到的不同的是他们的 process IDs 和他们的 parent process IDs(as returned by `getpid` and `getppid`)。 In the parent, `fork()`returns the child's process ID, while in the child, `fork()` returns 0.默认情况下,每个进程有私有的地址空间,并且内存修改对其他进程不可见。

你现在要提供一个更原始的 JOS 系统调用集合(more primitive set of JOS system calls) 用来创建用户态的环境,使用这些系统调用,你可以在用户空间实现 a Unix-like `fork()`(除了其他 style 的环境创建)。这些系统调用包括:

□ `sys_exofork`:

该系统调用创建一个几乎空白的新环境:没有东西映射到它地址空间的用户部分,也不能运行。在 `sys_exofork` 调用时,这个新环境和它父环境有一样的 registers. In the parent,`sys_exofork` 返回新环境的 `envid_t` (or a negative error code if the environment allocation failed).In the child, it will return 0(因为子环境被标记为不可用,`sys_exofork` 不

会返回,除非它的父环境把它标记为可用)

□ `sys_env_set_status`:

设置某个环境的状态为 `ENV_RUNNABLE` or `ENV_NOT_RUNNABLE`,这个系统调用一般被用来标记一个新环境已经准备好可用了(当它的地址空间和寄存器都已经初始化好)

□ `sys_page_map`:

复制一个环境的 page mapping(not the contents of a page!)到另一个环境,使得新老环境都指向物理内存的同一页。

□ `sys_page_unmap`:

Unmap a page mapped at a given virtual address in a given environment.

以上所有的系统调用都接受 environment IDs,JOS 内核支持简便写法 0 表示"the current environment",简便写法的实现在 `envid2env()` in `kern/env.c`.

我们已经提供了一个 Unix-like `fork()` 函数的原始实现 ( in the test program `user/dumbfork.c`),这个测试程序使用以上的系统调用创建并运行一个新的复制它自己地址空间的环境,这两个环境通过 `sys_yield` 来回切换。 The parent exits after 10 iterations, whereas the child exits after 20.

## 作业 6

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_INVALID` in that case. Test your JOS kernel with `user/dumbfork` and make sure it works before proceeding.

### 理论准备:

`fork` 作为一个系统调用，其功能是根据父进程创建出一个一模一样的子进程，若返回的是 0 则说明是子进程，否则是父进程，同时返回值为子进程的系统调用号。

JOS 实现 `fork` 过程采用的是用户态“类库”的形式封装了一系列系统调用，包括创建新进程、设置新进程状态、虚拟地址映射等等，这部分已经在 `user/dumbfork.c` 中封装好了，实验所要求的是实现相关的系统调用，包括：

`sys_exofork`: 若为父进程返回子进程号，子进程则返回 0。

`sys_env_set_status`: 设置子进程格式为 `runnable` 或者 `not_runnable`

`sys_page_alloc`: 分配一个物理页并对应到某个虚拟地址

`sys_page_map`: 拷贝父进程的某个 PTE，以此来建立子进程的虚拟内存映射

`sys_page_unmap`: 解除某个虚拟地址的映射（在 PART B 中使用）

具体实现(syscall.c 中):

#### (1) `sys_exofork()`

函数作用: 若为父进程返回子进程号，子进程则返回 0。

设计思路: 首先复制各寄存器的状态（`env_tf`），然后系统调用本身返回子进程的 `id`，因为调用此系统调用的进程为父进程。同时将子进程的 `eax` 寄存器设置为 0，因为系统调用的结果存放在 `eax` 寄存器中，这样子进程返回后得到的系统调用返回结果就为 0。

代码实现:

```
// LAB 4: Your code here.
//panic("sys_exofork not implemented");
struct Env *new_env;

if(env_alloc(&new_env, curenv->env_id) < 0)// 创建新环境
    return -E_NO_FREE_ENV;
//首先复制各寄存器的状态 (env_tf)
memcpy(&new_env->env_tf, &curenv->env_tf, sizeof(struct Trapframe));

//将子进程的eax寄存器设置为0, 因为系统调用的结果存放在eax寄存器中, 这样子进程返回后得到的系统调用
//返回结果就为0。|
new_env->env_tf.tf_regs.reg_eax = 0;

// 当前状态为不可运行
new_env->env_status = ENV_NOT_RUNNABLE;

return new_env->env_id;//:若为父进程返回子进程号, 子进程则返回0
```

## (2)sys\_env\_set\_status()

函数作用:设置子进程格式为 runnable 或者 not\_runnable

设计思路:首先判断状态会否合法, 然后设置状态。

代码实现:

```
// LAB 4: Your code here.
//panic("sys_env_set_status not implemented");
struct Env *env;

if(envid2env(env_id, &env, 1) < 0)// env_id 无效
    return -E_BAD_ENV;

if(status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE)//设置为可运行或者不运行态(配合exofork使用)
    return -E_INVALID;

env->env_status = status;// 设置状态

return 0;
```

函数作用:分配一个物理页并对应到某个虚拟地址

设计思路:先检测各种条件再进行分配和进行地址映射

代码实现:

```
// LAB 4: Your code here.
// panic("sys_page_alloc not implemented");

int ret;
struct Env * env;
struct Page * pp;

if ((ret = envid2env(envid, &env, 1)) < 0) //检测是否有效
    return -E_BAD_ENV;

if ((uintptr_t)va >= UTOP || PGOFF(va) != 0) //检测虚拟地址
    return -E_INVALID;

// 检测权限
if (((perm | PTE_SYSCALL) != PTE_SYSCALL) || ((perm | PTE_U | PTE_P) != perm))
    return -E_INVALID;

pp = page_alloc(ALLOC_ZERO); // 分配一个物理页
if (!pp)
    return -E_NO_MEM;

if ((ret = page_insert(env->env_pgdir, pp, va, perm)) < 0) //对应到某个虚拟地址
{
    page_free(pp);
    return -E_NO_MEM;
}

return 0;
```



#### (4)sys\_page\_map()

函数作用:建立子进程的虚拟内存映射

设计思路: 拷贝父进程的某个 PTE,进行映射

代码实现:

```
// LAB 4: Your code here.
//panic("sys_page_map not implemented");
struct Env *srcenv, *dstenv;
int res;

// 检查用户环境是否有效
if((res = envid2env(srcenvid, &srcenv, 1)) < 0)
    return res;

if((res = envid2env(dstenvid, &dstenv, 1) < 0))
    return res;

// 检查 va 是否超过UTOP , va 是否为一页的首地址
if ((uintptr_t)srcva >= UTOP || (uintptr_t)dstva >= UTOP
    || PGOFF(srcva) != 0 || PGOFF(dstva) != 0)
    return -E_INVALID;

// 检查 srcenv 是否有权访问 srcva
struct Page *pi;
pte_t *src_pte;
pi = page_lookup(srcenv->env_pgdir, srcva, &src_pte);
if(!pi)
    return -E_INVALID;

// 检查权限
if (((perm | PTE_SYSCALL) != PTE_SYSCALL) ||
    (!(*src_pte & PTE_W) && (perm & PTE_W)))
    return -E_INVALID;

// 地址映射
if (page_insert(dstenv->env_pgdir, pi, dstva, perm) < 0)
    return -E_NO_MEM;

return 0;
```

---

#### (5)sys\_page\_unmap()

函数作用:解除某个虚拟地址的映射 (在 PART B 中使用)

设计思路:借助 page\_remove()即可

代码实现：

```
// LAB 4: Your code here.
//panic("sys_page_unmap not implemented");
int res;
struct Env *env;

// 检查环境是否有效
if(envid2env(envid, &env, 1) < 0)
    return -E_BAD_ENV;

// 检查va是否超过UTOP, va是否为一页的首地址
if ((uintptr_t)va >= UTOP || (PGOFF(va) != 0))
    return -E_INVALID;

// 检查 env 是否有权访问 va
struct Page *pi;
pte_t *src_pte;
pi = page_lookup(env->env_pgdir, va, &src_pte);

if(!pi)
    return -E_INVALID;

page_remove(env->env_pgdir, va);

return 0;
```

在 syscall() 函数里添加相应处理：

```
case SYS_exofork:
    | ret = sys_exofork();
    break;
case SYS_env_set_status:
    ret = sys_env_set_status(a1, a2);
    break;
case SYS_page_alloc:
    ret = sys_page_alloc(a1, (void *)a2, a3);
    break;
case SYS_page_map:
    ret = sys_page_map(a1, (void *)a2, a3, (void *)a4, a5);
    break;
case SYS_page_unmap:
    ret = sys_page_unmap(a1, (void *)a2);
    break;
```

检测结果：

在 init.c 中设置环境为

```
ENV_CREATE(user_dumbfork, ENV_TYPE_USER);|
```

```
diana@ubuntu: ~/lab4
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!
[00001008] exiting gracefully
[00001008] free env 00001008
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!
[00001009] exiting gracefully
[00001009] free env 00001009
No more runnable environments!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

make grade:

```
+ mk obj/kern/kernel.img
make[1]: Leaving directory '/home/diana/lab4'
sh ./grade-lab4.sh
gmake[1]: Entering directory '/home/diana/lab4'
gmake[1]: Nothing to be done for 'all'.
gmake[1]: Leaving directory '/home/diana/lab4'
dumbfork: OK (2.2s)
Part A score: 5/5
```

partA 成功完成!

## 二 写时复制的 Fork(Copy-on-Write Fork)

如前所述,Unix 提供系统调用 Fork()其主要的进程创建方式。系统调用 fork()通过对所调用进程(父)的地址空间进行复制的方式来创建新的进程(子)。

UNIX xv6 的 fork()把父进程的整个数据段(data segment)复制到为子进程新分配的内存区域。这与 dumbfork()的实现是一致的。将父进程的地址复制到子进程是整个 fork()调用过程中代价最大的一部分。

通常情况下调用 fork()之后,子进程就会调 exec()函数以一个新的程序来覆盖当前子进程的空间。大多数情况下,shell 程序就是这么做的。这样,花费在拷贝父进程地址空间的时间被浪费了,因为子进程在调用 exec()之前只使用很少的内存。

基于这个原因,后续版本的 Unix 利用虚拟内存硬件,允许父进程和子进程共享同一块能分别映射到各自地址空间的内存区,直到其中的一个进程需要修改这片内存。这就是写时复制(copy-on-write,也翻译为写前拷贝)。基于这个目的,内核在执行 fork()时,将只拷贝父进程地址空间的映射到子进程,而不是映射页的内容,并且将当前这部分共享页的属性设置为只读的(read-only)。也就是说,两个进程共享了它们的内存区域。当这两个进程中的一个想要修改一个共享的内存页时,程序发生一个缺页错误(page fault)。此时,Unix 内核意识到这个页是一个虚拟的或者说是 copy-on-write 的,所以就会为这个程序创建一个新的私有的可写的内存页。这样,页的内容直到在被修改时才进行复制。这个优化使得 fork()之后在子进程调用 exec()变得更快:子进程在调用 exec()之前也许只需要拷贝一个内存页(堆栈的所在页)。

在接下来的试验中,我们将实现一个合适的类似于 Unix 的 fork()函数,作为用户的库函数,添加 copy-on-write 优化。在用户空间实现 fork()函数和 copy-on-write 支持的好处是使得内核依旧十分简单因而不容易出错。这也支持用户程序定义自己的 fork()。如果程序需要 fork()的不同实现(比如类似于每次都进行拷贝的 dumbfork(),或者父子进程一直共享内存),可以自行实现。

### 2.1. 用户级缺页处理

一个用户级 copy-on-write 的 fork()需要知道缺页错误(page faults)是否是因一个写保护页引起的,这是你首先要实现的。copy-on-write 只是用户级缺页处理的多种用法之一。

通常应该建立起一个内存地址空间的场景,使得在缺页错误发生时能够明确该做些什么。例如,大多数 Unix 内核在初始化时只为新进程的栈区映射一个页的空间,然后在经常用尽栈空间以及在未映射的栈地址处引发缺页错误时,分配并映射额外的栈页空间。一个典型的 Unix 内核应该知道在一个进程空间的每个区域发生缺页错误时该怎样处理。例如,一个发生在栈区域的缺页错误将引发分配和映射新的物理页;一个发生在 BSS 段区域的缺页错误将引发分配新的物理页,将之填充为零并把它映射到正确的地址上。在系统执行过程中,一个发生在程序正文段区域的缺页错误将导致从磁盘读入与其对应的的二进制数据至内存页,并把它映射到正确的地址上。

### 2.1.1. 设置缺页处理函数

#### 作业 7

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。

#### 理论准备:

为了让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(page fault handler entry point)。用户进程可以使用新的 `sys_env_set_pgfault_upcall` 系统调用来设置这个入口点。在 `Env` 结构中,已经添加了一个成员 `env_pgfault_upcall` 来记录这个入口信息。

#### 代码实现:

```
static int
sys_env_set_pgfault_upcall(envid_t envid, void *func)
{
    // LAB 4: Your code here.
    //panic("sys_env_set_pgfault_upcall not implemented");
    struct Env *env;

    if( envid2env(envid, &env, 1) < 0) // 检查 env id 有效性
        return -E_BAD_ENV;

    env->env_pgfault_upcall = func; // 设置处理函数

    return 0;
}
```

在 `syscall()` 函数里添加相应处理：

```
case SYS_env_set_pgfault_upcall:
    ret = sys_env_set_pgfault_upcall(a1, (void *)a2);
    break;
```

### 2.1.2. 调用用户的缺页处理函数

现在, 我们需要修改 `kern/trap.c` 中的缺页处理的代码来在用户模式下处理缺页错误。在此引入一个定义陷入时状态(trap-time state): 指进程在发生缺页错误时的状态。如果没有缺页处理函数被注册, 那么 JOS 内核将会像前面一样销毁用户进程。否则, 内核按照 `lib/trap.h` 中 `struct UTrapframe` 的格式设置异常堆栈。

之后内核会安排用户程序在异常堆栈上运行缺页处理函数来处理这个异常。变量 `fault_va` 是导致这个缺页错误的虚拟地址。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` 这里设置新的堆栈。需要首先压入一个 32 位空字(word), 然后是一个 `UTrapframe` 结构。要检验 `tf->tf_sep` 是否已经在用户的异常堆栈上, 检查它是否是在 `UXSTACKTOP-PGSIZE` 到 `UXSTACKTOP-1` 之间的区域即可。

## 作业 8

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

`page_fault` 该函数负责跳转到用户态的 `upcall` (也就是 `pfentry.S`) , 并为用户态的 `page_fault_handler` 设置好参数。

```
// LAB 4: Your code here.

//构造数据结构, 并复制, 这个数据结构将传递给用户态的处理函数
if(curenv->env_pgfault_upcall)
{
    struct UTrapframe *utf;

    if((uintptr_t)(UXSTACKTOP - PGSIZE) <= tf->tf_esp && tf->tf_esp < (uintptr_t)UXSTACKTOP)
        // 异常发生在缺页处理函数中
        utf = (struct UTrapframe *) (tf->tf_esp - 4 - sizeof(struct UTrapframe));

    else
        // 异常发生在正常代码中
        utf = (struct UTrapframe *) (UXSTACKTOP - sizeof(struct UTrapframe));

    user_mem_assert(curenv, utf, sizeof(struct UTrapframe), PTE_U | PTE_W | PTE_P);

    utf->utf_esp = tf->tf_esp; //复制esp
    utf->utf_eflags = tf->tf_eflags; //复制eflags
    utf->utf_eip = tf->tf_eip; //复制eip
    utf->utf_regs = tf->tf_regs; //复制寄存器
    utf->utf_err = tf->tf_err; //复制err
    utf->utf_fault_va = fault_va;
    // 设置在返回到user态的时候, 执行upcall函数。
    tf->tf_esp = (uintptr_t)utf;
    tf->tf_eip = (uintptr_t)curenv->env_pgfault_upcall;

    env_run(curenv); //返回用户态执行
}
```

### 2.1.3. 用户态缺页错误处理入口

现在需要用汇编语言调用 C 语言的缺页处理函数并且在恢复执行原始的出错指令。

这段汇编代码将要用函数 `sys_env_set_pgfault_upcall()` 注册到内核。

最后, 需要实现用户级的缺页处理机制。

## 补充作业

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.

### 理论准备:

user 通过调用 lib 的 `set_pgfault_handler` 注册 upcall 函数。user 的 upcall 函数是 `_pgfault_handler`, lib 实际注册的 upcall 函数是 `_pgfault_upcall`。 `_pgfault_upcall` 的作用是调用 `_pgfault_handler`, 在 `_pgfault_handler` 执行结束之后, 返回到 UTrapframe 中的 `trap-time` 的状态。对于 `trap-time-eip` 的位置:

如果 trap 之前, 是执行在 normal stack 上, 那么 `trap-time-eip` 位置在 `trap-time-esp-4B` 上。

果 trap 之前, 是执行在 exception stack 上, 那么 `trap-time-eip` 位置在 Utrapframe 之间留出的 4B 上。这样, 在恢复 `trap-time-esp` 的时候, 第一个弹出的一定是 `trap-time-eip`。

`pgfaultupcall` 是所有用户页错误处理程序的入口, 由这里调用用户自定义的处理程序, 并在处理完成后, 从错误栈中保存的 UTrapframe 中恢复相应信息, 然后跳回到发生错误之前的指令, 恢复原来的进程运行。

`_pgfault_handler` 结束后的堆栈:



```

// +-----USTACKTOP-----+   high
// |           ...           |
// +-----+
// |           |
// +-----+
// | trap-time-esp   (4B) |
// +-----+
// | trap-time-eflags (4B) |
// +-----+
// | trap-time-eip   (4B) |
// +-----+   low
// | trap-time-regs  (32B)|
// | ...            |
// | ...            |
// +-----+
// | err             (4B) |
// +-----+
// | fault_va        (4B) |
// +-----+   <-- cur_esp
//           (1)
//
// +----trap-time-stack-----+
// |           ...           |
// +-----+
// | trap-time-eip   (4B) |
// +-----+   <-- trap_time_esp
//
//           (2)

```

在 lib/pfentry.S 中添加如下代码：

```

// LAB 4: Your code here.
movl 0x30(%esp), %eax
subl $0x4, %eax
movl %eax, 0x30(%esp)
//将原出错程序的EIP (即trap-timeeip)放入留出的4字节空白区域,以便后来恢复运行
movl 0x28(%esp), %ebx
movl %ebx, (%eax)
// Restore the trap-time registers. After you do this, you
// can no longer modify any general-purpose registers.
// LAB 4: Your code here.
//恢复所有通用寄存器。从这句话完成以后开始所有的通用寄存器就不能再使用了
addl $0x8, %esp
popal
// Restore eflags from the stack. After you do this, you can
// no longer use arithmetic operations or anything else that
// modifies eflags.
// LAB 4: Your code here.
//恢复EFLAGS标志寄存器,从这句话以后就不能使用会修改EFLAGS操
//作的指令了,比如算数指令add,sub或者mov和int等等
addl $0x4, %esp
popfl
// Switch back to the adjusted trap-time stack.
// LAB 4: Your code here.
//切换回原来出错的程序运行栈
popl %esp
// Return to re-execute the instruction that faulted.
// LAB 4: Your code here.
//返回出错程序
ret

```

## 作业 9

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

实现用户级的缺页处理机制 `set_pgfault_handler`

在 lib/pgfault.c 中添加代码:

```
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");
        // 开辟一块空间作为异常栈
        r = sys_page_alloc(thisenv->env_id, (void*)(UXSTACKTOP-PGSIZE), PTE_U | PTE_P | PTE_W);
        if(r < 0){
            panic("set_pgfault_handler : sys_page_alloc failed. %e.\n", r);
        }

        // 设置缺页处理函数
        r = sys_env_set_pgfault_upcall(thisenv->env_id, (void*)_pgfault_upcall);
        if(r < 0){
            panic("set_pgfault_handler : sys_env_set_pgfault_upcall failed. %e.\n", r);
        }
    }

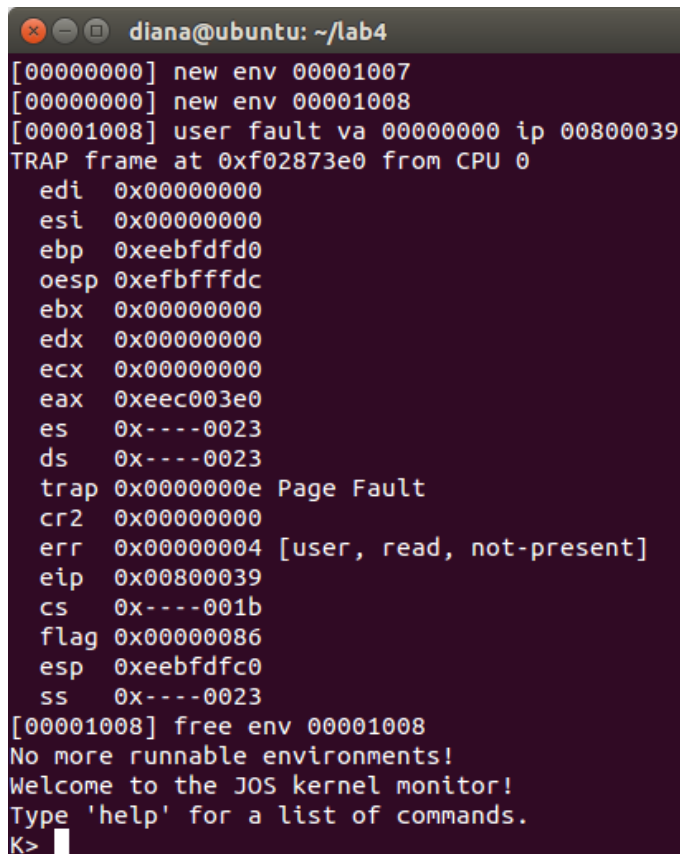
    // Save handler pointer for assembly to call.
    _pgfault_handler = handler;
}
```

检测结果:

(1)在 init.c 中设置环境:

```
ENV_CREATE(user_faultread, ENV_TYPE_USER);
```

检测:



```
diana@ubuntu: ~/lab4
[00000000] new env 00001007
[00000000] new env 00001008
[00001008] user fault va 00000000 ip 00800039
TRAP frame at 0xf02873e0 from CPU 0
edi 0x00000000
esi 0x00000000
ebp 0xeebdfd0
oesp 0xefbfffdc
ebx 0x00000000
edx 0x00000000
ecx 0x00000000
eax 0xeec003e0
es 0x---0023
ds 0x---0023
trap 0x0000000e Page Fault
cr2 0x00000000
err 0x00000004 [user, read, not-present]
eip 0x00800039
cs 0x---001b
flag 0x00000086
esp 0xeebdfdc0
ss 0x---0023
[00001008] free env 00001008
No more runnable environments!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

(2)在 init.c 中设置环境:

```
ENV_CREATE(user_faultdie|, ENV_TYPE_USER);
```

检测:

```
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2
[00000000] new env 00001000
[00000000] new env 00001001
[00000000] new env 00001002
[00000000] new env 00001003
[00000000] new env 00001004
[00000000] new env 00001005
[00000000] new env 00001006
[00000000] new env 00001007
[00000000] new env 00001008
i faulted at va deadbeef, err 6
[00001008] exiting gracefully
[00001008] free env 00001008
No more runnable environments!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
```

(3)在 init.c 中设置环境:

```
ENV_CREATE(user_faultalloc, ENV_TYPE_USER);
```

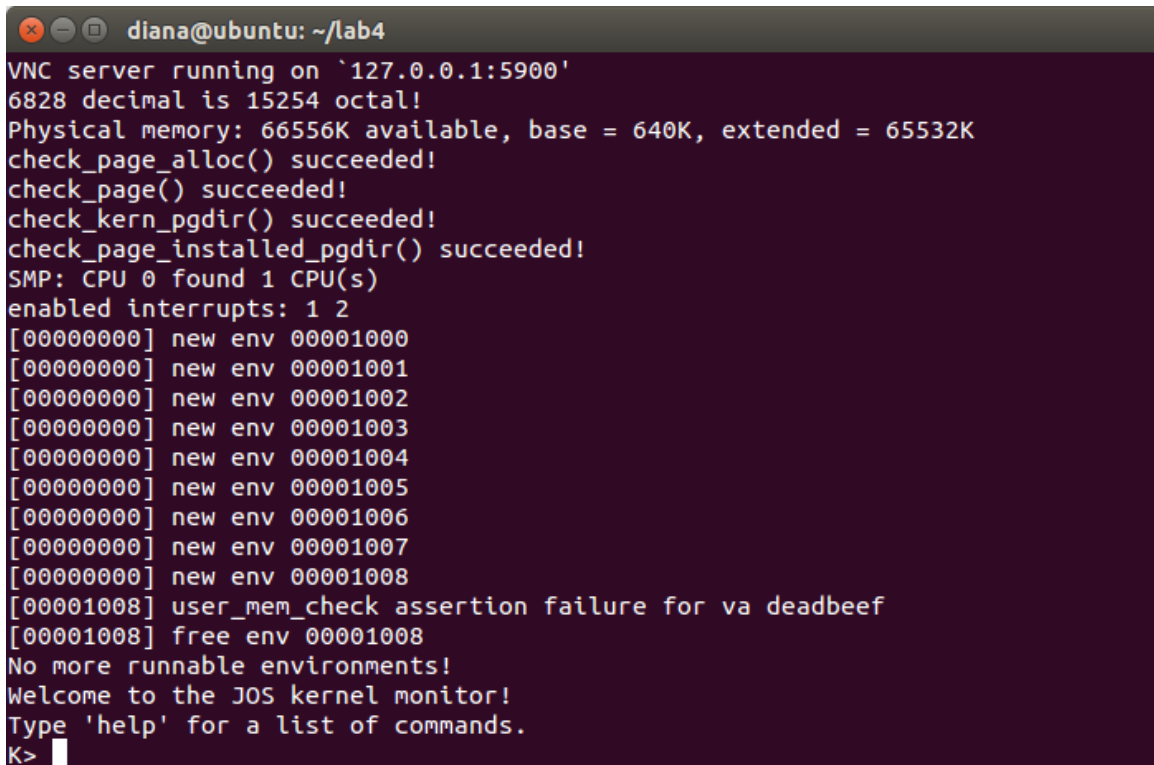
检测:

```
diana@ubuntu: ~/lab4
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2
[00000000] new env 00001000
[00000000] new env 00001001
[00000000] new env 00001002
[00000000] new env 00001003
[00000000] new env 00001004
[00000000] new env 00001005
[00000000] new env 00001006
[00000000] new env 00001007
[00000000] new env 00001008
fault deadbeef
this string was faulted in at deadbeef
fault cafebffe
fault cafec000
this string was faulted in at cafebffe
[00001008] exiting gracefully
[00001008] free env 00001008
No more runnable environments!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> 
```

(4)在 `init.c` 中设置环境:

```
ENV_CREATE(user_faultallocbad, ENV_TYPE_USER);
```

检测:

A terminal window titled 'diana@ubuntu: ~/lab4' showing the output of a VNC server. The text includes system information like 'VNC server running on 127.0.0.1:5900', memory details, and a series of 'new env' and 'free env' messages. It ends with an assertion failure 'user\_mem\_check assertion failure for va deadbeef' and a 'K>' prompt.

```
diana@ubuntu: ~/lab4
VNC server running on `127.0.0.1:5900'
6828 decimal is 15254 octal!
Physical memory: 66556K available, base = 640K, extended = 65532K
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2
[00000000] new env 00001000
[00000000] new env 00001001
[00000000] new env 00001002
[00000000] new env 00001003
[00000000] new env 00001004
[00000000] new env 00001005
[00000000] new env 00001006
[00000000] new env 00001007
[00000000] new env 00001008
[00001008] user_mem_check assertion failure for va deadbeef
[00001008] free env 00001008
No more runnable environments!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

## 2.2. 实现写时复制的 Fork

现在已经拥有足够的内核调用支持来在用户空间实现整个带有写时复制的 `fork()` 函数了。发布的代码中已经在 `lib/fork.c` 中为 `fork()` 函数提供了一个框架。类似于 `dumbfork()` 函数, `fork()` 创建一个新进程, 然后扫描父进程的整个地址空间并在子进程中设置对应的映射关系。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。函数 `fork()` 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。

## 作业 10

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.

```
1008: I am ''
1009: I am '0'
2008: I am '00'
2009: I am '000'
100a: I am '1'
3008: I am '11'
3009: I am '10'
200a: I am '110'
4008: I am '100'
100b: I am '01'
5008: I am '011'
4009: I am '010'
100c: I am '001'
100d: I am '111'
100e: I am '101'
```

### (1) duppage()

函数作用:实现写时复制方式的页复制

设计思路:

在 lib/entry.S 中定义了如下几个全局变量:

```
.data
// Define the global symbols 'envs', 'pages', 'vpt', and 'vpd'
// so that they can be used in C as if they were ordinary global arrays.
    .globl envs
    .set envs, UENVS
    .globl pages
    .set pages, UPAGES
    .globl vpt
    .set vpt, UVPT
    .globl vpd
    .set vpd, (UVPT+(UVPT>>12)*4)
```

可以看到其中定义的两个变量:

vpt: 指向 UVPT

vpd:  $(UVPT + (UVPT \gg 12) * 4)$ , 实际上就是把 UVPT 的前 10 位复制到

了第 11 到 20 位，正好对应上面提到页目录表项地址的构造方法。

在 inc/memlayout.h 中是这么描述这两个变量的：

```
typedef uint32_t pte_t;
typedef uint32_t pde_t;

#ifdef JOS_USER
/*
 * The page directory entry corresponding to the virtual address range
 * [VPT, VPT + PTSIZE) points to the page directory itself. Thus, the page
 * directory is treated as a page table as well as a page directory.
 *
 * One result of treating the page directory as a page table is that all PTEs
 * can be accessed through a "virtual page table" at virtual address VPT (to
 * which vpt is set in entry.S). The PTE for page number N is stored in
 * vpt[N]. (It's worth drawing a diagram of this!)
 *
 * A second consequence is that the contents of the current page directory
 * will always be available at virtual address (VPT + (VPT >> PGSHIFT)), to
 * which vpd is set in entry.S.
 */
extern volatile pte_t vpt[];      // VA of "virtual page table"
extern volatile pde_t vpd[];      // VA of current page directory
#endif
```

里面提到，假设需要查询的虚拟地址为 va，则其对应的页目录表项为：

vpd[VPD(va)]，其中 VPD 等价于 PDX 宏。其对应的页表表项为：

vpt[VPN(va)]，其中 VPN 等价于 PPN 宏。

由页目录和页表之间的关系可以知道：

(1) PGNUM(va) 为 N 的页表项就存放在 vpt[N]。

(2) vpd[PDX(va)] 就是 va 对应的页目录项的内容。

注意传入的参数是物理页号，所以要作相应的地址运算。

在 lib/fork.c 中添加代码：

```

static int
duppage(envid_t envid, unsigned pn)
{
    int r;

    // LAB 4: Your code here.
    //panic("duppage not implemented");
    void *va = (void *) (pn << PGSHIFT);

    if (!(vpd[PDX(pn << PGSHIFT)] & PTE_P ))// 检验page dir PTE_P 是否被设置
        panic("duppage : page dir PTE_P is not set.\n");

    if (!(vpt[pn] & ( PTE_W | PTE_COW )))//权限
        panic("duppage : page is not PTE_W or PTE_COW.\n");

    //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,
    r = sys_page_map(0, va, envid, va, PTE_U | PTE_COW | PTE_P);
    if (r < 0)
        panic("duppage : sys_page_map error : %e.\n", r);
    //重新将自身地址空间中的该页映射为写时复制属性
    r = sys_page_map(0, va, 0, va, PTE_U | PTE_COW | PTE_P);
    if (r < 0)
        panic("duppage : sys_page_map error : %e.\n", r);

    return 0;
}

```

函数作用:缺页处理

设计思路:

- 1) 内核将缺页错误传递给\_pgfault\_upcall,后者调用 fork()的 pgfault()函数。
  - 2) 函数 pgfault()检查这个错误是写操作(FEC\_WR)并且缺页的 PTE 是含有 PTE\_COW 属性的。如果不是,调用 panic 给出警告。
  - 3) 函数 pgfault()在临时空间分配一个页,然后将整个缺页的内容拷贝到这个页上。然后,缺页处理函数将这个页映射到适当的地方,并赋予读/写权限
- 代码实现:



```

// 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 vpt
//   (see <inc/memlayout.h>).

// LAB 4: Your code here.

//检查这个错误是写操作(FEC_WR)并且缺页的 PTE 是含有 PTE_COW 属性的。如果不是,调用 panic 给出警告。
if ( !(vpd[PDX(addr)] & PTE_P ) )
    panic("pgfault : page dir PTE_P not set.\n");
if (((err & FEC_WR) != FEC_WR) || !(vpt[PGNUM(addr)] & PTE_COW) )
    panic("pgfault : pagefault %08x not FEC_WR or PTE_COW.\n",err);


// 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.
//   No need to explicitly delete the old page's mapping.

// LAB 4: Your code here.
//panic("pgfault not implemented");

r = sys_page_alloc(0, PFTEMP, PTE_U | PTE_P | PTE_W); //在临时空间分配一个页
if (r < 0)
    panic("pgfault : sys_page_alloc error : %e.\n",r);

addr = ROUNDDOWN(addr, PGSIZE); //对齐
memmove(PFTEMP, addr, PGSIZE); //将整个缺页的内容拷贝到临时分配的页上

// 将这个页映射到适当的地方,并赋予读/写权限
r = sys_page_map(0, PFTEMP, 0, addr, PTE_U | PTE_P | PTE_W);
if (r < 0)
    panic("pgfault : sys_page_map error : %e.\n",r);

r = sys_page_unmap(0, PFTEMP); //解除映射
if (r < 0)
    panic("pgfault : sys_page_unmap error : %e.\n",r);

```

### (3)fork()

函数作用: 当一个进程需要对某个页执行写操作时对这个页进行复制

设计思路: 函数 fork() 基本的控制流程如下:

- 1) 父进程使用前面实现的函数 set\_pgfault\_handler() 设置 pgfault() 为缺页错误处理函数。

2) 父进程调用 `sys_exofork()`, 创建一个子进程。

3) 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 都设置为不可写, 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, 不能使用上述方式进行映射。应该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。

4) 父进程为子进程设置与其相同的用户缺页处理函数入口。

5) 子进程现在可以运行了, 因此父进程将其设置可运行的。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。

代码实现:

**//创建子环境**

```
envid_t envid;  
uintptr_t va;  
int r;
```

```
set_pgfault_handler(pgfault); //缺页处理
```

```
envid = sys_exofork(); //创建子进程
```

```
if (envid < 0)  
    panic("fork : sys_exofork error, %e.\n", envid);
```

```
if (envid == 0)  
{
```

```
    thisenv = &envs[ENVX(sys_getenvid())]; //设置当前进程  
    return 0;
```

```
}
```

```

//初始化用户空间, 忽略无 PTE_W or PTE_COW权限的页(出于保护的目无PTE_W or PTE_COW权限)
for (va = UTEXT ; va < USTACKTOP; va += PGSIZE)
    if ((vpd[PDX(va)] & PTE_P) && (vpt[PGNUM(va)] & PTE_P) &&
        (vpt[PGNUM(va)] & PTE_U) && (vpt[PGNUM(va)] & (PTE_W | PTE_COW)))
        duppage(env_id, PGNUM(va));

// 初始化异常栈, 父进程的异常栈不能被再次映射, 因为在这个时候它的缺页中断正在使用它, 它应该是可写的
r = sys_page_alloc(env_id, (void*)(UXSTACKTOP-PGSIZE), PTE_U | PTE_P | PTE_W);
if (r < 0)
    panic("[%08x] fork : sys_page_alloc error : %e.\n", thisenv->env_id, r);

// 设置子环境缺页中断处理函数
extern void _pgfault_upcall();
r = sys_env_set_pgfault_upcall(env_id, (void*)_pgfault_upcall);
if (r < 0)
    panic("[%08x] fork : sys_env_set_pgfault_upcall error : %e.\n",
        thisenv->env_id, r);

// 设置子环境状态为 RUNNABLE
r = sys_env_set_status(env_id, ENV_RUNNABLE);
if (r < 0)
    panic("[%08x] fork : sys_env_set_status error : %e", thisenv->env_id, r);

return env_id;

```

在 init.c 中设置环境:

```
ENV_CREATE(user_forktree, ENV_TYPE_USER);
```

检测:

```
diana@ubuntu: ~/lab4
1008: I am ''
[00001008] new env 00001009
[00001008] new env 0000100a
[00001008] exiting gracefully
[00001008] free env 00001008
1009: I am '0'
[00001009] new env 00002008
[00001009] new env 0000100b
[00001009] exiting gracefully
[00001009] free env 00001009
2008: I am '00'
[00002008] new env 00002009
[00002008] new env 0000100c
[00002008] exiting gracefully
[00002008] free env 00002008
2009: I am '000'
[00002009] exiting gracefully
[00002009] free env 00002009
100a: I am '1'
[0000100a] new env 00003009
[0000100a] new env 00003008
[0000100a] exiting gracefully
[0000100a] free env 0000100a
3008: I am '11'
[00003008] new env 0000200a
[00003008] new env 0000100d
[00003008] exiting gracefully
[00003008] free env 00003008
3009: I am '10'
[00003009] new env 00004008
[00003009] new env 0000100e
[00003009] exiting gracefully
[00003009] free env 00003009
4008: I am '100'
[00004008] exiting gracefully
[00004008] free env 00004008
```

make grade

```
+ mk obj/kern/kernel.img
make[1]: Leaving directory '/home/diana/lab4'
sh ./grade-lab4.sh
gmake[1]: Entering directory '/home/diana/lab4'
gmake[1]: Nothing to be done for 'all'.
gmake[1]: Leaving directory '/home/diana/lab4'
dumbfork: OK (1.6s)
Part A score: 5/5

faultread: OK (1.6s)
faultwrite: OK (1.6s)
faultdie: OK (1.5s)
faultregs: OK (1.6s)
faultalloc: OK (1.6s)
faultallocbad: OK (1.6s)
faultnostack: OK (1.5s)
faultbadhandler: OK (1.5s)
faultevilhandler: OK (1.6s)
forktree: OK (2.0s)
Part B score: 50/50
```

## 三 抢占式多任务和进程间通信

在本节中, 将修改内核以实现从互不协作的进程间进行调度, 并使得进程可以相互传递消息。

### 3.1. 时钟中断和抢占

修改文件 `kern/init.c` 以运行 `user/spin` 程序。这个程序创建一个子进程, 这个子进程在接到 CPU 控制权之后只是进行无限的循环。父进程或者内核都不会再次获得 CPU。这对于保护内核不受错误的或是恶意的用户程序的危害是很不理想的, 因为任意用户程序都可以通过不停地进行循环并不放弃 CPU 的控制权而使得系统失去响应。为保证内核可以从一个正在运行的进程中夺取 CPU 控制权, 就必须修改 JOS 内核以支持外部的时钟中断。

#### 3.1.1. 中断原理

外部中断(比如硬件中断)被称作 IRQ。一共有 16 种 IRQ, 编号从 0 到 15。从 IRQ 号到 IDT 的映射关系不是固定的。文件 `picirq.c` 中的 `pic_init()` 函数将 IRQ 0-15 映射到 IDT 的 `IRQ_OFFSET` `IRQ_OFFSET+15`。

在文件 `kern/picirq.h` 中, `IRQ_OFFSET` 被定义为 10 进制的 32。这

样, IDT 的 32 到 47 就对应着 IRQ 的 0 到 15。例如, 时钟中断是 IRQ 0。因此 IDT[IRQ\_OFFSET+0] (如 IDT[32]) 包含了内核中时钟中断函数的地址。这个 IRQ\_OFFSET 的选择使得设备中断不会和处理器的异常发生冲突(实际上, 在早期运行 MS-DOS 的 PC 上, IRQ\_OFFSET 被设置为 0, 这确实导致了处理硬件中断和处理处理器异常的一些冲突)。

在 JOS 中, 我们的实现相比于 Unix xv6 较为简单。在内核时外部设备中断是关闭的, 在用户模式下时才响应。外部设备中断被 %eflags 寄存器的 FL\_IF 标志位所控制 (inc/mmu.h)。当这位为 1 时, 外部中断是开启的。这一位可以通过多种方式进行修改, 但根据我们的简化实现, 我们将只在进出用户模式时通过保存和回写 %eflags 寄存器进行修改。

在进程中我们需要保证 FL\_IF 被置位, 这样在这个进程运行时出现中断, 就可以到达处理器并被相应的中断处理代码所处理。否则, 中断就会被屏蔽或者忽略直到外部中断被开启。在处理器重新启动之后, 中断是默认屏蔽的, 并且到目前为止我们还没有开启它。

## 作业 11

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.

设计思路：

这个部分模仿原来设置默认中断向量即可，文件 picirq.c 中的 pic\_init()函数将 IRQ 0-15 映射到 IDT 的 IRQ\_OFFSET 到 IRQ\_OFFSET+15。在文件 kern/picirq.h 中,IRQ\_OFFSET 被定义为 10 进制的 32。这样,IDT 的 32 到 47 就对应着 IRQ 的 0 到 15。例如,时钟中断是 IRQ 0。因此 IDT[IRQ\_OFFSET+0](如 IDT[32])包含了内核中时钟中断函数的地址。

代码实现：

先在 kern/trapentry.S 中定义相关的处理例程：

```
TRAPHANDLER_NOEC(irq_timer, IRQ_OFFSET+IRQ_TIMER)
TRAPHANDLER_NOEC(irq_kbd, IRQ_OFFSET+IRQ_KBD)
TRAPHANDLER_NOEC(irq_serial, IRQ_OFFSET+IRQ_SERIAL)
TRAPHANDLER_NOEC(irq_spurious, IRQ_OFFSET+IRQ_SPURIOUS)
TRAPHANDLER_NOEC(irq_ide, IRQ_OFFSET+IRQ_IDE)
TRAPHANDLER_NOEC(irq_error, IRQ_OFFSET+IRQ_ERROR)
```

然

后在 IDT 里注册，在 kern/trap.c 修改 trap\_init(),

```
extern void irq_timer();
extern void irq_kbd();
extern void irq_serial();
extern void irq_spurious();
extern void irq_ide();
extern void irq_error();
```

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

创建用户进程时，只需要在其 EFLAGS 寄存器中打开 IF 位即可，这样等到运行时就会自动开启外部中断响应了，在 env\_alloc()( kern/env.c )设置 tf\_eflags：



```
// Enable interrupts while in user mode.
// LAB 4: Your code here.

e->env_tf.tf_eflags=e->env_tf.tf_eflags | FL_IF;
```

检测结果：

在 init.c 中设置环境：

```
ENV_CREATE(user_spin, ENV_TYPE_USER);
```

检测

```
diana@ubuntu: ~/lab4
[00000000] new env 00001007
[00000000] new env 00001008
I am the parent. Forking the child...
[00001008] new env 00001009
TRAP frame at 0xf028a3e0 from CPU 0
  edi 0x00001009
  esi 0x00802000
  ebp 0xeebdfdb0
  oesp 0xefbfffdc
  ebx 0x1f501000
  edx 0x00000000
  ecx 0x00802000
  eax 0x00000000
  es 0x----0023
  ds 0x----0023
  trap 0x00000020 Hardware Interrupt
  err 0x00000000
  eip 0x0080103f
  cs 0x----001b
  flag 0x00000246
  esp 0xeebdfdf8
  ss 0x----0023
[00001008] free env 00001008
No more runnable environments!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

### 3.1.2. 处理时钟中断

在程序 user/spin 中, 当子进程第一次运行后, 它就不停地在循环, 内核再也不能得到控制权。

现在, 需要让硬件定期的触发时钟中断, 以强制将控制权转移到内核, 这样就可以切换到其他的进程来运行。

函数 pic\_init() 和 kclock\_init() (在 init.c 中的 i386\_init) 用来设置时钟和中断控制器来生成中断。现在, 需要编写代码来处理这些中断。



## 作业 12

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.

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

再次检测 user/spin, 这时运行 spin 看到子进程交出权限然后被父进程销毁。

```
diana@ubuntu: ~/lab4
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2
[00000000] new env 00001000
[00000000] new env 00001001
[00000000] new env 00001002
[00000000] new env 00001003
[00000000] new env 00001004
[00000000] new env 00001005
[00000000] new env 00001006
[00000000] new env 00001007
[00000000] new env 00001008
I am the parent. Forking the child...
[00001008] new env 00001009
I am the parent. Running the child...
I am the child. Spinning...
I am the parent. Killing the child...
[00001008] destroying 00001009
[00001008] free env 00001009
[00001008] exiting gracefully
[00001008] free env 00001008
No more runnable environments!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> 
```

回归测试:

make qemu CPUS=2

```
diana@ubuntu: ~/lab4
[00000000] new env 00001002
[00000000] new env 00001003
[00000000] new env 00001004
[00000000] new env 00001005
[00000000] new env 00001006
[00000000] new env 00001007
[00000000] new env 00001008
I am the parent. Forking the child...
[00001008] new env 00001009
I am the parent. Running the child...
I am the child. Spinning...
I am the parent. Killing the child...
[00001008] destroying 00001009
[00001008] free env 00001009
[00001008] exiting gracefully
[00001008] free env 00001008
No more runnable environments!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> kernel panic on CPU 1 at kern/env.c:572: PADDR called with invalid kva 00000000
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> 
```

make grade

```
diana@ubuntu: ~/lab4
faultbadhandler: OK (1.7s)
faultevilhandler: OK (1.7s)
forktree: OK (2.0s)
Part B score: 50/50

spin: OK (1.8s)
stresssched: OK (3.6s)
pingpong: missing '1009 got 0 from 1008'
missing '1008 got 1 from 1009'
missing '1009 got 8 from 1008'
missing '1008 got 9 from 1009'
missing '1009 got 10 from 1008'
missing '.00001008. exiting gracefully'
missing '.00001008. free env 00001008'
missing '.00001009. exiting gracefully'
missing '.00001009. free env 00001009'
WRONG (1.8s)
primes: missing 'CPU .: 2 .00001009. new env 0000100a'
missing 'CPU .: 3 .0000100a. new env 0000100b'
missing 'CPU .: 5 .0000100b. new env 0000100c'
missing 'CPU .: 7 .0000100c. new env 0000100d'
missing 'CPU .: 11 .0000100d. new env 0000100e'
missing 'CPU .: 1877 .00001128. new env 00001129'
WRONG (1.9s)
Part C score: 10/20

Score: 65/75
GNUmakefile:189: recipe for target 'grade' failed
make: *** [grade] Error 1
diana@ubuntu:~/lab4$
```

总得分 65, 说明至此是成功的!

## 3.2. 进程间通信

准确的说, 在 JOS 中这应该叫做"inter-environment communication"或"IEC", 但是其他人都叫做 IPC, 所以, 我们将使用 IPC 来代指。

在前面的试验中, 我们主要着眼于操作系统的独立面, 即给每个进程一个认为自己占有整个计算机的错觉。操作系统的另一个重要服务就是允许程序在需要时相互交流。Unix 的管道模式就是个经典的例子。

### 3.2.1. JOS 中的 IPC

我们需要实现一些额外的 JOS 内核的系统调用来提供一个简单的 IPC 机制。首先, 将实现两个系统调用 `sys_ipc_recv` 和 `sys_ipc_can_send`。之后再实现两个库函数 `ipc_recv` 和 `ipc_send`。

进程使用 JOS 的 IPC 机制所发送的消息包括两部分:一个 32 位的值以及可选的一个页的映射关系。在消息中允许进程传递页映射可以提供一个有效的方式来交换更多的数据,并使得进程之间可以很方便的设置共享内存。

### 3.2.2. 发送和接受消息

要接收一个消息,进程必须调用 `sys_ipc_recv`。这个系统调用挂起当前进程并在收到消息之前停止运行。当一个进程在等待接受消息时,任何其他的进程都可以发送消息给它,而不是只有父子进程可以。也就是说,在前面实现的权限检查在 IPC 时将不会应用到 IPC 中,因为 IPC 函数被认为是安全的:一个进程不可能通过发送消息就使得其他进程出现故障(除非目标进程本身有问题)。

为了发送一个消息,进程需要调用 `sys_ipc_try_send`,设置要接受的进程的 ID 和要发送的消息。如果目标进程正在接收数据(调用 `sys_ipc_recv` 并且还没有收到数据),则发送进程的函数就发送数据并返回 0。否则返回 `-E_IPC_NOT_RECV` 以表明目标进程当前并没有准备接受数据。

用户空间的库函数 `ipc_recv` 将会调用 `sys_ipc_recv` 然后在当前进程的 `Env` 结构中获得接收的数据。

与此相类似,另一个库函数 `ipc_send` 将会反复的调用 `sys_ipc_can_send` 直到发送成功。

### 3.2.3. 页传递

当一个进程调用 `sys_ipc_recv` 并传递一个非 0 的 `dstva` 参数时(在 `UTOP` 之下),就表明它想获得一个页映射。如果发送方发送了一个页,那么,这个页应当被映射到接收方地址空间的 `dstva`。如果接收方已经在这个地址上有了映射的页,那么之前的页就会被取消映射。

当一个进程调用 `sys_ipc_try_send` 并传递一个非 0 的 `srcva` 参数时(在 `UTOP` 之下),这说明发送方想把当前映射到 `srcva` 的页发送给接收方,并设置权限为 `perm`。经过一个成功的 IPC 过程后,发送方的地址空间保存着对 `srcva` 所在页的映射关系,而接收方的地址空间上也对该物理内存页具有一个相同的映射。这样,这个页就被两个进程所共享。

如果接收方或发送方中的任何一方没有指明有页要进行传递时,就不进行这个操作。IPC 结束之后,内核在接收方 `Env` 结构的 `env_ipc_perm` 设置该页的权限,

如果没有进行页传递, 就设置为 0。

## 作业 13

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.

代码阅读：

`inc/env.h` `struct Env` 中增加了 5 个成员：

```
// 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
```

`env_ipc_recving`：

当进程使用 `sysipcrecv()` 等待信息时，会将这个成员置为 1，然后阻塞等待；当一个进程向它发消息解除阻塞后，发送进程将此成员修改为 0。

`env_ipc_dstva`：

如果进程要接受消息，并且是传送页，则该地址  $\leq$  `UTOP`。

`env_ipc_value`：

若等待消息的进程接受到了消息，发送方将接受方此成员置为消息值。

`env_ipc_from`：

发送方负责设置该成员为自己的 envid 号。

env\_ipc\_perm :

如果进程要接受消息，并且传送页，那么发送方发送页以后将传送的页权限传给这个成员。

具体实现：

### (1) sys\_ipc\_recv()

函数作用：挂起当前进程并在收到消息之前停止运行。

设计思路：当一个进程调用 sys\_ipc\_recv 并传递一个非 0 的 dstva 参数时(在 UTOP 之下)，就表明它想获得一个页映射。如果发送方发送了一个页，那么，这个页应当被映射到接收方地址空间的 dstva。如果接收方已经在这个地址上有了映射的页，那么之前的页就会被取消映射。

代码实现：

```
static int
sys_ipc_recv(void *dstva)//挂起当前进程并在收到消息之前停止运行
{
    // LAB 4: Your code here.
    //panic("sys_ipc_recv not implemented");
    //进程要接受消息，并且是传送页，则该地址≤UTOP
    if((uintptr_t)dstva < UTOP && (PGOFF(dstva)) != 0)//dstva无效
        return -E_INVAL;

    curenv->env_ipc_recving = 1; //使用sysipcrcv()等待信息,然后阻塞等待；
    //当一个进程向它发消息解除阻塞后，发送进程将此成员修改为0.
    curenv->env_ipc_dstva = dstva;//地址

    curenv->env_status = ENV_NOT_RUNNABLE;// 挂起

    return 0;
}
```

### (2) sys\_ipc\_try\_send()

函数作用：设置要接受的进程的 ID 和要发送的消息

设计思路：如果目标进程正在接收数据(调用 sys\_ipc\_recv 并且还没有收到数据)，则发送进程的函数就发送数据并返回 0。否则返回 -E\_IPC\_NOT\_RECV 以表明目标进程当前并没有准备接受数据。

代码实现：

```

static int
sys_ipc_try_send(envid_t envid, uint32_t value, void *srcva, unsigned perm)
{
    // LAB 4: Your code here.
    //panic("sys_ipc_try_send not implemented");
    struct Env *dstenv;
    pte_t *pte;
    struct Page *pp;

    if(envid2env(envid, &dstenv, 0))//无效envid
        return -E_BAD_ENV;

    if(!dstenv->env_ipc_recving)//目标进程当前并没有准备接受数据
        return -E_IPC_NOT_RECV;

    if((uintptr_t)srcva < UTOP)//地址<UTOP,进程要接受消息，并且是传送页
    {
        if(PGOFF(srcva) != 0)//不是页对齐
            return -E_INVAL;

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

        if(((perm | PTE_SYSCALL) != PTE_SYSCALL))//权限
            return -E_INVAL;

        pp = page_lookup(curenv->env_pgdir, srcva, &pte);//得到va虚拟地址的虚拟页
        if((uintptr_t)srcva < UTOP && !pp)//虚拟页不存在,即并未被映射
            return -E_INVAL;

        if((perm & PTE_W) && !(*pte & PTE_W))//只读
            return -E_INVAL;

        if(dstenv->env_ipc_dstva)
        {
            if(page_insert(dstenv->env_pgdir, pp, dstenv->env_ipc_dstva, perm) < 0)
                return -E_NO_MEM;//没有内存进行映射
            //如果进程要接受消息，并且传送页，那么发送方发送页以后将传送的页权限传给env_ipc_perm.
            dstenv->env_ipc_perm = perm;
        }
    }

    dstenv->env_ipc_recving = 0;    // 解除阻塞
    dstenv->env_ipc_value = value;//发送方将接受方此成员置为消息值
    dstenv->env_ipc_from = curenv->env_id;//设置为发送方自己的envid号
    dstenv->env_status = ENV_RUNNABLE;//就绪

    return 0;
}

```

在 `syscall()` 中, 添加相应的分发机制, 把这两个调用号加上:

```

case SYS_ipc_recv:
    ret = sys_ipc_recv((void *)a1);
    break;
case SYS_ipc_try_send:
    ret = sys_ipc_try_send(a1, a2, (void *)a3, a4);
    break;

```



### (3) ipc\_recv()

函数作用:调用 sys\_ipc\_recv 然后在当前进程的 Env 结构中获得接收的数据。

代码实现:

在 lib/ipc.c 中:

```
int32_t
ipc_recv(envid_t *from_env_store, void *pg, int *perm_store)//获得数据并返回
{
    // LAB 4: Your code here.
    //panic("ipc_recv not implemented");
    int res;

    if(pg)//非空
        res = sys_ipc_recv(pg);//接受page并映射到pg
    else
        res = sys_ipc_recv((void*)UTOP);//pg=null,传参无page

    if(res < 0)//系统调用失败
    {
        if(from_env_store)
            *from_env_store = 0;
        if(perm_store)
            *perm_store = 0;

        return res;
    }

    if(from_env_store)
        *from_env_store = thisenv->env_ipc_from;//发送端的envid

    if(perm_store)
        *perm_store = thisenv->env_ipc_perm;//发送端的page permission

    return thisenv->env_ipc_value;//返回接收的数据
}
```

函数作用:反复的调用 sys\_ipc\_try\_send 直到发送成功。

代码实现:



```

void
ipc_send(envid_t to_env, uint32_t val, void *pg, int perm)
{
    // LAB 4: Your code here.
    // panic("ipc_send not implemented");
    int res = -E_IPC_NOT_RECV;
    while(res < 0){//反复的调用 sys_ipc_try_send 直到发送成功。

        if(res != -E_IPC_NOT_RECV){
            panic("ipc_send : sys_ipc_try_send error : %e.\n", res);
        }

        if(pg){
            res = sys_ipc_try_send(to_env, val, pg, perm);
        }
        else{
            res = sys_ipc_try_send(to_env, val, (void *)UTOP, perm);
        }

        sys_yield();//找到一个可运行用户进程并运行
    }
}

```

检测结果：

make grade

```

diana@ubuntu: ~/lab4
+ mk obj/kern/kernel.img
make[1]: Leaving directory '/home/diana/lab4'
sh ./grade-lab4.sh
gmake[1]: Entering directory '/home/diana/lab4'
gmake[1]: Nothing to be done for 'all'.
gmake[1]: Leaving directory '/home/diana/lab4'
dumbfork: OK (1.6s)
Part A score: 5/5

faultread: OK (1.6s)
faultwrite: OK (1.6s)
faultdie: OK (1.5s)
faultregs: OK (1.6s)
faultalloc: OK (1.6s)
faultallocbad: OK (1.5s)
faultnostack: OK (1.5s)
faultbadhandler: OK (1.6s)
faultevilhandler: OK (1.6s)
forktree: OK (1.9s)
Part B score: 50/50

spin: OK (1.7s)
stresssched: OK (3.4s)
pingpong: OK (1.6s)
primes: OK (31.2s)
Part C score: 20/20

Score: 75/75
diana@ubuntu:~/lab4$

```