Lab4 设计描述

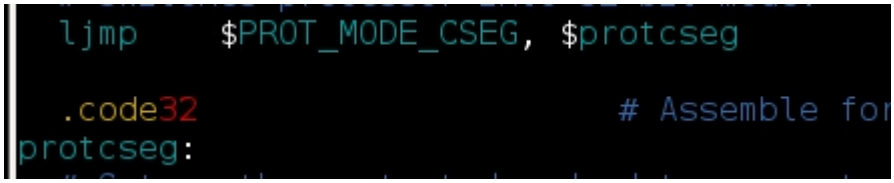
5130309029 沈匡济

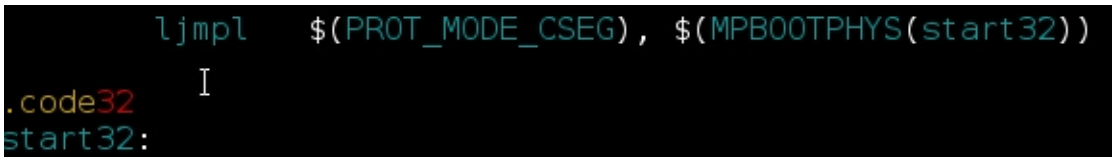
PartA

Exercise1

练习一的要求是，内核会把一段用于启动的代码加载到 MPENTRY_PADDR 处，因此要把这个页从空闲链表中去除，这里我直接在循环里判断如果到了该页就直接 continue，顺利通过 check_free_list 函数，在 check_kern_pgdir 处 panic 了，按照 lab 提示，先不管。

下面是问题一，仔细比较可以发现两个启动文件的区别在于：





详见文本文件。

Exercise2

这里的逻辑很简单，映射 NCPU 个 CPU 的内存到 KSTACKTOP 开始的虚拟地址，每项占 KSTKSIZE 的空间，两项间留出 KSTKGAP 的空档。

Exercise3

这里将原本的 ts 修改成当前 cpu 的 ts，然后做相应的替换就行了。运行提示在 env_run 函数中 panic 掉了因为这时 curenv 的 pgdir 还没有初始化，但是照理说这里根本还没有初始化 env 才对，这里我发现我 init.c 中 boot_ap 最后保留了 env_run 这个函数，很奇怪，应该是 merge lab3 的时候我手动将它保留了下来，注释之。

运行 make qemu CPUS=4，发现程序在提示 CPU 1 is starting 前卡住了！经过对代码的研究，发现这个问题的根源在于根本没有执行 mpentry.S 中的代码，这不得不让人联想到 Ex1 中的代码是不是写错了，或者又是合并代码的时候出错了，事实证明是后者，在 mem_init 函数里我发现映射 KERNBASE 处物理内存的操作被放在 mem_init_mp 后执行，这是不对的。

至此，Exercise3 结束。

Exercise4、4.1

按照 lab 中的说明在 4 处加上了锁。

这个 FIFO spinlock 很好理解，原本锁只用一个状态 locked 表示是否被占用，现在用两个状态 next 和 own，一个线程请求锁时，它获取 next 值当做自己的“ticket”并把 next 加一，直到 ticket 等于 own 值的时候才获取这把锁，释放锁时将 own 加一，这样就实现了 First-in-fist-out，这里也只要添加 4 处：初始化将 own 和 next 设为 0，判断是否持有锁时直接判断 own 和 next 是否相等，获取锁时进入循环等待 own 和 ticket 相等，释放锁时将 own

原子加一。运行 make qemu 看到 spinlock test succeed in cpu0。

Exercise5

这个 exercise 调试了我非常久，主要问题是 lab3 中的遗留问题，或是说应当被更新但是我觉得 lab 里没有讲清楚的问题。

首先是通过 sysenter 指令进入系统调用时需要在 syscall 函数上加锁，这里还比较好理解，其次就是在调用 sysenter 时需要保存当前 env 的状态，区别在于 lab3 中只使用了一个 env，而这里 sys_yield 会发生进程切换。为了减少代码修改量，我采用的方法是在栈上直接额外保存了一个 trapframe 格式的当前 env 信息，然后再压入指向这个结构的指针传递到 sysenter 的入口，直接更新 env 条目。对于 sched_yield 函数本身，实现的逻辑很简单，顺序扫描进程列表找到一个可执行的切换过去就行了。

Exercise6

进入 exercise6 首先要修正的一个问题是 syscall 需要传递 5 个参数，这里我观察到 a5 只用到来传一次 perm，所以直接将 a5 和 syscallno 拼在了一起，因为我觉得这两个参数在目前的应用范围看来只要 16 位就可以传了。

在我第一次测试的时候发现怎么都进入不了 sys_exofork，一直报 page fault，这里很令人纳闷，因为 sysenter 已经花很久去调试了。于是只好看汇编，设断点，终于发现，原来 dumbfork 里调用 exofork 是通过 int $30 来实现的，lab 有讲到但是自己在别的问题上纠结太久把这个给忘了，没想到这么快就遇到了。于是添加 T_SYSCALL 的 SETGATE 设置，以及路由里要相应的调用 syscall 函数，终于看到提示“exofork not implemented!”，可以开始真正写代码了……

这一系列函数的作用是给予用户态程序创建子进程的权限，包括复制当前进程信息、创建页表、映射页表等操作，本身的逻辑十分简单，代码量主要在一大堆的判断上，跟着注释写就可以了，比如当前 envid 是否合法、物理地址是否合法之类，这里在 exofork 函数里我选择了将父进程的状态完全拷贝给了子进程，在之后实现写时拷贝的时候应该会改。

至此，PartA 结束，初步建立起了一个多核系统。

PartB

Exercise7

写时拷贝技术会在复制父进程地址空间时将其保护起来，之后对其进行写操作时就会抛出一个 page fault，内核会在 user exception stack 上处理它，换言之，首先我们要实现一个栈切换的操作。ex7 本身很简单，设置一下用户态 page fault 的 upcall 函数。

Exercise8、9、10

实现调用用户态 page fault 处理函数的过程，如果发现没有注册 upcall 函数，就简单的停止这个用户进程并提示。然后在 user exception stack 上压入一个 UTrapframe 用来保存当前的状态，然后执行之前定义的 upcall 函数，这里初看之下有点不是很理解，就是这里说的递归错误该如何解决，仔细思考发现这不是很复杂的问题，只要判断当前栈指针是否已经在 exception stack 范围，如果在的话说明当前是一个递归错误处理，这里注释中有提到 _page_fault 中要为返回地址预留 1 字的空间。ex9 做的是紧接着如何恢复到 fault 处并继续执行，显然是要把之前压在栈上的 UTrapframe 取出然后调用 ret 即可，这里要写汇编让我觉得弹栈还是比较繁琐。Exercise10 中将函数在用户态中做了封装。

Exercise11

这里我们需要实现写时拷贝机制，在 fork 函数中首先调用 exofork 创建子进程，这里说一下我理解的创建过程：exofork 中父进程利用 env_alloc 创建一个新的进程，然后将父进程的内存空间映射到子进程上，为了在调用时区分两者，在 exofork 函数里将通过 env_alloc 出来的子进程 trapframe 的 eax 寄存器设为 0，而父进程的该值则为子进程的 id，在父进程被 env_destroy 时程序会自动做 sched_yield()切换至子进程继续工作。因此 fork 函数中最重要的是映射内存空间，即扫描父进程的内存，对存在且可写的部分做 duppage 操作，而写时拷贝就体现在这里，首先将页面全设置为“不可直接写”，在试图写的时候就会报 page fault，报错后转入用户态 page fault 处理函数，发现它其实是写时拷贝的，于是把页复制后重新映射，再返回用户态程序重新执行写操作。fork.c 一共有三个函数，pgfault 即为用户态 handler 函数，duppage 用作页映射，分工明确。至此可以通过 partB。

PartC

Exercise12

实现时钟中断，方法跟外部中断一样，详见 trapentry.S 和 trap_init 函数。

Exercise13

在 trap 分发机制 switch 语句中添加相应处理即可。

Exercise14

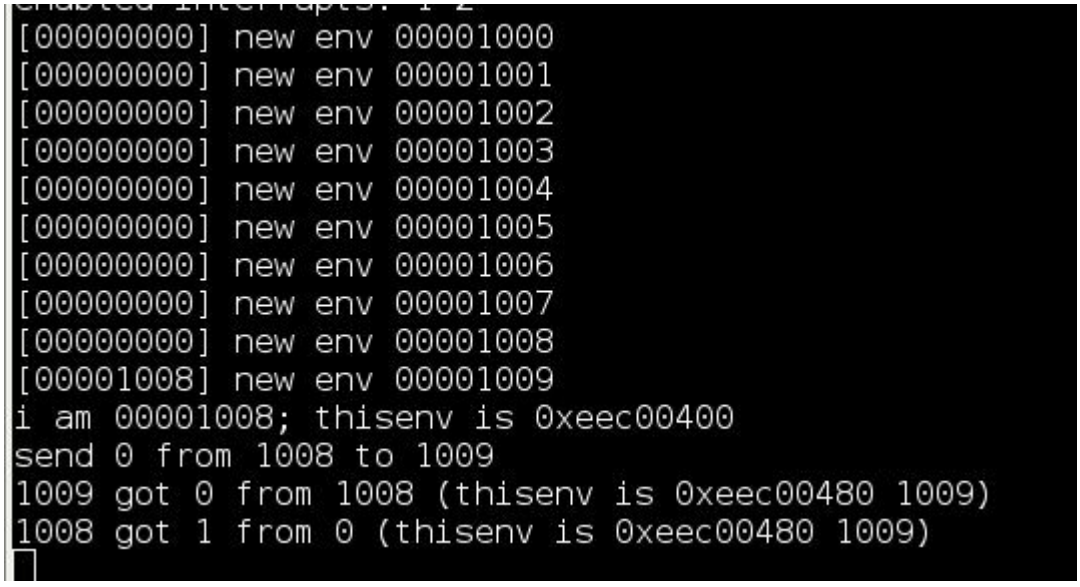
实现进程间的通信，分别在 syscall 中和 ipc.c 中实现 recv 和 send。主要的难点在于各种判断和封装，本身的逻辑很清晰。

Challenge

我选择了 sfork 这项 challenge。

这项 challenge 的要求很容易理解，让父进程和子进程共享内存空间，修改 duppage 使得并非所有可写的页都映射为 COW 页，这里的例外是进程的栈空间，在 sfork 中判断一下即可，如果是栈空间，那么进入 duppage 时强行修改为 COW 页，代码详见 lib/fork.c。

这是我运行 pingpongs 的结果：



可以看到，父子进程打印了不同的两条信息，却显示 thisenv 都等于 1009，这是因为 env

是全局变量，在父进程中修改了自然在子进程中也改变了。

至于这个程序为什么卡住了，我猜测正是受其影响。

为了验证我的 sfork 的正确性，我写了个测试版本的 pingpongs，可以在注释中看到。

```
}/*
uint32_t val=0;
void
umain(int argc, char **argv)
{
        envid_t id;
        id = sfork();
        if (id!=0)
        {
                cprintf("parent: %d\n",val);
                val = 1;
        }else
        {
                cprintf("child: %d\n",val);
        }
*/
```

经过本人测试，如果将 sfork 改为 fork，那么两次都会输出 0，改为 sfork 则是输出 0 和 1，足以说明实现了 lab 里的要求。