

# JOSLAB4 解题报告

Stone

2015 年 12 月 16 日

## Answers to Questions

Q1.在 mpentry.S Line46(我标记#stone 的位置), 可以看到其与 boot.S 不同, 使用了 MPBOOTPHYS macro。这个 macro 的主要作用是将 MPBOOT 的 linear address 转换成 physical address。Mpentry.S 需要做这个转换而 boot.S 不需要的的原因是: mpentry.S 的代码实际载入的物理地址应为 0x100000(KERNBASE)之上而在 boot\_aps()中将其复制在 0x7000(MPENTRY\_PADDR in inc/memlayout.h), 如果不用这个宏, 那么在启动时处于实模式, 无法访问到代码位置 0x7000 而启动失败。boot.S 的 linear address 就是其 physical address 所以不需要这个宏来进行转换。

Q2.如果多个 cpu share 同一个栈的话, 按照 lab 中 PAGEFLT 处理的逻辑, user 会先进入 user exception 栈之后再进入 kernel 栈调用 trap(), 那么当 user exception 栈是共享的, 即使有一把大锁也可能出现 race 导致 CPU 状态不一致问题。

Q3.因为对于不同的 user environment, 其页表的 KERNBASE 以上的部分相同, 所以在不同的 user environment 切换的过程中, 进入 kernel 态不需要修改 kernel virtual address pointer 就能访问任意一个页表。

# Challenge

我完成了 COW fork 部分的 `sfork()` 的 challenge。

根据对题目的分析,理解题目所要求的是修改页面映射,对于父进程,栈设置为 COW,其他不变。那么首先修改 `lib\fork.c` 中的映射函数,我添加了一个 `sduppage` 函数,与 `duppage` 函数不同的是添加了一个 `cow` 参数来决定是否需要设置为 `PTE|COW`,其他部分仿照 `duppage` 函数即可。对于 `sfork` 函数,主要添加了通过是否是栈位置的判断决定 `cow` 参数的值,核心代码如下:

```
+     int stack_cow = 1;
+     for (addr = USTACKTOP - PGSIZE; addr >= UTEXT; addr -= PGSIZE){
+         if (((vpd[VPD(addr)] & PTE_P) > 0) && ((vpt[VPN(addr)] & PTE_P) >
+             0) && ((vpt[VPN(addr)] & PTE_U) > 0))
+             sduppage(envid, VPN(addr), stack_cow);
+         else stack_cow = 0;
+     }
```

其他部分也仿照 `fork` 函数即可。

对于 challenge 的检测,我首先写了一个普通的函数 `\user\sforkcheck` 并登记,主要是在父进程修改一个参数值并在子进程打印出来来检测是否实现了题目要求,结果正确(可使用 `make run-sforkcheck` 来检测),之后按照题目要求修改了 `forktree` 为 `\user\sforktree` 并测试通过。

## Brief Design Description

Overall

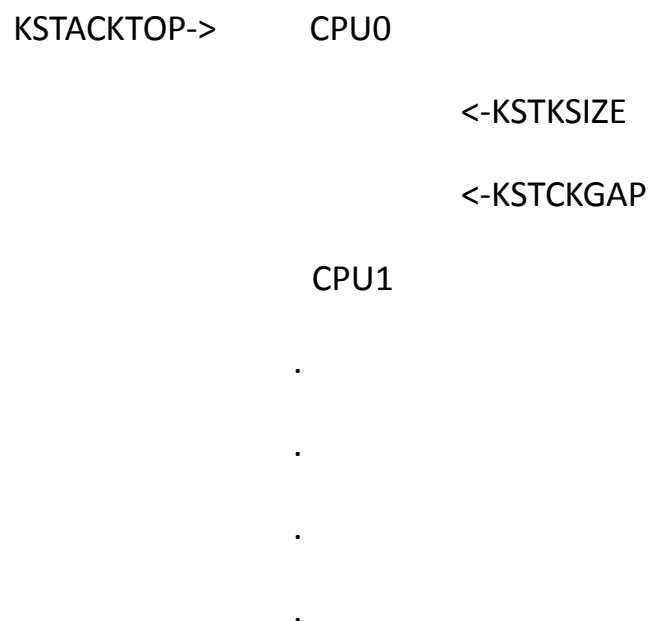
更新 `qemu` 版本并修改 `/conf/env.mk` 中的参数,添加 `QEMU='qemu-system-i386'` 以保证 lab 运行环境为新版本的 `qemu`。

## AP Bootstrap

修改/kern/pmap.c 中的 page\_init 函数 ,将 mp 位置的页除去空闲链表之外。

## Per-CPU State and Initialization

根据 inc/memlayout.h 中的结构图中 USTACKTOP 位置 ,画出结构图为 :



根据此结构可以完成 kern/pmap.c 中 mem\_init\_mp()函数关于每个 cpu 栈的映射。

之后将 kern/trap.c 中现有的只对一个 tss&tss descriptor 的初始化修改为多 cpu 的 ,对照 cpu 结构与 trap\_init\_percpu 的已有 ts 的代码仿照完成即可。

## Locking

一把大锁部分没什么好说的 , 按给的位置与拿锁放锁操作完成即可 ,

提示很详细。

Ticket spinlock 的完成：

Ticket spinlock is a FIFO spinlock that can avoid starving. The ticket spinlock has two fields, one is owner's ticket number, the other one is next ticket to be taken. When acquiring a lock, read the next ticket and increase it with an atomic instruction. Then wait until the owner's ticket equals to the read one. When releasing the lock, increase the owner's ticket number to give it to the next waiter.

这段提示中已经明确给出了 ticket spinlock 的逻辑：拿锁时候拿自己的票 a 把下移个票 b++，然后循环等待直到 a 等于 b，放锁时候把自己的票 a++。注意使用 `atomic_add_and_return` 完成原子的加和返回操作。注意测试的时候要添加 `TICKER_SPINLOCK` 宏。

## Round-Robin Scheduling

首先在 `kern/sched.c` 中完成 round-robin scheduler 的逻辑，这部分同样在网站上说的很清楚，看代码也比较容易，只贴代码不做赘述：

```
32 + /*stone's solution for lab4-A*/
33 + if (curenv != NULL){
34 +     uint32_t id = ENVX(curenv->env_id);
35 +     i = (id + 1) % NENV;
36 +     while (i != id){
37 +         if (envs[i].env_status == ENV_RUNNABLE && envs[i].env_type !=
38 +             ENV_TYPE_IDLE)
39 +             env_run(&envs[i]);
40 +         i = (i + 1) % NENV;
41 +     }
42 +     if (curenv->env_status == ENV_RUNNING)
43 +         env_run(curenv);
44 + }
```

之后再 `kern/syscall.c` 中添加 `sys_yield` 的路由指向包装过的 `sched_yield` 函数，并在 `/kern/init.c` 中的 `init_i386` 函数中按照要求添加 3 个(以上) `user_yield` 状态的环境测试即可。

## System Calls for Environment Creation

关于用户态的 syscall ,首先按照 lab3 的 idt 映射相关流程完成 syscall 指令的添加并将其权限位设置为 3(ring3, user mode) , 即修改 kern/trapentry.s 与 kern/trap.c。之后进入 kern/syscall.c 按照 hint 完成制定的函数即可。这里我遇到了一个比较奇怪的 bug , 对于 sys\_page\_map 函数 , 按照给定的参数传递方式 ( a1,a2,a3,a4,a5 ) 一直会挂 , 但是按照(@张弛)学长所给的将参数放置在数组中以指针形式传递就可以通过。然而我认为这两种方式并无区别 , 不能理解 , 特此注明。

## User level page fault handling

之前对于 user 态的 page fault , 处理方法是直接 destroy 这个 env , 现在要捕捉这个错误并传递到 kernel 态使用特殊的函数 (page\_fault\_handler)处理之。

这里在代码中有详细的 hint , 我不对代码实现过多赘述 , 而是着重解释一下栈结构和处理流程。

[KSTACKTOP, KSTACKTOP-KSTKSIZE)为 kernel 栈 , 称之为 A

[UXSTACKTOP, UXSTACKTOP-PGSIZE)为 User exception 栈 , 也就是用户处理 page fault 时进入的栈 , 称之为 B

[USTACKTOP, UTEXT)为 normal user 栈 , 也就是用户正常运行时的栈 , 称之为 C

对于用户在 C 栈也就是正常运行过程中 , 出现一个 page fault , 之后

会切入 kernel 态也就是 A 栈进行处理，调用 trap()函数路由进入 page\_fault\_handler 函数，进入 B 栈也就是进入用户态调用用户自己写好的特殊 handler 函数处理，之后返回 C 栈。

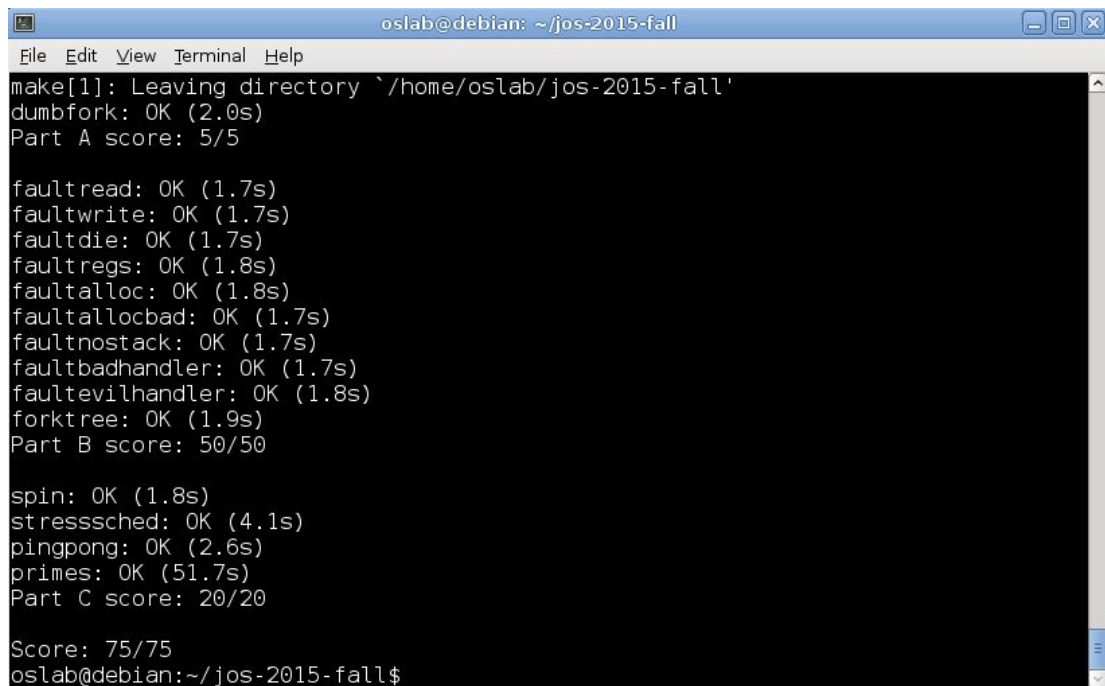
对于在 B 栈出现的 page fault，采用的方法是新建一个 B 栈处理新的 page fault，而不是将其和原本正在处理的 page fault 放到一起。

对于 lab 中的函数，和上面流程相关的对应代码为：  
set\_pgfault\_handler()->sys\_env\_set\_pgfault\_upcall()->page\_fault\_handler()->pgfault\_upcall->pfentry.S 中的汇编代码。

IPC

这部分代码中 HINT 非常非常非常详细，几乎是一步一步告诉怎么写，而且代码的结构也比较简单（为什么前面难的部分就那么多坑。。）  
我就不多说了。。

# Grade



```
oslab@debian: ~/jos-2015-fall
File Edit View Terminal Help
make[1]: Leaving directory `/home/oslab/jos-2015-fall'
dumbfork: OK (2.0s)
Part A score: 5/5

faultread: OK (1.7s)
faultwrite: OK (1.7s)
faultdie: OK (1.7s)
faultregs: OK (1.8s)
faultalloc: OK (1.8s)
faultallocbad: OK (1.7s)
faultnostack: OK (1.7s)
faultbadhandler: OK (1.7s)
faultevilhandler: OK (1.8s)
forktree: OK (1.9s)
Part B score: 50/50

spin: OK (1.8s)
stresssched: OK (4.1s)
pingpong: OK (2.6s)
primes: OK (51.7s)
Part C score: 20/20

Score: 75/75
oslab@debian:~/jos-2015-fall$
```

这是在没有 ticket\_spinlock 的情况下进行的测试，由于 primes 过慢，几经优化也没优化到 30s 内，我将其 grade 脚本时间限制改为 50 后可正常跑出结果。Ticket\_spinlock 与 challenge sfork 我都有写明相关的测试方法，请 ta 单独测试，如有问题请联系 [wy30123@163.com](mailto:wy30123@163.com)。