Lab4实验报告

思考题

Thinking 4.1

- 内核使用 `SAVE_ALL` 宏来保护现场，并在保护现场时仅用了 `k0` 寄存器，先把 `sp` 寄存器复制到 `k0` ，之后依次复制其他的寄存器
- 可以
- 用户调用 `msyscall` 以及陷入内核时， `a0 - a3` 的值不会被破坏，用户栈的内容也会拷贝到内核栈，因此 `sys` 开头的函数可以获取到和 `msyscall` 时相同的参数
- 将栈中存储的 `EPC` 寄存器值增加 4, 这是因为系统调用之后, 将会回到下一条指令, 而用户程序会保证系统调用操作不在延迟槽内, 所以直接加 4 得到下一条指令的地址

Thinking 4.2

生成 `env_id` 到调用的过程中, 若某一进程完成了运行, 资源被回收, 这些对应的进程控制块会插入回收 `env_free_list` 中. 当我们重新获取该进程进程控制块时, 并在为其赋予新的 `env_id` 之前, 就会出现旧的 `env_id` 和新的 `env_id` 都能通过, 所以需要判断这种情况

Thinking 4.3

```
u_int mkenvid(struct Env *e) {
    static u_int i = 0;
    return ((++i) << (1 + LOG2NENV)) | (e - envs);
}
```

由于 `(++i) << (i + LOG2NENV)` 不会为 0, 因此生成的 id 不会为 0

且在指导书中可知, 在IPC调用时, `envid = 0` 即为 `curenv`

Thinking 4.4

- C

Thinking 4.5

在 `0 ~ USTACKTOP` 范围内的内存需要使用 `duppage` 进行映射

`USTACKTOP ~ UTOP` 之间 user exception stack 是用来处理页写入异常的, 不会在处理 COW 异常时调用 `fork()` , 因此不需要共享

`USTACKTOP ~ UTOP` 之间的 invalid memory 是为处理页写入异常时做缓冲区的, 所以也不需要共享

`UTOP` 以上页面所有进程共享, 用户进程无权限, 不需要 `duppage`

Thinking 4.6

```
11 #define vpt ((const volatile Pte *)UVPT)
12 #define vpd ((const volatile Pde *)(UVPT + (PDX(UVPT) << PGSHIFT)))
```

- `vpd` 是页目录的首地址, `vpt` 是页表的首地址
- 从实现的角度, 操作系统在初始化页表时, 会将页表和页目录的一部分映射到虚拟地址空间中, 使得这些虚拟地址直接指向页表和页目录的实际物理地址
- `vpd` 的值恰好映射到用户页表的空间, 也就是说 `vpd` 本质上只是一个特殊的页表, 这就体现了自映射的设计
- 进程不可以修改自己的页表项, 该区域对于用户只读不写, 若想要增添页表项, 需要陷入内核进行操作

Thinking 4.7

- 当出现COW异常时, 需要使用用户态的系统调用发生中断, 即中断重入
- 内核将 `Trapframe` 复制到用户空间, 主要是为了让用户程序能够访问异常发生时的状态信息, 这在调试和信号处理中有用

Thinking 4.8

- 错误和崩溃被限制在单个进程中, 可以尽量减少内核崩溃的可能性
- 减少上下文切换, 解放内核

Thinking 4.9

- `syscall_exofork()` 返回后父子进程各自执行自己的进程, 子进程需要修改 `entry.S` 中定义的指针, 涉及到 COW 页面的修改, 会触发 COW 写入异常, COW 中断的处理机制依赖于 `syscall_set_tlb_mod_entry`, 所以将其的调用放置在 `syscall_exofork` 之前
- 父进程在调用 COW 机制可能引发缺页异常, 而异常处理未设置好, 则不能正常处理

实验难点

1. 系统调用全过程

在本实验中，需要我们掌握添加一个系统调用的全过程，对一个系统调用发生的各个阶段有充分的理 解，同时要理解每个阶段发生的位置（用户态还是内核态），理解各个阶段都做了什么

2. fork 的逻辑以及实现

   fork 的实现涉及的概念以及函数非常多, 需要对于整个操作系统有详细的深入的了解和认识

体会与感想

感谢助教老师在本次实验过程中给予的悉心指导!

我不仅清晰地理解了系统调用的完整流程，更对进程间通信机制建立了系统化的认知

每一次的 lab 都开启一张全新的地图, 真的很有意思！期待下一次的实验!