Lab4实验报告

> 学号：21371220
>
> 姓名：杨硕

一、思考题

thinking 4.1

- 内核在保存现场的时候是如何避免破坏通用寄存器的?

  在kern/entry.S中通过SAVE_ALL宏，将所有寄存器保存到指定位置，进而保证在后续步骤不会破坏通用寄存器

- 系统陷入内核调用后可以直接从当时的 $a0-$a3 参数寄存器中得到用户调用 msyscall

  留下的信息吗?

  可以，$a0-$a3参数寄存器值没有变化

- 我们是怎么做到让 sys 开头的函数“认为”我们提供了和用户调用 msyscall 时同样

  的参数的?

  在 do_syscall函数中先取出 $a0(系统调用号) 到 $a3(前三个参数)，再从用户栈中取出其他的参数，最后将这些参数保存到内核栈中，模拟使得内核态的 sys 函数可以正常将这些参数传入到函数中。sys开头的函数会从内核栈中找传递的参数，这样sys 开头的函数就”认为”我们提供了和用户调用 msyscall 时同样的参数

- 内核处理系统调用的过程对 Trapframe 做了哪些更改？这种修改对应的用户态的变

  化是什么?

  do_syscall函数取出Trapframe的EPC并加4，这样在返回用户态后，执行syscall的后一条指令，将返回值存入$v0 ，用户态可以正常获得系统调用的返回值

thinking 4.2

ENVX宏函数的定义为:

```
#define ENVX(envid) ((envid) & (NENV - 1))
```

而NENV为`1 << 10`，也就是说，ENVX(envid)是取的是envid的低10位（保证在数组范围内）

增加`e->env_id != envid`的判断，是为了防止envid的高位与env_id不相等，envid实际是无效的

thinking 4.3

`mkenvid()`函数

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

首先，在 IPC 中如果要发送消息，需要通过 envid 找到对应进程，这依靠 envid2env函数。

而在`envid2env`函数中，有这样一段代码

```
if(envid == 0){
        *penv = curenv;
        return 0;
}
```

即传入的envid是0时，找到的是当前进程，也就是说无法通过envid2env找到进程号为0的进程（当前进程未必就是进程号为0的进程）。此时如果当前进程发送消息，便会造成消息错误发送（发送给自己），接受方也无法收到对应的消息。

因此`mkenvid()`函数的返回值不为0（也就是不设置进程的进程号为0）

thinking 4.4

C、fork 只在父进程中被调用了一次，在两个进程中各产生一个返回值

thinking 4.5

从mmu.h的内存布局图来看，需要保护的用户空间页存在于UTEXT到USTACKTOP的这一段，USTACKTOP往上属于无效内存(不需要被保护和user exception stack(独属于每个进程，不能映射)

进一步，在UTEXT到USTACKTOP这一段，也不是所有页都需要被保护：

- 只读的页面不需要被保护。
- 共享的页面不需要被保护

thinking 4.6

- vpt 和 vpd 的作用是什么？怎样使用它们？

  vpd和vpt宏定义如下

  ```
  #define vpt ((volatile Pte *)UVPT)
  #define vpd ((volatile Pde *)(UVPT + (PDX(UVPT))))
  ```

  UVPT是用户虚拟页表的起始地址，即vpt指向页表基地址，vpd指向页目录基地址

  使用：*(vpd+vpn) / *(vpt+vpn)或vpd[vpn] / vpt[vpn]这样来找到对应页表项

- 从实现的角度谈一下为什么进程能够通过这种方式来存取自身的页表？

  vpt 与 vpd 本质上是通过宏定义的方式来对用户态的一段内存地址进行操作，因此使用这种方式实际上就是在使用 mmu.h 内存布局图中的地址，所以可以通过这种方式来存取进程自身页表

- 它们是如何体现自映射设计的？

  vpd本身处于vpt段中，也就是说，页目录本身处于其所映射的二级页表中的一个页面中。

- 进程能够通过这种方式来修改自己的页表项吗？

  不能，进程处于用户态，不能修改自身页表项

thinking 4.7

- 这里实现了一个支持类似于“异常重入”的机制，而在什么时候会出现这种“异常重入”?

  在用户进程触发了页写入异常的时候

- 内核为什么需要将异常的现场 Trapframe 复制到用户空间?

  在微内核结构中，对缺页错误的处理由用户进程完成，用户进程在处理过程中需要读取 Trapframe 的内容；

  处理结束后，由用户进程恢复现场，也需要用到 Trapframe 中的内容

thinking 4.8

在内核态处理会增加操作系统内核的工作量，将异常处理交给用户进程，可以让内核做其他更多的事情

thinking 4.9

- 为什么需要将 syscall_set_tlb_mod_entry 的调用放置在 syscall_exofork 之前?

  在调用syscall_exofork的过程中用到了env_alloc，在这里可能需要进行缺页中断异常处理

- 如果放置在写时复制保护机制完成之后会有怎样的效果?

  触发异常时异常处理未设置好导致无法正常处理

二、实验难点

lab4的难点之一便是系统调用的过程（在用户态和内核态来回切换，容易晕）

用户通过操作系统运行上上层程序（如系统提供的命令解释程序或用户自编程序），而这个上层程序的运行依赖于操作系统的底层管理程序提供服务支持，当需要管理程序服务时，系统则通过硬件中断机制进入内核态，运行管理程序；也可能是程序运行出现异常情况，被动的需要管理程序的服务，这时通过异常处理来进入内核态。管理程序运行结束时，用户程序需要继续运行，此时通过相应的保存的程序现场退出中断处理程序或异常处理程序。返回断点继续执行

在实验中，实现一个系统调用需要进行下面几步：

1. 用户态进程调用位于 user/syscall_lib.c 中的 syscall_xxx 函数。
2. syscall_xxx 将相关的参数传入 msyscall 中。
3. msyscall 跳转至handle_sys中，通过do_syscall函数将用户栈与相关寄存器转移到内核栈与寄存器中，并跳转到位于 lib/syscall_all.c 的相应的 sys_xxx 函数。
4. 执行 sys_xxx 函数并返回，并恢复栈。
5. 返回用户态。

三、实验心得

lab4的主要难点就在于系统调用过程中，用户态和内核态切换。开始不是很理解这个过程和机制，在几个文件之间来回查看，找实现对应功能的函数，一会就找晕了。在理解了系统调用的过程后，根据指导书的指导，逐渐找到了完成lab的方向

但感觉有些地方指导书还是写的模糊（可能是我学的不仔细），有些宏与函数光看指导书和hint并不能很快找到或是知道用法（有些是问了先做了的室友才搞懂），对我这样看指导书和代码不仔细的菜鸡真的蛮头疼的，经常有思路却不知道具体怎么实现。