BUAA-OS-Lab4

一、思考题

Thinking 4.1

> 思考并回答下面的问题：
>
> •内核在保存现场的时候是如何避免破坏通用寄存器的？
>
> •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？
>
> •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？
>
> •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？

1、

```
mfc0    k0, CP0_STATUS
    sw      k0, TF_STATUS(sp)
```

先把通用寄存器的值复制到k0中，再将k0中的值存储到内核栈中。

2、可以。

3、将参数复制到内核现场寄存器和内核栈中。

4、返回值存入$v0中，pc+4跳转到下一条指令。若是nop则pc不变。

Thinking 4.2

> 思考 envid2env函数: 为什么 envid2env中需要判断 e->env_id != envid 的情况？如果没有这步判断会发生什么情况？

因为使用envid从envs中取出的进程的env_id可能与envid不同，如果没有这步判断会导致进程出错。

Thinking 4.3

> 思考下面的问题，并对这个问题谈谈你的理解：请回顾 kern/env.c 文件 中 mkenvid() 函数的实现，该函数不会返回 0，请结合系统调用和 IPC 部分的实现与 envid2env() 函数的行为进行解释。

在mkenvid() 函数中i一开始为0，++i后使得其返回值不为0。

Thinking 4.4

> 关于 fork 函数的两个返回值，下面说法正确的是：
>
> A、fork 在父进程中被调用两次，产生两个返回值
>
> B、fork 在两个进程中分别被调用一次，产生两个不同的返回值
>
> 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 D、fork 只在子进程中被调用了一次，在两个进程中各产生一个返回值

C

Thinking 4.5

> 我们并不应该对所有的用户空间页都使用 duppage 进行映射。那么究竟哪 些用户空间页应该映射，哪些不应该呢？请结合 kern/env.c 中 env_init 函数进行的页 面映射、include/mmu.h 里的内存布局图以及本章的后续描述进行思考。

UCOW-UTEXT中的内存是被COW保护的所以不应该进行映射，0-UTEMP的内存为invalid memory ，不应该进行映射，UTEMP-UCOW的内存是暂时保护的内存，不需要映射。

Thinking 4.6

> 在遍历地址空间存取页表项时你需要使用到 vpd和 vpt这两个指针，请参 考user/include/lib.h 中的相关定义，思考并回答这几个问题：
>
> • vpt 和 vpd 的作用是什么？怎样使用它们？
>
> • 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？
>
> • 它们是如何体现自映射设计的？
>
> • 进程能够通过这种方式来修改自己的页表项吗？

作用: 在用户态下通过访问进程自己的物理内存获取用户页的页目录项页表项的 perm，用于 duppage 根据不同的 perm 类型在父子进程间执行不同的物理页映射；

使用：vpd是页目录首地址，以vpd为基地址，加上页目录项偏移数即可指向va对应页目录项; vpt是页表首地址，以vpt为基地址，加上页表项偏移数即可指向va对应的页表项。

因为页目录映射和页表映射使得进程能够存取自身页表。

vpd的地址在UVPT和UVPT + PDMAP之间，说明将页目录映射到了某一页表位置(即实现了自映射);

不能。该部分区域对用户只读不写。

Thinking 4.7

> 在 do_tlb_mod 函数中，你可能注意到了一个向异常处理栈复制 Trapframe 运行现场的过程，请思考并回答这几个问题：
>
> • 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”？
>
> • 内核为什么需要将异常的现场 Trapframe复制到用户空间？

当出现COW异常时，需要使用用户态的系统调用，此时会出现这种异常重入；由于用户态把异常处理完毕后仍然在用户态恢复现场，所以此时要把内核保存的现场保存在用户空间的用户异常栈。

Thinking 4.8

> 在用户态处理页写入异常，相比于在内核态处理有什么优势？

释放内核，不需要内核执行大量工作；影响小，用户状态下不能得到一些在内核状态才有的权限，避免改变不必要的内存空间，内核态处理失误产生的影响较大，可能会使得操作系统崩溃。

Thinking 4.9

> 请思考并回答以下几个问题：
>
> • 为什么需要将 syscall_set_tlb_mod_entry的调用放置在 syscall_exofork之前？
>
> • 如果放置在写时复制保护机制完成之后会有怎样的效果？

syscall_exofork()执行后，子进程和父进程各自执行，子进程需要改变entry.S中的env指针，会触发COW写入异常，中断处理机制需要syscall_set_tlb_mod_entry()执行，所以将syscall_set_tlb_mod_entry 的调用放置在 syscall_exofork 之前；父进程在调用写时复制保护机制可能会引发缺页异常，而异常处理未设置好，则不能正常处理。

二、实验难点

do-syscall函数

do_syscall函数借助保存的Trapframe结构体获取用户态中传递过来的值，从而恢复用户态现场。

首先需要从syscall_table中取得调用的函数:

```
func = syscall_table[sysno];
```

再将tf指针中的保存的值存入参数中：

```
u_int arg1 = tf->regs[5];
u_int arg2 = tf->regs[6];
u_int arg3 = tf->regs[7];
arg4 = *(u_int *)(tf->regs[29] + 16);  // $sp + 16 bytes
arg5 = *(u_int *)(tf->regs[29] + 20);
```

最后调用func函数，同时将其返回值存入寄存器中：

```
tf->regs[2] = func(arg1, arg2, arg3, arg4, arg5);
```

envid2env函数

这个函数主要功能是将envid转换为env，我们首先需要判断是否为curenv：

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

如果不是则从envs[]取出对应的env:

```
e = &envs[ENVX(envid)];
```

注意ENVX的功能是将envid转换为对应其在envs[]中的位置:

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

还需要判断checkperm:

```
if (checkperm && e->env_id != curenv->env_id && e->env_parent_id != curenv-
>env_id) {
        return -E_BAD_ENV;
    }
```

sys_exofork函数

我们需要复制一份当前进程的运行现场（进程上下文）Trapframe到子进程的进程控制 块中：

```
e->env_tf = *((struct Trapframe *)KSTACKTOP - 1);
```

duppage函数

我们要对不同权限位的页使用不同方式进行处理。

只读页面：对于不具有PTE_D权限位的页面，按照相同权限（只读）映射给子进程即可。

写时复制页面：即具有PTE_COW 权限位的页面。这类页面是之前的 fork 时 duppage 的结 果，且在本次fork前必然未被写入过。

共享页面：即具有PTE_LIBRARY 权限位的页面。这类页面需要保持共享可写的状态，即在 父子进程中映射到相同的物理页，使对其进行修改的结果相互可见。在文件系统部分的实 验中，我们会使用到这样的页面。

可写页面：即具有PTE_D权限位，且不符合以上特殊情况的页面。这类页面需要在父进程和 子进程的页表项中都使用PTE_COW权限位进行保护。

我们只需判断perm是否具有PTE_D同时不具有PTE_LIBRARY，如果满足条件，则还需调用syscall_mem_map函数将当前进程映射到其自身页面上：

```
r = 0;
    if ((perm & PTE_D) && ((perm & PTE_LIBRARY) == 0)) {
        perm = perm & ~PTE_D | PTE_COW;
        r = 1;
        try(syscall_mem_map(0, addr, envid, addr, perm));
    } else {
        try(syscall_mem_map(0, addr, envid, addr, perm));
    }

    if (r) {
        try(syscall_mem_map(0, addr, 0, addr, perm));
    }
```

三、体会与感想

本次实验对主要需要掌握：系统调用，IPC通信机制，fork进程创建，页面写入异常处理，难度相较而言比较大，需要更仔细。