BUAA-OS Lab4 实验报告

姓名：苏云鹤
班级：212114
学号：21373007

1. 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？

- 内核调用SAVE_ALL宏函数保存现场，同时这个宏函数只使用$k1$、k2两个寄存器操作，而不破坏其他寄存器的值。
- 不可以。在陷入内核调用后，`$a0` 寄存器中的值已被改变为系统调用号。虽然 `$a1-$a3` 未被使用，但其也可能被改变。若想得到原始信息，应从栈中获取。
- 将用户进程的寄存器 `$a0-$a3` 以及用户栈中的参数a4，a5拷贝至内核寄存器`$a0-$a3`和内核栈。
- 设置`tf->cp0_epc+4`，以便从内核调用返回用户态是可以正确执行调用后的下一条指令。

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?

某一进程完成运行，资源被回收，这时其对应的进程控制块会插入回 `env_free_list` 中。当我们需要再次创建内存时，就可能重新取得该进程控制块，并为其赋予不同的 envid。这时，已销毁进程的 envid 和新创建进程的 envid 都能通过 `ENVX` 宏取得相同的值，对应了同一个进程控制块。可是已销毁进程的 envid 却不应当再次出现，`e->env_id != envid` 就处理了 `envid` 属于已销毁进程的情况。

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。

如果传入的envid值为0，则会返回当前进程的进程控制块的指针。因此可以通过传入0值来直接获得当前进程控制块的指针，便于访问。

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

答案：C 父进程返回值是子进程的env_id，子进程返回值是0。

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。

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？

（1）vpt 和 vpd 的作用是什么？怎样使用它们？

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

`vpt` 可以理解成一个 `Pte` 类型的数组，其中按虚拟地址的顺序存储了所有的页表项；也可以理解成用户空间中的地址，并且正是页表自映射时设置的基地址。
`vpd` 则是`Pde` 类型的数组，储存的是页目录项。
使用时，通过 `基地址 + 偏移` 的方式获取需要的页表项/页目录项，类似数组的使用方式。

（2）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？
虚拟地址是顺序存储的，而且已经实现了自映射。

（3）它们是如何体现自映射设计的？

```
e->env_pgdir[PDX(UVPT)] = PADDR(e->env_pgdir) | PTE_V
```

（4）进程能够通过这种方式来修改自己的页表项吗？ 不可以。进程在用户空间运行，只能访问页表项，但无权修改。页表项是在内核态下维护的。
可以通过系统调用`syscall_mem_map`实现。

```
int flag = 0;
if ((perm & PTE_D) && !(perm & PTE_LIBRARY)) {
    perm = (perm & ~ PTE_D) | PTE_COW;
    flag = 1;
}

syscall_mem_map(0, addr, envid, addr, perm);

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

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?

中断重入就是在一个中断程序执行过程中又被另一个中断打断，转而又去执行另一个中断程序。 在内核态中，CPU因为中断屏蔽位被置为1而不支持异常重入。但由于MOS的微内核结构，对于 `COW` 的处理是部分在用户态下完成的，因此处理的全过程不能保证中断屏蔽位是1，因此可能出现异常重入。在 `do_tlb_mod` 的处理过程中，如果触发了新的 `do_tlb_mod` ，就会出现这种中断重入。在用户态中，页写入异常处理函数是由用户态程序通过调用`sys_set_user_tlb_mod_entry()` 指定自己的页写入异常处理函数。如果用户程序自定义的页写入异常处理函数写了一些新的全局变量，则会导致异常重入。

缺页错误是在用户空间处理的。由于用户进程无法访问到内核空间的数据，因此需要将异常现场的 `Trapframe` 复制到用户空间。

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?

- 减小内核的体积，符合微内核架构思想。
- 减少关中断时间，提高了中断处理的效率。
- 减少了内核出错的可能，利于系统的稳定。

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?

- 子进程执行的起始点是从syscall_exofork()返回用户态后的第一条指令，因此应在这之前父进程设置好自己的缺页处理机制。如果放在syscall_exofork()之后，则子进程会再次执行syscall_set_tlb_mod_entry()为env_user_tlb_mod_entry赋值并且为cow_entry分配空间，但是这实际上在子进程执行前就已经完成，所有造成了额外的资源开销。
- 如果放在写时复制保护机制完成后才执行，则会导致父进程进行duppage时遇到COW缺页异常时没有处理函数处理，导致页写入异常处理机制无法建立。

2. 实验难点

2.1. 系统调用的流程

- 用户调用 `sys_` 函数使用系统调用;
- `sys_` 调用 `mysyscall` 函数; `mysyscall`执行 `syscall` 指令，指定CPO_CAUSE为8 (系统调用的异常号)，陷入内核态
- 执行异常处理函数`exc_gen_entry`，用`SAVE_ALL`保存用户上下文至`TrapFrame`结构体，调用对应的`exception handler`然后调用`do_syscall`;
- `do_syscall`取出`TrapFrame`的参数，执行对应的实际处理系统调用的函数，保存系统调用的返回值至`TrapFrame`回到`exception_handler`，执行`ret_from_exception`将返回值返回`mysyscall`;
- `mysyscall`执行`jr ra`返回用户程序。

2.2. fork的流程

- 父进程通过·syscall_set_tlb_mod_entry·设置自己的TLB MOD异常处理函数。

- 调用`syscall_exofork`函数创建子进程。
- 父进程使用 `duppage` 将需要给子进程共享的页复制给子进程，标记`COW`并建立写时复制机制。
- 父进程通过`syscall_set_tlb_mod_entry`设置子进程的TLB MOD异常处理函数。
- 父进程通过`syscall_set_env_status`设置子进程的状态为可运行，此时子进程就已经可以运行了。

3. 实验体会

1. 初步掌握了操作系统系统调用的流程；
2. 进一步体会到了os实验的魅力~