Lab4实验报告

实验思考题

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？

答案

- 先存储原现场堆栈指针位置和2号寄存器位置，防止后续操作对其造成影响
- 不能，从堆栈的相应位置取出其值。
- 在handle_sys中将对应参数存入a0-a，以及堆栈的对应位置
- 对Trapframe里的epc+4，使得用户态返回后能进入正确的执行位置，而且把v0的值存入Trapframe的2号寄存器中可以让系统调用获得正确的返回值。

Thinking 4.2

思考下面的问题，并对这个问题谈谈你的理解： 请回顾 lib/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。

在操作系统中会使用envid=0来表示当前进程，所以在mkenvid中不会返回0，否则会产生矛盾。

Thinking 4.3

思考下面的问题，并对这两个问题谈谈你的理解：

子进程完全按照 fork() 之后父进程的代码执行，说明了什么？

但是子进程却没有执行 fork() 之前父进程的代码，又说明了什么？

父子进程具有相同的代码段。

子进程对之前父进程的内存状态、上下文、PC值与进程状态等都进行了复制。

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，在父进程中fork返回子进程id，子进程中返回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？ 请结合本章的后续描述、mm/pmap.c 中 mips_vm_init 函数进行的页面映射以及 include/mmu.h 里的内存布局图进行思考。

用户页中USTACKTOP以下中存在的物理地址需要进行映射，而USTACKTOP以上的部分不应该映射，他们有部分是内核态空间，没有访问权限，有些是空闲区，用户异常栈，页表，还有共享的物理内存控制块和进程控制块，不需要duppage

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“指针的指针”，请参考 user/entry.S 和 include/mmu.h 中的相关实现，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？

答案

- vpy表示页表所在位置，vpd表示页目录所在位置，使用它们通过页目录的子映射机制可以更好的获得对应的页目录项和页表项，具体的使用方法为`((Pte*)(*vpt))[va>>12]`表示页表项，`((Pde*)(*vpd))[va>>22]`表示页目录项。
- 页表的起始地址是4MB对齐的，通过页目录自映射机制可以实现每个页面的对齐
- vpt指向页表的起始地址，vpd指向页目录的起始地址从而实现自映射机制
- 进程不能通过这种方式修改自己的页表项

Thinking 4.7

page_fault_handler 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“中断重入”的机制，而在什么时候会出现这种“中断重入”？
- 内核为什么需要将异常的现场Trapframe复制到用户空间？

答案

- 因为在用户态进行异常的处理，所以这个操作是非原子的，在用户进行写时复制时，可能出现缺页，此事发生中断重入
- 因为MOS使用微内核设计，把终端缺页的处理交给了用户进程，所以用户进程需要读取Trapframe的值来获取哪一条指令发生了缺页，从而得到缺的页面是哪一页，并进行调页，用户进程处理完毕恢复现场的时候也需要使用Trapframe的数据

Thinking 4.8

到这里我们大概知道了这是一个由用户程序处理并由用户程序自身来恢复运行现场的过程，请思考并回答以下几个问题：

- 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？
- 从通用寄存器的用途角度讨论，在可能被中断的用户态下进行现场的恢复，要如何做到不破坏现场中的通用寄存器？

答案

- 减少了内核出现异常的可能，即使进程崩溃也不会导致系统崩溃
- 先恢复除堆栈寄存器以外的通用寄存器，最后通过延时机制恢复堆栈寄存器

Thinking 4.9

请思考并回答以下几个问题：

- 为什么需要将set_pgfault_handler的调用放置在syscall_env_alloc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？
- 子进程是否需要对在entry.S定义的字__pgfault_handler赋值？

答案

- 如果先进行 `syscall_env_alloc` 再进行 `set_pgfault_handler` 可能会导致之间出现写入异常
- 由于无法处理页写入异常，写时复制保护机制不能执行
- 不需要，这个值已经在父进程中进行了相关的设置

实验难点图示

handle_sys函数

该函数的主要功能是设置epc+4，然后设置好系统调用时传入的参数，设置栈指针，并跳转到对应的处理函数，此外提一下系统调用的过程

用户态下调用 `user/lib.h` 中syscall开头的系统调用函数，随后在 `syscall_lib.c` 跳转到mysyscall，在 `user/syscall_wrap.S` 中syscall调转到内核态触发系统中断，并在 `lib/syscall.S` 的 `handle_sys` 中根据 `include/unistd.h` 中的系统调用号跳转到对应的系统函数(在 `lib/syscall_all.c` 里)，处理完毕后跳转到原位置。

sys_ipc_can_send与sys_ipc_recv函数

这两个函数主要用来处理进程间通信，进程间通信的主要原理是通过读写一段公共的区域(或者在struct Env中的env_ipc_value中写入)来完成，ipc_recv主要是让进程处于可接受状态，随后sys_ipc_can_send会根据envid找到接收信息的进程，如果目标进程处于可接受状态则发送信息，并设置目标进程为 `runnable` (在ipc_recv函数中被设置为not runnable)，这样进程间通信就完成了。

fork函数

写fork函数之前需要先了解fork函数是用来干嘛的

对于 `fork` 函数，在调用后便兵分两路分为两个进程(父子进程)，二者的区分是 `fork` 的返回值不同，父函数的 `fork` 返回子进程的 `pid` ，子进程的 `fork` 返回0，因此可以使用以下结构实现两个进程。

```
int pid=fork();
if(pid==0){
    //子进程 do things
}else{
    //父进程 do things
}
```

因此可以看出fork函数实现的过程，父子进程的状态和上下文在调用fork时是相同的，所以需要共享一部分内存空间，也进而引入写时复制来实现有一方写入时的问题。

duppage函数

这个函数难点主要是判断权限位的值进而进行不同的操作，仔细想想就问题不大。

体会与感想

lab4主要是系统调用的流程需要理解，此外自己实现了fork函数也加深了对他的理解。lab4写了大概一天，但是又花了很长时间自己翻阅代码并加深理解应对上机测试。

说一说两次的上机，第一次上机题目比较中规中矩，lab4-1-exam是加一个自旋锁，维护和一个锁变量即可，实现比较简单。Extra是实现一个新版的进程间通信，这个版本比课下实现的版本稍微复杂一些，做之前应该仔细看看实验提示弄明白题意（在这里浪费了很多时间，不要题目没读懂上来就开始写代码），然后一步步按着实现就行，接受、发送的函数各自又两种处理情况，目标进程的状态如何，记得在里面加上`sys_sched`就没啥问题，题目说的挺清楚的

对于lab4-2-exam，也比较简单，根据vpt和vpd读页表内容，然后按要求更改权限位就行，Extra有点离谱，目测工作量得一天，果断放弃，有空再来写写。

总体来说，lab4的系统调用部分写代码时感觉比前几次舒服很多，可能是自己对mos的理解更深了（但愿如此）

指导书反馈

无

残留难点

lab4-2-Extra，题目要求还应该实现处理函数的继承，感觉需要对处理函数做存储并使用写时复制机制来完成功能，函数放的位置还没想明白，有机会问问别的同学或者再自己想一想。