Lab4实验报告

一、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？**

  采用通用栈帧寄存器sp的偏移来控制存储地址

- **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，syscall之后就是陷入异常处理，在进到handle_sys中前均没有破坏过$a0-$a3

- **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？**

  存的时候都按6个参数存

- **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？**

  修正EPC（EPC+4），处理了异常在延迟槽的情况（没必要，msyscall的包装防止了syscall出现在延迟槽）

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。

mkenvid函数中`return (asid << (1 + LOG2NENV)) | (1 << LOG2NENV) | idx`有一项`1 << LOG2NENV`，注定返回值不会为0。

解释：在系统调用和IPC实现中，我们大量使用`envid2env(0, ...)`，因为在envid2env中，envid == 0的情况是用来找当前运行进程env的，所以0不能分配给特定某进程。

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？**

  子进程从父进程复制而来，两者程序内容镜像方面相同

- **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？**

  子进程继承当前父进程运行上下文状态，通用寄存器和程序计数器状态相同。

Thinking 4.4

C，已在课程网站作答

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。

UTEXT~USTACKTOP：正常用户程序可操纵的空间，需要复制映射

USTACKTOP~UTOP：共两页，低地址页为Invalid memory，自然无需映射；高地址页为user exception

stack，进程异常栈，各进程不同，无需映射

UTOP以上：内核段，用户不可操纵，无需映射

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？**

  作用：获取页目录项和页表项内容

  使用方法：就vpt具体谈，`(*vpt)`获得该指针数组的第一项，即UVPT，`(*vpt)[va >> PGSHIFT]`即获得页表项内容；同理，`(*vpd)[va >> PDSHIFT]`即获得页目录项内容

- **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？**

  user/entry.S文件：

  ```
      .global vpt     # .global告诉编译器后续跟的是一个全局可见的名字
  vpt:
      .word UVPT  # vpt标签是地址UVPT，即vpt表示一个地址，这个地址指向UVPT
                  # UVPT是MOS虚拟页表的地址，因此相应操作见下：
                  # (*vpt)表示UVPT，即页表项基地址，(*vpt)[offset]即各偏移量页表项内
  容

      .global vpd
  vpd:
      .word (UVPT+(UVPT>>12)*4)
  ```

  include/mmu.h文件：

  ```
  extern volatile Pte *vpt[];     // 指针数组，数组每项为Pte *，也未为不可
  extern volatile Pde *vpd[];
  ```

- **它们是如何体现自映射设计的？**

  `(UVPT+(UVPT>>12)*4)`即为页目录地址，也即*vpd，是为自映射

- **进程能够通过这种方式来修改自己的页表项吗？**

  不可。

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”？**

  用户在处理页写入异常时，可能再进入其他中断

- **内核为什么需要将异常的现场Trapframe复制到用户空间？**

  页写入异常等操作实在用户态下进行的，所以每个用户进程都需要存入相关的现场数据

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？**

  无需陷入内核，处理更快捷；解耦，处理出错不会影响内核运行。总的说就是具有微内核的优势。

- **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？**

  在保存现场时，将现场的寄存器值存到专门异常处理栈中（UXSTACKTOP啥的），多次中断便不会因覆盖而损失现场信息。（不懂问题意思

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？**

  `syscall_env_alloc()`处理过程中也可能产生页写入异常

- **如果放置在写时复制保护机制完成之后会有怎样的效果？**

  写时复制保护机制中时分配新页时可能遇到页写入异常，此时将无法处理该异常。

- **子进程是否需要对在entry.S定义的字__pgfault_handler赋值？**

  不需要，两者__pgfault_handler相同

二、实验难点

1.增加一个系统调用的流程

- user/lib.h：增加 syscall_xxx函数 的声明，用户态函数声明应该都在这
- user/syscall_lib.c：增加 用户态系统调用包装函数syscall_xxx的实现，即调用`msyscall(...)`
- include/unistd.h：为该系统调用设定一个编号，即`define SYS_xxx ...`
- lib/syscall.S：将该系统调用加入syscall table，此番过后syscall即可通过该表跳转进内核函数sys_xxx——真正的系统调用目标函数
- lib/syscall_all.c：增加 内核态系统调用函数sys_xxx的实现

2.vpt和vpd的理解

mmu.h的引用感觉不是太合适，换成数组指针或者二重指针可能会好理解些。

3.fork流程理解

父进程：

1. `set_pgfault_handler(pgfault)`设置页写入异常处理函数入口地址
2. `syscall_env_alloc()`创建子进程
3. `duppage(newenvid, pn)`父进程对子进程页面空间进行映射以及设置PTE_COW
4. `syscall_set_pgfault_handler()`为子进程分配异常处理栈;`syscall_set_env_status()`为子进程设置相关状态状态，插入调度队列

子进程：

1. 设置子进程env

2. 子进程处理的页写入异常（适配之前的写时复制机制），假惺惺的同一般异常一样陷入内核，但什么处理都不做，返回时进入子进程的异常处理函数，真正处理相关异常：分配新的页并删除PTE_COW

三、体会与感想

做lab4的时候总的感觉较为通透。前半理解应该较为到位，后半段页写入异常时又有些糊了。主要是有多种类似保存现场的代码还有复制现场到另一位置的操作，关系挺混乱，再者lab2属实没全想清（？），就不太理解了。lab4-2的Extra感觉也是因此而想不清各种调用关系和异常发生状况而不好下手。

想通了一些事，又多了些没想通的事。

fork的最后一节容量分配和介绍不是很恰当，有些多而乱。