Lab4

> 班级：222111
>
> 学号：22373340
>
> 姓名：詹佳博

思考题

Thinking 4.1

- 系统从用户态切换到内核态后，内核首先需要将原用户进程的运行现场保存到内核空间（在kern/entry.S 中通过 SAVE_ALL 宏（实现于include/stackframe.h）完成），进而保证了在后续步骤不会破坏通用寄存器。
- 可以直接得到。以 MOS 操作系统为例，msyscall 函数一共有 6 个参数，前 4 个参数会被 syscall_* 的函数分别存入 $a0-a3$ 寄存器。内核态也能访问这几个寄存器。
- 调用方在自身栈帧的底部预留被调用函数的参数存储空间，由被调用方从调用方的栈帧中读取参数。当 user/lib/syscall_lib.c 中定义的用户包装函数 syscall_* 调用 msyscall 时，根据以上的调用规范，**需要传递给内核的系统调用号以及其他参数已经被合理安置。**内核首先需要将原用户进程的运行现场保存到内核空间，然后可以在 C 语言中通过这个 struct Trapframe * 来获取用户态现场中的参数。
- cp0_epc 字段指示了进程恢复运行时 PC 应恢复到的位置。内核处理系统调用的过程中，将其自增1个字的大小（在32位mips中，即4字节），也就是将 PC 指向了下一条指令。

Thinking 4.2

- 考虑到 `envs[]` 最多有 `NENV` 个进程。`ENVX(envid)` 保证了其访问的进程下标一定在`[0, NENV-1]`。我们确实可以访问到 `envid` 在进程控制块数组里的进程 `envs[ENVX(envid)]`，但我们**无法确保输入的id是进程id号，而仅仅是进程的物理位置与另一个进程相同的情况。**如果不存在就会出现 `e->env_id != envid`，此时应该返回错误。

Thinking 4.3

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

  观察该函数发现，其低10位的部分正好就是进程控制块 e 所对应的下标，这与 `envid2env()` 函数中取 `ENVX(envid)` 得到的下标一致。

  而因为 i >= 1，所以 `((++i) << (1 + LOG2NENV))` 恒为正数；e - envs 代表着 e 进程的下标，恒为自然数；所以返回值为正数与运算自然数，不会为 0。最差的情况即为 `e == envs[0]`，此时 `mkenvid(e) = 1000 0000 0000`。

Thinking 4.4

- 选 `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`
  ```

- 对于操作系统，子进程开始运行时的大部分上下文状态与原进程相同，包括程序和 fork 运行时的现场（包括通用寄存器和程序计数器 PC 等）。此时fork在子进程不会再执行了，只会接受到其之后的返回值。

Thinking 4.5

- 我们需要将 **0 到 USTACKTOP** 之间的用户空间进行映射，具体在 fork 函数中进行体现。可用 `for(i = 0; i < VPN(USTACKTOP); i++)` 遍历。在这部分地址空间，其中一部分是内核，另一部分是所有用户进程共享的空间。另外，需要跳过 `PTE_V` 为 0 的部分。

Thinking 4.6

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

  vpt可见是页表基地址，vpd可见是页目录基地址。vpt加上页表偏移数量就是va所对应的页表项。vpd加上页目录偏移量就是va所对应的页目录项。
- 在user/include/lib.h声明定义了**用户页表**和**用户页目录**的va。进程的页表在UVPT中保存。切换进程时，页表也会切换。
- 根据Lab3的自映射机理，则有PDBase = UVPT + UVPT >> 10，PDX 为右移22位，也就是vpd = UVPT + PDX(UVPT) << 12 = UVPT + (PDX(UVPT) << PGSHIFT)，也就实现了自映射。
- 不能。该区域对用户只读。

Thinking 4.7

- 在该函数还没有执行完的时候，因为各种原因导致这个函数又被调用执行。就会异常重入。
- 由lab3-exam的加深理解，tlb_mod的异常处理函数在用户态执行，需要知道异常发生时的状态，最后需要通过tf保存好的现场tf->regs[29]，使用其他函数恢复事先保存好的现场。

Thinking 4.8

- 微内核设计主张将传统操作系统中的设备驱动、文件系统等可在用户空间实现的功能，移出内核，作为普通的用户程序来实现。这样，**即使它们崩溃，也不会影响到整个系统的稳定。**
- 内核态处理失误产生的影响较大，**可能会使操作系统崩溃**。此外，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。

Thinking 4.9

- 因为 `syscall_exofork` 可能也需要处理这个异常。且如果在 `syscall_exo_fork` 之后再 `syscall_set_tlb_mod_entry`，那么子进程也会执行这个系统调用。
- 可能在duppage的时候有中断，但是没有异常处理的情况，即找不到cow_entry函数，无法进入缺页中断异常。

难点分析

1. `Exercise 4.2` 中有如下代码块：
  ```

```
/* Step 4: Last 2 args are stored in stack at [$sp + 16 bytes], [$sp + 20
bytes]. */
u_int arg4, arg5;
/* Exercise 4.2: Your code here. (3/4) */
// sp -> reg[29]
arg4 = *(u_int *)(tf->regs[29] + 16);
arg5 = *(u_int *)(tf->regs[29] + 20);
```

一开始没有完全理解 `in stack at` 的意思。注意 arg4，arg5 需要从**用户栈**中 sp+16，sp+20 处获取。

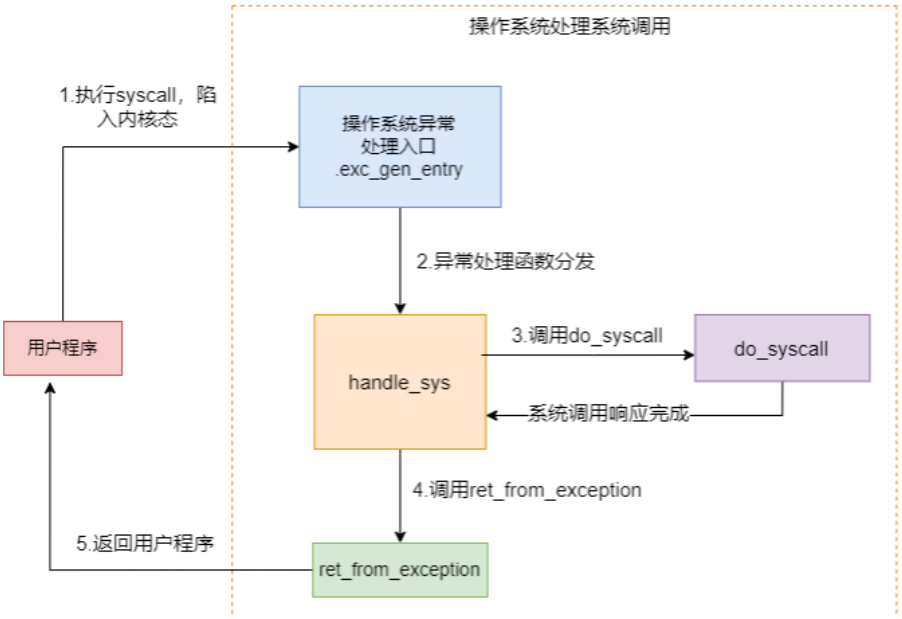
2. `Exercise 4.9` 中，有Copy the current Trapframe below 'KSTACKTOP' to the new env's 'env_tf'.
   `e->env_tf = *((struct Trapframe *)KSTACKTOP - 1);` 灵感来源于 `env_run()` 中类似代码。
3. 有点不懂为什么是0。后来懂了：0代表当前进程。

```
/* Step 3: Allocate a new page at 'UCOW'. */
/* Exercise 4.13: Your code here. (3/6) */
panic_on(syscall_mem_alloc(0, (void *)UCOW, perm));
// Step 5: Map the page at 'UCOW' to 'va' with the new 'perm'.
/* Exercise 4.13: Your code here. (5/6) */
panic_on(syscall_mem_map(0, (void *)UCOW, 0, (void *)va, perm));
```

4. `Exercise 4.15` 中，有遍历父进程地址空间，进行 duppage。但是我发现他总会在duppage的 `panic_on(syscall_mem_map(0, (void *)addr, envid, (void *)addr, perm));` 报错。
   发现遍历地址空间寻页的时候，可能会访问到 PTE_V 为 0 的页，它不可用，应当跳过。
   `if((PTE_FLAGS(vpt[i]) & PTE_V) && ((PTE_FLAGS(vpd[i >> 10]) & PTE_V)))`

实验体会

1. `set A to B` 是将 **A 设置为 B**！
2. 对 fork 函数的理解很难。虽然在注释的帮助下填写代码十分简单，但是其流程理解非常之困难。尤其是Thinking 4.9，直到目前为止还是有点不能理解。
3. Lab4-1 extra都提交git了，没来得及测评，其实考试前网络都卡了五分钟，但是没有延时...强烈建议课程组在课下之后能把21:00:00之前的评测都自动评测一下（
4. 加强了fork.c的理解。在exofork后面，父进程作为0应该把strace变成0，而子进程则不需要，故而strace要写在child==0里面。
5. Lab4-2 exam当时没有仔细想明白，没有将用户态的strace函数通过某一方法陷入内核，在处理ipc_send和set_status函数时候引发了未知的错误。正确来说应该用ipc_send让系统陷入内核。加强理解！
6. 处理系统调用的大致流程：


操作系统处理系统调用
1.执行syscall，陷入内核态
操作系统异常处理入口
.exc_gen_entry
2.异常处理函数分发
用户程序
handle_sys
3.调用do_syscall
do_syscall
系统调用响应完成
4.调用ret_from_exception
5.返回用户程序
ret_from_exception