杜佳男 21210110

操作系统lab4实验报告

1、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？**

  答：内核在保存现场时会用一个叫 `SAVE_ALL` 的汇编宏保存所有寄存器，首先利用 `$k0` 和 `$k1` 两个寄存器获取应该使用的 `$sp`，然后将 `$sp` 入栈，再利用 `$sp` 将通用寄存器入栈，保存现场以在退出异常时恢复。

- **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？**

  答：可以，因为在系统调用用到 `$a0~$a3` 前，这三个寄存器的值没有被改变过。

- **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？**

  答：在syscall.S中将参数拷贝到内核栈和对应寄存器中，模拟正常调用sys开头函数时的状态。

- **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：

  1. 将 `CP0_EPC` 的值加四写回，在返回用户态后从 `syscall` 的后一条指令开始执行；
  2. 将返回值存入 `v0` 寄存器，用户态可以正常获得系统调用的返回值。

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。

答：

1. 首先在上次的实验报告中已经分析过 `mkenvid` 函数的构成，从中可以发现，`envid` 由 `asid` 左移11位构成高5位，代表地址空间，第11位固定为1，低10位为进程在相对数组 `envs` 基地址的偏移。高5位和低10位都有可能为0，但因为第11位固定为1，因此保证了此函数不会返回0；

```
u_int mkenvid(struct Env *e) {
    u_int idx = e - envs;
    u_int asid = asid_alloc();
    return (asid << (1 + LOG2NENV)) | (1 << LOG2NENV) | idx;
}
```

```
#define LOG2NENV    10
#define NENV        (1<<LOG2NENV)
#define ENVX(envid) ((envid) & (NENV - 1))
#define GET_ENV_ASID(envid) (((envid)>> 11)<<6)
```

2. 如果`mkenvid`函数可以返回0，则在进行`env_alloc`等操作时，就可能会申请到`envid`为0的进程。而我们已经知道，在`envid2env`函数中，如果接受到进程号为0的进程则直接返回当前进程（`curenv`），因此上述`envid`为0的进程无法通过进程号被找到；
3. 紧接上条，在`IPC`中如果要发送消息，需要通过`envid`找到对应进程，而通过`envid2env`找到的是当前进程而不一定是想要发送到的`envid`为0的进程，从而造成消息错误发送（发送给自己），接受方也无法收到对应的消息；
4. 此外，在`fork`函数中，父进程的返回值为紫禁城的`envid`，子进程的返回值为0，如果存在进程号为0的进程，系统很可能会把一个父进程误认为是子进程，从而执行错误的操作。

Thinking 4.3

测试结果：

```
git@21210110:~/test_dir/test_fork$ ls
a.out  fork_test.c
git@21210110:~/test_dir/test_fork$ ./a.out
Before fork, var = 1.
After fork, var = 1.
After fork, var = 1.
parent got 16776, var = 1, pid: 16775
git@21210110:~/test_dir/test_fork$ child got 0, var = 2, pid: 16776
```

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？**

  答：子进程完全按照fork后父进程的代码执行，说明子进程和父进程共享代码段且具有相同的状态和数据。

- **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？**

  答：子进程没有执行fork之前父进程的代码，说明子进程在创建时复制了父进程的上下文环境、PC值等信息，与父进程有着相同的状态。

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

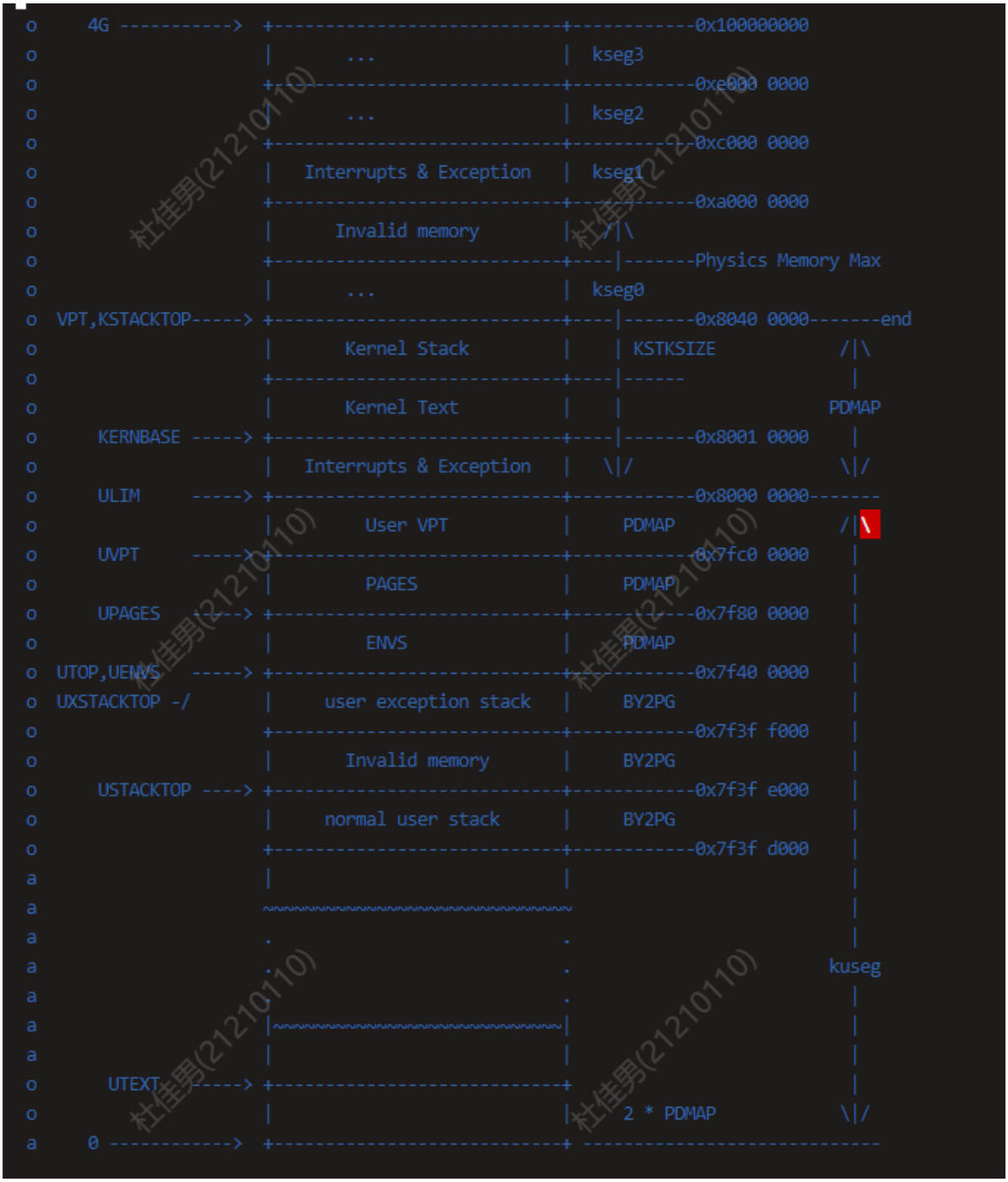
C、fork 只在父进程中被调用了一次，在两个进程中各产生一个返回值

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：选C。子进程在父进程调用fork时被创建，并赋予不同返回值，子进程返回值为0，父进程返回值为子进程的进程号。

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。



答：

- 从 0 到 USTACKTOP 的地址空间中，除不可写(perm&PTE_R=0)和父子进程共享(perm&PTE_LIBRARY=0)的页外都可以用 PTE_COW 保护；
- 从 USTACKTOP 到 UTOP 的地址空间中有两段，分别为无效内存和用户缺页异常处理栈，前者自然无需保护，后者若保护可能陷入死循环；
- UTOP 以上的地址空间对用户进程来讲不可变，无需保护。

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?**

  答：`vpt` 是指向二级页表指针的指针，`vpd` 是指向一级页表，也即页目录指针的指针。对于虚拟地址 `va`，`（*vpd）[va >> 22]` 为二级页表的物理地址，`(*vpt)[va >> 12]` 为 `va` 对应的物理页面。

- **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?**

  答：因为 `vpt` 和 `vpd` 通过宏定义对应了内存中页表所在的虚拟地址。

- **它们是如何体现自映射设计的?**

  答：`(UVPT+(UVPT>>12)*4)` 就是 `vpd` 指向的地址，只需要在页目录项中映射一个表项就可以将整个页表映射到 `UVPT~ULIM` 的 `4M` 空间。

- **进程能够通过这种方式来修改自己的页表项吗?**

  答：不能，对用户进程来说这些页表项的权限是只读的。

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”?**

  答：在用户发生写时复制（`COW`）引发的缺页中断并进行处理时，可能会再次发生缺页中断，从而出现“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：在微内核结构中，对缺页错误的处理由用户进程完成，用户进程在处理过程中需要读取 `Trapframe` 的内容；同时，在处理结束后同样是由用户进程恢复现场，会用到 `Trapframe` 中的数据。

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?**

  答：符合微内核设计理念，可以减小内核体积，提高中断处理效率。

- **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：在中断执行前将现场入栈，恢复时首先通过 `sp` 寄存器从栈中恢复其他通用寄存器，最后在在跳转延迟槽中恢复 `sp` 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?**

  答：因为在调用`syscall_env_alloc`的过程中也可能需要进行异常处理，在调用`fork`时，有可能当前进程已是之前进程的子进程，从而需要考虑是否会发生写时复制的缺页中断异常处理。

- **如果放置在写时复制保护机制完成之后会有怎样的效果?**

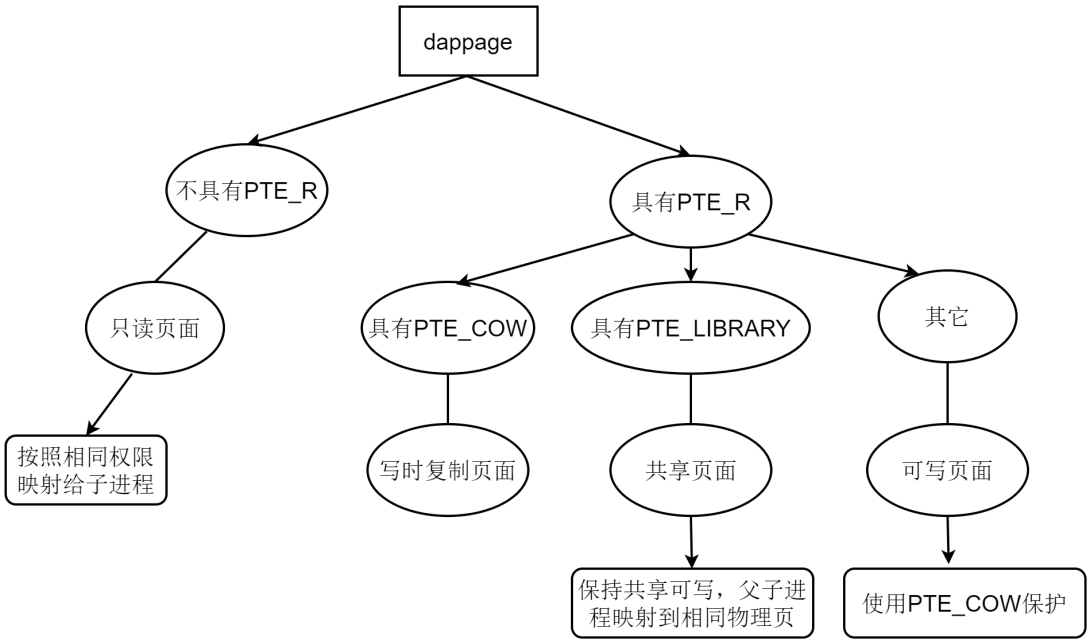
  答：进程在执行`set_pgfault_handler`前就会触发缺页中断，但因中断处理未设置好导致无法进行正常处理。

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

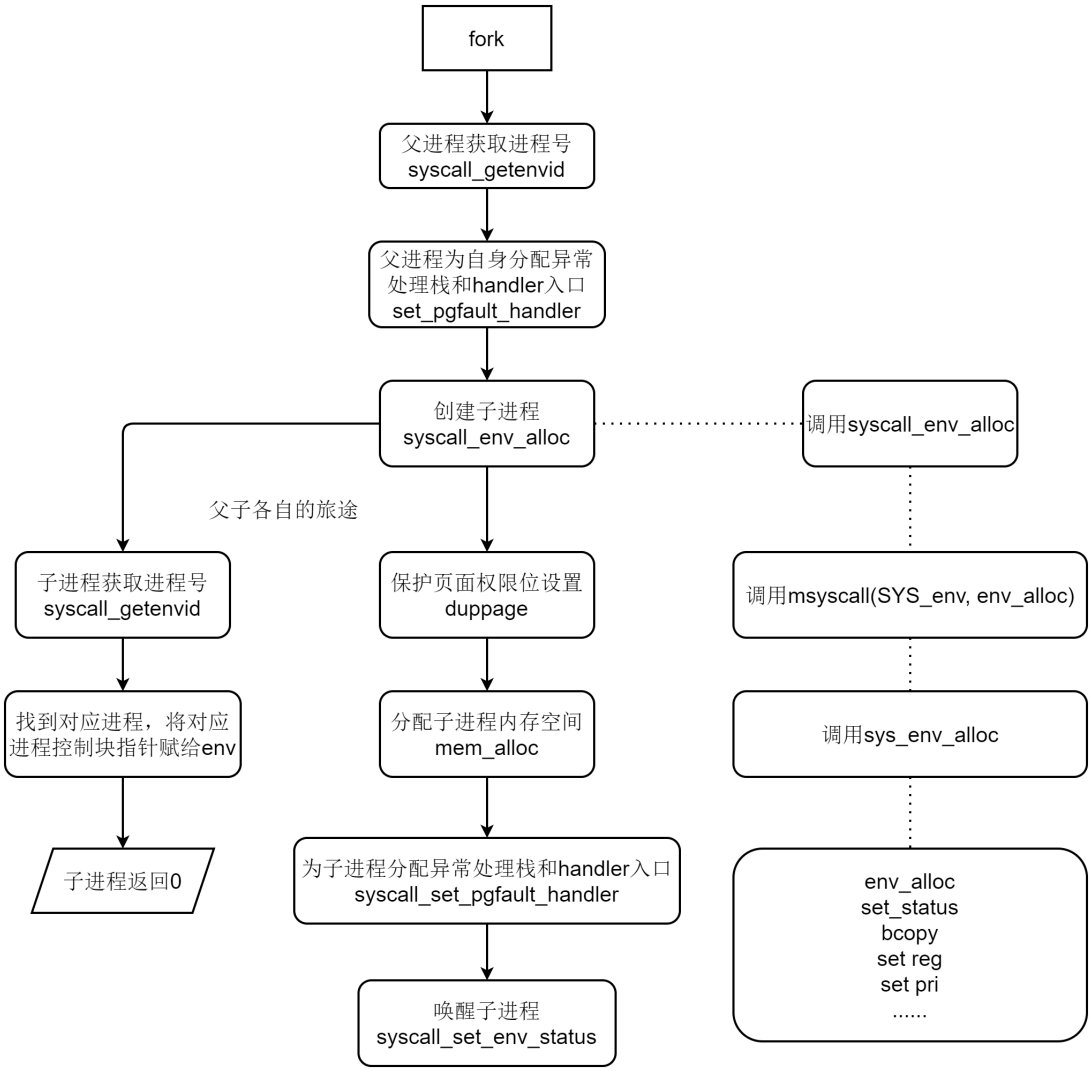
  答：不需要。父进程在之前设置过`__pgfault_handler`的值，而子进程复制了父进程中__pgfault_handler变量值，因此无需再次设置。

2、实验难点图示

- `duppage`函数



- `fork`函数



3、体会与感受

难度评价较高，实验加`debug`花费时间一周以上，实验报告约`5h`，实验量较大，但课上`Exam`难度合理，可以短时间内完成。这一部分debug的难度比实现难度大很多，因为有可能有前面lab的问题，如`env.c`（特别是`load_icode_mapper`函数）、`sched.c`等，特别建议前面添加强测，同样，也希望在`bcopy`和`bzero`的实现中可以提供字对齐。异常处理的执行流程有一点点长有一点点乱，需要细心去理清整个流程。最后，自己写汇编代码是真的痛苦……

4、指导书反馈

- 希望实验中的bzero和bcopy的可以提供字对齐；
- 因为历史遗留问题，代码部分好多注释内容没有修改（如`COW`）；