一， 实验思考题

Thinking 4.1 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？

答：说明子进程和父进程是同一个程序，代码都一样只是进程运行的 pc 开始位置不一样；没有执行之前的说明 fork 函数产生的子进程是由 fork 之后的代码开始，而且实际上是从父进程的 env_tf.cp0_epc 开始运行。

Thinking 4.2 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：C，从代码中可以看出来，子进程执行的开始位置在 epc。而父进程中断的时候是

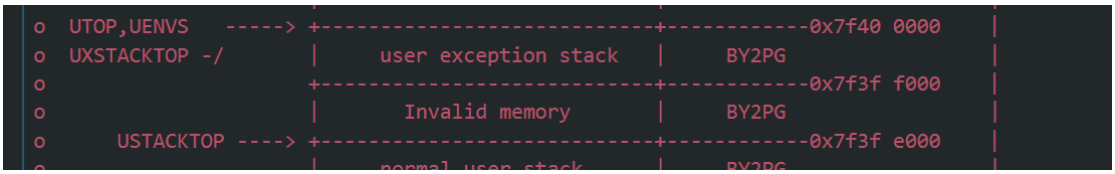
```
    bcopy(KERNEL_SP - sizeof(struct Trapframe), &e->env_tf,
        sizeof(struct Trapframe));
    e->env_tf.pc = e->env_tf.cp0_epc;
    e->env_status = ENV_NOT_RUNNABLE;
    e->env_tf.regs[2] = 0;
    printf("sys env alloc id %d\n",e->env_id);

    return e->env_id;
```

在 fork 运行过程中的 syscall_env_alloc()处中断，那么 epc 此时就会指向下一条语句也就是函数返回值赋值给 newenvid 的那一步，而我们已经将 v0 寄存器设置为 0，因此子进程中的返回值会是 0，父进程中是正常的 env_id.

```
    newenvid = syscall_env_alloc();
    if (newenvid < 0) {
        return newenvid;
    }
    if (newenvid == 0) {
        writef("%x",syscall_getenvid());
        env = &envs[ENVX(syscall_getenvid())];
        return 0;
    }
    else {
```

Thinking 4.3 如果仔细阅读上述这一段话，你应该可以发现，我们并不是对所有的用户空间页都使用 duppage 进行了保护。那么究竟哪些用户空间页可以保护，哪些不可以呢，请结合 include/mmu.h 里的内存布局图谈谈你的看法。

答：首先因为每一个进程都需要有自己的错误栈，一则 UTOP 也是 UXTOP，而另一方面，根据内存分布图来看



在 UTOP-BY2PG 的位置就是用户异常栈的位置了，因此这一页是不需要映射的，因此，映射范围只需要到如下图所示即可：

```
for (i = 0; i < UTOP/BY2PG-1; i++) {
    if ((*vpd)[i / PTE2PT] != 0 && (*vpt)[i] != 0) {
        duppage(newenvid,i);
    }
}
```

> Thinking 4.4　请结合代码与示意图，回答以下两个问题：1、vpt 和 vpd 宏的作用是什么，如何使用它们？2、它们出现的背景是什么？如果让你在 lab2 中要实现同样的功能，可以怎么写？ ■

答：

1. 首先在 mmu.h 和 entry.S 中发现了这两个变量的声明，如下图，

```
extern volatile Pte *vpt[];
extern volatile Pde *vpd[];
```

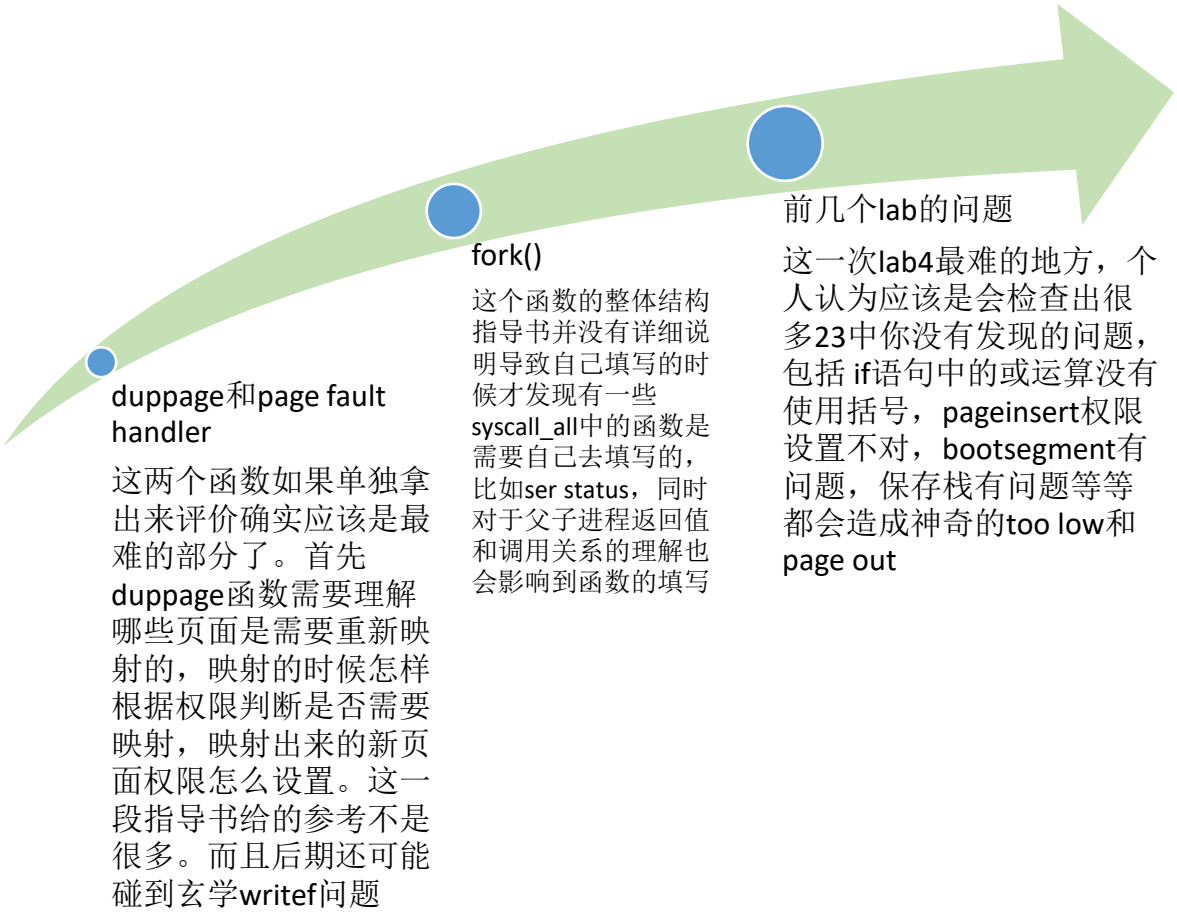
```
    .globl vpt
vpt:
    .word UVPT
    .globl vpd
vpd:
    .word (UVPT+(UVPT>>12)*4)
```

可以看到这是两个全局变量，而且分别存储了页表和页目录虚地址，因此在用户空间中，由于我们不能直接访问 pgdir 等内核中的变量，因此需要这两个全局变量来获取页目录和页表位置。由于页表和页目录本质上也就是数组，因此我们通过首地址和偏移量即可获得我们需要的页表项和页目录项，如下图：(进一步来说，va 拆分成三部分就会分别对应页目录项索引号，页表项索引号，页内偏移，然后结合 vpd 和 vpt 即可访问要映射的物理地址)

```
        for (i = 0; i < UTOP/BY2PG-1; i++) {
            if ((*vpd)[i / PTE2PT] != 0 && (*vpt)[i] != 0) {
                duppage(newenvid,i);
            }
        }
```

2. 这两个变量都出现在用户空间里，处于一个不能直接用 pgdir 访问页表项地址的背景下.而在 lab2 中可以直接使用 pgdir_walk 来实现效果，使用给定的 pgdir 和 va 获得对应的物理页面

二， 实验难点图示



三， 体会与感想

此次最深的体验来自于各种各样的花式 too low 和 pageout。

在调试过程中也逐渐明白了用户空间和内核空间的区别，至少在输出上一个调用 printf 一个是 writef，前者内核后者用户，也就是说陷入内核的操作和用户空间的操作是需要严格区分开来的。但是直接在 vim 上编辑代码难免有时候会混淆一些函数和变量，因此使用一个合适的 ide 还是很有必要的，为此我每次 lab 还是坚持将代码复制到 vs 中进行编写。

此处作业对于 fork 函数算是做到了理论和实践相结合，理解上的关卡一共有两个，一个是父子进程一次 fork 两个返回值，另外一个则是 pingpong.c 的函数调用栈到底应该长什么样。对于这两条，我都是通过打印语句输出 curenv 和相关数据对比理解的。前文已经说过了父子进程的两个返回值，这里主要讲讲 pingpong.c 的调用关系。

最后细节上来看 lab4 自身代码最后一个难点在与 duppage 和 fork 中对他的调用关系，pagefault 的实现。Duppage 重点在于理解 COW 和 library 这两个权限位对于具体处理映射关系的影响。首先对于父子进程共享的页面，如果这个页面标记了 COW 保护而且没有要求共享内存的话说明这可能会是两个不同的页面因此需要重新都标记上 COW

```
if ((perm & PTE_R) != 0 || (perm & PTE_COW) != 0) {
    if ((perm & PTE_LIBRARY) != 0) {
```

但是如果共享内存，则意味着两者会经过同一页面共享数据，这时 COW 反而会阻碍通信，故而不能加这个权限。对于 page_fault 来说，重点在于找到这个临时存储的位置，这里我使用了 mmu.h 里标注的 invalid space 中的位置——BY2PG，同时要记得重新映射之后需要将临时页的映射关系从进程页表中删除

```
pgfault(u_int va)
{
    u_int *tmp;
    //  writef("fork.c:pgfault():\t va:%x\n",va);

    if ((*vpt)[VPN(va)] & PTE_COW == 0) {
        user_panic("fork.c:pgfault()1:\t va:%x\n", va);
    }
    //map the new page at a temporary place
    if (syscall_mem_alloc(0, BY2PG, PTE_V | PTE_R) < 0) {
        user_panic("fork.c:pgfault()2:\t va:%x\n", va);
    }
    //copy the content
    tmp = (u_int *)ROUNDDOWN(va, BY2PG);
    user_bcopy(tmp, BY2PG, BY2PG);
    //map the page on the appropriate place
    if (syscall_mem_map(0, BY2PG, 0, va, PTE_V|PTE_R) < 0) {
        user_panic("fork.c:pgfault()3:\t va:%x\n", va);
    }
    //unmap the temporary place
    if (syscall_mem_unmap(0, BY2PG) < 0) {
        user_panic("fork.c:pgfault()4:\t va:%x\n", va);
    }
}
```

最后，这次调试过程中我收获最大的就是学会了合理使用 user_panic 函数，可以大大简化调试的过程，只要出现问题，就可以 panic 获取错误信息。

四， 指导书反馈

1. 此次指导书我认为在 duppage 的地方需要讲的 ***更加详细*** ，尤其是 fork 调用的时候，怎么映射过去，需要检查页目录和页表项有效性，这些细节是为什么做的，值得进一步考虑。还有共享内存部分的处理也没有提及。
2. Syscall 的函数中有两个并没有在指导书中提到，但是需要填写的函数，有很多同学一开始觉得就填写完了，调试的时候出了问题总是不知道怎么回事，问别人才知道是少写了函数。我觉得这样的问题应该通过指导书 ***明确的任务纲要*** 来解决，而不是通过同学们“极佳”的猜谜能力来知道哪些部分需要写。
3. 我认为 fork 部分的逻辑顺序需要调整，应该是从整体框架要干什么理解之后，才能去深入细节填写 duppage 和 fault，但是现在从细节开始向上层抽象，过程中一下子

少了很多细节，理解上造成了较大麻烦。

4.
```
*
* Hint: the only function you need to call is envid2env.
*/
int sys_ipc_can_send(int sysno, u_int envid, u_int value, u_int srcva,
    u_int perm)
{
```
Ipc 这个函数不得不说，注释里说只需要调用一个 envid2env，结果呢，我们需要 lookup 和 insert，我觉得类似的误导性注释还是需要改一下，以尽量避免加深大家本身就已经很迷惑的学习状态。

五， *残留难点*

1. Round 函数在 bcopy 中的使用，直接复制 va 不行吗？
2. FRUP 是什么？
3. Perm 到底应该怎么判断什么时候需要什么权限？