Lab4实验报告

思考题

T1

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？***

当发生异常时，执行异常处理程序会执行`SAVE_ALL`宏，该宏调用了`get_sp`宏，将`sp`寄存器的值置为`KERNEL_SP`中的值（`lw sp, KERNEL_SP`）。之后将所有通用寄存器的值和所有CP0寄存器（除了CP0.PC）保存到`sp`的栈中，注意此时`sp`的栈是在内核空间中 。

- ***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？***

能，因为`msyscall`函数调用时，寄存器$a0-$a3用于存放前四个参数。执行`syscall`并没有改变这四个寄存器。但是在内核态下，之后可能这四个寄存器会被操作为其他值，故 再次需要使用`msyscall`函数参数时需从栈`sp`按相应偏移量取出这四个寄存器的值。

- ***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？***

调用`msyscall`函数传入的参数之后会被保存在栈`sp`中，因此将前四个参数从栈中取出并存入寄存器$a0~$a3, 后两个参数保存到栈`sp`的偏移量为16和20的位置，然后调用`sys_*`就会让它"认为"我们提供了同样的参数。

- ***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？***

将Trapframe中epc的值加4，使返回用户态时pc指向系统调用指令的下一条指令。

```
lw t0, TF_EPC(sp)
addiu t0, t0, 4
sw t0, TF_EPC(sp)
```

T2

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。

函数`envid2env(u_int envid, struct Env **penv, int checkperm)`根据`envid`找到对应的进程，如果`envid`为0，则对应当前进程。

系统调用的诸多函数如`sys_mem_alloc`也都需要envid寻找要操作的进程，它们都调用了`envid2env`函数。

因此，如果要实现给定的`envid`为0时，对应当前进程，我们需要保证没有一个进程的`envid`为0。因此，`mkenvid()`函数不会返回0。

T3

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？***

子进程的代码段与父进程相同

- ***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？***

子进程的进程上下文状态与父进程相同，fork()结束后子进程执行fork()的下一条 指令。

T4

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

正确答案为C

T5

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。

应该映射的页面是从虚拟地址`UTEXT`到`USTACKTOP`之间的页面，因为往高是异常栈和无效内存区域，不需要映射。再往高`UTOP`以上的用户空间，`env_setup_vm`函数已经做过映射了，而且按理说这块区域所有的进程都应该映射到相同内存，包括`ENVS`，`PAGES`，`VPT`。

T6

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？***

```
//user/entry.S
    .globl vpt//建立一个全局变量vpt
vpt:
    .word UVPT//vpt是一个地址，指向UVPT，即二级页表基地址

    .globl vpd
vpd:
    .word (UVPT+(UVPT>>12)*4)//vpd指向页目录基地址
```

由注释可知，**vpt**是一个变量，值为地址UVPT，**vpt**也就是一个指向UVPT的指针，指向用户页表虚拟基地址；**vpd**是一个指向`(UVPT+(UVPT>>12)*4)`的指针，由页目录自映射可知这是用户页目录的虚拟基地址。

使用时，给出虚拟地址**va**

则页目录项虚拟地址为`(Pde *)(*vpd) + (va >> 22)`，页目录项内容为`(*vpd)[va>>22]`；

页表项为`(Pte *)(*vpt) + (va >> 12)`，二级页表项内容为`(*vpt)[va>>12]`

- ***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?***

在用户态，**UVPT**是用户页表的虚拟地址，而由上述代码知，**vpt**就指向这个地址；由于已经建立了页目录自映射机制，`(UVPT+(UVPT>>12)*4)`就是用户页目录的虚拟基地址，因此通过**vpd**就可以访问页目录。

- ***它们是如何体现自映射设计的?***

`*vpd = (UVPT+(UVPT>>12)*4)`，说明地址空间中页目录处于页表的某个指定位置，实现了自映射。

- ***进程能够通过这种方式来修改自己的页表项吗?***

不能，用户态不能修改页表项，因为在创建进程时，`env_setup_vm`函数设定了`e->env_pgdir[PDX(UVPT)] = e->env_cr3 | PTE_V;`，并没有对页表所在虚拟页设置PTE_R位，因此用户进程不能修改。

T7

`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"?***

当发生页写入异常时，用户态正执行页写入异常处理函数时，发生时钟中断。

- ***内核为什么需要将异常的现场Trapframe复制到用户空间?***

因为真正处理页写入异常的处理 函数是，它是用户态下实现的，为了实现微内核的目的。因此需要将异常现场复制到用户空间，以便页写入异常完全处理结束后，进程可以恢复回最初状态。

T8

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?***

多个进程并发执行的时候，让用户进程分担操作系统内核的任务，可以减少操作系统内核工作的时间，可以提高并发效率。实现了微内核思想，也使得即使处理异常时发生崩溃，也不会影响到整个系统的稳定

- ***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?***

已经恢复的寄存器在中断前后值一定保持不变（中断处理机制保证的），未恢复的寄存器的值保存在栈中（这里是异常处理栈），中断返回后能找到栈指针即可，而中断后恢复进程上下文是会恢复栈指针的。

T9

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?***

这样做子进程从`syscall_env_alloc`函数返回之后不会调用这个函数。

- ***如果放置在写时复制保护机制完成之后会有怎样的效果?***

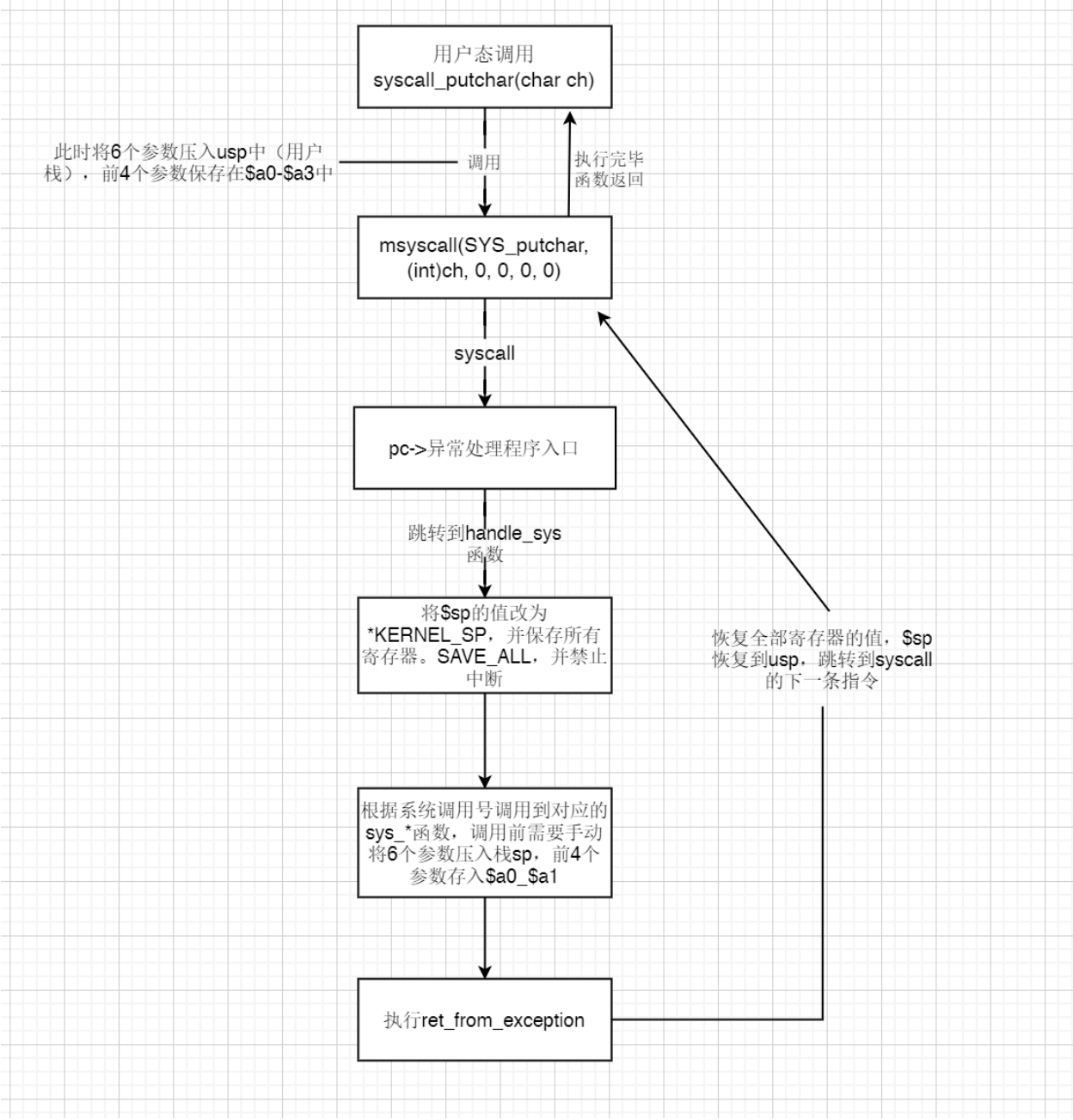
如果写时复制保护机制还未完成，父进程在函数调用的过程中操作了堆栈，则会发生页写入异常，但由于没有设置页写入异常处理函数，无法正确处理。

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

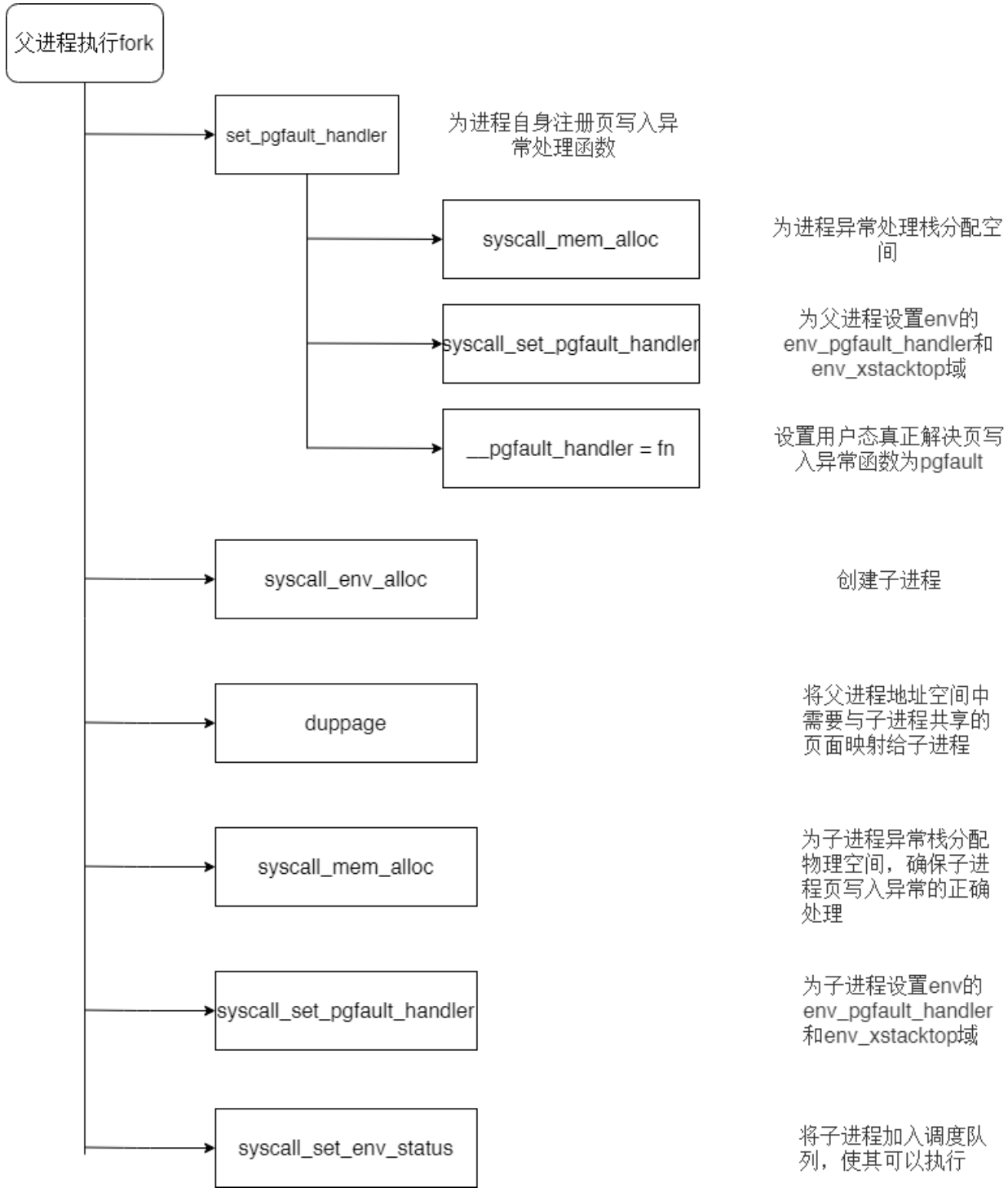
不需要，`__pgfault_handler`是在entry.S中定义，有`.data`标识，表示这个变量在数据段定义。由于父进程和子进程的数据段都一致，因此子进程的`__pgfault_handler`和父进程的值相同。

实验难点

1. 系统调用的过程，以syscall_putchar为例

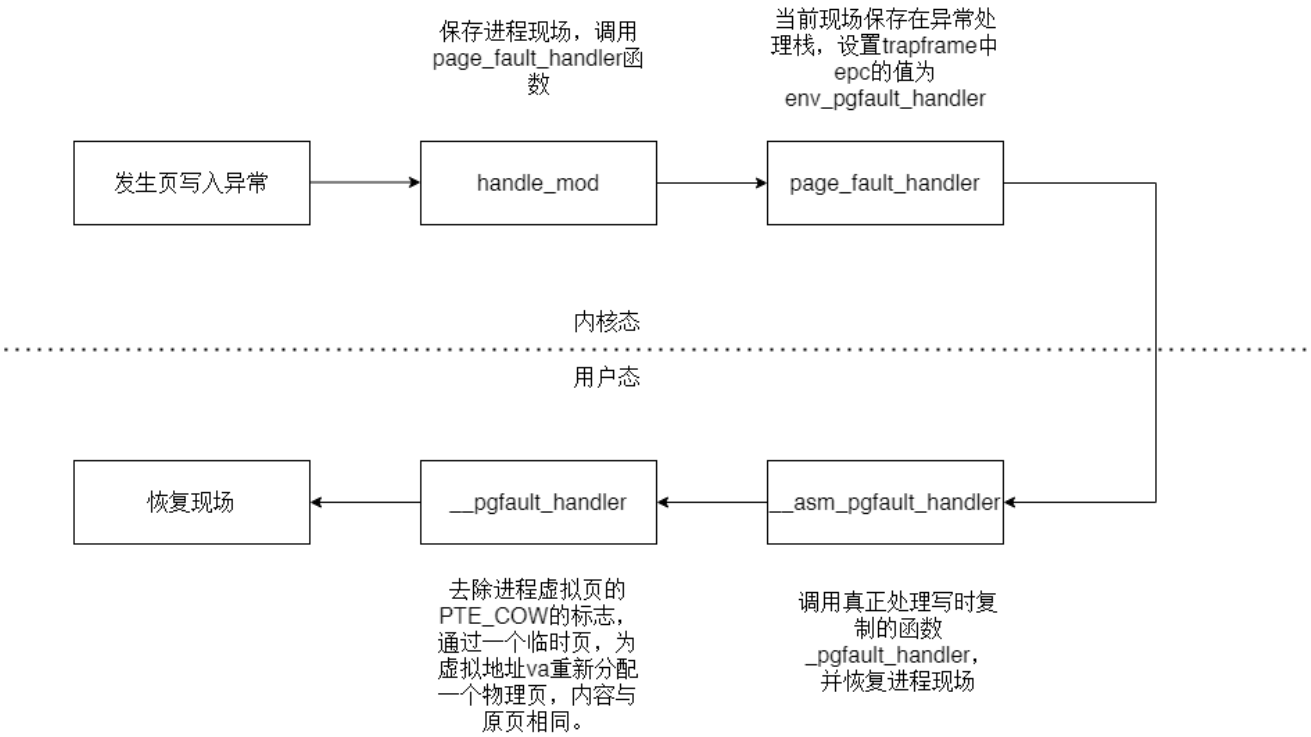


2. 页写入异常设置流程，子进程和父进程都是在fork函数中设置。



3. 页写入异常的处理过程

CPU 的**页写入异常**会在用户进程写入被标记为 PTE_COW 的页面时产生。经过异常分发进入异常处理函数 `handle_mod` 中。



体会与感想

Lab4实验的fork函数感觉是本学期最难的一部分，做的过程中脑子很乱，做完之后再次整理才明白各函数究竟做了什么。但是助教王廉杰非常热心，提出的所有问题都会积极回答，以下的实验疑问如果没有王助教无法解决。

实验疑问

本次实验过程中，实验疑问非常多，但是目前都已解决。

1. 为什么exersize4.0让`env_tf.cp0_status`*的值为* `0x1000100c`呢?

   在gxemul中，KU_c表示是否处于用户态，1表示处于用户态，这和之前学习的不太一样。

2. 处理系统调用时的内核仍然是代表当前进程的，如何理解?

   在内核处理系统调用时，并没有切换 CPU 的地址空间（页目录地址），最后也没有将进程上下文（`Trapframe`）保存到进程控制块`env_tf`中，只是切换到内核态下， 执行了一些内核代码。执行完后也没有进行`sched_yield()`进行进程的切换，而是通过`ret_from_exception`返回，并没有执行`env_run()`函数。

   而发生时钟中断时，切换进程会执行`env_run()`函数，其会执行`lcontext()`函数切换到新进程的页目录地址，而且会将进程上下文保存到旧进程控制块中，再切换到新进程。

3. user/libos.c 的实现中，用户程序在运行时入口会将一个用户空间中的指针变量 `struct Env *env` 指向当前进程的控制块，如何理解?

   `user/user.lds`指明了用户进程的入口地址为`_start`，`lds`会把用户程序里的`_start`链接到 UTEXT

   注意，`entry_point`就是`UTEXT`，值为`0x400000`。

   - **_start**，用户进程初始从这里开始执行

```
#user/entry.S 用户态
    .text
    .globl _start
_start:
    lw  a0, 0(sp)#$sp 此时为 USTACKTOP，这两句话没用，lab6才有用
    lw  a1, 4(sp)
    jal libmain
    nop
```

- **libmain**函数，用户进程入口的 C 语言部分，负责完成执行用户程序 umain 前后 的准备和清理工作，是我们这次需要了解的函数之一。

```
//user/libos.c
void exit(void)
{
    syscall_env_destroy(0);
}

struct Env *env;

void libmain(int argc, char **argv)
{
    // set env to point at our env structure in envs[].
    env = 0;    // Your code here.
    //writef("xxxxxxxxx %x  %x  xxxxxxxxx\n",argc,(int)argv);
    int envid;
    envid = syscall_getenvid();
    envid = ENVX(envid);
    env = &envs[envid];
    // call user main routine
    umain(argc, argv);//用户程序开始执行，在本lab里它是一个测试函数
    // exit gracefully
    exit();
}
```

这里需要一个`env`变量的原因是有些函数需要`env`，如`ipc.c`。

4. **note 4.6**是什么意思

> **Note 4.6**. *在用户态实现的fork并不是一个原子的过程，所以会出现一段时间（也就是在duppage之前的时间）我们没有来得及为堆栈所在的页面 设置写时复制的保护机制，在这一段时间内对堆栈的修改（比如发生了其他的函数调用），会将非叶函数syscall_env_alloc 函数调用的栈帧中的返回地址覆盖。这一问题对于父进程来说是理所当然的，然而对于子进程来说，这个覆盖导致 的后果则是在从syscall_env_alloc返回时跳转到一个不可预知的位置造成panic。当然你现在看到的代码已经通过一个 优雅的办法来修补这个 bug：与其他系统调用函数不同，syscall_env_alloc是一个内联（inline）的函数，也就是说 这个函数并不会被编译为一个函数，而是直接内联展开在fork函数内。所以syscall_env_alloc的栈帧就不存在了，msyscall 函数直接返回到了fork函数内，如此这个 bug 就解决了。*

在堆栈所在页面设置PTE_COW之前，在`for`和`if`之间、`duppage`函数调用的过程中，可能会对堆栈进行操作，而此时由于父进程已经从`syscall_env_alloc`返回，故对应的调用栈帧也就销毁了，很可能其保存的函数返回地址被覆盖。因此把`syscall_env_alloc`函数设置成内联函数。

```
    newenvid = syscall_env_alloc();////子进程在此返回0，父进程在此返回子进程id
    if(newenvid == 0){
        //若为子进程返回,注意，开始执行这段代码时，父进程
        newenvid = syscall_getenvid();//当前进程为子进程，获取子进程的id
        env = &envs[ENVX(newenvid)];
        return 0;
    }
    //将父进程地址空间中需要与子进程共享的页面映射给子进程
    for(i=0;i<VPN(USTACKTOP);i++){
        if(((*vpd)[i>>10] & PTE_V) && ((*vpt)[i] & PTE_V)){
            duppage(newenvid, i); // i is virtual page number, newenvid is the id of
son
        }
    }
```

5. `duppage`函数为什么必须保证子进程先映射PTE_COW，之后父进程才能映射?

   因为执行`duppage`时，仍处于用户态，如果先为父进程页面置PTE_COW，那么如果这个页面是堆栈的话，执行之后的`syscall_mem_map`函数可能会改变这个堆栈产生页写入异常，进而父进程该页面消除PTE_COW标记，而子进程会有`PTE_COW`标记。

   这就导致如果父进程之后修改 了这个 页面，那么不会产生页写入异常，子进程会同时修改，产生问题。

   另外，经过实验验证，如果除了堆栈以外页面的映射都是父进程在前，子进程在后，是正确的。

```
//duppage函数...
    if((perm & PTE_R) && !(perm & PTE_LIBRARY)){
        perm = perm | PTE_COW;
        //flag = 1;
        if(addr != USTACKTOP - BY2PG){
            syscall_mem_map(0, addr, 0, addr, perm);
            syscall_mem_map(0, addr, envid, addr, perm);
        }
        else {
            syscall_mem_map(0, addr, envid, addr, perm);
            syscall_mem_map(0, addr, 0, addr, perm);
        }
    }
    else {
        syscall_mem_map(0, addr, envid, addr, perm);
    }
```

6. `__pgfault_handler = fn;//如果 执行过一次fork了，别的进程的__pgfault_handler的值为多少`

   注意`__pgfault_handler`是在entry.S中定义，有`.data`标识，表示这个变量在数据段定义，相当于用户程序里的全局变量。由于每个进程是独立的，因此这个变量不是每个进程共有的，而是不同进程有着不同的变量值。