林子杰_20373980_lab4

一、思考题

Thinking 4.1

- 保存现场时，内核会使用一个汇编宏函数`SAVE_ALL`，将相关通用寄存器的值保存到栈帧中。
- 可以，寄存器的值未被改动。
- 通过寄存器传递了四个参数，通过栈帧传递了了两个参数至函数的位置。
- 修改了`Trapframe`中`EPC`的值为下一条指令的地址，确保指令执行正确。

Thinking 4.2

0在MOS中用于表示当前进程，是为`curenv`专门保留的一个位置。在**系统调用**和**IPC**两部分中，很多函数都传入了`envid`；除此之外，我们要注意到，在我们使用系统调用时，很多情况下都是在一个进程和内核之间进行往返。为了便于识别进程，保留一个0用于表示当前进程是很方便的。在`envid2env()`函数中，识别到`envid`为0就直接得到`curenv`，毋需再从数组中存取。

Thinking 4.3

子进程仅执行了fork函数后的代码，没有执行fork函数前的代码，可以猜测子进程可能和父进程共享代码段，也可以猜测子进程在被系统调用创建后，系统调用令子进程回到了fork函数结束的位置，继续执行代码。

Thinking 4.4

关于 fork 函数的两个返回值，下面说法正确的是：

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

在这一部分中，我们有很多函数需要填写。我会首先独立地介绍每个函数的功能，随后将会归纳总结他们之间的调用关系。

Thinking 4.5

用户空间的大部分区域都需要进行映射，内核空间则不需要。对于每个分区，我们进行如下分析：

- `UTOP`到`ULIM`的区域存储了`envs`数组、`pages`数组等关键信息，这一部分可以被访问，但是不可被修改，其已经被保护好，不需要在再进行保护。
- `USTACKTOP`到`UTOP`的区域为`Invalid memory`和用户异常栈，不能设置写时保护。

`USTACKTOP`以下的页面都是需要进行写时保护的。

Thinking 4.6

vpt为二级页表的起始地址（所有的二级页表的起始），vpd为一级页表的起始地址。我们来一步一步地追溯它们的本源。

首先，我们可以在`mmu.h`中找到他们被定义为数组的地方:

```
typedef u_long Pde;
typedef u_long Pte;

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

可见，这两个变量的本质都是`u_long`类型的指针，指向改类型的数组。那么他们的值在哪里被定义呢？我们来到`entry.S`中，发现这个汇编文件中，定义了`vpt`和`vpd`这两个宏。其中，`vpt`为一个字，其中填充了`UVPT`；而`vpd`则要复杂一些，它所对应的字中填充了`(UVPT+(UVPT>>12)*4)`。咋一看是否有点眼熟？这时候我们会发现，兜兜转转又回到了`mmu.h`中，我们可以在内存地图上看到`UVPT`的地址，就是`0x7fc00000`，也就是二级页表的起始地址。到这里，这两个变量的意义就已经明晰了。

由于这两者根据其存放在内存中的位置实现了内存自映射机制，故而我们的进程可以通过访问内存来访问一级页表，进而访问二级页表，进而访问其自身的整个页表。

二级页表的起始位置在`0x7fc00000`，这个位置往上4KB的内存空间对应的是`(UVPT>>12)`的虚拟页面。要实现自映射机制，一级页表的起始位置应当在二级页表中，且我们知道一级页表中的第一个32位数应当能拿取到第一个二级页表所在页的虚拟地址，而二级页表的第一个32位数则能取到所有页中的第一个页，故而一级页表的第一个32位数应当相对`UVPT`发生偏移，需要偏移`(UVPT>>12)`个bit即`(UVPT>>12)*4)`个字节。

用户进程无法修改页表项。

Thinking 4.7

首先，我查询了一下"中断重入"这一概念的意思，得到的结论大致是：中断的时候发生了中断。我大致讲一下我的理解。对于我们的MOS操作系统来说，页写入异常和缺页异常是两种不同的TLB异常。当我们想从TLB中调取页面但是发生中断的时候，我们就会进入内存或者磁盘寻找页面并且将其调入TLB。倘若我们遇到了一种更糟糕的情况：进程出发了页写入异常，我们不得不进入中断并且为其分配页面；然而我们分配的页面不在TLB中，上一个中断却还没有服务完，操作系统不得不再进入一层中断。这应该是可能发生"中断重入"的原因，归纳地来讲，就是连续的重叠的中断所致。

MOS实现的是微内核，它将缺页异常的主要部分置于用户态下处理。用户需要依靠Trapframe的信息执行中断处理和恢复现场，故而我们将其保存在用户的"异常处理栈"中

Thinking 4.8

在用户态处理页写入异常符合微内核的设计理念，能够精简内核大小，同时也使该异常处理起来更加方便，性能更好。

第二问不是很理解题目的意思。从寄存器的使用方式来讲，在用户态可能发生指令跳转的情况时，将寄存器的内容依照栈指针寄存器sp的目标依次压入栈中，最后再跳转；恢复现场时，先取出sp寄存器的值，并依照其重定位恢复寄存器的值，最后再跳转。这样使用可以保证栈指针的安全性，从而使得通用寄存器在栈中有安全的备份。标准操作可参见`entry.S`中的汇编函数。

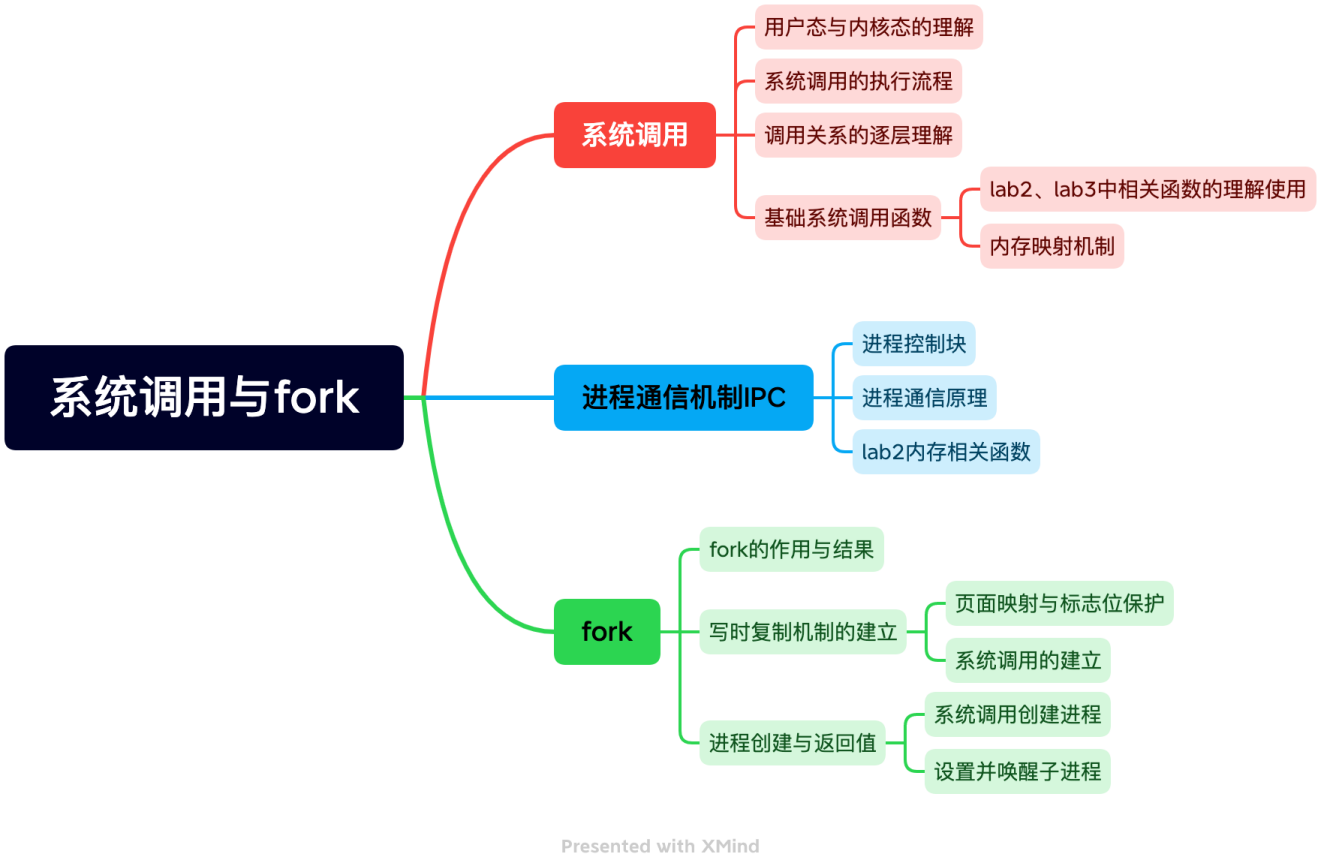
Thinking 4.9

父进程在调用系统调用创建子进程的时候，可能会提前出现缺页中断，需要设置缺页处理条件。

父进程会给`__pgfault_handler`变量赋值时，会触发缺页中断。由于没有提前配置中断处理，无法处理这样的缺页中断。

不需要，子进程与父进程共享这个值。

二、实验难点



三、体会与感想

lab4的理解难度相对于前面几个lab更加高，~~当然好像我写一次报告都会写类似的话~~。认真分析，其难度增高主要有以下两个原因：

- 要求综合理解lab2、lab3的相关知识，运用lab2、lab3的相关函数，综合程度很高
- C语言和MIPS汇编语言穿插交替出现，理解难度大

其中，理解参数传递、中断和系统调用的本质对于完成实验会有很大的帮助。如果有时间，回头温习一下内存映射机制部分的内容，对完成本部分的实验将会很有帮助。但在使用之前的写好工具时，也无需过多纠结其实现原理，以提高效率。

另外，个人感觉fork部分指导书的组织有些混乱，阅读起来感觉不是很顺畅。希望能够寻找更好的组织方式。

下面附上本次的实验笔记（非最终版）。

Lab4实验笔记

O、前记

lab4的主要内容是**系统调用与fork**。在此之前大致浏览了一下本次实验指导书的内容，应当是以文字阅读和理解为主要内容，故而本次笔记也会有一些这方面的特色。

理论上来讲，硬件操作、动态内存分配等操作是被内核认定为比较"危险"的行为，这一类行为放任用户使用可能会造成不可预知的后果，故而它们只能交给内核执行；然而有些用户进程又不可避免地要使用这些操作。故而操作系统会给用户提供"接口"，使得用户能够以安全的方式调用这些内核功能。这就是**系统调用**的初级理解。

借用指导书内容，本次实验的主要任务如下：

- 掌握系统调用的概念及流程
- 实现进程间的通讯机制
- 实现进程创建机制`fork()`函数
- 掌握页写入异常的处理流程

一、系统调用

1.概念回忆

首先进行概念的整理，便于后面的学习解释。

概念名词	解释
内核态/用户态	CPU运行的两种模式，拥有不同级别的权限。该状态由 CP0协处理器的SR 寄存器中KUc位的值标志
内核空间/用户空间	进程的虚拟地址的两部分（在同一进程中存在）。在虚拟地址映射中，用户空间的虚拟页通过页表映射到物理页，内核空间的虚拟页则映射到固定的物理页和外部设备。CPU在内核态下，才可以访问进程的内核空间。
进程/内核	进程是资源分配和调度的基本单位，内核负责管理和分配系统资源。内核的调度功能决定了它可以和进程共存。

在lab3中，我们的进程是运行在内核态下的。为了使进程运行在用户态下，我们需要通过某种办法修改CP0中SR寄存器的值（还记得吗？SR低六位寄存器是作为一个二重栈使用的）。我们通过修改`Trapframe`结构体中的值保证进入中断时写入该寄存器的值正确。`exercise 4.0`就是要求在创建进程时向`env_tf.cp0_status`写入0x1000100c。

2.系统调用溯源

在指导书中，举了一个调用`puts()`函数的例子，来为我们揭示了系统调用的本源。我们知道，标准IO操作必须在中断下进行（也就是内核态），这一过程中就使用到了系统调用。指导书通过一系列汇编、反汇编以及调试操作实现了溯源过程，整理出调用`puts()`函数发挥作用的全过程：

- 调用`puts()`的下层函数`write()`
- `write()`函数为寄存器设定了相应的值，并执行`syscall`
- 系统进入内核态，根据设定的值运行发挥作用
- 返回`write()`函数，取出寄存器值，继续返回直至`puts()`函数

这其中有些操作，和我们在组成原理的P7部分完成的工作很是相似（当然，有的同学没有接触P7，指导书需要照顾这些同学所以要写的详细）。我们可以通过这一过程了解到：

- IO操作需要内核来完成，即需要系统进入内核态（当然，不仅仅是IO操作）
- syscall指令可以使系统陷入内核态（在Mars中我们都是用过这个指令）
- 系统状态切换时，需要进行数据保护

当今，用户已经很少直接使用系统调用了。很多语言都已经帮我们封装好了一些底层的操作，~~不要重复造轮子~~。

3.实现系统调用

系统调用也是一种中断，我们在lab3中已经介绍过中断处理流程。异常向量组分发的8号异常就是专门处理这一中断的。指导书指出，我们将在`./user/printf.c`函数中学习相关流程。

```
static void user_myoutput(void *arg, const char *s, int l)
{
  int i;

  // special termination call
  if ((l == 1) && (s[0] == '\0')) {
    return;
  }

  for (i = 0; i < l; i++) {
    syscall_putchar(s[i]);

    if (s[i] == '\n') {
      syscall_putchar('\n');
    }
  }
}

void writef(char *fmt, ...)
{
  va_list ap;
  va_start(ap, fmt);
  user_lp_Print(user_myoutput, 0, fmt, ap);
  va_end(ap);
```

```
}
```

其中的`user_lp_Print()`函数用于输出字符串，它调用了`user_myoutput()`函数；`use_myoutput()`函数又调用了`syscall_putchar()`函数；`syscall_putchar()`函数又调用了`msyscall()`。如果你使用ctag，你就会发现，到此就是调用的终点了，因为`msyscall`就是一个汇编函数，系统在此进入了内核态。结束函数的执行后，会一层层向上返回。

如果你细心的话，你会发现在`syscall_putchar()`同文件下，还有很多长相相似的函数，如`syscall_yield()`、`syscall_env_destory()`等。这些函数都会调用`msyscall()`函数，由他们的名字可以知道它们是何种中断触发的系统调用。

`msyscall()`传入了六个参数，第一个通常是和本函数对应中断问题相关的宏，学名为系统调用号。剩余的参数随着功能而改变。`msyscall()`传递参数的方法，和组成原理课程中的栈帧操作一样：函数调用时，将当前层函数的内容压栈（通过移动栈指针实现），供下一层函数使用；返回时则推栈（也是移动栈指针）。在`msyscall()`中，我们有六个参数，其中四个可以通过寄存器堆约定俗成的`$a0~$a4`寄存器传递，剩下的要通过栈来传递；不过，在压栈时，我们也会为前四个参数预留栈空间（不写入）。

上面都是一些原理性的内容，我们接下来需要在`exercise4.1`中完成`msyscall()`函数。这样的简单操作，我们在上学期已经做的够多了。

```
LEAF(msyscall)
    syscall
    jr  ra
    nop
END(msyscall)
```

可以看到汇编语言中的`syscall`指令，此时系统便陷入内核态了。接下来，我们要实现一个`handle_sys`汇编函数。我们上次实现了一个`handle_int`函数用于处理时钟中断，这个函数则是用于处理系统调用中断。这一函数中，我第一次填写的时候尚且有很多地方没能完全弄明白，故而先摘取指导书中的一些解释，后续还会补充我的理解。

`syscall`使我们陷入了内核态，在进入内核态之前，我们需要保存当前函数的部分"运行现场"（要注意，陷入内核态并不是函数跳转）。

```
NESTED(handle_sys,TF_SIZE, sp)
    SAVE_ALL                                /* 用于保存所有寄存器的汇编宏 */
    CLI                                     /* 用于屏蔽中断位的设置的汇编宏 */
    nop
    .set at                                 /* 恢复$at寄存器的使用 */
  /* 取出Trapframe的EPC寄存器的值，将其修改为下一条指令的值并写回。由于之前Trapframe结构体已经使用汇编宏保存，此处也只需要使用汇编指令存取 */
  lw    t0, TF_EPC(sp)
  addiu t0, t0, 4
  sw    t0, TF_EPC(sp)
  /* 将系统调用号存入a0寄存器 */
  lw    a0, TF_REG4(sp)
    addiu   a0, a0, -__SYSCALL_BASE     /* a0 <- “相对”系统调用号 */
    sll     t0, a0, 2                   /* t0 <- 相对系统调用号 * 4 */
    la      t1, sys_call_table          /* t1 <- 系统调用函数的入口表基地址 */
```

```
    addu    t1, t1, t0                  /* t1 <- 特定系统调用函数入口表项地址 */
    lw      t2, 0(t1)                   /* t2 <- 特定系统调用函数入口函数地址 */
    /* 经过上面的一系列操作，成功将系统调用号对应的函数地址放入了t2寄存器 */
    lw      t0, TF_REG29(sp)            /* t0 <- 用户态的栈指针 */
    lw      t3, 16(t0)                  /* t3 <- msyscall的第5个参数 */
    lw      t4, 20(t0)                  /* t4 <- msyscall的第6个参数 */
    /* 使用栈指针为六个参数分配空间，并将参数写入正确的位置。最后两个参数只能使用内存传递值；其余四个参数可以使用a组寄存器传值 */
    lw    a0, TF_REG4(sp)
    lw    a1, TF_REG5(sp)
    lw    a2, TF_REG6(sp)
    lw    a3, TF_REG7(sp)
    /* 栈向下增长，并且将第五个和第六个参数写入寄存器，传入下一层函数；其余的参数通过参数寄存器传递 */
    addiu sp, sp, -24
    sw    t3, 16(sp)
    sw    t4, 20(sp)

    jalr    t2                          // Invoke sys_* function
    nop

    /* 恢复栈指针 */
    addiu sp, sp, 24
    sw      v0, TF_REG2(sp)             /* 将$v0中的sys_*函数返回值存入Trapframe */
    j       ret_from_exception          /* 从异常中返回（恢复现场） */
    nop
END(handle_sys)
```

5.15补充：今天回顾了一下系统调用机制的部分，重新梳理了系统调用机制的实现过程。即将进行系统调用的时候，调用顺序是这样的：`syscall_*()`、`msyscall()`、`syscall`、`handle_sys()`、`sys_*()`。`syscall_*()`是一个带有六个参数的函数，第一个参数是系统调用号；`msyscall`是一个汇编函数，里面仅仅有两条有效指令即`syscall`和`jr`，`handle_sys()`则是`syscall`后进入的处理函数。指导书中有一处写道"而且函数的第一个参数都是一个与调用名相似的宏……把这个参数称为**系统调用号**……系统调用号是内核区分这究竟是何种系统调用的唯一依据。"第一次阅读指导书的时候，我对此处感到困惑，因为`msyscall()`这一汇编函数根本不带任何参数，但是`sysccall_*()`函数却传了六个参数，这是怎么做到的？今天再仔细想了一下，突然发现自己之前将C语言和汇编语言割裂了，C语言最终还是会被编译为汇编语言执行，那么就可以用汇编语言的参数传递方式来理解此处的参数传递了，即使用`$a`系列寄存器和栈帧传递。理解了这一部分，我在上面提到的"不能完全明白"的一些问题也迎刃而解了。`handle_sys()`中，由于内核已经提前向栈中保存了Trapframe，故而有些值可以用栈指针寄存器方寸。连续向`$a`中写入四个值同时`sw`压栈两次，是为了准备下一次跳转的`sys_*()`函数提供参数，这样系统调用机制函数之间的关系也更明晰了。

Thinking 4.1

- 保存现场时，内核会使用一个汇编宏函数`SAVE_ALL`，将相关通用寄存器的值保存到栈帧中。
- 可以，寄存器的值未被改动。
- 通过寄存器传递了四个参数，通过栈帧传递了了两个参数至函数的位置。
- 修改了`Trapframe`中`EPC`的值为下一条指令的地址，确保指令执行正确。

4.3系统调用函数

我们已经能够调用相关的系统调用函数了。接下来，我们将补充几个系统调用函数，以完善系统调用机制。

首先是`sys_mem_alloc()`函数。这个函数将会为进程号为`envid`的进程分配一页空间。

```
int sys_mem_alloc(int sysno, u_int envid, u_int va, u_int perm)
{
    // Your code here.
    struct Env *env;
    struct Page *ppage;
    int ret;
    ret = 0;
    /* 判断权限与va是否合法 */
    if(perm & PTE_COW) return -E_INVAL;
    if(((perm & PTE_V) == 0) || (va >= UTOP)) return -E_INVAL;
    /* 安全地声明一个页面 */
    if((ret = page_alloc(&ppage)) < 0) return ret;
    /* 安全地得到envid对应的进程 */
    if((ret = envid2env(envid, &env, 0)) < 0) return ret;
    /* 安全地将该页面插入到进程的页目录中 */
    if((ret = page_insert(env->env_pgdir, ppage, va ,perm)) < 0) return ret;
    return ret;
}
```

然后是`sys_mem_map()`函数，用于将一个进程的某个虚拟地址对应的页面和另一个进程某个虚拟地址对应的页面关联起来，即共享同一个物理页面。

```
int sys_mem_map(int sysno, u_int srcid, u_int srcva, u_int dstid, u_int dstva,
                u_int perm)
{
    int ret;
    u_int round_srcva, round_dstva;
    struct Env *srcenv;
    struct Env *dstenv;
    struct Page *ppage;
    Pte *ppte;

    ppage = NULL;
    ret = 0;
    /* 向下取整，保证对齐 */
    round_srcva = ROUNDDOWN(srcva, BY2PG);
    round_dstva = ROUNDDOWN(dstva, BY2PG);

    //your code here
    /* 判断权限和va是否合法 */
    if((perm & PTE_V) == 0) return -E_INVAL;
    if((srcva >= UTOP) || (dstva >= UTOP)) return -E_INVAL;
    /* 根据envid寻找目标进程 */
```

```
    if((ret = envid2env(srcid, &srcenv, 0)) < 0) return ret;
    if((ret = envid2env(dstid, &dstenv, 0)) < 0) return ret;
    /* 在源进程的目录下寻找源虚拟地址对应的页面 */
    ppage = page_lookup(srcenv->env_pgdir, round_srcva, &ppte);
    /* 判断寻找结果与权限 */
    if(ppage == NULL) return -E_INVAL;
    if((ppte != NULL) && ((perm & PTE_R) == 1) && ((*ppte & PTE_R) == 0)) return -
E_INVAL;
    /* 将寻找到的页面插入目标进程的页目录中 */
    ret = page_insert(dstenv->env_pgdir, ppage, round_dstva, perm);
    return ret;
}
```

下面一个函数是`sys_mem_unmap()`，其作用是解除某个虚拟地址在某个进程当中与某个物理页面的映射关系。

```
int sys_mem_unmap(int sysno, u_int envid, u_int va)
{
    int ret;
    struct Env *env;

    ret = 0;
    /* 判断va是否合法 */
    if(va >= UTOP) return -E_INVAL;
    /* 安全地寻找目标进程 */
    if((ret = envid2env(envid, &env, 0)) < 0) return ret;
    /* 解除虚拟地址va的映射 */
    page_remove(env->env_pgdir, va);
    return ret;
}
```

最后一个函数是`sys_yield()`，它使得当前进程放弃CPU，并且进行进程调度。

```
void sys_yield(void)
{
    /* 保护进程现场 */
    bcopy((void *)KERNEL_SP - sizeof(struct Trapframe),
          (void *)TIMESTACK - sizeof(struct Trapframe),
          sizeof(struct Trapframe));
    /* 调用进程调度函数 */
    sched_yield();
}
```

相比于**实现系统调用**的部分，这几个系统调用函数的作用和实现方法都更好理解。了解并且能够正确使用在前几个lab中写好的函数，应当能够顺利完成；如果对某些函数的功能有所遗忘，还需回头温习。

二、进程通信IPC

在微内核系统中，进程的一些相关功能如文件系统、驱动等都被移出内核。为了让不同的进程能够相互沟通、请求资源，人们实现了IPC机制。IPC机制有如下的要点：

- 实现两个进程的通讯
- 通过系统调用发挥作用
- 以页为基础交换数据

我们知道，每一个进程都有自己的虚拟地址空间，页号相同的的虚拟页面可能映射不同的物理页面。故而实现进程通信的关键在于让两个不同的进程读取到同样的物理页面。顺着这一思路寻找，我们会发现不同的进程其实还是拥有同一片空间的，那就是内核空间。这是因为每一个进程在初始化时，它们的地址空间在内核区域的映射都是统一的。故而，我们可以利用内核空间实现IPC。具体的来说，就是：

- 数据发出进程在系统调用前将需要传输的数据放入内核空间
- 数据接收进程通过系统调用从内核空间读取数据

为了配合系统调用实现IPC机制，我们在进程控制块中添加了一些域，便于标记进程状态、保存沟通信息。

域名	功能
env_ipc_value	进程传递的具体数值
env_ipc_from	发送方的进程ID
env_ipc_recving	进程状态标记，1表示进程等待接受数据中，0表示不可接受数据
env_ipc_dstva	接收到的页面需要与自身的哪个虚拟页面完成映射
env_ipc_perm	传递的页面的权限位设置

很显然，我们在进行系统调用时，需要对这些值进行设置，然后陷入内核态。从本质上来讲，IPC机制也是系统调用的一种。完成这一机制，需要实现两个系统调用函数，即`sys_ipc_recv()`和`sys_ipc_can_send()`。

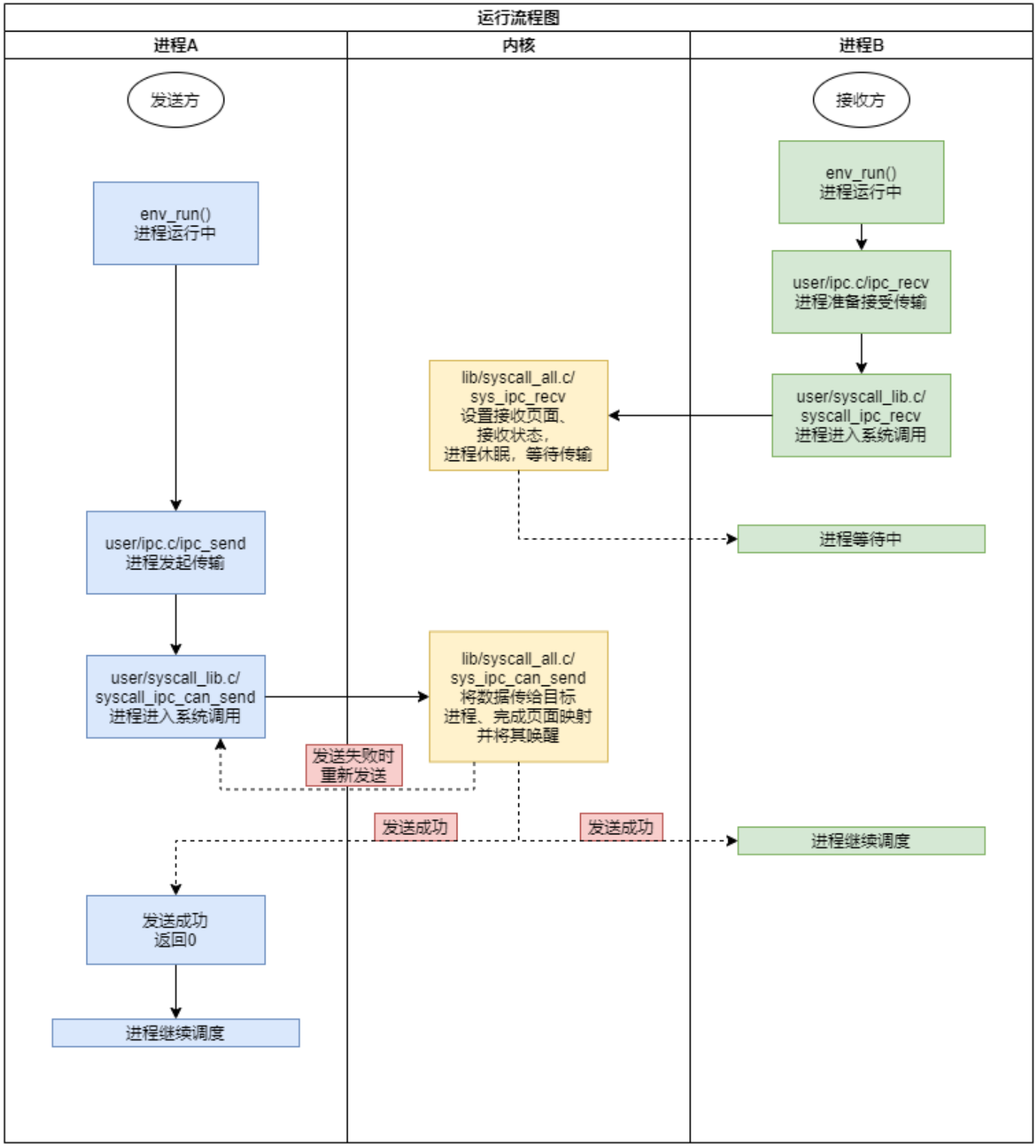
```
void sys_ipc_recv(int sysno, u_int dstva)
{
    /* 判断dstva是否合法，dstva是本进程需要映射到页面 */
    if(dstva >= UTOP) return;
    if(curenv != NULL){
        /* 设置进程状态为可接受数据 */
        curenv->env_ipc_recving = 1;
        /* 设置进程接收数据的虚拟地址 */
        curenv->env_ipc_dstva = dstva;
        /* 设置进程运行状态为阻塞态，即需要等待数据 */
        curenv->env_status = ENV_NOT_RUNNABLE;
        /* 调用调度算法，进程开始等待其他进程传输数据 */
        sys_yield();
        //Env;
    }
}

int sys_ipc_can_send(int sysno, u_int envid, u_int value, u_int srcva,
                     u_int perm)
```

```
{

    int r;
    struct Env *e;
    struct Page *p;
    struct Pte *ppte;
    if(srcva >= UTOP) return -E_INVAL;
    /* 安全地获得目标进程，即envid对应的进程 */
    if((r = envid2env(envid, &e, 0)) < 0) return r;
    /* 目标进程不能接受信息，返回错误 */
    if(e->env_ipc_recving != 1) return -E_IPC_NOT_RECV;
    /* 设置目标进程控制块相应值 */
    e->env_ipc_value = value;          //设置value，一个可传递的值
    e->env_ipc_from = curenv->env_id; //设置目标进程的接收进程id为curenv的id
    e->env_ipc_recving = 0;            //设置目标进程为不可接受，因为其即将完成数据接收
    e->env_ipc_perm = perm;            //设置页面权限
    e->env_status = ENV_RUNNABLE;      //设置目标进程为就绪态，即唤醒该进程
    //if(srcva >= UTOP) return -E_INVAL;
    /* 源地址不为零，则需要进行页面映射 */
    if(srcva != 0) {
        /* 根据虚拟地址，在当前进程的页目录中寻找目标页 */
        p = page_lookup(curenv->env_pgdir, srcva, &ppte);
        if(p == 0) return -E_INVAL;
        /* 将找到的页面根据虚拟地址插入到目标进程的页目录中 */
        if((r = page_insert(e->env_pgdir, p, e->env_ipc_dstva, perm)) < 0) return r;
    }
    return 0;
}
```

搬用指导书的图片，进程通信的流程大致如下：



此图很好地表现了进程通信过程中系统调用发挥的作用。

Thinking 4.2

0在MOS中用于表示当前进程，是为`curenv`专门保留的一个位置。在**系统调用**和**IPC**两部分中，很多函数都传入了`envid`；除此之外，我们要注意到，在我们使用系统调用时，很多情况下都是在一个进程和内核之间进行往返。为了便于识别进程，保留一个0用于表示当前进程是很方便的。在`envid2env()`函数中，识别到`envid`为0就直接得到`curenv`，毋需再从数组中存取。

三、Fork

1.基本认知

fork，本质上也是系统调用的一种。在Linux环境下，其被封装成一个库函数API，可以在高级语言中直接调用。关于fork功能的最简单的描述是：某一个进程通过fork创建一个新的进程，且新的进程在被创建时有着和原来的进程相同的上下文环境。

某一进程调用fork函数后，该进程将会在宏观上发生"分岔"，即由此伸出了两个进程——一个父进程，一个子进程。在两个进程中，fork函数的返回值并不相同：在父进程中返回子进程的id，在子进程中返回0。

```
#include <stdio.h>
#include <unistd.h>

int main() {
  int var = 1;
  long pid;
  printf("Before fork, var = %d.\n", var);
  pid = fork();
  /* 在fork()函数后，已经存在两个不同的进程 */
  printf("After fork, var = %d.\n", var);
  /* 子进程中，pid为0，会执行第一个if判断 */
  /* 父进程会执行第二个判断。var两个进程中独立 */
  if (pid == 0) {
    var = 2;
    sleep(3);
    printf("child got %ld, var = %d", pid, var);
  } else {
    sleep(2);
    printf("parent got %ld, var = %d", pid, var);
  }
  printf(", pid: %ld\n", (long) getpid());
  return 0;
}
```

Thinking 4.3

子进程仅执行了fork函数后的代码，没有执行fork函数前的代码，可以猜测子进程可能和父进程共享代码段，也可以猜测子进程在被系统调用创建后，系统调用令子进程回到了fork函数结束的位置，继续执行代码。

Thinking 4.4

关于 fork 函数的两个返回值，下面说法正确的是：

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

在这一部分中，我们有很多函数需要填写。我会首先独立地介绍每个函数的功能，随后将会归纳总结他们之间的调用关系。

2.写时复制机制

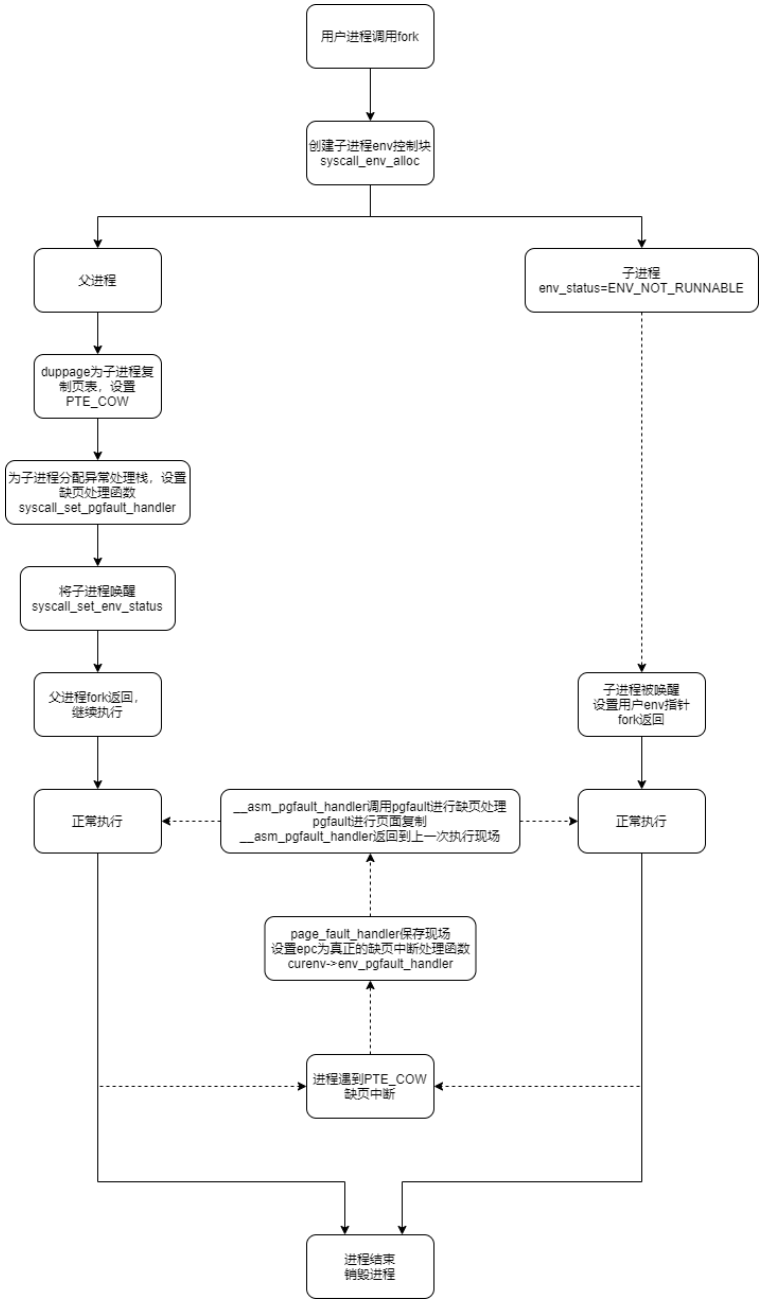
在前面的思考题当中，我们猜测父子进程可能共享代码段。实际上不仅仅如此，父子进程甚至共用了物理内存。子进程的代码段、数据段、堆栈都映射到了父进程中相同区段对应的页面。

我们学过多线程和并发，既然不同的进程共享了相同的物理页面，那么会不会在读写的时候产生数据竞争呢？这么想是没有错的，因为这种情况的确可能发生。但在介绍解决方法前，我们先关注fork函数的一些特征。我们使用fork函数的时候，可能不总是希望它个exce函数一样创建一个全新的子进程，而是希望子进程延用父进程的一些东西。所以我们并不会选择每一次调用fork函数就将父进程的所有数据照搬一遍，而是会令它们共享某些数据，这样可以节约空间和时间。那么，假如在没有调用exce的情况下也出现了修改物理内存的情况怎么办？这时候我们就需要写时复制机制了。

写时复制机制的实现思路大致如下：在执行fork函数之后，内核在给子进程配置环境的时候，会将共享页面的`PTE_COW`权限置为有效。随后进程执行的时候，倘若仅对这些页面进行读操作，那么会被认为是安全的；倘若某个进程对他们进行了写操作，那么就会产生异常，进入异常处理函数。在异常处理函数中，内核会解除原来虚拟页面的映射，并将其映射到一个新的物理页面，将原物理页面的内容拷贝到新页面中，同时取消虚拟页面的`PTE_COW`标记。写时复制机制保证仅当进程修改共享页面时才重新分配物理页面，能够节省相当的空间和时间。

3.子进程的创建

前面提到过，在Linux系统当中，fork函数实际上是一个封装好的系统调用API，即其实现功能的根本依赖还是系统调用。可以借鉴指导书的流程图来观察这个过程。



可见，子进程是父进程通过调用`syscall_env_alloc()`函数及之后一系列函数创建的。父进程通过通过系统调用，从内核态返回用户态时恢复现场；而子进程则是在进程被调度时返回用户态时恢复现场。系统调用使得他们返回现场时`$v0`寄存器储存的值不相同，故而可以通过进程id区分父子进程。

在`syscall_env_alloc()`中，我们开辟了一个新的进程。随后我们需要将当前进程的一些信息填入这个新的进程当中。

```
int sys_env_alloc(void)
{
    // Your code here.
    int r;
    struct Env *e;
    /*  申请一个新的进程控制块 */
    if((r = env_alloc(&e, curenv->env_id)) < 0) return r;
    /* 将当前栈的内容，即curenv的栈的内容，拷贝到目标进程的栈空间当中 */
    bcopy((void *)KERNEL_SP - sizeof(struct Trapframe),
```

```
            (void *)&(e->env_tf),
            sizeof(struct Trapframe));
    /* 设置进程控制块的相关值 */
    /* 将pc寄存器的值设置为系统调用发生时的后一条地址，即保证返回时从系统调用下一条指令继续执行 */
    e->env_tf.pc = e->env_tf.cp0_epc;
    /* 覆盖返回值，子进程中$v0寄存器得到的值为0 */
    e->env_tf.regs[2] = 0;
    /* 设置子进程为阻塞状态，需要等待父进程完成创建后再唤醒 */
    e->env_status = ENV_NOT_RUNNABLE;
    /* 设置子进程的优先级 */
    e->env_pri = curenv->env_pri;
    return e->env_id;
    //  panic("sys_env_alloc not implemented");
}
```

4.进程分岔

MOS操作系统允许进程访问自身的进程控制块。用户程序在入口会将一个指针变量指向当前进程的进程控制块。我们在创建子进程的同时需要修改这个值，使其指向自身进程的进程控制块。这是`exercise4.9`的内容。

```
/* 得到系统调用sys_env_alloc的返回值。注意，newenvid的值获得已经是在系统调用结束之后，故而此时已经发生了进程分岔 */
newenvid = syscall_env_alloc();
/* 如果改进程为新创建的子进程，即系统调用函数返回值为0，则设置env */
if(newenvid == 0) {
  /* 使用系统调用得到当前进程的id，并从进程控制块数组中得到该进程赋给env */
  env = &envs[ENVX(syscall_getenvid())];
  return 0;
}
```

以上是子进程在fork函数中执行的最后一个部分，因为子进程的id为0，很快就将从fork函数中返回。但仅仅如此，子进程还不足以运行起来，我们还需要在父进程的fork函数中设置一些有关于子进程的内容。

回想一下，首先我们需要设置子进程的页面。使其能够和父进程共享物理内存空间。此时，我们需要遍历父进程的用户空间页（也就是kuseg空间），并且设置`PTE_COW`将这些页面进行保护（原因已经在**写时复制机制**时提及）

在`exercise4.10`中，我们需要对`USTACKTOP`以下的空间进行`duppage()`操作，将这一部分的空间共享给父子进程；同时，我们还需要给某些页面设置`PTE_COW`权限，以确保**写时复制机制**的正常实现。首先我们在`fork.c`中进行页面遍历，这部分在`syscall_env_alloc()`之后。

```
    /* 将USTACKTOP以下的页面duppage */
    for(i = 0;i < USTACKTOP;i += BY2PG) {
        /* 用*vpd取出地址i对应的一级页表表项，不为空则继续 */
        if((Pde *)(*vpd)[i >> PDSHIFT]) {   //PDSHITF equals 22
            /* 用*vpt取出地址i对应的二级页表表项，又不为空表示该页面的确存在，可duppage */
            if((Pte *)(*vpt)[i >> PGSHIFT]) {   //PGSHIFT equals 12
                /* VPN(i)是i对应的虚拟页号 */
                duppage(newenvid, VPN(i));
            }
        }
    }
```

接下来，我们来关注`duppage()`函数的实现。

```
static void
duppage(u_int envid, u_int pn)
{
    u_int addr;
    u_int perm;
    /* addr为pn这一虚拟页号对应的虚拟地址 */
    addr = pn * BY2PG;      //Get the value of 'addr'.
    /* 从二级页表中取出pn对应的页表项，并拿取低12位的权限位 */
    perm = ((*vpt)[pn]) & 0xfff;
    /* 如果该页面的权限不是PTE_R，或者是PTE_LIBRARY，或者是PTE_COW，按照原页面的权限进行
     * 映射，此时只需要映射子进程 */
    if(((perm & PTE_R) == 0) || (perm & PTE_LIBRARY) || (perm & PTE_COW)) {
        if(syscall_mem_map(0, addr ,envid ,addr ,perm) < 0) {
            user_panic("duppage not implemented\n");
        }
    } else {
        /* 否则该页面的权限有变，在父子进程中都需要重新设置标志位 */
        if(syscall_mem_map(0 ,addr ,envid ,addr ,perm | PTE_COW) < 0) {
            user_panic("duppage not implemented\n");
        }
        if(syscall_mem_map(0 ,addr ,0 ,addr ,perm | PTE_COW) < 0) {
            user_panic("duppage not implemented\n");
        }
    }
    //  user_panic("duppage not implemented");
}
```

`duppage()`后，页面的权限设置就完成了。不过到此，子进程还不能被唤醒，因为我们仅仅是给页面设置了标志位，还没有完成写时复制机制。

Thinking 4.5

用户空间的大部分区域都需要进行映射，内核空间则不需要。对于每个分区，我们进行如下分析：

- `UTOP`到`ULIM`的区域存储了`envs`数组、`pages`数组等关键信息，这一部分可以被访问，但是不可被修改，其已经被保护好，不需要在再进行保护。
- `USTACKTOP`到`UTOP`的区域为`Invalid memory`和用户异常栈，不能设置写时保护。

`USTACKTOP`以下的页面都是需要进行写时保护的。

Thinking 4.6

vpt为二级页表的起始地址（所有的二级页表的起始），vpd为一级页表的起始地址。我们来一步一步地追溯它们的本源。

首先，我们可以在`mmu.h`中找到他们被定义为数组的地方:

```
typedef u_long Pde;
typedef u_long Pte;

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

可见，这两个变量的本质都是`u_long`类型的指针，指向改类型的数组。那么他们的值在哪里被定义呢？我们来到`entry.S`中，发现这个汇编文件中，定义了`vpt`和`vpd`这两个宏。其中，`vpt`为一个字，其中填充了`UVPT`；而`vpd`则要复杂一些，它所对应的字中填充了`(UVPT+(UVPT>>12)*4)`。咋一看是否有点眼熟？这时候我们会发现，兜兜转转又回到了`mmu.h`中，我们可以在内存地图上看到`UVPT`的地址，就是`0x7fc00000`，也就是二级页表的起始地址。到这里，这两个变量的意义就已经明晰了。

由于这两者根据其存放在内存中的位置实现了内存自映射机制，故而我们的进程可以通过访问内存来访问一级页表，进而访问二级页表，进而访问其自身的整个页表。

二级页表的起始位置在`0x7fc00000`，这个位置往上4KB的内存空间对应的是`(UVPT>>12)`的虚拟页面。要实现自映射机制，一级页表的起始位置应当在二级页表中，且我们知道一级页表中的第一个32位数应当能拿取到第一个二级页表所在页的虚拟地址，而二级页表的第一个32位数则能取到所有页中的第一个页，故而一级页表的第一个32位数应当相对`UVPT`发生偏移，需要偏移`(UVPT>>12)`个bit即`(UVPT>>12)*4)`个字节。

用户进程无法修改页表项。

5.页写入异常

内核捕捉到缺页中断时，即TLB缺失，将会进入异常处理状态，处理该异常的向量已经在lab3中填写好了，其对应的异常码为8，异常处理函数为`handle_tlb`。该函数定义在`genex.S`中，化名为`do_refill`（详见lab3的思考题解答），其行为：

- 若该物理页面在页表中存在，则将其填入TLB并且返回异常发生处继续执行
- 若该物理页面不存在，则重新分配

举这个例子是为了便于我们理解写时复制机制的作用原理，其也是依靠异常处理来实现的。在`traps_init()`函数中这一异常的码号为1，对应的异常处理函数为`handle_mod`，不过这个函数不使用汇编语言完成的，而是实现为`lib/traps.c`中的`page_fault_handler()`函数。这一函数主要负责保存现场，以及设置`tf->cp0_epc`的值为`env_pgfault_handler`即异常处理函数的地址。

```
void page_fault_handler(struct Trapframe *tf)
{
    struct Trapframe PgTrapFrame;
    extern struct Env *curenv;

    bcopy(tf, &PgTrapFrame, sizeof(struct Trapframe));

    if (tf->regs[29] >= (curenv->env_xstacktop - BY2PG) &&
        tf->regs[29] <= (curenv->env_xstacktop - 1)) {
            tf->regs[29] = tf->regs[29] - sizeof(struct  Trapframe);
            bcopy(&PgTrapFrame, (void *)tf->regs[29], sizeof(struct Trapframe));
        } else {
            tf->regs[29] = curenv->env_xstacktop - sizeof(struct  Trapframe);
            bcopy(&PgTrapFrame,(void *)curenv->env_xstacktop - sizeof(struct
 Trapframe),sizeof(struct Trapframe));
        }
    // TODO: Set EPC to a proper value in the trapframe
    tf->cp0_epc = curenv->env_pgfault_handler;
    return;
}
```

当然，上面提到，这个函数并没有实现页面复制，它只是将现场保存到了异常处理栈中，并在`cp0_epc`中设置了一个新的异常处理函数，以便跳转执行。跳转到的函数使定义在`fork.c`中的`pgfault()`函数。

- 判断页面是否为COW标记页面，不是则`panic`
- 分配临时物理页，将内容拷贝到该页面
- 将发生异常写入的地址映射到新分配的临时页面，设置权限位，并解除临时页面的映射

```
static void pgfault(u_int va)
{
    u_int *tmp;
    /* 取出权限位，并判断 */
    u_int perm = (*vpt)[VPN(va)] & 0xfff;
    if((perm & PTE_COW) == 0) {
        user_panic("Not a COW page!\n");
        return;
    }
    tmp = USTACKTOP;
    /* 将发生异常的地址进行页对齐 */
    va = ROUNDDOWN(va ,BY2PG);
    /* 在USTACKTOP适用系统调用分配页面 */
    if(syscall_mem_alloc(0 ,tmp ,PTE_V | PTE_R) < 0) {
        user_panic("Failed to alloc!\n");
```

```
        return;
    }
    /* 拷贝页面内容 */
    user_bcopy((void *)va ,(void *)tmp ,BY2PG);
    /* 映射临时页面到异常页面 */
    if(syscall_mem_map(0, tmp, 0, va, PTE_V | PTE_R) < 0) {
        user_panic("Failed to map!\n");
        return;
    }
    /* 解除临时页面的映射 */
    if(syscall_mem_unmap(0 ,tmp) < 0) {
        user_panic("Failed to unmap!\n");
        return;
    }
    return;
}
```

`page_fault_handler()`函数没有进行页面复制的操作，相关操作实现在用户态函数`pgfault()`中。如果我们要在用户态实现异常处理操作，那么我们就不能使用正常的堆栈，因为正常的堆栈也可能会发生写入异常。为此，我们在内存布局中为进程分配了一个**异常处理栈**，在`UXSTACKTOP`的位置。父子进程都需要在fork函数中设置自己的异常处理栈：

```
//父进程异常处理栈设置，在sys_env_alloc()前
set_pgfault_handler(pgfault);
......
//子进程异常处理栈设置，在sys_env_alloc()后
if((ret = syscall_set_pgfault_handler(newenvid, __asm_pgfault_handler, UXSTACKTOP)) <
0)
```

然后，我们就能在发生异常的时候，通过`page_fault_handler()`函数存取异常处理栈了。我们一起来看一下这两个设置函数。

```
void set_pgfault_handler(void (*fn)(u_int va))
{
    /* 如果__pgfault_handler未被设置过，则设置。该变量是一个汇编宏 */
    if (__pgfault_handler == 0) {
        /* 分配异常处理栈以及处理程序__asm_pgfault_handler，该处理程序是一个汇编函数。
         * 使用系统调用为进程控制块设置 */
        if (syscall_mem_alloc(0, UXSTACKTOP - BY2PG, PTE_V | PTE_R) < 0 ||
        syscall_set_pgfault_handler(0, __asm_pgfault_handler, UXSTACKTOP) < 0) {
            writef("cannot set pgfault handler\n");
          return;
        }
    }
    /* 设置内核处理函数为pgfault */
    __pgfault_handler = fn;
}
```

```
int sys_set_pgfault_handler(int sysno, u_int envid, u_int func, u_int xstacktop)
{
    struct Env *env;
    int ret;
    /* 安全地获取进程控制块 */
    if((ret = envid2env(envid ,&env ,0)) < 0) return ret;
    /* 设置进程控制块的异常栈的值以及异常处理函数入口为__asm_pgfault_handler */
    env->env_xstacktop = xstacktop;
    env->env_pgfault_handler = func;
    return 0;
}
```

可以看到，前一个函数实质上也是调用了后一个函数进行设置；而后一个函数有一点引人注目的地方，就是其异常处理函数`__asm_pgfault_handler`，这是定义在`entry.S`中的一个汇编函数。这些值都在前面向内存中写好了，汇编函数只需从内储存取即可。

```
/* 从内核返回后，栈指针的值在Trapframe的底部 */
__asm_pgfault_handler:
lw      a0, TF_BADVADDR(sp)    //将发生写入异常的地址传入参数寄存器
lw      t1, __pgfault_handler //将异常处理函数入口写入t1
jalr    t1                     //跳转至该函数
nop
/* 恢复现场 */
lw      v1, TF_LO(sp)
mtlo    v1
lw      v0, TF_HI(sp)
lw      v1, TF_EPC(sp)
mthi    v0
mtc0    v1, CP0_EPC
lw      $31, TF_REG31(sp)

lw      $1, TF_REG1(sp)
lw      k0, TF_EPC(sp)
jr      k0
lw      sp, TF_REG29(sp)
```

到这里，父进程当中有关子进程的信息都已经设置完毕，我们只需要将子进程唤醒，并且插入到就绪队列中即可。

```
int sys_set_env_status(int sysno, u_int envid, u_int status)
{
    struct Env *env;
    int ret;
    /* 判断传入参数status是否合法 */
    if((status != ENV_RUNNABLE) && (status != ENV_NOT_RUNNABLE) && (status !=
ENV_FREE)) {
        return -E_INVAL;
    }
```

```
    /* 安全获得进程控制块 */
    if((ret = envid2env(envid ,&env ,1)) < 0) return ret;
    /* 如果要设置的进程状态为ENV_RUNNABLE并且目标进程的状态不是ENV_RUNNABLE
     * 那么将该进程控制块插入至env_sche_list[0]的尾部；反之，将该控制块从该队
     * 列中移除 */
    if (status == ENV_RUNNABLE && env->env_status != ENV_RUNNABLE) {
        LIST_INSERT_TAIL(&env_sched_list[0], env, env_sched_link);
    } else if (status != ENV_RUNNABLE && env->env_status == ENV_RUNNABLE) {
        LIST_REMOVE(env, env_sched_link);
    }
    /* 现在可以修改进程控制块的状态了 */
    env->env_status = status;
    return 0;
}
```

将以上的各个步骤组合在`fork.c`函数中，我们的任务就完成了。

```
int fork(void)
{
    u_int newenvid;
    extern struct Env *envs;
    extern struct Env *env;
    u_int i;
    u_int ret = 0;

    /* 设置父函数的写入异常机制 */
    set_pgfault_handler(pgfault);
    /* 使用系统调用声明一个新的进程控制块 */
    newenvid = syscall_env_alloc();
    if(newenvid == 0) {
        env = &envs[ENVX(syscall_getenvid())];
        return 0;
    }
    /* 映射用户空间页面 */
    for(i = 0;i < USTACKTOP;i += BY2PG) {
        if((Pde *)(*vpd)[i >> PDSHIFT]) {
            if((Pte *)(*vpt)[i >> PGSHIFT]) {
                duppage(newenvid, VPN(i));
            }
        }
    }
    /* 设置程序异常栈 */
    if((ret = syscall_mem_alloc(newenvid, UXSTACKTOP - BY2PG, PTE_V | PTE_R)) < 0)
        return ret;
    if((ret = syscall_set_pgfault_handler(newenvid, __asm_pgfault_handler, UXSTACKTOP))
< 0)
        return ret;
    if((ret = syscall_set_env_status(newenvid, ENV_RUNNABLE)) < 0)
        return ret;
```

```
    return newenvid;
}
```

Thinking 4.7

首先，我查询了一下"中断重入"这一概念的意思，得到的结论大致是：中断的时候发生了中断。我大致讲一下我的理解。对于我们的MOS操作系统来说，页写入异常和缺页异常是两种不同的TLB异常。当我们想从TLB中调取页面但是发生中断的时候，我们就会进入内存或者磁盘寻找页面并且将其调入TLB。倘若我们遇到了一种更糟糕的情况：进程出发了页写入异常，我们不得不进入中断并且为其分配页面；然而我们分配的页面不在TLB中，上一个中断却还没有服务完，操作系统不得不再进入一层中断。这应该是可能发生"中断重入"的原因，归纳地来讲，就是连续的重叠的中断所致。

MOS实现的是微内核，它将缺页异常的主要部分置于用户态下处理。用户需要依靠Trapframe的信息执行中断处理和恢复现场，故而我们将其保存在用户的"异常处理栈"中

Thinking 4.8

在用户态处理页写入异常符合微内核的设计理念，能够精简内核大小，同时也使该异常处理起来更加方便，性能更好。

第二问不是很理解题目的意思。从寄存器的使用方式来讲，在用户态可能发生指令跳转的情况时，将寄存器的内容依照栈指针寄存器sp的目标依次压入栈中，最后再跳转；恢复现场时，先取出sp寄存器的值，并依照其重定位恢复寄存器的值，最后再跳转。这样使用可以保证栈指针的安全性，从而使得通用寄存器在栈中有安全的备份。标准操作可参见`entry.S`中的汇编函数。

Thinking 4.9

父进程在调用系统调用创建子进程的时候，可能会提前出现缺页中断，需要设置缺页处理条件。

父进程会给`__pgfault_handler`变量赋值时，会触发缺页中断。由于没有提前配置中断处理，无法处理这样的缺页中断。

不需要，子进程与父进程共享这个值。

6.小结

这一部分的顺序有点混乱，不管是指导书还是本片笔记，第一次阅读的时候都会觉得比较吃力。我本人也是在勉强填完代码后再次阅读才能整理出一些零散的思路。为了不让这些宝贵的思路流失，以便于我日后复习，我现在把这一部分的梳理记录下来。

我们在这一部分实际上是完成的任务主要达到了两个目的：

- 完成fork功能，设置好父子进程的信息，完善写时复制机制
- 完善写时复制时的异常处理功能，或是说系统调用

Fork功能

在调用fork函数的时候，父进程先要设置自己的异常处理栈，并且调用系统调开辟子进程，子进程初始处于阻塞状态，等待父进程唤醒。

```
set_pgfault_handler(pgfault);
newenvid = syscall_env_alloc();
```

父进程继续设置子进程的一系列信息（包括写时复制机制，父进程调用系统调用为子进程设置）并在最后将子进程唤醒。此时父子进程就可以并发执行了。写时复制机制的设置过程如下：

写入时异常

父子进程在修改被`PTE_COW`保护的页面时，会触发写入时异常。处理的流程如下：

- 在异常向量组中注册`handle_mod`异常
- 保存`tf`至异常处理栈
- 进入`__asm_pgfault_handler`汇编函数，在内存中取出函数值，并跳转到`pgfault()`函数
- `pgfault()`函数处理，判断异常并进行页面拷贝
- 逐层返回