操作系统第五次实验报告-Lab4

以下为学生闫思桥(19241091)的$Lab4$实验报告

$Part1$ 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？**
- **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？**
- **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？**
- **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？**

MyAnswer

- **内核在保存现场的时候是如何避免破坏通用寄存器的？**

保存现场过程中只改写了$k0, $k1, $v0寄存器的值。$k0暂存了sp的值，$k1则用于帮助更新sp的值，$v0用于帮助存储非通用寄存器的值。$k0, $k1在MIPS规则中为暂时的、随便使用的寄存器，$v0则用于存放函数返回值，这三个寄存器的改写不会造成影响。

- **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？**

可以直接获取，系统调用核心部分执行前（需要用到寄存器$a0–$a3前）的过程中（从syscall_*类型的函数到msyscall再到syscall指令），寄存器$a0–$a3的值没有被改变过，因此可以直接使用。

- **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？**

在调用sys开头函数前我们先人工把参数加载到了sys开头函数认为的位置。在syscall开头的函数中，我们传入了6个参数，第一个参数是**宏定义的系统调用号**，通过系统调用号，就可以在handle_sys函数中正确调用对应的sys开头的函数。而sys开头的函数的第一个参数是用于占位的（即对应着对它来说已经没用的系统调用号），这样sys开头的函数实际需要的5个参数正好对应着syscall开头的函数传递的参数的位置。

- **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？**

处理中，Trapframe中cp0_epc增加了4，并将执行系统调用后的返回值存入了$v0寄存器，使得系统调用结束后用户态可以获得正确的系统调用返回值并从发生系统调用的下一条指令开始执行。

下面是handle_sys函数中的相应片段

cp0_epc增加4

```
/* TODO: 将Trapframe的EPC寄存器取出，计算一个合理的值存回Trapframe中 */
lw      t0, TF_EPC(sp)
addiu   t0, t0, 4
sw      t0, TF_EPC(sp)
/* TODO: 将系统调用号“复制”入寄存器$a0 */
lw      a0, TF_REG4(sp)
```

将执行系统调用后的返回值存入了$v0寄存器

```
/* TODO: 恢复栈指针到分配前的状态 */
addiu   sp, sp, 24                  /* 这里是将为了存放6个参数而分配的栈空间弹出（实际上只有第5个参数和第6个参数的位置：16(sp) 20(sp) 真的存放了参数值，其他4个是在a0-a3寄存器中，栈空间仅是占位置的） */
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
```

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。

MyAnswer

根据相关代码可以看出，mkenvid()函数不会返回0（在第10位上始终是1）。

这是因为在lab4涉及到的进程间通信（IPC）中，会大量使用srcva为0的调用来表示只传value值，而不需要传递物理页面。换句话说，当srcva不为0时，我们才建立两个进程的页面映射关系。

再联系envid2env函数中对于 `envid==0` 的特判（直接向参数中存入当前进程的物理内存控制块），就可以说明：**envid==0 是特意空出来的，用于使用envid2env()函数找到“当前进程 `curenv` ”：`envid2env(srcid, &srcenv, 0)`，其中`srcid == 0` 。事实上，在lab4中，envid为0是MOS修改了子进程的函数返回值，用于区分父子进程。MOS希望系统调用在内核态返回的 envid 只传递给父进程，对于子进程则需要对它的保存的现场Trapframe进行一个修改，从而在恢复现场时用 0 覆盖系统调用原来的返回值。**

Thinking 4.3

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？**

MyAnswer

- **子进程完全按照 fork() 之后父进程的代码执行，说明了什么？**

这说明了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？**

说明子进程在创建时保存了父进程的上下文、PC值等，状态与父进程一致。子进程保留的上下文环境是父进程运行到 fork() 函数的时候的上下文环境，因此只会沿着 fork() 之后父进程的代码执行，不会执行 fork() 之前父进程的代码。

更加准备地说，通过在后面关于fork函数的分析可以发现，epc记录的是syscall函数的后一条指令的地址。因此子进程会在syscall后一条指令处返回继续运行。由于此时子进程的tf.regs[2]处被填入了0，那么它从fork()函数里出来的时候，fork()函数在子进程的运行里，返回值是0。

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

MyAnswer

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。**

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。

MyAnswer

- UTOP以上的用户空间对于用户进程来说不可改变；系统空间对于每个进程来说是相同且不可改变的，因此UTOP以上空间不用进行保护；
- UXSTACKTOP - BY2PG到UXSTACKTOP的空间是用户进程的异常栈，若进行写时复制保护可能陷入死循环，因此不能被保护。
- USTACKTOP到USTACKTOP + BY2PG的空间在空间分布图上是Invalid memory，用不到所以无需保护。
- 去除上述，需要保护的是在UTEXT与USTACKTOP - BY2PG之间且被使用的页面。

Thinking 4.6 vpt vpd 极度重要

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？**
- **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？**
- **它们是如何体现自映射设计的？**
- **进程能够通过这种方式来修改自己的页表项吗？**

MyAnswer

先观察以下相关代码，说明vpt[]和vpd[]都是指针数组，**vpt和vpd本身 分别是 Pte* 类型的指针数组和 Pde* 类型的指针数组的 首个元素的指针，也是对于页表项的指针的指针，也可以理解为是整个指针数组的指针，类似于&数组名** ，数组中分别是 Pte* 类型和 Pde* 类型的指针。vpt被设置为了 `UVPT` ，vdt被设置为了 `(UVPT+(UVPT>>12)*4)` 。就是两个指针的指针。

> &arr表示的是整个数组的地址，arr表示的是该数组首元素的地址。

```
// mmu.h
extern volatile Pte* vpt[];
extern volatile Pde* vpd[];
```

```
# entry.S
    .globl vpt
vpt:
    .word UVPT

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

然后来回顾一下lab2中讲到的二级页表自映射

二级页表目录自映射

二级页表目录自映射的总体逻辑是：首先先用4M的二级页表区映射4G的虚拟空间，然后再用二级页表中的某一个二级页表（4K）作为一级页表表示这些二级页表（4M）。

首先是虚拟空间（4G）到二级页表区（4M）的映射

首先，用含有 1M 个页表项的二级页表区（1024个页表）映射4G的虚拟空间（4K为基本单位，共2^20页），则一个页表项代表着4G/1024=4M=2^12 B大小的虚拟空间（恰好是1页），则二级页表区起始地址 $UVPT$ `0x7fc0 0000` 对应的是二级页表区中的第 `(0x7fc0 0000)>>12` 个页表项，而每个页表项大小是4B，则二级页表区起始地址 `0x7fc0 0000` 在二级页表区内的偏移量是 `( (0x7fc0 0000)>>12 )<<2 = 0x1ff000 B`，即在二级页表区中对应的地址是 `0x7fc0 0000 +  0x1ff000 = 0x7fdff000 B`。

然后是二级页表区（4M）到一级页表/页目录（4K）的映射

然后，再用二级页表区起始地址 `0x7fc0 0000` 在二级页表区中对应的地址 `0x7fdff000` 的这一个页表作为一级页表，映射整个二级页表区。那么 `0x7fdff000` 改称为一级页表/页目录基地址。

一级页表在二级页表区中的偏移量是 `(0x7fc0 0000)>>10`，则一级页表对应的地址在一级页表中的偏移量是这个量再右移12位（对应的页表项）乘以4（每个页表项的大小），所以就是 `(0x7fc0 0000)>>20`。

可以发现，全局变量vpt的值是 `UVPT ( 0x7fc0 0000 )`，也就是指向当前进程空间中二级页表区（也就是二级页表项数组）的地址的指针。 `(*vpt)[0]` **就是二级页表区的第一个二级页表项。**为什么这么说呢?

因为vpt的变量类型为Pte*，它作为一个指针，指向的就是 `u_long(unsigned long)` 类型。因此在用 (*vpt) 进行指针的加减运算的时候，步长其实就是u_long的32位，也就是一个二级页表项的大小。

> 注意不是直接拿vpt运算，因为无论是指向什么类型的指针，指针变量本身都是32位。

vpt的变量类型为Pte*，所以可以通过 $(*vpt)[index], index \in [0, 1024 * 1024 - 1]$ 范围内的下标进行数组寻址（1024个二级页表打破彼此的隔阂，以二级页表项为单位进行遍历），找到任意一个进程空间中的二级页表区的二级页表项（**的指针**）。进而通过这个页表项，按页为单位找到4G空间。

> 对于一维数组来说，$a[i] = *(a + i)$，那么
> $(*vpt)[index] = *((*vpt) + index) = *((*\&\text{指针数组}Array) + index) = *(\text{指针数组}Array + index) = Array[index]$
>
> vpt存放的就是Array的地址，vpt是指向Pte类型的指针。

而全局变量vpd的值是 `(UVPT+(UVPT>>12)*4)`，也就是当前进程空间中用于表示二级页表区的一级页表的基地址。 `(*vpd)[0]` **就是一级页表的第一个一级页表项。**为什么这么说呢?

因为vpd*变量类型是Pde*，它作为一个指针，指向的就是 `u_long(unsigned long)` 类型。因此在用 (*vpt) 进行指针的加减运算的时候，步长其实就是u_long的32位，也就是一个一级页表项的大小。

vpd的变量类型为Pde*，所以可通过 $(*vpd)[index], index \in [0, 1023]$ 范围内的下标进行数组寻址，找到任意一个一级页表项（**的指针**），每一个一级页表项又代表一个二级页表区中的二级页表。

下面举一个fork()函数中，利用vpd和vpt检查页表项有效性的例子。

```
for (i = 0; i < VPN(USTACKTOP); ++i) {
// i是虚拟页表号，也就是二级页表区中的第i个二级页表项
// 而一个一级页表项表示一个二级页表，
// 那么 i >> 10 就是计算出这个二级页表项属于第几个二级页表，也就是由第几个一级页表项所代表
if (((*vpd)[i >> 10] & PTE_V) && ((*vpt)[i] & PTE_V))
  // 一级页表项 和 二级页表项 均有效
  duppage(newenvid, i);
}
```

如果是直接拿到一个虚拟地址va，那么检查方法如lab5中的 `va_is_mapped` 函数。

```
u_int va_is_mapped(u_int va)
{    // 用PDX(va)得到是在第几个一级页表项，用VPN(va)得到虚拟页号，就是第几个二级页表项
    return (((*vpd)[PDX(va)] & (PTE_V)) && ((*vpt)[VPN(va)] & (PTE_V)));
}
```

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

MyAnswer

- **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”？**

在用户发生写时复制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“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？**

我们在用户进程处理此缺页中断，因此用户进程需要读取Trapframe中的值；同时用户进程在中断结束恢复现场时也需要用到Trapframe中数据，因此存到用户空间。

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：

- **用户处理相比于在内核处理写时复制的缺页中断有什么优势？**
- **从通用寄存器的用途角度讨论用户空间下进行现场的恢复是如何做到不破坏通用寄存器的？**

MyAnswer

- **用户处理相比于在内核处理写时复制的缺页中断有什么优势？**

符合微内核设计理念，精简系统。

- **从通用寄存器的用途角度讨论用户空间下进行现场的恢复是如何做到不破坏通用寄存器的？**

首先使用存放函数调用返回值的$v0，$v1恢复非通用寄存器，之后通过$v1恢复非通用寄存器，之后通过$sp恢复通用寄存器，最后恢复$sp，保证了恢复后寄存器值正确。

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？**
- **如果放置在写时复制保护机制完成之后会有怎样的效果？**
- **子进程是否需要对在entry.S定义的字__pgfault_handler赋值？**

MyAnswer

- **为什么需要将 `set_pgfault_handler` 的调用放置在 `syscall_env_alloc` 之前?**

`set_pgfault_handler` 的调用放置在 `syscall_env_alloc` 之前是为了给父进程注册异常处理栈。`syscall_env_alloc` 过程中亦可能需要进行异常处理。

- **如果放置在写时复制保护机制完成之后会有怎样的效果?**

此时进程给__pgfault_handler变量赋值时就会触发缺页中断，但中断处理没有设置好，故无法进行正常处理。

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

不需要，子进程复制了父进程中__pgfault_handler变量值。

$Part2$ 实验难点图示

$Part2.1$ 系统调用(System Call)

用户态与内核态

首先记录一下我自己对于进程空间的理解：

每一个进程的页目录建立的二级页表体系，映射了4G空间。也就是说，这总共1024*1024=1M个页表项，映射了4G空间，那么很明显，一个页表项代表了4K大小，也就是一个BY2PG。这样一种映射关系下的4G空间，叫做进程空间。

进程空间的4G空间是对标内核空间的4G空间的。在建立这样一个映射体系（即为进程申请页目录）的时候，我们会发现，其实这个页目录只有$[0, PDX(ULIM) - 1]$即0-511项是会用到的，因为在内核的4G空间中，ULIM代表了用户态能操作的空间kuseg地址的上限，再往上就是内核态专属空间kseg0和kseg1了。而在0-511项中，第509项和第510项是从 $boot\_pgdir$ 中复制过来的，这是内核曝露给用户空间的两个一个页表项，这两个页表项分别是$PDX(UENVS)$和$PDX(UPAGES)$，是内核空间中进程控制块们和物理内存控制块们所对应的一级页表项。

因此，进程空间的4G空间中的虚拟地址，很明显并不是内核空间的4G空间中的虚拟地址。进程空间中的虚拟地址va，只是为了在进程页目录的二级页表体系下，找到对应的二级页表项pte_entry；并在二级页表项pte_entry中，存入这个地址所代表的内容（ELF文件也好，其他内容也好）真正被存放到的物理地址对应的页面号码+权限位。当然，这里的二级页表，在内核空间中是分配了一页内存的。也就是说：进程的一级页表和二级页表本身是在内核空间中占有了内存的。

而在二级页表项中存放的物理页面号20位左移12位（也可以通过二级页表项的内容的低12位置零）得到物理地址，最高位置1转为虚拟地址，就是内核空间中，这个二级页表的虚拟地址。

因此，这也就是为什么，各个进程的地址空间是相互独立的。同一个虚拟地址在不同的进程中会对应不同的物理页面。可以说：进程页目录所映射的进程空间，就是“进程自己的内核空间”。当然，和真正的内核空间没有关系。

我们首先回顾以下几组概念：

1. 用户态和内核态（也称用户模式和内核模式）： 它们是 CPU 运行的两种状态。根据 lab3 的说明，在 MOS 操作系统实验使用的仿真 R3000 CPU 中，该状态由 CP0 SR 寄存器中 KUc 位的值标志。
2. 用户空间和内核空间：它们是虚拟内存（进程的地址空间）中的两部分区域。根据 lab2 的说明，MOS 中的用户空间包括 kuseg， 而内核空间主要包括 kseg0 和 kseg1。**每个进程的用户空间通常通过页表映射到不同的物理页，而内核空间则直接映射到固定的物理页以及外部硬件设备。** CPU 在内核态下可以访问任何内存区域，对物理内存等硬件设备有完整的控制权，而在用户态下则只能访问用户空间。

> **每个进程的用户空间通常通过页表映射到不同的物理页，而内核空间则直接映射到固定的物理页以及外部硬件设备。**这句话再次说明了，每个进程要建立映射的地址（如二进制文件要加载到的虚拟地址），都是在进程的页目录映射的4G空间下的地址，而不是真正的内核虚拟地址，这些被实际加载到的地方并不是进程页目录视野下的虚拟地址。

3. （用户）进程和内核：进程是资源分配与调度的基本单位，拥有独立的地址空间，而内核负责管理系统资源和调度进程，使进程能够并发运行。 与前两对概念不同，进程和内核并不是对立的存在，可以认为内核是存在于所有进程地址空间中的一段代码。

> 这里体现在为每一个进程申请一个物理页作为它的进程页目录时，都要从boot_pgdir上复制第509和第510项。（从第0项开始）

事实上，**由于仿真器 gxemul 的实现与 IDT R30xx 手册存在的差异**，lab3 中并没有使用真正的用户态。在 gxemul 的实现中，KUc 为 1 表示 CPU 处于用户态，此时若进程试图访问内核空间，则会触发 TLB 异常，这在 MOS 中通常导致内核输出 TOO LOW 并 panic， 而在 Linux 中进程通常会收到 SIGSEGV 信号，输出“段错误”（segmentation fault）等信息并退出。

由于 lab3 使用的进程仍在内核态运行，程序可以在不使用系统调用的情况下，直接读写内核空间的硬件地址，从而向控制台输出文本。 为了让进程被调度后严格在用户态下运行，我们需要修改进程控制块中保存的 SR 寄存器的初始状态。 结合 `env_pop_tf` 函数的实现，我们知道内核开始调度一个进程时，首先恢复其进程上下文， 然后使用 `rfe` 指令进入用户态，因此我们不能直接设置 KUc 位的值。

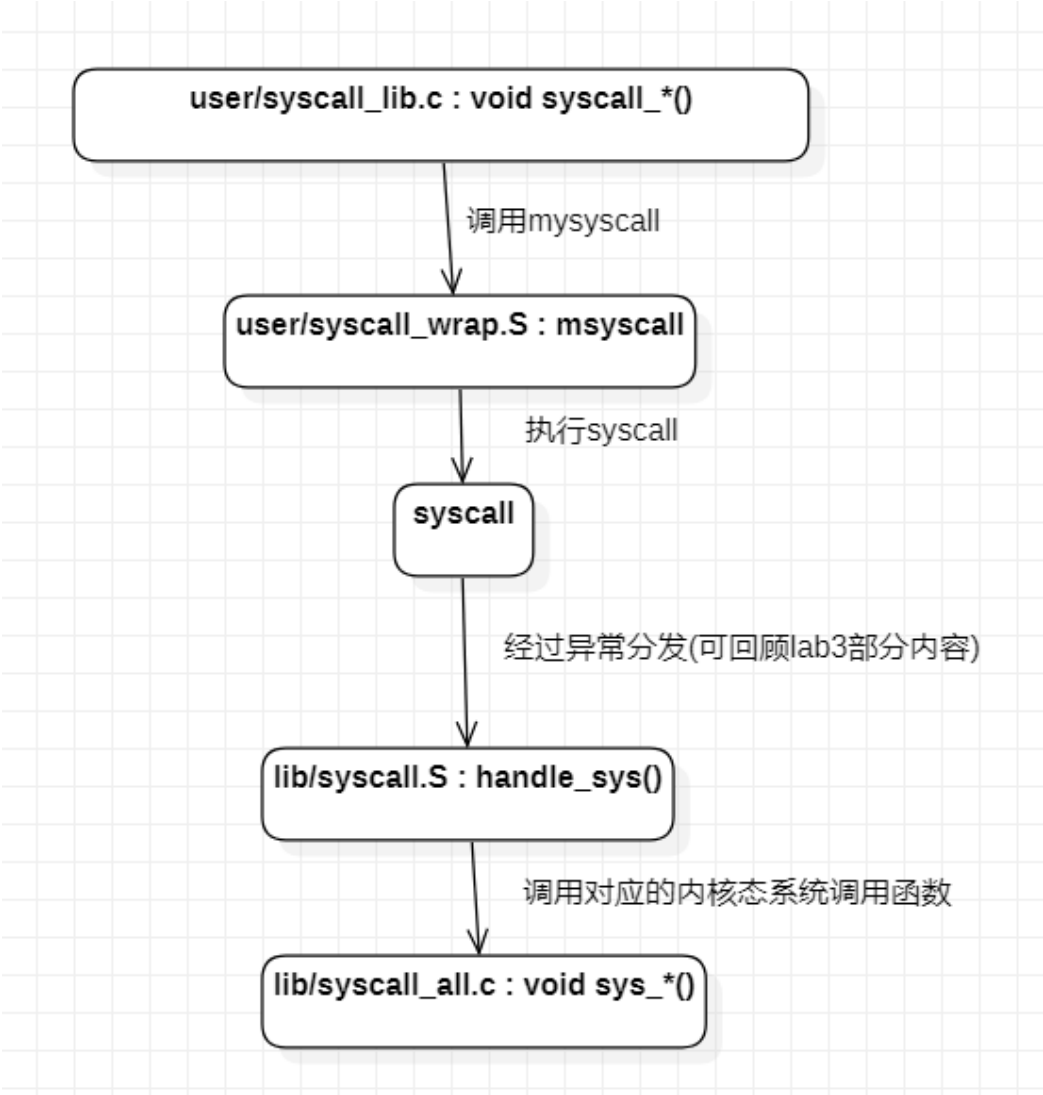
> 这也就是为什么需要：修改 lib/env.c 中的 `env_alloc()` 函数，使在新创建的进程控制块中，`env_tf.cp0_status` 的值为 `0x1000100c`。（在lab3中这个值是 `0x10001004`）
>
> `0x0001 0000 0000 0000 0001 0000 0000 1100`第28位、第12位、第2、3位（lab4）
>
> `0x0001 0000 0000 0000 0001 0000 0000 0100`第28位、第12位、第2位（lab3）
>
> 在进程运行以后，执行rfe指令，然后SR寄存器的低6位二重栈出栈，低两位在lab4中是11，这说明当前是用户态且会响应中断，而在lab3中（也就是实际的MOS中），是当前是内核态且会响应中断。

syscall这个极为特殊的指令。从它的名字我们就能够猜出它的用途， 它使得进程陷入到内核态中，执行内核中的相应函数，完成相应的功能。在系统调用完成后，用户空间的相关函数会将系统调用的结果， 通过一系列的返回过程，最终反馈给用户程序。

由此我们了解到，**系统调用实际上是操作系统为用户态提供的一组接口，进程在用户态下通过系统调用可以访问内核提供的文件系统等服务。**

系统调用机制的实现

系统调用的流程如下：



实际上，在我们的MOS操作系统实验中，这些 `syscall_*` 的函数与内核中的系统调用函数（ `sys_*` 的函数）是一一对应的： **`syscall_*` 的函数是我们在用户空间中最接近的内核的也是最原子的函数**，而 `sys_*` 的函数是内核中系统调用的具体实现部分。 `syscall_*` 的函数的实现中，它们毫无例外都调用了msyscall函数，而且函数的第一个参数都是一个与调用名相似的宏（如 `SYS_putchar` ）， 在我们的MOS操作系统实验中把这个参数称为**系统调用号**。

系统调用号

系统调用号是一群以SYS_开头的宏，均定义于include/unistd.h中。

```
#define __SYSCALL_BASE 9527
#define __NR_SYSCALLS 20

#define SYS_putchar          ((__SYSCALL_BASE ) + (0 ) )
#define SYS_getenvid         ((__SYSCALL_BASE ) + (1 ) )
#define SYS_yield            ((__SYSCALL_BASE ) + (2 ) )
#define SYS_env_destroy      ((__SYSCALL_BASE ) + (3 ) )
#define SYS_set_pgfault_handler ((__SYSCALL_BASE ) + (4 ) )
#define SYS_mem_alloc        ((__SYSCALL_BASE ) + (5 ) )
#define SYS_mem_map          ((__SYSCALL_BASE ) + (6 ) )
#define SYS_mem_unmap        ((__SYSCALL_BASE ) + (7 ) )
#define SYS_env_alloc        ((__SYSCALL_BASE ) + (8 ) )
#define SYS_set_env_status   ((__SYSCALL_BASE ) + (9 ) )
#define SYS_set_trapframe        ((__SYSCALL_BASE ) + (10 ) )
#define SYS_panic            ((__SYSCALL_BASE ) + (11 ) )
#define SYS_ipc_can_send         ((__SYSCALL_BASE ) + (12 ) )
#define SYS_ipc_recv         ((__SYSCALL_BASE ) + (13 ) )
#define SYS_cgetc            ((__SYSCALL_BASE ) + (14 ) )
```

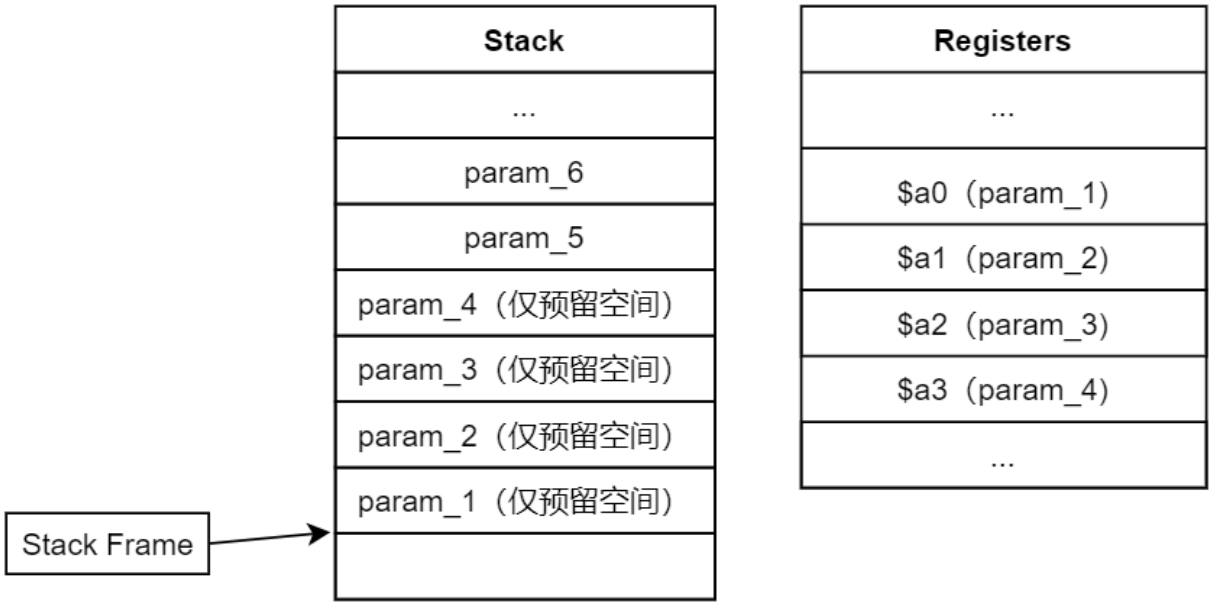
类似于不同异常类型对应不同异常号，系统调用号是内核区分这究竟是何种系统调用的唯一依据。除此之外msyscall函数还有5个参数，这些参数是系统调用时需要传递给内核的参数。而为了方便传递参数，我们采用的是最多参数的系统调用所需要的参数数量（`syscall_mem_map`函数需要5个参数）。

msyscall参数的传递（从用户态到内核态）

进一步的问题是，这些参数究竟是如何从用户态传递入内核态的呢？这里就需要用MIPS的调用规范来说明这件事情了。 我们把函数体中不存在函数调用语句的函数称为**叶函数**，自然如果函数体中存在函数调用语句，那么该函数称为**非叶函数**。

严格的讲，在MIPS的调用规范中，进入函数体时会通过对栈指针做减法（压栈）的方式 为该函数自身的**局部变量、返回地址、调用函数的参数**分配存储空间（叶函数没有后两者），在函数调用结束之后会对栈指针做加法（弹栈）来释放这部分空间，我们把 这部分空间称为**栈帧（Stack Frame）**。 调用方是在自身的栈帧的底部预留被调用函数的参数存储空间（被调用方g从调用方f的栈帧中取得参数）。

进一步，MIPS寄存器使用规范中指出，寄存器$a0-$a3用于存放函数调用的前四个参数（但在栈中仍然需要为其预留空间），剩余的参数仅存放在栈中。以我们的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（寄存器传参的部分）同时栈帧底部保留16字节的空间（不要求存入参数的值），后2个参数只会被存入在预留空间之上的8字节空间内（没有寄存器传参），于是总共24字节的空间用于参数传递。**这些过程在C代码中会由编译器自动编译为mips代码，但是 我们的`handle_sys`函数是以汇编的形式编写的， 需要手动在内核中以汇编的方式显式地把函数的参数值“转移”到内核空间中。详情请见下图：



既然参数的位置已经被合理安置，那么接下来我们需要编写msyscall函数，这个叶函数没有局部变量，也就是说这个函数不需要分配栈帧，我们**只**需要执行 特权指令（syscall）来陷入内核态并保证处理结束后函数能正常返回即可。

msyscall汇编函数（执行完它以后，CPU陷入内核态）

实现于syscall_wrap.S。也是这个文件的唯一函数。

用户态的`syscall_*`的函数的实现中，它们调用了msyscall函数，并将它的参数（系统调用号+5个系统调用时需要传递给内核的参数）存入了 $a0-$a4，以及在预留空间之上的8字节空间内。**注意这里是存放在用户态的栈中。**（如上图）

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

msyscall执行syscall指令触发8号异常

执行特权指令syscall，会触发了8号异常。PC转至 `0x80000080` ，开始由被链接器放在这一地址处的**异常分发函数.text.exc_vec3 段**来进行异常分发，通过加载**异常向量组**，计算偏移量，最终在异常分发向量组中选择合适的中断处理子函数，并跳转到这个中断处理子函数的地址处开始执行。8号异常，对应的便是 `handle_sys` 函数。

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，完成这部分功能的代码就是汇编宏SAVE_ALL。** `handle_sys` 函数也在最开始就调用了它。

在SAVE_ALL的最开始（还没有执行get_sp），此时的$sp还是用户栈指针（因为产生中断的时候硬件只会改变pc和一些cp0寄存器），$sp先将用户栈指针存放在$k0，然后调用get_sp宏函数得到异常处理栈存入$sp，无论是 `KERNEL_SP` 还是 `TIMSESTACK` ，都是内核栈指针。再将$sp下压一个 `Struct Trapframe` 大小，利用此时的$sp寻址，将$k0所存放的用户栈指针存在了 `TF_REG29(sp)` 处。

栈指针$sp指向这个结构体的起始位置， 因此我们正是借助这个保存的结构体来获取用户态中传递过来的值（例如：用户态下$a0寄存器的值保存在了当前栈下的TF_REG4(sp)处，这就是前一份实验报告中提到的TF_XXX的宏和寄存器的对应关系）。

handle_sys汇编函数

这个函数完成了将msyscall的6个参数从用户态的栈存入内核态的栈（内核态栈指针压栈）。并且通过计算系统调用函数的入口偏移量，调用对应的系统调用函数。并且将函数的返回值在内核态栈指针恢复的时候，存入 `TF_REG2(sp)` 。最后，执行恢复现场和回滚。

```
NESTED(handle_sys,TF_SIZE, sp)
    SAVE_ALL                                /* 用于保存所有寄存器的汇编宏 */
    # 此时 内核空间的栈指针sp被设置为 KERNEL_SP，其地址往上一个Struct Trapframe的大小内存放了原用户进程的运行现场---sp就是异常处理栈的栈底
    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 */

    /* TODO: 将Trapframe的EPC寄存器取出，计算一个合理的值存回Trapframe中 */
    lw      t0, TF_EPC(sp)
    addiu   t0, t0, 4
    sw      t0, TF_EPC(sp)
    /* TODO: 将系统调用号“复制”入寄存器$a0 ，这也是对msyscall第一个参数的处理*/
    lw      a0, TF_REG4(sp)

    addiu   a0, a0, -__SYSCALL_BASE     /* a0 <- “相对”系统调用号，即偏移量 */
    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 */

    lw      t0, TF_REG29(sp)            /* t0 <- 用户态的栈指针 */
    # 这里是SAVE_ALL中将用户态的栈指针存放在 TF_REG29(sp) 中
    # 利用用户态的栈指针将 用户态的栈中存放的参数 存入寄存器
    lw      t3, 16(t0)                  /* t3 <- msyscall的第5个参数 */
    lw      t4, 20(t0)                  /* t4 <- msyscall的第6个参数 */
    # 这里是16，20 是因为 第1到4个参数分别占据了0(t0),4(t0),8(t0),12(t0)
    /* TODO: 在当前栈指针分配6个参数的存储空间，并将6个参数安置到期望的位置 */
    // 这实际上就是让当前的异常处理栈 压栈，存放msyscall的参数，用于供给sys_*形式的函数用
    lw      a0, TF_REG4(sp)             /* a0 <- msyscall的第1个参数 */
    lw      a1, TF_REG5(sp)             /* a1 <- msyscall的第2个参数 */
    lw      a2, TF_REG6(sp)             /* a2 <- msyscall的第3个参数 */
    lw      a3, TF_REG7(sp)             /* a3 <- msyscall的第4个参数 */
```

```
    addiu   sp, sp, -24                 # 压栈，分配6个参数的存储空间
    # 将 得到的6个参数 存入内核态的栈中，这6个参数是给sys_*形式的函数用的
    sw      a0, 0(sp)
    sw      a1, 4(sp)
    sw      a2, 8(sp)
    sw      a3, 12(sp)
    sw      t3, 16(sp)
    sw      t4, 20(sp)

    jalr    t2                          /* 调用sys_*函数 */
    nop

    /* TODO: 恢复栈指针到分配前的状态 */
    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)

sys_call_table:                         /* 系统调用函数的入口表 */
.align 2
    .word sys_putchar
    .word sys_getenvid
    .word sys_yield
    .word sys_env_destroy
    .word sys_set_pgfault_handler
    .word sys_mem_alloc
    .word sys_mem_map
    .word sys_mem_unmap
    .word sys_env_alloc
    .word sys_set_env_status
    .word sys_set_trapframe
    .word sys_panic
    .word sys_ipc_can_send
    .word sys_ipc_recv
    .word sys_cgetc
    /* 每一个整字都将初值设定为对应sys_*函数的地址 */
    /* 在此处增加内核系统调用的入口地址 */ # 对应了上面的15个宏
```

保存现场到异常处理栈 SAVE_ALL

保存现场的主要逻辑处于汇编宏 `SAVE_ALL` 中，主要逻辑如下：

前四行其首先取出了 SR 寄存器的值, 然后利用移位等操作判断第 28 位的值, 根据前面的讲述我们可以知道, 也即判断当前是否处于用户模式下。

接下来将当前运行进程的用户栈的地址保存到 k0 中；然后调用 get_sp 宏，根据中断异常的种类判断需要保存的位置，获取处理栈指针（内核栈指针），再将处理栈顶下移一个 struct Trapframe 的大小；将之前存入 k0 的用户栈指针与 2 号寄存器 $v0 先保存起来，便于后面可以放心的使用 sp 寄存器与 v0 寄存器。剩下的部分便是用 `xxx(sp)` 的寻址方式复制整个 struct Trapframe 需要的信息，将各个cpu寄存器中的值复制到sp以上的对应位置。

> 1: ;A
> cmp r0, #0
> beq 1f ; r0==0那么 **向前**跳转到B处执行
> bne 1b ; 否则 **向后**跳转到A处执行
> 1: ;B
>
> 1b，1f里的b和f表示backward和forward，1表示局部标签1

```
.macro SAVE_ALL

        mfc0    k0,CP0_STATUS   # 将 SR 寄存器中的值存放到 k0 寄存器中
```

```
        sll     k0,3       /* extract cu0 bit */ # 现在cu0被移到了最高位
        # 第28bit CU0设置为1，表示允许在用户模式下使用 CP0 寄存器。
        bltz    k0,1f       # 如果cu0是1，则在移到最高位的情况下，就会小于0
        nop
        /*
         * Called from user mode, new stack
         */
        //lui   k1,%hi(kernelsp)
        //lw    k1,%lo(kernelsp)(k1)  //not clear right now

1:
        move    k0,sp       # 首先将用户栈指针存入 k0
        get_sp              # 然后使用 get_sp 获取处理栈指针（两种结果都是内核栈指针），存入sp
        move    k1,sp       # 再将找到的处理栈指针存入k1
        subu    sp,k1,TF_SIZE   # 将处理栈顶下移一个 struct Trapframe 的大小，压栈后的指针存
入sp
                                # （TF_SIZE 为 sizeof(struct Trapframe)）
        sw  k0,TF_REG29(sp) # 将用户栈指针 k0 存入 Trapframe 中的 TF_REG29 处
        sw  $2,TF_REG2(sp)  # v0 寄存器即 $2。由于借助 v0 这个用户态下会用到的寄存器，因此必须
先将用户态下 v0 的值存入 Trapframe 中
    # 借助 v0 寄存器将CP0寄存器以及HI，LO寄存器的值存入 Trapframe 中的对应位置
        mfc0    v0,CP0_STATUS
        sw  v0,TF_STATUS(sp)
        mfc0    v0,CP0_CAUSE
        sw  v0,TF_CAUSE(sp)
        mfc0    v0,CP0_EPC
        sw  v0,TF_EPC(sp)
        mfc0    v0, CP0_BADVADDR
        sw  v0, TF_BADVADDR(sp)
        mfhi    v0
        sw  v0,TF_HI(sp)
        mflo    v0
        sw  v0,TF_LO(sp)
        # 下面的GRF复制中没有$2  $29
        # 由于 $2（v0 寄存器），$29（sp 寄存器）都已经保存了，所以此处跳过它们.
        sw  $0,TF_REG0(sp)
        sw  $1,TF_REG1(sp)      # 跳过 $2
        sw  $3,TF_REG3(sp)
        sw  $4,TF_REG4(sp)
        sw  $5,TF_REG5(sp)
        sw  $6,TF_REG6(sp)
        sw  $7,TF_REG7(sp)
        sw  $8,TF_REG8(sp)
        sw  $9,TF_REG9(sp)
        sw  $10,TF_REG10(sp)
        sw  $11,TF_REG11(sp)
        sw  $12,TF_REG12(sp)
        sw  $13,TF_REG13(sp)
        sw  $14,TF_REG14(sp)
        sw  $15,TF_REG15(sp)
        sw  $16,TF_REG16(sp)
        sw  $17,TF_REG17(sp)
        sw  $18,TF_REG18(sp)
        sw  $19,TF_REG19(sp)
        sw  $20,TF_REG20(sp)
        sw  $21,TF_REG21(sp)
        sw  $22,TF_REG22(sp)
        sw  $23,TF_REG23(sp)
        sw  $24,TF_REG24(sp)
        sw  $25,TF_REG25(sp)
        sw  $26,TF_REG26(sp)
        sw  $27,TF_REG27(sp)
        sw  $28,TF_REG28(sp)        # 跳过 $29
        sw  $30,TF_REG30(sp)
        sw  $31,TF_REG31(sp)
.endm
```

基础系统调用函数

先介绍几个宏

```
#define E_INVAL     3   // Invalid parameter
```

syscall_mem_alloc()

```
int syscall_mem_alloc(u_int envid, u_int va, u_int perm)
{
    return msyscall(SYS_mem_alloc, envid, va, perm, 0, 0);
}
```

sys_mem_alloc()

该函数为id为envid的进程的**进程空间内的地址va**，申请一页实际的物理内存，并将这页物理内存的地址存放在该进程页目录映射的二级页表体系中va对应的二级页表项中（当然对应的二级页表也是需要申请物理内存的），并设置权限为perm。

这个函数的副作用（side-effect）是：如果va对应的二级页表项中已经存在了物理页面号+权限位（即，之前已经有了一页物理内存和va建立了映射关系），那么原有的映射关系会被取消并被新申请的映射关系覆盖。

权限方面：PTE_V 是必须的；PTE_COW 是不被允许的（报错 `-E_INVAL` ）

注意：va必须严格低于UTOP（$va < UTOP$）；进程只可以修改他自己的进程空间和直接子进程的进程空间。

> 这个函数的主要功能是分配内存，简单的说，用户程序可以通过这个系统调用给该程序所允许的 虚拟内存 空间内存**显式地**分配实际的物理内存。在我们程序员的视角而言，是我们编写的程序在内存中申请了一片空间；而对于操作系统内核来说，是一个进程请求将其运行空间中的某段地址 与实际物理内存进行映射，从而可以通过该虚拟页面来对物理内存进行存取访问。
>
> > 这里还是我们上面说的意思：每个进程要建立映射的地址（如二进制文件要加载到的虚拟地址），都是在进程的页目录映射的4G空间下的地址，而不是真正的内核虚拟地址，这些被实际加载到的地方并不是进程页目录视野下的虚拟地址。

> 在set_pgfault_handler()函数中，有如下调用
>
> ```
> // 为当前进程的进程空间的UXSTACKTOP - BY2PG处申请一页新的物理内存，作为exception stack
> if (syscall_mem_alloc(0, UXSTACKTOP - BY2PG, PTE_V | PTE_R) < 0 || ...
> ```
>
> 在fork()函数中，父进程在调用syscall_env_alloc()申请子进程之前，必须先给自己用set_pgfault_handler()函数设置好异常处理栈和异常处理函数

```
int sys_mem_alloc(int sysno, u_int envid, u_int va, u_int perm)
{
    struct Env *env;
    struct Page *ppage;
    int ret = 0;
    if ((perm & PTE_V) == 0)// 权限位检查：PTE_V is required,
        return -E_INVAL;
    // va 必须小于 UTOP，因为无论是在4G内核空间还是在4G进程空间中，UTOP都代表用户空间的顶端
    if (va >= UTOP)
        return -E_INVAL;
    if (perm & PTE_COW) // PTE_COW is not allowed(return -E_INVAL),
        return -E_INVAL;
    // 进程只可以修改他自己的进程空间和直接子进程的进程空间。
    // 所以这里在进行 id<->进程 的对应时，检查位的参数要置1
    ret = envid2env(envid, &env, 1);
    if (ret < 0)
        return ret;
    ret = page_alloc(&ppage);// 申请一页新的物理内存
    if (ret < 0)
        return ret;
    // 将申请的新的一页物理内存和 进程页目录下的va建立映射关系（将对应的新申请的一页物理内存的物理页号以及权限perm填入虚拟地址va对应的二级页表项）
```

```
    ret = page_insert(env->env_pgdir, ppage, va, perm);
    if (ret < 0)
        return ret;
    return 0;
}
```

syscall_mem_map()

```
int syscall_mem_map(u_int srcid, u_int srcva, u_int dstid, u_int dstva, u_int perm)
{
    return msyscall(SYS_mem_map, srcid, srcva, dstid, dstva, perm);
}
```

sys_mem_map() - 增加了值得商榷的内容

这个函数的参数很多，但是意义很直接：将源进程地址空间中的 相应内存 映射到目标进程的地址空间 的 相应虚拟内存中去。换句话说，此时两者共享一页物理内存。

那么具体的逻辑为：首先找到需要操作的两个进程，其次获取源进程的虚拟地址所在的虚拟页面对应的实际物理页面，最后将该物理页面与目标进程的相应地址完成映射即可。

```
/* Overview:
 *  Map the page of memory at 'srcva' in srcid's address space
 * at 'dstva' in dstid's address space with permission 'perm'.
 * Perm has the same restrictions as in sys_mem_alloc.
 * (Probably we should add a restriction that you can't go from
 * non-writable to writable?)这段关于权限的描述似乎有问题
 *
 * Post-Condition:
 *  Return 0 on success, < 0 on error.
 * Note:
 *  Cannot access pages above UTOP.
 */
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;      // 源进程
    struct Env *dstenv;      // 目标进程
    struct Page *ppage;      // 源进程的源进程虚拟地址 srcva 对应的二级页表项中存放的物理页面对应
的物理内存控制块（这里注意区分，进程虚拟地址不等于内核虚拟地址，因此不能说虚拟地址对应的物理地址，这是错
误的）
    Pte *ppte;

    ppage = NULL;
    ret = 0;
    round_srcva = ROUNDDOWN(srcva, BY2PG);  //将地址进行页对齐向下取整
    round_dstva = ROUNDDOWN(dstva, BY2PG);

    if ((perm & PTE_V) == 0)     // 检查要设置的权限是否含有PTE_V，应该是要含有才行
        return -E_INVAL;
    if (srcva >= UTOP || dstva >= UTOP)
        return -E_INVAL;
    if ((ret = envid2env(srcid, &srcenv, 0)) < 0)
        return ret;
    if ((ret = envid2env(dstid, &dstenv, 0)) < 0)
        return ret;
    // 找到源进程的进程空间中的srcva所对应的二级页表项中存放的物理页号对应的内存控制块
    // 并将Pte *ppte赋值为指向该二级页表项的指针
    ppage = page_lookup(srcenv->env_pgdir, round_srcva, &ppte);
    if (ppage == NULL)
        return -E_INVAL;
    // 根据注释所说：这里要防止把只读 改成 可写（ 这个if可以不写）
    if ( ((*ppte & PTE_R) == 0) && ((perm & PTE_R) != 0) )
        return -E_INVAL;
```

```
    // 在目标进程的进程空间中的dstva所对应的二级页表项中写入该物理页号，设置权限perm
    ret = page_insert(dstenv->env_pgdir, ppage, round_dstva, perm);
    return ret;
}
```

syscall_mem_unmap()

```
int syscall_mem_unmap(u_int envid, u_int va)
{
    return msyscall(SYS_mem_unmap, envid, va, 0, 0, 0);
}
```

sys_mem_unmap()

这个系统的功能是解除id为envid的进程的进程地址空间中虚拟地址 va 对应的虚拟内存和物理内存之间的映射关系。（主要是将进程页目录的二级页表体系中va对应的二级页表项清零，顺便减少物理页面引用次数，看情况释放物理页面。做完这些后更新TLB。）

```
/* Overview:
 *  Unmap the page of memory at 'va' in the address space of 'envid'
 * (if no page is mapped, the function silently succeeds)
 *
 * Post-Condition:
 *  Return 0 on success, < 0 on error.
 *
 * Cannot unmap pages above UTOP.
 */
int sys_mem_unmap(int sysno, u_int envid, u_int va)
{
    int ret;
    struct Env *env;

    if (va >= UTOP)
        return -E_INVAL;
    ret = envid2env(envid, &env, 0);
    if (ret < 0)
        return ret;
    page_remove(env->env_pgdir, va);
    return ret;
    //  panic("sys_mem_unmap not implemented");
}
```

syscall_yield()

```
void syscall_yield(void)
{
    msyscall(SYS_yield, 0, 0, 0, 0, 0);
}
```

sys_yield()

这个函数的功能主要是实现用户进程对CPU的放弃， 从而调度其他的进程。可以利用我们之前已经编写好的函数 `sched_yield` ， 另外为了通过我们之前编写的进程切换机制保存现场，这里需要先在KERNEL_SP和TIMESTACK上做一点准备工作。

思考一下在保存进程现场时二者的区别———时钟中断时保存在TIMESTACK；系统调用时保存在KERNEL_SP；而env_run时，默认要运行的进程的环境是保存在TIMESTACK的。

> 回忆一下相关代码：

```
if (curenv != NULL) { // 当前运行进程不为空，保存当前进程上下文
struct Trapframe *old = (struct Trapframe*)( TIMESTACK-sizeof(struct Trapframe) );
bcopy((void *) old, (void *)(&(curenv->env_tf)), sizeof(struct Trapframe));
// 在发生进程调度，或当陷入内核时，会将当时的进程上下文环境保存在env_tf变量中。
curenv->env_tf.pc = curenv->env_tf.cp0_epc;
}
```

可以看出，env_run()是直接将环境从$[TIMESTACK-sizeof(structTrapframe),TIMESTACK-1]$中取出，复制到当前进程的环境结构体中的。

所以在 `sys_yield` 函数的最开始，我们就需要把要放弃CPU的进程的上下文环境从 `KERNEL_SP - sizeof(struct Trapframe)` 拷贝到 `TIMESTACK - sizeof(struct Trapframe)` 。然后再调用 `sched_yield` 函数。

```
void sys_yield(void)
{
    bcopy((void *)KERNEL_SP - sizeof(struct Trapframe),
          (void *)TIMESTACK - sizeof(struct Trapframe),
          sizeof(struct Trapframe));
    sched_yield();
    // sched_yield()函数中调用env_run()，当前curenv的环境被从TIMESTACK复制到自己的env_tf，
    // 新选出的进程将自己的env_tf存放的上下文信息复制到CPU，开始运行
    // 这样就表现为当前进程“放弃CPU”
}
```

$Part2.2$ 进程间通信机制(IPC)

IPC机制远远没有我们想象得那样神秘，特别是在我们这个被极度简化了的MOS操作系统中。IPC机制的实现使得我们系统中的 进程之间拥有了相互传递消息的能力，为后续实现fork、文件系统服务、管道与shell等均有着极大的帮助。 根据之前的讨论，我们能够确定的是:

- IPC的目的是使两个进程之间可以通讯
- IPC需要通过系统调用来实现
- IPC还与进程的数据、页面等信息有关

所谓通信，最直观的一种理解就是交换数据。假如我们能够将让一个进程有能力将数据传递给另一个进程， 那么进程之间自然具有了相互通讯的能力。但是，要实现交换数据，我们所面临的最大的问题是什么呢？ 没错，问题就在于**各个进程的地址空间是相互独立的**。每个进程都有各自的地址空间，这些地址空间之间是相互独立的，同一个虚拟地址但却可能在不同进程下对应不同的物理页面，自然 对应的值就不同。因此，要想传递数据， 我们就需要想办法**把一个地址空间中的东西传给另一个地址空间**。

想要让两个完全独立的地址空间之间发生联系，最好的方式是去找它们是否存在共享的部分。 虽然地址空间本身独立，但是有些地址也许被映射到了同一物理内存上。

在 `env_setup_vm` 这个函数里面可以发现，所有的进程都没有涉足进程空间中**内核所在的高2G空间**。在每个进程的页表中，高512个一级页表项是没有设置的。 因此，可以看成：对于任意的进程空间，高2G都是一样的。想要在不同空间之间交换数据，我们就可以借助于内核空间的高2G空间来实现。也就是说，发送方进程可以将数据以系统调用的形式存放在内核空间中，接收方进程同样以系统调用的方式在内核找到对应的数据，读取并返回。

那么，我们把要传递的消息放在和进程最相关的地方——进程控制块！

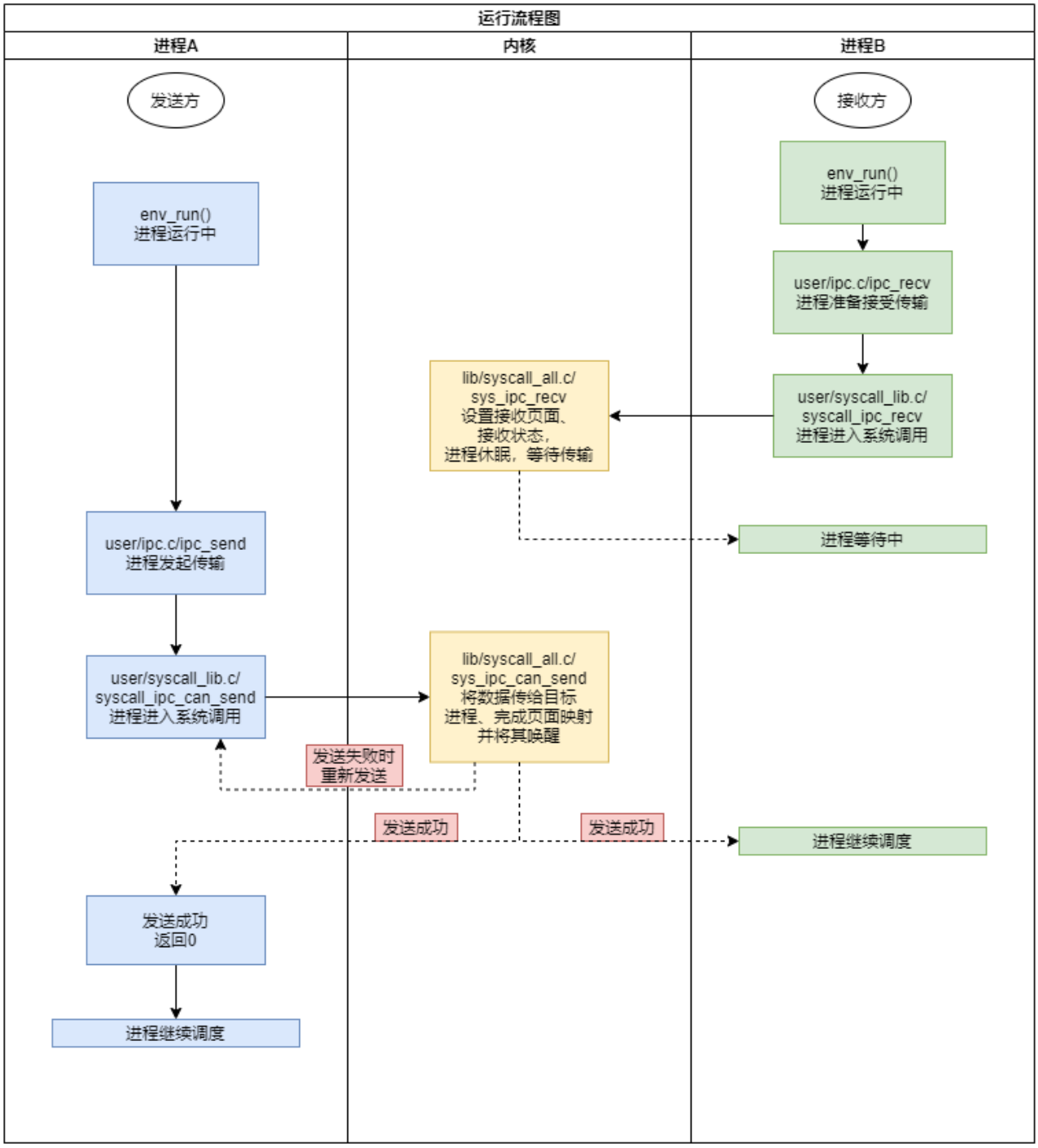
进程控制块在lab4要用到的东西

```
struct Env {
    // Lab 4 IPC
    u_int env_ipc_value;            // data value sent to us
    u_int env_ipc_from;             // envid of the sender
    u_int env_ipc_recving;          // env is blocked receiving
    u_int env_ipc_dstva;            // va at which to map received page
    u_int env_ipc_perm;             // perm of page mapping received
    // Lab 4 fault handling
    u_int env_pgfault_handler;      // page fault state
    u_int env_xstacktop;            // top of exception stack
};
```

- 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。**1：等待接受数据中；0：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**

IPC的大致流程

IPC的大致流程总结如下图所示：



值得一提的是，由于在我们的用户程序中，会大量使用srcva为0的调用来表示只传value值，而不需要传递物理页面， 换句话说，当srcva不为0时，我们才建立两个进程的页面映射关系。因此在编写相关函数时也需要注意此种情况。

下面我们来梳理一下：

任意一个进程的运行流程和时间片机制

首先，进程运行是基于时间片机制的，而时间片机制，说白了就是由模拟的实时钟（具体表现为 `set_timer` 汇编函数），通过一定的频率，持续产生4号中断（时钟中断）。由于中断是异常的一种，只要触发异常，就会进入内核态，并触发异常处理机制（MIPS 将PC 指向0x80000080），系统跳转到异常分发程序后，判定为0号异常（0号异常就是各种中断），于是交由 `handle_int` 汇编函数处理（全名就是 `handle_interrupt`）。

`handle_int` 汇编函数中，调用了SAVE_ALL汇编函数，根据不同情况判断，得到不同的内核异常处理栈的地址，并且将被中断 打断的当前进程的环境存放在了内核异常处理栈。由于当前系统中只会出现时钟中断，那么上述过程是确定的：将被**时钟中断**打断的**当前进程**的上下文环境存放在 `TIMESTACK-sizeof(struct Trapframe)` 位置。

接下来 `handle_int` 汇编函数会判断是不是4号中断位引发中断，是则调用timer_ irq汇编宏函数，而 `timer_ irq` 汇编函数就是在设置响应时钟中断后，调用了 `sched_yield()` 函数。 `sched_yield()` 函数会在调度队列中选择一个合适的进程，交由 `env_run()` 函数将其运行；而 `env_run()` 函数则会调用 `env_tf_pop` 汇编函数，将用户现场中记录的所有寄存器的值恢复到CPU寄存器中。然后自己再将用户现场中的EPC存入k0寄存器，将sp恢复为用户态指针。再**通过R3000标准回滚指令序列 `jr k0 + rfe` 恢复用户态**。

此时，系统在 `jr k0` 后（k0存放进程PC值）来到了进程的PC值处（结合env_run()函数可以看出这里PC就是EPC），**在用户态下真正运行进程一个时间片.**

注意，在 `timer_irq` 中， `ret_from_exception` 汇编函数并没有执行。

无论是否切换进程， `env_run()` 函数都会执行下面的代码：

将存放在 `TIMESTACK-sizeof(struct Trapframe)` 位置的 被时钟中断打断的当前进程的上下文环境 取出，放回进程自己的环境结构体env_tf中。

注意：进程的环境结构体真实存放的位置是 作为进程控制块的一部分，存放在kseg0中（即在最开始分配出去的两大块内存）。

```
if (curenv != NULL) { // 当前运行进程不为空，保存当前进程上下文
    struct Trapframe *old = (struct Trapframe*)( TIMESTACK-sizeof(struct Trapframe) );
    bcopy((void *) old, (void *)(&(curenv->env_tf)), sizeof(struct Trapframe));
    // 在发生进程调度，或当陷入内核时，会将当时的进程上下文环境保存在env_tf变量中。
    curenv->env_tf.pc = curenv->env_tf.cp0_epc;
}
curenv = e;
```

而只要不满足切换进程的条件，那么在 `sched_yield()` 函数中，不会选择新的进程用于顶替，在 `sched_yield()` 函数中调用的 `env_run()` 函数也不会进行进程切换。

那么上面这段代码中，由于 `curenv` 和传入的即将要运行的进程e是同一个进程，那么其实就是：

接着，再通过env_pop_tf函数，将位于**上一个时间片运行的进程curenv**的环境结构体env_tf中的上下文环境，放到CPU环境中，进程正式**继续运行**一个时间片。

而进程切换的契机是：

当前正在运行的进程 `curenv` 的时间片消耗完毕（在 `sched_yield()` 函数中的静态局部变量 `count == 0` 时）；或者是当前正在运行的进程 `curenv` 的状态在当前运行的一个时间片中被改为了 `ENV_NOTRUNNABLE` 。无论是哪一种，这样都会在下一次 `sched_yield()` 函数的检查中，触发切换机制，选择一个新的状态为 `ENV_RUNNABLE` 的进程，执行 `env_run()` 函数。

此时，上面这段代码就会使得 `curenv` 代表的当前进程的环境切换为即将要运行的进程e。

接着，再通过env_pop_tf函数，将位于**即将要运行的进程e**的环境结构体env_tf中的上下文环境，放到CPU环境中，进程正式**开始运行**一个时间片。那么，就表现为，上一个时间片运行的进程“放弃”了CPU环境。

信息接收方进程的流程

ipc_recv()

用于接受一个value值，并将接收到的value作为函数返回值，**并将信息发送者的envid存储在*whom中**；将此次信息传递的信息权限存储在*perm中。

```
// 函数中的env是外部变量，定义于 libos.h 里
// Receive a value.  Return the value and store the caller's envid in *whom.
// Hint: use env to discover the value and who sent it.
u_int ipc_recv(u_int *whom, u_int dstva, u_int *perm)
{
    //printf("ipc_recv:come 0\n");
    syscall_ipc_recv(dstva);
    if (whom) {
        *whom = env->env_ipc_from;  // 信息发送方进程的envid
    }
```

```
    if (perm) {
        *perm = env->env_ipc_perm;  // 信息的权限
    }
    return env->env_ipc_value;  // 信息的值
}
```

syscall_ipc_recv()

```
void syscall_ipc_recv(u_int dstva)
{
    msyscall(SYS_ipc_recv, dstva, 0, 0, 0, 0);
}
```

sys_ipc_recv()

函数用于接受消息。在该函数中：

1. 首先要将当前进程自身的env_ipc_recving设置为1，表明当前进程准备接受发送方的消息
2. 之后给env_ipc_dstva赋值，表明自己要将接受到的页面与dstva完成映射
3. 阻塞当前进程，即把当前进程的状态置为不可运行（ENV_NOT_RUNNABLE）
4. 最后放弃CPU（调用 `sys_yield()` 重新进行调度），安心等待发送方将数据发送过来

```
void sys_ipc_recv(int sysno, u_int dstva)
{
    if (dstva >= UTOP)
        return;
    curenv->env_ipc_recving = 1;         // 状态调整为正在等待接收信息
    curenv->env_ipc_dstva = dstva;
    curenv->env_status = ENV_NOT_RUNNABLE;
    // 这里设置当前进程curenv的状态为 ENV_NOT_RUNNABLE 以后，调度函数在运行时
    // 一定会找到一个新的、状态为 ENV_RUNNABLE的进程来执行，这样curenv就会暂停
    sys_yield();
    // sys_yield()函数中调用sched_yield，后者又调用env_run()，当前curenv的环境被从TIMESTACK复制到自己的env_tf，
    // 新选出的进程将自己的env_tf存放的上下文信息复制到CPU，开始运行
    // 这样就表现为当前进程“放弃CPU”
}
```

此时，信息接收方的进程被阻塞，放弃CPU，等待着信息的发送。

信息发送方进程的流程

ipc_send()

这个函数**向whom代表的进程发送信息value**，并且一直在尝试，直到成功。其中srcva是发送进程对应的进程空间中的虚拟地址。

> 一个应用的例子：
>
> ```
> // 当前进程向自己的子进程发送信息0，映射的地址是当前进程空间的0（srcva==0），说明只需要传入value，不需要传递物理页面。信息要设置的权限也是0
> writef("\n@@@@@send 0 from %x to %x\n", syscall_getenvid(), who);
> ipc_send(who, 0, 0, 0);
> ```

除了-E_IPC_NOT_RECV之外，任何错误都会导致panic()。

```
// Send val to whom.  This function keeps trying until
// it succeeds.  It should panic() on any error other than
// -E_IPC_NOT_RECV.
// Hint: use syscall_yield() to be CPU-friendly.
void ipc_send(u_int whom, u_int val, u_int srcva, u_int perm)
{
```

```
    int r;
    while ((r = syscall_ipc_can_send(whom, val, srcva, perm)) == -E_IPC_NOT_RECV) {
        // 结合sys_ipc_can_send中会报错-E_IPC_NOT_RECV的情形，这里应该是
        // 接受信息 的进程e 还没有进入等待状态，e->env_ipc_recving == 0
        syscall_yield();
        //writef("QQ");
    }
    if (r == 0) {
        return;
    }
    user_panic("error in ipc_send: %d", r);
}
```

syscall_ipc_can_send()

```
int
syscall_ipc_can_send(u_int envid, u_int value, u_int srcva, u_int perm)
{
    return msyscall(SYS_ipc_can_send, envid, value, srcva, perm, 0);
}
```

sys_ipc_can_send()

> 值得一提的是，由于在我们的用户程序中，会大量使用srcva为0的调用来表示只传value值，而不需要传递物理页面， 换句话说，当srcva不为0时，我们才建立两个进程的页面映射关系。因此在编写相关函数时也需要注意此种情况。

这个函数用于向参数envid代表的进程发送消息value：

1. 根据envid找到相应进程，如果指定进程为可接收状态(考虑env_ipc_recving)，则发送成功
2. 否则，函数返回-E_IPC_NOT_RECV，表示目标进程未处于接受状态
3. 清除接收进程的接收状态，将相应数据填入进程控制块，传递物理页面的映射关系
4. 修改进程控制块中的进程状态，使接受数据的进程可继续运行(ENV_RUNNABLE)

```
int sys_ipc_can_send(int sysno, u_int envid, u_int value, u_int srcva,
                     u_int perm)
// 参数解读：envid是要接受信息的进程id，value的信息值，srcva是发送信息需要与自身的哪个虚拟页面完成映射，perm是传递的页面的权限位设置
{
    int r;
    struct Env *e;
    struct Page *p;

    if (srcva >= UTOP)
        return -E_INVAL;
    r = envid2env(envid, &e, 0);     // 找到要接受信息的进程e
    if (r < 0)
        return r;
    if (e->env_ipc_recving == 0)
        return -E_IPC_NOT_RECV;
    e->env_ipc_value = value;            // 接受方接收信息value
    e->env_ipc_from = curenv->env_id;    // 记录发送方的进程id，也就是调用发送函数的当前进程
    e->env_ipc_perm = perm;              // 传递的页面的权限位设置
    e->env_ipc_recving = 0;              // 清除接收方进程的接收状态
    e->env_status = ENV_RUNNABLE;        // 接收方进程的状态设置为可以继续运行
    /* srcva不为0的时候，我们才需要建立两个进程之间的映射关系 */
    if (srcva != 0)
    {// 找到当前进程的页目录映射的二级页表体系中 srcva 对应的二级页表项存放的物理页面所对应的物理内存控制块p。
        p = page_lookup(curenv->env_pgdir, srcva, NULL);
        if (p == NULL || e->env_ipc_dstva >= UTOP)
            return -E_INVAL;
        // 将接受信息的进程e的记录“接收到的页面需要与自身的哪个虚拟页面完成映射”的 e->env_ipc_dstva 所对应的接受信息的进程e中页目录映射的二级页表体系中的二级页表项 写入p对应的物理页号。注意这里如果原来有映射关系的话，这里会覆盖原有映射关系（副作用）
        r = page_insert(e->env_pgdir, p, e->env_ipc_dstva, perm);
        if (r)
```

```
            return r;
    }
    return 0;
}
```

$Part2.3$ fork机制

fork总览

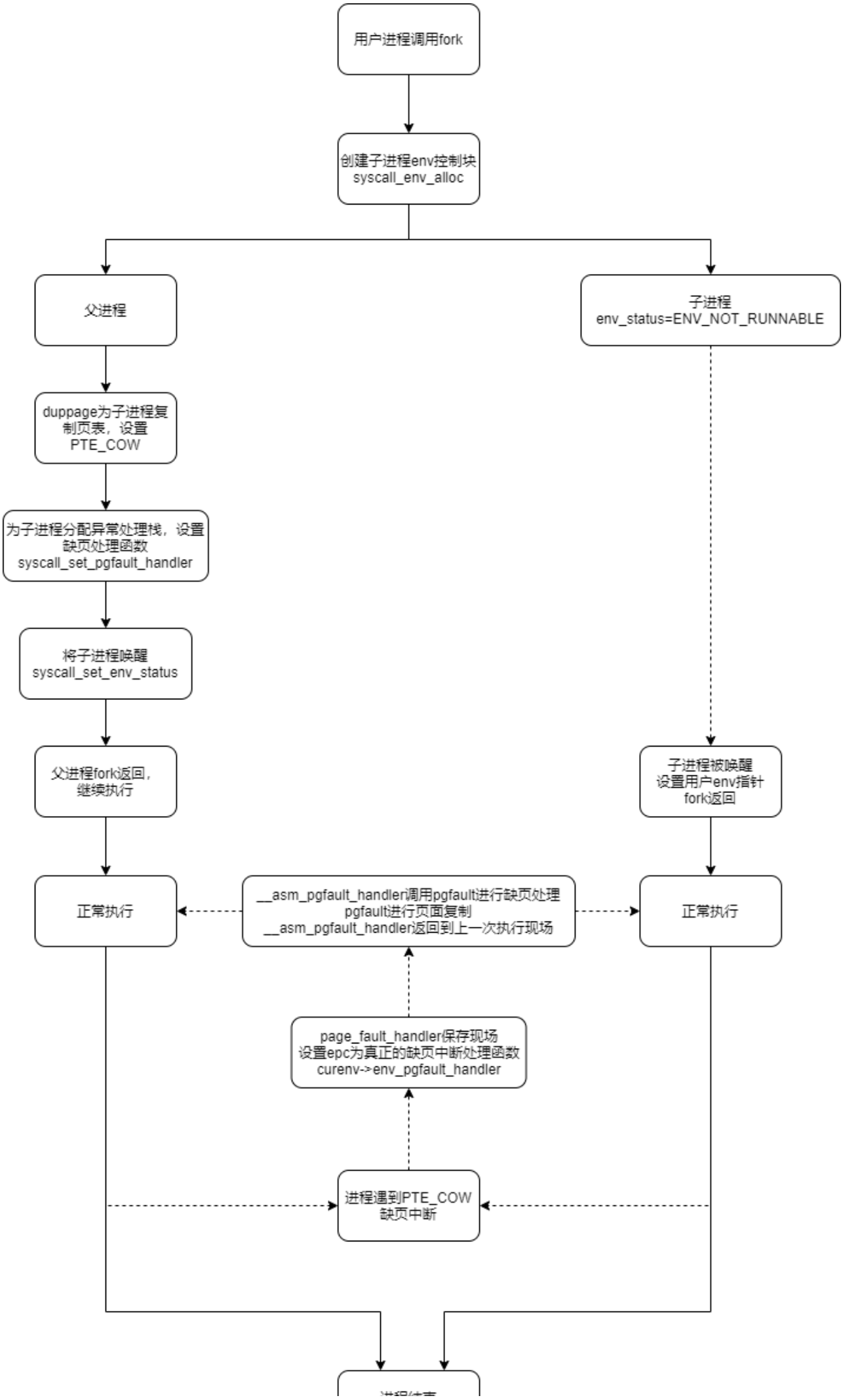
在 lab3 我们曾提到过，内核通过 `env_alloc` 函数创建一个进程。但如果要让一个进程创建一个进程， 就像是父亲与孩子那样，我们就需要基于系统调用，引入 fork 机制。

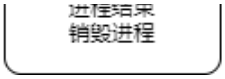
fork，直观意象是叉子的意思，而在操作系统中更像是分叉的意思，就好像一条河流动着，遇到一个分叉口，分成两条河一样，fork就是那个分叉口。在操作系统中，一个进程在调用 fork() 函数后，将从此分叉成为两个进程运行，其中新产生的进程称为原进程的**子进程**。

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，包括程序镜像、通用寄存器和程序计数器 PC 等。在新的进程中，这一 fork() 调用的返回值为 0，而在旧进程，也就是所谓的父进程中，同一调用的返回值是子进程的进程 ID（MOS 中的 env_id），且一定大于 0。 fork 在父子进程中产生不同返回值这一特性，让我们能够在代码中调用 fork 后判断当前在父进程还是子进程中，以执行不同的后续逻辑，也使父进程能够与子进程进行通信。

本实验中 MOS 系统的 fork 函数流程大致如下图所示，其中的大部分函数也是这次本次实验的任务，会在后续详细介绍。请注意，图中提到的缺页中断不是虚拟内存管理中涉及的**页缺失异常**，而是指本节后续将介绍的**页写入异常**。

下面是fork总的流程


用户进程调用fork
创建子进程env控制块
syscall_env_alloc
父进程
子进程
env_status=ENV_NOT_RUNNABLE
duppage为子进程复制页表，设置PTE_COW
为子进程分配异常处理栈，设置缺页处理函数
syscall_set_pgfault_handler
将子进程唤醒
syscall_set_env_status
父进程fork返回，继续执行
子进程被唤醒
设置用户env指针
fork返回
正常执行
__asm_pgfault_handler调用pgfault进行缺页处理
pgfault进行页面复制
__asm_pgfault_handler返回到上一次执行现场
正常执行
page_fault_handler保存现场
设置epc为真正的缺页中断处理函数
curenv->env_pgfault_handler
进程遇到PTE_COW
缺页中断



写时复制机制

在 fork 时，操作系统会为新进程分配独立的虚拟地址空间，但分配独立的地址空间并不意味 着一定会分配额外的物理内存。实际上，刚创建好的子进程使用的仍然是其父进程使用的物理内存，子进程地址空间中的代码段、数据段、堆栈等都被映射到 父进程中相同区段对应的页面，这也是子进程能“复刻”父进程的状态往后执行的一个原因。也就是说，虽然两者的地址空间（页目录和页表）是不同的，但是它们此时还对应相同的物理内存。

既然父子进程需要独立并发运行，而现在又说共享物理内存，这不是矛盾吗？按照共享物理内存的说法，父子进程执行不同逻辑时对相同的内存进行读写，岂不是会造成数据冲突？

这两种说法实际上不矛盾，因为**父子进程共享物理内存是有前提条件的：共享的物理内存不会被任一进程修改。**

那么，对于那些父进程或子进程修改的内存我们又该如何处理呢？ 这里我们引入一个新的概念——写时复制（Copy On Write，简称 COW)。COW 类似于一种对虚拟页的保护机制，通俗来讲就是：在 fork 后的父子进程中有**修改**内存（一般是数据段或栈）的行为发生时，内核会捕获到一种**页写入异常**，并在异常处理时为**修改内存的进程的地址空间中相应地址 分配新的物理页面**。

> 这里记录一下我的理解：也就是，本来进程A和进程B共享一页物理内存P，A和B的进程空间中对应的虚拟地址的二级页表项的内容都是P的物理页号。现在B想对P进行写入修改，那么此时就需要新分配一页物理内存P′，P′复制了所有P的信息，然后将B的进程空间中原来和P对应的虚拟地址的二级页表项的内容修改为P‘的物理页号。此后A和B相当于分别与P和P′建立了映射，进行不同的写入修改时就不会发生冲突了。

一般来说，子进程的代码段仍会共享父进程的物理空间，两者的程序镜像也完全相同。在这样的保护下，用户程序可以在行为上认为 fork 时父进程中内存的状态被完整复制到了子进程中，此后父子进程可以独立操作各自的内存。

> 这里记录一下我的理解：所谓“复制”：也就是最终父进程和子进程拥有两份一样的代码段、数据段、堆栈等，而不是共享父进程的代码段、数据段、堆栈等。当然，这里只能是作为理解，事实上，如果真的全部复制一份进程，开销很大。MOS并没有这么做。只是以一个类似“懒汉”的方式，在标记了PTE_COW位的页面被进程写入的时候，才触发页写入异常，进行页面复制。

在我们的 MOS 操作系统实验中，进程调用 fork 时，其所有的可写入的内存页面，都需要通过设置页表项标志位 PTE_COW 的方式被保护起来。 无论父进程还是子进程何时试图写一个被保护的页面，都会产生一个页写入异常，而在其处理函数 `pgfault()` 中，操作系统会进行**写时复制**，把该页面重新映射到一个新分配的物理页中，并将原物理页中的内容复制过来，同时取消虚拟页的这一标志位。

fork的父子进程的两个返回值

首先要明确一点：**fork 只在父进程中被调用了一次，在父子两个进程中各产生一个返回值。**

我们前面是提到了子进程执行 fork 之后的代码，实则不准确：因为在 fork 内部， 就要用 `sys_env_alloc` 的两个返回值区分开父子进程，好安排他们在返回之后执行不同的任务。这是因为：虽然子进程在被创建出来就已经有了 进程控制块和进程上下文，但是**子进程是否能够开始被调度是要由父进程决定的。**

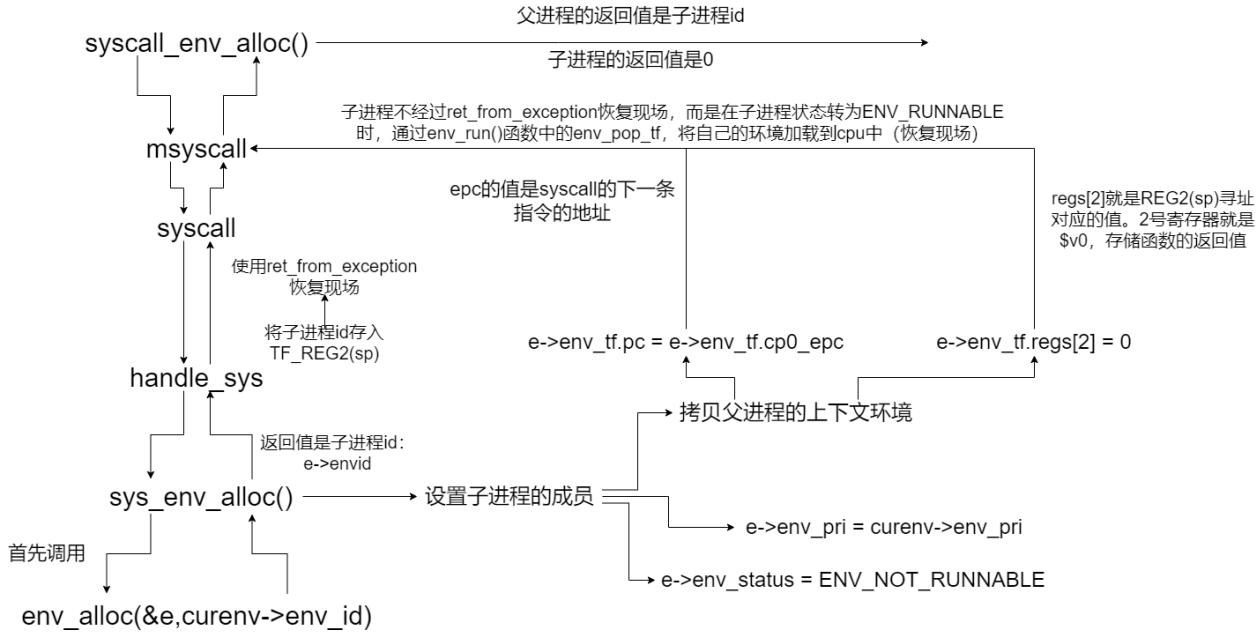
在我们的MOS操作系统实验中，需要强调的一点是我们实现的fork是一个用户态函数，fork函数中需要若干个“原子的”系统调用来完成所期望的功能。其中最核心的一个系统调用就是新进程的创建 `syscall_env_alloc` 。

在fork的实现中，我们是通过判断 `syscall_env_alloc` 的返回值来决定fork的返回值以及后续动作，所以会有类似这样结构的代码：

```
envid = syscall_env_alloc();
if (envid == 0) {
    // 子进程
    ...
}
else {
    // 父进程
    ...
}
```

既然fork的目的是使得父子进程处于几乎相同的运行状态，我们可以认为在返回用户态时，父子进程应该经历了同样的恢复运行现场的过程，只不过对于父进程是**从系统调用中返回时**恢复现场， 而对于子进程则是在**进程被调度时**恢复现场。在现场恢复后，父子进程都会从内核返回到 msyscall 函数中，而它们的现场中存储的返回值（即 v0 寄存器的值）是不同的。这一返回值随后再被返回到 `syscall_env_alloc` 和 fork 函数，使 fork 函数也能区分两者。

要记住，fork()函数中，父进程和子进程的恢复现场不是同步的，是异步事件



syscall_env_alloc()

`syscall_env_alloc()` 函数就是内核函数 `sys_env_alloc()` 函数的用户态对应版。

```
inline static int syscall_env_alloc(void)
{
    return msyscall(SYS_env_alloc, 0, 0, 0, 0, 0);
}
```

###

sys_env_alloc() 创建子进程控制块

为了实现上面说到的这一特性，在 `sys_env_alloc` 分配一个新的进程控制块后，还需要用一些当前进程的信息作为模版来填充这个控制块：

运行现场

要复制一份当前进程的运行现场（进程上下文）Trapframe 到子进程的进程控制块中。

程序计数器

子进程的现场中的程序计数器（PC）应该被设置为从内核态返回后的地址，也就是使它陷入异常的 syscall 指令的后一条指令的地址。由于我们之前完成的任务，这个值已经保存于 Trapframe 中。（具体来说是 e->env_tf.cp0_epc ）

返回值有关

这个系统调用本身是需要一个返回值的，我们希望系统调用在内核态返回的 envid 只传递给父进程，对于子进程则需要对它的保存的现场Trapframe进行一个修改，从而在恢复现场时用 0 覆盖系统调用原来的返回值。

进程状态

我们当然不能让子进程在父进程的 `syscall_env_alloc` 返回后就直接被调度，因为这时候它还没有做好充分的准备，所以我们需要避免它被加入调度队列。

其他信息

观察Env结构体的结构，思考下还有哪些字段需要进行初始化，这些字段的初始值应该是继承自父进程还是使用新的值，如果这些字段没有初始化会有什么后果（提示：env_pri）。

那么，总的来说，这个函数就是为当前运行进程curenv申请一个新的子进程，并将父进程curenv的运行 上下文环境 拷贝到子进程的 进程控制块中的环境结构体中。然后，将子进程状态设置为 `ENV_NOT_RUNNABLE` ，pc设置为cp0_epc，存放2号寄存器的位置设为0（即函数返回值设为0）

```
int sys_env_alloc(void)
{
    int r;
    struct Env *e;
    r = env_alloc(&e, curenv->env_id);  // 为当前的进程curenv申请一个子进程
    if (r < 0)
        return r;
    // 子进程是否能够开始被调度是要由父进程决定的
    e->env_status = ENV_NOT_RUNNABLE;
    // 子进程的时间片和父进程保持一致
    e->env_pri = curenv->env_pri;

    // 将父进程的运行 上下文环境 拷贝到子进程的 进程控制块中的环境结构体中
    bcopy((void *)KERNEL_SP - sizeof(struct Trapframe),
          (void *)(&(e->env_tf)), sizeof(struct Trapframe));
    e->env_tf.pc = e->env_tf.cp0_epc;
    e->env_tf.regs[2] = 0; // $v0 <- return value

    return e->env_id; // 返回子进程的envid
    //  panic("sys_env_alloc not implemented");
}
```

画龙点睛的一笔：在sys_env_alloc()函数中的e->env_tf.regs[2] = 0

这里 将为当前进程 `curenv` 申请到的子进程 `e` 的上下文环境结构体中的 `e->env_tf.regs[2]` 置零，从而在 `ret_from_exception` 进行恢复现场和回滚的时候，用0覆盖了原本的v0寄存器的值（v0寄存器就是2号寄存器，存放函数返回值）。

syscall_getenvid()

```
u_int syscall_getenvid(void)
{
    return msyscall(SYS_getenvid, 0, 0, 0, 0, 0);
}
```

sys_getenvid()

获取当前运行进程的envid。

```
u_int sys_getenvid(void)
{
    return curenv->env_id;
}
```

sys_env_destory()

销毁参数 envid 对应的进程。

```
int sys_env_destroy(int sysno, u_int envid)
{
    /*
        printf("[%08x] exiting gracefully\n", curenv->env_id);
        env_destroy(curenv);
    */
    int r;
    struct Env *e;
    if ((r = envid2env(envid, &e, 1)) < 0) {
        return r;
    }
    printf("[%08x] destroying %08x\n", curenv->env_id, e->env_id);
    env_destroy(e);
    return 0;
}
```

父子进程各自的旅途

首先要了解user/libos.c文件，包括exit()和libmain()函数

这个文件是用户进程入口的 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文件之一。

```
struct Env *env;// libos.c中的全局变量
void exit(void)
{
    //close_all();
    syscall_env_destroy(0);
}

void libmain(int argc, char **argv)
{
    // set env to point at our env structure in envs[].
    env = 0;
    //writef("xxxxxxxxx %x  %x  xxxxxxxxx\n",argc,(int)argv);
    int envid;
    envid = syscall_getenvid(); // 得到当前进程的envid
    envid = ENVX(envid);        // 得到当前进程envid的低10位，即idx部分
    env = &envs[envid];         // 拿到当前进程对应的进程控制块
    // env成为了指向当前进程的进程控制块的指针
    // call user main routine
    umain(argc, argv);
    // exit gracefully
    exit();
    //syscall_env_destroy(0);
}
```

这里的umain主要实现于各个用于本地测试进程的c文件，如：uesr/fktest.c，user/pingpong.c，user/tltest.c等。这里暂时不表。

结合上面对libmain的分析，我们就明白了为什么指导书会这么说：

MOS 允许进程访问自身的进程控制块，而在 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。

对于 fork 后的子进程，它具有了一个与父亲不同的进程控制块，因此在子进程第一次被调度的时候（当然这时还是在fork函数中）需要对 env 指针进行更新，使其仍指向当前进程的控制块。这一更新过程与运行时入口对 env 指针的初始化过程相同，具体步骤如下：

1. 通过一个系统调用来取得自己的envid，因为对于子进程而言 `syscall_env_alloc` 返回的是一个0值。

2. 根据获得的envid，计算对应的进程控制块的下标，将对应的进程控制块的指针赋给 env。

> 指导书这段话实际上对应着fork()函数中的这段代码
>
> ```
> if (newenvid == 0) // 说明是子进程
> {
>  env = envs + ENVX(syscall_getenvid());
>  return 0;
> }
> ```

做完上面步骤，当子进程醒来时，就可以从fork函数中正常返回，开始自己的旅途了。

当然只完成子进程部分，子进程还不能正常跑起来，因为父进程在子进程醒来之前还需要做更多的准备，这些准备中最重要的一步是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，这需要我们遍历父进程的**大部分用户空间页**，并使用将要实现的 `duppage` 函数来完成这一过程。 `duppage` 时，对于可以写入的页面的页表项，**在父进程和子进程都需要**加以PTE_COW标志位保护起来。

duppage()

该函数将当前进程的进程空间的虚拟页号 pn（对应进程空间虚拟地址为pn*BY2PG）映射到envid 所代表的进程的进程空间视野下的相同虚拟地址处。

它实现“当前进程”的方式是：在调用 `sys_mem_map()` 函数时，令参数srcid=0，甚至dstid=0。这样在 `sys_mem_map()` 函数内部调用 `envid2env(srcid, &srcenv, 0)` 时，就会由于envi2env()函数的约定：在envid为0时直接将当前进程的指针存储进参数。

在 `duppage` 函数中，需要强调的一点是，要对具有不同权限位的页使用不同的方式进行处理。你可能会遇到这几种情况：

只读页面

对于不具有 PTE_R 权限位的页面，按照相同权限（只读）映射给子进程即可。

写时复制页面

即具有 PTE_COW 权限位的页面。这类页面是之前的 fork 时 `duppage` 的结果，且在本次 fork 前必然未被写入过。

可写且共享页面

即同时具有 PTE_R 权限位和 PTE_LIBRARY 权限位的页面。这类页面需要保持共享可写的状态，即在父子进程中映射到相同的物理页，使对其进行修改的结果相互可见。在文件系统部分的实验中，我们会使用到这样的页面。

可写非共享页面

即具有 PTE_R 权限位，且不符合以上特殊情况的页面（其实也就是具有 PTE_R 权限位且不具有 PTE_LIBRARY 权限位）。这类页面需要在父进程和子进程的页表项中都使用 PTE_COW 权限位进行保护。

```
static void duppage(u_int envid, u_int pn)
{
    u_int addr = pn << PGSHIFT; // 取第 pn 个用户空间虚拟页号对应的虚拟地址
    u_int perm = (*vpt)[pn] & 0xfff;
    // vpt是二级页表区的基地址，这里是取第 pn 个二级页表项的 权限位

    int flag = 0;
    if ((perm & PTE_R) && !(perm & PTE_LIBRARY))// 具有 PTE_R 权限位且非共享
    { // 需要在父进程和子进程的页表项中都使用 PTE_COW 权限位进行保护
        perm |= PTE_COW;
        flag = 1;
    }
    // 将当前进程的虚拟页号pn 对应的虚拟地址 addr 在当前进程的二级页表体系中对应的二级页表项中 存放的物理页号，存入 envid 对应的进程的 进程空间的虚拟地址 addr 在envid对应进程的二级页表体系中 对应的二级页表项中。
    syscall_mem_map(0, addr, envid, addr, perm);
```

```
    // 只要当前页面是 可写非共享页面，就需要 也给父进程的这个页面也加上PTE_COW
    if (flag)
        syscall_mem_map(0, addr, 0, addr, perm);
    //  user_panic("duppage not implemented");
}
```

备注：这里不可以先映射父进程，再映射子进程！——即把 `syscall_mem_map(0, addr, 0, addr, perm)` 直接写在第一个判断（即判断具有PTE_R 权限位且非共享）里面。

这个问题需要借助指导书上的一句话慢慢梳理：

> 由于历史原因，MOS 操作系统的源代码中也使用 page fault 代指这里的页写入异常，但这种异常与之前的缺页中断（页缺失异常）是两种不同的 TLB 异常。具体来说，R3000 的页写入异常会在尝试写入页表项中不带有 dirty bit（表示页面可写入的权限位，即代码中的 PTE_R）的页时产生，而页缺失异常则在尝试访问的页表项中不带有 valid bit（有效位，即 PTE_V）时产生。MOS 操作系统的实现巧妙地利用了一个硬件保留的权限位作为 PTE_COW，并在**内核进行 TLB 重填时将标记为 PTE_COW 的页表项中的 dirty bit 置零**，因此用户程序在处理时可认为这种页写入异常在且仅在写入 PTE_COW 页面时产生。

关于上面部分中的红字部分的实现，在do_refill函数中有如下描述

```
##7. 判定权限位：若权限位显示该表项无效（无 PTE_V），则调用 page_out ，随后回到第一步；（PTE_COW
为写时复制权限位，将在 lab4 中用到，此时将所有页表项该位视为 0 即可）
            nop
                    move        t0,k1    # k1 -> t0
                    and     t0,0x0200    # PTE_V 0x0200，这里就是检查二级页表项的权限位（有效
位）
                    beqz        t0,NOPAGE    # 如果为0，那么跳转到NOPAGE
                                             # 在NOPAGE中跳回 pageout
            nop
            move        k0,k1    # k1 -> k0
            and         k0,0x1   # PTE_COW 0x0001，这里就是检查二级页表项的权限位（写时复制位）
            beqz        k0,NoCOW     # 如果PTE_COW为0，那么跳转到NOCOW
            nop
##8. 将物理地址存入 EntryLo ，并调用 tlbwr 将此时的 EntryHi 与 EntryLo 写入到 TLB 中（EntryHi
中保留了虚拟地址相关信息）
            # 如果PTE_COW不为0，就会执行下面的代码（将可写位PTE_R置零）
            and     k1,0xfffffbff    # 将k1（二级页表项）的第10位（即可写位PTE_R 0x0400）置零
                                     # 即现在的二级页表项是 只读
NoCOW:
            mtc0    k1,CP0_ENTRYLO0 # 将k1寄存器的值（即物理页号+权限位）存入 EntryLo
            nop
            tlbwr                   # 调用 tlbwr 将此时的 EntryHi 与 EntryLo 写入到 TLB 中

            j       2f              # 调转到下面的 2 标签处 f 即 forward
            nop
```

do_refill函数正是内核态下进行TLB重填的函数，它的这段代码正是：**进行 TLB 重填时将标记为 PTE_COW 的页表项中的 dirty bit 置零**。

那么：**如果直接在判断里就给父进程重新映射物理页面，如果父进程里的页设置 COW 之后父进程恰好写入了这一页，并且触发了页缺失中断，然后子进程再被 map 的话，这个 COW 就没有意义了。因为 dirty bit 即 PTE_R 被置零后，谁都无法修改这个页面，那么写时复制的保护机制就无意义了。**

fork() 极度重要

父进程在子进程醒来之前还需要做更多的准备，这些准备中最重要的一步是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，这需要我们遍历父进程的**大部分用户空间页**，并使用将要实现的 `duppage` 函数来完成这一过程。 `duppage` 时，对于可以写入的页面的页表项，**在父进程和子进程都需要**加以PTE_COW标志位保护起来。

```
int fork(void)
```

```
{
    // Your code here.
    u_int newenvid;
    extern struct Env *envs;
    extern struct Env *env;
    u_int i;

    //The parent installs pgfault using set_pgfault_handler
    /* 父进程为自己注册异常处理函数 */
    // 父进程先是在UXSTACKTOP - BY2PG处申请一页新的物理内存，作为异常处理栈（exception stack）
    // 然后将父进程自己的env_pgfault_handler域设为异常处理函数__asm_pgfault_handler，将
env_xstacktop 域设置为 UXSTACKTOP。
    // 在函数的最后，将entry.S中的字 __pgfault_handler 赋值为fn（即pgfault）
    set_pgfault_handler(pgfault);

    //alloc a new alloc
    newenvid = syscall_env_alloc();// 为当前运行进程 curenv 申请一个新的子进程并返回子进程的
envid
    if (newenvid == 0) // 说明是子进程
    {
        env = envs + ENVX(syscall_getenvid());
        return 0;
    }
    /* 下面是：父进程在子进程醒来之前还需要做更多的准备 */
    // USTACKTOP = 0x7f3f e000, VPN(USTACKTOP)=0x7f3fe=521,214
    for (i = 0; i < VPN(USTACKTOP); ++i)
    {
        // i是虚拟页表号，也就是二级页表区中的第i个二级页表项
        // 而一个一级页表项表示一个二级页表，
        // 那么 i >> 10 就是计算出这个二级页表项属于第几个二级页表，也就是由第几个一级页表项所代表
        if (((*vpd)[i >> 10] & PTE_V) && ((*vpt)[i] & PTE_V))
            // 一级页表项 和 二级页表项 均有效
            duppage(newenvid, i);
    }
    syscall_mem_alloc(newenvid, UXSTACKTOP - BY2PG, PTE_V | PTE_R);
    // 为子进程先是在UXSTACKTOP - BY2PG处申请一页新的物理内存，作为异常处理栈（exception stack）
    syscall_set_pgfault_handler(newenvid, __asm_pgfault_handler, UXSTACKTOP);
    // 将子进程的env_pgfault_handler域设为异常处理函数__asm_pgfault_handler，将env_xstacktop
域设置为 UXSTACKTOP。
    syscall_set_env_status(newenvid, ENV_RUNNABLE);
    // 现在子进程可以运行了，状态改为ENV_RUNNABLE
    return newenvid;
}
```

最后三行代码可以结合父进程在通过 `set_pgfault_handler(pgfault)` 调用 `syscall_mem_alloc()` 和 `syscall_set_pgfault_handler()` 两个函数去理解。相当于为子进程办一样的事。

```
if (syscall_mem_alloc(0, UXSTACKTOP - BY2PG, PTE_V | PTE_R) < 0 ||
    // register assembly handler and stack with operating system
    // 将当前进程的env_pgfault_handler域设为异常处理函数__asm_pgfault_handler，将
env_xstacktop 域设置为 UXSTACKTOP。
    syscall_set_pgfault_handler(0, __asm_pgfault_handler, UXSTACKTOP) < 0) {
    writef("cannot set pgfault handler\n");
    return;
}
```

页写入异常

内核在捕获到 一个常规的缺页中断（page fault）时（在 MOS 中这个情况特指页缺失），会进入到一个在 `trap_init` 中“注册”的 `handle_tlb` 的内核处理函数中， 这一汇编函数的实现在 lib/genex.S 中，化名为一个叫 `do_refill` 的函数。如果物理页面在页表中存在，则会将其填入 TLB 并 返回异常地址再次执行内存存取的指令。如果物理页面不存在，则会触发一个一般意义的缺页错误，并跳转到 mm/pmap.c 中的 `pageout` 函数中。 如果存取地址是合法的用户空间地址，内核会为对应地址分配并映射一个物理页面（被动地分配页面）来解决缺页的问题。

前文中我们提到了写时复制（COW）特性，这种特性也是依赖于异常处理的。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，我们在 `trap_init` 中为其注册了一个处理函数——`handle_mod`，这一函数会跳转到 lib/traps.c 的 `page_fault_handler` 函数中，这个函数正是 处理写时复制特性的内核函数。

> 由于历史原因，MOS 操作系统的源代码中也使用 page fault 代指这里的页写入异常，但这种异常与之前的缺页中断（页缺失异常）是两种不同的 TLB 异常。具体来说，R3000 的页写入异常会在尝试写入页表项中不带有 dirty bit（表示页面可写入的权限位，即代码中的 PTE_R）的页时产生，而页缺失异常则在尝试访问的页表项中不带有 valid bit（有效位，即 PTE_V）时产生。MOS 操作系统的实现巧妙地利用了一个硬件保留的权限位作为 PTE_COW，并在内核进行 TLB 重填时将标记为 PTE_COW 的页表项中的 dirty bit 置零，因此用户程序在处理时可认为这种页写入异常在且仅在写入 PTE_COW 页面时产生。

处理页写入异常的异常处理栈UXSTACKTOP

我们发现，`page_fault_handler()` 这个函数似乎并没有做任何的页面复制操作。事实上，我们的MOS操作系统按照微内核的设计理念， 尽可能地将功能实现在用户空间中，其中也包括了页写入异常的处理，因此**页写入异常主要的处理过程是在用户态下完成的。**

如果需要在用户态下完成页面复制等处理过程，是不能直接使用正常情况下的进程堆栈的（因为发生页写入异常 的也可能是正常堆栈的页面），所以用户进程就需要一个单独的堆栈来执行处理程序，我们把这个堆栈称作**异常处理栈**，它的栈顶对应的是内存布局中的 UXSTACKTOP。父进程需要为自身以及子进程的异常处理栈映射物理页面。 此外，内核还需要知晓进程自身的处理函数所在地址，它的地址存在于进程控制块的env_pgfault_handler域中，这个地址 也需要事先由父进程通过系统调用设置。

处理页写入异常的流程

处理页写入异常的大致流程可以概括为：

1. 用户进程触发页写入异常，跳转到 `handle_mod` 函数，再跳转到 `page_fault_handler` 函数。
2. `page_fault_handler` 函数负责将当前现场保存在异常处理栈中，并设置epc寄存器的值，使得从中断恢复后能够跳转到env_pgfault_handler域存储的异常处理函数的地址。
3. 退出中断，跳转到异常处理函数中，这个函数首先跳转到 `pgfault` 函数（定义在fork.c中）进行写时复制处理，之后恢复事先保存好的现场，并恢复sp寄存器的值，使得子进程恢复执行。

关于上文提到的异常处理函数，在下文中会做具体介绍。

page_fault_handler()

根据 `BUILD_HANDLER` 中的分析，page_fault_handler()函数的参数 `struct Trapframe *tf` 一定是 `KERNEL_SP - struct Trapframe` 。

这里，page_fault_handler()函数实际上出现了**嵌套异常**。

因为在第2个bcopy中（无论是if还是else），传入的参数是涉及到了用户可操作空间（即kuseg，低2G内核空间）的地址，因此对应到物理内存是需要经过TLB转换的。但是一开始的TLB是什么也没有的，所以又会引发TLB缺失异常。

那么此时，在处理页写入异常的同时又触发了嵌套异常，这就是嵌套异常。

```
void page_fault_handler(struct Trapframe *tf)
{
    struct Trapframe PgTrapFrame;
    extern struct Env * curenv;
//printf("^^^^cp0_BadVAddress:%x\n",tf->cp0_badvaddr);
    // 将将异常处理栈中KERNEL_SP - struct Trapframe的现场信息拷贝到struct Trapframe PgTrapFrame里
    bcopy(tf, &PgTrapFrame,sizeof(struct Trapframe));
    // 如果当前环境中用户栈指针在 [curenv->env_xstacktop - BY2PG, env_xstacktop - 1] 之间，也就是用户栈指针在当前运行进程的 异常处理栈xstacktop区域（其实也就是[UXSTACKTOP-BY2PPG, UXSTACKTOP-1]）
    // 这说明发生了嵌套异常
```

```
    if(tf->regs[29] >= (curenv->env_xstacktop - BY2PG) &&
       tf->regs[29] <= (curenv->env_xstacktop - 1))
    {
        //panic("fork can't nest!!");
        tf->regs[29] = tf->regs[29] - sizeof(struct Trapframe);
        // 需要将用户栈指针向下移动一个struct Trapframe的大小
        bcopy(&PgTrapFrame, tf->regs[29], sizeof(struct Trapframe));
        // 将struct Trapframe PgTrapFrame的信息拷贝到 已经用 用户栈指针regs[29] 分配好的空间
中，也就是异常处理栈xstacktop区域
    }
    else
    {
        // 如果当前环境中用户栈指针不在异常处理栈xstacktop区域
        // 令用户栈指针成为当前进程的 异常处理栈顶 xstacktop 下移一个struct Trapframe 大小的位置
        tf->regs[29] = curenv->env_xstacktop - sizeof(struct Trapframe);
//      printf("page_fault_handler(): bcopy(): src:%x\tdes:%x\n",(int)&PgTrapFrame,
(int)(curenv->env_xstacktop - sizeof(struct  Trapframe)));
        bcopy(&PgTrapFrame, curenv->env_xstacktop - sizeof(struct  Trapframe),
sizeof(struct Trapframe));
    }

    tf->cp0_epc = curenv->env_pgfault_handler;
    return;
}
```

syscall_set_pgfault_handler()

```
int syscall_set_pgfault_handler(u_int envid, void (*func)(void), u_int xstacktop)
{
    return msyscall(SYS_set_pgfault_handler, envid, (int)func, xstacktop, 0, 0);
}
```

sys_set_pgfault_handler()

这个函数将id为envid的进程env的env_pgfault_handler域设为异常处理函数func，将env_xstacktop 域设置为xstacktop。实际上在 `set_pgfault_handler()` 中调用的时候具体情况是：将当前进程的env_pgfault_handler域设为异常处理函数__asm_pgfault_handler，将env_xstacktop 域设置为UXSTACKTOP。

```
int sys_set_pgfault_handler(int sysno, u_int envid, u_int func, u_int xstacktop)
{
    struct Env *env;
    int ret;
    ret = envid2env(envid, &env, 0);
    if (ret)
        return ret;
    env->env_pgfault_handler = func;
    env->env_xstacktop = xstacktop;
    return 0;
    //  panic("sys_set_pgfault_handler not implemented");
}
```

set_pgfault_handler()

在fork函数中，这个函数起到了如下作用

1. 父进程先是在UXSTACKTOP - BY2PG处申请一页新的物理内存，作为异常处理栈（exception stack）
2. 然后将父进程自己的env_pgfault_handler域设为异常处理函数__asm_pgfault_handler，将env_xstacktop 域设置为 UXSTACKTOP。
3. 函数的最后，将entry.S中的字 __pgfault_handler 赋值为fn

这个函数中，进程**为自身**分配映射了异常处理栈， 同时也用系统调用告知内核自身的处理程序是 `__asm_pgfault_handler` （在entry.S定义），随后内核也需要将进程控制块的env_pgfault_handler域设为它。

在函数的最后，将entry.S中的字 `__pgfault_handler` 赋值为fn。而 `sys_set_pgfault_handler()` 函数完成的是内核中的系统调用，它设置了进程控制块中的两个域。

set_pgfault_handler()函数实现于user/pgfault.c中，下面是这个文件除了头文件以外所有的内容。

set_pgfault_handler()的参数是一个函数指针 `void (*fn)(u_int va)` ，**在调用的时候传入的其实是pgfault()函数**。

```
extern void (*__pgfault_handler)(u_int);
extern void __asm_pgfault_handler(void);
void set_pgfault_handler(void (*fn)(u_int va))
{
    // 最开始entry.S中的全局变量__pgfault_handler是0，要让*__pgfault_handler成为函数指针
    // *__pgfault_handler 如果还没有存入pgfault()，说明栈空间还没有申请
    if (__pgfault_handler == 0) {
        // 为当前进程的进程空间的UXSTACKTOP - BY2PG处申请一页新的物理内存，作为exception stack
        if (syscall_mem_alloc(0, UXSTACKTOP - BY2PG, PTE_V | PTE_R) < 0 ||
            // register assembly handler and stack with operating system
            // 将当前进程的env_pgfault_handler域设为异常处理函数__asm_pgfault_handler，将
env_xstacktop 域设置为 UXSTACKTOP。
            syscall_set_pgfault_handler(0, __asm_pgfault_handler, UXSTACKTOP) < 0) {
            writef("cannot set pgfault handler\n");
            return;
        }
        //      panic("set_pgfault_handler not implemented");
    }
    // Save handler pointer for assembly to call.
    // 将entry.S中的字 __pgfault_handler 设置为 fn，其实就是pgfault()函数
    // 这样在__asm_pgfault_handler中就可以用pgfault()
    __pgfault_handler = fn;
}
```

我们现在知道了页写入异常处理会返回到entry.S中的 `__asm_pgfault_handler` 函数，我们再来看这个函数会 做些什么。

__asm_pgfault_handler

从内核返回后，此时的栈指针是由内核设置的，处于异常处理栈中，且指向一个由内核复制好的 Trapframe 结构体的底部。该函数通过宏定义的偏移量 TF_BADVADDR，用 lw 指令读取了 Trapframe 中的 cp0_badvaddr 字段的值，这个值 也正是 CPU 设置的发生页写入异常的地址。该函数将这个地址作为参数去调用了 __pgfault_handler 这个变量内存储的函数指针，其指向的函数就是**“真正”进行处理的函数pgfault()**（这个赋值过程发生在上面的 `set_pgfault_handler()` 函数）。 随后就是一段用于恢复现场的汇编，最后使用 MIPS 的延时槽特性，在跳转的同时恢复了正常的栈指针。

```
.globl __pgfault_handler
__pgfault_handler:
        .word 0
.globl __asm_pgfault_handler
__asm_pgfault_handler:
    // save the caller-save registers
    //  (your code here)
//1: j 1b
nop
    lw  a0, TF_BADVADDR(sp)     # 读取了 Trapframe 中的 cp0_badvaddr 字段的值，这个值 也正是
CPU 设置的发生页写入异常的地址，存入a0 作为pgfault参数
    //sw    t0, (sp)
    //subu  sp,16
    lw  t1, __pgfault_handler
    jalr    t1

nop
```

```
    // push trap-time eip, eflags onto trap-time stack
    //  (your code here)
    //addu  sp,16
    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  $30,TF_REG30(sp)
        lw  $28,TF_REG28(sp)
        lw  $25,TF_REG25(sp)
        lw  $24,TF_REG24(sp)
        lw  $23,TF_REG23(sp)
        lw  $22,TF_REG22(sp)
        lw  $21,TF_REG21(sp)
        lw  $20,TF_REG20(sp)
        lw  $19,TF_REG19(sp)
        lw  $18,TF_REG18(sp)
        lw  $17,TF_REG17(sp)
        lw  $16,TF_REG16(sp)
        lw  $15,TF_REG15(sp)
        lw  $14,TF_REG14(sp)
        lw  $13,TF_REG13(sp)
        lw  $12,TF_REG12(sp)
        lw  $11,TF_REG11(sp)
        lw  $10,TF_REG10(sp)
        lw  $9,TF_REG9(sp)
        lw  $8,TF_REG8(sp)
        lw  $7,TF_REG7(sp)
        lw  $6,TF_REG6(sp)
        lw  $5,TF_REG5(sp)
        lw  $4,TF_REG4(sp)
        lw  $3,TF_REG3(sp)
        lw  $2,TF_REG2(sp)
        lw  $1,TF_REG1(sp)
        lw  k0,TF_EPC(sp)   //atomic operation needed
        jr  k0
        lw  sp,TF_REG29(sp)  /* Deallocate stack */
```

下面是实现真正进行处理的函数：user/fork.c 中的 `pgfault` 函数，它也正是父进程在fork中使用 `set_pgfault_handler` 函数注册的处理函数。

pgfault() 在用户态中真正处理写入页异常

`pgfault` 需要完成这些任务：

1. 判断参数虚拟地址va是否为 COW 的页面，是则进行下一步，否则报错
2. 分配一个新的临时物理页到临时位置，将要复制的内容拷贝到刚刚分配的页中（临时页面位置可以自定义，观察mmu.h的地址分配查看哪个地址没有被用到，思考这个临时位置可以定在哪）

   这里的临时位置就是USTACKTOP地址往上的一页，在mmu.h中是Invalid memory。

   注意是USTACKTOP不是UXSTACKTOP！
3. 将发生页写入异常的地址映射到临时页面上，注意设定好对应的页面权限位，然后解除临时位置的内存映射

```
static void pgfault(u_int va)
{
    u_int *tmp = USTACKTOP;
    //  writef("fork.c:pgfault():\t va:%x\n",va);
    u_long perm = (*vpt)[VPN(va)] & 0xfff;// 取出va对应二级页表项的权限位
    if ((perm & PTE_COW) == 0)
    {
        user_panic("pgfault err: COW not found");
```

```
    }
    perm = perm & (~PTE_COW);// 取消写时复制保护
    // perm -= PTE_COW;      // 这种写法不好，虽然能保证是一定有PTE_COW位，但是如果没有，就不安全
    //map the new page at a temporary place
    syscall_mem_alloc(0, tmp, perm);
    //copy the content
    user_bcopy(ROUNDDOWN(va, BY2PG), tmp, BY2PG);
    //map the page on the appropriate place
    // 将复制好的新页面映射到当前进程的进程空间的va处，这里的va和发生异常的va是一样的
    syscall_mem_map(0, tmp, 0, va, perm);
    //unmap the temporary place
    syscall_mem_unmap(0, tmp);// 因为在上面syscall_mem_alloc申请页面内存的时候会建立映射
}
```

父进程还需要通过类似 `set_pgfault_handler` 的方式，用若干系统调用分配子进程的异常处理栈，并设置其 处理函数为 `__asm_pgfault_handler` 。 最后，父进程通过系统调用 `syscall_set_env_status` 设置子进程为可以运行的状态。在内核中实现 `sys_set_env_status` 函 数时，不仅需要设置进程控制块的env_status域，还需要在env_status被设为RUNNABLE时将控制块加入到可调度进程的链表中。

sys_set_env_status()

```
int sys_set_env_status(int sysno, u_int envid, u_int status)
{
    // Your code here.
    struct Env *env;
    int ret;

    if (status != ENV_RUNNABLE && status != ENV_NOT_RUNNABLE && status != ENV_FREE)
    {
        return -E_INVAL;
    }
    ret = envid2env(envid, &env, 0);
    if (ret)
        return ret;
    if (env->env_status != ENV_RUNNABLE && status == ENV_RUNNABLE)
    {
        LIST_INSERT_HEAD(&env_sched_list[0], env, env_sched_link);// or tail?
    }
    if (env->env_status == ENV_RUNNABLE && status != ENV_RUNNABLE)
    {
        LIST_REMOVE(env, env_sched_link);
        // 这里使用 LIST_REMOVE 直接去掉也可以，因为一个进程总可以通过进程控制块数组下标来拿到
    }
    env->env_status = status;
    return 0;
    //  panic("sys_env_set_status not implemented");
}
```

说到这里我们需要整理一下思路，fork中父进程在 `syscall_env_alloc` 后还需要做的事情有：

1. 遍历父进程地址空间，进行duppage。
2. 为子进程分配异常处理栈。
3. 设置子进程的异常处理函数，确保页写入异常可以被正常处理。
4. 设置子进程的运行状态

最后再将子进程的envid返回，fork函数就大功告成了！

Bonus 页写入异常的处理流程

关键词：三个指针（内核态的异常处理指针 `KERNEL_SP - TF_SIZE`，用户态的被 `page_fault_handler` 设置在[UXSTACKTOP - BY2PG, UXSTACKTOP - 1]区域的指针，以及**真正发生页写入异常**的用户态栈指针）

两个EPC：一个是 `page_fault_handler` 函数给 `KERNEL_SP - TF_SIZE` 设置的epc，存放 `__asm_pgfault_handler` 的地址，用于在回到用户态的时候，系统处于这个函数的地址。

一个则是在 设置在[UXSTACKTOP - BY2PG, UXSTACKTOP - 1]区域的指针 指向的环境中，存放的**真正发生页写入异常**的epc值。

*Step*0 触发页写入异常，进入内核态

用户态下，当对带有PTE_COW的页面进行写入时，就会触发写入页异常，系统进入内核态。

> **中断异常入口**
>
> 根据R3000手册, R3000的中断异常入口有5个, 此处列出其中在MOS中最重要的两个入口:
>
> | 入口地址 | 所在内存区 | 描述 |
> | --- | --- | --- |
> | `0x80000000` | kseg0 | TLB缺失时, PC转至此处 |
> | `0x80000080` | kseg0 | 对于除了TLB缺失之外的其他异常, PC转至此处 |

> 注意，在MOS中，仅仅实现了 `0x80000080` 处的中断异常处理程序。这样也能工作的原因是：当TLB缺失时，PC转至 `0x80000000` 后，**空转32周期(执行了32条 `nop` 指令)**，到达 `0x80000080` ，随后也就与其他的中断异常一同处理了。也就是说我们只需要实现 `0x80000080` 处的程序即可。

因此，**一旦 CPU 发生除了TLB缺失以外的异常，就会自动跳转到地址 0x80000080 处**，开始执行异常分发程序。

*Step*1 执行异常分发程序 except_vec3

.text.exc_vec3 段需要被链接器放到特定的位置，在 R3000 中这一段是要求放到地址 0x80000080 处，这个地址处存放的是异常处理程序的入口地址。一旦 CPU 发生异常，就会自动跳转到地址 0x80000080 处，开始执行。因此，我们在lab3中需要在 tools/scse0_3.lds 中的开头位置增加如下代码，即将.text.exc_vec3 放到 0x80000080 处, 来为操作系统增加异常分发功能。

```
. = 0x80000080;
.except_vec3 : {
    *(.text.exc_vec3)
}
```

异常分发程序的具体代码位于start.S的开头。

概括来说，这个函数的作用是：

1. 将Cause寄存器中的值取出，存到 k1 寄存器
2. 将 exception_handlers 这个数组的首地址存到了 k0 寄存器
3. 将 k1 中 Cause寄存器的值 和0x7c做按位与，使得 k1 中仅保留Cause寄存器中的 ExcCode段 乘4的值

> 因为ExcCode是 Cause寄存器 的2-6位，当按位与后，低2位置0，就相当于ExcCode表示的异常号左移了两位，即乘以4；而异常向量组中每一个成员的大小都是4字节，这样就相当于直接计算出了对于成员相对于数组基地址的偏移量。

4. addu k0,k1　这条指令做偏移, 使得 k0 的值为 exception_handlers 数组的第ExcCode项
5. jr k0 跳转到该中断处理子函数

```
.section .text.exc_vec3
NESTED(except_vec3, 0, sp)
    .set noat
```

```
    .set noreorder
1:
    mfc0 k1,CP0_CAUSE           # 将Cause寄存器中的值取出，存到 k1 寄存器
    la k0,exception_handlers    # la 为扩展指令，意为load address。
                                # 该指令将 exception_handlers 这个数组的首地址存到了 k0 寄存
器。
    andi k1,0x7c                # 此处和0x7c做按位与，使得 k1 中仅保留Cause寄存器中的ExcCode
段
                                    # 0x7c=01111100，说明 k1 中存的是ExcCode乘4的值
    addu k0,k1
    # 在前面的语句中，k0 寄存器保存了 exception_handlers 的首地址，k1 保存了ExcCode乘4的值，
    # 此处做偏移，使得 k0 的值为 exception_handlers 数组的第ExcCode项
    lw k0,(k0)
    # 将此时 k0 指向的数组项取出，仍存到 k0 中去。这里 lw k0 (k0) 就是 lw k0 (k0)
    # 此时，k0 的值就是数组第ExcCode项的值了，也就是异常码ExcCode对应的中断异常处理子函数的入口地
址.
    nop
    jr k0                       # 跳转到该中断处理子函数
    nop
END(except_vec3)
.set at
```

值得注意的是, 这里完全使用 `k0` , `k1` 两个内核保留的寄存器，**可以保证用户态下其他通用寄存器的值不被改变**，也就保持了现场不变。

这里再回想起TF开头的宏中（即TrapFrame相关），有这样的注释

```
/*
 * $26 (k0) and $27 (k1) not saved
 */
```

联系这里，就不难理解上面这句注释的含义了。

> 此外，这里涉及到了两个 `.set` 指令：
>
> `noat` : 意味着接下来的代码中**不允许汇编器使用 `at` 寄存器(即 1 号寄存器)**。这是因为此时刚刚陷入内核，还未保存现场，用户态下除了 `k0` , `k1` 之外都不能够被改变。
>
> `noreorder` : 意味着接下来的代码中不允许汇编器重排指令顺序。

$Step1.5$ 通过 except_vec3异常向量组，进入异常处理子函数

异常分发程序通过 exception_handlers 数组定位中断处理程序，而 exception_handlers 就称作异常向量组。

lib/traps.c 中的 trap_init() 函数说明了异常向量组里存放了什么。

trap_init() 和 set_except_vector()

```
extern void handle_int();
extern void handle_reserved();
extern void handle_tlb();
extern void handle_sys();
extern void handle_mod();
unsigned long exception_handlers[32];
void trap_init()
{
    int i;
    for (i = 0; i < 32; i++) {
        set_except_vector(i, handle_reserved);
    }
    set_except_vector(0, handle_int);
    set_except_vector(1, handle_mod);
    set_except_vector(2, handle_tlb);
    set_except_vector(3, handle_tlb);
    set_except_vector(8, handle_sys);
```

```
}
// 向异常向量组 exception_handlers 中数组下标为 n 的地方存入异常处理函数地址 addr
void *set_except_vector(int n, void *addr)
{
    unsigned long handler = (unsigned long)addr;
    unsigned long old_handler = exception_handlers[n];
    exception_handlers[n] = handler;
    return (void *)old_handler;
}
```

实际上，trap_init()函数实现了对全局变量 exception_handlers[32] 数组初始化的工作，即通过把相应处理函数的地址填到对应数组项中，初始化了如下异常：

0 号异常的处理函数为 `handle_int`，表示中断，由时钟中断、控制台中断等中断造成

1 号异常的处理函数为 `handle_mod`，表示存储异常，进行存储操作时该页被标记为只读

2 号异常的处理函数为 `handle_tlb`，TLB 异常，TLB 中没有和程序地址匹配的有效入口

3 号异常的处理函数为 `handle_tlb`，TLB 异常，TLB 失效，且未处于异常模式（用于提高处理效率）

8 号异常的处理函数为 `handle_sys`，系统调用，陷入内核，执行了 syscall 指令

注意：异常不等于中断，x号中断和x号异常不一样。

那么这里，页写入异常，对应着1号异常。现在，我们由异常分发程序，其实就已经进入了异常向量组中注册过的的异常处理子函数。

Step2 执行对应异常处理子函数handle_mod

handle_mod函数实际上是由BUILD_HANDLER汇编宏函数构造出来的。

BUILD_HANDLER

```
BUILD_HANDLER mod   page_fault_handler cli

.macro  __build_clear_sti
    STI
.endm

.macro  __build_clear_cli
    CLI
.endm

.macro  BUILD_HANDLER exception handler clear
    .align  5
    NESTED(handle_\exception, TF_SIZE, sp)
    .set    noat
nop
    SAVE_ALL                    # 保存现场到异常处理栈
    __build_clear_\clear        // 对于写入页异常来说，这里是允许用户态下使用cp0寄存器，且禁用中
断
    .set    at
    move    a0, sp      # 将压栈以后的异常处理栈存入a0
    // 在页写入异常中，这就是KERNEL_SP - struct Trapframe
    # (4号中断是TIMESTACK - struct Trapframe处，其他异常则是KERNEL_SP - struct Trapframe
处)
    // 此时 a0 作为下面的 \handle（即 page_fault_handler ）对应的函数 的 参数

    jal \handler        # 在ra处存入这里下一部分的地址，跳转到 \handle （即
page_fault_handler ）参数对应的函数的位置
    nop                 # tlb异常的时候执行 do_refill，存储异常的时候调用 page_fault_handler
    j   ret_from_exception      # 调转到 ret_from_exception 恢复现场与回滚
    nop
    END(handle_\exception)
.endm
```

保存现场到异常处理栈

这里就是调用SAVE_ALL函数，将发生写入页异常时的环境存储到 异常处理栈 `KERNEL_SP - TF_SIZE` 往上一个结构体的空间。具体来说是：

1.将用户栈指针存入 k0 寄存器

2.调用get_sp 将sp置为异常处理栈指针，并将处理栈指针下压一个 struct Trapframe 的大小

3.将 k0 寄存器中存储的用户栈指针存入 `TF_REG29(sp)`

4.将用户态下的 v0 寄存器存入 `TF_REG2(sp)`

5.将CP0寄存器 和 其他通用寄存器(0-31，除了2和29)存入异常处理栈下压出来的栈空间

sti cli

```
#define STATUS_CU0 0x10000000
// 表示允许用户态下使用cp0寄存器，且 启用中断
.macro STI
    mfc0    t0, CP0_STATUS
    li      t1, (STATUS_CU0 | 0x1)  # 0x1000 0001 28位0位为1
    or      t0, t1
    mtc0    t0, CP0_STATUS

.endm
// 表示允许用户态下使用cp0寄存器，且 禁用中断
.macro CLI
    mfc0    t0, CP0_STATUS
    li      t1, (STATUS_CU0 | 0x1)
    or      t0, t1
    xor     t0, 0x1
    mtc0    t0, CP0_STATUS
.endm
```

page_fault_handler

具体代码解析上面说过了，这里作概括。

传入的参数 `struct Trapframe *tf` 实际上就是 `KERNEL_SP - struct Trapframe` 。

1.将当前环境中用户栈指针 `tf->regs[29]` 设置为进程异常处理栈区域 [curenv->env_xstacktop - BY2PG, curenv->env_xstacktop - 1]。而在函数 `set_pgfault_handler` 中，env_xstacktop 域被设置为了 UXSTACKTOP，所以这里就是把用户栈指针设置在了[UXSTACKTOP - BY2PG, UXSTACKTOP - 1]的区域。

2.将 `KERNEL_SP - struct Trapframe` 存放的进程上下文环境复制进入**设置以后的的用户栈指针** `tf->regs[29]` 指向的区域，因此这里：

如果说记 `tf->regs[29]` 存储的用户栈指针为 `user_sp` ，那么 `TF_EPC(user_sp)` 才是发生页写入异常时的PC值，这点很重要！！！因为下面会改变 `KERNEL_SP - struct Trapframe` 区域中的EPC，用于开始执行__asm_pgfault_handler。

2.设置当前环境中的epc寄存器为 `env_pgfault_handler` 域的值，也就是__asm_pgfault_handler的地址。（ `tf->cp0_epc = curenv->env_pgfault_handler` ）

由此可见，其实内核态下，页写入异常的处理函数只是设置了用户态栈指针位于异常处理栈区域，并且设置了epc寄存器，存放异常处理函数__asm_pgfault_handler的地址。

恢复现场与回滚 RESTORE_SOME

恢复现场的主要逻辑由汇编宏 `RESTORE_SOME` (include/stackframe.h)支持。此宏将现场中除了 `sp` `k1` `k0` 寄存器之外的所有寄存器都恢复到CPU中：

```
.macro RESTORE_SOME
    .set mips1
```

```
    lw      v0, TF_STATUS(sp)
    mtc0    v0, CP0_STATUS
    lw      v1, TF_LO(sp)
    mtlo    v1
    lw      v0, TF_HI(sp)
    mtlo    v0
    lw      v1, TF_EPC(sp)
    mtc0    v1, CP0_EPC
    lw      $31, TF_REG31(sp)
    lw      $30, TF_REG30(sp)
    //lw      $29, TF_REG29(sp) <~~ $29 <=> sp
    lw      $28, TF_REG28(sp)
    //lw      $27, TF_REG27(sp) <~~ $27 <=> k1
    //lw      $26, TF_REG26(sp) <~~ $26 <=> k0
    lw      $25, TF_REG25(sp)
    /* $24 - $2 都有，这里就不写了*/
    lw      $1, TF_REF1(sp)
.endm
```

此处并未恢复 `k0/k1` 这两个内核态寄存器, 这是**由于在 MIPS 规范下, 用户程序不会使用这两个寄存器. 而用户程序的编写者也应该遵循这个规范.**

- 若需要直接编写汇编代码, 应当时刻遵循此规范.
- 若通过编译器得到汇编代码, 则应当选择符合规范的编译器, 如: mips-4KC.

恢复现场与回滚 ret_from_exception

通过 `RESTORE_SOME` ，可以使得用户现场中，除了 `sp` 栈指针寄存器外的信息都被恢复，而 `sp` 寄存器将在回滚现场前的最后关头才恢复。**MOS中定义了汇编函数 `ret_from_exception` (lib/genex.S)用于恢复现场并回滚用户态**。

在页写入异常中，这里是把**内核态指针 `KERNEL_SP - TF_SIZE` 记录的环境**中的epc中存放的 `__asm_pgfault_handler` 地址存入了k0寄存器，并由 `jr  k0` 跳转到这个函数的入口，并执行 rfe 回到用户态（SR寄存器低六位的二重栈压栈）

新申请的进程的cp0_status被写入0001 0000 0000 0000 0001 0000 0000 1100，在出栈后，最后两位 KUc，IEc 为 [1,1]，表示CPU目前**在用户态下运行，并且开启了中断**。

至此，sp寄存器恢复成了用户态指针，这个值位于用户态的异常处理区域[UXSTACKTOP - BY2PG, UXSTACKTOP - 1]，并且指向的环境结构体就是发生页写入异常时的上下文环境。

存在k0中的，是内核态指针 `KERNEL_SP` 指向的epc位置，这里存放的是__asm_pgfault_handler；

而在恢复sp为用户栈指针后，sp指向的epc位置，是存放着发生页写入异常的pc值。是处理完页写入异常以后函数真正应该跳转回去的值。

```
FEXPORT(ret_from_exception)
    .set noat
    .set noreorder
    RESTORE_SOME
    .set at
    lw  k0,TF_EPC(sp)           # 将异常处理栈中EPC对应的位置存入k0，
    // 那么这里就是相当于把epc中存放的__asm_pgfault_handler存入了k0
    # 此时sp的值依然是0x82000000-TF_SIZE (TF_SIZE 为 sizeof(struct Trapframe)
    lw  sp,TF_REG29(sp) /* Deallocate stack */
    # 在SAVE_ALL中，将用户栈指针由sp 存入 k0 再存入 Trapframe 中的 TF_REG29 处，这里就是将sp重置为用户栈指针
//1:    j   1b
    nop
    jr  k0                      # 异常处理完毕，跳转到中断时的位置继续执行
    rfe
```

Step3 在用户态下完成页写入异常的处理

在回到用户态以后，系统现在位于__asm_pgfault_handler函数的入口。

__asm_pgfault_handler

从内核返回后，此时的栈指针是由内核设置的，处于异常处理栈中，且指向一个由内核复制好的 Trapframe 结构体的底部。

`__asm_pgfault_handler` 函数通过宏定义的偏移量 TF_BADVADDR，用 lw 指令读取了 Trapframe 中的 cp0_badvaddr 字段的值，这个值 也正是 CPU 设置的发生页写入异常的地址。

该函数将这个地址作为参数去调用了 __pgfault_handler 这个变量内存储的函数指针，其指向的函数就是**“真正”进行处理的函数pgfault()**（这个赋值过程发生在上面的 `set_pgfault_handler()` 函数）。

```
lw  a0, TF_BADVADDR(sp)     # 读取了 Trapframe 中的 cp0_badvaddr 字段的值，这个值 也正是 CPU 设置的发生页写入异常的地址，存入a0 作为pgfault参数
lw  t1, __pgfault_handler
jalr    t1
```

随后就是一段用于恢复现场的汇编，最后使用 MIPS 的延时槽特性，在跳转的同时恢复了正常的栈指针。

由于**设置在[UXSTACKTOP - BY2PG, UXSTACKTOP - 1]区域以后**的用户态指针指向的epc值是发生页写入异常时真正的epc值，所以这里的跳转才没有问题，由于延迟槽特性，`lw    sp,TF_REG29(sp)` 会在跳转发生前指向，而被设置在区域的用户态指针sp的 `TF_REG29(sp)` 中存放的，才是 **发生页写入异常的用户栈指针。因此需要再次还原。**

```
lw  k0,TF_EPC(sp)   //atomic operation needed
jr  k0
lw  sp,TF_REG29(sp)  /* Deallocate stack */
```

pgfault() 在用户态中真正处理写入页异常

`pgfault` 需要完成这些任务：

1. 判断参数虚拟地址va是否为 COW 的页面，是则进行下一步，否则报错
2. 分配一个新的临时物理页到临时位置，将要复制的内容拷贝到刚刚分配的页中（临时页面位置可以自定义，观察mmu.h的地址分配查看哪个地址没有被用到，思考这个临时位置可以定在哪）

   这里的临时位置就是USTACKTOP地址往上的一页，在mmu.h中是Invalid memory。

   注意是USTACKTOP不是UXSTACKTOP！
3. 将发生页写入异常的地址映射到临时页面上，注意设定好对应的页面权限位，然后解除临时位置的内存映射

Part3 体会与感想&指导书反馈

感谢『MOS』 Introduction - Coekjan