实验目的

- 掌握系统调用的概念及流程
- 实现进程间通讯机制
- 实现 fork 函数
- 掌握写时复制特性和页写入异常的处理流程

在 lab4 中，我们需要实现**系统调用机制**，并在此基础上实现**进程间通信机制** IPC 和一个重要的**进程创建机制** fork。在 fork 部分的实验中需要掌握写时复制（COW）特性，以及与其相关的页写入异常处理。

实验难点

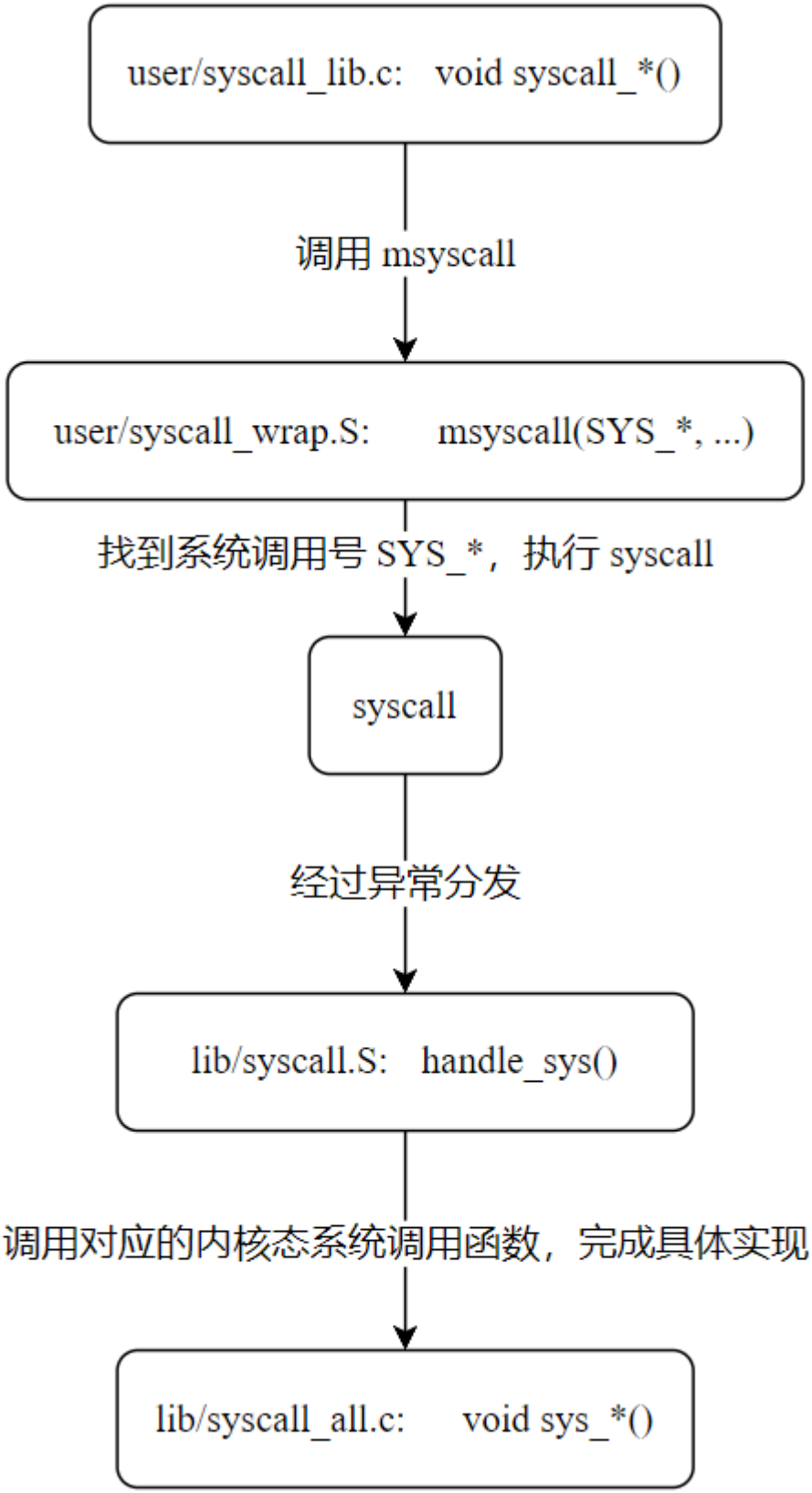
机制总览

- **系统调用机制**

  总地来说，系统调用机制的存在是为了将一些内核中的服务提供给用户空间，使得用户程序能够在保证系统安全性的前提下完成一些特殊的系统级操作。系统调用机制中存在着巧妙的层次化设计，这一点在实验过程中也能清晰地体会到（尤其是在漏写了某个函数而导致出锅时），这样能够让程序更为灵活，并具有更好的可移植性，通过执行 syscall 指令，就能够让用户进程陷入内核态，请求内核提供的服务。

  系统调用的处理流程和时钟中断的处理流程相似：用户使用系统调用后，CPU进入内核态，并通过异常分发程序的分发进入到了异常服务函数 `handle_sys`。然后，`handle_sys` 函数根据传入的系统调用号跳转到具体的处理函数 `sys_*`。经过一系列处理，最后返回用户态。

  其中的难点基本是：在**层次化**设计的种种函数间补全某个系统调用的完整实现机制及其功能；在**用户态与内核态之间**进行数据传递与保护，尤其要注意实现系统调用的代码中是否使用了正确的宏和函数调用。整理了如下的图之后，基本就不会在补全实现机制时遗漏一些文件。



- **handle_sys()**

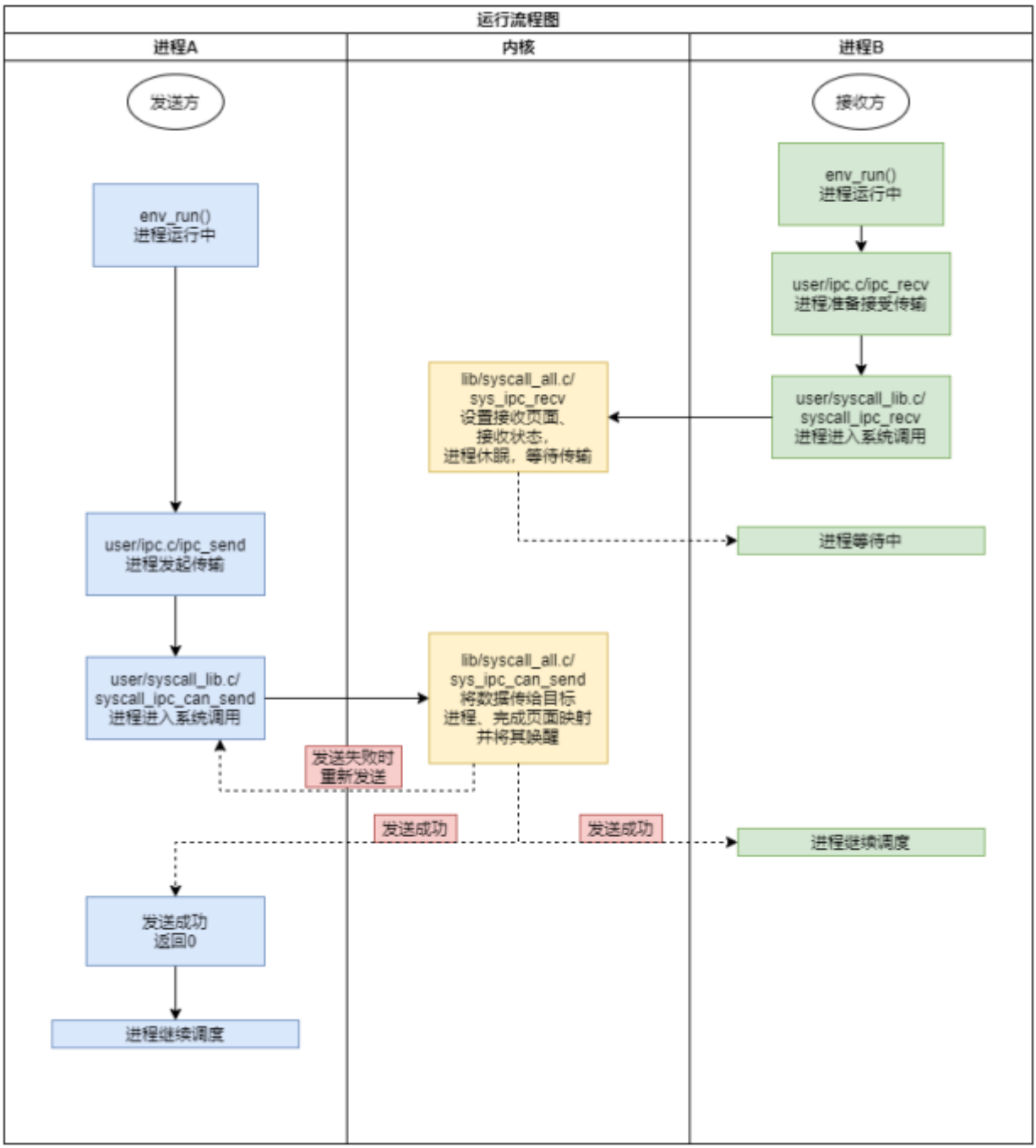
  汇编的部分总是会让人多少有点烦恼，而在系统调用部分，使得内核部分的系统调用机制可以正常工作的 lib/syscall.S 无疑是汇编的重点。大致总结其过程如下：

  > 1. 从 TF 中取出 EPC 的值，加 4 之后放回 TF 中，使得异常处理结束后可以继续执行下一条指令；
  > 2. 从 TF 中读取 $a0 的值（系统调用号）；

> 3. 在内核栈上分配储存六个参数的空间，并且把六个参数放到正确的位置；
>
> *注意在mips中按照规定，前四个参数并不放在栈上，而是直接留在寄存器中，说明在跳转到相应的系统调用函数之前，前四个参数可以由寄存器传递，但后两个参数需要通过开辟新的栈空间传递*
>
> 4. 恢复内核栈的位置。

- **进程间通信机制**

  IPC 是微内核最重要的机制之一，目的是使得两个进程之间可以通讯，其实现是显然的，就是在共享区域**交换数据**，并且通过**系统调用**来实现——为什么？因为各个进程的地址空间相互独立，沟通两个进程需要内核态来操作（其权限凌驾于进程之上）。

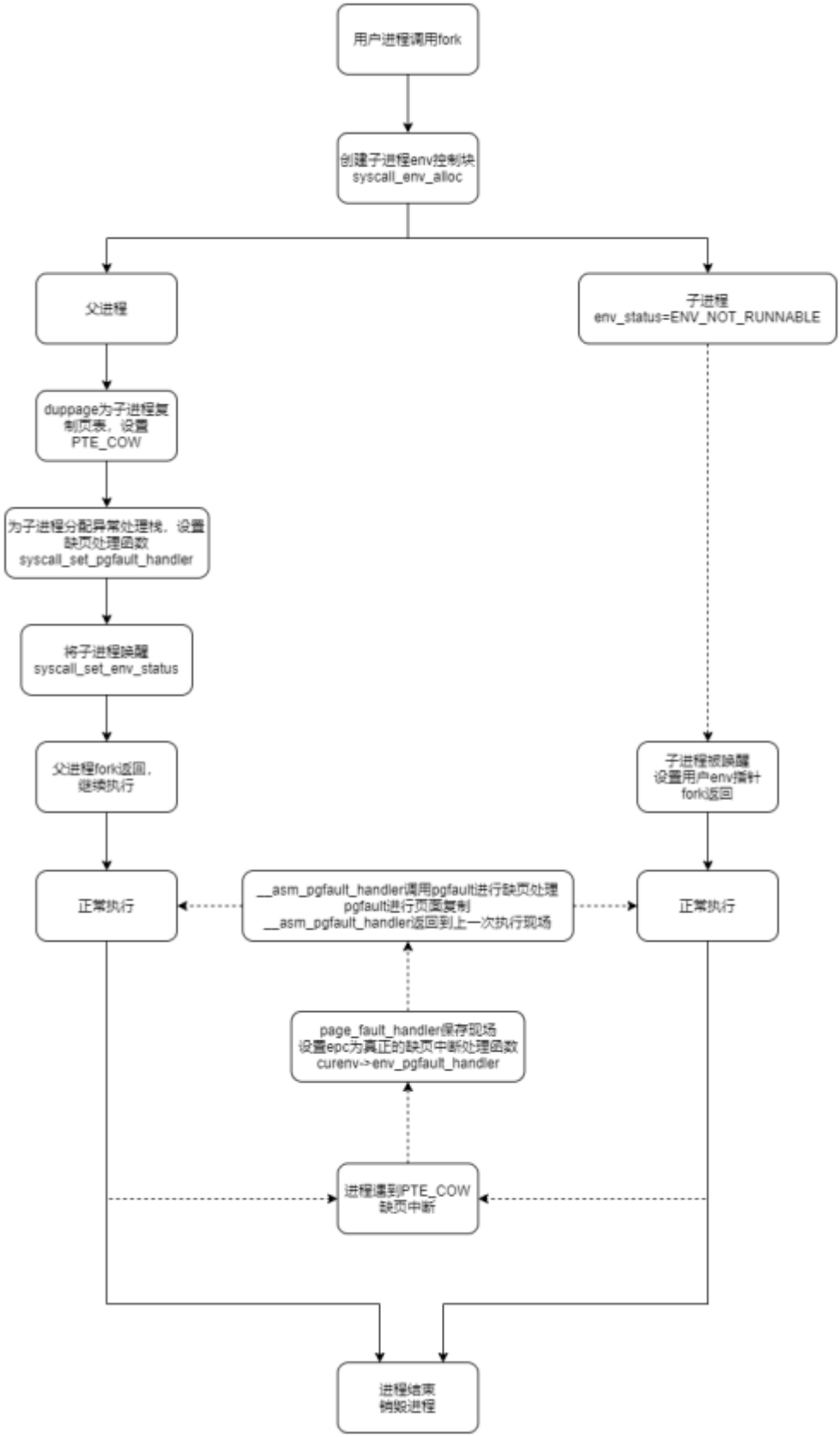


  这部分的难点主要是捋清系统调用部分收发消息的**规则**，描述略多，需要仔细阅读注释以实现。

- **进程创建机制**

这一部分的难点其实在于对**写时复制**机制的处理。

写时复制的原理是：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。这样父子进程在试图写被保护页面时都会产生页写入异常，在它的处理函数中，操作系统进行写时复制，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，取消虚拟页的这一标志位。

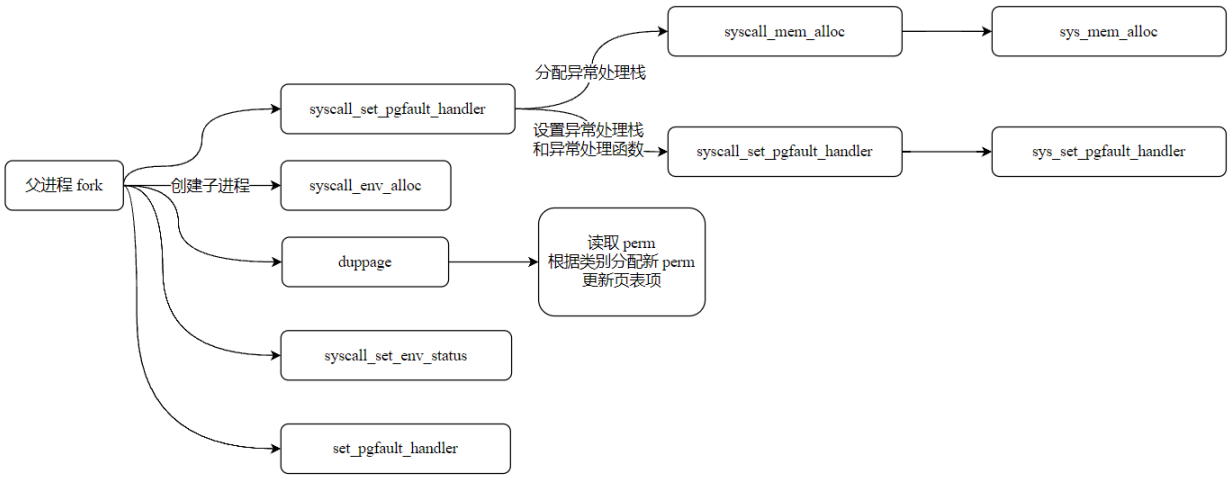


其具体实现比较繁琐，个中需要注意的点总结如下：

1. 对进程用户空间页设置保护时，需要先检查页目录项是否有效，再检查页表项是否有效；
2. 需要确定的是，可写、非共享且非写时复制的页面需要写时复制保护。这时需要对父子进程都分别设置 PTE_COW 位进行保护。

   > perm = `((Pte*)(*vpt))[pn] &0xfff`，通过取得页表 vpt 中第 pn 项查看其低位值获取 perm 位；
   >
   > 子进程赋值的规则是：
   >
   > 1. Invalid: panic
   > 2. 只读 || 共享 || perm: 设置相同的 perm 位
   > 3. 可写: 父子进程都设置 PTE_COW 位
   >
   > 注意通过 syscall_mem_map 修改 perm 位

3. 进行空间映射时先映射子进程再映射父进程



细节难点

- **GDB调试**

  `GDB` 的主要功能就是监控程序的执行流程。这也就意味着，只有当源程序文件编译为可执行文件并执行时，并且该文件中必须包含必要的调试信息（比如各行代码所在的行号、包含程序中所有变量名称的列表（又称为符号表）等），`GDB` 才会派上用场。

  所以在编译时需要使用 `gcc/g++ -g` 选项编译源文件，才可生成满足 `GDB` 要求的可执行文件。

  本实验中只需要简单了解一些调试指令（指导书中也都详细列出了），但依旧总结一些常用命令如下：

调试命令	作用
(gdb) break (b) xxx	在源代码指定的某一行设置断点，其中 xxx 用于指定具体打断点的位置
(gdb) run (r)	执行被调试的程序，它会在第一个断点处暂停执行
(gdb) continue (c)	当程序在某一断点处停止后，用该指令可以继续执行，直至遇到断点或者程序结束
(gdb) next (n)	令程序一行代码一行代码地执行
(gdb) step (s)	如果有调用函数，则进入调用的函数内部，否则和 next 命令作用相同
(gdb) until (u) location	单纯使用 until 命令可以运行程序直到退出循环体。如果指定行号 location ，则该命令会使程序运行至第 location 行代码处停止
(gdb) print (p) xxx	打印指定变量的值
(gdb) list (l)	显示源程序代码的内容，包括各行代码所在的行号
(gdb) finish (fi)	结束当前正在执行的函数，并在跳出函数后暂停程序的执行
(gdb) return (return)	结束当前调用函数并返回指定值，到上一层函数调用处停止程序执行
(gdb) jump (j)	使程序从当前要执行的代码处，直接跳转到指定位置处继续执行后续的代码
(gdb) quit (q)	终止调试

- **汇编方式**

  这里需要回顾一些计组知识，即在跳转指令等部分不要忘记考虑延迟槽，不论是上机还是课下实验都对这一点做出了考察要求。

- **进程与内核的关系并非对立**

  在内核处理进程发起的系统调用时，我们并没有切换 CPU 的地址空间（页目录地址），也不需要将进程上下文（Trapframe）保存到进程控制块中，只是切换到内核态下，执行了一些内核代码。可以说，处理系统调用时的内核仍然是代表当前进程的，这也是系统调用等中断与时钟中断的本质区别，也是我们引入 KERNEL_SP 和 TIMESTACK 两种机制来保存进程上下文的一个原因。

思考题

Thinking 4.1

- **内核在保存现场的时候是如何避免破坏通用寄存器的?**

  通过一个汇编宏 SAVE_ALL 将所有通用寄存器的值保存到栈里（sp），不过 k0 和 k1 两个寄存器由中断程序保留，内核使用它们来保存用户栈、取出内核栈，再进行寄存器值的保存，从而避免破坏大多数通用寄存器。

- **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。因为调用函数时的前四个参数传入 a0 - a3 寄存器后这些值保存在了当前栈下的相应偏移处，内核保存现场的过程中没有破坏 a0 - a3 参数寄存器的值，只改变了 k0, k1, v0 的值，因此可以从上述 4 个参数寄存器中取出原来保存的信息。

- **我们是怎么做到让 sys 开头的函数"认为"我们提供了和用户调用 msyscall 时同样的参数的?**

  调用函数时前四个参数都按顺序传入 a0 - a3 寄存器，后两个参数按顺序存入内核栈的相同偏移位置——由于参数的传递仅依赖于 a0 - a3 寄存器和栈，因此只要这样操作就能使得 sys 开头的函数认为参数相同。

- **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?**

  首先，内核必然改变了 Trapframe 中 v0 寄存器的值，用于在用户态中传递系统调用函数的返回值；同时，内核改变了 CP0_EPC 的值，使得程序返回用户态后能够从正确的位置继续执行下一条指令。

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

Thinking 4.2

从 mkenvid() 代码可以发现 envid 的第十位恒为1，这保证了生成的 envid 非0；而从 envid2env() 函数的具体实现中可以找到保留 0 值的理由：

```
int envid2env(u_int envid, struct Env **penv, int checkperm)
{
    /* Hint: If envid is zero, return curenv.*/
    if (envid == 0) {
        *penv = curenv;
        return 0;
    }
    //...
    return 0;
}
```

可以看到，当 envid 为 0 值时，函数会返回一个指向当前进程控制块的指针，因此保留 0 值是为了方便程序直接通过 mkenvid 函数访问当前进程的进程控制块。

Thinking 4.3

- 子进程完全按照 fork() 之后父进程的代码执行。说明子进程代码段和父进程共享同一片物理空间;
- 子进程不执行 fork() 之前父进程的代码，说明子进程恢复到的上下文位置为 fork() 函数，所以会执行之后的代码。

Thinking 4.4

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

在 0~USTACKTOP 范围内的内存，除了只读、共享页面外的页面都设置 PTE_COW 权限位进行保护。

Thinking 4.6

- vpt 是指向用户页表（Page Table）的指针数组的指针，以指向指针数组第一个元素的指针 *vpt 为基地址，加上页表项的偏移数后可以指向 va 对应的页表项，使用方法为 `((Pte*)(*vpt)) + (va >> 12)` ；而 vpd 是指向用户页目录（Page Directory）的指针，以指向数组中第一个元素的指针 *vpd 为基地址，加上页目录项的偏移数后可以指向 va 对应的页目录项，使用方法为 `((Pde*)(*vpd)) + (va >> 22)` .
- 因为在 user/entry.S 的有关定义中，可以看到 vpt 和 vpd 分别指向 UVPT 和 (UVPT + (UVPT >> 12) * 4) ，可以得到用户页表和用户页目录的虚拟地址。得到基地址后，再根据从虚拟地址中获得的偏移数就可以实现对进程自身页表的存取操作。
- 用户页目录虚拟地址 vpd 在 UVPT 和 (UVPT + PDMAP) 之间，是说明页目录被映射到了某一个页表的位置，实现了页目录自映射。
- 不能，因为在用户态不可以修改页表项。

Thinking 4.7

- 处理缺页中断时又发生了新的中断时；
- 因为需要在用户态根据异常现场的 Trapframe 处理异常。

Thinking 4.8

- 这一做法体现了微内核模式的优点，即让用户进程实现一部分内核的功能，在用户态处理可以尽量减少内核出现错误的可能，即使程序崩溃也不会影响系统的稳定，内核出了问题操作系统也可以运行；同时，在用户态下新页面的映射与分配也更加灵活方便；
- 通用寄存器的用途是用于保存 CPU 计算所需的数据以及结果，使得进入现场恢复阶段前，通用寄存器中的计算结果都已完成利用或者存入了相应区域，此时通用寄存器中的内容可以被覆盖而无须进行保护。因此用户空间下进行的现场恢复过程不会破坏通用寄存器。

Thinking 4.9

- 进程分配过程中也可能会发生缺页中断，需要进行异常处理；
- 由于无法处理缺页中断错误，会导致写时复制保护机制无法执行；
- 不需要，因为该值已经在父进程中被设置，子进程与父进程保持一致即可。

实验体会

本次实验给我带来的感觉是：读代码的过程更顺畅，但写代码更难。因为本单元的一个特点就是动辄十几个函数、文件彼此密切相关，有时为了实现一个功能往往要连锁地动很多地方（这一点在课上实验中也体现出来了），如果漏写或者写错了其中的一环都会导致错误。但这一特性的好处就是使得整个系统调用、进程通信以及 fork 的流程是环环相扣的，体现在整个流程的层次设计上，许多函数的命名也很有特点，方便了我的整理与理解，这样一来将代码顺着捋下来的过程逻辑感就很强，面对 MOS 似乎也不再有前几个单元那样强烈的“迷雾中寻道路”之感。

不过另一个深刻的体会是：上机跟之前几个单元真不是一个难度了（泪）