北航操作系统实验报告 Lab 4

学号：71066001

姓名：陈伟杰

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什么？

■

答：

- 内核保护现场的时候会将寄存器存到栈里，在保存现场之前，内核通常会使用指令如 push 和 pop 来将栈指针移动到正确的位置，以便寄存器的值可以正确地保存到栈中
- 是的，可以从 a0–a3 寄存器中获取用户调用 msyscall 时留下的信息，因为保存现场的过程没有破坏 a0-a3 寄存器的值
- 参数的传递依赖于 a0-a3 参数寄存器和栈，sys 开头的函数会从这些寄存器中获取参数。因此，我们只需要在调用 sys 函数时，将参数保存在 a0–a3 寄存器中并不变即可。内核处理系统调用时，会对 Trapframe 结构体做以下更改：保存被调用函数的返回地址，即 $ra 寄存器的值，到 Trapframe 结构体的相应位置。保存被调用函数的参数，即 $a0-$a3 寄存器的值，到 Trapframe 结构体的相应位置。将 Trapframe 结构体的 sp 寄存器指向内核栈，以便内核可以在栈上分配空间，变化是用户态的参数被移动到内核栈中，以便内核可以访问它们

Thinking 4.2 思考 envid2env 函数：为什么 envid2env 中需要判断 e->env_id != envid 的情况？如果没有这步判断会发生什么情况？ ■

答：在 envid2env 函数中，判断 e->env_id != envid 的情况是为了确保找到的环境 id 与传入的环境 id 相同。如果没有这个判断，可能会有这个情况：如果没有加这个判断，可能会返回找到的任意一个环境，这可能会导致程序出现错误，因为后续对该环境的操作可能会出错。

Thinking 4.3 思考下面的问题，并对这个问题谈谈你的理解：请回顾 kern/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。 ■

答：

```
5 u_int mkenvid(struct Env *e) {
6         static u_int i = 0;
7         return ((++i) << (1 + LOG2NENV)) | (e - envs);
8 }
```

该函数不会返回 0 是因为 i 的初始值为 0，执行完 ++i 后，i 的值就变成了 1，然后将其左移 (1 + LOG2NENV) 位，即将其乘以 2^(1 + LOG2NENV)，得到的结果不为 0，在 envid2env() 当 envid==0 的时候会返回当前进程的控制块，所以我们可以通过这个方法获取指向当前进程控制块的指针和访问

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 ■

答：C

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 kern/env.c 中 env_init 函数进行的页面映射、include/mmu.h 里的内存布局图以及本章的后续描述进行思考。 ■

答：在 0 ~ USTACKTOP 范围的内存需要进行映射,然后我们在 mmu.h 看到 USTACKTOP 往上的 user exception stack 无需映射

Thinking 4.6 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，请参考 user/include/lib.h 中的相关定义，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？

■

答：#define vpt ((volatile Pte *)UVPT)
#define vpd ((volatile Pde *)(UVPT + (PDX(UVPT)<<PGSHIFT)))

```
#define vpt ((volatile Pte *)UVPT)
#define vpd ((volatile Pde *)(UVPT + (PDX(UVPT) << PGSHIFT)))
```

vpt 宏定义实际上是将虚拟地址 UVPT 转换为一个指向 Pte 类型的指针，其中 Pte 表示页表项。vpd 宏定义实际上是将虚拟地址 UVPT 所在页目录的地址计算出来，并将其转换为一个指向 Pde 类型的指针，其中 Pde 表示页目录项。通过访问 vpd 指向的页目录项，内核可以获取到虚拟地址对应的页表的物理地址，以及该页表的属性信息等,在使用 vpt 和 vpd 时，可以通过指针访问它们指向的页表项和页目录项，从而获取虚拟地址所对应的物理地址，以及相应的页表属性信息。在使用 vpt 和 vpd 时，可以通过指针访问它们指向的页表项和页目录项，从而获取虚拟地址所对应的物理地址，以及相应的页表属性信息。

因为在 MIPS CPU 架构中页表和页目录都是通过虚拟地址映射到物理地址空间中的。因此，通过访问虚拟地址 UVPT 所对应的物理地址，就可以获取当前进程页表的起始地址也就是说这使得每个进程的页表都能在 UVPT 中保存，切换进程时，页表也会切换。

//PGSHIFT=虚拟地址中二级页表号的右移位数(12 位）
所以 vpd 的地址有偏移>>12 项得到也就是 UVPT 对应的页表项，显然这是自映射的特点，说明页表中存在某一页映射了整个页表

不能，因为用户态只有访问权限，只读页表项但不写，没有修改的权限

Thinking 4.7 在 do_tlb_mod 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“异常重入”的机制，而在什么时候会出现这种“异常重入”？
- 内核为什么需要将异常的现场 Trapframe 复制到用户空间？

■

答：

- 当 TLB 缺失异常的时候就会有异常重入机制
- 内核需要将异常的现场 Trapframe 复制到用户空间，在用户态程序中就可以通过访问堆栈中的 Trapframe 来获取异常现场的信息并根据需要进行处理。复制 Trapframe 到用户态堆栈的过程也是为了保护现场。

Thinking 4.8 在用户态处理页写入异常，相比于在内核态处理有什么优势？ ■

答：在用户态处理页写入异常相比于在内核态处理有这个优势：性能的提升和易于调试和维护，在用户态处理页写入异常可以减少从用户态到内核态的上下文切换，从而提高性能以及可以利用更多的用户态调试工具，因此对内核态的干扰较小。

Thinking 4.9 请思考并回答以下几个问题：

- 为什么需要将 syscall_set_tlb_mod_entry 的调用放置在 syscall_exofork 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？

■

答：

- syscall_exofork 这段代码调用 exofork 后会返回一个整数值子进程，如果该值为 0，则表示当前进程是新创建的子进程，否则表示当前进程是原始进程，将 syscall_set_tlb_mod_entry 调用在 exfork 之前是因为设置当前进程的 TLB 缺失异常处理函数。而在创建子进程时，需要将当前进程的地址空间和状态复制到子进程中，包括 TLB 缺失异常处理函数，因为如果不是这样那么子进程将无法正确地处理 TLB 缺失异常，从而可能导致异常退出或崩溃
- 写时复制保护机制会将父进程和子进程的页表项分离，从而可能导致子进程无法访问父进程的 TLB 缺失异常处理函数

实验困难：

我觉得这次 lab 主要最难在于 IPC 机制和 fork 函数和缺页中断这边的实现理解，在这里先贴指导书的 fork 流程图

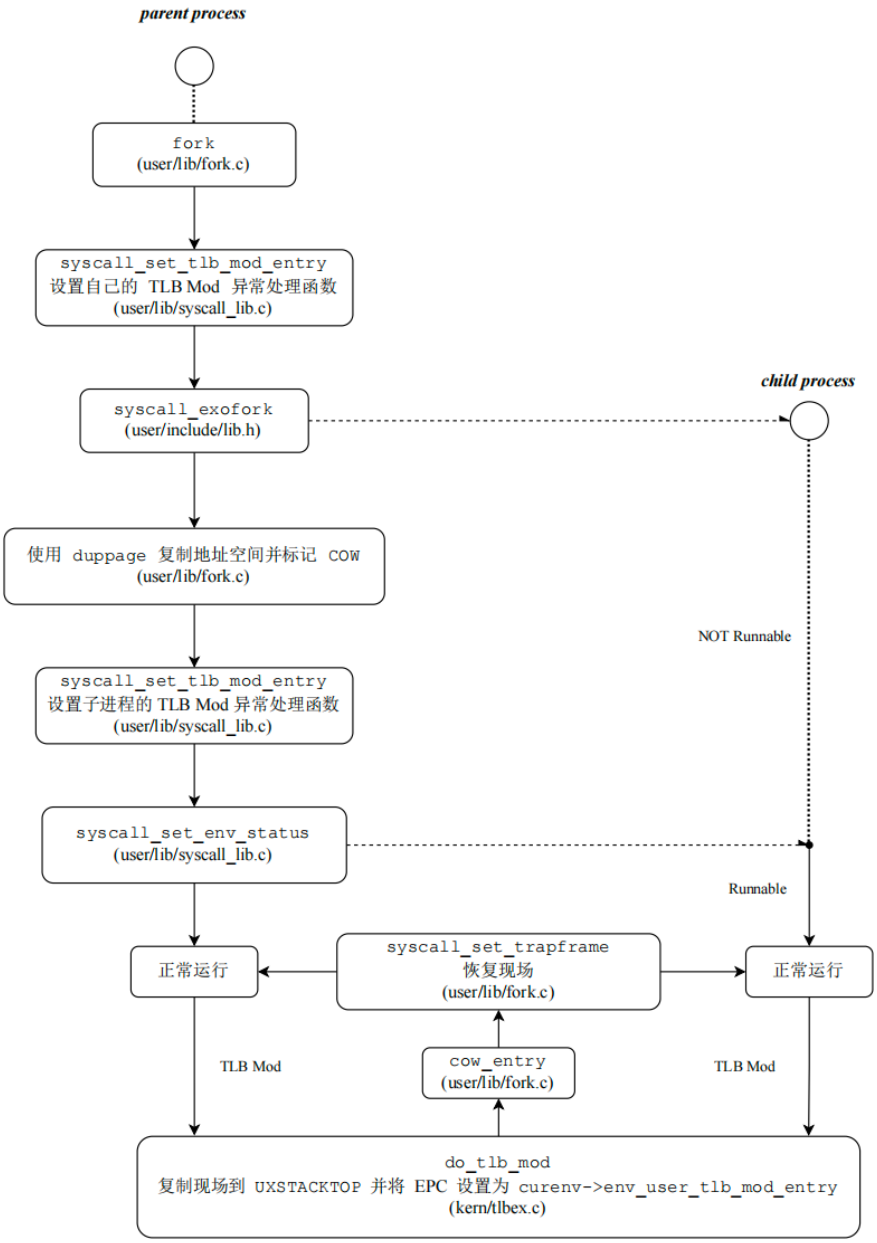
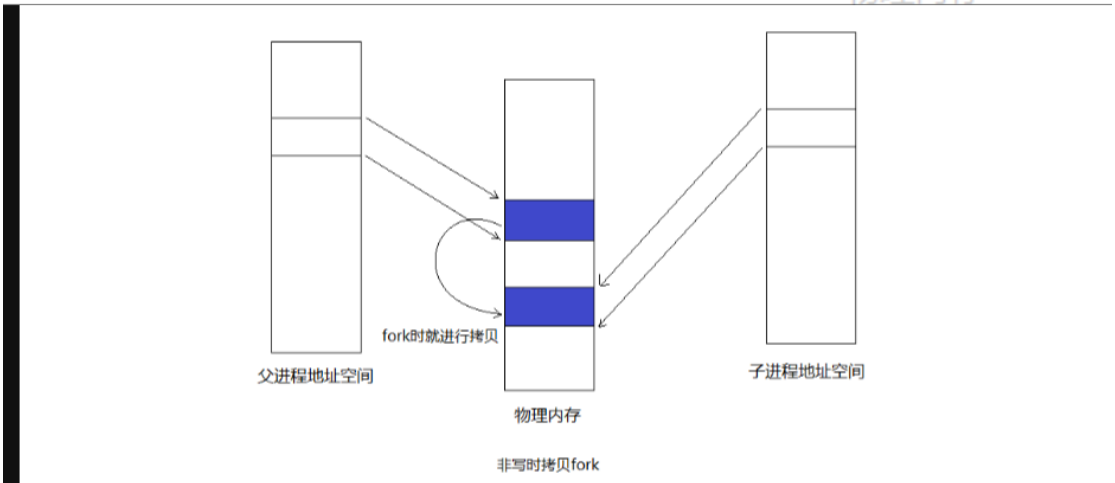
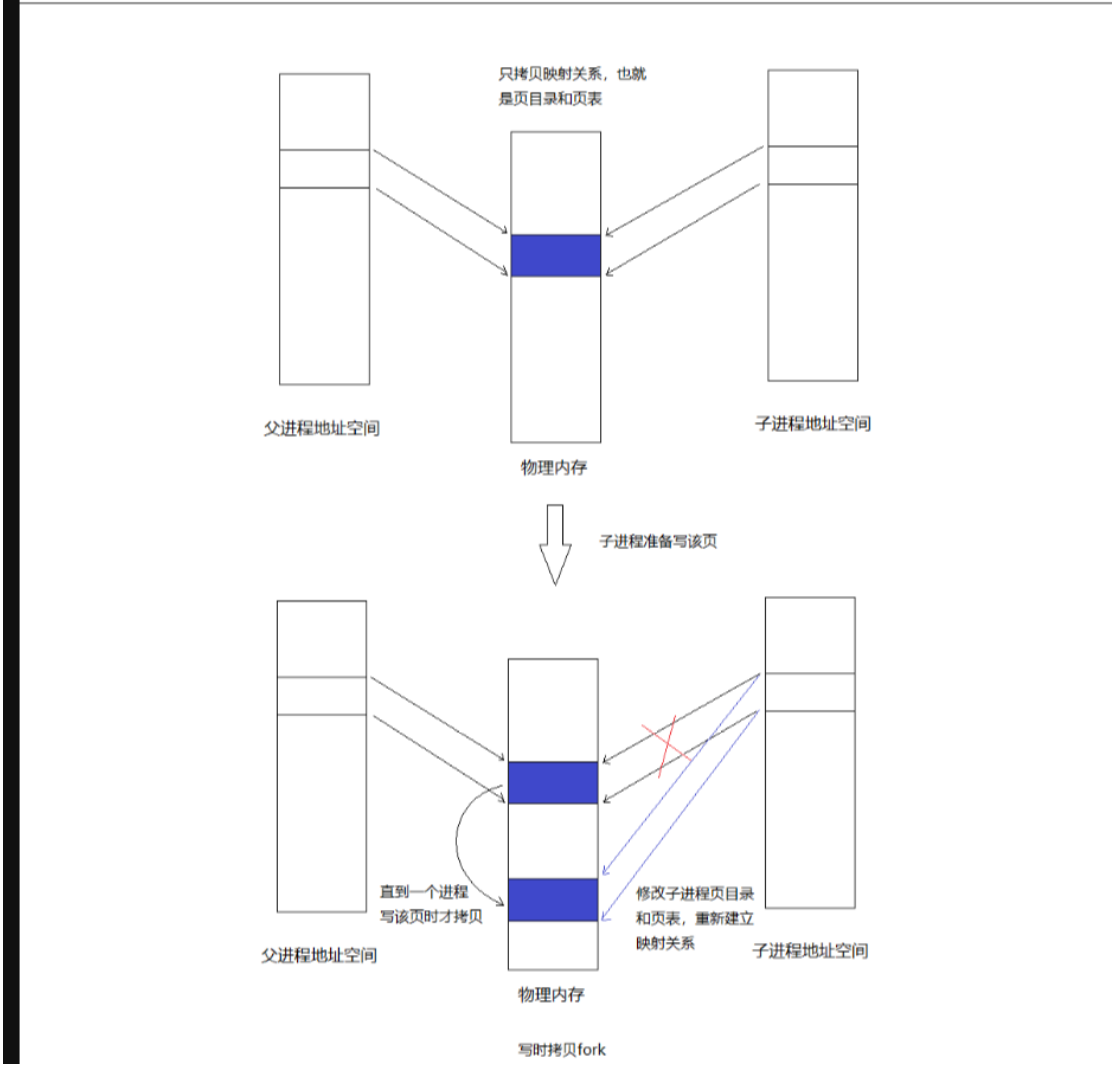


图 4.4: fork 流程图

//写时复制机制



非写时拷贝fork



写时拷贝fork

对于我来说在这次的最大难题是父进程和子进程之间的操作比如怎么为子进程分配异常处理栈，设置缺页处理函数入口，到用户态恢复现场，返回执行还有写时复制机制，我是看着上面这两张图才完成 fork 部分的，在 IPC 机制也就是进程间通信，首先围绕着我的点是它整个流程是怎么样的，数据接受方的那一方要做什么，数据发送方要怎么做

```
/* Step 2: Set 'curenv->env_ipc_recving' to 1. */
/* Exercise 4.8: Your code here. (1/8) */
curenv->env_ipc_recving = 1 ;
```

比如这个 当时我觉得难点在于我不理解 =1 的整个变化，=1 表明该进程准备接受其它进程的消息也就是说 sys_ipc_recv 这个函数所表达的意思就是将当前进程挂起，放弃 CPU，准备接受其他进程发来的消息，明白了这一点这一部分就好做了，因为下面那个是发送发，其所表达含义差不多一样。

体会感谢：在整个 lab4 我分为两部分去做，整体来说很困难，第一部分我只做到了 IPC 部分就已经花了我很长的时间，因为一直在花时间学习系统调用的概念及流程和实现进程间通讯机制，期间 debug 了好久，很多问题都要看回 lab3 来理解，但幸好看着指导书的图示来慢慢摸索才完成第一部份，在这个 lab 的第二部分就是需要实现 fork 函数和明白中断缺页处理流程，对我来说也是比较困难，通过 lab4，我对于 OS 系统调用和进程间协作的又学习到了很多，无论怎么样还是继续加油吧，对操作系统又认识了一点点。