CHAPTER 4

系统调用与 FORK

4.1 实验目的

1. 掌握系统调用的概念及流程
2. 实现进程间通讯机制
3. 实现 fork 函数
4. 掌握缺页中断的处理流程

一般情况下，进程不能够存取系统内核的地址空间，也就是说它不能存取内核使用的内存数据，也不能调用内核函数，这一点是由 CPU 的硬件结构保证的。然而，用户进程在特定的场景下是需要进行一些只能在内核中执行的操作，如对硬件的操作。这种时候允许内核执行用户提供的代码显然是不安全的，所以操作系统也就设计了一系列内核空间的函数，当用户进程以特定的方式陷入异常后，能够由内核调用对应的函数，我们把这些函数称为**系统调用**。在这一节的实验中，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（IPC）机制和一个重要的系统调用 fork。在 fork 的实验中，我们会介绍一种被称为写时复制的特性，而与这种特性相关的正是内核的缺页中断处理机制。

4.2 系统调用 (System Call)

本节中，我们着重讨论系统调用的作用，并完成实现相关的内容。

4.2.1 一探到底，系统调用的来龙去脉

说起系统调用，你冒出的第一个问题一定是：系统调用到底长什么样子？为了一探究竟，我们选择一个极为简单的程序作为实验对象。在这个程序中，我们通过 puts 来输出一个字符串到终端。

```c
#include <stdio.h>

int main() {
        puts("Hello World!\n");
        return 0;
}
```

Note 4.2.1 如果你还记得 C 语言课上关于标准输出的相关知识的话，你一定知道在 C 语言中，终端被抽象为了标准输出文件 stdout。通过向标准输出文件写东西，就可以输出内容到屏幕。而向文件写入内容是通过 write 系统调用完成的。因此，我们选择通过观察 puts 函数，来探究系统调用的奥秘。

我们通过 GDB 进行单步调试，逐步深入到函数之中，观察 puts 具体的调用过程[1]。运行 GDB，将断点设置在 puts 这条语句上，并通过 stepi 指令[2] 单步进入到函数中。当程序到达 write 函数时停下，因为 write 正是 Linux 的一条系统调用。我们打印出此时的函数调用栈，可以看出，C 标准库中的 puts 函数实际上通过了很多层函数调用，最终调用到了底层的 write 函数进行真正的屏幕打印操作。

```
(gdb)
0x00007ffff7b1b4e0 in write () from /lib64/libc.so.6
(gdb) backtrace
#0  0x00007ffff7b1b4e0 in write () from /lib64/libc.so.6
#1  0x00007ffff7ab340f in _IO_file_write () from /lib64/libc.so.6
#2  0x00007ffff7ab2aa3 in ?? () from /lib64/libc.so.6
#3  0x00007ffff7ab4299 in _IO_do_write () from /lib64/libc.so.6
#4  0x00007ffff7ab462b in _IO_file_overflow () from /lib64/libc.so.6
#5  0x00007ffff7ab5361 in _IO_default_xsputn () from /lib64/libc.so.6
#6  0x00007ffff7ab3992 in _IO_file_xsputn () from /lib64/libc.so.6
#7  0x00007ffff7aaa4ef in puts () from /lib64/libc.so.6
#8  0x0000000000400564 in main () at test.c:4
```

通过 gdb 显示的信息，我们可以看到，这个 write() 函数是在 libc.so 这个动态链接库中的，也就是说，它仍然是 C 库中的函数，而不是内核中的函数。因此，该 write() 函数依旧是个用户空间函数。为了彻底揭开这个函数的秘密，我们对其进行反汇编。

```
(gdb) disassemble 0x00007ffff7b1b4e0
Dump of assembler code for function write:
=> 0x00007ffff7b1b4e0 <+0>:     cmpl   $0x0,0x2bf759(%rip)        # 0x7ffff7ddac40
   0x00007ffff7b1b4e7 <+7>:     jne    0x7ffff7b1b4f9 <write+25>
   0x00007ffff7b1b4e9 <+9>:     mov    $0x1,%eax
   0x00007ffff7b1b4ee <+14>:    syscall
   0x00007ffff7b1b4f0 <+16>:    cmp    \$0xfffffffffffff001,%rax
   0x00007ffff7b1b4f6 <+22>:    jae    0x7ffff7b1b529 <write+73>
   0x00007ffff7b1b4f8 <+24>:    retq
End of assembler dump.
```

通过 gdb 的反汇编功能，我们可以看到，这个函数最终执行了 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

[1]这里为了方便大家在自己的机器上重现，我们选用了 Linux X86_64 平台作为实验平台

[2]为了加快调试进程，可以尝试 stepi N 指令，N 的位置填写任意数字均可。这样每次会执行 N 条机器指令。笔者使用的是 stepi 10。

核中的相应函数，完成相应的功能。在系统调用返回后，用户空间的相关函数会将系统调用的结果，通过一系列的过程，最终返回给用户程序。

由此我们可以看到，系统调用实际上是操作系统和用户空间的一组接口。用户空间的程序通过系统调用来访问操作系统的一些服务，谋求操作系统提供必要的帮助。

在进行了上面的一系列探究后，我们将我们的发现罗列出来，整理一下我们的思路:

- 存在一些只能由操作系统来完成的操作（如读写设备、创建进程等)。
- 用户程序要实现一些功能（比如执行另一个程序、读写文件)，必须依赖操作系统的帮助。
- C 标准库中的一些函数的实现必须依赖于操作系统（如我们所探究的 puts 函数)。
- 通过执行 syscall 指令，我们可以陷入到内核态，请求操作系统的一些服务。
- 直接使用操作系统的功能是很繁复的(每次都需要设置必要的寄存器,并执行 syscall 指令)

之后，我们再来整理一下调用 C 标准库中的 puts 函数的过程中发生了什么:

1. 调用 puts 函数
2. 在一系列的函数调用后，最终调用了 write 函数。
3. write 函数为寄存器设置了相应的值，并执行了 syscall 指令。
4. 进入内核态，内核中相应的函数或服务被执行。
5. 回到用户态的 write 函数中，将系统调用的结果从相关的寄存器中取回，并返回。
6. 再次经过一系列的返回过程后，回到了 puts 函数中。
7. puts 函数返回。

综合上面这些内容，相信你一定已经发现了其中的巧妙之处。操作系统将自己所能够提供的服务以系统调用的方式提供给用户空间。用户程序即可通过操作系统来完成一些特殊的操作。同时，所有的特殊操作就全部在操作系统的掌控之中了（因为用户程序只能通过由操作系统提供的系统调用来完成这些操作，所以操作系统可以确保用户不破坏系统的稳定) 。而直接使用这些系统调用较为麻烦，于是由产生了用户空间的一系列 API，如 POSIX、C 标准库等，它们在系统调用的基础上，实现更多更高级的常用功能，使得用户在编写程序时不用再处理这些繁琐而复杂的底层操作，而是直接通过调用高层次的 API 就能实现各种功能。通过这样巧妙的层次划分，使得程序更为灵活，也具有了更好的可移植性。对于用户程序来说，只要自己所依赖的 API 不变，无论底层的系统调用如何变化，都不会对自己造成影响，使得程序更易于在不同的系统间移植。整个结构如表4.1所示。

表 4.1: API、系统调用层次结构

用户程序 User Program	
应用程序编程接口 API	POSIX, C Standard Library, etc.
系统调用	read, write, fork, etc.

4.2.2 系统调用机制的实现

在发现了系统调用的本质之后，我们就要着手在我们的小操作系统中实现一套系统调用机制了。不过不要着急，为了使得后面的思路更清晰，我们先来整理一下系统调用的流程：

1. 调用一个封装好的用户空间的库函数（如 writef）
2. 调用用户空间的 syscall_* 函数
3. 调用 msyscall，用于陷入内核态
4. 陷入内核，内核取得信息，执行对应的内核空间的系统调用函数（sys_*）
5. 执行系统调用，并返回用户态，同时将返回值“传递”回用户态
6. 从库函数返回，回到用户程序调用处

在用户空间的程序中，我们定义了许多的函数，以 writef 函数为例，这一函数实际上并不是最接近内核的函数，它最后会调用一个名为 syscall_putchar 的函数，这个函数在 user/syscall_lib.c 中。在我们的小操作系统实验中，这些 syscall 开头的函数与内核中的系统调用函数（sys 开头的函数）是一一对应的，syscall 开头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 开头的函数是内核中系统调用具体内容。syscall 开头的函数的实现中，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），在我们的小操作系统实验中把这个参数称为**系统调用号**（请找到这个宏的定义，了解系统调用号的排布规则），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实际需要使用的参数，而为了方便我们使用了取了最多参数的系统调用所需要的参数数量（syscall_mem_map 函数具有 5 个参数）。

在 syscall_* 系列函数中，我们将参数传递给了 msyscall 函数，而这些参数究竟是如何安置的呢？这里就需要用 MIPS 的调用规范来说明这件事情了，我们把函数体中没有函数调用语句的函数称为**叶函数**，自然如果有函数调用语句的函数称为非叶函数。在 MIPS 的调用规范中，进入函数体时会通过对栈指针做减法的方式为自身的局部变量、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来释放这部分空间，我们把这部分空间称为**栈帧（Stack Frame）**。非叶函数是在调用方的栈帧的底部预留被调用函数的参数存储空间（被调用方从调用方函数的栈帧中取得参数）。以我们的小操作系统为例，msyscall 函数一共有 6 个参数，前 4 个参数会被 syscall 开头的函数分别存入 $a0-$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前 4 的参数的预留空间之上的 8 字节空间内（没有寄存器传参）。这些过程虽然不需要我们显式地编写汇编来完成，但是需要在内核中是以汇编的方式显式地把函数的参数值“转移”到内核空间中的。

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）来陷入内核态以及函数调用返回即可。

Exercise 4.1 填写 user/syscall_wrap.S 中的 msyscall 函数，使得用户部分的系统调用机制可以正常工作。 ■

在通过特权指令 syscall 陷入内核态后，处理器将 PC 寄存器指向一个相同的内核异常入口。在 trap_init 函数中将系统调用类型的异常的入口设置为了 handle_sys 函数，这一函数在 lib/syscall.S 中。需要注意的是，此处的栈指针是内核空间的栈指针，内核将运行现场保存到内核空间后（其保存的结构与结构体`struct Trapframe`等同），栈指针指向这个结构体的起始位置，你可以借助 include/trap.h 的宏使用 lw 指令取得保存现场的一些寄存器的值。

Listing 14: 内核的系统调用处理程序

```
1  NESTED(handle_sys,TF_SIZE, sp)
2      SAVE_ALL                                /* 用于保存所有寄存器的汇编宏 */
3      CLI                                     /* 用于屏蔽中断位的设置的汇编宏 */
4      nop
5      .set at                                 /* 恢复$at 寄存器的使用 */
6
7      /* TODO: 将Trapframe 的 EPC 寄存器取出, 计算一个合理的值存回Trapframe 中 */
8
9      /* TODO: 将系统调用号“复制”入寄存器$a0 */
10
11     addiu   a0, a0, -__SYSCALL_BASE         /* a0 <- “相对”系统调用号 */
12     sll     t0, a0, 2                       /* t0 <- 相对系统调用号 * 4 */
13     la      t1, sys_call_table              /* t1 <- 系统调用函数的入口表基地址 */
14     addu    t1, t1, t0                      /* t1 <- 特定系统调用函数入口表项地址 */
15     lw      t2, 0(t1)                       /* t2 <- 特定系统调用函数入口函数地址 */
16
17     lw      t0, TF_REG29(sp)                /* t0 <- 用户态的栈指针 */
18     lw      t3, 16(t0)                      /* t3 <- msyscall 的第 5 个参数 */
19     lw      t4, 20(t0)                      /* t4 <- msyscall 的第 6 个参数 */
20
21     /* 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 */
22
23
24     jalr    t2                              /* 调用sys_*函数 */
25     nop
26
27     /* TODO: 恢复栈指针到分配前的状态 */
28
29     sw      v0, TF_REG2(sp)                 /* 将$v0 中的 sys_*函数返回值存入Trapframe */
30
```

```
31      j       ret_from_exception          /* 从异常中返回（恢复现场） */
32      nop
33  END(handle_sys)
34
35  sys_call_table:                         /* 系统调用函数的入口表 */
36  .align 2
37      .word sys_putchar
38      .word sys_getenvid
39      .word sys_yield
40      .word sys_env_destroy
41      .word sys_set_pgfault_handler
42      .word sys_mem_alloc
43      .word sys_mem_map
44      .word sys_mem_unmap
45      .word sys_env_alloc
46      .word sys_set_env_status
47      .word sys_set_trapframe
48      .word sys_panic
49      .word sys_ipc_can_send
50      .word sys_ipc_recv
51      .word sys_cgetc
52      /* 每一个整字都将初值设定为对应sys_*函数的地址 */
53      /* 在此处增加内核系统调用的入口地址 */
```

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？

■

Exercise 4.2 按照 lib/syscall.S 中的提示，完成 handle_sys 函数，使得内核部分的系统调用机制可以正常工作。 ■

做完这一步，整个系统调用的机制已经可以正常工作，接下来我们要来实现几个具体的系统调用。

4.2.3 基础系统调用函数

在系统调用机制搞定之后，我们自然是要弄几个系统调用玩一玩了。我们实现些什么系统调用呢？打开 lib/syscall_all.c，可以看到玲琅满目的系统调用函数等着我们去填写。这些系统调用都是基础的系统调用，不论是之后的 IPC 还是 fork，都需要这些基础

的系统调用作为支撑。首先我们看向 sys_mem_alloc。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，需要用到一些我们之前在 pmap.c 中所定义的函数

Exercise 4.3 实现 lib/syscall_all.c 中的 int sys_mem_alloc(int sysno,u_int envid, u_int va, u_int perm) 函数 ■

我们再来看 sys_mem_map，这个函数的参数很多，但是意义也很直接：将源进程地址空间中的相应内存映射到目标进程的相应地址空间的相应虚拟内存中去。换句话说，此时两者共享着一页物理内存。

Exercise 4.4 实现 lib/syscall_all.c 中的 int sys_mem_map(int sysno,u_int srcid, u_int srcva, u_int dstid, u_dstva, u_int perm) 函数 ■

关于内存的还有一个函数：sys_mem_unmap, 正如字面意义所显示的，这个系统调用的功能是解除某个进程地址空间虚拟内存和物理内存之间的映射关系。

Exercise 4.5 实现 lib/syscall_all.c 中的 int sys_mem_unmap(int sysno,u_int envid, u_int va) 函数 ■

除了与内存相关的函数外，另外一个常用的系统调用函数是 sys_yield，这个函数的功能主要就在于实现用户进程对 CPU 的放弃，可以利用我们之前已经编写好的函数，另外为了通过我们之前编写的进程切换机制保存现场，这里需要在 KERNEL_SP 和 TIMESTACK 上做一点准备工作

Exercise 4.6 实现 lib/syscall_all.c 中的 void sys_yield(void) 函数 ■

可能你也注意到了，在此我们的系统调用函数并没使用到它的第一个参数 sysno，在这里，sysno 作为系统调用号被传入，现在起的更多是一个”占位“的作用，能和之前用户层面的系统调用函数参数顺序相匹配。

4.3 进程间通信机制 (IPC)

在系统调用机制搞定之后，我们自然是要弄几个系统调用玩一玩了。作为一个微内核系统，我们要来实现个什么系统调用呢？没错，当然是 IPC 了。IPC 可是微内核最重要的机制之一了。

Note 4.3.1 上世纪末，微内核设计逐渐成为了一个热点。微内核设计主张将传统操作系统中的设备驱动、文件系统等可在用户空间实现的功能，移出内核，作为普通的用户程序来实现。这样，即使它们崩溃，也不会影响到整个系统的稳定。其他应用程序通过进程间通讯来请求文件系统等相关服务。因此，在微内核中 IPC 是一个十分重要的机制。

接下来进入正题，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

度简化了的小操作系统中。根据之前的讨论，我们能够得知这样几个细节:

- 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

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。那么，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**。相信你在实现内存管理的时候已经深刻体会到了这一点，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。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，最好的方式是什么呢？对，我们要去找一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。如果你之前详细地看过进程的页表建立的部分的话，想必你已经找到线索了。是的，线索就在 env_setup_vm() 这个函数里面。

```
static int
env_setup_vm(struct Env *e)
{
    //略去的无关代码

    for (i = PDX(UTOP); i <= PDX(~0); i++) {
        pgdir[i] = boot_pgdir[i];
    }
    e->env_pgdir = pgdir;
    e->env_cr3   = PADDR(pgdir);

    //略去的无关代码
}
```

如果你之前认真思考了这个函数的话会发现，所有的进程都共享了内核所在的 2G 空间。对于任意的进程，这 2G 都是一样的。因此，想要在不同空间之间交换数据，我们就需要借助于内核的空间来实现。那么，我们把要传递的消息放在哪里比较好呢？恩，发送和接受消息和进程有关，消息都是由一个进程发送给另一个进程的。内核里什么地方和进程最相关呢？啊哈！进程控制块！

```
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
};
```

果然，我们看到了我们想要的东西，env_ipc_value 用于存放需要发给当前进程的数据。env_ipc_dstva 则说明了接收到的页需要被映射到哪个虚地址上。知道了这些，我们就不难实现 IPC 机制了。只需要做做赋值，填充下对应的域，映射下该映射的页之类的就好了。

Exercise 4.7 实现 lib/syscall_all.c 中的 void sys_ipc_recv(int sysno,u_int dstva) 函数和 int sys_ipc_can_send(int sysno,u_int envid, u_int value, u_int srcva, u_int perm) 函数。 ■

sys_ipc_recv(int sysno,u_int dstva) 函数首先要将 env_ipc_recving 设置为 1，表明该进程准备接受其它进程的消息了。之后阻塞当前进程，即将当前进程的状态置为不可运行。之后放弃 CPU（调用相关函数重新进行调度)。

int sys_ipc_can_send(int sysno,u_int envid, u_int value, u_int srcva, u_int perm) 函数用于发送消息。如果指定进程为可接收状态，则发送成功，清除接收进程的接收状态，使其可运行，返回 0，否则，返会 _E_IPC_NOT_RECV。

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 的调用来表示不需要传递物理页面，因此在编写相关函数时也需要注意此种情况。

讲完 IPC 后你已经可以参照编写用户进程，利用实现好的 IPC 系统调用来实现一些有意思的小程序了。

4.4 FORK

在 Lab3 我们曾提到过,env_alloc 是内核产生一个进程。但如果想让一个进程创建一个进程，就像是父亲与儿子那样，我们就需要使用到 fork 了。那么 fork 究竟是什么呢？

4.4.1 初窥 fork

fork，直观意象是叉子的意思。在我们这里更像是分叉的意思，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，在某个进程中调用 fork() 之后，将会以此为分叉分成两个进程运行。新的进程在开始运行时有着和旧进程**绝大部分相同的信息**，而且在新的进程中 fork 依旧有一个返回值，只是该返回值为 0。在旧进程，也就是所谓的父进程中，fork 的返回值是子进程的 env_id，是大于 0 的。在父子进程中有不同的返回值的特性，可以让我们在使用 fork 后很好地区分父子进程，从而安排不同的工作。

你可能会想，fork 执行完为什么不直接生成一个空白的进程块，生成一个几乎和父进程一模一样的子进程有什么用呢？换成创建一个空白的进程多简单！按笔者的理解，这是因为：

- 与不相干的两个进程相比，父子进程间的通信要方便的多。因为 fork 虽然没法造成进程间的统治关系[3]，但是因为在子进程中记录了父进程的一些信息，父进程也可以很方便地对子进程进行一些管理等。
- 当然还有一个可能的原因在于安全与稳定，尤其是关于操作权限方面。对这方面有兴趣的同学可以查看链接[4] 探索一下。

[3]这是因为进程之间是并发的，在操作系统看来，父子进程之间更像是兄弟关系。

[4]http://www.jbxue.com/shouce/apache2.2/mod/prefork.html

fork 之后父子进程就分道扬镳，互相独立了。而和 fork“针锋相对”却又经常“纠缠不清”的，是名为 exec 系列的系统调用。它会“勾引”子进程抛弃现有的一切，另起炉灶。若在子进程中执行 exec，完成后子进程从父进程那拷贝来的东西就全部消失了。取而代之的是一个全新的进程，就像太乙真人用莲藕为哪吒重塑了一个肉身一样。exec 系列系统调用我们将会作为一个挑战性任务放在后面来实现，暂时不做过多介绍。

为了让你对 fork 的认识不只是停留在理论层面，我们下面来做一个小实验，复制到你的 linux 环境下运行一下吧。

Listing 15: 理解 fork

```
#include<stdio.h>
#include<sys/types.h>

int main(void){
        int var;
        pid_t pid;
        printf("Before fork.\n");
        pid = fork();
        printf("After fork.\n");
        if(pid==0){
                printf("son.");
        }else{
                sleep(2);
                printf("father.");
        }
        printf("pid:%d\n",getpid());
        return 0;
}
```

使用`gcc fork_test.c`，然后 `./a.out` 运行一下，你得到的正常的结果应该如下所示：

```
Before fork.
After fork.
After fork.
son.pid:16903 (数字不一定一样)
father.pid:16902
```

我们从这段简短的代码里可以获取到很多的信息，比如以下几点：

- 在 fork 之前的代码段只有父进程会执行。
- 在 fork 之后的代码段父子进程都会执行。
- fork 在不同的进程中返回值不一样，在父进程中返回值不为 0，在子进程中返回值为 0。
- 父进程和子进程虽然很多信息相同，但他们的 env_id 是不同的。

从上面的小实验我们也能看出来——子进程实际上就是按父进程的绝大多数信息和状态作为模板而雕琢出来的。即使是以父进程为模板，父子进程也还是有很多不同的地方，不知细心的你从刚才的小实验中能否看出父子进程有哪些地方是明显不一样的吗？

Thinking 4.2 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？

■

Thinking 4.3 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 ■

4.4.2 写时复制机制

通过使用初步了解 fork 后，先不着急实现它。俗话说“兵马未动，粮草先行”，我们先来了解一下关于 fork 的底层细节。根据维基百科的描述，在 fork 时，父进程会为子进程分配独立的地址空间。但是分配独立的虚拟空间并不意味着一定会分配额外的物理内存：父子进程用的是相同的物理空间。子进程的代码段、数据段、堆栈都是指向父进程的物理空间，也就是说，虽然两者的虚拟空间是不同的，但是他们所对应的物理空间是同一个。

Note 4.4.1 Wiki Fork: In Unix systems equipped with virtual memory support (practically all modern variants), the fork operation creates a separate address space for the child. The child process has an exact copy of all the memory segments of the parent process, though if copy-on-write semantics are implemented,the physical memory need not be actually copied. Instead, virtual memory pages in both processes may refer to the same pages of physical memory until one of them writes to such a page: then it is copied. This optimization is important in the common case where fork is used in conjunction with exec to execute a new program: typically, the child process performs only a small set of actions before it ceases execution of its program in favour of the program to be started, and it requires very few, if any, of its parent's data structures.

那你可能就有问题了：既然上文提到了父子进程之间是独立的，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，那岂不是变成了“父教子从，子不得不从”？

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）。通俗来讲就是当父子进程中有**修改**内存（一般是数据段）的行为发生时，内核捕获这种缺页中断后，再为**发生内存修改的进程**相应的地址分配物理页面，而一般来说子进程的代码段继续共享父进程的物理空间（两者的代码完全相同）。

> **Note 4.4.2** 如果在 fork 之后在子进程中执行了 exec，由于这时和父进程要执行的代码完全不同，子进程的代码段也会分配单独的物理空间。

在我们的小操作系统实验中，对于所有的可被写入的内存页面，都需要**通过设置页表项标识位 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物理页，就会产生一个异常（一般指缺页中断 Page Fault），这一异常的处理会在后文详细介绍。

> **Note 4.4.3** 早期的 Unix 系统对于 fork 采取的策略是：直接把父进程所有的资源复制给新创建的进程。这种实现过于简单，并且效率非常低。因为它拷贝的内存也许是需要父子进程共享的，当然更糟的情况是，如果新进程打算通过 exec 执行一个新的映像，那么所有的拷贝都将前功尽弃。

4.4.3 返回值的秘密

小红：“咦，不科学啊。fork 的两个返回值为啥是系统调用 syscall_env_alloc 的功劳？不是说子进程只执行 fork 之后的代码吗？”

小绿：“你还别不信，还真的就是系统调用 syscall_env_alloc 的功劳。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 内部呀，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呀！你想想，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。

在我们的小操作系统实验中，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一个新的进程的创建 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片段：

```
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，我们可以认为它们都应该经历了同样的“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的恢复现场，而对于子进程则是在进程调度时进行的现场恢复。在现场恢复后，进程会从同样的地方返回到 fork 函数中。而它们携带的函数的返回值是不同的，这也就能够在 fork 函数中区分两者。

为了实现这一特性，你可能需要先实现 sys_env_alloc 的几个任务，它除了创建一个新的进程外，还需要用一些当前进程的信息作为模版来填充这个进程：

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

程序计数器 子进程的程序计数器应该被设置为 syscall_env_alloc 返回后的地址，也就是它陷入异常地址的下一行指令的地址，**这个值已经存在于 Trapframe 中**。

返回值有关 这个系统调用本身是需要一个返回值的（这个返回过程只会影响到父进程），对于子进程则需要对它的运行现场 Trapframe 进行一个修改。

进程状态 我们当然不能让子进程在父进程 syscall_env_alloc 返回后就直接进入调度，因为这时候它还没有做好充分的准备，所以我们需要设定不能让它被加入调度队列。

Exercise 4.8 填写 lib/syscall_all.c 中的 sys_env_alloc 函数 ■

在解决完返回值的问题之后，父与子就能够分别走上各自的旅途了。

4.4.4 父子各自的旅途

进程在很多时候（如进程通信）都是需要访问自身的进程控制块的，用户程序初次运行时会将一个 `struct Env *env` 指针指向自身的进程控制块。作为子进程，它很明显具有了一个与父亲不同的进程控制块，所以在第一次被调度的时候（当然这时还是在 fork 函数中）它就需要将 env 指针指向自身的进程控制块，这个时候还需要通过另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。做完这一步，子进程就可以从 fork 函数中返回，开始自己的旅途了。

Exercise 4.9 填写 user/fork.c 中的 fork 函数中关于 sys_env_alloc 的部分和“子进程”执行的部分 ■

父亲在儿子醒来之前则需要做更多的准备，而这些准备中最重要的一步是遍历进程的**大部分用户空间页**，对于所有可以写入的页面的页表项，**在父进程和子进程都**加以 PTE_COW 标志位保护起来。这里需要实现 duppage 函数来完成这个过程。

Thinking 4.4 如果仔细阅读上述这一段话，你应该可以发现，我们并不是对所有的用户空间页都使用 duppage 进行了保护。那么究竟哪些用户空间页可以保护，哪些不可以呢，请结合 include/mmu.h 里的内存布局图谈谈你的看法。 ■

Thinking 4.5 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能够通过这种方式来存取进程自身页表？
- 它们是如何体现自映射设计的？

- 进程能够通过这种存取的方式来修改自己的页表项吗？

■

在 duppage 函数中，唯一需要强调的一点是要对不同权限的页有着不同的处理方式，你可能会遇到这几种情况：

只读页面 按照相同权限（只读）映射给子进程即可

共享页面 即具有 PTE_LIBRARY 标记的页面，这类页面需要保持共享的可写的状态

写时复制页面 即具有 PTE_COW 标记的页面，这类页面是上一次的 fork 的 duppage 的结果

可写页面 需要给父进程和子进程的页表项都加上 PTE_COW 标记

Exercise 4.10 结合注释，填写 user/fork.c 中的 duppage 函数 ■

Note 4.4.4 在我们的小操作系统实验中实现的 fork 并不是一整个原子的过程，所以会出现一段时间（也就是在 duppage 之前的时间）我们没有来得及为堆栈所在的页面设置写时复制的保护机制，在这一段时间内对堆栈的修改（比如发生了其他的函数调用），则会将非叶函数 syscall_env_alloc 函数调用的栈帧中的返回地址覆盖。这一问题对于父进程来说是理所当然的，然而对于子进程来说，这个覆盖导致的后果则是在从 syscall_env_alloc 返回时跳转到一个不可预知的位置造成 panic。当然你现在看到的代码已经通过一个优雅的办法来修补这个臭虫：与其他系统调用函数不同，syscall_env_alloc 是一个内联（inline）的函数，也就是说这个函数并不会被编译为一个函数，而是直接内联展开在 fork 函数内。所以 syscall_env_alloc 的栈帧就不存在了，而 msyscall 函数的返回指令也直接返回到了 fork 函数内。如此这个困扰了学生若干年的臭虫就解决了。

在完成写时复制的保护机制后，还不能让子进程处于能被调度的状态，因为作为父亲它还有其他的责任——为写时复制特性的**缺页中断**处理做好准备。

4.4.5 缺页中断

内核在捕获到一个常规的**缺页中断（Page Fault）**时（在 MIPS 中这个情况特指 TLB 缺失，因为 MIPS 不存在 MMU 只存在 TLB，TLB 缺失查找填入都是内核以软件编程的方式完成的），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 TLB 填入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，在存取地址合法的情况下，内核会在用户空间的对应地址分配映射一个物理页面（被动的分配页面）来解决缺页的问题。

前文中我们提到了写时复制特性，而写时复制特性也是依赖于缺页中断的。我们在 trap_init 中注册了另外一个处理函数——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缺页中断的内核处理函数。

你会发现在这个函数中似乎并没有做任何的页面复制操作，这是为什么呢？答案是这样的，在我们的小操作系统实验中按照微内核的设计理念，会将大部分的功能都从内核移到用户程序，其中也包括了写时复制的缺页中断处理。真正的处理过程是用户进程自身去完成的。

如果需要用户进程去完成页面复制等处理过程，是不能直接使用原先的堆栈的(因为发生缺页错误的也可能是正常堆栈的页面)，所以这个时候用户进程就需要一个另外的堆栈来执行处理程序，我们把这个堆栈称作 **异常处理栈**，它的栈顶对应的是宏 UXSTACKTOP。异常处理栈需要父进程为自身以及子进程分配映射物理页面。此外内核还需要知晓进程自身的处理函数所在的地址，这个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。

Exercise 4.11 完成 lib/traps.c 中的 page_fault_handler 函数 ■

Thinking 4.6 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？

■

让我们回到 fork 函数，在提示使用 syscall_env_alloc 之前，有另一个提示——使用 set_pgfault_handler 函数来“安装”处理函数。

```
1  void set_pgfault_handler(void (*fn)(u_int va))
2  {
3    if (__pgfault_handler == 0) {
4      if (syscall_mem_alloc(0, UXSTACKTOP - BY2PG, PTE_V | PTE_R) < 0 ||
5        syscall_set_pgfault_handler(0, __asm_pgfault_handler, UXSTACKTOP) < 0) {
6        writef("cannot set pgfault handler\n");
7        return;
8      }
9    }
10   __pgfault_handler = fn;
11 }
```

上面的 set_pgfault_handler 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)，随后内核也需要就将进程控制块的 env_pgfault_handler 域设为它。在函数的最后，将在 entry.S 定义的字 ___pgfault_handler 赋值为 fn，而这个 fn 究竟是什么我们稍后再说。这里需要你完成内核设置进程控制块中的两个域的系统调用。

Exercise 4.12 完成 lib/syscall_all.c 中的 sys_set_pgfault_handler 函数 ■

我们现在知道了缺页中断会返回到 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:
lw      a0, TF_BADVADDR(sp)
lw      t1, __pgfault_handler
jalr    t1
nop

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)
```

从内核返回后，此时的栈指针是由内核设置的在异常处理栈的栈指针，而且指向一个由内核复制好的 Trapframe 结构体的底部。通过宏 TF_BADVADDR 用 lw 指令取得了 Trapframe 中的 cp0_badvaddr 字段的值，这个值也正是发生缺页中断的虚拟地址。将这个地址作为第一个参数去调用了 ___pgfault_handler 这个字内存储的函数，不难看出这个函数是真正进行处理的函数。函数返回后就是一段类似于恢复现场的汇编，最后非常巧妙地利用了 MIPS 的延时槽特性跳转的同时恢复了栈指针。

Thinking 4.7 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？

■

说到这里，我们就要来实现真正进行处理的函数：user/fork.c 中的 pgfault 函数了，pgfault 需要完成这些任务：

1. 判断页是否为写时复制的页面，是则进行下一步，否则报错
2. 分配一个新的内存页到临时位置，将要复制的内容拷贝到刚刚分配的页中
3. 将临时位置上的内容映射到发生缺页中断的虚拟地址上，然后解除临时位置对内存的映射

Exercise 4.13 填写 user/fork.c 中的 pgfault 函数 ■

这里的 pgfault 也正是父进程在 fork 中使用 set_pgfault_handler 函数安装的处理函数。

Thinking 4.8 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？
- 子进程需不需要对在 entry.S 定义的字 ___pgfault_handler 赋值？

■

父进程还需要为子进程通过类似于 set_pgfault_handler 函数的方式，用若干系统调用分配子进程的异常处理栈以及设置处理函数为 ___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。在实现内核 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 status 为 RUNNABLE 时将进程加入可以调度的链表。

Exercise 4.14 填写 lib/syscall_all.c 中的 sys_set_env_status 函数 ■

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：**地址空间的遍历以及 duppage、分配子进程的异常处理栈、设置子进程的处理函数、设置子进程的运行状态**。最后再将子进程的 envid 返回就大功告成了。

Exercise 4.15 填写 user/fork.c 中的 fork 函数中关于“父进程”执行的部分 ■

至此，我们的 lab4 实验已经基本完成了，接下来就一起来愉快地调试吧！

4.5 实验正确结果

本次测试分为两个文件，当基础系统调用与 fork 写完后，单独测试 fork 的文件是 **user/fktest.c**，测试时将

ENV_CREATE(user_fktest) 加入 init.c 即可测试。

正确结果如下：

```
1  ^^Imain.c:^^Imain is start ...
2
3  ^^Iinit.c:^^Imips_init() is called
4
5  ^^IPhysical memory: 65536K available, base = 65536K, extended = 0K
6
7  ^^Ito memory 80401000 for struct page directory.
8
9  ^^Ito memory 80431000 for struct Pages.
10
```

```
^^Imips_vm_init:boot_pgdir is 80400000

^^Ipmap.c:^^I mips vm init success

^^Ipanic at init.c:31: ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

^^Ipageout:^^I@@@___0x7f3fe000___@@@  ins a page

^^Ithis is father: a:1

^^Ithis is father: a:1

^^Ithis is father: a:1

^^Ithis is father: a:1

^^Ithis is father: a:1

^^Ithis is father: a:1

^^Ithis is father: a:1

^^Ithis is father: a:1

^^Ithis is father: a:1

^^I  child :a:2

^^I^^Ithis is child :a:2

^^I^^Ithis is child :a:2

^^I^^I^^I^^Ithis is child2 :a:3

^^I^^I^^I^^Ithis is child2 :a:3

^^I^^I^^I^^Ithis is child2 :a:3

^^I^^I^^I^^Ithis is child2 :a:3

^^Ithis is father: a:1

^^Ithis is father: a:1

^^Ithis is father: a:1

^^Ithis is father: a:1

^^Ithis is father: a:1

^^I^^Ithis is child :a:2

^^I^^Ithis is child :a:2

^^I^^Ithis is child :a:2
```

另一个测试文件主要测试进程间通信, 文件为 **user/pingpong.c**, 测试方法同上。正确结果如下:

```
main.c:^^Imain is start ...

init.c:^^Imips_init() is called

Physical memory: 65536K available, base = 65536K, extended = 0K

to memory 80401000 for struct page directory.

to memory 80431000 for struct Pages.

mips_vm_init:boot_pgdir is 80400000

pmap.c:^^I mips vm init success

panic at init.c:31: ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

pageout:^^I@@@___0x7f3fe000___@@@  ins a page


@@@@@send 0 from 800 to 1001

1001 am waiting.....

800 am waiting.....

1001 got 0 from 800

@@@@@send 1 from 1001 to 800

1001 am waiting.....

800 got 1 from 1001



@@@@@send 2 from 800 to 1001

800 am waiting.....

1001 got 2 from 800



@@@@@send 3 from 1001 to 800

1001 am wa800 got 3 from 1001



@@@@@send 4 from 800 to 1001

iting.....

800 am waiting.....

1001 got 4 from 800


```

```
@@@@@send 5 from 1001 to 800

1001 am waiting.....

800 got 5 from 1001


@@@@@send 6 from 800 to 1001

800 am waiting.....

1001 got 6 from 800


@@@@@send 7 from 1001 to 800

1001 am waiting.....

800 got 7 from 1001


@@@@@send 8 from 800 to 1001

800 am waiting.....

1001 got 8 from 800


@@@@@send 9 from 1001 to 800

1001 am waiting.....

800 got 9 from 1001


@@@@@send 10 from 800 to 1001

[00000800] destroying 00000800

[00000800] free env 00000800

i am killed ...

1001 got 10 from 800

[00001001] destroying 00001001

[00001001] free env 00001001

i am killed ...
```

4.6 实验思考

- 思考-系统调用的实现
- 思考-不同的进程代码执行
- 思考-fork 的返回结果
- 思考-用户空间的保护
- 思考-vpt 的使用
- 思考-缺页中断-内核处理
- 思考-缺页中断-用户处理-1
- 思考-缺页中断-用户处理-2

4.7 挑战性任务——线程和信号量

线程（thread）和信号量（semaphore）是同步与互斥问题的重要概念。

在这个挑战任务中，我们希望你能够按照 POSIX 标准，在小操作系统中实现这两个机制。

4.7.1 POSIX Threads

POSIX Threads 也就是 pthreads 库是由 IEEE Std 1003.1c 标准定义的一组函数接口。它至少包含以下的函数：

- pthread_create 创建线程
- pthread_exit 退出线程
- pthread_cancel 撤销线程
- pthread_join 等待线程结束

4.7.2 POSIX Semaphore

POSIX Semaphore 是由 IEEE Std 1003.1b 标准定义的一组函数接口：

- sem_init 初始化信号量
- sem_destroy 销毁信号量
- sem_wait 对信号量 P 操作（阻塞）
- sem_trywait 对信号量 P 操作（非阻塞）
- sem_post 对信号量 V 操作
- sem_getvalue 取得信号量的值

4.7.3 题目要求

题目要求如下:

线程

要求实现全用户地址空间共享，线程之间可以互相访问栈内数据。可以保留少量仅限于本线程可以访问的空间用以保存线程相关数据。(POSIX 标准有规定相关接口也可以实现)

信号量

POSIX 标准的信号量分为有名和无名信号量。无名信号量可以用于进程内同步与通信，有名信号量既可以用于进程内同步与通信，也可以用于进程间同步与通信。要求至少实现无名信号量（上述函数）的全部功能。

要求

学生要至少实现上述的两组函数，并实现要求的功能，请合理增加系统调用以及相应的数据结构。其他 POSIX 线程信号量相关接口会根据实现难度加分。

提交方式

以 lab4 分支为基础，新建 lab4-challenge 分支，完成之后提交到同名的远程分支。