Lab 4 实验报告

一、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？**

> SAVE_ALL函数，将当时现场所有的通用寄存器都保存到栈中，此时的栈是内核空间的栈

> 不可以，寄存器已经被破坏了，应从栈中读入，根据首地址（用户态栈指针`TF_REG29 (sp)`）+ 偏移量的方法获取寄存器中的参数.

> 用户调用msyscall函数时，第一个参数是系统调用函数的位置，在宏定义中是”系统调用的基准值+偏移“。然后从第二个参数开始是真正传递给系统调用函数的参数。sys开头的函数也采用了相同的参数排布。

> 1. 将当前寄存器的值储存在栈中
> 2. 改变寄存器的值，在这之后所有对之前寄存器的值的工作都需要在栈中进行存取
> 3. 调用完毕后回到函数，需要将sys_*函数的返回值存入`Trapframe`。

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？**

> 说明在fork()之后，子进程与父进程共享同一段代码

> 说明fork的时候子进程的pc设置的是父进程调用fork函数之后的位置，并且子进程从此处开始执行

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

> C

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。

> 根据mmu.h中的内存布局，UTOP是用户空间的极限，duppage肯定为比UTOP小的地址空间服务。

> 其次，我们还注意到，UTOP也叫UXSTACKTOP （`#define UXSTACKTOP (UTOP)`）它的下面一页是exception stack，每个进程的异常栈都是我们单独处理的，因这一页也不需要duppage。
>
> 综上，duppage保护的页应该是截至到(UTOP/BY2PG-1)这一页。

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？**

> `vpd`是指向页目录的指针，`vpt`是指向用户页表的指针
>
> 用法就是`(*vpd)[(VPN(va)/1024)]`即先计算得到`va`所属的页目录位置，再转换成指向用户页目录的指针，得到对应的页目录项，`(*vpt)[VPN(va)]`同理，得到对应的页表项

> 在fork函数中，通过`vpd`和`vpt`实现在用户地址空间中对内核相应地址的访问。由于页目录和二级页表的每一页都是连续的一片地址空间，因此可以使用数组的方式进行访问和遍历

```
    if(((*vpd)[VPN(i)/1024]) != 0 && ((*vpt)[VPN(i)]) != 0 )
```

> 当二者同时指向页目录时，这两者会重合，即为自映射机制

> 不能，会出现进程修改内核的情况

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？**

> 当异常栈指针没有指向栈顶时，也就是说已经出现过中断异常，并且使用过这个异常栈，还没有将保存的现场还给用户态空间

> 这样用户空间才能使用异常处理的结果

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？**

> 把缺页中断交给用户处理，可以使得一个进程缺页中断后被挂起，另一个进程执行。而内核不受影响，可以继续支持另一个进程的运行，提高整体的执行效率

> 通用寄存器的用途：暂存数据和地址，传送数据。在处理异常时，会先SAVE_ALL将所有通用寄存器保存到栈中，再进行异常的处理，完毕之后再将栈中保存的通用寄存器的值返回给寄存器

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?**

> 在开始生成一个新进程之前先给他安排好异常栈和异常入口等，如果在开始新进程的时候就发生了异常，能够立刻使用异常栈进行应对

> 这样不能处理在完成写时复制保护机制完成之前的异常

> 不用，这个作为函数指针使用。在entry.S中，有关键字`.globl`说明，这个变量名不在这个文件中定义，是一个全局变量，存储函数地址，不需要另外赋值

二、流程梳理

一、系统调用机制的实现

- 调用一个需要内核配合才能完成的函数，该函数会调用`syscall_xxx`函数(`user/syscall_lib.c`)
- `syscall_xxx`函数会调用我们写的汇编函数`msyscall(user/syscall_wrap.S)`，该函数使用特权指令`syscall`
- 此时CPU触发异常，陷入内核态，异常向量分发器检测到是系统调用（异常编号为8），进入`handle_sys`函数(`lib/syscall.S`)，进行处理
- `handle_sys`函数会进一步读取系统调用号，进行进一步分发，分发进C函数(`lib/syscall_all.c`)，在C语言中进行处理。
- 在内核态中处理完毕，返回用户态，并将返回值(位于$v0寄存器)传递回去，一层层回到调用处。

二、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，需要通过系统调用来实现。通讯最直观的一种理解就是交换数据。

两个进程之间之所以没法相互交换数据，是因为**各个进程的地址空间相互独立**。我们在之前写的函数，正是为了实现地址空间之间的沟通。而沟通两个进程，自然需要一个**权限凌驾两个进程之上的存在**来进行操作，即内核态。

`IPC`的操作，本质是在内核态中对这些部分进行赋值。我们需要填的两个函数位于`lib/syscall_all.c`中。

```
/* Overview:
 *      这个函数使得调用进程可以接收其他进程发送的信息。更准确地说，
 *      这个函数可以标记当前进程，使得其他进程可以向其发送信息。
 * Pre-Condition:
 *      dstva必须合法（NULL也是合法的）。
 * Post-Condition:
 *      这个系统调用函数会将当前进程状态置为NOT RUNNABLE，并释放CPU。
 */
void
sys_ipc_recv(int sysno, u_int dstva)

/* Overview:
 *      Try to send 'value' to the target env 'envid'.
 *      将value传给目标进程envid。
 *      如果目标进程尚未处于可接收状态，返回值应当为-E_IPC_NOT_RECV。
 *      其他情况下，发送成功后，目标进程的IPC部分数据应当按照如下规则更新：
 *      env_ipc_recving设置为0，防止多余的接收。
```

```
 *      env_ipc_from设置为发送进程的id。
 *      env_ipc_value设置为函数参数value。
 *      目标进程需要标记为RUNNABLE，以便重新运行。
 * Post-Condition:
 *      返回值0代表成功，小于0代表出错。
 *
 * Hint: 你唯一需要调用的函数只有envid2env()。
 */
int
sys_ipc_can_send(int sysno, u_int envid, u_int value, u_int srcva, u_int perm)
```

三、fork

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。

- 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。

- 先调用了`set_pgfault_hadler(pgfault)`来设置缺页入口为`static void pgfault(u_int va)`函数
- 然后调用了`syscall_env_alloc()`函数，进行了新进程的创建。 新的进程已经有了父进程的寄存器等运行信息，但是还缺少对内存的管理信息。
    - **父进程** 对分裂的子进程做如下操作：
        - 遍历 **0 ~ UTOP** 的空间，把有效的页信息利用 **duppage** 来复制给儿子
        - 利用**syscall_mem_alloc()** 在**USTACKTOP** 下面分配一页空间作为子进程的**异常处理栈**
        - 利用**syscall_set_pgfault_handler()** 函数把子进程的缺页中断函数设置为 **__asm_pgfault_hander()** 并且告知其异常处理栈为**UXSTACKTOP**
        - 处理完成之后，利用**syscall_set_env_status()** 把子进程的状态设置为**可运行**
        - 返回**子进程的envid**
    - **子进程**开始运行之后做了如下操作：
        - 把 ***env** 指向自身进程控制块
        - 返回 **0**

三、体会与感想

lab 4的难度很大，刚开始写的时候对于整个要完成的内容几乎没有认知，几乎写到最后甚至开始调试的时候，才一步步弄明白这些要填写的函数是要干什么的。指导书的作用感觉仅仅只是一个引导，但是理论课的教学又和实验课有一定的距离，相关的资料少之又少，感觉操作系统实验课很难，这种难和计组实验课的难是不一样的，是一种茫然无知，想要弄懂却又无从下手感觉，真的很头疼。

四、遗留难点

关于fork还有进程通信还是懵懵懂懂的，甚至不知道代码的正确与否，怀疑前几个lab还残存未知的bug。不会调试，肉眼debug，只能提出问题等待有同学遇到相同问题求一个解决办法。