操作系统lab4实验报告

姓名：高远 学号：19374242 班级：202113

实验思考题

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?*

  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

- *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，前四个参数放在对应的寄存器上，后两个参数存在该空间之上的位置。

- *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?*

  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。

Thinking 4.2

`mkenvid()` 函数的作用是给进程分配相应的进程号，其值必定是大于0的，因此在*系统调用和 IPC 部分的实现 与* `envid2env()` 中可以通过判断其值是否小于0来判断是否成功找到相应的进程。

Thinking 4.3

- *子进程完全按照 fork() 之后父进程的代码执行，说明了什么?*

  说明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。

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

答案：选择C

Thinking 4.5

从0到USERSTACKTOP的地址空间里，对于非不是共享的和只读的页面进行映射。

Thinking 4.6

- *vpt和vpd的作用是什么？怎样使用它们?*

  vpt存放着二级页表，vpd存放着一级页表，使用时把它们当作数组头。
  对于第i页，(*vpd)[i >> 10]为i对应的一级页表，(*vpt)[i]为i对应的二级页表。

- *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?*

  由源码可知，vpt指向UVTP区域，也就是当前进程页表项所在的位置，变量类型为Pte*数组。vpd指向(UVPT+4(UVPT>>12))*区域，是自映射机制下的一级页表地址，变量类型是Pde*数组。

- *它们是如何体现自映射设计的?*

  (UVPT+(UVPT>>12)*4)是vpd指向的地址体现了自映射。

- *进程能够通过这种方式来修改自己的页表项吗?*

  可以

Thinking 4.7

1.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”。

2.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，因此存到用户空寂。

Thinking 4.8

1.体现了微内核的思想，让用户进程实现内核的功能，就算内核出了问题操作系统也能运行

2.通过把所有通用寄存器压入栈中，在使用时取出

Thinking 4.9

1.syscall_env_alloc过程中也可能需要进行异常处理。

2.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。

3.不需要，子进程复制了父进程中__pgfault_handler变量值

实验难点图示

难点1：三类函数

本次实验的函数分为三类：

syscall……：*用户空间内的函数，与sys……*成对存在

msyscall：设置**系统调用号**并让系统陷入内核态的函数

sys_……：内核函数

MOS进行系统调用的流程：

1. 调用一个封装好的用户空间的库函数
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

难点2：sys_yield

```
65  void sys_yield(void)
66  {
67
68      bcopy((void *)KERNEL_SP - sizeof(struct Trapframe),
69            (void *)TIMESTACK - sizeof(struct Trapframe),
70            sizeof(struct Trapframe));
71      sched_yield();
72  }
```

lib/syscall_all.c中的void sys_yield(void)函数作用是实现用户进程对CPU的放弃， 从而调度其他的进程。在本次实验中，它需要先保存相关信息在在KERNEL_SP，然后调用sched_yield()函数。

难点3：IPC

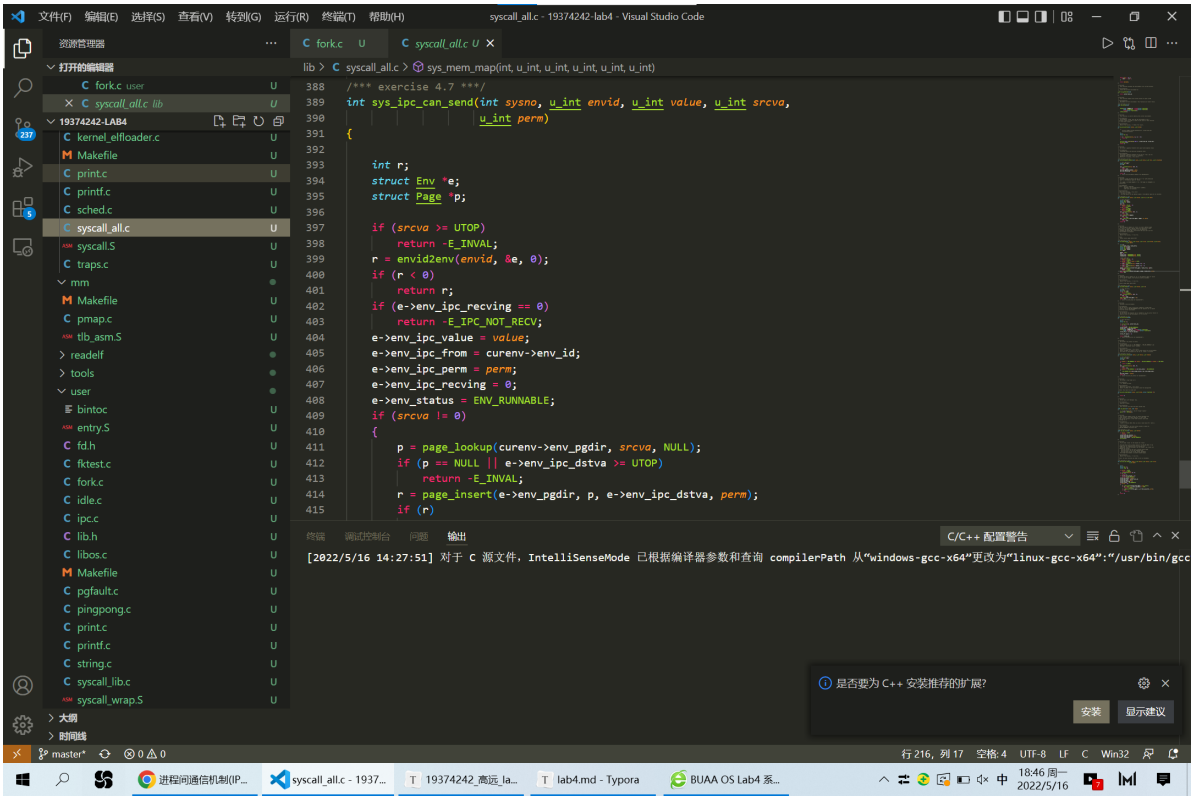
所有的进程都共享了内核所在的2G空间。对于任意的进程，这2G都是一样的。 因此，想要在不同空间之间交换数据，我们就可以借助于内核的空间来实现。也就是说，发送方进程可以将数据以系统调用的形式存放在内核空间中，接收方进程同样以系统调用的方式在内核找到对应的数据，读取并返回。我们使用进程控制块来实现相关操作，如图：

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

我们通过*lib/syscall_all.c*

中sys_ipc_recv(int sysno,u_int dstva)函数用于接受消息。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。如图：

```
361 void sys_ipc_recv(int sysno, u_int dstva)
362 {
363     if (dstva >= UTOP)
364         return;
365     curenv->env_ipc_recving = 1;
366     curenv->env_ipc_dstva = dstva;
367     curenv->env_status = ENV_NOT_RUNNABLE;
368     sys_yield();
369 }
```



难点四：duppage

我认为这个函数是本次实验最难的函数之一，花费了我大量的时间，需要注意有以下几点：

1.PTE_R是可写而不是可读。。。。

2.进程id是0代表当前进程，在envi2env内有写

3.如果可写且不共享，标记 PTE_COW （写时复制）

4.在syscall_mem_map(0, addr, *envid*, addr, perm);中只修改了子进程的perm。所以需要syscall_mem_map(0, addr, 0, addr, perm);修改父进程perm。

本函数的目的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。如图：

```
121  static void
122  duppage(u_int envid, u_int pn)
123  {
124
125      //  user_panic("duppage not implemented");
126      u_int addr = pn << PGSHIFT;
127      u_int perm = ((Pte *)(*vpt))[pn] & 0xfff;
128
129      int flag = 0;
130      if ((perm & PTE_R) && !(perm & PTE_LIBRARY))
131      {
132          perm |= PTE_COW;
133          flag = 1;
134      }
135      syscall_mem_map(0, addr, envid, addr, perm);
136      if (flag)
137          syscall_mem_map(0, addr, 0, addr, perm);
138  }
```

体会与感想

本次实验花费了我大量的时间，对于在用户态和内核态之间的切换，还有各种寄存器、堆栈、数据的设置，汇编代码的编写都给我带来了很大的困难，也让我尝到了基础不扎实带来的后果。在很长一段时间内，甚至连编译都无法通过。我花了很长时间才把第一部分写完，ipc较为简单，很快就完成了，但是fork又花费了我大量的时间。尤其是vpt和vpd的理解和duppage的使用，让我百思不得其解，通过询问助教和查阅资料才勉强做出来。

通过本次实验的学习，使我更加认识到在填写一些复杂度很高的函数，不能拿到代码，看到hint就往上一股脑的填，而应该去尝试复盘整个过程，并考虑到各种情况，对每个情况进行处理。本次实验使我对系统调用，ipc以及fork函数有了更深一步的理解，可以说，lab3实验使我受益匪浅。

指导书反馈

本次指导书在ipc方面写的非常详细完美，但是在fork函数中页写入异常的相关函数和一些汇编代码的讲解中有些含糊不清，希望以后能够写的更加清楚。

残留难点

对__asm_pgfault_handler函数中的汇编代码并没有非常充分的理解。