南京航空航天大学

操作系统实践

第一次实验

班级： 1618001
学号： 161840230
姓名： 王可

一、 题目简述

1. 带参数的系统调用(30%)

修改系统调用使得shutdown可以接受一个整形参数，当用户输入shutdown命令后，屏幕上打印一行字并退出，示例如下：

```
$ shutdown 20
Leaving with code 20.
```

2. 理解进程切换(20%)

阅读源码并理解，回答一系列问题：

- CPU在执行`scheduler()`时运行在用户态还是内核态？运行在哪个栈上面？
- 当你在命令行敲下`shutdown`时，系统会创建一个进程执行shutdown.c中的代码，当CPU执行以下三条指令时`movl $SYS_shutdown, %eax; int $T_SYSCALL; ret`时，CPU运行在用户态还是内核态？运行在哪个栈上面？（注意：CPU在执行这三条指令时的状态和栈是一样的）
- 在执行命令`shutdown`的过程中，当cpu执行到涉及特权指令的函数`outw()`时，CPU运行在用户态还是内核态？运行在哪个栈上面？
- 为何在执行完`swtch`函数后，cpu没有像普通函数调用一样返回到`scheduler`函数中？

3. 子进程优先的fork(50%)

阅读proc.c中的int fork(void)函数。xv6执行完fork后，父进程先运行，然后子进程运行。

修改这个行为，使得子进程优先运行。实现一个系统调用，以控制是否启用算法。

```
void fork_winner(int winner);
```

回答问题：

- 在父进程优先的情况下，偶尔会有子进程先于父进程打印到屏幕的情况出现；在子进程优先的情况下，偶尔也会有父进程先于子进程打印到屏幕的情况。解释可能的原因。

二、 实验过程

1. 带参数的系统调用

1. 阅读源码得知，shutdown用户函数调用了名为shutdown的系统调用(该函数在usys.S文件中被定义)，在中断trap中，通过syscall调用到sys_shutdown函数，在此函数中，需要实现获取一个参数，并将其输出到屏幕的功能。
2. 仿照sleep函数，使用了argint函数获取int型参数，argint有两个参数
    - 其一为**n**，从外部意义上控制了获取函数第n个变量，而从函数内部是控制了栈地址，取得地址空间为(myproc()->tf->esp) + 4 + 4*n，即栈指针所指位置加4(n+1)个字节，系数是4是因为一个整形数所占存储空间为4个字节，而之所以使用n+1即第一个参数需要向上跨过4个字节是由函数调用时栈的组织结构决定的，根据xv6官方指南Figure2-3中的结构所示，函数栈中，底

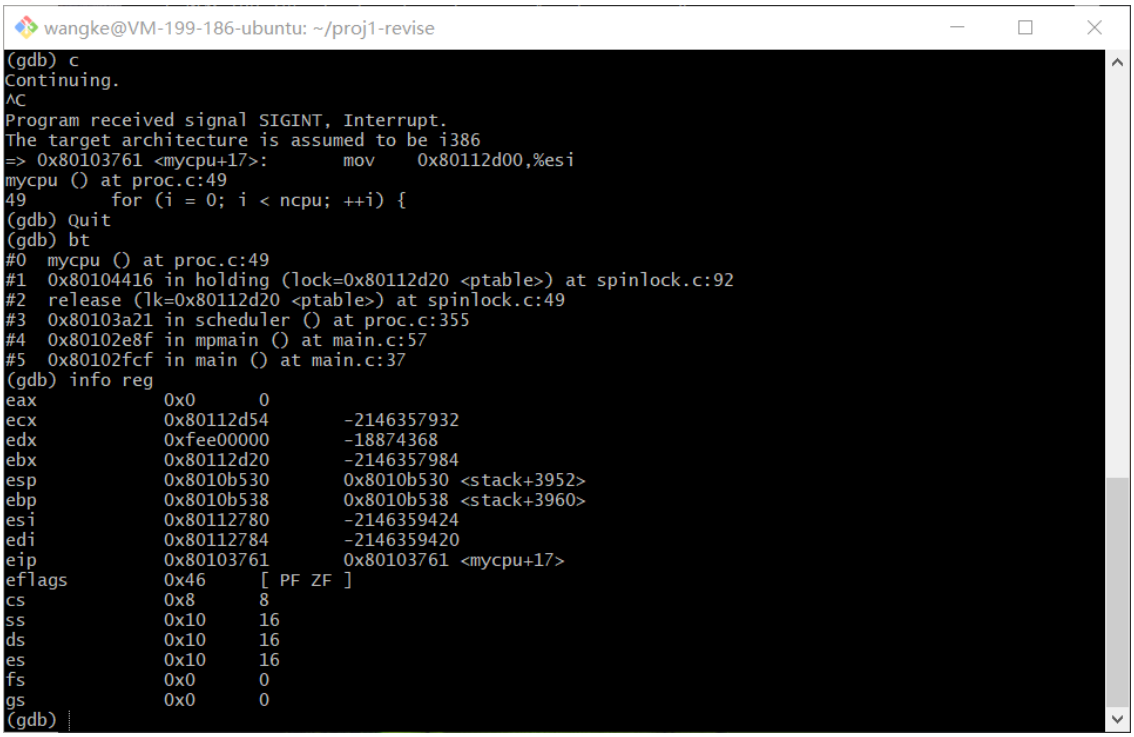
部为函数返回地址，上面为被调用者参数自左向右依次入栈，即第一个参数在最底部，也就是esp+4的位置，因此，所寻地址为esp+4+4*n.
    - argint 第二个参数即为所获取的数据存放的地址，通过调用fetchint判断地址是否越界后获得数据。
    - 通过这种方法，即完成了目标需求。
3. 此外，考虑到不使用Leaving code的情况，本使用判断输入参数是否为0的方法决定直接退出系统还是先输出信息再退出系统，但由于给出的参数范围为止，即存在Leaving code本身为0的情况，因此在用户函数获取命令行参数时，根据参数数量生成另一系统调用参数ctrl用以控制。实现代码如下：

```
// sysproc.c
int
sys_shutdown(void){
  int arg;
  int ctrl;
  // for(int i = 0 ; i < 40;i++){
  //   argint(i,&arg);
  //   cprintf("%d  :  %d\n",i,arg);
  // }
  if(argint(0,&ctrl) < 0){
    return -1;
  }
  if(argint(1,&arg) < 0){
    return -1;
  }  // the function argument is defined in file syscall.c
  if(ctrl == 1){
    cprintf("Leaving with code %d\n",arg);
  }
  outw(0x604, 0x2000);
  return 0;
}

// shutdown.c --main
int
main(int argc, char *argv[])
{
  int arg = 0;
  int ctrl = 0;
  char *number = "";
  if(argc > 1){
    number = argv[1];
    ctrl = 1;         // the variable ctrl is initialized as 0
  }
  while(strlen(number)>0){  // transform the char* argument into int
    arg = arg * 10;
    arg+=number[0]-'0';
    number += 1;
  }
  shutdown(ctrl,arg);
  exit();
}
```

2. 理解进程切换

阅读源码并理解，回答一系列问题：

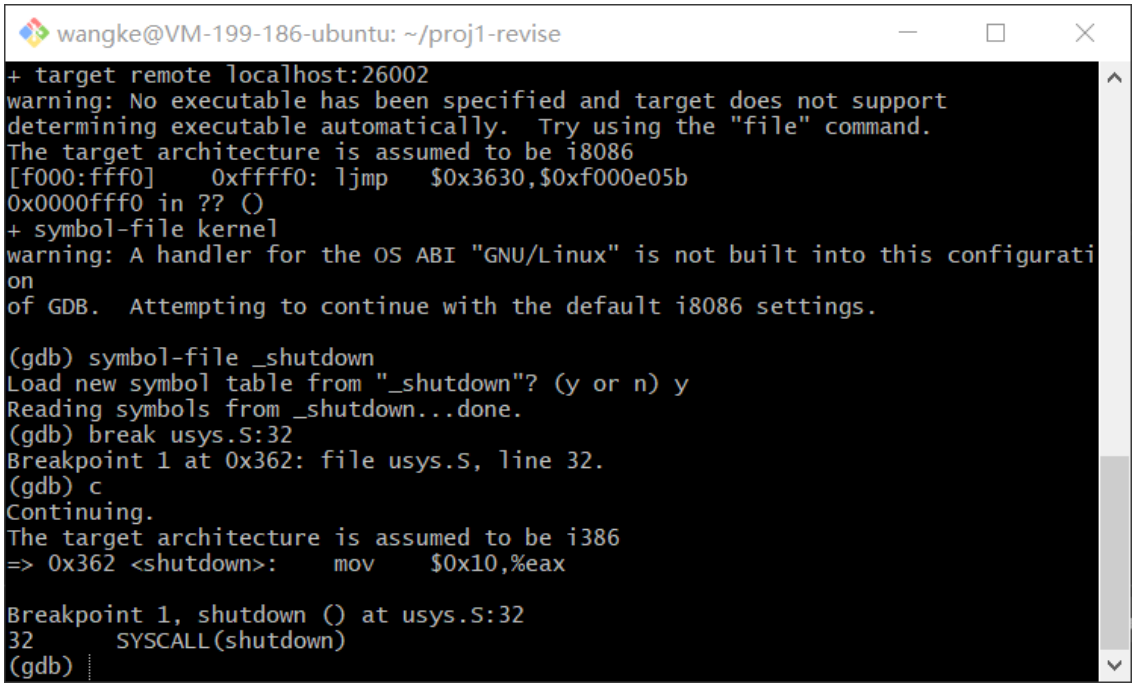
- CPU在执行`scheduler()`时运行在用户态还是内核态？运行在哪个栈上面？
  - 运行时在gdb中使用ctrl+C打断程序执行，进入scheduler()函数，通过bt命令确认后，确实进入了该函数：

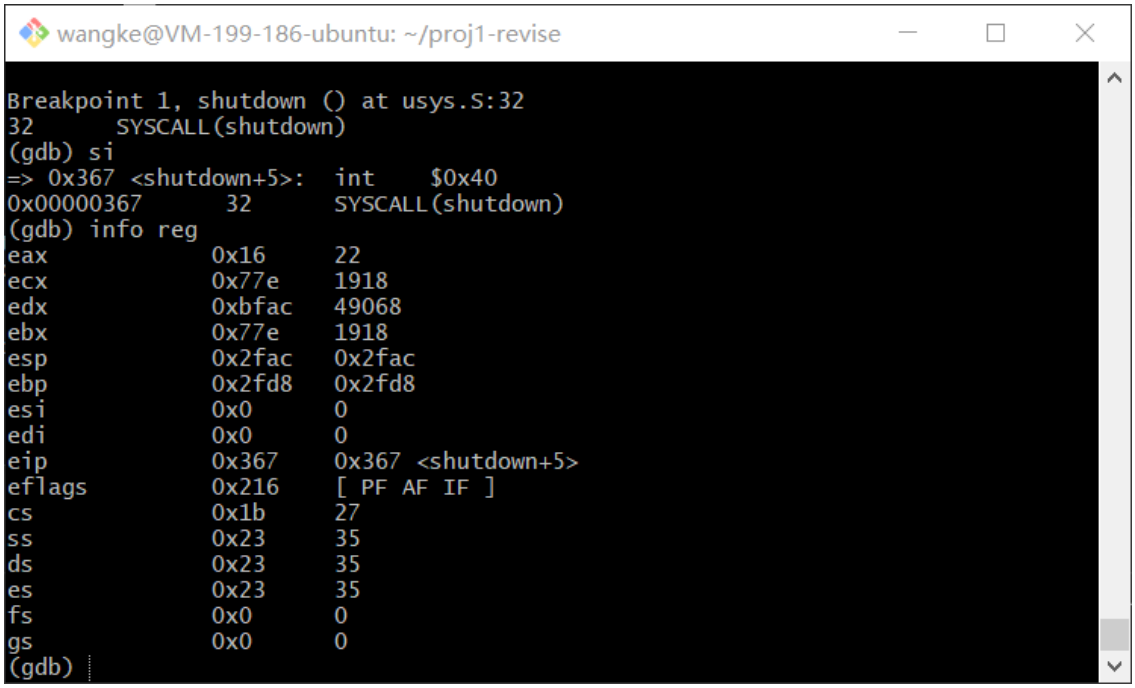


    - 查看寄存器可以发现：cs值为8，即1000，最后两位是00，即处于内核态中，同时esp寄存器中的值为0x8010b530，大于0x8000 0000。由此可见当前栈地址在内核空间，函数scheduler()运行在内核栈中。
- 当你在命令行敲下`shutdown`时，系统会创建一个进程执行shutdown.c中的代码，当CPU执行以下三条指令时`movl $SYS_shutdown, %eax; int $T_SYSCALL; ret`时，CPU运行在用户态还是内核态？运行在哪个栈上面？（注意：CPU在执行这三条指令时的状态和栈是一样的）
  - 使用如下命令加载_shutdown并将断点设置于如题三条指令前：

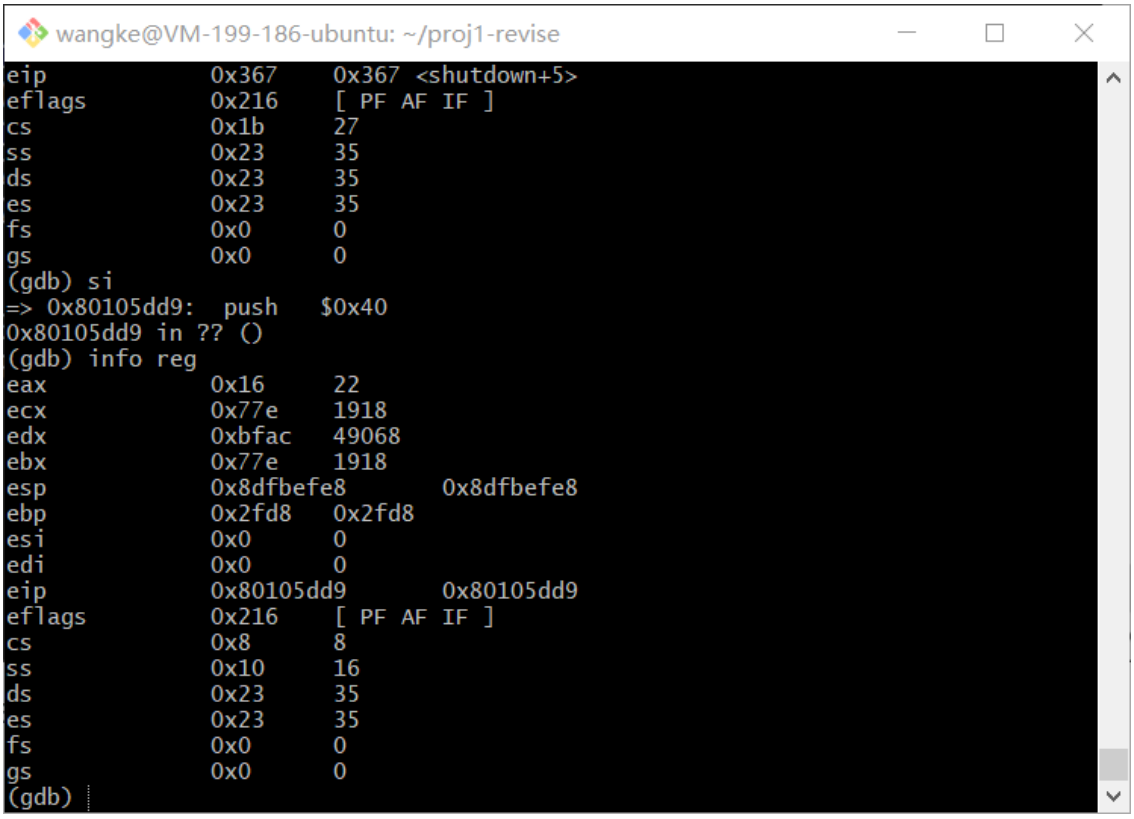
```
symbol-file _shutdown
break usys.S:32  # 需要将断点打到函数中而不是define中
```

  - 多次使用continue进入执行后，输入shutdown命令，进入断点并使用info reg查看寄存器，结果如下：



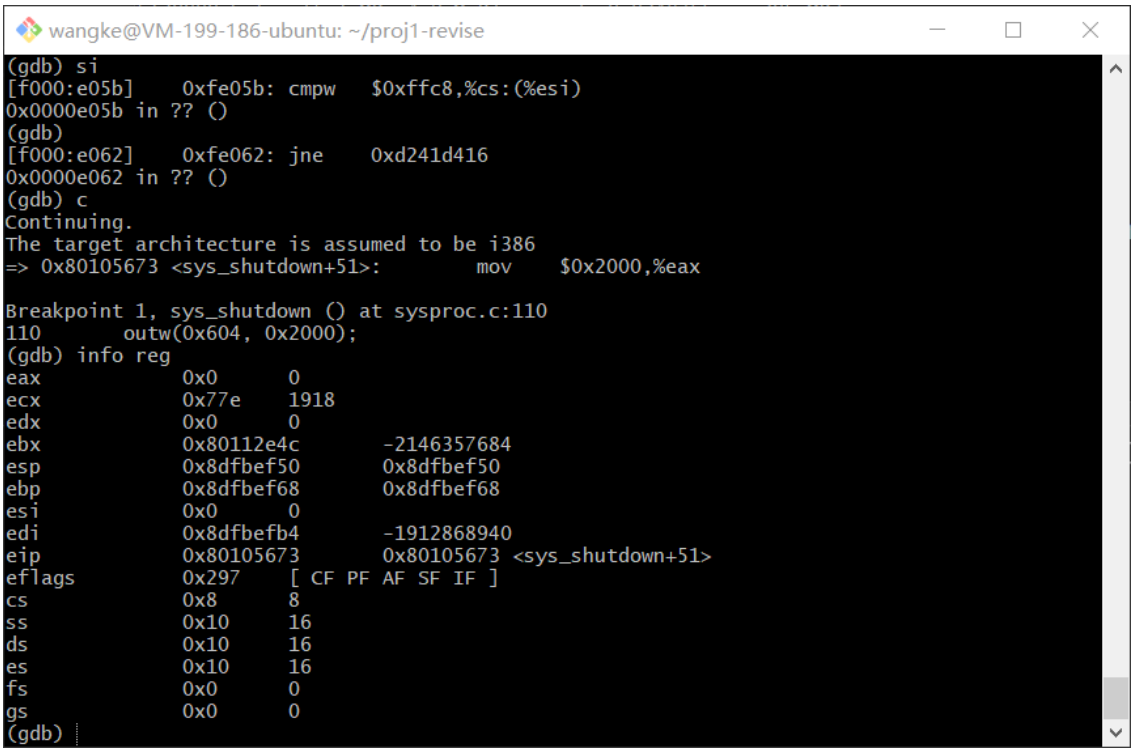


- %eax寄存器值为22，正是syscall.h中设置的系统调用号，断点正确。



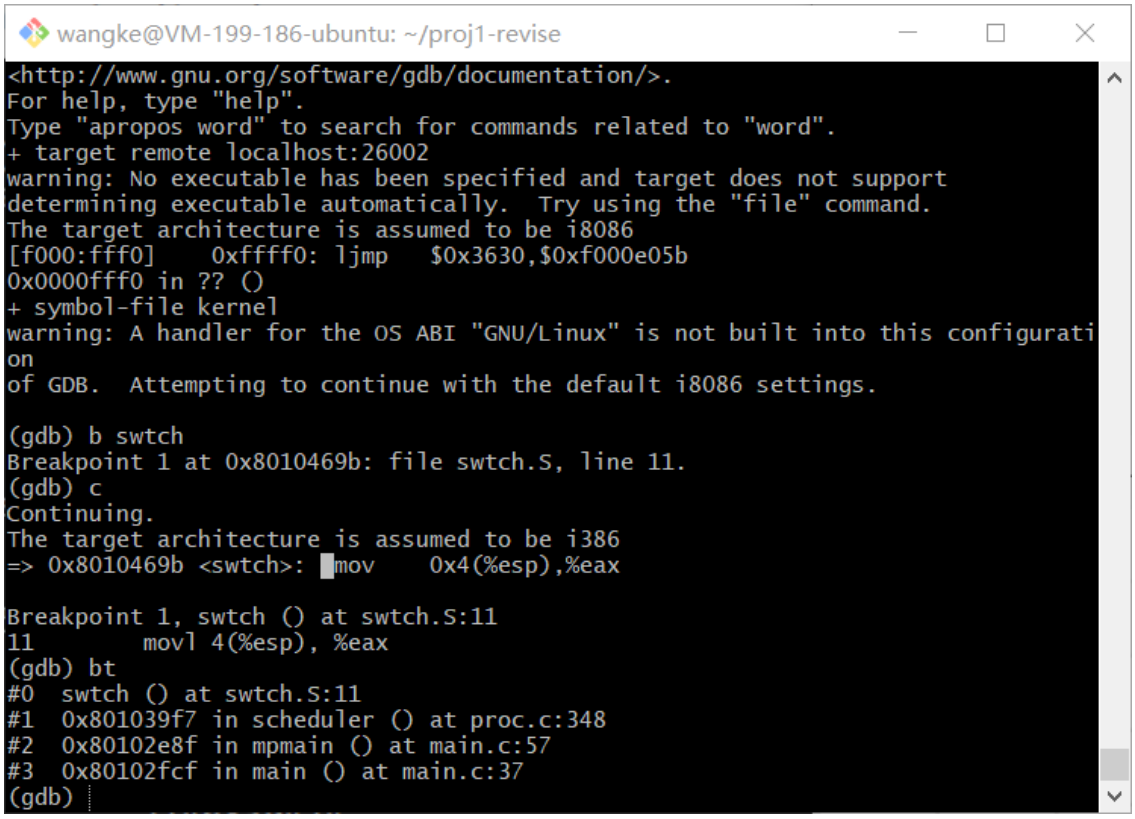
- 进入中断后cs寄存器值变为8，最后两位为00，进入了内核态，而进入终端之前，值为27(00011011) 在用户态中。进入中断后esp为0x8dfb efe8 大于0x8000 0000 ，也证明栈在内核空间中。因此这段代码本身运行在用户态用户栈中，而进入中断后，进入内核态内核栈。

- 在执行命令`shutdown`的过程中，当cpu执行到涉及特权指令的函数`outw()`时，CPU运行在用户态还是内核态？运行在哪个栈上面？
  - 使用命令break x86.h:29 设置断点进入outw函数如下：

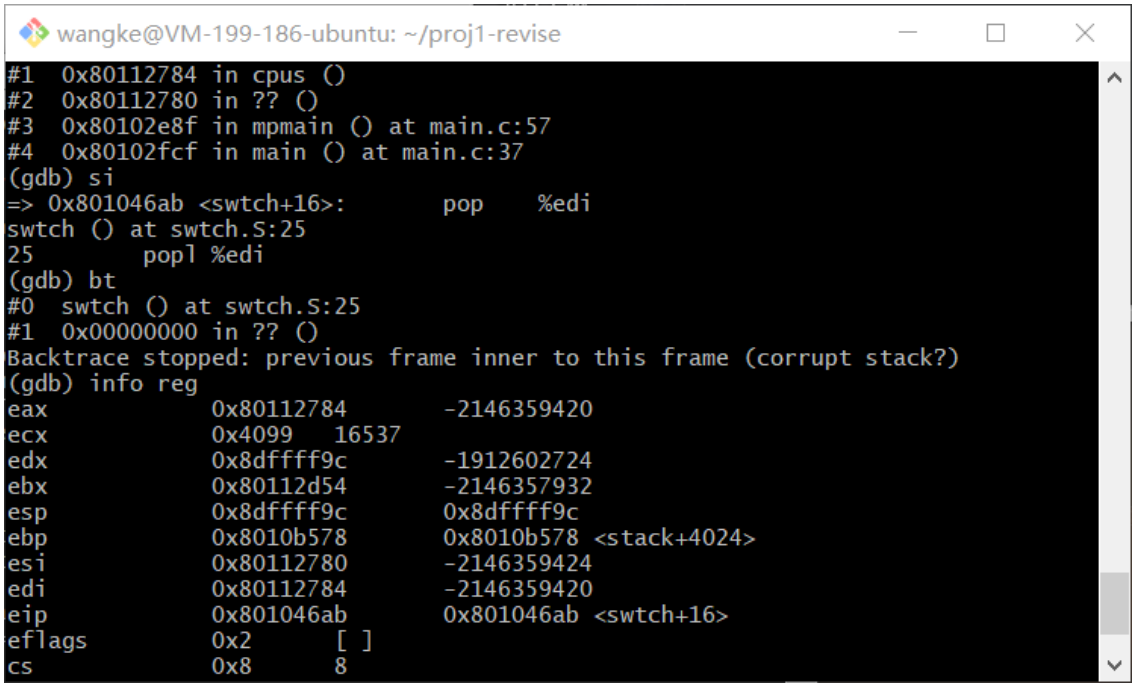


- 查看寄存器状况得到cs为8，运行在内核态，同时esp为0x8dfg ef50，大于0x8000 0000 因此栈处于内核空间中。

- 为何在执行完`swtch`函数后，cpu没有像普通函数调用一样返回到`scheduler`函数中？
  - 使用“ b swtch”设置断点，触发断点后查看如下：



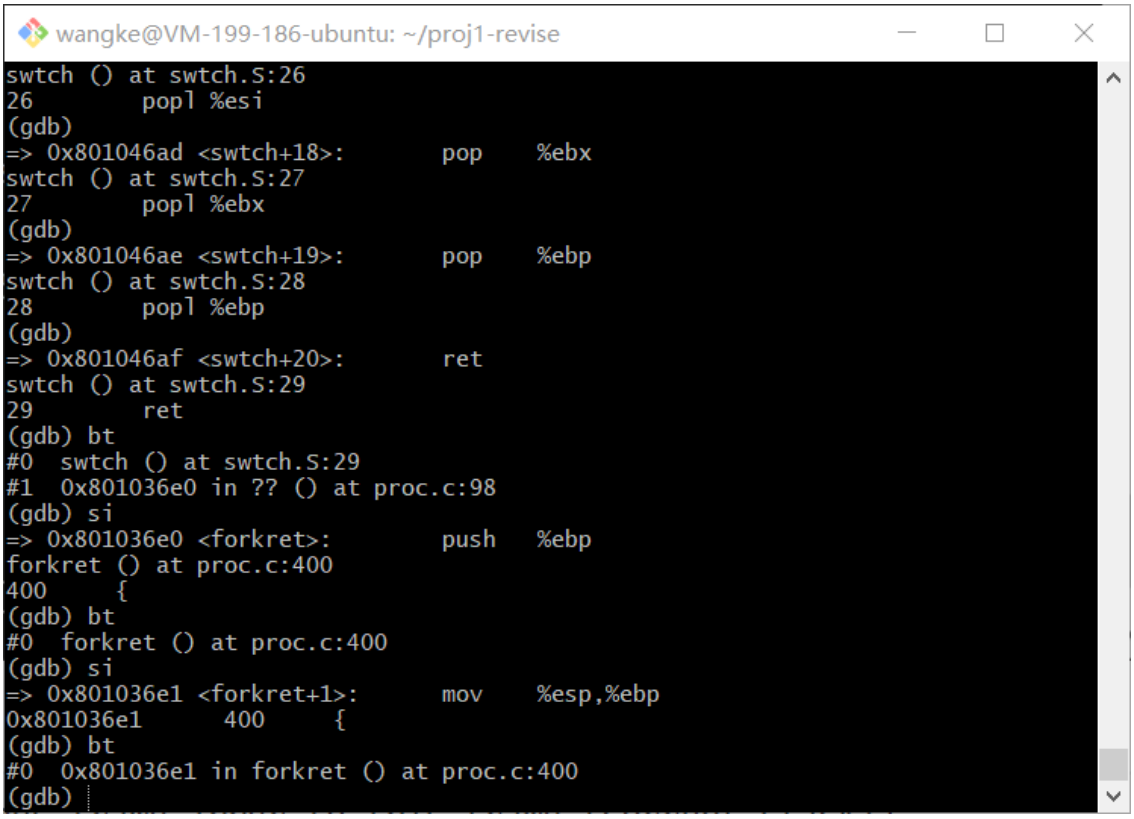
- 不断使用si单步执行并查看寄存器内容，与调用层次，均为main()->mpmain()---，esp变化幅度不大，直到一条pop指令后：



- esp明显变化，且调用层次提示

```
Backtrace stopped: previous frame inner to this frame (corrupt stack?)
```

- 继续单步执行，至ret后，返回到forkret()函数中，如下所示:



- 在程序中，创建进程空间时将其上下文中eip设置为forkret()函数的起始地址，因此进程运行返回时，会继续执行forkret()代码，而其上还有一个指向trapret的指针，因此forkret()执行完毕返回时，将继续执行trapret()函数，而该函数恢复用户寄存器并跳转到process代码。由于这样的机制，完成swtch函数后，cpu不会返回到scheduler()函数中。

3. 子进程优先的fork

阅读源码可知，父进程创建完子进程空间并复制内容后，并未进行额外操作，只是将子进程状态修改为RUNNABLE，使其进入就绪状态，等待时间片轮转。因此大多时间父进程需要执行完毕后才会将时间片释放交给子进程执行（父进程执行所需时长小于时间片长度）。若想要子进程优先执行，有两种方案：

- 一是让父进程调用sleep进入休眠，待到子进程执行完毕后，使用wakeup函数将父进程唤醒继续执行。然而因为test程序给定无法修改，因此只能使用第二种方法。
- 二是设置一个信号值，使父进程创建完子进程后交出时间片，这样就可以使得子进程相对父进程而言优先执行。完成子进程优先级更高的需求。

实现sys_fork_winner(void)系统调用的过程中需要传入一个整型参数winner，采用第一题中的方法用argint函数读取栈内存，获得forktest函数中给出的参数，并将其存入一个全局变量地址空间中(该全局变量winner在proc.h中被定义，由于该头文件被proc.c和sysproc.c同时引用，所以可以直接定义)。同时，在fork函数创建完子进程后，若winner为1，则调用yield()函数交还时间片，由此完成需求。代码如下:

```
// proc.c
// int fork(void)
if(winner)
    yield();

//sysproc.c
int
sys_fork_winner(void){

  if(argint(0,&winner)<0)
    return -1;
  return 0;
}
```

回答问题:

- 父进程优先时，子进程先于父进程打印的原因：
  - 若父进程fork后恰好时钟发出中断，使得父进程丢失时间片，则子进程将优先完成输出。
- 子进程优先时，父进程先于子进程打印的原因：

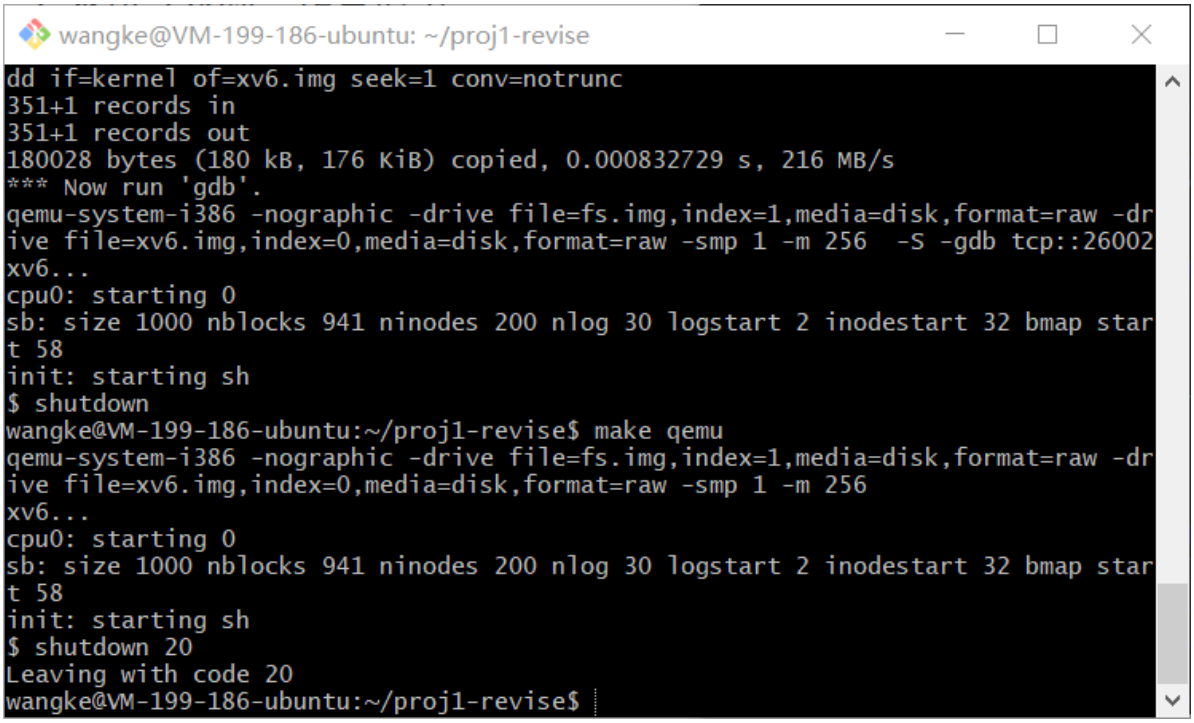
```
  acquire(&ptable.lock);

  np->state = RUNNABLE;

  release(&ptable.lock);
  if(winner)
    yield();
```

  - 对于如上代码，若编译器进行优化，调换第7和第3行语句执行顺序，或者运行时出错导致第七行优先执行，则父进程退出时子进程在这一轮时间片分配中未处于就绪状态，不会执行，等到父进程执行完毕后，才进入就绪，开始执行。

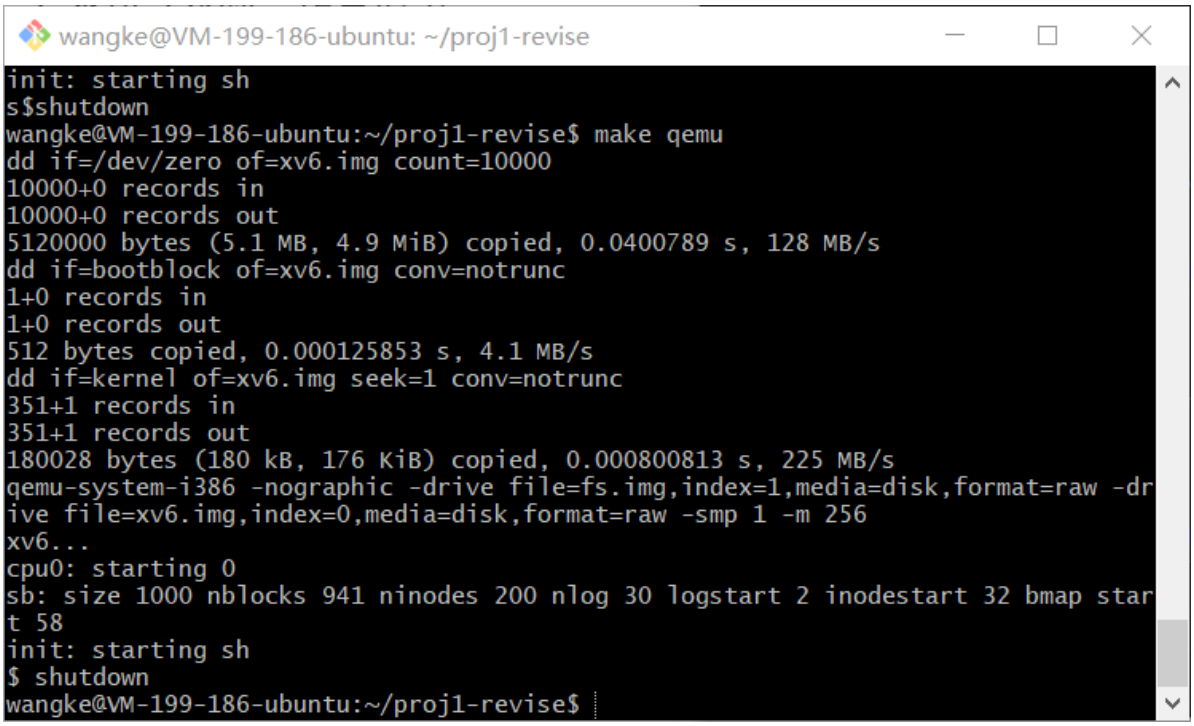
三、 实验结果

1. 带参数的系统调用

在xv6中输入shutdown 20，按要求输出了指定内容：

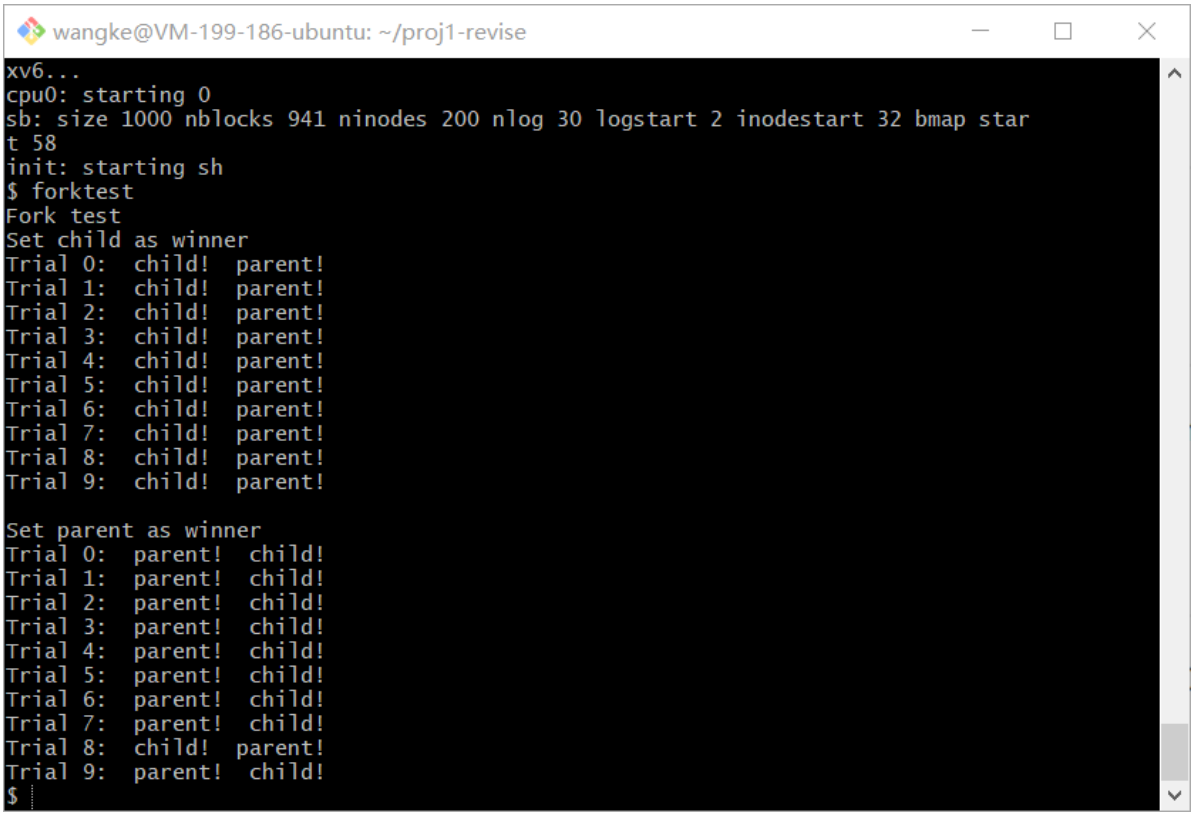


若不包含参数，正常退出：



3. 子进程优先的fork

在xv6中输入forktest，输出以下信息：



- 实验心得

本次实验更多的阅读了xv6系统的源码，对其运行机制，特别是进程调度的方法有了进一步的认识。同时学习了gdb调试方法，增加了新的知识。