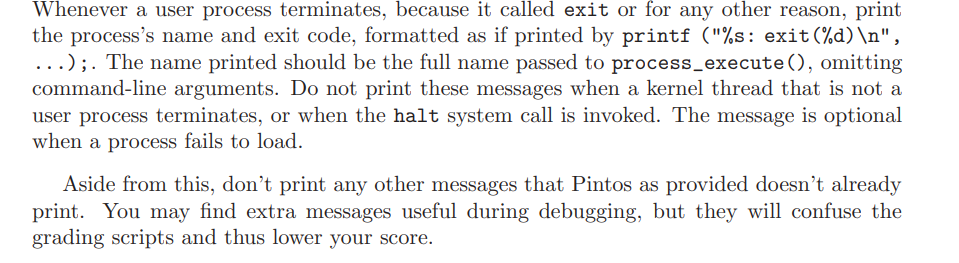
# project2

## Task1进程终止信息



### 1.1大致要求

在pintos文档中，这个任务要求当用户进程终止的时候，打印一条任务终止信息（包括进程名字和退出码）。格式是printf ("%s: exit(%d)\n",...);并且说明了传递给process\_execute()函数的参数就是用户执行进程的命令。同时文档中还要求我们需要分离命令中的进程名字和命令行参数。

### 1.2思路

唯一要做的，就是在进程结束的时候输出一行包括进程名和退出码的信息。在process.c中可以找到这个函数process\_exit()，这个函数的作用是释放用户过程的资源。其中包含两个步骤，第一个是释放页目录中指向的所用页表（二级），然后释放页目录本身。但这个函数显然没有包括用户过程关闭的所有操作，实际上它是在thread\_exit()中调用的，即thread\_exit()中先调用process\_exit()释放用户过程的资源，然后再执行关闭线程的操作。所以我们可以在process\_exit()的末尾或thread\_exit()加入一行代码来输出这个信息。

但是，我们还需要知道现在这个进程退出码。这是主要难点，这需要理解整个系统调用的过程。对源代码进行了研究之后，我的理解如下：

pintos的所有系统调用都是在30号中断中完成的。每当pintos需要执行一个系统调用，都会执行一个pintos系统的30号中断。

具体过程。当一个用户程序执行一个系统调用，实际上会执行lib/usr/syscall.h中的函数。 比如exit(int status)就是pintos系统中需要返回时候执行的系统调用函数。exit(int status)函数做到的事情就是：通过80x86汇编指令，把status（返回值）和一个系统调用编号先后压入当前用户线程的调用栈顶，并执行30号中断。

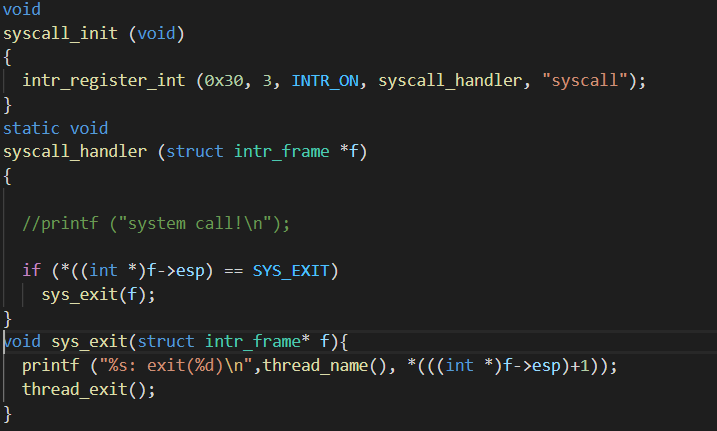
而30号中断是啥呢。在userprog/syscall.c中的函数syscall\_init()中完成了中断的注册，在中断表中加入的中断函数地址是syscall\_handler()的地址，参数就是当前栈指针（这个在中断的头文件中实现）。所以syscall\_handler()需要做到的事情就是取出esp指向的系统调用编号和返回值，然后执行相关的系统调用。syscall\_handler目前几乎是空的，而这些的系统调用是要在task3中实现的。

因此事情就非常明白了，当pintos调用exit(0)返回时，会入栈这个系统调用的编号(SYS\_EXIT,它的值是1，在lib/syscall-nr.h中定义)以及返回值0,然后触发操作系统的30号中断。30号中断函数是syscall\_handler()，它需要做的事，是从参数指向的栈中取出编号和返回值，并且根据编号执行相应的系统调用，完成在exit()中本应完成的事情。

### 1.3实现

返回到Task1，现在要做的事情就非常清楚了。一个用户过程结束，调用了exit()，把返回状态和系统调用码1（也就是sys\_exit()的编号）压入了当前的栈中。然后触发30号中断执行syscall\_handler(\*frame f)。在syscall\_handler(\*frame f)中根据编号1执行sys\_exit()系统调用（还没实现），然后在sys\_exit()中调用thead\_exit(),在thread\_exit()中调用了process\_exit()。我们只需要在这一个过程中找到一个节点(既包括栈帧f的信息又包括线程名字的信息)输出进程名字+返回值就行了。拥有栈帧f的地方有lib/syscall.c中的exit(f)、syscall\_handler(f)和sys\_exit(f)(没实现)。

原始的pintos代码syscall\_handlder默认直接调用了thread\_exit()，将其修改为如果当前栈顶的系统调用编号是SYS\_EXIT，则调用sys\_exit()函数，在sys\_exit()函数中调用thread\_exit()，并加入输出语句，注意esp类型为void \*，它的+4运算相当于int\*类型的+1运算。



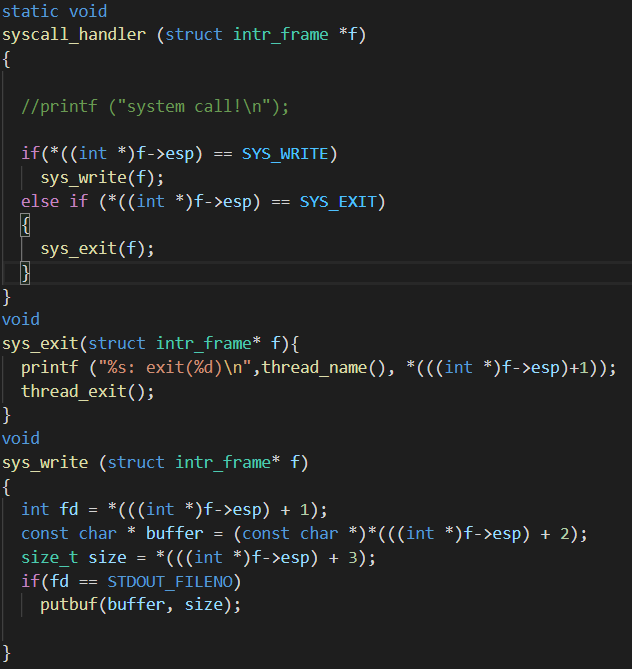
第一个修改

目前还没有分割参数，理论上说能通过第一个测试点(args-none)，但是，并没有。



研究一下测试文件发现，代码的过程评测信息是调用lib/user/syscall.c中的write()输出的，write()和exit()一样会触发30号中断。这也就意味着在syscall\_handler()中，会接受到type为SYS\_WRITE(编号9)的系统调用编号，在实现sys\_write()系统调用之前，是不可能通过任何一个测试点的。

而write()的第一个参数是一个fd，在stdio.h中定义。为1时表示STDOUT。write()会先后入栈size, buffer, fd和SYS\_WRITE，而在syscall\_handler接受到编号为9时，从栈顶往下分别是SYS\_WRITE, fd，buffer和size。往控制台输出的函数在console.c中定义，putbuf (const char \*buffer, size\_t n)。因此，只需要



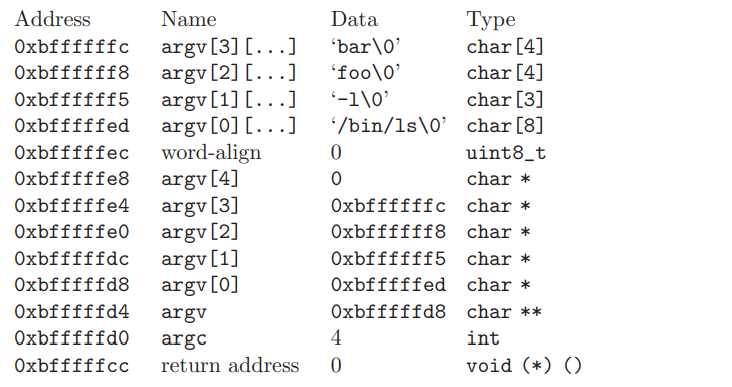
就完成了sys\_write()关于fd==1情况下的代码。

但是。。。这还是不能通过第一个测试点。在pintos文档的3.5.1中详细解释了参数堆栈的原理，显然就算是没有参数，文件名字也是需要入栈的，而这一步可以和TASK2一起完成。TASK1部分已经实现。

## Task2参数分离和入栈

### 2.1大致要求

我们需要在分割命令的文件名字和参数，并且把他们准确的放入栈中。整个过程在3.5.1中讲述的非常清楚。



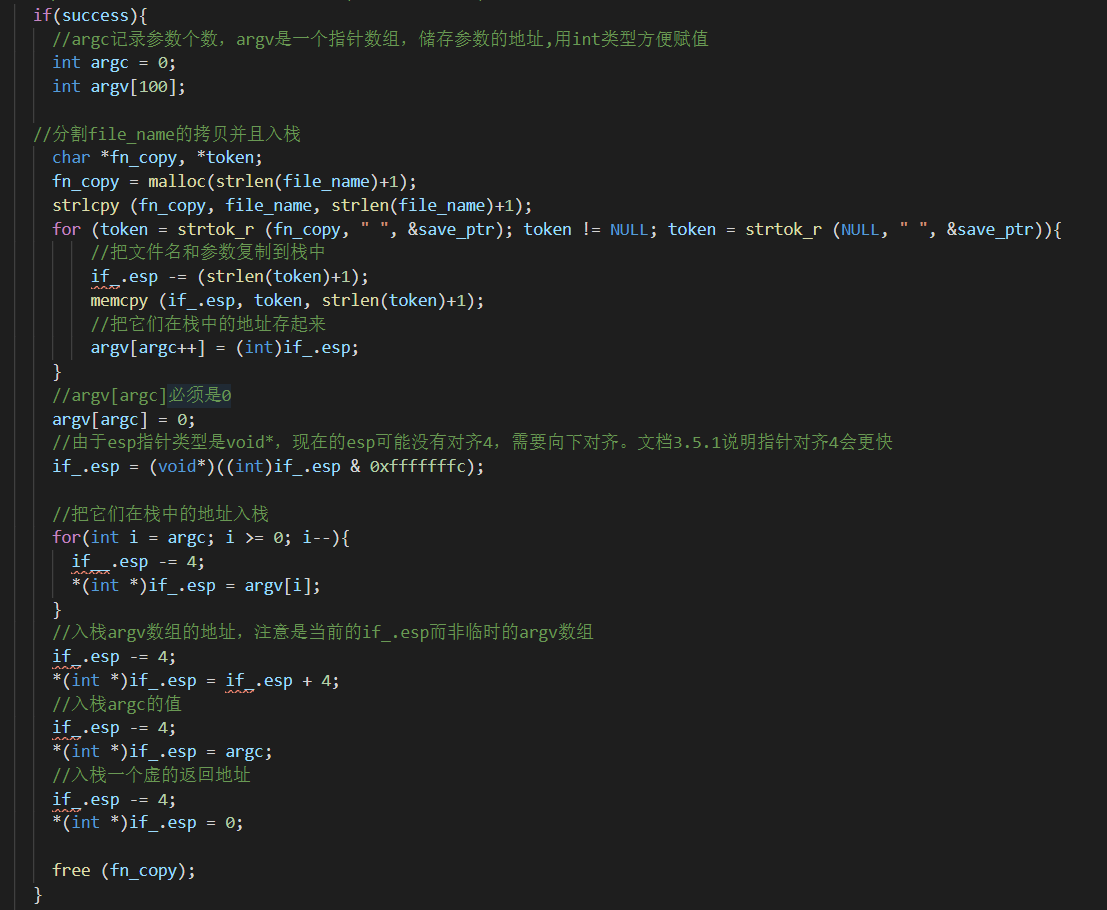
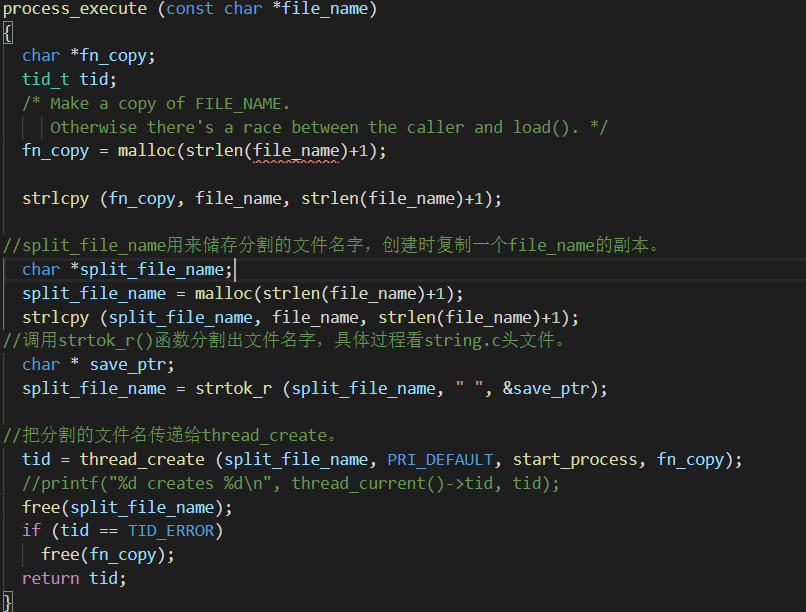
先入栈的是文件名字argv[0][..]和所有参数（顺序无关紧要），然后入栈一个0，再按顺序入栈参数和文件名的地址。最后入栈argv、argc和返回地址。pintos文档中提示了关于命令分割的字符串处理函数strtok \_ r，源代码也给出了很详细的例子。直接照抄就行。

### 2.2思路

Pintos文档中的要求写到了process\_execute(\*file name)函数的参数是接受到的是包含命令和参数的字符串。我们需要对它进行处理。

process\_execute()负责创建一个运行着file\_name文件的线程。用户命令行的输入就是\*file\_name。分割完文件名之后需要入栈，但这个时候栈的空间还未被分配，process\_execute()的执行会进入start\_process()进行内存分配，应当在那里实现参数的入栈。

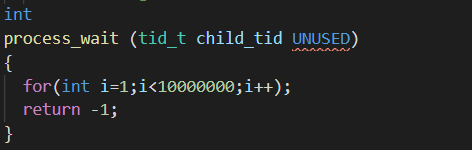
所有的修改涉及process\_execute()和start\_process()两个函数。



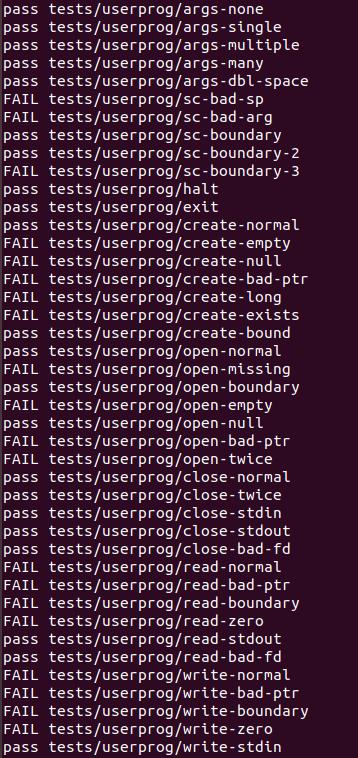
修改主要难点是指针的操作，慢慢理清楚还是不难的。

但是。。。。。评测结果依旧是没有得到任何输出。为了弄清楚原因，我在process\_execute ()中创建线程之前打印了一条输出，评测结果是含有这条输出的。在各个地方试图输出之后，我大概弄明白了原因。

在进行测试的时候，pintos的main函数里执行了process\_wait (process\_execute (task)),这个pintos进程(tid为1)启动了task测试线程(tid为3)，并且调用process\_wait等待这个task测试线程测试结束。但是pintos源代码的process\_wait()是直接返回-1，还未等待测试线程结束，pintos就结束了。自然就没有输出了。process\_wait()的实现应该是TASK3中最难实现的部分，目前先不着急。但是为了检测TASK2的成果，可以在process\_wait()中加入一条大数据循环，延长process\_wait()的执行时间，变相达成等待子线程结束的目的。



加入这一个for循环后，



可以通过25个测试点，其中包括所有的args-开头的测试点，完成了TASK1-2。剩下的剩下的测试点通过的可能原因是它本身就没有任何输出，以及write中试图打开一个错误的文件（事实上根本没有实现打开文件的功能）。

## Task3

### 3.1要求

我们需要实现13个系统调用。包括之前没有完成全部任务的sys\_exit(),sys\_wait()和sys\_write()。

文档中提出了几个要求。首先是对地址有效性的检查，这在前两个task中没有考虑。其次是同步系统调用，禁止多个进程进程对于文件的访问，以及系统调用在各种情况下能够正常执行。其中wait()函数的实现应该是主要难点。

## Task4