Lab2

实验报告

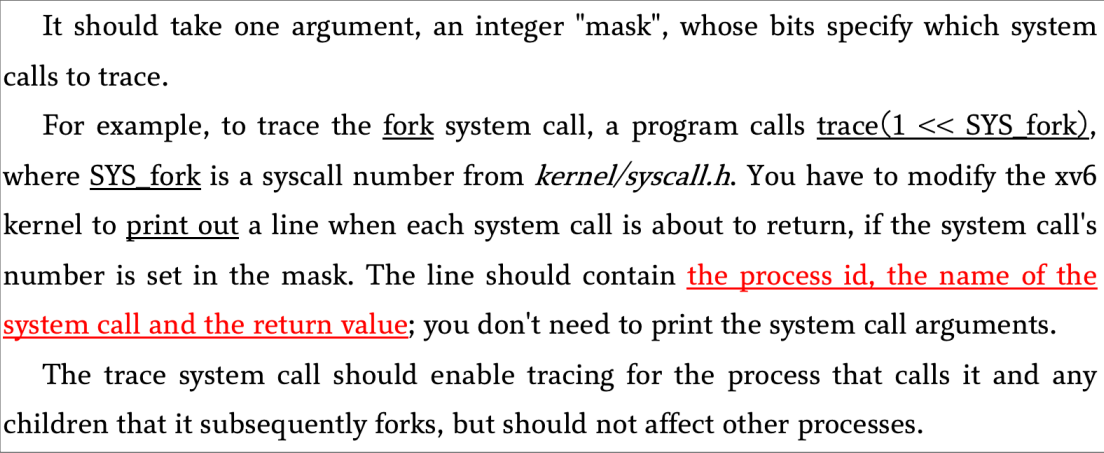
姓名 李晓畅

学号 20307130261

班级 计算机科学技术

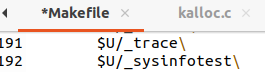
# 实现思路

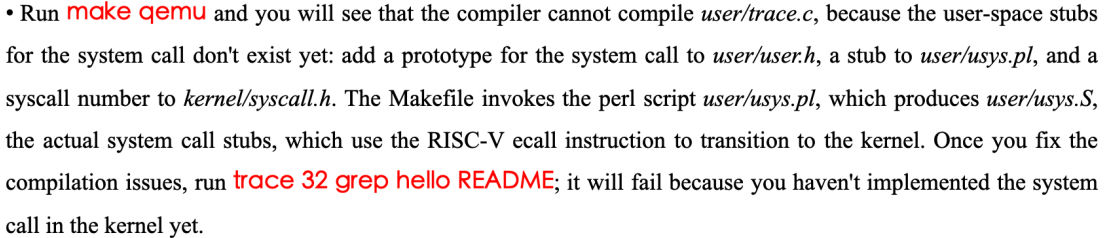
## Part A



按照提示进行：

* 

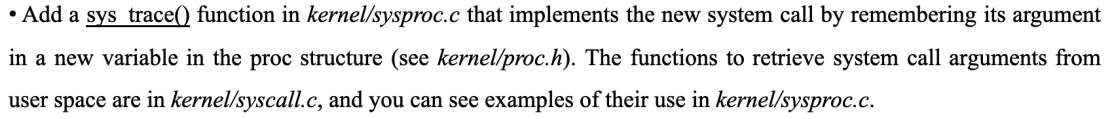
因为我们需要编译这些代码，所以我们把它们添加到***Makefile***中

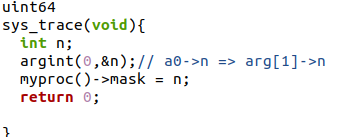
* 

 1. 需要在***user.h***中添加函数声明；

 2. 在***usys.pl***中添加，这是一个Perl文件，在make时生成***usys.S***文件，这个文件作用在于系统调用号通过***li(load imm)***存入a7寄存器，之后使用***ecall***进入内核态，最后返回；

 3. 在***syscall.h***中添加系统调用号；

* 下面就可以实现函数的接口了，具体来说：



按照其他函数的方式调用***argint***读取参数；

***argint*** 利用用户空间的 ***%esp*** 寄存器定位第 n 个参数：***%esp*** 指向系统调用结束后的返回地址。参数就恰好在 ***%esp*** 之上***（%esp+4）***。因此第 n 个参数就在 ***%esp+4+4\*n***；

换句话说这里读取了第一个参数，也就是mask；

根据题目描述，应该只追踪进程和它的子进程的对应系统调用，所以应该更新对应进程的mask信息；

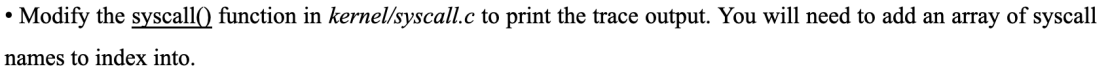
因而对应的结构体也需要修改：

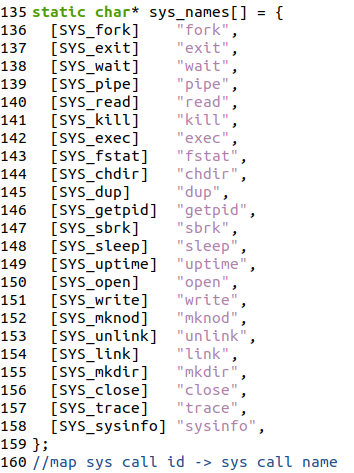


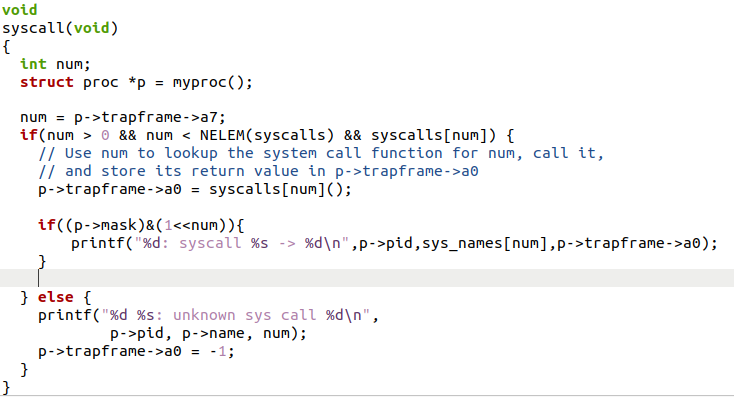
Mask应该是私有信息，所以之后也不需要获得lock

* 

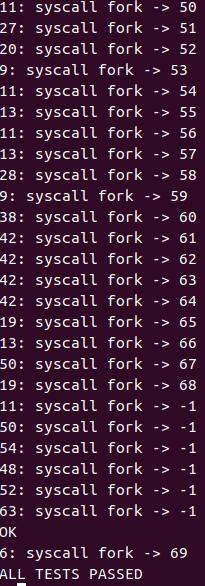
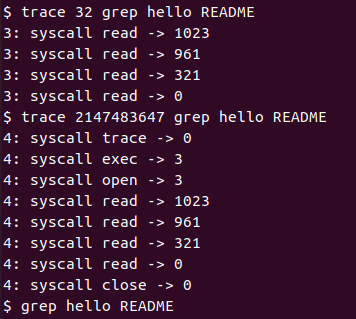
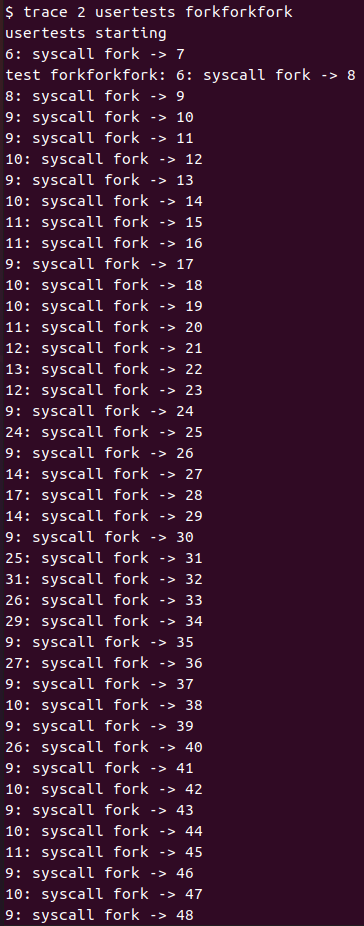
需要对***forck***函数进行一些修改，使得子进程可以继承父进程的mask

* 

因为需要打印调用的名称，所以需要一张调用号与名字间的映射表。参照上方不难写出这样的映射。

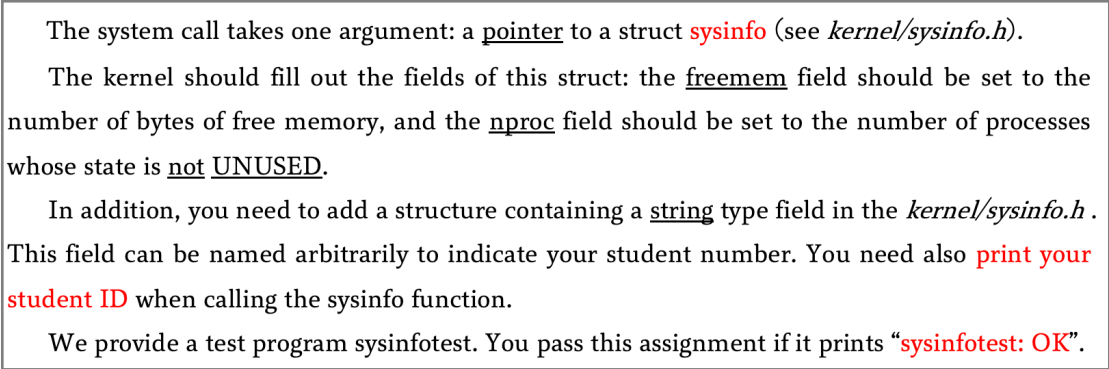
如果系统调用成功，就根据mask检查是否需要追踪对应的系统调用，若需要则做对应输出。

测试结果截图：

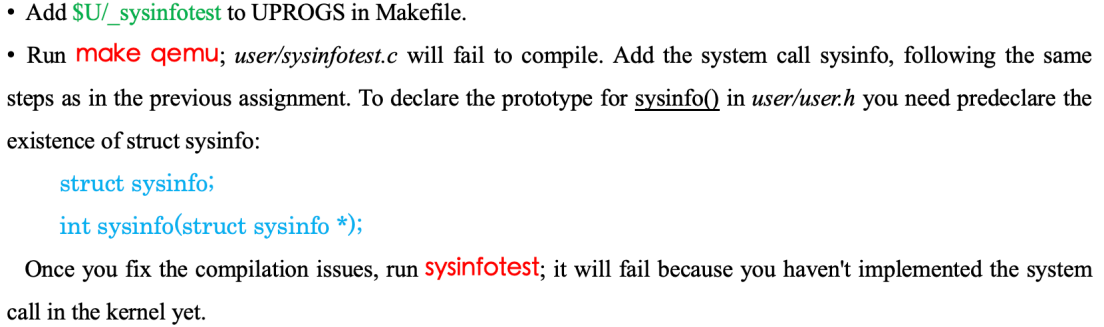


运行成功

## Part B



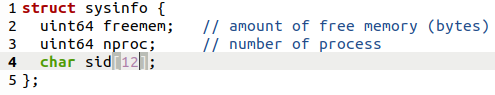
按照提示进行：

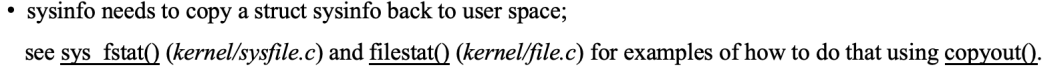
* 

类似于之前修改文件，修改可参PART A中图片；

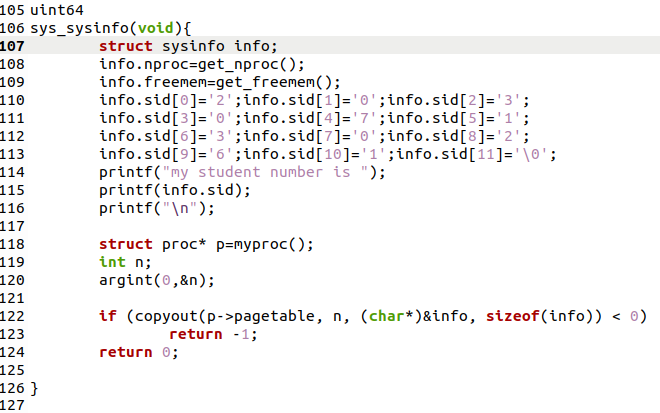
因为包含新的结构体，所以在***user.h***中除函数声明外还要添加结构体声明；

另外，***sysinfo.h***是这样的：



* 

提示中指出我们实现syscall中的代码使用copyout实现：



调用时先计算进程数和空间剩余，并对应初始化一个sysinfo以用于拷贝；

类似可以获取参数；

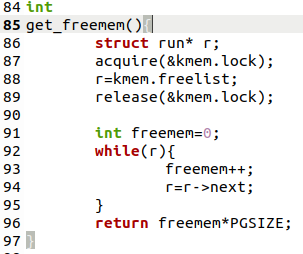
根据提示参考***copyout()***用法，见***filestat()***:





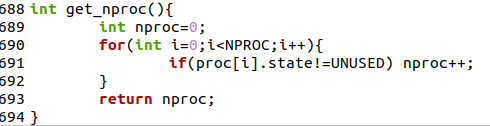
参照即可调用***copyout()***实现复制

* 

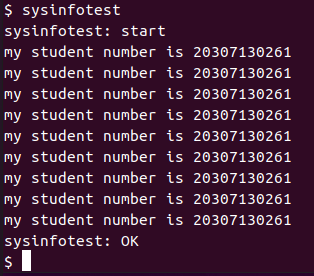


先获取空闲存储页链表，遍历，计数即可得到总共的空闲页；再乘以页大小即可获得空闲内存。

* 

直接统计进程表中非UNUSED的部分就是对应结果了。

测试结果截图：



运行成功

# 问题回答

## trace 全流程

* 在用户态通过系统调用trace为当前进程的proc设置mask表示要求追踪的系统调用；
* 用户进行系统调用，由用户态切换为内核态，用户进程经由***syscall()***进行系统调用时进行检查，如果mask指示该进程需要追踪，那么在调用结束后就会打印对应信息；
* 另外，如果进程创建了子进程，那么子进程也会有相同的mask信息，这意味着在子进程进行系统调用时也会进行检查和输出；
* 系统调用结束后会回到用户态，继续执行用户代码。

## kernel/syscall.h

正如注释所说，定义了系统调用号，这在***syscall.c***中我们就使用过。

每个系统调用被赋予一个系统调用号。这样，通过独一无二的号就可以关联系统调用。当用户空间的进程执行一个系统调用的时候，这个系统调用号就用来指明到底是要执行哪个系统调用；进程不会提及系统调用的名称。

系统调用号相当重要，一旦分配就不能再有任何变更，否则编译好的应用程序就会崩溃。此外，如果一个系统调用被删除，它所占用的系统调用号也不允许被回收利用，否则，以前编译过的代码会调用这个系统调用，但事实上却调用的是另外一个系统调用。

## trace 字段

用户态下的。

正如在前面流程分析中表明的那样，这里的trace意味着发起了一个系统调用，具体实现，也就是***sys\_trace()***将在内核态中执行。

# 问题和解决

## 调试和检查

在PART B中，代码可以运行，但是一直跑不出结果。

利用添加输出语句、GDB调试方法并加以分析，定位问题是获取空余存储死循环。修改加入改变循环变量即可正确运行。

## 语法细节

在修改学号时发现不管怎么写都不对，不论是：

还是：

似乎都不行，最后采取的办法是逐个赋值。

参考***string.h***,似乎并不支持C++那样赋值？

# 实验感想

## 面向切面

PART A中的需求实现似乎就是一种希望基于切面编程的思想。

我们希望为调用前后添加一些公共的操作，但并不希望改变调用的过程，如果可以尽可能少的改变结构，那就再好不过了。

我们在实现上直接修改了代码，实现了这一需求。但是直接修改代码或许并不一定是最好的方法，如果提示没有说明的话，准确更改代码其实应该是不容易的一件事情。

由此我们看到面向切面编程（Aspect Oriented Programming，AOP）是有实际意义的，它可以通过预编译方式和运行期动态代理的方式实现不修改源代码的情况下给程序动态统一添加功能。

## 类比

Xv6的函数不像很多库那样有非常完备的注释，因而理解它的作用或许需要观察它在其他函数中的使用来实现。参照其他地方编写映射、调用函数，这样同样可以实现目的。换句话说，我们理解代码的方式其实是多种多样的。