# 系统调用实习报告

姓名 张煌昭 学号 1400017707
日期 2017.11.27

# 目录

# 内容一：总体概述

本次 lab 需要在虚拟内存机制上实现系统调用，这些系统调用可以向用户提供所需要的功能。当用户调用这些系统调用时，nachos 从用户态转入为系统态，并执行相应的处理函数。本次 lab 的系统调用分为 10 个，和文件系统相关的有 5 个，和用户程序相关的有 5 个。

# 内容二：任务完成情况

## 任务完成列表（Y/N）

| | Exercise1 | Exercise2 | Exercise3 | Exercise4 | Exercise5 |
|---|---|---|---|---|---|
| 第一部分 | Y | Y | Y | Y | Y |

## 具体 Exercise 的完成情况

# 第一部分. 源码阅读

**Exercise 1　源代码阅读**

**阅读与系统调用相关的源代码，理解系统调用的实现原理。**

- **code/userprog/syscall.h**
- **code/userprog/exception.cc**
- **code/test/start.s**

**code/userprog/syscall.h**

该文件中定义了系统调用对应的系统调用号，以及声明了每个需要实现系统调用。

**code/userprog/exception.cc**

实际上在前面的Lab4已经修改过这个文件中的部分代码，通过Exceptionhandler来处理缺页和TLB miss，并通过修改Halt来实现测试中需要的功能。

ExceptionHandler是一个中断处理函数，传入的参数为ExceptionType。当ExceptionType为SyscallException时，该函数处理的是系统调用，系统调用号存储在2号寄存器中，可以根据系统调用号来判断是何种系统调用。在处理完成后，如果有返回值，则将它存储在2号寄存器中。

**code/test/start.s**

该文件中定义了进入系统调用处理函数之前的汇编代码，类似于系统调用的总入口。在每个系统调用对应的入口程序中，将系统调用号放入2号寄存器中，然后跳转到exception.cc文件中执行。

当用户执行一条系统调用命令时，OneInstruction函数解析该指令并发现其为系统调

用，之后进入start.s进行系统调用的操作。

在具体实现系统调用函数之前，需要先实现一个 PC 增加 4 的函数。在 nachos 中，每个系统调用返回时，都需要对 PC 增加 4，否则就会推出之后又进行系统调用而进入死循环。所以，在 machine 类中增加一个 AddPC 函数，该函数用于对 PC 值得更新，该函数具体需要更新的 PC 相关的寄存器有 PrevPCReg，PCReg 和 NextPCReg。其中 PrevPCReg 更新为 PCReg，PCReg 和 NextPCReg 各更新为原来的值+4。

# 第二部分. 文件系统相关的系统调用

**Exercise 2 & 3　文件的系统调用实现 & 编写用户程序**

**类比 Halt 的实现，完成与文件系统相关的系统调用：Create, Open，Close，Write，Read。Syscall.h 文件中有这些系统调用基本说明。**

**编写并运行用户程序，调用实现的文件系统调用，测试正确性。**

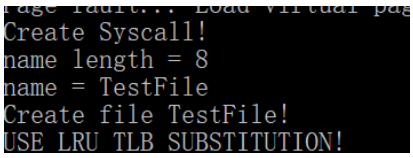
本次 Lab 中均使用 Linux 的文件系统，Makefile 中使用 FILESYS_STUB 宏定义。在 ExceptionHandler 中仿照 Halt 系统调用，添加 Create，Open，Close，Write 和 Read 系统调用。

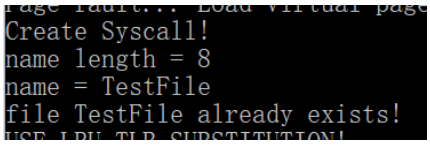
**Create 系统调用**

查看 syscall.h 中关于 Create()的定义，有一个参数指明文件名。因此需要在函数的开始阶段将文件名取出，得到文件名后首先判断文件是否存在，若不存在则使用 FileSystem 中的 Create 函数创建文件。返回时使用 AddPC 增加 PC 值。

整个函数如下，分作几步：首先获取文件名的地址指针，之后从模拟器主存中获取该地址下的字符串长度，接下来获取字符串；检查 FileSystem 中该文件是否存在，若存在则创建失败返回；否则文件不存在，可以进行创建，使用 FileSystem 的 Create 方法创建文件，默认创建的文件大小为 100B，由于存在动态调整机制，该大小不会有什么影响。

修改 Lab4 编写的 my_matrixtest.c，调用 Create 系统调用创建名为 TestFile 的文件。运行./nachos -x ../test/my_matrixtest，结果如下图，创建了 TestFile。再次运行，创建失败，显示已创建过文件。查看 test 文件夹下，已有 TestFile 文件。




**Open 系统调用**

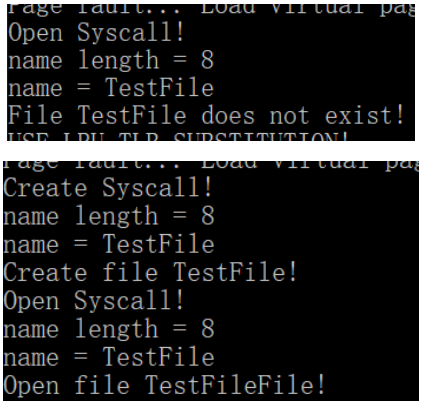
查看 syscall.h 中关于 Open()的定义，有一个参数指明文件名。因此需要在函数的开始阶段将文件名取出，之后通过 FileSystem->Open()打开文件，若返回指针为空则打开失败返回；否则打开成功，将返回的打开文件指针强制转换为 int 类型放入 2 号寄存器作为返回值。返

回时使用 AddPC 增加 PC 值。

此外，为了方便 Close，Read 和 Write 系统调用中判断文件是否打开，修改 Thread 类，在其中设置一个打开文件数组 fileOpen，用于记录打开文件指针。

整个函数如下，分作几步：首先获取文件名的地址指针，之后从模拟器主存中获取该地址下的字符串长度，接下来获取字符串；通过 fileSystem 打开该文件，若打开失败则打印信息并返回；否则打开文件成功，将打开的文件添加到现成的 fileOpen 数组中再返回。

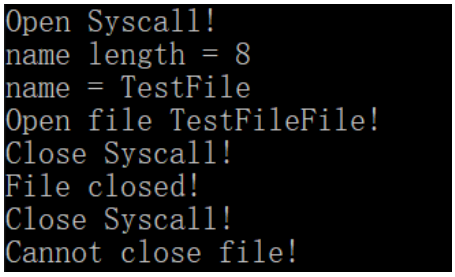
修改 my_matrixtest.c，调用 Open 系统调用打开 TestFile 文件。首先删除创建的 TestFile，运行./nachos -x ../test/my_matrixtest，结果如下图，找不到 TestFile；再按照 Create 中的创建方式创建 TestFile，再次运行./nachos -x ../test/my_matrixtest，成功打开 TestFile。




**Close 系统调用**

查看 syscall.h 中关于 Close()的定义，有一个参数作为待关闭的打开文件的指针。读入该参数后，首先在线程的打开文件数组中查找是否有该文件，若找不到说明该文件根本没有打开，报错返回；否则找到，关闭该文件，清除数组中该描述符。最后使用 AddPC 更新 PC。

修改 my_matrixtest.c，先调用 Open 系统调用打开 TestFile 文件，之后再接连两次调用 Close 系统调用关闭文件。结果如下图，第一次 Close 调用成功关闭文件，第二次报错。




**Write 系统调用**

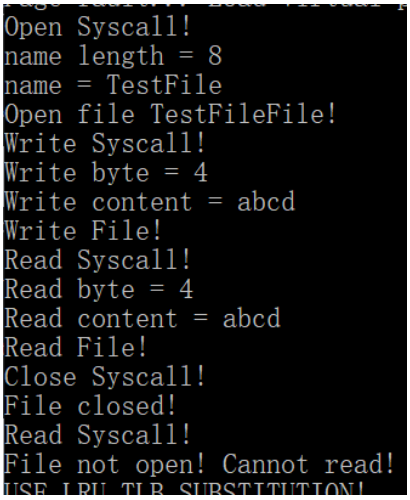
查看 syscall.h 中关于 Write()的定义，第一个参数为要写入内容的指针，第二个参数为写入长度，第三个参数为打开文件指针。从 4，5，6 号寄存器读出参数后，首先遍历线程内的 fileOpen 数组判断文件是否打开，若没有则报错返回；否则文件已经打开，调用 OpenFile->Write()将待写内容写入文件，真正写入的长度为待写入长度和待写入内容真实长度中的较小值。最后使用 AddPC 更新 PC。

测试与 Read 系统调用一起进行。

**Read 系统调用**

查看 syscall.h 中关于 Read()的定义，第一个参数为要读入数据的缓冲区指针，第二个参数为读入数据长度，第三个参数为打开文件指针。操作基本与 Write 调用相同。读出参数后，首先判断文件是否打开，若打开则调用 OpenFile->Read()读出数据，由于有返回值（真正读出的字节数），将其放入 2 号寄存器。最后使用 AddPC 更新 PC。

修改 my_matrixtest.c，首先打开 TestFile 文件，之后 Write 写入'abcd'，再 Read 读出 4 个字节，最后关闭文件，关闭文件后再次读 4 个字节。结果如下图，第一次写入和读出均正常，第二次读没有打开的文件报错。




# 第三部分. 执行用户程序相关的系统调用

**Exercise 4　用户程序执行的系统调用实现**

**实现如下系统调用：Exec，Fork，Yield，Join，Exit。Syscall.h 文件中有这些系统调用基本说明。**

**Exec 系统调用**

Exec 系统调用执行一个可执行文件。

Exec 系统调用有一个参数 name，指明文件名。因此处理函数需要一开始解析参数，调用 ReadMem 函数从主存中读出 name。然后处理函数所做的操作就有点类似 Addrspace 类的构造函数所做的工作了，可以仿照 ptest 中的 StartProgress()函数进行实现。

首先尝试将文件打开，如果无法打开说明该文件存在问题，需要打印提示信息并返回。如果该文件存在且能打开，则新创建一个线程，并让该线程执行 start 函数，将可执行文件名作为 start 函数的参数传入。

在 start 函数中，按照 Addrspace 类的构造函数为用户程序申请空间，并初始化寄存器和 machine 的 pagetable 指针，然后调用 machine 类的 run 函数即可。由于 Exec 系统调用需要返回值，由 syscall.h 知返回值为 spaceId，由于在 nachos 中一个线程有一个地址空间，所以可以将其理解为线程指针，因此在 exec 函数的最后将函数中创建的线程强制转化为 int 后写入 2 号寄存器。

**Fork 系统调用**

Fork 系统调用完全赋值一个线程的地址空间和寄存器堆，并返回其指针。

Fork 系统调用需要一个参数，该参数是一个函数指针，指明新创建的线程需要执行的函数。这个 Fork 的功能与 Nachos 中的 Fork 不同，Nachos 中的 Fork 更类似于 Exec，此处的 Fork 有点类似 pthread_create，新创建的线程只是为了运行一个函数。

fork 出的线程和原线程具有完全相同的地址空间的拷贝，因而不存在共享数据的问题，这种方法更像是 linux 中的 fork 系统调用。在后一种实现中，在 Addrspace 类中实现了 Addrspacecopy 函数，将另一线程的地址空间复制进本线程的地址空间，然后再将之前线程的寄存器的值保存进本线程的寄存器，并修改 PC 为新线程运行函数的基地址。

其余的所有操作均类似于 AddrSpace 的构造函数。Fork 函数返回之后，AddPC()更新 PC。

**Join 系统调用**

Join 系统调用有一个参数，指明需要等待的线程的指针。因此在处理函数的开始需要解析参数，由于参数是 int 类型的，需要将其强制转化为线程指针类型。然后需要在线程池中查找是否存在等待的线程，如果不存在，打印错误信息返回即可。

如果等待的线程确实存在的话，就需要一直等待，直到等待的线程调用了 finish 之后。在我的实现中，我让调用 join 的线程一直执行 yield 函数，直到等待的线程在线程池中不存在了，也就是等待的线程 finish 之后，当前线程才能跳出循环继续执行。在 Join 函数返回后，AddPC()将 pc 更新。

**Yield 系统调用**

非常简单，直接调用 Yield 函数即可。

**Exit 系统调用**

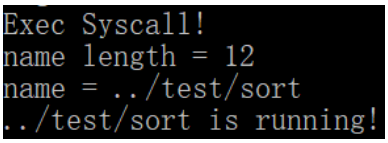
Lab4 中已经实现了 Exit 系统调用，可以回收地址空间并接收一个参数标明推出状态。根据参数打印不同的信息，若为 0 则正常退出，否则打印报错。

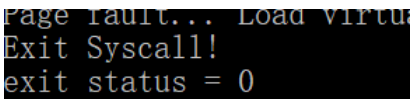
**Exercise 5　编写用户程序**

**编写并运行用户程序，调用练习 4 中所写系统调用，测试其正确性。**

修改 my_matrixtest.c，通过 Exec 系统调用执行 sort 程序，在 sort 程序中通过 Exit(A[0])来判断排序是否正确。由于原始的 sort 程序有误，需要修改 sort 为正确的冒泡排序算法。

打印的结果如下。程序正常正常切换到 sort 并执行，退出状态也正常。




# 内容三：遇到的困难以及解决方法

**1. 地址对齐报错**

在测试时会遇到 mipssim.cc 中的一个地址检查的 ASSERT 断言报错，查看该断言发现是检查地址对齐的，但由于我在获取文件名时需要逐个字节地访问内存，因此删去此处断言。

**2. 判断打开文件**

由于 Close，Read 和 Write 都需要判断文件是否打开，但一直没有想到如何进行判断。最后想到 Linux 下各个进程会保存自己的打开文件的描述符，指向打开文件，最终也采用这一方法。

**3. test/sort.c 排序出错**

由于原始的排序代码有误，使得我一直误以为 Exit 实现出错。最终发现是排序出错，修改后解决。

# 内容四：收获及感想

由于本次 Lab 需要使用的接口原生的 Nachos 也都已经提供了，因此实现起来相对容易。但是在细节需要非常注意，尤其是检查出错状态等需要格外留心。

# 内容五：对课程的意见和建议

暂无

# 内容六：参考文献

暂无