`

系统调用实习报告

杨东升 1400012898

2017.5.24

目录

内容一：总体概述

这次的实习内容是关于 Nachos 的系统调用的实现，需要考虑到很多因素，例如如何处理系统调用，参数如何传递，返回值如何设置等等，还需要考虑到系统调用和 Linux 宿主机的协作。对于文件系统的调用，需要借用 Linux 的文件系统的操作；而对于执行用户程序的调用，需要手动加载用户程序，增加线程，以及添加新的数据结构维护标号等等。

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				
第二部分		Y	Y		
第三部分				Y	Y

（总之就是都完成了）

具体 Exercise 的完成情况

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

在 syscall.h 中，准备了两种机制满足用户程序的系统调用需求。当 IN_ASM 这个宏被定义了，就使用宿主机 Linux 系统的系统调用指令 syscall 来实现，即通过代码 start.s，使用汇编语言来借用 Linux 的系统调用函数。如果没有定义 IN_ASM，则需要自己实现，函数名称序号的定义已经写在 syscall.h 文件中。

在 Start.s 中定义了__start 函数，11 个系统调用函数，和主函数。每个系统调用函数都只是简单的跳转到了 linux 相应系统函数的入口，执行后返回。__start 函数是初始化调用主

函数，主函数什么也不做直接返回。

系统调用是异常的一种。exception.cc 中的 ExceptionHandler()是 nachos 的异常处理函数，首先通过 which 来判断异常号，如果异常号是 SyscallException，则是系统调用。然后可以通过第二个寄存器的值来得到系统调用号。Nachos 需要的系统调用一共有 11 个，halt()已经实现了，其余的我们在后面的 exercise 中依次实现。

关于参数的传递，Nachos 规定分别使用寄存器 r4、r5、r6、r7 来传递第 1、2、3、4 个参数。系统调用号通过 r2 寄存器传递，返回值同样也放到 r2 寄存器中。

Exercise2　系统调用实现

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。

这些系统调用是文件相关的，实现的基本思路是先从寄存器读取参数，然后调用相关的文件系统函数，最后令 PC 值增加，并把返回值写回寄存器。

我在 AddrSpace 类中加入了一个 BitMap 类型的变量，用于存储已被使用的文件描述符。

```
BitMap * filemap ;
```

我改动的文件主要是 exception.cc，首先要引文件系统和 map 的头文件。

```
#include "openfile.h"
#include <map>
```

然后在 exceptionHandler()中，根据异常号和系统调用号进行系统调用处理。

首先是 Create 系统调用，它接受一个字符数组参数。我从 4 号寄存器读取文件名在 nachos 内存中的地址，然后逐个把字符从内存读入数组。不能连续读，因为读内存函数是以 4 字节为一个单位的，而字符类型是 1 字节。如果字符串长度超过 100 或字符串结束，就不再读入。然后，我调用文件系统的创建文件函数。最后，我更新三个 PC 寄存器的值。代码实现如下：

```
else if ((which == SyscallException) && (type == SC_Create)) {
    char content[100] ;
    int offset = machine->ReadRegister(4), count ;
    for( count = 0 ; count < 100 ; count ++)
    {
        int temp ;
        machine->ReadMem(offset + count, 1, &temp) ;
        printf("%c", temp) ;
        content[count] = temp ;
        if(content[count] == '\0') break ;
    }
    printf("create: %s\n", content) ;
    fileSystem->Create(content, 10) ;
    machine->registers[PrevPCReg] = machine->registers[PCReg] ;
    machine->registers[PCReg] = machine->registers[NextPCReg] ;
    machine->registers[NextPCReg] = machine->registers[NextPCReg] + 4 ;
}
```

接下来是 Open 打开文件的系统调用，它接受一个字符串参数，返回一个整数描述符。先读参数，然后在 BitMap 中找到一个未分配的描述符，调用文件系统打开文件函数，再把描述符和打开文件指针成对插入到 map 容器中。最后更新 PC 寄存器，需要把返回值写入 2 号寄存器。代码如下：

```
else if ((which == SyscallException) && (type == SC_Open)) {
    char content[100] ;
    int offset = machine->ReadRegister(4), count ;
    for( count = 0 ; count < 100 ; count ++)
    {
        int temp ;
        machine->ReadMem(offset + count, 1, &temp) ;
        content[count] = temp ;
        if(content[count] == '\0') break ;
    }
    OpenFile* openfile = fileSystem->Open(content) ;
    int symbol = currentThread -> space -> filemap -> Find() ;
    dcrpMap.insert(std::make_pair(symbol, openfile)) ;
    machine->registers[PrevPCReg] = machine->registers[PCReg] ;
    machine->registers[PCReg] = machine->registers[NextPCReg] ;
    machine->registers[NextPCReg] = machine->registers[NextPCReg] + 4 ;
    machine->WriteRegister(2, symbol) ;
    printf("open %s into %d\n", content, symbol) ;
}
```

关闭文件的系统调用接受整数文件描述符，删去 BitMap 和 map 中的字段，代码如下：

```
else if ((which == SyscallException) && (type == SC_Close)) {
    int id = machine->ReadRegister(4) ;
    OpenFile * openfile = dcrpMap[id] ;
    currentThread -> space -> filemap -> Clear(id) ;
    dcrpMap.erase(id) ;
    printf("close: %d\n", id) ;
    machine->registers[PrevPCReg] = machine->registers[PCReg] ;
    machine->registers[PCReg] = machine->registers[NextPCReg] ;
    machine->registers[NextPCReg] = machine->registers[NextPCReg] + 4 ;
}
```

读文件和写文件的系统调用也是一样的流程，这次有三个参数，所以要分别从 4,5,6 号寄存器中读取。代码如下：

需要额外说明的是，在 userprog 编译时，默认使用 Unix 宿主机的文件系统函数，因此创建的文件在文件夹下是对我们可见的。

```
else if ((which == SyscallException) && (type == SC_Write)) {
    int offset = machine->ReadRegister(4) ;
    int size = machine->ReadRegister(5) ;
    int id = machine->ReadRegister(6) ;
    char * content = new char[size] ;
    for( int i = 0 ; i < size; i ++ )
        machine -> ReadMem(offset+i, 1, (int*)(content+i)) ;
    OpenFile * openfile = dcrpMap[id] ;
    openfile -> Write(content, size) ;
    printf("write %s into %d\n", content, id) ;
    delete content ;
    machine->registers[PrevPCReg] = machine->registers[PCReg] ;
    machine->registers[PCReg] = machine->registers[NextPCReg] ;
    machine->registers[NextPCReg] = machine->registers[NextPCReg] + 4 ;
}
else if ((which == SyscallException) && (type == SC_Read)) {
    int offset = machine->ReadRegister(4) ;
    int size = machine->ReadRegister(5) ;
    int id = machine->ReadRegister(6) ;
    char * content = new char[size] ;
    OpenFile * openfile = dcrpMap[id] ;
    openfile -> Read(content, size) ;
    for( int i = 0 ; i < size; i ++ )
        machine -> WriteMem(offset+i, 1, (int*)(content+i)) ;
    machine->WriteRegister(2, content) ;
    printf("read %s into %d\n", content, id) ;
    delete content ;
    machine->registers[PrevPCReg] = machine->registers[PCReg] ;
    machine->registers[PCReg] = machine->registers[NextPCReg] ;
    machine->registers[NextPCReg] = machine->registers[NextPCReg] + 4 ;
```

Exercise3　编写用户程序

编写并运行用户程序，调用练习 2 中所写系统调用，测试其正确性。

我在 halt 程序中加入了文件操作：

```
int fd ;
char content[10] ;
int main()
{
    Create("testFile") ;
    Create("anotherFile") ;
    fd = Open("testFile") ;
    content[0] = '1', content[1] = '2', content[2] = '3',
    content[3] = '4', content[4] = '5' ;
    Write(content, 5, fd) ;
    Close(fd) ;
    fd = Open("testFile") ;
    Read(content+5, 5, fd) ;
    Close(fd) ;
    Halt();
}
```

由于 nachos 编译器的一些限制，数组必须一个一个字符赋值。在读写之间需要关闭并重开一次文件，来复位文件游标。

运行结果如下：

```
vagrant@precise32:/vagrant/nachos/n
testFile create: testFile
using Linux fileSystem.
anotherFile create: anotherFile
using Linux fileSystem.
open testFile into 0
write 12345 into 0
close: 0
open testFile into 0
read 12345 into 0
close: 0
Machine halting!
```

说明程序运行正确。

Exercise4　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 是在当前线程中新建一个线程，并且让新线程执行一个可执行文件。首先还是读取文件名参数，然后创建线程并令其执行 StartProg 函数，接下来把该线程插入到线程池中，方便 Join 时对其进行监视。最后返回线程号，令 PC 前移。创建新线程以后并不立即切换线程，而是让新线程等待。代码如下：

```
else if ((which == SyscallException) && (type == SC_Exec)) {
    char * content ;
    content = new char[100] ;
    int offset = machine->ReadRegister(4), count ;
    for( count = 0 ; count < 100 ; count ++)
    {
        int temp ;
        machine->ReadMem(offset + count, 1, &temp) ;
        content[count] = temp ;
        if(content[count] == '\0') break ;
    }
    printf("ready to exec %s\n", content) ;
    Thread * mthread = new Thread(content) ;
    int symbol = currentThread -> space -> progmap -> Find() ;
    progMap.insert(std::make_pair(symbol, mthread)) ;
    mthread->Fork(StartProg, (int)content) ;
    machine->WriteRegister(2, symbol) ;
    machine->registers[PrevPCReg] = machine->registers[PCReg] ;
    machine->registers[PCReg] = machine->registers[NextPCReg] ;
    machine->registers[NextPCReg] = machine->registers[NextPCReg] + 4 ;
}
```

StartProg 函数是仿照 StartProcess 函数编写的，唯一区别就是加了一句输出。之所以重新写一遍，是为了防止编译时交叉引用出现问题。

```
void
StartProg(int ifilename)
{
    char * filename = (char *) ifilename ;
    printf("here: %s\n", filename) ;
    OpenFile *executable = fileSystem->Open(filename);
    AddrSpace *space;

    if (executable == NULL) {
    printf("Unable to open file %s\n", filename);
    return;
    }
    space = new AddrSpace(executable);
    currentThread->space = space;

    delete executable;          // close file

    space->InitRegisters();     // set the initial register values
    space->RestoreState();      // load page table register
    printf("start program %s\n", filename) ;
    machine->Run();         // jump to the user progam
    ASSERT(FALSE);          // machine->Run never returns;
                    // the address space exits
                    // by doing the syscall "exit"
}
```

Join 的功能是等待目标线程结束。实现方法是读取目标线程的状态，如果目标线程被析构，或者处于结束状态，则退出循环，否则挂起当前线程，等待下次循环。返回退出值。

```
else if ((which == SyscallException) && (type == SC_Join)) {
    int id = machine->ReadRegister(4) ;
    int PrevPC = machine->registers[PCReg] ;
    int PC = machine->registers[NextPCReg] ;
    printf("wait for prog %d\n", id) ;
    while(1)
    {
        Thread * thread = progMap[id] ;
        if( thread == NULL ) break ;
        if( thread -> status == BLOCKED ) break ;
        currentThread->Yield() ;
    }
    currentThread->space->RestoreState() ;
    machine->WriteRegister(2, exitNum) ;
    machine->registers[PrevPCReg] = PrevPC ;
    machine->registers[PCReg] = PC ;
    machine->registers[NextPCReg] = PC + 4 ;
}
```

Fork 是在当前线程的上下文中运行一个新的函数，这样其实原来的函数就不能再运行了。实现方法是设置 PC 值为目标函数地址。

```
else if ((which == SyscallException) && (type == SC_Fork)) {
    int PC = machine->ReadRegister(4) ;
    printf("prog jump(fork) to %d\n", PC) ;
    machine->registers[PrevPCReg] = machine->registers[PCReg] ;
    machine->registers[PCReg] = PC ;
    machine->registers[NextPCReg] = PC + 4 ;
}
```

Yield 是挂起当前线程，只需要调用 thread 的 Yield 函数即可。

```
else if ((which == SyscallException) && (type == SC_Yield)) {
    printf("prog yield\n") ;
    currentThread->Yield() ;
    machine->registers[PrevPCReg] = machine->registers[PCReg] ;
    machine->registers[PCReg] = machine->registers[NextPCReg] ;
    machine->registers[NextPCReg] = machine->registers[NextPCReg] + 4 ;
}
```

Exit 是令线程结束，还要设置返回值供 Join 读取。

```
else if ((which == SyscallException) && (type == SC_Exit)) {
    int arg = machine->ReadRegister(4) ;
    exitNum = arg ;
    printf("prog exit with %d\n", arg) ;
    currentThread->Finish() ;
    machine->registers[PrevPCReg] = machine->registers[PCReg] ;
    machine->registers[PCReg] = machine->registers[NextPCReg] ;
    machine->registers[NextPCReg] = machine->registers[NextPCReg] + 4 ;
}
```

Exercise5　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 halt.c，原来的主函数放到了 simple 函数中，现在的主函数先 Exec sort 程序，然后 Join 子线程，然后 Yield 当前线程，最后 Fork simple 函数。simple 和 sort 都会调用 Exit

现在 halt 可以同时测试所有我实现的系统调用。

```c
#include "syscall.h"

int fd, td ;
char content[10] ;

void simple()
{
    Create("testFile") ;
    fd = Open("testFile") ;
    content[0] = '1', content[1] = '2', content[2] = '3',
    content[3] = '4', content[4] = '5' ;
    Write(content, 5, fd) ;
    Close(fd) ;
    fd = Open("testFile") ;
    Read(content, 5, fd) ;
    Close(fd) ;
    Exit(0) ;
}
int main()
{
    td = Exec("sort") ;
    Join(td) ;
    Yield() ;
    Fork(simple) ;
```

运行结果如下所示，首先 exec sort，然后执行 join sort，这时主线程下 CPU，sort 开始运行，然后 sort 以 1023 的返回值退出；主线程 Yield，然后主线程 Fork simple 函数，simple 函数创建文件，打开文件，写文件，关闭文件，打开文件，读文件，关闭文件。最后主线程以返回值 0 结束。

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/userpr
ready to exec sort
wait for prog 0
here: sort
start program sort
prog exit with 1023
prog yield
prog jump(fork) to 208
testFile create: testFile
using Linux fileSystem.
open testFile into 0
write 12345 into 0
close: 0
open testFile into 0
read 12345 into 0
close: 0
prog exit with 0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

内容三：遇到的困难以及解决方法

困难 1：理解系统调用的流程

系统调用是用户函数通过 syscall.h 定义的接口函数，实际调用 start.c 中的汇编程序，这些汇编程序只是填入了系统调用号，然后统一调用的 exceptionHandler 函数，所以我们所有的系统调用处理只需要在 exceptionHandler 中根据系统调用号来分别完成就可以。

困难 2：传参，更新 PC

异常和系统调用不同，系统调用总是返回到下一条指令，因此需要对 PC 值进行更新，这一点很容易被忽视；使用 Exec 进行进程切换时我也保存了 PC 值，但实际上不用，因为在 scheduler.cc 中 nachos 已经帮我们存储和取回了线程的运行状态；传递参数分别使用 4,5,6 号寄存器，参数只能传整形，其他形式要强转，返回值要放到 2 号寄存器。

困难 3：需要内存系统支持多线程

我是从一个全新的 nachos 系统开始实现这个 lab 的，在进行 exec 时，发现从子线程返回到主线程会导致指令错误异常，经过仔细研究，我发现是子线程覆盖了主线程的内存空间，所以需要实现多线程虚拟内存机制，让新线程从新的地方开始一段地址空间。

困难 4：文件游标的问题

写文件之后直接读文件会读到空值，检查后发现是文件操作的游标没有归零，而是从写的内容后面开始读，所以读不到写的内容。因此，需要在写和读之间重新打开文件。

内容四：收获及感想

系统调用是程序员和系统之间的纽带，实现系统调用，让我对于操作系统执行用户文件的具体行为有了更深刻的认识；编写用户程序也让我对 nachos 的编译器，执行单元有了更深的认识。这次 lab 以后，我可以从更宏观的角度看操作系统了。

内容五：对课程的意见和建议

无。

内容六：参考文献

【1】现代操作系统　陈向群