shell 实现实习报告

姓名张颜 学号 2001210541
日期 2020/1/6

目录

内容一：总体概述

本次实验主要是需要实现一个 nachos 下的 shell。Shell 相当于基于 nachos 操作系统给用户提供一个使用接口，用户通过 shell 输入命令来和操作系统进行交互。由于 shell 是操作系统提供给用户的接口，所以在逻辑上 shell 应当运行在用户态上。这就意味着，不应当把 shell 和操作系统内核代码直接连接起来。本次实验结合用户程序和系统调用进行 shell 的实现。主要实现的功能有 cd ls mkdir touch rmf rmd 等命令。

内容二：任务完成情况

任务完成列表（Y/N）

	Exercise
	Y

具体 Exercise 的完成情况

Exercise

设计实现一个用户程序 shell，通过./nachos -x shell 进入用户交互界面中。在该界面中可以查询支持的功能、可以创建删除文件或目录、可以执行另一个用户程序并输出运行结果，类似 Linux 上跑的 bash。

你实现的越完善，碰到的问题越多，学到的也会越多。

本实验所修改的代码包括内核和用户程序两部分。

Shell 是操作系统提供给用户的接口。用户通过 shell 命令和操作系统进行交互。故而，逻辑上 shell 本身也是一种用户程序。Nachos 提供了一个 shell 框架用于循环接受用户输入，但是没有进一步实现 shell 的命令。通过执行用户程序 shell.c，打开 shell。这样就使得 shell 运行在 nachos 的虚拟机器上，而不是直接运行在宿主机上。由于用户程序运行在用户态，无法直接和操作系统内核代码进行交互，故而，想要使用操作系统提供的功能，需要使用系统调用陷入内核态，在内核态中使用操作系统实现的功能。

Nachos 已经提供了十个系统调用，这些系统调用满足了部分需求，但是要增加 shell 的功能还需要提供更多的系统调用扩展。在查阅了资料以后，扩展了部分的系统调用。

一 扩展系统调用：

扩展了 11 到 17 号系统调用。增加的系统调用对应的功能如下：

系统调用名称	系统调用号	函数名	功能
SC_Ls	11	Ls()	显示当前路径下所有的文件和目录
SC_Pwd	12	Pwd()	显示当前工作路径
SC_Cd	13	Cd(char* path)	切换工作路径
SC_Rmf	14	Rmf(char *name)	删除文件
SC_Rmd	15	Rmd(char *name)	删除目录
SC_Mkdir	16	Mkdir(char *name)	新建目录
SC_Touch	17	Touch(char *name)	新建文件

下面以 Ls 为例子，介绍系统调用添加的整个过程。

首先，在 syscall.h 中声明自己添加了新的系统调用。并且为新的系统调用分配系统调用号。

```
#define SC_Ls 11
#define SC_Pwd 12
#define SC_Cd 13
#define SC_Rmf 14
#define SC_Rmd 15
#define SC_Mkdir 16
#define SC_Touch 17
```

其次在同一个文件中添加对系统调用接口的声明。包括用户调用函数名称，以及函数的参数。通过这种声明，当用户程序引入了 syscall.h 后可以找到对应的系统调用。

```
void Ls();


void Pwd();

void Cd(char *path);

void Rmf(char *filename);

void Rmd(char *directname);

void Mkdir(char *name);

void Touch(char *name);
```

在 syscall.h 中完成系统调用的声明以后，下一步就需要链接了。

在 start.c 和 start.s 文件中添加对应系统调用的链接。（以 Ls 系统调用为例）

```
    .globl Ls
    .ent    Ls
Ls:
    addiu $2,$0,SC_Ls
    syscall
    j   $31
    .end Ls
```

```
    .globl Ls
    .ent    Ls
Ls:
    addiu $2,$0,SC_Ls
    syscall
    j   $31
    .end Ls
```

最后，在 exception.cc 中添加系统调用的处理函数。

```
else if((which == SyscallException) && (type == SC_Ls))
{
    //printf("SC_LS\n");
    system("ls");
    machine->PCOneTick();
}
```

通过这些流程，一个系统调用就可以被添加到 nachos 中。用户程序在 include<”syscall.h”> 文件后，就可以直接调用这些系统调用。

二、Shell.c 文件

Nachos 给了一个 shell 的实现框架。
在该框架中，通过两个 while 循环的嵌套，不断从标准的控制台输入读取用户的输入命令。对于控制台的输入，定义了一个 while 循环，一次一个字符，通过 Read 系统调用从控制台读入命令。每次读入完成后，对命令进行判断，判断完成以后执行，执行完毕之后开始接受其他的命令。While 循环的嵌套方法如下所示。

```
while( 1 )
{
Write(prompt, 2, output);

i = 0;

do {

    Read(&buffer[i], 1, input);
    tmp = (buffer[i]);

} while( buffer[i++] != '\n' );
```

可以看见，该文件中对控制台的输入输出都通过系统调用 Read 和 Write。在前面的实验中，Read 和 Write 的实现都是针对文件进行，现在要对这两个系统调用的处理函数进行改写。可以发现，针对控制台的输入输出，在调用系统调用时，最后一个参数不同。

```
OpenFileId input = ConsoleInput;
OpenFileId output = ConsoleOutput;
```

```
#define ConsoleInput    0
#define ConsoleOutput   1
```

```
Read(&buffer[i], 1, input);
```

```
Write(prompt, 2, output);
```

故而修改 Read 和 Write 的系统调用处理函数，对于不同的参数，支持不同的功能。

```
if (fid == ConsoleInput)
{
    for (int i = 0 ; i < size; i++)
    {
        content[i] = getchar();
        machine->WriteMem(bufferAddr+i , 1 , int(content[i]));
    }

}
```

对于 Read，当最后一个参数等于 ConsoleInput 的时候，将会从控制台一个字符一个字符地读入用户输入进入内存，最后将内存中的字符逐个写入到用户给定的地址空间。在 shell.c 中，定义了一个大小为 60bytes 的输入缓冲数组 buffer。

```
if(fid == ConsoleOutput)
{
    for (int i = 0 ; i < size; i++)
    {
        machine->ReadMem(bufferAddr + i , 1 , &data);
        putchar(data);
    }
}
```

对于 Write，当最后一个参数等于 ConsoleOutput 的时候，将会从用户输入的地址中一个一个地取出字符输出到控制台上。

最后我们添加一个简单的 shell 命令 quit。用户调用该命令的时候会退出 nachos 系统。我们在循环后添加一个字符串判断函数，当用户输入 quit 命令时，会直接调用 Halt 系统调用来退出。测试结果如下：

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code2/userprog$ ./nachos -x ../test/shell
Successfully Add Thread main to GlobalList!
-$quit
this is halt syscall  current thread is : main
Machine halting!

Ticks: total 239, idle 0, system 10, user 229
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

执行 shell 用户程序，可以看见命令行输出字符 -$。而后键入 quit 命令，系统陷入 Halt。

三、添加各种功能。

添加 ls 命令：

在 shell.c 中添加对该命令的检测代码，如果检测到该命令那么直接陷入系统调用 Ls()。Ls()系统调用的实现直接调用 Linux 的 ls 命令。该函数需要引入#include<stdlib.h>头文件。

```
else if((which == SyscallException) && (type == SC_Ls))
{
    //printf("SC_LS\n");
    system("ls");
    machine->PCOneTick();
}
```

添加 pwd 命令：

如上添加对该命令的检测代码，检测到该命令以后直接陷入系统调用 Pwd()。

Pwd()系统调用也是直接调用 Linux 的 ls 命令。

```
else if((which == SyscallException) && (type == SC_Pwd))
{
    //printf("SC_Pwd\n");
    system("pwd");
    machine->PCOneTick();

}
```

测试：

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code2/userprog$ ./nachos -x ../test/shell1
Successfully Add Thread main to GlobalList!
-$pwd
/vagrant/nachos/nachos-3.4/code2/userprog
-$ls
addrspace.cc  bitmap.h      exception.o  main.o     progtest.cc  switch.o     syscall.h  thread.o      utility.o
addrspace.h   bitmap.o      interrupt.o  Makefile   progtest.o   swtch.s      sysdep.o   threadtest.o
addrspace.o   console.o     list.o       mipssim.o  scheduler.o  synchlist.o  system.o   timer.o
bitmap.cc     exception.cc  machine.o    nachos     stats.o      synch.o      testfile   translate.o
-$cd ..
-$pwd
/vagrant/nachos/nachos-3.4/code2
-$ls
bin  filesys  machine  Makefile  Makefile.common  Makefile.dep  network  test  threads  userprog  vm
-$
```

可以看见在不同的目录显示不同的目录下内容。

添加 cd 命令：

在 shell 中添加 cd 命令的相关判断代码，当检测到 cd 命令的时候直接陷入 Cd 系统调用。Cd 系统调用的处理函数也是直接调用 chdir 函数切换当前工作目录。但是 Cd 系统调用是由参数的，故而需要在四号寄存中中找到对应的参数。

```
else if((which == SyscallException) && (type == SC_Cd))
{
    int nameAddr = machine->ReadRegister(4);
    char name[20];
    int offset = 0;
    int data;
    while(true)
    {
        machine->ReadMem(nameAddr + offset , 1 , &data);
        if (data == 0)
        {
            name[offset] = '\0';
            break;
        }
        name[offset] = char(data);
        offset+=1;
    }
    chdir(name);

    machine->PCOneTick();
}
```

测试：

```
agrant@precise32:/vagrant/nachos/nachos-3.4/code2/userprog$ ./nachos -x ../test/shell
uccessfully Add Thread main to GlobalList!
$pwd
/vagrant/nachos/nachos-3.4/code2/userprog
$cd ..
$pwd
/vagrant/nachos/nachos-3.4/code2
$cd /
$pwd
/
$cd /vagrant
$pwd
/vagrant
```

Cd 切换目录，通过已经实现的 pwd 命令来检查结果是否正确。

添加 mkdir rmf rmd touch 命令

这些命令的添加都需要先在 shell.c 中添加对应的命令检测代码，而后陷入相应的系统调用。各自实现的代码如下：

```
else if((which == SyscallException) && (type == SC_Rmf))
{
    int nameAddr = machine->ReadRegister(4);
    char name[20];
    int offset = 0;
    int data;
    while(true)
    {
        machine->ReadMem(nameAddr + offset , 1 , &data);
        if (data == 0)
        {
            name[offset] = '\0';
            break;
        }
        name[offset] = char(data);
        offset+=1;
    }
    remove(name);
    machine->PCOneTick();
}
```

```
else if((which == SyscallException) && (type == SC_Rmd))
{
    int nameAddr = machine->ReadRegister(4);
    char name[20];
    int offset = 0;
    int data;
    while(true)
    {
        machine->ReadMem(nameAddr + offset , 1 , &data);
        if (data == 0)
        {
            name[offset] = '\0';
            break;
        }
        name[offset] = char(data);
        offset+=1;
    }
    remove(name);
    machine->PCOneTick();
}
```

```
else if((which == SyscallException) && (type == SC_Mkdir))
{
    int nameAddr = machine->ReadRegister(4);
    char name[20];
    int offset = 0;
    int data;
    while(true)
    {
        machine->ReadMem(nameAddr + offset , 1 , &data);
        if (data == 0)
        {
            name[offset] = '\0';
            break;
        }
        name[offset] = char(data);
        offset+=1;
    }
    mkdir(name,0777);
    machine->PCOneTick();
}
```

```
else if((which == SyscallException) && (type == SC_Touch))
{
    int nameAddr = machine->ReadRegister(4);
    char name[20];
    int offset = 0;
    int data;
    while(true)
    {
        machine->ReadMem(nameAddr + offset , 1 , &data);
        if (data == 0)
        {
            name[offset] = '\0';
            break;
        }
        name[offset] = char(data);
        offset+=1;
    }
    fileSystem->Create(name , 128);
    machine->PCOneTick();
}
```

测试：

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code2/userprog$ ./nachos -x ../test/shell
Successfully Add Thread main to GlobalList!
-$ls
addrspace.cc  bitmap.h      exception.o  main.o     progtest.cc  switch.o     syscall.h  thread.o      utility.o
addrspace.h   bitmap.o      interrupt.o  Makefile   progtest.o   swtch.s      sysdep.o   threadtest.o
addrspace.o   console.o     list.o       mipssim.o  scheduler.o  synchlist.o  system.o   timer.o
bitmap.cc     exception.cc  machine.o    nachos     stats.o      synch.o      testfile   translate.o
-$mkdir testdir
-$ls
addrspace.cc  bitmap.h      exception.o  main.o     progtest.cc  switch.o     syscall.h  testfile      translate.o
addrspace.h   bitmap.o      interrupt.o  Makefile   progtest.o   swtch.s      sysdep.o   thread.o      utility.o
addrspace.o   console.o     list.o       mipssim.o  scheduler.o  synchlist.o  system.o   threadtest.o
bitmap.cc     exception.cc  machine.o    nachos     stats.o      synch.o      testdir    timer.o
-$cd testdir
-$pwd
/vagrant/nachos/nachos-3.4/code2/userprog/testdir
-$ls
-$touch ytest
-$ls
ytest
-$rmf ytest
-$ls
-$cd ..
-$pwd
/vagrant/nachos/nachos-3.4/code2/userprog
-$ls
addrspace.cc  bitmap.h      exception.o  main.o     progtest.cc  switch.o     syscall.h  testfile      translate.o
addrspace.h   bitmap.o      interrupt.o  Makefile   progtest.o   swtch.s      sysdep.o   thread.o      utility.o
addrspace.o   console.o     list.o       mipssim.o  scheduler.o  synchlist.o  system.o   threadtest.o
bitmap.cc     exception.cc  machine.o    nachos     stats.o      synch.o      testdir    timer.o
-$rmd testdir
-$ls
addrspace.cc  bitmap.h      exception.o  main.o     progtest.cc  switch.o     syscall.h  thread.o      utility.o
addrspace.h   bitmap.o      interrupt.o  Makefile   progtest.o   swtch.s      sysdep.o   threadtest.o
addrspace.o   console.o     list.o       mipssim.o  scheduler.o  synchlist.o  system.o   timer.o
bitmap.cc     exception.cc  machine.o    nachos     stats.o      synch.o      testfile   translate.o
-$
```

新建目录 testdir，而后进入该目录在该目录下新建文件 ytest。而后删除对应文件，退出目录删除该目录。各项功能正常。

内容三：遇到的困难以及解决方法

困难

遇到的主要困难是不够细致，对 Read 和 Write 系统调用理解不够深刻导致无法控制台接受和显示用户的输入输出，卡在此处花了不少时间 Debug。

在调用和宿主机相关的操作系统接口的时候，查阅资料有时会因为平台不对，相应的接口也改变的情况，需要不断细致的进行资料查阅。

内容四：收获及感想

之前对 Linux 的 shell 理解不够深刻，不知道它具体的实现到底是什么样子的。这个实验让我对操作系统的用户态和内核态有了更加具体的认识，了解了为什么操作系统要分用户态和内核态。

同时了解了不同平台上很多接口的差别，这让我对很对技术如 java 虚拟机的出现有了逻辑上的理解。

创造用户接口的感觉很棒。

内容五：对课程的意见和建议

暂时无。

内容六：参考文献

[1]《nachos 中文教程》
https://wenku.baidu.com/view/905197a9e209581b6bd97f19227916888486b90b.html
[2]《nachos 学习笔记（五）》https://blog.csdn.net/darord/article/details/83303765