Lab5 系统调用

李糖 2001210320

任务完成情况

Exercises	Y/N
Exercise1	*Y*
Exercise2	*Y*
Exercise3	*Y*
Exercise4	*Y*
Exercise5	*Y*

概述

本实习希望通过修改*Nachos*系统的底层源代码，达到“实现系统调用”的目标。

一、理解*Nachos*系统调用

Exercise1 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

> 定义*nachos*的系统调用,主要包括系统调用号和系统调用函数，内核通过识别用户程序传递的系统调用号确定系统调用类型。

```
//定义系统调用号
#define SC_Halt 0
#define SC_Exit 1
#define SC_Exec 2
#define SC_Join 3
#define SC_Create 4
#define SC_Open 5
#define SC_Read 6
#define SC_Write 7
#define SC_Close 8
#define SC_Fork 9
#define SC_Yield 10

//文件系统相关，5个：
void Create(char *name);                                    //创建名为name的文件
```

```
OpenFileId Open(char *name);                          //打开文件名为name的文件，返回打开文件的标识符（指针）
void Write(char *buffer, int size, OpenFileId id); //从buffer中写size个字节的数据到标识符为id的文件中
int Read(char *buffer, int size, OpenFileId id);   //从文件中读取size个字符到buffer中，返回实际读取的字节数
void Close(OpenFileId id);                            //关闭标识符为id的文件

//用户级线程相关，5个：
void Fork(void (*func)()); //创建和当前线程拥有相同地址空间的线程，运行func指针指向的函数
void Yield();              //当前线程让出CPU
void Exit(int status);     //用户程序执行完成，status = 0表示正常退出
SpaceId Exec(char *name);  //加载并执行名为name的可执行文件，返回其地址空间标识符SpaceId
int Join(SpaceId id);      //等待标识符为id的用户线程执行完毕，返回其退出状态
```

code/userprog/exception.cc

> 定义进行异常处理的*ExceptionHandler*函数，主要流程是根据异常信息处理不同异常，包括系统调用。

在注释中给出了寄存器的作用：

```
//  system call code -- r2
//    arg1 -- r4
//    arg2 -- r5
//    arg3 -- r6
//    arg4 -- r7
```

> *r2->*系统调用返回值
>
> *r4/5/6/7->*系统调用的四个参数

code/test/start.s

辅助用户程序运行的汇编代码，主要包括初始化用户程序和系统调用相关操作

背景知识

> MIPS中j，jr，jal这三个跳转指令有什么区别？
>
> j 跳转
> *eg：j 2500 //*跳转到目标地址*10000*，指令中的地址是字地址，所以需要乘以4，转换成字节地址。
> jal 跳转并链接
> *eg：jal 2500 //$ra=PC+4, PC=10000*，指令中的地址也是字地址，乘以4转换成字节地址。一般用于主程序调用函数时候的跳转，设置函数的返回地址为主程序中跳转指令的下一个指令，意思就是执行完函数就得回到主程序继续执行。寄存器$ra专门用来保存函数的返回地址。
> jr 跳转到寄存器所指的位置
> *eg：jr $ra //*跳转到寄存器中的地址。一般用于函数执行完返回主函数时候的跳转。

1. 初始化用户程序：通过调用*main*函数运行用户程序

```
  .globl __start
  .ent  __start
__start:
  jal main
  move  $4,$0
  jal Exit   /* if we return from main, exit(0) */
  .end __start
```

2. 系统调用：将系统调用号存入*r2*寄存器，跳转到*exception.cc*，以系统调用*Halt*为例：

```
.globl Halt
  .ent  Halt
Halt:
  addiu $2,$0,SC_Halt
  syscall
  j $31
  .end Halt
```

因为用的是j指令，所以在执行系统调用之后需要人为地将*PC*的值增加*4*，否则程序会进入死循环。

系统调用流程

> *machine*的*Run*函数运行用户程序，实现在*machine/mipssim.cc*，基本流程是通过*OneInstruction*函数完成指令译码和执行，通过*interrupt->OneTick*函数使得时钟前进
> （1）*OneInstruction*函数判断当前指令是系统调用，转入*start.s*
> （2）通过start.s确定系统调用入口，通过寄存器r2传递系统调用号，转入*exception*.cc（此时系统调用参数位于相应寄存器）
> （3）*exception*.cc通过系统调用号识别系统调用，进行相关处理，如果系统调用存在返回值，那么通过寄存器r2传递，流程结束时，需要更新*PC*
> （4）系统调用结束，程序继续执行

添加系统调用

> （1）*syscfall*.h定义系统调用接口、系统调用号
> （2）*code/test/start*.s添加链接代码
> （3）*exception*.cc添加系统调用处理过程

二、文件系统相关的系统调用

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。

为此我专门写了一个函数*FileSystemHandler*来处理文件系统系统调用:

```
else if (type == SC_Create || type == SC_Open || type == SC_Write || type ==
SC_Read || type == SC_Close)
        {
            FileSystemHandler(type);
            // Increment the Program Counter before returning.
            advancePC();
        }
```

Machine::advancePC()

> 自增*PC*

```
void Machine::advancePC()
{
    int currPC = ReadRegister(PCReg);
    WriteRegister(PrevPCReg, currPC);
    WriteRegister(PCReg, currPC + 4 );
    WriteRegister(NextPCReg, currPC + 8);
}
```

getNameFromNachosMemory()

> 从*Nachos*内存中获取文件名的辅助函数

```
char* getNameFromNachosMemory(int address) {
    int position = 0;
    int data;
    char *name = new char[FileNameMaxLength + 1];
    do {
        // each time read one byte
        machine->ReadMem(address + position, 1, &data);
        name[position++] = (char)data;
    } while(data != 0);
    return name;
}
```

Create()

```
void Create(char *name);
```

1. 通过*r4*寄存器获取文件名在*Nachos*中的地址
2. 通过*getNameFromNachosMemory*()获得文件名
3. 调用*filesys->Create*()创建文件
4. 调用*advancePC*函数自增*PC*

Open()

```
OpenFileId Open(char *name);
```

1. 通过*r4*寄存器获取文件名在Nachos中的地址
2. 通过*getNameFromNachosMemory*()获得文件名
3. 调用*filesys->Open*()打开文件
4. 将*openfile*的地址（所谓的*OpenFileId*）写入r2寄存器
5. 调用*advancePC*函数自增*PC*

Write()

```
void Write(char *buffer, int size, OpenFileId id);
```

1. 读取r4/5/6获取三个参数（*buffer*在*nachos*内存中，其余两个在宿主主机内存，直接强转即可）
2. 文件已经打开，通过*id*强转为*openfile*指针类型即可获得该打开文件

3. 调用*openfile->Write()*向*buffer*中写*size*个字节
4. 调用*advancePC*函数自增*PC*

Read()

```
int Read(char *buffer, int size, OpenFileId id);
```

1. 读取r4/5/6获取三个参数（*buffer*在*nachos*内存中，其余两个在宿主主机内存，直接强转即可）
2. 文件已经打开，通过*id*强转为*openfile*指针类型即可获得该打开文件
3. 调用*openfile->Read()*从*buffer*中读*size*个字节
4. 将*numBytes*存入*r2*寄存器
5. 调用*advancePC*函数自增*PC*

Close()

```
void Close(OpenFileId id);
```

1. 读取r4获取*openFile*型指针（初始是*int*型，强转为对应类型即可）
2. 使用*delete*析构该*openFile*
3. 调用*advancePC*函数自增*PC*

```
void FileSystemHandler(int type)
{
    if (type == SC_Create)
    {
        char *name = getNameFromNachosMemory(machine->ReadRegister(4));
        fileSystem->Create(name, 0);
        cout << "Success create file name " << name << endl;
        delete[] name; //回收内存
    }
    else if (type == SC_Open)
    {
        char *name = getNameFromNachosMemory(machine->ReadRegister(4));
        machine->WriteRegister(2, (OpenFileId)fileSystem->Open(name));
        cout << "Success open file name " << name << endl;
        delete[] name;
    }
    else if (type == SC_Close)
    {
        OpenFile *openFile = (OpenFile *)machine->ReadRegister(4);
        cout << "Success delete file ID " << (OpenFileId)openFile << endl;
        delete openFile;
    }
    else if (type == SC_Read)
    {
        int addr = machine->ReadRegister(4);
        int size = machine->ReadRegister(5);
        OpenFile *openFile = (OpenFile *)machine->ReadRegister(6);
        int numByte = openFile->Read(&machine->mainMemory[addr], size);
        machine->WriteRegister(2, numByte);
        cout << "Success read " << numByte <<
        "bytes from file ID " << (OpenFileId)openFile << endl;
    }
    else if (type == SC_Write)
    {
```

```
        currentThread->SaveUserState();
        int addr = machine->ReadRegister(4);
        int size = machine->ReadRegister(5);
        OpenFile *openFile = (OpenFile *)machine->ReadRegister(6);
        int numByte = openFile->Write(&machine->mainMemory[addr], size);
        cout << "Success write " << numByte << "bytes to file ID" <<
        (OpenFileId)openFile << endl;
        currentThread->RestoreUserState();
    }
}
```

思考

在阅读了*xv6*源代码之后，我发现*Nachos*缺少压栈过程：对于每一个*syscall*，在执行之前，都应该将*PC*保存起来，处理完*syscall*之后也需要将*PC*恢复。所以每个系统调用的正确写法都应该在首尾分别加上如下语句，以*Write*为例：

```
else if (type == SC_Write)
    {
        currentThread->SaveUserState();     //AKA压栈
        ...
        currentThread->RestoreUserState(); //AKA出栈
    }
```

然而在*start.s*中：

```
  .globl Write
  .ent  Write
Write:
  addiu $2,$0,SC_Write
  syscall ==>PC
  j $31
  .end Write
```

可以发现*syscall*处理过程中*PC*的值一直没有变过，所以，对于文件系统来说，压栈和出栈的过程可以省略，*exercise3*的结果将证明此结论的正确性。

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。

本次测试我使用的是*UNIX*文件系统。

我在*code/userprog/*目录下建立了一个文件*file1*,里面存放了一句诗：

```
rose is a rose is a rose.
```

在*code/test/*中编写了我的测试程序*file_syscall_test.c*, 在*code/test/MakeFile*中添加如下语句：

```
all: halt shell matmult sort 99table file_syscall_test

file_syscall_test.o: file_syscall_test.c
  $(CC) $(CFLAGS) -c file_syscall_test.c
file_syscall_test: file_syscall_test.o start.o
  $(LD) $(LDFLAGS) start.o file_syscall_test.o -o file_syscall_test.coff
  ../bin/coff2noff file_syscall_test.coff file_syscall_test
```

这个程序的目的是把*file1*中的内容*copy*到*file2*中，本测试*cover*全部的*5*个文件系统调用:

```
#include "syscall.h"
#define QUOTE_LEN 40    //诗的长度
int fileID1, fileID2;   //两个打开文件的id
int numBytes;           //实际读取的字符数
char buffer[QUOTE_LEN]; //缓冲区
int main()
{
    Create("file1");                              //创建文件
    fileID1 = Open("file1");                      //打开文件file1
    fileID2 = Open("file2");                      //打开文件file2
    numBytes = Read(buffer, QUOTE_LEN, fileID1); //从file1中读取字节到buffer中
    Write(buffer, numBytes, fileID2);             //再从buffer中写入file2中
    Close(fileID1);                               //关闭两个文件
    Close(fileID2);
    Halt();
}
```

测试

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/code/userprog$ ./nachos -x
../test/file_syscall_test
Success create file name file2
Success open file name file1
Success open file name file2
Success read 25bytes from file ID 136279000
Success write 25bytes to file ID136279016
Success delete file ID 136279000
Success delete file ID 136279016
Machine halting!
```

在*code/userprog/*目录下出现了*file2*，内容为:

```
rose is a rose is a rose.
```

结论

实验成功!

三、执行用户程序相关的系统调用

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。

前面提到了文件系统中可以省略压栈和出栈的过程，这是*Nachos*系统的性质导致的。然而在本次*exercise*中，因为涉及到线程的切换，是不是就意味着压栈/出栈过程不可省略呢？结论是否定的，因为在*Scheduler::Run*()中会自动保存和恢复*context*，所以之前的结论在本次实验中依然成立。

```
void Scheduler::Run(Thread *nextThread)
{
    ...
    if (currentThread->space != NULL)
    {                                   // if this thread is a user program,
        currentThread->SaveUserState(); // save the user's CPU registers
        currentThread->space->SaveState();
    }
    ...
    SWITCH(oldThread, nextThread);
    ...
    if (currentThread->space != NULL)
    {                                     // if there is an address space
        currentThread->RestoreUserState(); // to restore, do it.
        currentThread->space->RestoreState();
    }
}
```

我写了一个函数*FileSystemHandler*来处理用户程序系统调用:

```
else if (type == SC_Exit || type == SC_Exec || type == SC_Join || type == SC_Fork
|| type == SC_Yield)
{
    UserProgHandler(type);
}
```

Exec()

```
SpaceId Exec(char *name);
```

加载并执行名为*name*的可执行文件，具体流程可以参考*startProcess*():

```
void StartProcess(char *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

    machine->Run(); // jump to the user progam
    ASSERT(FALSE);  // machine->Run never returns;
                    // the address space exits
                    // by doing the syscall "exit"
}
```

关于*spaceID*：

```
/* A unique identifier for an executing user program (address space) */
typedef int SpaceId;
```

注释里给出的定义为“每个用户程序的一个”独特的标识符”，这个标识符可以是*space*的地址，也可以是*thread*的地址，甚至可以是*thread*的*ID*（*lab1*），我认为三者都可以，因为它们都是对thread的唯一描述，并且”同生（构造时）同死（析构时）“，我选用*threadID*的来实现, 因为*Nachos*的*Finish*函数有*BUG*，可能会导致*threadToBeDestroyed*指针不被释放，这点在*lab1*中有详细论述。

Fork()

```
void Fork(void (*func)());
```

1. 获取函数在*Nachos*内存中的地址
2. *new*一个线程，并*Fork*辅助函数*fork_helper*
3. 在辅助函数中，对新线程的寄存器和页表初始化
4. 调用*Machine::Run*()开始执行
5. PC自增

Yield()

```
void Yield();
```

1. 调用*currentThread->Yield*()
2. *PC*自增

Join()

```
int Join(SpaceId id);
```

1. 获取线程标识符
2. 当该线程存在的情况下调用*currentThread->Yield()*主动让出*CPU*
3. *PC*自增

Exit()

```
void Exit(int status);
```

1. 获取线程退出状态，并打印
2. 释放内存中与*currentThread*有关的位图中的位
3. *PC*自增
4. 调用*currentThread->Finish*()函数结束运行

```
//----------------------------------------------------------------------
//  UserProgHandler
//  Handling user program related system call.
//----------------------------------------------------------------------
void UserProgHandler(int type)
{
    if (type == SC_Exec)
    {
        int addr = machine->ReadRegister(4);
        Thread *thread = new Thread("exec_thread");
        thread->Fork(exec_func, addr);
        machine->WriteRegister(2, (SpaceId)thread->getTID());
        machine->advancePC();
    }
    else if (type == SC_Fork)
    {
        int funcAddr = machine->ReadRegister(4);
        // Create a new thread in the same addrspace
        Thread *thread = new Thread("fork_thread");
        thread->space = currentThread->space;
        thread->Fork(fork_func, funcAddr);
        machine->advancePC();
    }
    else if (type == SC_Yield)
    {
        currentThread->Yield();
        machine->advancePC();
    }
    else if (type == SC_Exit)
    {
        int status = machine->ReadRegister(4);
        printf("%s exist with status %d.\n", currentThread->getName(), status);
        machine->bitmap->freeMem(); //释放位图
        machine->advancePC();
        currentThread->Finish();
    }
    else if (type == SC_Join)
    {
        int threadID = machine->ReadRegister(4);
        while (!isAllocatable[threadID])
            currentThread->Yield();
        machine->advancePC();
    }
}
```

其中两个辅助函数如下：

```
//helper function to call by Fork()
void exec_helper(int addr) //mainMemory start positon
{
    char *name = getNameFromNachosMemory(addr);
    OpenFile *executable = fileSystem->Open(name);
    AddrSpace *space = new AddrSpace(executable);
    currentThread->space = space;
    delete[] name; //回收内存
    delete executable;
    space->InitRegisters();
```

```
    space->RestoreState();
    machine->Run();
    ASSERT(FALSE); //never return
}
//helper function to called by Fork()
void fork_func(int arg)
{
    currentThread->space->InitRegisters(); //初始化寄存器
    currentThread->space->RestoreState();  //继承父进程的页表
    // Set PC to *arg*
    machine->WriteRegister(PCReg, arg); //从函数处开始执行
    machine->WriteRegister(NextPCReg, arg + 4);
    machine->Run();
    ASSERT(FALSE); //never return
}
```

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。

测试

我在*code/test/*中编写了我的测试程序*user_prog_test.c*, 并在*code/test/MakeFile*中添加相关依赖。

```
#include "syscall.h"
int spaceID;
void func()
{
    Create("text1");
    Exit(0);
}

int main()
{
    Fork(func);
    Yield();
    spaceID = Exec("../test/file_syscall_test");
    Exit(0);
}
```

这个函数的流程是：

1. 主函数先*fork*一个线程，它将创造一个名为*text1*的文件；
2. 主函数让出*CPU*，让这个*fork*线程先执行完毕，正常退出；
3. 然后主函数调用*Exec*执行*exercise2*中使用的测试文件*file_syscall_test*;
4. 主函数正常退出；
5. *exercise2*的测试程序正常退出。

本测试*cover*了*Exit/Fork/Yield/Exec*四个系统调用，测试结果如下：

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/code/userprog$ ./nachos -d t -x
../test/user_prog_test
Forking thread "fork_thread" with func = 0x804f5ac, arg = 208
Switching from thread "main" to thread "fork_thread"
Success create file name text1
fork_thread exists with status 0.
```

```
Switching from thread "fork_thread" to thread "main"
Forking thread "exec_thread" with func = 0x804f4c5, arg = 360
main exists with status 0.
Switching from thread "main" to thread "exec_thread"
Success create file name file2
Success open file name file1
Success open file name file2
Success read 25bytes from file ID 147248656
Success write 25bytes to file ID147248672
Success delete file ID 147248656
Success delete file ID 147248672
exec_thread exists with status 0.
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

结论

结果显示，主函数先*fork*一个线程，让出*CPU*；这个线程创造了一个名为*text1*的文件，正常退出；然后主函数调用Exec执行*exercise2*中使用的测试文件*file_syscall_test*，主函数正常退出，最终，*exercise2*的测试程序正常退出。

符合预期，实验成功。

困难&解决

Assertion failed: line 156, file "../machine/sysdep.cc"

为了获取文件名，我使用了强制类型转换：

```
if (type == SC_Create)
   {
       char* name = (char*)machine->ReadRegister(4);
       fileSystem->Create(name, 0);
   }
```

报错：

```
Assertion failed: line 156, file "../machine/sysdep.cc"
Aborted
```

错误原因：*register4*中存放的是*Nachos*模拟的*mainMemory*的元素的下标（*eg. mainMemory[200]*，*register*中存放的就是*200*），而不是宿主主机的指针。强转只能针对后一种情况。

栈区VS.堆区

*getNameFromNachosMemory()*获取的文件名为乱码：

```
read.@
>,h•••••••
>,h•••••••
```

乱码一般因为指针指向了错误的内存。审查代码：分配数组的方式为

```
char name[FileNameMaxLength + 1];
```

这个数组被分配到了栈区，*getNameFromNachosMemory*()结束之后会被回收。

解决：改用动态数组：

```
char *name = new char[FileNameMaxLength + 1];
```

这个数组在堆区，*getNameFromNachosMemory*()结束之后依然存在，但是用完之后一定要记得手动释放，避免内存泄漏。

```
delete[] name;
```

收获&感想

搞清楚每个变量到底是在*Nachos*内存，还是在宿主主机内存。这关系到是使用辅助函数还是直接使用强制类型转换。

参考文献

https://github.com/daviddwlee84/OperatingSystem/blob/master/Lab/Lab6_SystemCall/README.md

https://wenku.baidu.com/view/bd03f40ee97101f69e3143323968011ca300f71e.html?re=view

https://github.com/SergioShen/Nachos