Lab1 Nachos 基础系统调用的添加

唐凯成，PB16001695

1. 实验目的

1、在 Nachos 内核添加下面五个系统调用：
（1）创建文件：int Create(char*name);
（2）打开文件：OpenFileId Open(char *name);
（3）写入文件：int Write(char *buffer, int size, OpenFileId id);
（4）读取文件：int Read(char *buffer, int size, OpenFileId id);
（5）关闭文件：int Close(OpenFileId id);
2、使用测试代码文件对每个系统调用进行测试运行，并检查输出结果

2. 实验步骤

（1）定义系统调用号和系统调用接口

对于本次需要添加的五个系统调用，Nachos 系统中均已预写好其调用号与调用接口，无需再次添加，在 code/userprog/syscall.h 中五个系统调用的系统调用号分别为：

```
#define SC_Create       4
#define SC_Open         6
#define SC_Read         7
#define SC_Write        8
#define SC_Close        10
```

（2）、添加进入内核系统调用的接口

本次实验中的五个系统调用接口也已预先写好，检查 code/test/start.S 即可，故无需再次添加相关系统调用的接口。

（3）、在内核中修改中断入口处理函数与实现系统调用函数

由于这两者为本次实验主要添加的内容且其联系较为紧密，故此处将这两步合并并分别对五个系统调用的添加进行分析：

1）Create 系统调用

a）修改中断入口处理函数：在 code/userprog/exception.cc 中加入以下程序段：

```
case SC_Create:
//Create 系统调用的中断入口
    DEBUG(dbgSys, "Create" << kernel->machine->ReadRegister(4) << "\n");
    //系统调用中预设的 DEBUG 调用，方便 make 时给出错误的具体位置
    int c_address,result_create;
    //定义创建文件的地址和 SysCreate 的返回值整形变量
    c_address = (int)kernel->machine->ReadRegister(4);
    /*从寄存器 r4 中读取创建文件的地址，ReadRegister 是 kernel->machine 类中定义用来读取寄存器值的函数，此处仿照其他地方调用的格式进行调用*/
    result_create = SysCreate(c_address);
    //调用 SysCreate 系统调用函数完成调用并将返回值赋给 result_create 变量
    kernel->machine->WriteRegister(2, result_create);
    //将 result_create 写入寄存器 r2 中，WriteRegister 的调用与上述 ReadRegister 类似
```

```
{
kernel->machine->WriteRegister(PrevPCReg,kernel->machine->ReadRegister(PCReg));
kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) + 4);
kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg) + 4);
//调用结束后将 PC 指针值加 1，即向后移 4 位，在每个中断结束后的固定套用语句
}
    return;
    ASSERTNOTREACHED();
    break;//中断结束跳出 switch 语句
```

至此即完成了 Create 系统调用中断入口的设置，将所需创建文件的地址从寄存器取出并代入系统调用函数 SysCreate 中。

b）在内核中实现系统调用函数：在 code/userprog/ksyscall.h 中添加以下函数：

```
int SysCreate(int address)
{
    int i;//定义循环变量 i
    int letter=1;// 定义用来转存文件名的变量 letter 并将初始值置 1
    char filename[128];//定义新建文件的文件名字符串（最大长度 128）
    for(i=0;i<128, letter!= 0;i++)//在 i<128 不超过长度上限且读入的字母不为空时循环
    {
        kernel->machine->ReadMem(address+i,1,& letter);
        //调用 ReadMem 从存储器中读取地址为 address+i 的 1 个字母并存入 letter 中
        filename[i]=(char) letter;
        //将刚读入的字母 letter 存入文件名字符串 filename 的第 i 位
    }//该循环读取并存储了新建文件的命名
    if(kernel->fileSystem->Create(filename)) return 1;
    else return -1;
    /*调用系统内核预设好的 kernel->fileSystem 类中的 Create(char *filename)函数，用来创建文件名为 filename 的文件。创建成功时 Create 返回 1，SysCreate 也返回 1；失败时 Create 返回 0，SysCreate 返回-1。*/
}
```

2）Open 系统调用

a）修改中断入口处理函数：（省略 PC 指针后移及其后的重复段内容）

```
case SC_Open:
    DEBUG(dbgSys, "Open" << kernel->machine->ReadRegister(4) << "\n");
    int o_address,result_open;
    o_address = (int)kernel->machine->ReadRegister(4);
    result_open = SysOpen(o_address);
    //整体与 Create 调用无本质区别，将要打开文件的地址代入 SysOpen 函数中
    kernel->machine->WriteRegister(2, result_open);
```

b）在内核中实现系统调用函数：（与 SysCreate 类似的地方不再解释说明）

```
int SysOpen(int address)
```

```
    {
        int i,id;//定义循环变量 i 和打开文件的 id 变量
        int letter=1;
        char filename[128];
        for(i=0;i<128,letter!='\0';i++)
        {
            kernel->machine->ReadMem(address+i,1,&letter);
            filename[i]=(char)letter;
        }//同样从内存中一个个字母读出文件名
        OpenFile* openfile= kernel->fileSystem->Open(filename);
        /*调用系统内核预设好的 kernel->fileSystem 类中的 Open(char *filename)函数，打开
成功时返回值为 OpenFile*类型为文件指针，失败时为 NULL*/
        if (openfile == NULL) return -1;//打开失败时判断 openfile 为 NULL，则返回-1
        else
        {
            id = openfile->Getid();
            //调用 OpenFile 类的 Getid()函数得到打开文件的 id，Getid()定义见下
            return id;//打开文件成功时返回文件 id
        }
    }
```

c）补充函数：在 code/filesys/openfile.h 中加入 Getid()函数

```
class OpenFile {
  public:
    ……
    ……
    int Getid() {return file;}//此处为添加，直接返回 file 即为文件 id
  private:
    int file;
    int currentOffset;
};
```

3）Write 系统调用

a）修改中断入口处理函数：（省略 PC 指针后移及其后的重复段内容）

```
    case SC_Write:
        DEBUG(dbgSys, "Write" << kernel->machine->ReadRegister(4) <<
kernel->machine->ReadRegister(5) << kernel->machine->ReadRegister(6) << "\n");
        //由于 Write 调用函数有三个变量所以需要从三个寄存器读入
        int buf, w_size, result_write;
        //定义变量所需写入内容的内存地址 buf，写入字符数 w_size 和返回值 result_write
        OpenFileId w_id;
        //定义变量文件 Id 为 w_id
        buf = (int)kernel->machine->ReadRegister(4);
```

```
        w_size = (int)kernel->machine->ReadRegister(5);
        w_id = (int)kernel->machine->ReadRegister(6);
        //分别从$4、$5、$6 中读入三个变量值
        result_write = SysWrite(buf,w_size,w_id);
        //调用 SysWrite 完成对文件的写入并返回写入字符数
        kernel->machine->WriteRegister(2, result_write);
```

b）在内核中实现系统调用函数：

```
int SysWrite(int buf, int size,OpenFileId id)
{
    OpenFile* openfile =new OpenFile(id);
    //调用 new 函数生成编号 Id 为 id 的 OpenFile 类
    char buffer[256];//定义 buffer 字符数组用来存储需要写入文件的内容
    int i;
    int letter=1;
    for(i=0;i<size,letter!='\0';i++)
    {
        kernel->machine->ReadMem(buf+i,1,&letter);
        buffer[i]=(char)letter;
    }
    //从内存 buf 里读入要写入文件的字符串存入 buffer 中
    if(openfile->Write(buffer,size)) return i-1;
    else return -1;
    /*调用 openfile 类里的预设接口函数 Write(char *from, int numBytes)，Write 函数返回写入的字符数，不为 0 时代表写入成功，返回 i-1 即代表写入的字符数；返回值为 0 时代表写入失败，返回-1。*/
}
```

c）补充修改函数：在 code/filesys/openfile.h 中修改 Write 函数

```
int Write(char *from, int numBytes)
{
    int currentOffset=Length();//设置当前偏移量为文件中字符长度
    int numWritten = WriteAt(from, numBytes, currentOffset); //从当前偏移量处追加写
    currentOffset += numWritten;
    return numWritten;
}
```

此修改将本来直接从头对文件的写入改为从文件末尾写入，从而符合实验要求。

4）Read 系统调用

a）修改中断入口处理函数：（省略 PC 指针后移及其后的重复段内容）

```
case SC_Read:
    DEBUG(dbgSys, "Read" << kernel->machine->ReadRegister(4) <<
kernel->machine->ReadRegister(5) << kernel->machine->ReadRegister(6) << "\n");
```

```
        int r_buf, r_size, result_read;
        //定义变量所需读出内容的内存地址 r_buf，读入字符数 r_size 和返回值 result_read
        OpenFileId r_id;
        //定义变量文件 Id 为 r_id
        r_buf = (int)kernel->machine->ReadRegister(4);
        r_size = (int)kernel->machine->ReadRegister(5);
        r_id = (int)kernel->machine->ReadRegister(6);
        result_read = SysRead(r_buf,r_size,r_id);
        //从三个寄存器中读出三个变量，并代入 SysRead 中
        kernel->machine->WriteRegister(2, result_read);
```

b）在内核中实现系统调用函数：

```
int SysRead(int buf, int size,OpenFileId id)
{
        int i;
        OpenFile* openfile =new OpenFile(id);
        char filecontent[size]; //定义 filecontent 字符数组用来存储所读取的文件内容
        int readnumber=0;//定义 readnumber 用以存储读取字符的长度
        readnumber=openfile->Read(filecontent,size);
        /*调用 openfile 类接口函数 Read(char *into, int numBytes)，用来从 openfile 中读取
size 个字符存储进字符数组 filecontent 中，返回值为读入的字符数*/
        filecontent[size]= '\0';//将 filecontent 字符串的最后一位标为“\0”
        for(i = 0;i<size;i++)
        if (!(kernel->machine->WriteMem(buf+i,1,(int)filecontent[i]))) return -1;
        //调用 WriteMem 将 filecontent 串中的字符一位位存入内存 buf 中，失败时返回-1
        return readnumber;//返回读取的字符数
}
```

5）Close 系统调用

a）修改中断入口处理函数：

```
case SC_Close://基本与 Create 相同
        DEBUG(dbgSys, "Close" << kernel->machine->ReadRegister(4) << "\n");
        int id_close,result_close;
        id_close = (int)kernel->machine->ReadRegister(4);
        result_close = SysClose(id_close);
        kernel->machine->WriteRegister(2, result_close);
```

b）在内核中实现系统调用函数：

```
int SysClose(int id)
{
        if(Close(id)==0) return 1;//调用 Close 函数，关闭成功 Close 返回 0，SysClose 返回 1
        else return -1;//关闭失败 Close 返回非 0，SysClose 返回-1
}
```

（4）、编写用户测试程序

根据教学平台提供的 syscallTest.c 测试程序，在 code/test/Makefile 中添加：

```
PROGRAMS = add halt shell matmult sort segments sub
syscallTest.o: syscallTest.c
    $(CC) $(CFLAGS) -c syscallTest.c
syscallTest: syscallTest.o start.o
    $(LD) $(LDFLAGS) start.o syscallTest.o -o syscallTest.coff
    $(COFF2NOFF) syscallTest.coff syscallTest
```

至此完成全部 Nachos 五大基础系统调用的编写和测试准备。

3. 实验结果

（1）实验运行结果截图

```
tkc@ubuntu:~/nachos/NachOS-4.0/code/test$ ../build.linux/nachos -x syscallTest
Machine halting!

Ticks: total 1516, idle 0, system 170, user 1346
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
tkc@ubuntu:~/nachos/NachOS-4.0/code/test$ cat test.txt
SysCall Test for Nachos!
tkc@ubuntu:~/nachos/NachOS-4.0/code/test$ cat result.txt
1
6
7
-1
25
1
7
SysCall
```

（2）运行结果分析

根据测试文件分析，运行结果由以下步骤构成：

A. 创建 test.txt 和 result.txt 两个文件

B. 成功创建 test.txt 文件，返回值为 1 赋值给 result

C. 打开存在的文件 test.txt，打开成功，返回值为文件 id=6，赋值给 fid1

D. 打开存在的文件 result.txt，打开成功，返回值为文件 id=7，赋值给 fid2

E. 打开不存在的文件，打开失败，返回值为-1，赋值给 fid3

F. 将 result、fid1、fid2、fid3 写入 result.txt 文件中

G. 向 test.txt 文件(fid1)写入字符串“SysCall Test for Nachos!\n”，返回值为写入字符串的长度 25，并将返回值写入 result.txt 文件中

H. 关闭文件 test.txt，成功后返回值为 1，写入 result.txt 文件中

I. 从 test.txt 文件读取指定长度为 7 的数据，返回值为 7，写入 result.txt 文件中

J. 将读出的字符串“SysCall”写入 result.txt 中

即得到如结果截图的运行结果。

4. 实验错误与分析

（1）对接口函数的理解错误

最初添加系统调用的时候以为只要在进入中断后直接调用接口函数即可，如在最初的SysCreate 中，直接写成了：

```
int SysCreate(char * filename)
{
    if(kernel->fileSystem->Create(filename)) return 1;
    else return -1;
}
```

其中 filename 直接从寄存器中读入，后来查阅资料才知道寄存器中最初存储的应该是文件名的地址，文件名应该再从存储器中的该地址读取。于是考虑使用函数 kernel->machine 类的函数 ReadMem(char*,int,char*)，但由于文件名的长度不知道，所以难以一次性从存储器内读入整个文件名。思考后最终使用 for 循环一个个读入文件名，每次循环后判断本次读入的字符是否为‘\0’，若是则停止循环，另外如果文件名长度达到字符串上限也会跳出循环。如此便可以读入文件名并实现对接口函数 Create(char *filename)的调用。

在对另外几个接口函数的调用初期也存在同样的问题，经过长时间的尝试，终于明白了使用接口的基本原理：先从寄存器内读入所需的地址，再从对应地址的存储器内读入文件名和字符串等各种信息，最后调用接口函数完成对应系统调用，不同的系统调用编写的区别主要在于对存储器内不同信息的读取。

（2）Open 系统调用的返回值问题

实验要求完成的 Open 系统调用需在打开文件成功时返回该文件在进程中的 file id，可本来系统预设的接口函数 Open 的返回值为打开文件的地址，无法找到文件的 file id。最终通过查找 openfile 类函数，发现只要返回该类中的 int file 的值即可，故在该类中添加：

```
int Getid() {return file;}
```

即可调用 openfile->Getid 函数取得所打开文件的 file id。

（3）Close 系统调用的特殊性

在编写 Close 系统调用时，起初并未在 filesys/filesys.h 和 filesys/openfile.h 中找到对应的接口函数 Close。后来发现在这两个文件中均直接调用过 Close 函数，如以下语句：

```
~OpenFile() { Close(file); }
```

反向推出 Close 函数应该为 Close(int id)，返回值根据测试：在关闭成功时为 False，故在 SysClose 中判断 Close 的返回值等于 0 时返回 1，才使得程序符合实验要求。

（4）各种编译小错误

由于整体编写中各种接口函数调用函数都有大量变量，故出现了不少变量书写或代入错误，因为在每个调用中断处都加入了系统自带的 DEBUG 函数，方便确认 bug 的具体位置并加以修正。另外在整个 switch 语句中，起初所有的 result 均定义成一个，导致很多输出错误，最后将所有 case 语句中的变量修改为互不相同的名字，才解决该问题。

5. 实验总结与收获

本次实验为操作系统课程的第一次实验，也算是第一次走近操作系统的源代码层次。Nachos 系统整体还是比较友好的，其主要采用 C++语言书写，因此大部分地方的理解都不难，只有一些很复杂的类调用函数在起初的理解时给我带来了一定的困难。在尝试完成了第一个系统调用后感觉后面的就都简单了很多，我对这种通过多接口多文件互相调用的程序编

写还是第一次体验，虽然实际的程序编写量很小不过还是收获了很多和以前编写 C 语言程序不一样的体验，也算是真正认识了简单操作系统的内部。

希望在后面的实验中也能通过自己的一步步尝试完成，也希望自己能有对操作系统更深层次的理解！

参考文献:

[1] Abraham Silberschatz. 操作系统概念. 高等教育出版社, 2007.3.
[2] Nachos-3.4 系统调用 实现 Create Open Read Write Close
https://blog.csdn.net/small_snowflake/article/details/51407554
[3] Nachos 系统调用实习报告
https://max.book118.com/html/2015/0521/17426315.shtm