系统调用实习报告

姓名 马少鹏　　学号 1500012893

日期 2018.5.15

目录

内容一：总体概述

本实习希望通过修改 Nachos 系统平台的底层源代码，达到“实现虚拟存储系统”的目标。

内容二：任务完成情况

任务完成列表（Y/N）

Exercise1	Exercise2	Exercise3	Exercise4
Y	Y	Y	Y

具体 Exercise 的完成情况

第一部分 理解 Nachos 系统调用

Exercise1 源代码阅读

code/userprog/syscall.h

定义了各个系统调用的系统调用号，并声明了系统调用的函数。

code/userprog/exception.cc

定义了异常处理函数 ExceptionHandler，也就是系统调用的入口处理函数。根据储存在 2 号寄存器中的系统调用号，判断是哪个系统调用，然后做相应的处理。

值得注意的是，异常处理程序结束之后，需要把 PC 向前调整，否则异常处理程序结束之后仍旧返回到系统调用这条指令，无限循环。故在 machine 类中实现成员函数 advance_pc：

```
194  // lab5
195      void advance_pc()
196      {
197          WriteRegister(PrevPCReg, ReadRegister(PCReg));
198          WriteRegister(PCReg, ReadRegister(PCReg)+sizeof(int));
199          WriteRegister(PCReg, ReadRegister(NextPCReg)+sizeof(int));
200      }
```

在每个异常处理程序的结尾使用这个函数。

code/test/start.s

描述了具体的系统调用过程。把系统调用号存入 2 号寄存器，然后执行 syscall。

第二部分 文件系统相关的系统调用

Exercise2 系统调用实现 Create Open Close Write Read

Create

从 4 号寄存器读入文件名，使用 FileSystem 类的 Create 创建文件即可。

测试：修改在 test 文件夹下的 halt.c：执行 Create(test_file)

```
msp@msp-virtual-machine:~/nachos_dianti/nachos-3.4/code/userprog$ ./nachos -x ../test/halt
create file test_file
Machine halting!
```

成功创建，并且打开 userprog 文件夹，确实有这个文件。

Open

从四号寄存器读入文件名，然后使用 FileSystem 类的 Open 函数。注意要把 OpenFile 类中的文件描述符 file 写到 2 号寄存器，作为 Open 系统调用的返回值。

测试：打开创建的 test_file 文件：

```
msp@msp-virtual-machine:~/nachos_dianti/nachos-3.4/code/userprog$ ./nachos -x ../test/halt
open file name: test_file  id: 3
Machine halting!
```

Close

先从 4 号寄存器读取文件描述符，然后直接使用 sysdep.cc 中的 close 即可。

测试：

```
33      tmp = Open("test_file");
34      Close(tmp);
```

先打开文件。再关闭。

```
msp@msp-virtual-machine:~/nachos_dianti/nachos-3.4/code/userprog$ ./nachos -x ../test/halt
open file name: test_file  id: 3
close file id:3
Machine halting!
```

Write

从 4 号寄存器读取写入内容的起始地址。从 5 号寄存器读取写入内容的大小。从 6 号寄存器读取要写入文件的文件描述符。然后使用 OpenFile 类的 Write 方法。代码如下：

```
131     else if((which == SyscallException) && (type == SC_Write))
132     {
133         int base = machine->ReadRegister(4);
134         int size = machine->ReadRegister(5);
135         int file = machine->ReadRegister(6);
136         int value;
137         int length =0;
138         do
139         {
140             machine->ReadMem(base++, 1, &value);
141             length++;
142         }while(value!=0);
143         base -= length;
144         char content[1000];
145 //printf("length: %d\n",length);
146         int i = 0;
147         for(i=0;i<length-1;++i)
148         {
149 //printf("i: %d\n",i);
150             machine->ReadMem(base+i, 1, &value);
151             content[i] = (char)value;
152 //printf("value: %c\n",content[i]);
153         }
154         //content[i] = '\n';
155 printf("write file id: %d    write content: %s\n",file,content);
156         OpenFile * o = new OpenFile(file);
157         o->Write(content,size);
158         delete o;
159 
160         machine->advance_pc();
```

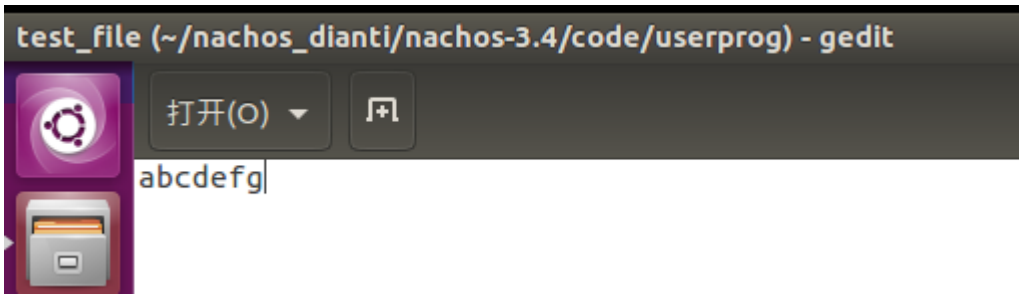
测试：

```
33      tmp = Open("test_file");
34      Write("abcdefg", 7, tmp);
35      Close(tmp);
```

结果：

```
msp@msp-virtual-machine:~/nachos_dianti/nachos-3.4/code/userprog$ ./nachos -x ../test/halt
open file name: test_file  id: 3
write file id: 3    write content: abcdefg
Machine halting!
```

打开 test_file 文件，可以发现确实写入了内容：



Read

从 4 号寄存器读取要读入的 buffer 起始地址。从 5 号寄存器读入要读的内容的大小。从 6 号寄存器读取要读的文件的文件描述符。然后使用 OpenFile 类的 Read 方法，并将返回值 read_num 写到 2 号寄存器作为系统调用 Read 的返回值。代码如下：

```
162     else if((which == SyscallException) && (type == SC_Read))
163     {
164         int base = machine->ReadRegister(4);
165         int size = machine->ReadRegister(5);
166         int file = machine->ReadRegister(6);
167 
168         OpenFile *o = new OpenFile(file);
169         char tmp[1000];
170         int read_num = 0;
171         read_num = o->Read(tmp, size);
172         int i = 0;
173         for(i=0;i<size;++i)
174         {
175             machine->WriteMem(base, 1, tmp[i]);
176         }
177         tmp[i] = '\0';
178 printf("read_num: %d  content: %s\n",read_num,tmp);
179         machine->WriteRegister(2,read_num);
180         delete o;
181         machine->advance_pc();
182     }
```

测试：

```
34      tmp = Open("test_file");
35      read_num = Read(buffer, 3, tmp);
```

从之前创建并写入的文件中读取三个字符：

```
msp@msp-virtual-machine:~/nachos_dianti/nachos-3.4/code/userprog$ ./nachos -x ../test/halt
open file name: test_file  id: 3
read_num: 3  content: abc
Machine halting!
```

发现成功读取’abc’。

第三部分 执行用户程序相关的系统调用

Exercise3 系统调用实现 Exec Fork Yield Join Exit

Exec

从 4 号寄存去读取要执行的用户文件名的起始地址。然后使用 ReadMem 读取一次得到文件名长度，然后读取这个文件名。使用 OpenFile 类的 Open 方法打开这个文件，新建一个地址空间，将当前线程的地址空间修改为新的地址空间，并调用 InitRegisters 和 RestoreState 初始化寄存器，保存状态。然后使用 Machine 类的 Run 函数执行用户程序。总体上仿照之前内存管理 lab 的测试中的 StartProcess 函数写即可。代码如下：

```
183     else if((which == SyscallException) && (type == SC_Exec))
184     {
185         int base = machine->ReadRegister(4);
186         int value;
187         int length = 0;
188         do
189         {
190             machine->ReadMem(base++, 1, &value);
191             length++;
192         }while(value != 0);
193         base -= length;
194         char filename[100];
195         for(int i=0;i<length;++i)
196         {
197             machine->ReadMem(base+i, 1, &value);
198             filename[i] = (char)value;
199         }
200         OpenFile *o = fileSystem->Open(filename);
201         AddrSpace *space;
202         if(o == NULL)
203         {
204 printf("can not find file %s\n",filename);
205         machine->advance_pc();
206         return;
207         }
208         space = new AddrSpace(o);
209         currentThread->space = space;
210         currentThread->filename = filename;
211         delete o;
212         space->InitRegisters();
213         space->RestoreState();
214 //printf("file : %s\n",filename);
215         machine->Run();
216 
217         machine->advance_pc();
218     }
```

测试：Exec(“matmult”)。该用户程序是矩阵乘法，在之前的 lab 中有用到，结果是 80.

```
msp@msp-virtual-machine:~/nachos_dianti/nachos-3.4/code/userprog$ ./nachos -x ../test/halt
now run user program: matmult
Exec_result: 80
thread main(tid:0) finish
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

可以看到执行了矩阵乘法，且结果正确。

Fork

从 4 号寄存器读取要执行的函数的起始地址。新建一个线程，并初始化新线程的地址空间为当前线程的地址空间。然后执行 Thread 类的 Fork 方法，第一个参数是一个函数 fork_func。函数代码如下：

```
275   void
276   fork_func(int pc)
277   {
278       machine->WriteRegister(PCReg,pc);
279       machine->WriteRegister(NextPCReg,pc+4);
280       machine->Run();
281   }
```

该函数的参数是之前得到的要执行的函数的起始地址。函数的功能是将 PC 寄存器的值改为要执行的函数的起始地址，然后再执行 Machine 类的 Run，就可以在子线程中执行该函数了。

Fork 的处理程序：

```
219       else if((which == SyscallException) && (type == SC_Fork))
220       {
221           int pc = machine->ReadRegister(4);
222           AddrSpace *space = currentThread->space;
223           Thread *t = new Thread("fork_thread");
224           t->space = space;
225   //printf("fork thread id: %d\n",t->get_thread_id());
226           t->Fork(fork_func, pc);
227           machine->advance_pc();
228       }
```

测试：主线程 Fork 一个子线程执行函数 func，而函数 func 的功能是创建一个名为”msp”的文件。

```
msp@msp-virtual-machine:~/nachos_dianti/nachos-3.4/code/userprog$ ./nachos -x ../test/halt
fork halt(tid: 1)
fork fork_thread(tid: 2)
thread (main, tid:0 priority: 128) yield to thread (fork_thread, tid: 2 priority: 129)
create file msp
Machine halting!
```

使用了之前实现的基于优先级的抢占式调度算法，并设置子线程优先级高于主线程，结果符合预期，且打开文件夹发现文件创建成功。

Yield

直接使用 Thread 类的 Yield 方法即可。

测试：先 Fork 一个线程，然后主线程执行 Yield。不使用基于优先级的抢占式调度算法。

```
msp@msp-virtual-machine:~/nachos_dianti/nachos-3.4/code/userprog$ ./nachos -x ../test/halt
fork fork_thread(tid: 1  priority: 1)
syscall yield
thread (main, tid:0 priority: 128) yield to thread (fork_thread, tid: 1 priority: 1)
create file msp
thread fork_thread(tid:1) finish
thread main(tid:0) finish
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

主线程 fork 了子线程之后继续执行，直到系统调用 Yield，才转换为子线程执行并创建了文件 msp。结果符合预期。

Join

在 Thread 类中增加成员变量 List *child。记录线程的子线程集合。增加 Thread *father。记录父线程。

在 Thread 类中增加成员函数 join：

```
487  void
488  Thread::join(int tid)
489  {
490      if(whole_thread[tid].Thread_id)
491      {
492  //printf("thread %s(tid: %d) join thread %s(tid: %d)\n
493  //whole_thread[tid].thread->get_thread_id());
494          whole_thread[tid].thread->father = this;
495          child->Append(whole_thread[tid].thread);
496      }
497  }
```

功能是将当前线程作为目标线程的父线程。其中储存所有线程信息的数据结构在第一次 lab 中已经实现。

在此基础上，Join 的处理程序非常简单：从 4 号寄存器读取要 join 的目标线程的 tid，然后直接使用 Thread 类的 join 方法，然后判断该子线程是否结束，如果没有则 yield。

```
241         if(whole_thread[arg].Thread_id == 1 && whole_thread[arg].thread->father == currentThread)
242         {
243 printf("thread %d wait thread %d\n",currentThread->get_thread_id(),arg);
244             currentThread->Yield();
245         }
```

测试：主线程 Fork 一个新线程，然后 join 新线程。

```
msp@msp-virtual-machine:~/nachos_dianti/nachos-3.4/code/userprog$ ./nachos -x ../test/halt
fork fork_thread(tid: 1  priority: 0)
thread 0 wait thread 1
thread (main, tid:0 priority: 128) yield to thread (fork_thread, tid: 1 priority: 0)
create file msp
thread fork_thread(tid:1) finish
thread main(tid:0) finish
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

可以看到，主线程 join 了线程 1，然后发现新线程没有结束，等待子线程结束之后才继

续执行。结果符合预期。

Exit

由于之前 lab 的需要，Exit 在内存管理时已经实现过：直接使用 Thread 类的 Finish 方法即可。从 4 号寄存器读取 Exit 的 status。

测试：直接执行 Exit(123)。

```
msp@msp-virtual-machine:~/nachos_dianti/nachos-3.4/code/userprog$ ./nachos -x ../test/halt
Exit_status: 123
thread main(tid:0) finish
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

内容三：收获及感想

更加深刻的理解了系统调用的原理和作用。

内容四：对课程的意见和建议

目前对课程非常满意，没有意见。

内容五：参考文献

[1] Andrew S. Tanenbaum 著．陈向群 马洪兵 译 .现代操作系统［M］. 北京：机械工业出版社，2011：47-95

[2] 和队友刘雨曦同学的讨论