

Lab6 系统调用实 习报告

姓名 姜慧强 学号 1801210840
日期 2019.06.02

目录

内容一：总体概述	3
内容二：任务完成情况	3
任务完成列表（Y/N）	3
具体 Exercise 的完成情况	3
内容三：遇到的困难以及解决方法	14
内容四：收获及感想	14
内容五：对课程的意见和建议	14
内容六：参考文献	14

内容一：总体概述

本次实验是操作系统高级课程的第六次实验。在 Lab5 阅读文件系统机制，完成文件扩充，间接索引机制实现基础上，主要完成了 Nachos 系统调用相关代码的实践。本次实验完成了 nachos 系统调用原理的阅读，实现相关系统调用正确性的验证。

内容二：任务完成情况

任务完成列表 (Y/N)

EXER1	EXER2	EXER3	EXER4	EXER5
Y	Y	Y	Y	Y

具体 Exercise 的完成情况

一、理解 Nachos 系统调用

Exercise1

Userprog/syscall.h 定义了 nachos 的系统调用，包括系统调用号，和系统调用函数。Nachos 内核通过识别用户程序传递的系统调用号确定调用类型。已经实现的系统调用包括。

```
void Halt();
void Exit(int status);
SpaceId Exec(char *name);
void Create(char *name);
OpenFileId Open(char *name);
void Write(char *buffer, int size, OpenFileId id);
int Read(char *buffer, int size, OpenFileId id);
void Close(OpenFileId id);
void Fork(void (*func)());
void Yield();
```

这些函数都是我们平时在写代码时候用的比较多的函数，比如说 **Halt** 是用来中止用户进程的。**Create** 是用来创建文件，然后还包括一些读写调用，打开关闭文件，**Yield**, **Fork**。

Exception.cc 文件定义了异常处理相关代码。目前支持的异常包括。

```
enum ExceptionType { NoException,          // Everything ok!
                    SyscallException,      // A program executed a system call.
```

```

    PageFaultException,    // No valid translation found
    ReadOnlyException,     // Write attempted to page marked
                          // "read-only"
    BusErrorException,     // Translation resulted in an
                          // invalid physical address
    AddressErrorException, // Unaligned reference or one that
                          // was beyond the end of the
                          // address space
    OverflowException,     // Integer overflow in add or sub.
    IllegalInstrException, // Unimplemented or reserved instr.

    NumExceptionTypes
};

```

处理系统调用的时候，通过 2 号寄存器来传递系统调用类型。

Code/test/start.s 中存储的是辅助用户程序运行的汇编代码。包括初始化用户程序和一些系统调用操作的实现。

```

.globl __start
.ent __start

```

来初始化用户程序，通过 main 函数运行用户程序。

```

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

```

通过跳转至 main 函数执行用户程序。

```

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

```

系统调用相关，通过把系统调用号放在 2 号寄存器，来实现系统调用。

Exercise2

类比 Halt，完成 Create, Open, Close, Write 的实现。

① Create

通过寄存器 r4 来获取文件名指针，然后通过文件名指针通过 ReadMe 获取文件名。再通过 Create 函数创建文件，最后更新 PC

```

void CreateSyscallHandler() {
    currentThread->SaveUserState();
}

```

```

// First, get the length of filename
int fileNameBase = machine->ReadRegister(4);
int value;
int length = 0;
do {
    machine->ReadMem(fileNameBase++, 1, &value);
    length++;
} while(value != '\0');

// Copy filename
char *fileName = new char[length];
fileNameBase -= length; length--;
for(int i = 0; i < length; i++) {
    machine->ReadMem(fileNameBase++, 1, &value);
    fileName[i] = char(value);
}
fileName[length] = '\0';
DEBUG('a', "File name: %s\n", fileName);

bool result = fileSystem->Create(fileName, 0);

if(result)
    DEBUG('a', "Create file %s done\n", fileName);
else
    DEBUG('a', "Can not create file %s\n", fileName);
delete fileName;
currentThread->RestoreUserState();
}

```

② Open

Open 操作也是相似的思路，通过 **r4** 获得文件名指针，再去获取文件名，最后调用 **Open** 函数更新 **PC** 值。

```

void OpenSyscallHandler() {
currentThread->SaveUserState();
// First, get the length of filename
int fileNameBase = machine->ReadRegister(4);
int value;
int length = 0;
do {
    machine->ReadMem(fileNameBase++, 1, &value);
    length++;
} while(value != '\0');

// Copy filename

```

```

char *fileName = new char[length];
fileNameBase -= length; length--;
for(int i = 0; i < length; i++) {
    machine->ReadMem(fileNameBase++, 1, &value);
    fileName[i] = char(value);
}
fileName[length] = '\0';
DEBUG('a', "File name: %s\n", fileName);

OpenFile *openFile = fileSystem->Open(fileName);

if(openFile != NULL)
    DEBUG('a', "Open file %s done\n", fileName);
else
    DEBUG('a', "Can not open file %s\n", fileName);

currentThread->RestoreUserState();
machine->WriteRegister(2, (int)openFile);
}

```

③ Close

Close 与上面的系统调用基本相同，先通过 **r4** 寄存器获取文件数据结构，然后打开文件数据结构解析函数，最后更新 **PC**。

```

void CloseSyscallHandler() {
    currentThread->SaveUserState();
    OpenFile *openFile = (OpenFile *)machine->ReadRegister(4);
    DEBUG('a', "Close File\n");
    delete openFile;
    currentThread->RestoreUserState();
    machine->WriteRegister(2, 0);
}

```

④ Write

Write 通过 **r4** 指针获得缓冲区指针，通过寄存器 **r5** 获得所需要写入的文件长度，通过 **r6** 打开文件数据结构。然后利用 **ReadMe** 获取缓存区数据，利用 **Write** 向缓存区写入数据，最后再更新 **PC** 指针。

```

void WriteSyscallHandler() {
    currentThread->SaveUserState();
    int buffer = machine->ReadRegister(4);
    int size = machine->ReadRegister(5);
    OpenFile *openFile = (OpenFile *)machine->ReadRegister(6);
    char *kernelBuffer = new char[size + 1];
    int value, i;
    for(i = 0; i < size; i++) {

```

```

        bool success = machine->ReadMem(buffer++, 1, &value);
        if(!success) {
            buffer--; i--;
            continue;
        }
        kernelBuffer[i] = char(value);
    }
    kernelBuffer[i] = '\0';

    // Write into file
    int result = openFile->Write(kernelBuffer, size);
    delete kernelBuffer;
    currentThread->RestoreUserState();
    machine->WriteRegister(2, result);
}

```

⑤ Read

Read 操作和 Write 操作基本相同，通过 r4 获取缓存区指针，r5 获取数据长度，r6 获取文件数据结构。利用 Read 函数读取文件内容，获得实际长度，将获取的实际内容写入缓存区，返回实际读入字节数给寄存器 r2，PC+4.

```

void ReadSyscallHandler() {
    currentThread->SaveUserState();
    int buffer = machine->ReadRegister(4);
    int size = machine->ReadRegister(5);
    OpenFile *openFile = (OpenFile *)machine->ReadRegister(6);

    char *kernelBuffer = new char[size];

    // Read from file into kernel space
    int result = openFile->Read(kernelBuffer, size);

    // Write into user space
    for(int i = 0; i < result; i++) {
        bool success = machine->WriteMem(buffer++, 1, (int)kernelBuffer[i]);
        if(!success) {
            buffer--; i--;
        }
    }
}

```

```

delete kernelBuffer;
currentThread->RestoreUserState();
machine->WriteRegister(2, result);
}

```

Exercise3

依次完成文件 Create, Open, Read, Write, Close。

```

#include "syscall.h"
int main() {
    char a[6], b[6];
    OpenFileId fd1;
    OpenFileId fd2;
    a[0] = 'a';
    a[1] = '.';
    a[2] = 't';
    a[3] = 'x';
    a[4] = 't';
    a[5] = '\0';
    Create(a);
    fd1 = Open(a);
    fd2 = Open(a);
    Write(a, 5, fd1);
    Read(b, 5, fd2);
    Close(fd1);
    Close(fd2);
    Exit('a' - b[0]);
}

```



```
1. root@1801210840: ~/nachos/nachos_diant/lab5/nachos_diant/nachos-3.4/code/userprog (docker)
../userprog/exception.cc:177:50: warning: deprecated conversion from string constant to 'char*' [-Wwrite-strings]
  DEBUG('f', "\033[91m Syscall: Exit\n");
  ^
../userprog/exception.cc:183:59: warning: deprecated conversion from string constant to 'char*' [-Wwrite-strings]
  DEBUG('f', "\033[92m Syscall: Create\n\033[0m");
  ^
../userprog/exception.cc:187:57: warning: deprecated conversion from string constant to 'char*' [-Wwrite-strings]
  DEBUG('f', "\033[92m Syscall: Open\n\033[0m");
  ^
../userprog/exception.cc:191:58: warning: deprecated conversion from string constant to 'char*' [-Wwrite-strings]
  DEBUG('f', "\033[92m Syscall: Close\n\033[0m");
  ^
../userprog/exception.cc:195:58: warning: deprecated conversion from string constant to 'char*' [-Wwrite-strings]
  DEBUG('f', "\033[93m Syscall: Write\n\033[0m");
  ^
../userprog/exception.cc:199:57: warning: deprecated conversion from string constant to 'char*' [-Wwrite-strings]
  DEBUG('f', "\033[93m Syscall: Read\n\033[0m");
  ^
g++ main.o list.o scheduler.o synch.o synchlist.o system.o thread.o utility.o threadtest.o interrupt.o stats.o sysdep.o timer.o elevator.o elevatortest.o add
rspace.o bitmap.o exception.o progtest.o console.o machine.o mipssim.o translate.o switch.o -o nachos
➔ 1801210840 userprog git:(develop) x ./nachos -x ./test/test -d f
The current date/time is: Mon Jun  3 14:01:51 2019

Syscall: Create
Syscall: Open
Syscall: Open
Syscall: Write
Syscall: Read
Syscall: Close
Syscall: Close
Syscall: Exit

Thread main finished with exit code 0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 89, idle 0, system 10, user 79
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...
➔ 1801210840 userprog git:(develop) x
```

Exercise 4

①Eexc

利用寄存器 `r4` 获得文件名指针，然后建立线程，然后执行用户程序。将线程 `ID` 返回给寄存器 `r2`，最后更新 `PC`。

具体来说，通过 `ReadMe` 把文件名指针转换为文件名。利用 `Open` 打开文件，利用 `AddrSpace` 构造地址空间，利用 `InitRegister` 初始化寄存器，利用 `RestoreState` 装载页表。然后在 `Run` 用户程序。

```
void ExecRoutine(int arg) {
    currentThread->space->InitRegisters();
    currentThread->space->RestoreState();
    Machine *p = (Machine *)arg;
    p->Run();
}

void ExecSyscallHandler() {
    currentThread->SaveUserState();
    // First, get the length of filename
    int fileNameBase = machine->ReadRegister(4);
    int value;
    int length = 0;
    do {
```

```

        machine->ReadMem(fileNameBase++, 1, &value);
        length++;
    } while(value != '\0');

    // Copy filename
    char *fileName = new char[length];
    fileNameBase -= length; length--;
    for(int i = 0; i < length; i++) {
        machine->ReadMem(fileNameBase++, 1, &value);
        fileName[i] = char(value);
    }
    fileName[length] = '\0';
    DEBUG('a', "Executable file name: %s\n", fileName);

    OpenFile *executable = fileSystem->Open(fileName);

    if(executable != NULL)
        DEBUG('a', "Open file %s done\n", fileName);
    else {
        DEBUG('a', "Can not open file %s\n", fileName);
        machine->WriteRegister(2, (int)executable);
        return;
    }

    // Create an address space and a new thread
    AddrSpace *addrSpace = new AddrSpace(executable);
    Thread *forked = new Thread(fileName);
    forked->space = addrSpace;

    // Run user program
    forked->Fork(ExecRoutine, (int)machine);
    DEBUG('t', "Exec done\n");
    currentThread->RestoreUserState();
    machine->WriteRegister(2, (int)addrSpace);
}

```

② Fork

同样利用寄存器 **r4** 获得函数位置，再复制当前的地址空间，然后在建立线程，设置地址空间，初始化寄存器，装载页表，设置 PC，然后 Run 程序，最后再更新 PC 值。

```

void ForkRoutine(int arg) {
    currentThread->space->RestoreState();
}

```

```

// Set PC to *arg*
machine->WriteRegister(PCReg, arg);
machine->WriteRegister(NextPCReg, arg + 4);
machine->Run();
}

void ForkSyscallHandler() {
    currentThread->SaveUserState(); // Save Registers
    int funcAddr = machine->ReadRegister(4);

    // Create a new thread in the same addressspace
    Thread *thread = new Thread("forked thread");
    thread->space = currentThread->space;
    thread->space->refNum++; // Increase RefNum
    // thread->RestoreUserState();
    thread->Fork(ForkRoutine, funcAddr);
    currentThread->RestoreUserState(); // Save Registers
}

```

③ Exit

同样 r4 获得退出状态，输出相关信息，clear 页表状态，更新 PC 值，结束进程。

```

else if(type == SC_Exit) {
    DEBUG('a', "Syscall: Exit\n");
    int exitCode = machine->ReadRegister(4);
    printf("\nThread %s finished with exit code %d\n\n",
currentThread->getName(), exitCode);
    currentThread->space->refNum--;
    DEBUG('a', "AddrSpace reference num: %d\n",
currentThread->space->refNum);
    if(currentThread->space->refNum == 0) {
        currentThread->space->Broadcast(exitCode);
    }
    currentThread->Finish();
}

```

Exercise 5

① EXEC

```

#include "syscall.h"
int main() {
    char a[5];
    int exitCode = -1;
    SpaceId sp;

    a[0] = 't';
    a[1] = 'e';
    a[2] = 's';
    a[3] = 't';
    a[4] = '\0';

    sp = Exec(a);

    exitCode = Join(sp);
    Exit(exitCode);
}

```

相应的改写./test 的 MakeFile
测试结果如下：

```

1. root@1801210840: ~/nachos/nachos_dianti/lab5/nachos_dianti/nachos-3.4/code/userprog (docker)
".data", filepos 0x220, mempos 0x150, size 0x0
".bss", filepos 0x0, mempos 0x150, size 0x0
.../gnu-decstation-ultrix/decstation-ultrix/2.95.3/gcc -B.../gnu-decstation-ultrix/ -G 0 -c -I.../userprog -I.../threads -c fork.c
.../gnu-decstation-ultrix/decstation-ultrix/2.95.3/ld -T script -N start.o fork.o -o fork.coff
.../bin/coff2nooff fork.coff fork
numsections 3
Loading 3 sections:
".text", filepos 0xd0, mempos 0x0, size 0x280
".data", filepos 0x350, mempos 0x280, size 0x0
".bss", filepos 0x0, mempos 0x280, size 0x0
→ 1801210840 test git:(develop) ✕ ./userprog
→ 1801210840 userprog git:(develop) ✕ ./nachos -x ../test/exec
The current date/time is: Mon Jun 3 14:25:08 2019

Thread main finished with exit code 3
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 49, idle 0, system 10, user 39
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...
→ 1801210840 userprog git:(develop) ✕ ./nachos -x ../test/exec -d f
The current date/time is: Mon Jun 3 14:25:12 2019

Syscall: Exec
Syscall: Exit

Thread main finished with exit code 3
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 49, idle 0, system 10, user 39
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...
→ 1801210840 userprog git:(develop) ✕

```

② Fork

```

#include "syscall.h"

```

```

void ThreadA() {
    char a[6], ch = 'a';
    int i;
    OpenFileId fd;
    a[0] = 'a';
    a[1] = '.';
    a[2] = 't';
    a[3] = 'x';
    a[4] = 't';
    a[5] = '\0';
    fd = Open(a);
    for(i = 0; i < 10; i++) {
        Write(&ch, 1, fd);
        Yield();
    }
}

```

```

void ThreadB() {
    char a[6], ch = 'b';
    int i;
    OpenFileId fd;
    a[0] = 'a';
    a[1] = '.';
    a[2] = 't';
    a[3] = 'x';
    a[4] = 't';
    a[5] = '\0';
    fd = Open(a);
    for(i = 0; i < 10; i++) {
        Write(&ch, 1, fd);
        Yield();
    }
}

```

```

int main() {
    Fork(ThreadA);
    Fork(ThreadB);
}

```

相应的改写./test 的 MakeFile
测试结果如下：

```
1. root@1801210840: ~/nachos/nachos_dianti/lab5/nachos_dianti/nachos-3.4/code/userprog (docker)
Network I/O: packets received 0, sent 0
Cleaning up...
→ 1801210840 userprog git:(develop) x ./nachos -x ../test/fork -d f
The current date/time is: Mon Jun 3 14:25:36 2019

Syscall: Fork
Syscall: Fork
Syscall: Exit

Thread main finished with exit code 0
Syscall: Open
Syscall: Write
Syscall: Yield
Interval Time is: 140 ms
NextPrio: 4; CurrentPrio: 4
Switch 2
Syscall: Open
Syscall: Write
Syscall: Yield
Interval Time is: 10 ms
NextPrio: 4; CurrentPrio: 4
Syscall: Write
Syscall: Yield
Interval Time is: 20 ms
NextPrio: 4; CurrentPrio: 4
Syscall: Write
Syscall: Yield
Interval Time is: 30 ms
NextPrio: 4; CurrentPrio: 4
Syscall: Write
Syscall: Yield
Interval Time is: 40 ms
NextPrio: 4; CurrentPrio: 4
Syscall: Write
Syscall: Yield
Interval Time is: 50 ms
NextPrio: 4; CurrentPrio: 4
Switch 2
Syscall: Write
Syscall: Yield
Interval Time is: 10 ms
NextPrio: 4; CurrentPrio: 4
Syscall: Write
Syscall: Yield
Interval Time is: 20 ms
```

内容三：遇到的困难以及解决方法

这次实验最大的困难是时间短，很多地方都是强行上手，完成报告的情况比较粗糙。总体而言，这一节的内容相较于前面的内容量上来说没有那么大。通过本次实验加深了自己对系统调用的认识。

内容四：收获及感想

本次实验是 Nachos 实验的最后一个单元，通过这次实验加深了自己对系统调用过程的认识。也加深了自己对动手能力。

内容五：对课程的意见和建议

内容六：参考文献

[1] Stevens, W. R. (2002). UNIX 环境高级编程：英文版．机械工业出版社．

