

# 系统调用实习报告

姓名： 学号：

日期： 2020/12/28

## 目录

内容一：任务完成情况.....	3
任务完成列表（Y/N） .....	3
具体 Exercise 完成情况 .....	3
<b>Exercise1</b> .....	3
<b>Exercise2</b> .....	9
<b>Exercise3</b> .....	16
<b>Exercise4</b> .....	29
<b>Exercise5</b> .....	40
内容二：遇到的困难以及收获.....	52
内容三：对课程或 Lab 的意见和建议.....	53
内容四：参考文献.....	53

# 内容一：任务完成情况

## 任务完成列表 (Y/N)

	Exercise1	Exercise2	Exercise3	Exercise4	Exercise5
第一部分	Y	Y	Y	Y	Y

## 具体 Exercise 完成情况

### Exercise1

这一部分要求阅读三份源代码，下面我将逐一分析各部分的具体内容。总的来说，这三份代码包含了 Nachos 对系统调用的实现。

#### (一) syscall.h

这份代码中主要包含了 Nachos 中支持的系统调用的声明。

#### 1. 系统调用号

Nachos 目前支持了 11 个系统调用，每个系统调用都有自己的系统调用号，如下图所示。

```
#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
```

#### 2. 系统调用及期望功能

上面的系统调用号对应的系统调用函数及期望功能依次为：

Halt：把系统停机，并输出 Nachos 的 performance。

#### 【控制地址空间情况的操作】

Exit：停止当前执行的用户程序，参数为停机状态，0 认为是正常的状态。

Exec：运行可执行文件，这个可执行文件储存在 Nachos 文件系统中，名称为参数中的

name，返回用户程序地址空间的唯一 id。

Join：只有在标号为参数 id 的用户级程序结束并退出时才返回，返回值为退出状态。

## 【文件系统操作】

Create：在 Nachos 的文件系统中创建一个名称为 name 的文件

Open：在 Nachos 的文件系统中打开名称为 name 的文件，并返回一个“OpenFileId”，作为可用于读写的文件描述符。

Write：向描述符为 id 的 Nachos 文件中写入 buffer 起始长为 size 的数据

Read：从描述符为 id 的 Nachos 文件中读取长为 size 的数据，并写入 buffer。

Close：关闭文件描述符为 id 的文件

## 【用户级的线程操作】

Fork：创建一个用户级线程，它会执行传入的函数。

Yield：让当前的线程主动让出 CPU。

上述所有系统调用都在#ifndef IN\_ASM 时才会被声明

## (二) Exception.cc

这份代码中包含的是 Nachos 的异常处理方法。核心函数为 ExceptionHandler。这份代码在 Lab4 中已经进行过分析，但 Lab4 过后我们对它进行了修改，现在再来看看一下。

这份代码中，核心的处理函数是 ExceptionHandler。当 Nachos 的 Machine 遇到异常时，便会调用这个函数来处理。函数首先根据参数判断异常类型。在 Lab4 完成后，我们首先判断是否为缺页异常，如果是的话，执行我们在 Lab4 中实现的处理程序（请查阅 Lab4 的实习报告）。

```
/* added by leesou 1800012826 Lab4 Exercise 2 */

    if (which == PageFaultException)
    {
#ifndef INVERTED_PAGETABLE
        if(machine->tlb == NULL) // PageTable error
        {
            DEBUG('m', "====> Page Table Fault.\n");
            ASSERT(FALSE);
        }
        else // TLB miss
        {
            DEBUG('m', "====> TLB miss arises!\n");
            int badVAddr = machine->ReadRegister(BadVAddrReg);
            TLBMissHandler(badVAddr);
        }
        return;
#else
        printf("Unexpected user mode exception %d %d\n", which, type);
        ASSERT(FALSE);
#endif
    }
```

接下来，函数判断是否为系统调用，如果是，函数根据系统调用号进行处理。系统调用号储存在下标为 2 的寄存器中。在完成 Lab4 后，目前只进行了对 Halt 和 Exit 的处理。

```
int type = machine->ReadRegister(2);

if (which == SyscallException)
{
    if(type == SC_Halt)
    {
        DEBUG('a', "Shutdown, initiated by user program.\n");
        TLBMissRate();
    #ifdef USER_PROGRAM
        if(currentThread->space != NULL)
        {
    #ifdef USE_BITMAP
            machine->freeMem();
    #endif
            delete currentThread->space;
            currentThread->space = NULL;
        }
    #endif
        interrupt->Halt();
    }
    else if(type == SC_Exit)
    {
        ExitHandler();
    }
}
```

如果系统调用是 Halt，那么函数会在释放用户级程序的地址空间和位图占用（如果使用了的话）后，调用 Interrupt 类中的 Halt 方法退出。这个函数会输出状态信息，并调用 Cleanup 释放所有使用的内存，正式退出整个 Nachos 系统。

```
void
Interrupt::Halt()
{
    if(VERBOSE)
    {
        printf("Machine halting!\n\n");
        stats->Print();
    }
    Cleanup();      // Never returns.
}
```

```
void
Cleanup()
{
    if(VERBOSE)
        printf("\nCleaning up...\n");
#ifndef NETWORK
    delete postOffice;
#endif

#ifndef USER_PROGRAM
    delete machine;
#endif

#ifndef FILESYS_NEEDED
    delete fileSystem;
#endif

#ifndef FILESYS
    delete synchDisk;
#endif

    delete timer;
    delete scheduler;
    delete interrupt;

    Exit(0);
}
```

如果系统调用是 Exit，那么函数调用我们先前实现的处理函数。对 Exit 的处理是，首先输出退出状态，接下来根据条件编译的情况释放占用的内存，最后调用当前线程的 Finish 方法来结束目前线程的运行，这样不会造成系统的停机。

```
void
ExitHandler()
{
    // deal with exit status
    int status = machine->ReadRegister(4); // get the first argument ----- exit status
    if(status == 0)
    {
        DEBUG('E', "*****User program exit normally*****\n");
    }
    else
    {
        DEBUG('E', "*****User program exit with status %d*****\n", status);
    }

    // release space
#ifndef USER_PROGRAM
    if(currentThread->space != NULL)
    {
#ifndef USE_BITMAP || INVERTED_PAGETABLE
        machine->freeMem();
#endif
        currentThread->space->SaveState(); // clear tlb when current thread exits
        delete currentThread->space;
        currentThread->space = NULL;
    }
#endif

#ifndef INVERTED_PAGETABLE
    printf("=====PAGE TABLE======\n");
    for(int i=0; i<NumPhysPages; ++i)
    {
        printf("%d\t%d\t%d\t%d\n", machine->pageTable[i].virtualPage, machine->pageTable[i].physicalPage, machine->pageTable[i].valid, machine->pageTable[i].TID);
    }
    printf("=====PAGE TABLE======\n");
#endif
#endif

    // finish this user thread
    currentThread->Finish();
}
```

对于其他系统调用，或者其他异常，目前的处理是直接使用 ASSERT 语句强行停止。

```
    else
    {
        printf("Unexpected user mode exception %d %d\n", which, type);
        ASSERT(FALSE);
    }
```

### (三) start.s

这份代码中包含了在 Nachos 运行用户级程序时要做的准备工作。

首先，可以看到，这一部分代码中声明了 IN\_ASM 后进行了 include “syscall.h”。根据上面的分析，这样只会引入 syscall.h 中的系统调用号，这些系统调用号也会在汇编语言中传入储存异常号的寄存器中（编号为 2），在异常处理时读入寄存器。

```
#define IN_ASM
#include "syscall.h"

.text
.align 2
```

接下来，代码声明了程序运行的入口：\_\_start。这个函数跳转到 main，如果 main 结束后未退出，函数会把储存第一个参数的寄存器清零，并跳转到 Exit 系统调用，这样程序便可以正常状态退出。

```

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

```

这份代码剩下的内容便是异常处理的汇编入口。所有的汇编函数结构相同，大致流程为：把系统调用号放入相应的寄存器——执行 syscall 指令，运行系统调用——跳转到返回地址寄存器中储存的地址来执行。11个系统调用的不同在于，传入 2 号寄存器的数值不同，为要执行的系统调用号。

<pre> .globl Halt .ent   Halt 3 references Halt: addiu \$2,\$0,SC_Halt syscall j    \$31 .end Halt </pre>	<pre> .globl Exit .ent   Exit 4 references Exit: addiu \$2,\$0,SC_Exit syscall j    \$31 .end Exit </pre>
<pre> .globl Exec .ent   Exec 3 references Exec: addiu \$2,\$0,SC_Exec syscall j    \$31 .end Exec </pre>	<pre> .globl Join .ent   Join 3 references Join: addiu \$2,\$0,SC_Join syscall j    \$31 .end Join </pre>
<pre> .globl Create .ent   Create 3 references Create: addiu \$2,\$0,SC_create syscall j    \$31 .end Create </pre>	<pre> .globl Open .ent   Open 3 references Open: addiu \$2,\$0,SC_Open syscall j    \$31 .end Open </pre>
<pre> .globl Read .ent   Read 3 references Read: addiu \$2,\$0,SC_Read syscall j    \$31 .end Read </pre>	<pre> .globl Write .ent   Write 3 references Write: addiu \$2,\$0,SC_Write syscall j    \$31 .end Write </pre>

<pre> .globl close .ent   Close 3 references close:     addiu \$2,\$0,SC_Close     syscall     j    \$31 .end Close </pre>	<pre> .globl Fork .ent   Fork 3 references Fork:     addiu \$2,\$0,SC_Fork     syscall     j    \$31 .end Fork </pre>
<pre> .globl Yield .ent   Yield 3 references Yield:     addiu \$2,\$0,SC_Yield     syscall     j    \$31 .end Yield </pre>	

最后，代码中定义了一个`_main`函数，用于满足gcc要求。

```

/* dummy function to keep gcc happy */
.globl __main
.ent  __main
3 references
__main:
    j     $31
.end __main

```

## (四) 小结：系统调用的主要流程

在先前的分析中，我们可以知道，Nachos运行用户级程序的方式为：Run函数中循环取指令，直到所有指令都运行结束，每条指令由OneInstruction这个函数来解析和运行。此外，还可以观察到，函数每执行完一条用户级程序的指令，都会调用OneTick让时钟前进。

```

void
Machine::Run()
{
    Instruction *instr = new Instruction; // storage for decoded

    if(Debug.IsEnabled('m'))
        printf("Starting thread \"%s\" at time %d\n",
               currentThread->getName(), stats->totalTicks);
    interrupt->setStatus(UserMode);
    for (;;) {
        OneInstruction(instr);
        interrupt->OneTick();
        if (singleStep && (runUntilTime <= stats->totalTicks))
            Debugger();
    }
}

```

而OneInstruction方法中，在解析指令时，**存在OP\_SYSCALL的分支，当遇到系统调**

用语句时，**OneInstruction** 会调用 **RaiseException** 函数，触发系统调用异常。

```
case OP_SYSCALL:  
    RaiseException(SyscallException, 0);
```

RaiseException 中，函数会设置寄存器状态，切换至内核模式，并调用异常处理函数。

```
void  
Machine::RaiseException(ExceptionType which, int badVAddr)  
{  
    DEBUG('m', "Exception: %s\n", exceptionNames[which]);  
  
    // ASSERT(interrupt->getStatus() == UserMode);  
    registers[BadVAddrReg] = badVAddr;  
    DelayedLoad(0, 0); // finish anything in progress  
    interrupt->setStatus(SystemMode);  
    ExceptionHandler(which); // interrupts are enabled at this point  
    interrupt->setStatus(UserMode);  
}
```

处理完毕后，函数切换回用户模式，设置新 PC，并继续执行之后的语句。

```
// Do any delayed load operation  
DelayedLoad(nextLoadReg, nextLoadValue);  
  
// Advance program counters.  
registers[PrevPCReg] = registers[PCReg]; // for debugging, in case we  
// are jumping into lala-land  
registers[PCReg] = registers[NextPCReg];  
registers[NextPCReg] = pcAfter;
```

此外，注意到，非退出/停机的系统调用执行后应该对 PC 进行递增操作。

### SYSCALL -- *System call*

Description:	Generates a software interrupt.
Operation:	advance_pc (4);
Syntax:	syscall
Encoding:	0000 00-- ---- ----- ----- --00 1100

最后，根据 MIPS 的手册，我们可以知道，系统调用的参数依次储存在 r4 ~ r7。

【注】本次 Lab 实现的系统调用基于 Lab4 和 Lab5 中实现的完整功能版的虚拟内存机制+文件系统，故有些需要前面宏的地方进行了省略，开启的宏选项如下

```
# DEFINES = -DTHREADS -DUSER_PROGRAM -DVM -DFILESYS_NEEDED -DFILESYS  
DEFINES = -DUSER_PROGRAM -DVM -DFILESYS_NEEDED -DFILESYS -DUSE_INDIRECT -DMULTI_LEVEL_DIR -DUSE_TLB -DUSE_BITMAP -DUSE_DISK
```

## Exercise2

这一部分要求我们实现与文件系统相关的系统调用。从上面对系统调用的实现的分析，我们只需要在 exception.cc 中完成相应的处理。

我们首先来处理文件的创建、打开、关闭。注意到，Create 和 Open 传入的都是指向名

称字符串的指针，因此，我们需要提供一个获取文件名的辅助函数，实现如下。大致思路为：从传入的起始地址处开始，逐字节读取文件名，直到第一次遇到'\0'，认为读取结束。这里，为了避免出现结尾内容不确定的问题，我在字符串的结尾赋值为'\0'。此外，由于在实现中我开启了 TLB 模式，所以在读取内存时，根据 Lab4 的实现，我给予了内存读取第二次机会，以应对 TLB miss 带来的读取失败。这里的 FileNameMaxLen 宏与 Lab5 完成后最终规定的长度相同。

```
#define FileNameMaxLen 85

char*
GetFileName(int address)
{
    int pos = 0;
    int data;
    char* fileName = new char[FileNameMaxLen+1];

    do
    {
        // the first time might meet TLB miss
        if(!machine->ReadMem(address+pos, 1, &data))
            ASSERT(machine->ReadMem(address+pos, 1, &data));
        fileName[pos++] = (char)data;
        ASSERT(pos <= FileNameMaxLen); // should not be too long
    } while (data != '\0');
    fileName[pos] = '\0';

    // printf("name is %s \n", fileName);
    return fileName;
}
```

此外，从功能上考虑，这几个系统调用执行结束后，应该执行下一条语句，但是 mipssim 中，执行完系统调用会直接返回，所以我们还需要添加 PC 递增的操作的辅助函数。

```
void
IncreasePC()
{
    int PC = machine->ReadRegister(PCReg);
    int NextPC = machine->ReadRegister(NextPCReg);

    machine->WriteRegister(PrevPCReg, PC);
    machine->WriteRegister(PCReg, NextPC);
    machine->WriteRegister(NextPCReg, NextPC+4);
}
```

对于 Create，我们在获取文件名后，调用文件系统的 Create 函数即可。由于先前我们实现了文件大小的自动扩展，我们只需要在创建时设置文件长度为 0。

```
void
CreateHandler()
{
    DEBUG('S', "Start to create new file\n");
    int startAddr = machine->ReadRegister(4);
    // printf("\naddr %x\n\n", startAddr);
    char* fileName = GetFileName(startAddr);
    // printf("\n%s filename\n\n", fileName);
    ASSERT(fileSystem->Create(fileName, 0));
    IncreasePC();
}
```

为了实现 `OpenFileId` 的机制，我在 `FileSystem` 类中添加了一个储存打开文件指针的数据组，用数组下标作为返回的 `OpenFileId`，其中的 `0` 和 `1` 作为保留的控制台输出标识符。调用 `Open` 时，首先检查数组中是否存在空余位置。如果存在，占用这个位置，返回对应下标；如果不存在位置，或者打开文件失败，那么（打开文件成功时）关闭文件，并返回`-1`。

因此，我添加了一个 `FileRecord` 类，并在 `FileSystem` 类中添加了 `FileRecord` 类的数组。

```
class FileRecord
{
public:
    OpenFile* file;
    int TID;
    bool valid;
};

FileRecord* OpenedFiles[MAX_FILE_NUMBER];

#define MAX_FILE_NUMBER 500
```

同时，修改构造函数如下，用于对打开文件表初始化：

```
FileSystem::FileSystem(bool format)
{
    DEBUG('f', "Initializing the file system.\n");
    if (format) ...
    else ...

    for (int i=0; i<MAX_FILE_NUMBER; ++i)
    {
        OpenedFiles[i] = new FileRecord;
        OpenedFiles[i]->file = NULL;
        OpenedFiles[i]->TID = -1;
        // 0 and 1 is used for console
        OpenedFiles[i]->valid = (i>=2 ? FALSE : TRUE);
    }
    tableLock = new Lock("file system table lock");
}
```

为了实现对这个数组的操作。我还添加了以下几个方法，分别用于添加已打开的文件、移除并关闭文件、获取打开文件、输出打开文件表中所有打开文件的信息。`FileRecord` 类中的 `TID` 可以避免当前线程操作其他线程的打开文件，`valid` 可以区分这个表项是否有效。

```
int
FileSystem::AddNewFile(OpenFile* file)
{
    int pos = 0;
    for (pos=2; pos<MAX_FILE_NUMBER; ++pos)
    {
        if (!OpenedFiles[pos]->valid)
        {
            OpenedFiles[pos]->valid = TRUE;
            OpenedFiles[pos]->file = file;
            OpenedFiles[pos]->TID = currentThread->getTID();
            return pos;
        }
    }
    return -1;
}
```

```

bool
FileSystem::RemoveFile(int openFileID)
{
    if(openFileID < 2)
    {
        DEBUG('S', "You cannot remove console input & output!\n");
        return FALSE;
    }

    if(OpenedFiles[openFileID]->valid && OpenedFiles[openFileID]->TID == currentThread->getTID())
    {
        OpenedFiles[openFileID]->valid = FALSE;
        delete OpenedFiles[openFileID]->file;
        OpenedFiles[openFileID]->file = NULL;
        OpenedFiles[openFileID]->TID = -1;
        return TRUE;
    }
    return FALSE;
}

OpenFile*
FileSystem::GetFile(int openFileID)
{
    if(openFileID<2)
    {
        DEBUG('S', "console input and output should not be get.\n");
        return NULL;
    }

    if(OpenedFiles[openFileID]->valid && OpenedFiles[openFileID]->TID == currentThread->getTID())
    {
        return OpenedFiles[openFileID]->file;
    }
    DEBUG('S', "This file doesn't belong to you or invalid ID.\n");
    return NULL;
}

void
FileSystem::PrintTable()
{
    printf("\n\n-----open file table-----\n\n\n");
    for(int i=2; i<MAX_FILE_NUMBER; ++i)
    {
        if(OpenedFiles[i]->valid)
        {
            printf("file ID: %d, owner's TID is %d\n", i, OpenedFiles[i]->TID);
        }
    }
    printf("\n\n-----\n\n\n");
}

```

注意到，用户级程序可以并行运行。因此，我们要为这个全局数组加上互斥锁。由于受到头文件依赖关系的限制，这个锁在 system.h 中声明，在 system.cc 中定义。

```

extern Lock* tableLock;

bool used_TID[MAX_THREAD_NUM];
Thread *Thread_Pointer[MAX_THREAD_NUM];
bool RoundRobin;
Lock* tableLock;

Lock* tableLock = new Lock("openfile table lock");

```

这里需要说明一点，实际上，我们也可以直接把调用 fileSystem->Open()得到的指针直接作为文件 ID，但如果这样做，打开文件的内容可能会被随意篡改，造成意想不到的后果。除此之外，使用打开文件表提供 ID，也是**基于 Linux 中文件描述符的思想**。

因此，基于以上实现，Open 的处理函数内容如下。首先从寄存器中读取文件名指针，并获取文件名，再从文件系统中打开文件，并添加到打开文件表。最后把文件 ID 写入返回值寄存器中。上述操作完成，后，修改 PC 值。

```
void
OpenHandler()
{
    DEBUG('S', "Try to open file\n");
    // get file name
    int startAddr = machine->ReadRegister(4);
    // printf("\naddr %x\n\n", startAddr);
    char* fileName = GetFileName(startAddr);

    // get file
    OpenFile* openFile = fileSystem->Open(fileName);
    ASSERT(openFile != NULL);

    // add to list
    tableLock->Acquire();
    int id = fileSystem->AddNewFile(openFile);
    tableLock->Release();
    ASSERT(id>=2);

    // write return value
    machine->WriteRegister(2, (openFileDialog)id);
    IncreasePC();
}
```

对于 Close，我们需要做的就是清空对应的表项，并关闭这个打开文件。

```
void
CloseHandler()
{
    DEBUG('S', "Try to close file\n");
    // tableLock->Acquire();
    // fileSystem->PrintTable();
    // tableLock->Release();
    // get file id
    int id = machine->ReadRegister(4);
    // printf("\nid is %d\n\n", id);

    // remove file
    tableLock->Acquire();
    ASSERT(fileSystem->RemoveFile(id));
    // fileSystem->PrintTable();
    tableLock->Release();
    IncreasePC();
}
```

下面我们再来看一下 Read 和 Write 的实现。

对于 Read，我们首先要从寄存器中读取 3 个参数，然后获取对应的打开文件，并调用打开文件的 Read 方法来读取文件内容，最后，把内容写入内存中对应的地址。这里我也是

逐字节写入，且由于 TLB 的实现，为每次写赋予了第二次机会。写完成以后，设置返回值，并增加 PC。

此外，我们还需要处理描述符为 0（控制台输入）时的特殊情况。这时，我们直接使用 `scanf` 来完成对数据的读取。

```
void
ReadHandler()
{
    DEBUG('S', "Try to read file\n");
    // get parameters
    int addr = (char*) machine->ReadRegister(4);
    int size = machine->ReadRegister(5);
    int id = machine->ReadRegister(6);

    // read file
    char* buffer = new char[size+1];
    int numBytes = 0;
    if(id == 0)
    {
        scanf("%s", buffer);
        numBytes = strlen(buffer);
    }
    else
    {
        tableLock->Acquire();
        OpenFile* file = fileSystem->GetFile(id);
        tableLock->Release();
        ASSERT(file!=NULL);
        numBytes = file->Read(buffer, size);
    }
    buffer[numBytes] = '\0'; // ensure content

    // write to memory
    // printf("\nread content %s\n\n", buffer);
    for(int i=0; i<numBytes; ++i)
    {
        if(!machine->WriteMem(addr+i, 1, (int)buffer[i]))
            machine->WriteMem(addr+i, 1, (int)buffer[i]);
    }

    // set return value and PC
    machine->WriteRegister(2, numBytes);
    IncreasePC();
}
```

Write 与 Read 的实现类似。首先获取三个参数；然后尝试获取打开文件；接下来函数从内存中逐字节读取要写入的内容，与上面的原理相同，这里也为内存的读操作提供了第二次机会。接下来，函数向文件中写入内容。最后，函数递增 PC。同样地，我们要对 ID 为 1（控制台输出）的情况进行处理。此时，我们直接使用 `printf` 来进行控制台输出。

实际上，在我的实现中，在打开文件表初始化时便强行占用的 0 和 1 两个 ID，以防干扰控制台输入输出。另外，当打开和关闭文件时，对打开文件表的操作中，如果发现传入的 id 为 0，那么函数会返回空指针/FALSE 等错误信息。Read 中只能使用控制台输入 id，Write 中只能使用控制台输出 id，交叉使用便会在上述实现中报错。

```

void
WriteHandler()
{
    DEBUG('S', "Try to write file\n");
    // get parameters
    int addr = (char*) machine->ReadRegister(4);
    int size = machine->ReadRegister(5);
    int id = machine->ReadRegister(6);

    // get content(read memory)
    char* buffer = new char[size+1];
    for(int i=0; i<size; ++i)
    {
        if(!machine->ReadMem(addr+i, 1, (int*)&buffer[i]));
        | machine->ReadMem(addr+i, 1, (int*)&buffer[i]);
    }
    buffer[size] = '\0'; // ensure content

    // write to file
    // printf("\nwrite content %s\n\n", buffer);
    int numBytes = 0;
    if(id == 1)
    {
        numBytes = strlen(buffer);
        printf("%s", buffer);
    }
    else
    {
        tableLock->Acquire();
        OpenFile* file = fileSystem->GetFile(id);
        tableLock->Release();
        ASSERT(file!=NULL);
        numBytes = file->Write(buffer, size);
    }

    // increase PC
    IncreasePC();
}

```

以上我们便完成了对文件操作的系统调用的实现。现在，ExceptionHandler 中关于系统调用的分支处理如下。其中的 Halt 和 Exit 是在 Lab4 中初步实现的。

```

if (which == SyscallException)
{
    if(type == SC_Halt)
    {
        DEBUG('a', "Shutdown, initiated by user program.\n");
        TLBMissRate();
#ifndef USER_PROGRAM
        if(currentThread->space != NULL)
        {
#endif
            machine->freeMem();
#ifndef USE_BITMAP
            delete currentThread->space;
            currentThread->space = NULL;
#endif
        }
#endif
        interrupt->Halt();
    }
    else if(type == SC_Exit)
        ExitHandler();
    else if(type == SC_Create)
        CreateHandler();
    else if(type == SC_Open)
        OpenHandler();
    else if(type == SC_Read)
        ReadHandler();
    else if(type == SC_Write)
        WriteHandler();
    else if(type == SC_Close)
        CloseHandler();
}

```

## Exercise3

这一部分要求我们对上面实现的各项系统调用进行测试。

首先，我测试了创建文件的实现。测试程序中，函数给出一个文件名，然后调用 Create 系统调用创建文件。**注意，这里如果不逐字符对文件名字符串赋值，在运行时会报错，这是 Nachos 对 MIPS 指令集模拟的实现导致的。**代码在 test 文件夹中，使用与其他程序类似的 Makefile 语句进行编译。

```
#include "syscall.h"
|
int main()
{
    char name[9];
    name[0] = 't';
    name[1] = 'e';
    name[2] = 's';
    name[3] = 't';
    name[4] = '.';
    name[5] = 't';
    name[6] = 'x';
    name[7] = 't';
    name[8] = '\0';

    Create(name);
}
```

目前我们已经完成了文件系统、线程管理、内存管理，因此我在 vm 文件夹下开启了先前 Lab 的全部功能来进行测试。

```
# DEFINES = -DTHREADS -DUSER_PROGRAM -DVM -DFILESYS_NEEDED -DFILESYS
DEFINES = -DUSER_PROGRAM -DVM -DFILESYS_NEEDED -DFILESYS -DUSE_INDIRECT -DMULTI_LEVEL_DIR -DUSE_TLB -DUSE_BITMAP -DUSE_DISK
```

为了更好地测试，我编写了一段 shell 脚本来连续地完成把文件放入 Nachos 文件系统到执行的全过程。

```
#!/bin/sh

cd ../

echo "===" clean file system ==="
./nachos -V -f

echo ""
echo "===" move executable file to file system ==="
./nachos -V -cp ../test/Create Create

echo ""
echo "===" show file system contents"
./nachos -V -D

echo ""
echo "===" try to create a file ==="
./nachos -V -x Create -d S
# delete vm, later will implement in exit
./nachos -V -rm VirtualMemory0

echo ""
echo "===" show file system contents"
./nachos -V -D
```

运行上述脚本的结果如下：



可以看到，程序成功地在 Nahcos 文件系统中创建了 test.txt 文件。因此，可以认为 Create 的实现是成功的。

接下来，我又完成了另一份代码，来对上述所有对文件系统操作的功能进行测试。代码创建一个文件 `test.txt`，并从一个在执行这份代码之前已经加载进 Nachos 文件系统的文件中（`hello`）读取字符串，写入 `test.txt`。然后，代码从 `text.txt` 中读取刚才写入的内容，并写入 `hello`。这里为 `test.txt` 创建两个描述符是为了分别从文件位置 0 处开始读/写。而 `hello` 用于测试从操作过的文件位置后进一步写入内容。

```
#include "syscall.h"

#define len 16 // hello, my world!\a

int main()
{
    int fd_read, fd_write, fd2;

    char name[9];
    char name1[6];

    char buffer[13];
    char out_buffer[13];

    name[0] = 't';
    name[1] = 'e';
    name[2] = 's';
    name[3] = 't';
    name[4] = '.';
    name[5] = 't';
    name[6] = 'x';
    name[7] = 't';
    name[8] = '\0';
    Create(name);
    fd_write = Open(name);

    name1[0] = 'h';
    name1[1] = 'e';
    name1[2] = 'l';

    name1[3] = 'l';
    name1[4] = 'o';
    name1[5] = '\0';
    fd2 = Open(name1);

    Read(buffer, len, fd2);
    Write(buffer, len, fd_write);

    fd_read = Open(name);
    Read(out_buffer, len, fd_read);
    Write(out_buffer, len, fd2); // write after

    Close(fd_read);
    Close(fd_write);
    Close(fd2);
}
```

为了完成上述测试，我编写了如下的 shell 脚本。在运行上面的程序之前，要先把 hello 装入 Nachos 文件系统。

```
#!/bin/sh

cd ../

echo "===" reset file system ==="
./nachos -V -f

echo ""
echo "===" move a normal file to file system ==="
./nachos -V -cp test/hello hello

echo ""
echo "===" move executable file to file system ==="
./nachos -V -cp ./test/FileSyscall FileSyscall

echo ""
echo "===" show file system contents"
./nachos -V -D

echo ""
echo "===" test file system ==="
./nachos -V -x FileSyscall -d S
# delete vm, later will implement in exit
./nachos -V -rm VirtualMemory0

echo ""
echo "===" show file system contents"
./nachos -V -D
```

上述脚本运行结果如下。



```
write content hello,my world!

Try to open file
addr 630

Try to read file
read content hello,my world!

Try to write file
write content hello,my world!

Try to close file

-----open file table-----

file ID: 2, owner's TID is 0
file ID: 3, owner's TID is 0
file ID: 4, owner's TID is 0

-----
id is 4

-----open file table-----

file ID: 2, owner's TID is 0
file ID: 3, owner's TID is 0
```

```
-----
Try to close file

-----open file table-----

file ID: 2, owner's TID is 0
file ID: 3, owner's TID is 0

-----
id is 2

-----open file table-----

file ID: 3, owner's TID is 0

-----
Try to close file

-----open file table-----

file ID: 3, owner's TID is 0

-----
```



**Bitmap set:** 0..1..2..3..4..5..6..7..8..9..10..11..12..13..14..15..16..17..18..19..51..55

可以看到，执行完用户级程序后，文件系统中确实创建了 `test.txt`，并且写入了 `hello` 的原始内容。同时，`hello` 中也新写入了一份 `test.txt` 中的内容。可以认为，我们上述的文件系统相关的系统调用的实现是正确的。

最后，我们还要对控制台输入输出进行测试。我编写了如下的测试代码。

```
#include "syscall.h"

#define len 17 //test_for_console.

int main()
{
    int fd_read, fd_write, num;

    char in_buffer[len];
    char out_buffer[len];
    char name[9];
    name[0] = 't';
    name[1] = 'e';
    name[2] = 's';
    name[3] = 't';
    name[4] = '.';
    name[5] = 't';
    name[6] = 'x';
    name[7] = 't';
    name[8] = '\0';

    Create(name);
    fd_write = Open(name);

    num = Read(in_buffer, len, ConsoleInput);
    Write(out_buffer, num, fd_write);

    fd_read = Open(name);
    num = Read(out_buffer, len, fd_read);
    Write(out_buffer, num, ConsoleOutput);

    Close(fd_read);
    Close(fd_write);
}
```

同时，我编写了下面的脚本来对上述代码进行测试：

```
#!/bin/sh

cd ../

echo "==== clean file system ===="
./nachos -V -f

echo ""
echo "==== move executable file to file system ===="
./nachos -V -cp ..//test/testConsole testConsole

echo ""
echo "==== show file system contents"
./nachos -V -D

echo ""
echo "==== try to create a file ===="
./nachos -V -x testConsole -d S
# delete vm, later will implement in exit
./nachos -V -rm VirtualMemory0

echo ""
echo "==== show file system contents"
./nachos -V -D
```

上述脚本运行结果如下。

**Bitmap set:**  
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,

```
--- try to create a file ---  
Start to create new file
```

addr 5d8

test.txt filename

Try to open file

addr 5d8

Try to read file

test\_for\_console.

```
read content test_for_console.
```

Try to write file

```
write content test_for_console
```

Try to open file

addr 5d8

Try to read file

```
read content test_for_console.
```

Try to write file

```
write content test_for_console
```

test\_for\_console.Try to close file

-----open file table-----

```
file ID: 2, owner's TID is 0  
file ID: 3, owner's TID is 0  
  
-----  
  
id is 3  
  
-----open file table-----  
  
file ID: 2, owner's TID is 0  
  
-----  
  
Try to close file  
  
-----open file table-----  
  
file ID: 2, owner's TID is 0  
  
-----  
  
id is 2  
  
-----open file table-----
```

可以看到，程序可以从控制台读取内容，写入 Nachos 文件系统的文件中。程序也可以从 Nachos 的文件系统中读取内容，并输出到控制台上。因此，可以认为，上述系统调用的实现是正确的。

【注】为了更好地展示运行流程，我在系统调用中添加了必要的输出，在 Exercise2 展示的代码中注释掉了。此外，为了展示 Exit 时的打开文件情况，测试时，我在 Exit 最后也添加了对打开文件表的输出。

【注】命令行中的-V 用于去除结束时的状态输出，详见 Lab5 的报告。

## Exercise4

这一部分要求我们实现与线程控制相关的系统调用。从上面对系统调用的实现的分析，我们只需要在 exception.cc 中完成相应的处理。

首先来看 Exec 的实现。根据注释，这个系统调用的功能是，根据参数加载可执行文件并执行它，返回值为可以区分不同线程的标识符。

实际上，由于在先前的实现中，我们引入了唯一的线程 ID（Thread 类中的 TID），所以我们可以直接使用 TID 来充当线程标识符。此外，由于这个函数会提供返回值，因此，我认为这个系统调用与 Linux 中的 Execve 应该不完全相同。Execve 是直接重置调用它的进程的代码等内容，且不返回。在这里，为了体现返回值的作用，我认为应该把这个系统调用设计为：新开辟一个线程，在新线程内进行代码段等内容的初始化。

基于以上分析，Exec 的实现如下，大致流程为：首先利用参数获取可执行文件名的起始地址；然后创建一个新线程，执行 Exec 的初始化函数（以文件名的起始地址作为参数），最后把新创建的线程的 TID 放入返回值，并递增 PC。

```
void
ExecHandler()
{
    DEBUG('S', "Exec system call\n");
    // get file name
    int addr = machine->ReadRegister(4);

    // create new thread and set it
    Thread* thread = new Thread("Exec by other thread");
    thread->Fork(ExecFunc, (void*)addr);

    // fill return value, increase PC
    machine->WriteRegister(2, thread->getTID());
    IncreasePC();
}
```

Exec 的初始化函数流程为：首先根据传入的地址获取文件名，并打开文件；然后为当前线程创建并初始化地址空间；最后，关闭文件，并开始执行指令。

```

void
ExecFunc(int addr)
{
    DEBUG('S', "initialize thread created by Exec\n");
    char* fileName = GetFileName(addr);

    // open executable file
    OpenFile* file = fileSystem->Open(fileName);
    ASSERT(file!=NULL);

    // create thread and address space
    AddrSpace* space = new AddrSpace(file);
    space->InitRegisters();
    space->RestoreState();
    currentThread->space = space;

    delete file;
    // printf("----start to run---\n");
    machine->Run();
}

```

接下来来看 **Join** 的实现。根据注释，这个系统调用的功能为，传入某个线程的 TID，当前线程等待这个线程执行完毕，最终返回被等待线程的退出状态。

由于 **Nachos** 并没有提供类似于 **sigsuspend** 的功能，所以我选择使用忙等的方法：使用 while 循环来等待线程的结束，如果没有结束，则调用当前线程的 Yield 函数，主动让出 CPU。由于在 **Lab1** 实现 **TID** 机制的同时，我们维护了一个全局的线程表，所以我们可以使用这个线程表来判断终止情况。

为了获取终止状态，我在 **system.h** 中声明了一个全局的变量，用于储存相应线程的返回状态。

```

extern int end_state[MAX_THREAD_NUM];

```

注意到线程数组、返回状态数组、线程使用状态数组都是全局的、被所有线程共享的，所以我们还应该为它们加上互斥锁。实际上，我们只需要一把锁，把它们看作一个线程的整体内容即可。

```

extern Lock* threadLock;

for(int i=0; i<MAX_THREAD_NUM; ++i)
{
    end_state[i] = -1;
    used_TID[i] = false;
    Thread_Pointer[i] = NULL;
}
tableLock = new Lock("openfile table lock");
threadLock = new Lock("thread table lock");

```

注意到先前在创建线程和销毁线程处理上述全局结构时，我们并没有对并发进行处理，但由于 **Nachos** 的内在初始化逻辑，**main** 线程创建会早于对锁的初始化。因此，在构造函数中，我选择通过开关中断的方式来实现原子化（因为在 **Nachos** 的实现中，如果不

主动 Yield，只有触发中断时才会进行线程调度）。而析构函数中我们可以直接使用锁操作。

```
this->TID = -1;
IntStatus old = interrupt->SetLevel(IntOff);
for(int i=0; i<MAX_THREAD_NUM; ++i)
{
    if(!used_TID[i])
    {
        this->TID = i;
        used_TID[i] = true;
        Thread_Pointer[i] = this;
        break;
    }
}
(void) interrupt->SetLevel(old);

Thread::~Thread()
{
    DEBUG('t', "Deleting thread \"%s\"\n", name);

    ASSERT(this != currentThread);
    if (stack != NULL)
        DeallocBoundedArray((char *) stack, StackSize * sizeof(int));

    //***** added by Li Cong 1800012826
    threadLock->Acquire();
    used_TID[this->TID] = false;
    threadLock->Release();

    //***** added by Li Cong 1800012826
}
```

基于以上想法，Join 的处理程序实现如下。

```
void
JoinHandler()
{
    // get Thread id
    DEBUG('S', "System call Join\n");
    int TID = machine->ReadRegister(4);
    // printf("\nwait for thread with tid %d\n", TID);

    // check whether the thread ends, yield if still running
    while(TRUE)
    {
        threadLock->Acquire();
        if(!used_TID[TID])
        {
            threadLock->Release();
            break;
        }
        // run for too many times
        // DEBUG('S', "wait for thread\n");
        threadLock->Release();
        currentThread->Yield();
    }

    // after waiting, write return value and increase PC
    DEBUG('S', "Finish waiting\n");
    threadLock->Acquire();
    machine->WriteRegister(2, end_state[TID]);
    threadLock->Release();
    IncreasePC();
}
```

然后我们来看一下 Yield 系统调用的实现。这个系统调用的功能就是让当前线程主动让出 CPU。因此，我们只需要把 Thread 类中的 Yield 函数封装一下。此外，**我们应该在调用 currentThread 的 Yield 之前先递增 PC，否则，由于 mipssim.cc 中系统调用执行完毕后没有主动递增 PC，当前线程便会不停地重复执行 Yield 系统调用。**

```
void
YieldHandler()
{
    DEBUG('S', "System call Yield\n");

    // or we will loop infinitely
    IncreasePC();
    currentThread->Yield();
    DEBUG('S', "Back from Yield\n");
}
```

下面来看一下 Exit 的实现。实际上，Lab4 中，我已经初步实现了 Exit 系统调用，但是我们现在还需要为文件系统和上面的 Join 进行相应地调整。函数流程如下：

首先获取退出状态并记录到全局的退出状态表中，供 Join 获取。

```
// deal with exit status
int status = machine->ReadRegister(4); // get the first argument ----- exit status
if(status == 0)
    DEBUG('S', "*****User program exit normally*****\n");
else
    DEBUG('S', "*****User program exit with status %d*****\n", status);
// record exit status
threadLock->Acquire();
end_state[currentThread->getTID()] = status;
threadLock->Release();
```

接下来清理地址空间。由于 Lab4 完成时还没有实现文件系统功能，所以现在我们在退出时也要把磁盘中的虚拟内存缓存给清除。

```
// release space
#ifndef USER_PROGRAM
    if(currentThread->space != NULL)
    {
#endif
        #if USE_BITMAP || INVERTED_PAGETABLE
            machine->freeMem();
        #endif
        currentThread->space->SaveState(); // clear tlb when current thread exits
        // should delete virtual memory file
        fileSystem->Remove(currentThread->space->VMName);

        delete currentThread->space;
        currentThread->space = NULL;
    }

#ifndef INVERTED_PAGETABLE
    printf("=====PAGE TABLE=====\\n");
    for(int i=0; i<NumPhysPages; ++i)
    {
        printf("%d\t%d\t%d\t%d\\n", machine->pageTable[i].virtualPage, machine->pageTable[i].physicalPage,
               machine->pageTable[i].valid, machine->pageTable[i].TID);
    }
    printf("=====\\n");
#endif
```

最后，Exit 处理程序检查当前线程是否还存在用户层面上打开的文件，如果存在，关

闭所有当前线程的打开文件。

```
// should clear unclosed files
tableLock->Acquire();
fileSystem->CloseAll();
// fileSystem->PrintTable();
tableLock->Release();

// finish this user thread
currentThread->Finish();
```

为了实现这一功能，我在 FileSystem 中也添加了相应的方法。检查完毕后，调用 currentThread 的 Finish 函数来退出。

```
void
FileSystem::CloseAll()
{
    DEBUG('S', "close opened files before user program exit\n");
    for(int i=2; i<MAX_FILE_NUMBER; ++i)
    {
        if(OpenedFiles[i]->valid && OpenedFiles[i]->TID == currentThread->getTID())
        {
            OpenedFiles[i]->valid = FALSE;
            delete OpenedFiles[i]->file;
            OpenedFiles[i]->file = NULL;
            OpenedFiles[i]->TID = -1;
        }
    }
}
```

最后，我们来看一下 Fork 系统调用的实现。Fork 要求创建一个与当前线程地址空间完全相同的新线程，并从传入的函数处开始继续执行。为了实现这一功能，我认为应该按如下步骤来操作。

## (1) 读取寄存器，获取函数的起始地址

```
DEBUG('S', "System call Fork\n");
// get func ptr, also regarded as new PC for child thread
int addr = machine->ReadRegister(4);
```

## (2) 进行地址空间的拷贝。

为了实现这一功能，我们应该考虑以下几个问题。

### 【完善 AddrSpace 类】

为了实现这一步操作，我们需要对 AddrSpace 类进行进一步的完善。

首先，注意到 **Nachos** 中的地址空间都必须以可执行文件的打开文件对象作为构造函数的参数，但是在先前我们并没有办法直接获取。又注意到，**Nachos** 的文件系统中，创建一个打开文件需要获取文件头扇区号和文件名中的一个。因此，我在 AddrSpace 类中添加了可执行文件的文件头扇区号，用于标识打开文件。

```
int exeSector;
```

```
AddrSpace::AddrSpace(OpenFile *executable)
{
    NoffHeader noffH;
    unsigned int i, size;

    executable->ReadAt((char *)&noffH, sizeof(noffH), 0);
    if ((noffH.noffMagic != NOFFMAGIC) && (WordToHost(noffH.noffMagic) == NOFFMAGIC))
        SwapHeader(&noffH);
    ASSERT(noffH.noffMagic == NOFFMAGIC);

    exeSector = executable->GetHeaderSector();
```

接下来，由于当前线程在运行中可能会对内存进行操作，所以，我们还需要在 Fork 时把当前线程在内存中存放的有效页写回虚拟内存文件中。因此，我在 AddrSpace 中添加了一个函数。

```
void
AddrSpace::WriteBackAll()
{
    OpenFile *vm = fileSystem->Open(currentThread->space->VMName);
    ASSERT(vm!=NULL);
    for(int i=0; i<numPages; ++i)
    {
        if(pageTable[i].valid)
            vm->WriteAt(&(machine->mainMemory[PageSize*pageTable[i].physicalPage]), PageSize, i*PageSize);
    }
    delete vm;
}
```

此外，为了获取虚拟内存空间大小，由于每个页的大小固定，所以我还添加了获取地址空间页数的函数。

```
int GetNumPages() {return numPages;}
```

### 【ForkHandler 的实现】

基于对地址空间的以上完善，我通过如下流程来实现地址空间的拷贝：

首先，处理函数把当前线程的地址空间写回虚拟内存文件中，并利用当前线程地址空间中储存的文件头扇区号，创建可执行文件的打开对象，并初始化一个新的地址空间对象。

```
// update vm and initialize new space
currentThread->space->WriteBackAll();
OpenFile* executable = new OpenFile(currentThread->space->exeSector);
AddrSpace* addrSpace = new AddrSpace(executable);
delete executable;
```

接下来，由于刚才初始化的地址空间所对应的虚拟内存文件的内容是原始的可执行文件的内容，我们还需要把它拷贝为目前执行状态下的地址空间内容。因此，我打开了当前线程的地址空间和新线程的地址空间文件，读取当前线程地址空间的全部内容，写入新建立的地址空间中。这样，便实现了“相同地址空间”的要求。

注意，如果不做这一步刷新，新线程在继续执行时，无法使用已赋值的全局变量，这

与“相同的地址空间”要求相违背。

```
DEBUG('S', "---flush virtual memory from %s to %s---\n", currentThread->space->VMName, addrSpace->VMName);
OpenFile* vm = fileSystem->Open(currentThread->space->VMName);
OpenFile* nextvm = fileSystem->Open(addrSpace->VMName);
ASSERT(vm != NULL && nextvm != NULL);
int size = currentThread->space->GetNumPages()*PageSize;
char* buf = new char[size];
vm->Read(buf, size);
nextvm->Write(buf, size);
delete vm; delete nextvm;
DEBUG('S', "---finish flush virtual memory---\n");
```

### (3) 考虑如何为新线程设置地址空间和 PC 值

事实上，为了实现 Fork 系统调用，我们肯定会用到 **Thread** 类中的 **Fork** 函数，但是这个函数只能传入一个参数。因此，我在 **addrspace.h** 中声明了一个 **ForkInfo** 类，内容包括：一个 **AddrSpace** 类的指针，新的 PC 值。我们在调用 **Fork** 之前先初始化一个 **ForkInfo** 对象，然后把它的指针传进去。

```
class ForkInfo
{
public:
    AddrSpace* addrSpace;
    int PC;
};

// initialize info
ForkInfo* info = new ForkInfo;
info->addrSpace = addrSpace;
info->PC = addr;
```

### (4) 创建新线程，并让其执行初始化函数

接下来，便可以使用上面的结构，用 **Fork** 创建线程了。为了调用 **Fork**，我们还需要实现一个初始化函数。函数内容如下，流程为：首先根据传入的参数生成 **ForkInfo** 结构，接下来根据结构中的数值，设置地址空间并进一步初始化，再设置新的 PC 值。最后，调用 **machine** 对象中的 **Run** 函数，准备执行指令。

```
void
ForkFunc(int infoAddr)
{
    DEBUG('S', "initialize thread created by Fork\n");
    // get info
    ForkInfo* info = (ForkInfo*) infoAddr;

    // initialize addr space
    DEBUG('S', "set addr space\n");
    currentThread->space = info->addrSpace;
    currentThread->space->InitRegisters();
    currentThread->space->RestoreState();

    // change PC
    DEBUG('S', "set PC\n");
    machine->WriteRegister(PCReg, info->PC);
    machine->WriteRegister(NextPCReg, info->PC+4);
    machine->Run();
}
```

而在 ForkHandler 中，我们需要做的就是创建线程并调用 Fork。

```
// execute fork function
Thread* thread = new Thread("created by Fork Syscall");
thread->Fork(ForkFunc, (void*)info);
```

## (5) 拷贝当前线程的已打开文件

Fork 函数还应该考虑打开文件的拷贝问题。由于新线程与原线程拥有相同的地址空间，因此它们也拥有相同的文件描述符。我们应该为这个情形提供一种解决方案。因此，我们还需要进一步修改文件系统中的相应实现。

### 【对 FileSystem 类的修改】

实际上，虽然不做任何处理，新线程也可以操作文件描述符对应的文件，但是，如果原始线程在新线程执行文件操作之前关闭了这个文件，那么新线程便无法正常操作这个文件了。**因此，出于对所有线程都能互不干扰地操作文件的考虑，我选择使用复制文件表项的方式来实现对打开文件的拷贝。**（由于时间关系，我并没有实现 Linux 的三级描述符结构，现在实际上只有两级，为了实现在我看来更重要的保证文件操作正常进行的抽象，两级结构下的实现效果会与 Linux 中略有不同）

因此，我在原来的 FileRecord 类中添加了一个 **ForkFileID**，用于记录当前文件是否是由于 **Fork** 得来的，如果是，它是从哪个原始标识符拷贝得到的。这个值会被初始化为-1。

```
class FileRecord
{
public:
    OpenFile* file;
    int TID;
    int ForkFileID;
    bool valid;
};

FileSystem::FileSystem(bool format)
{
    DEBUG('f', "Initializing the file system.\n");
    if (format) ...
    else ...

    for(int i=0; i<MAX_FILE_NUMBER; ++i)
    {
        OpenedFiles[i] = new FileRecord;
        OpenedFiles[i]->file = NULL;
        OpenedFiles[i]->TID = -1;
        OpenedFiles[i]->ForkFileID = -1;
        // 0 and 1 is used for console
        OpenedFiles[i]->valid = (i>=2 ? FALSE : TRUE);
    }
    tableLock = new Lock("file system table lock");
}
```

为了完成 Fork 中的拷贝，我添加了一个 CopyFile 函数。这个函数首先检查是否能容纳所有的表项，如果不能，返回 FALSE。接下来，函数遍历打开文件表，依次为当前线程的打开文件进行拷贝。注意，为了实现真正的互不干扰（`delete` 后仍能继续使用），我对每个打开文件对象进行了深拷贝（创建新指针，并同步文件位置）。在使用 `AddNewFile` 寻找到空闲表项后，还需要对 `TID`、`ForkFileID` 进行相应的修改（否则 `TID` 仍为当前线程的 ID）。

\*注：我认为，当线程正式退出后，打开文件都应该彻底关闭，而不是简单地提供一个文件描述符不能使用的抽象。因此，考虑到在原线程先 `Exit` 然后 `Fork` 线程执行文件操作的情形，我选择了使用深拷贝来完成上述功能。这也导致了 `Fork` 后的打开文件操作与 `Linux` 中的 `fork` 会有所不同。

```
bool
FileSystem::CopyFile(int newTID)
{
    DEBUG('S', "copy opened files for Fork syscall\n");
    // check whether table can hold all files
    int leftNum = 0;
    int openNum = 0;
    for(int i=2; i<MAX_FILE_NUMBER; ++i)
    {
        if(!OpenedFiles[i]->valid)
            leftNum++;
        else
        {
            if(OpenedFiles[i]->TID == currentThread->getTID())
                openNum++;
        }
    }
    if(leftNum<openNum)
        return FALSE;
    // copy files
    for(int i=2; i<MAX_FILE_NUMBER; ++i)
    {
        if(OpenedFiles[i]->valid && OpenedFiles[i]->TID == currentThread->getTID())
        {
            // do deep copy
            OpenFile* file = new OpenFile(OpenedFiles[i]->file->GetHeaderSector());
            file->Seek(OpenedFiles[i]->file->GetPos());
            // add to table
            int fd = AddNewFile(file);
            ASSERT(fd != -1);
            OpenedFiles[fd]->ForkFileID = i;
            OpenedFiles[fd]->TID = newTID;
        }
    }
    // PrintTable();
    return TRUE;
}
```

此外，先前实现的一些打开文件表操作也需要进行修改，以适配 `ForkFileID` 机制。

```
void
FileSystem::CloseAll()
{
    DEBUG('S', "close opened files before user program exit\n");
    for(int i=2; i<MAX_FILE_NUMBER; ++i)
    {
        if(OpenedFiles[i]->valid && OpenedFiles[i]->TID == currentThread->getTID())
        {
            OpenedFiles[i]->valid = FALSE;
            delete OpenedFiles[i]->file;
            OpenedFiles[i]->file = NULL;
            OpenedFiles[i]->ForkFileID = -1;
            OpenedFiles[i]->TID = -1;
        }
    }
}
```

```

bool
FileSystem::RemoveFile(int openFileID)
{
    if(openFileID < 2)
    {
        DEBUG('S', "You cannot remove console input & output!\n");
        return FALSE;
    }

    if(OpenedFiles[openFileID]->valid && OpenedFiles[openFileID]->TID == currentThread->getTID())
    {
        OpenedFiles[openFileID]->valid = FALSE;
        delete OpenedFiles[openFileID]->file;
        OpenedFiles[openFileID]->file = NULL;
        OpenedFiles[openFileID]->TID = -1;
        OpenedFiles[openFileID]->ForkFileID = -1;
        return TRUE;
    }

    for(int i=2; i<MAX_FILE_NUMBER; ++i)
    {
        if(OpenedFiles[i]->valid && OpenedFiles[i]->TID == currentThread->getTID() && OpenedFiles[i]->ForkFileID == openFileID)
        {
            OpenedFiles[i]->valid = FALSE;
            delete OpenedFiles[i]->file;
            OpenedFiles[i]->file = NULL;
            OpenedFiles[i]->TID = -1;
            OpenedFiles[i]->ForkFileID = -1;
            return TRUE;
        }
    }

    return FALSE;
}

OpenFile*
FileSystem::GetFile(int openFileID)
{
    if(openFileID<2)
    {
        DEBUG('S', "console input and output should not be get.\n");
        return NULL;
    }

    if(OpenedFiles[openFileID]->valid && OpenedFiles[openFileID]->TID == currentThread->getTID())
    {
        return OpenedFiles[openFileID]->file;
    }

    for(int i=2; i<MAX_FILE_NUMBER; ++i)
    {
        if((OpenedFiles[i]->valid) && (OpenedFiles[i]->TID == currentThread->getTID()) && (OpenedFiles[i]->ForkFileID == openFileID))
        {
            DEBUG('S', "This file doesn't belong to you or invalid ID.\n");
            return NULL;
        }
    }
}

void
FileSystem::PrintTable()
{
    printf("\n\n-----open file table-----\n\n");
    printf("current TID is %d \n", currentThread->getTID());
    for(int i=2; i<MAX_FILE_NUMBER; ++i)
    {
        if(OpenedFiles[i]->valid)
        {
            printf("file ID: %d, owner's TID is %d, fork id is %d\n", i, OpenedFiles[i]->TID, OpenedFiles[i]->ForkFileID);
        }
    }
    printf("\n\n-----\n\n");
}

```

### 【在 ForkHandler 中的实现】

由于上面我们已经提供了一个完善的拷贝方法，所以在 ForkHandler 中直接调用即可。但是要注意需要加锁以避免竞争问题。

```

// copy open-file table
DEBUG('S', "---copy---\n");
tableLock->Acquire();
ASSERT(fileSystem->CopyFile(thread->getTID()));
tableLock->Release();
DEBUG('S', "---finish copy---\n");

```

## (6) 递增当前线程的 PC

最后，我们还需要递增 PC，以防止无限循环调用处理函数。

```
// increase PC  
IncreasePC();
```

综上所述，整个处理函数内容如下：

```
void  
ForkHandler()  
{  
    DEBUG('S', "System call Fork\n");  
    // get func ptr, also regarded as new PC for child thread  
    int addr = machine->ReadRegister(4);  
  
    // update vm and initialize new space  
    currentThread->space->WriteBackAll();  
    OpenFile* executable = new OpenFile(currentThread->space->exeSector);  
    AddrSpace* addrSpace = new AddrSpace(executable);  
    delete executable;  
    // addrSpace->CopyTable(currentThread->space);  
  
    DEBUG('S', "---flush virtual memory from %s to %s---\n", currentThread->space->VMName, addrSpace->VMName);  
    OpenFile* vm = fileSystem->Open(currentThread->space->VMName);  
    OpenFile* nextvm = fileSystem->Open(addrSpace->VMName);  
    ASSERT(vm != NULL && nextvm != NULL);  
    int size = currentThread->space->GetNumPages()*PageSize;  
    char* buf = new char[size];  
    vm->Read(buf, size);  
    nextvm->Write(buf, size);  
    delete vm; delete nextvm;  
    DEBUG('S', "---finish flush virtual memory---\n");  
  
    // initialize info  
    ForkInfo* info = new ForkInfo;  
    info->addrSpace = addrSpace;  
    info->PC = addr;  
  
    // execute fork function  
    Thread* thread = new Thread("created by Fork Syscall");  
    thread->Fork(ForkFunc, (void*)info);  
  
    // copy open-file table  
    DEBUG('S', "---copy---\n");  
    tableLock->Acquire();  
    ASSERT(fileSystem->CopyFile(thread->getTID()));  
    tableLock->Release();  
    DEBUG('S', "---finish copy---\n");  
    // increase PC  
    IncreasePC();  
}
```

现在我们已经实现了所有关于线程的系统调用，ExceptionHandler 中涉及系统调用的部分内容如下。

```

        if (which == SyscallException)
        {
            if(type == SC_Halt)
            {
                DEBUG('a', "Shutdown, initiated by user program.\n");
                TLBMissRate();
            #ifdef USER_PROGRAM
                if(currentThread->space != NULL)
                {
# ifdef USE_BITMAP
                    machine->freeMem();
# endif
                    delete currentThread->space;
                    currentThread->space = NULL;
                }
            #endif
                interrupt->Halt();
            }
            else if(type == SC_Exit)
                ExitHandler();
            else if(type == SC_Create)
                CreateHandler();
            else if(type == SC_Open)
                OpenHandler();
            else if(type == SC_Read)
                ReadHandler();
            else if(type == SC_Write)
                WriteHandler();
            else if(type == SC_Close)
                CloseHandler();
            else if(type == SC_Exec)
                ExecHandler();
            else if(type == SC_Join)
                JoinHandler();
            else if(type == SC_Fork)
                ForkHandler();
            else if(type == SC_Yield)
                YieldHandler();
        }
    }
}

```

## Exercise5

这一部分要求我们对上面实现的线程相关的系统调用进行测试。

由于 Fork 和 Exec 都涉及到新线程的创建，如果放在一起测试，输出结果会比较混乱，所以我编写了两份测试代码。

第一份用于测试 Exec、Join、Yield、Exit。这份代码的内容很简单，代码首先使用 Exec 创建一个线程，接下来调用 Join 等待线程运行完毕，再以获取到的新线程的退出状态退出。

```

#include "syscall.h"
|
int main()
{
    char name[8];
    SpaceId TID;
    int status;

    name[0] = 'm'; name[1] = 'a'; name[2] = 't'; name[3] = 'm';
    name[4] = 'u'; name[5] = 'l'; name[6] = 't'; name[7] = '\0';

    TID = Exec(name);
    status = Join(TID);
    Exit(status);
}

```

上面的测试程序中，用于 Exec 的可执行文件为 matmult.c 生成的可执行文件。原始代码的内容如下。可以看到，这个函数会进行矩阵运算，且在内层循环中达到一定次数便会调用 Yield 让出 CPU。

```
#include "syscall.h"

#define Dim      16

int A[Dim][Dim];
int B[Dim][Dim];
int C[Dim][Dim];

int
main()
{
    int i, j, k;
    int cnt=0;
    char test[6];
    test[0] = 't'; test[1] = 'e'; test[2] = 's'; test[3] = 't';
    test[4] = '\n'; test[5] = '\0';

    for (i = 0; i < Dim; i++) /* first initialize the matrices */
        for (j = 0; j < Dim; j++)
        {
            A[i][j] = i;
            B[i][j] = j;
            C[i][j] = 0;

            cnt++;
            if(cnt % 128 == 0)
            {
                Yield();
                Write(test, 6, ConsoleOutput);
            }
        }

    for (i = 0; i < Dim; i++) /* then multiply them together */
        for (j = 0; j < Dim; j++)
            for (k = 0; k < Dim; k++)
            {
                C[i][j] += A[i][k] * B[k][j];

                cnt++;
                if(cnt % 128 == 0)
                {
                    Yield();
                    Write(test, 6, ConsoleOutput);
                }
            }

    Exit(C[Dim-1][Dim-1]); /* and then we're done */
}
```

为了连续执行，我编写了一段 shell 脚本。

```
#!/bin.sh

cd ..

echo "===" clean file system ==="
./nachos -V -f

echo ""
echo "===" move executable file to file system ==="
./nachos -V -cp ../test/testThread testThread
./nachos -V -cp ../test/matmult matmult

echo ""
echo "===" show file system contents ==="
./nachos -V -D

echo ""
echo "===" try to run test program ==="
./nachos -V -x testThread -d S

echo ""
echo "===" show file system contents"
./nachos -V -D
```

执行上述脚本后得到的运行结果如下。









可以看到，我们确实可以通过 `Exec` 创建一个新线程并让两个线程并行运行（出现了 2 个 `TID` 和 2 个 `Exit` 状态）。此外，当 `matmult` 所在线程每次调用 `Yield` 后，也确实会进入函数并切换线程（由于中间的 `Join` 等待输出过于频繁，所以我在结果中略去了）。而主线程也确实是在 `matmult` 线程结束之后，退出 `Join`（`1` 先于 `0` 输出 `Exit` 状态），且 `Join` 的返回值也确实是等待线程的退出状态（`Exit` 状态相同）。综上所述，可以认为上述系统调用的实现可以达到我们预期的功能。

第二份代码主要用于测试 Fork 的实现。代码首先创建一个文件，并把它储存到全局变量中，然后 main 对全局的字符数组赋值，并写入刚才创建的文件。接下来，main 在 Fork 前后分别对全局字符串进行修改。

Fork 之后，函数会继续使用刚才的描述符，写入全局字符串中的内容。然后，Fork 线程新建另外的文件，打开并写入全局字符串的内容。

```
#include "syscall.h"

char str[5];
int fd;

void func()
{
    int fd1;
    char name[10];
    name[0] = 't'; name[1] = 'e'; name[2] = 's'; name[3] = 't'; name[4] = '2';
    name[5] = '.'; name[6] = 't'; name[7] = 'x'; name[8] = 't'; name[9] = '\0';
    Write(str, 4, fd);

    Create(name);
    fd1 = Open(name);
    Write(str, 4, fd1);
    Exit(fd);
}

void main()
{
    char name[10];
    name[0] = 't'; name[1] = 'e'; name[2] = 's', name[3] = 't'; name[4] = '1';
    name[5] = '.'; name[6] = 't'; name[7] = 'x'; name[8] = 't'; name[9] = '\0';

    str[0] = 't'; str[1] = 'e'; str[2] = 's'; str[3] = 't'; str[4] = '\0';
    Create(name);
    fd = Open(name);
    Write(str, 4, fd);
    str[0] = 'T';
    Fork(func);
    str[1] = 'E';
    Exit(fd);
}
```

同样，我也编写了一段 shell 脚本来运行上述程序。

```
#!/bin/sh

cd ../

echo "==== clean file system ==="
./nachos -V -f

echo ""
echo "==== move executable file to file system ==="
./nachos -V -cp ./test/testFork testFork

echo ""
echo "==== show file system contents ==="
./nachos -V -D

echo ""
echo "==== try to run test program ==="
./nachos -V -x testFork -d S

echo ""
echo "==== show file system contents"
./nachos -V -D
```

运行结果如下。



```
---finish copy---
*****User program exit with status 2*****
close opened files before user program exit

-----open file table-----

current TID is 0
file ID: 3, owner's TID is 1, fork id is 2

-----
Try to write file
Start to create new file, addr 6d8
file name is test2.txt
Try to open file
Try to write file
*****User program exit with status 2*****
close opened files before user program exit

-----open file table-----

current TID is 1

-----
==== show file system contents
Bit map file header:
----- FileHeader contents -----
File Extension: BHdr
Create Time: Wed Dec 30 01:40:50 2020
Last Visit Time: Wed Dec 30 01:40:50 2020
Last Modify Time: Wed Dec 30 01:40:50 2020
File size: 128. File blocks:
```

可以看到，我们可以通过 Fork 创建一个新线程，并且新线程在运行初始拥有与原始线程相同的地址空间内容（可以获取全局变量在 Fork 之前的改变结果，详见最后两个文件的内容），Fork 之后，两个线程之间的操作互不干扰（Fork 之后原始线程对全局字符

串的修改，**Fork** 得到的线程感知不到）。而且，**Fork** 之后，即使原始线程先退出，**Fork** 生成的线程也可以继承文件标识符，对那个文件进行操作。最后，两个线程之间相互无关的内容也可以正常地完成（**Fork** 线程可以继续开启文件，使用局部变量，并进行文件操作）。因此，可以认为，上述 **Fork** 的实现是正确的。

## 内容二：遇到的困难以及收获

1. 对于 **MIPS** 寄存器的使用。由于我之前没有接触过 **MIPS** 体系结构，因此在实验初始进行寄存器操作时并不是很熟练。在反复查阅手册和介绍文档之后，我对寄存器（参数寄存器、返回值寄存器、PC&NextPC）也有了更深刻的记忆，使用起来也更加熟悉。

2. 打开文件表机制的设计。在完成文件系统相关的系统调用时，简单地只使用描述符并不能很好地处理自动关闭文件的问题（尤其是未主动关闭时的文件会在系统运行中持续打开下去）。后来，在回顾课堂知识之后，我考虑到采取类似于打开文件表的机制来维护用户级线程的文件使用情况。但是，由于时间有限，我只采用了 **Descriptor---OpenFile** 的二级结构。这种简化机制完全可以提供保护、追踪等功能，但是会和 **Linux** 的三级文件表结构的实际结果有所出入。

3. **AddrSpace** 的扩展与操作。在实现 **Fork** 时，我在地址空间的拷贝和写回上也遇到了一些问题。在实验的最开始，我试图不添加任何新成员来完成 **Fork** 对地址空间操作的要求，但是在尝试很久之后，我发现，我没办法获取到原始的可执行文件内容，而且在处理函数中调用循环进行写回会让本就繁杂的处理函数可读性进一步下降，因此我在 **AddrSpace** 类中相应地添加了成员和方法。此外，**Fork** 对于地址空间“相同”的要求，导致了地址空间操作的复杂度增加。这也花费了我很多时间去搞清地址空间操作的流程，以及一些实现上的细节问题（某些判断条件等）。

4. 对于 **Fork** 的实现。**Fork** 的实现中涉及到地址空间的拷贝，和打开文件的复制。这两个核心操作都让我在原有的实现上进行了大幅调整，花费了不少时间。

5. 互斥锁的引入。注意到，系统调用中使用了定义在 **system.cc** 中的全局变量，这些全局变量可以让不同线程同时获取，因此我们要提供互斥机制。互斥机

制既可以通过信号量来实现，在目前的 Nachos 实现中，实际上也可以通过手动的开关中断来实现，因为现在的 Nachos 只有时钟中断会被动地进行线程调度。但是由于头文件引用的限制，确定应该在哪里添加互斥锁也花费了我一些时间。

### 内容三：对课程或 Lab 的意见和建议

希望可以把引导手册进一步完善，现在每次阅读代码的时候需要我们递归阅读和寻找其他额外的代码，但我们往往不知道应该去哪里能找到，这样浪费了许多不必要的空间。我觉得可以把某些常用函数（或者重要辅助函数）的位置标注出来，这样可以节省遍历搜索的时间。

也希望可以为调研类的作业提供更多的提示和说明。现在我们只能漫无目的地去遍历搜索百度、google 的结果，我们无法确认博客等网站表述内容的正确性和严谨性，而且有的调研题目很难找到相关博客。让我们在短时间内查询所有体系结构/操作系统的手册，对于还有许多其他课程的本科生来说在时间上不是很现实。所以希望课程可以提供更多相关资源，或者提示我们应该去看手册的哪些章节，去找什么样的权威资料。这样既可以让我们学到准确的知识，也可以提升我们的学习效率。

### 内容四：参考文献

1. MIPS 寄存器介绍：

<http://www.cs.uwm.edu/classes/cs315/Bacon/Lecture/HTML/ch05s03.html>

2. MIPS 汇编语法：

<http://www.cs.uwm.edu/classes/cs315/Bacon/Lecture/HTML/ch05s04.html>

3. MIPS 汇编语言 SYSCALL 指令的用法：

<https://blog.csdn.net/fmup20115412/article/details/13170767>

4. 【MIPS 汇编】ADDI, ADDIU, ADD, ADDU 的区别、有符号无符号的谬误：[https://blog.csdn.net/sinat\\_42483341/article/details/89511856](https://blog.csdn.net/sinat_42483341/article/details/89511856)

5. Microsoft Word - MIPS\_IR.doc：

[https://www.eit.lth.se/fileadmin/eit/courses/edt621/MIPS\\_IR.pdf](https://www.eit.lth.se/fileadmin/eit/courses/edt621/MIPS_IR.pdf)