Lab6 系统调用的实现

Nachos系统调用实现的相关原理

1. nachos给用户程序提供了11个系统调用，函数定义在`syscall.h`中，提供给用户程序的结构为汇编语言编写，位于`start.s`中，因此用户程序在编译时需要将`start.s`源代码链接到真正的用户代码前，才能完成的使用系统调用。
2. 在nachos模拟器中，`mipssim.cc`中的`Machine::Run()`函数，逐条从内存中取出指令，并解码运行，当发现指令为`Syscall`指令时， 则调用`RaiseException()`,使用异常处理函数进入`Exception.cc::ExceptionHandler()`函数中。
3. 函数`ExceptionHandler()`解析异常类型，分别处理缺页异常，系统调用等情况，并返回。

注意事项

1. 用户传入的整形或者指针数据均在`r4,r5,r6,r7`寄存器中，但是对于指针数据来说，其地址是用户模式下的用户虚拟地址，在测试用例中就是`0x180`类似的地址，在nachos内核中该地址是非法的，因此需要将该地址按照进程的页表转换为实际的内存地址，为此编写如下转换函数，为了防止地址转换出现缺页异常，先对该地址进行读操作

```
int translateAddr(int vaddr) {
    int faddr = 0;
    machine->ReadMem(vaddr, 4, &faddr);
    machine->Translate(vaddr, &faddr, 4, FALSE);
    return machine->mainMemory + faddr;
}
```

2. 在`Machine::OneInstruction`函数中，其处理系统调用之后，直接返回，并没有对PC寄存器进行相关的更新，这回导致在系统调用函数处无线循环，因此将`case OP_SYSCALL:`下的`return`改为`break`，考虑到系统调用都是请求系统完成功能，功能完成则该条指令执行完毕，这样处理是没问题的

文件系统相关调用

文件系统相关的系统调用较为简单，对于文件的创建，打开关闭，读取写入五个操作，在`FileSystem`类中已经进行过一次封装，这里只需要正确的读出用户传入的参数，并按照相应的指令执行即可。由于在之前的文件系统实现中，文件系统创建后返还的是一个打开的结构体`OpenFile`，这里就直接将`OpenFile`结构体指针地址当做该文件的标识符，用户打开后返回，用户读取时传入，代码示例如下，完整代码附加于最后：

```
case SC_Write: {
    /*获取文件名*/
    char* name = (char*) translateAddr(machine->ReadRegister(4));
    int size = machine->ReadRegister(5);
    /*获取OpenFile结构体指针*/
    OpenFile* file = (OpenFile*) machine->ReadRegister(6);
    /*写操作*/
    fileSystem->fwrite(file, name, size);
    break;
}
```

线程相关系统调用

线程相关的系统调用较为复杂，需要考虑到用户程序的执行流程是逐条指令的翻译执行，其内存地址和nachos内核的内存地址是相互独立不可见的，在nachos内核中很难再程序执行中间改变程序走向。因此需要在执行用户程序的进程创建之初就设定好相关的信息。

1. `Exec`函数较为简单，创建新的线程并执行一个新的用户程序，将新进程的`tid`写回`r2`寄存器即可
2. `Yield`函数较为简单，直接执行`Thread->Yield()`函数即可
3. `Exit`函数同样较为简单，为了方便调试，打印`exitCode`并终止当前线程即可
4. `Fork`函数相对较为复杂，该函数要求新的线程`fork`在当前线程`current`地址空间内执行，并且需要执行指定的函数，因此作出如下处理，
    - 针对当前地址空间要求，对每一个页表项增加标志位`code_data`，代表当前页表项下的内存中保存的是否为程序加载时读取的永久性数据，对于这些数据，应该让两个线程共享同一份，而对于栈空间，为运行时内存，则需要单独使用。因此创建新的`AddrSpace(Thread*)`函数，在创建页表时只拷贝代码和数据相关的内存映射，其余不予拷贝，留待运行时分配。
    - 针对需要执行指定函数的要求，作出如下修改。创建一个新的函数为`startForkProcess(addr)`，其作用类似于`progtest.cc::startProgess(name)`函数，该函数针对传入的地址，设置最初的`PC`寄存器值为指定位置，并调用`machine::RUn()`函数执行，以达直接执行指定函数的目的。
    - 而在系统调用的处理阶段，则需要手动为`fork`线程分配空间，剩余的`PC`寄存器的设置在程序执行最初初始化时设置，相关代码如下：

```
void StartForkProcess(int func) {

    currentThread->space->InitRegisters(); // set the initial register values
    currentThread->space->RestoreState();  // load page table register
    currentThread->space->setPC(func);
    machine->Run(); // jump to the user progam
    ASSERT(FALSE);  // machine->Run never returns;
}
/*  fork 系统调用*/
case SC_Fork: {
    int func = machine->ReadRegister(4);
    Thread* fork = new Thread("fork");
    fork->StackAllocate(StartForkProcess, func);
    AddrSpace* newspace = new AddrSpace(currentThread);
    fork->space = newspace;
    printf("forking func addr %x in thread : %d\n", func, currentThread-
>getTid());
    scheduler->ReadyToRun(fork);
    break;
}
```

5. `join`函数要求线程等待指定的线程结束，并获得其`exit code`因此添加如下数据结构的支持
    - 在`Thread`结构体中添加变量`exit code`，当等待的线程退出后，内核将其`exit code`保存至本线程的`exit code`变量中
    - 在`Thread`结构体中添加`waitingList`变量，维护等待当前进程结束的进程地址
      然后在`join`函数调用时，将当前进程加入指定进程的`waitingList`中，并`Sleep()`。在某个线程调用`Exit`时，检查该线程的`waitingList`，唤醒其中的额所有线程并设置对应线程的`exit code`为当前的`exit code`，实现如下：

```
case SC_Join: {
    int tid = machine->ReadRegister(4);
    Thread* waitFor = scheduler->getThreadByTid(tid);
    waitFor->waitingList->Append((void*) currentThread);
    printf("thread %d join thread %d\n", currentThread->getTid(),
            waitFor->getTid());
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    currentThread->Sleep();
    interrupt->SetLevel(oldLevel);
    machine->WriteRegister(2, currentThread->exitCode);
    break;
}
```

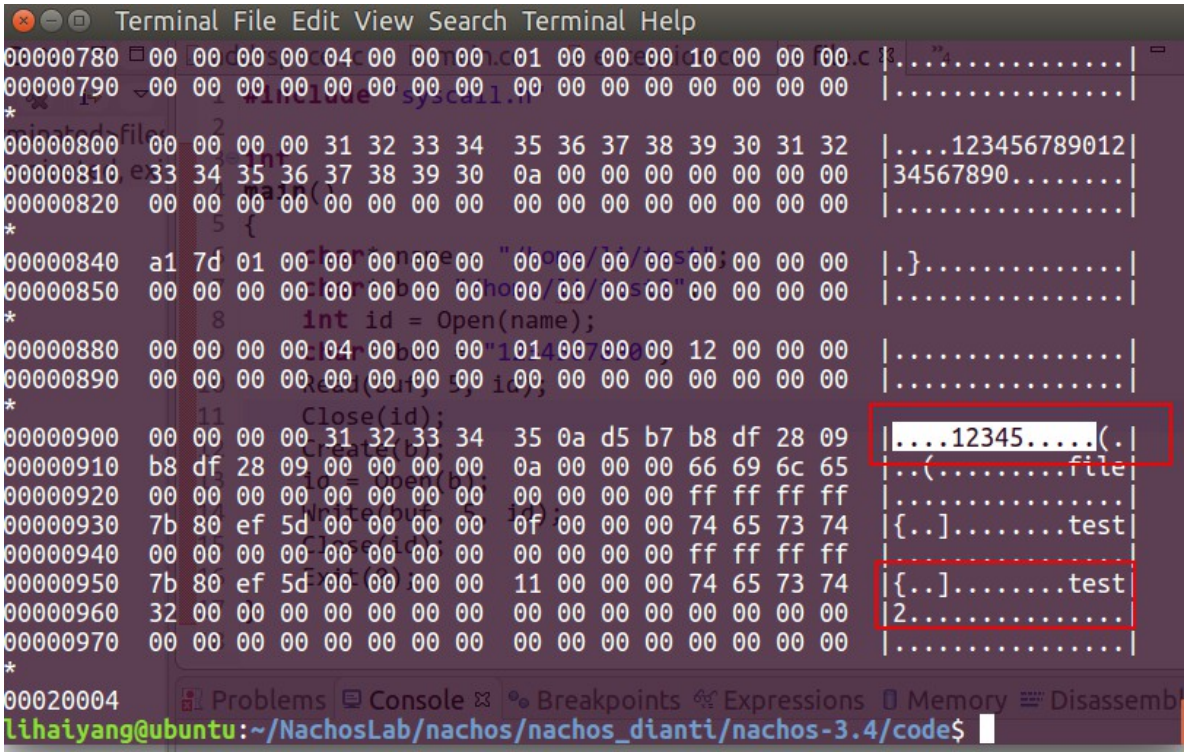
测试程序：

测试中，首先将测试程序拷贝至nachos磁盘中，然后执行对应的测试程序如下：

1. 文件测试:

```
#include "syscall.h"
int main()
{
    char* name = "/home/li/test";
    char* b = "/home/li/test2";
    int id = Open(name);
    char* buf = "1234567890";
    Read(buf, 5, id);
    Close(id);
    Create(b);
    id = Open(b);
    Write(buf, 5, id);
    Close(id);
    Exit(0);
}
```

测试结果如下：文件成功拷贝完成，并写入磁盘



2. 线程测试，测试程序包含两个程序，代码如下

```
/*程序1*/
#include "syscall.h"
int main(){
    char* name = "/home/li/thread2";
    int exitCode = -2;
    SpaceId id = Exec(name);
    exitCode = Join(id);
    Exit(exitCode);
}
/*程序2*/
#include "syscall.h"
int all[3] = {1,2,3};
int i,j;
void fork(){
    for(i = 0; i < 2; i ++){
        all[i] = all[i] * all[i+1];
    }
    Exit(all[1]);
}
int main(){
    for(j = 0; j < 2; j ++){
        all[j] = all[j] + all[j+1];
    }
    Fork(fork);
    Yield();
    Exit(all[0]);
}
```

测试结果如下：可以看到，两个程序都能够正确执行，fork操作也正确的在同一个变量上作出 了更改

```
<terminated> (exit value: 0) filesys [C/C++ Application] /home/lihaiyang/NachosLab/nachos/nachos_dianti/nachos-3.4/
Hello world
execing prog /home/li/thread2, tid is :2
thread 1 join thread 2
forking func addr d0 in thread : 2
yield thread : 2
thread : 3 is exit , Exit code is 15
thread : 2 is exit , Exit code is 15
thread : 1 is exit , Exit code is 15
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 909391, idle 902820, system 6290, user 281
Disk I/O: reads 137, writes 69
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

代码

```
/*exception.cc*/
void SyscallHandler(int type) {
    switch (type) {
    case SC_Halt: {
        DEBUG('a', "Shutdown, initiated by user program.\n");
        interrupt->Halt();
        break;
    }
    case SC_Exit: {
        printf("thread : %d is exit , Exit code is %d\n",currentThread-
>getTid(),
                machine->ReadRegister(4));

        IntStatus oldLevel = interrupt->SetLevel(IntOff);
        while (!currentThread->waitingList->IsEmpty()) {
            Thread *t = (Thread *) currentThread->waitingList->Remove();
            if (t != NULL) {
                t->exitCode = machine->ReadRegister(4);
                scheduler->ReadyToRun(t);
            }
        }
        interrupt->SetLevel(oldLevel);

        currentThread->Finish();
        break;
    }
    case SC_Exec: {
        char* name = (char*) translateAddr(machine->ReadRegister(4));
        Thread* exec = new Thread("thread2");
        exec->Fork(StartProcess, (void*) name);
        machine->WriteRegister(2, exec->getTid());
        printf("execing prog %s, tid is :%d\n", name, exec->getTid());
        break;
    }
    case SC_Fork: {
        int func = machine->ReadRegister(4);

        Thread* fork = new Thread("fork");
```

```
        fork->StackAllocate(StartForkProcess, func);

        AddrSpace* newspace = new AddrSpace(currentThread);
        fork->space = newspace;

        printf("forking func addr %x in thread : %d\n", func, currentThread-
>getTid());
        scheduler->ReadyToRun(fork);
        break;
    }
    case SC_Yield: {
        printf("yield thread : %d\n", currentThread->getTid());
        currentThread->Yield();
        break;
    }
    case SC_Join: {
        int tid = machine->ReadRegister(4);
        Thread* waitFor = scheduler->getThreadByTid(tid);
        waitFor->waitingList->Append((void*) currentThread);

        printf("thread %d join thread %d\n", currentThread->getTid(),
                waitFor->getTid());

        IntStatus oldLevel = interrupt->SetLevel(IntOff);
        currentThread->Sleep();
        interrupt->SetLevel(oldLevel);

        machine->WriteRegister(2, currentThread->exitCode);
        break;
    }
    case SC_Create: {
        char* name = (char*) translateAddr(machine->ReadRegister(4));
        fileSystem->Create(name, 0, TRUE);
        break;
    }
    case SC_Open: {
        char* name = (char*) translateAddr(machine->ReadRegister(4));
        OpenFile* openfile = fileSystem->Open(name);
        machine->WriteRegister(2, (int) openfile);
        break;
    }
    case SC_Close: {
        int fileAddr = machine->ReadRegister(4);
        fileSystem->Close((OpenFile*) fileAddr);
        break;
    }
    case SC_Write: {
        char* name = (char*) translateAddr(machine->ReadRegister(4));
        int size = machine->ReadRegister(5);
        OpenFile* file = (OpenFile*) machine->ReadRegister(6);

        fileSystem->fwrite(file, name, size);
        break;
    }
    case SC_Read: {
        char* name = (char*) translateAddr(machine->ReadRegister(4));
        int size = machine->ReadRegister(5);
```

```
        OpenFile* file = (OpenFile*) machine->ReadRegister(6);

        int num = fileSystem->fread(file, name, size);
        machine->WriteRegister(2, num);
        break;
    }
    default:{
        printf("wrong syscall type %d\n", type);
        ASSERT(FALSE);
    }
    }

}
```