CS342301: Operating System
MP1: System Call
Team Number: 71

Team Members & Contributions

- Members: 108022138 楊宗諺、111062649 許峻源
- Contributions: 合力製作MP1 report、討論四個I/O system calls的implementation details

Trace Codes

1. **SC_Halt**

Machine::Run()

- 當系統執行User program的時候，會先執行mipssim.cc中的Machine::Run()函數。
- 執行User program的時候需要在user mode下執行，Run()函數中的setStatus(UserMode)可以將處理器的status轉為user mode。
- 物件Instruction指標instr可以用來儲存instruction的相關資訊，instr會被當作參數傳入Machine::OneInstruction()函數中。

Machine::OneInstruction()

- OneInstruction()函數主要是用來模擬instruction的執行過程，一開始會先去program counter register對應到的memory位置取出instruction，再來利用物件instr中的Decode()函數將instruction轉換成opCode，最後根據不同的opCode處理不同的動作。
- 如果處理器根據opCode發現了system call的指令，處理器會利用RaiseException()函數拋出SyscallException異常，程式碼如下：

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

Machine::RaiseException()

- RaiseException()函數會將發生exception的address存入特定的register(BadVAddrReg)中，並呼叫DelayedLoad()函數結束其他in progress的動作。
- 接著將switch to kernel mode, 呼叫ExceptioHandler() 傳入ExceptionType，呼叫結束後，再switch to user mode。

```
kernel->interrupt->setStatus(SystemMode);
ExceptionHandler(which);
kernel->interrupt->setStatus(UserMode);
```

ExceptionHandler()

- ExceptionHandler()函數會依據我們傳入的變數(which)來判斷是那一種類型的Exception。
- Exception Type定義在machine.h中。

```
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
};
```

- 在知道是SyscallException的情況下，使用從 Register 2 中讀取到的 Syscall Type來判斷是那一種system call。

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

- 呼叫SysHalt()

```
case SC_Halt:
    DEBUG(dbgSys, "Shutdown, initiated by user program.\n");
    SysHalt();
    cout<<"in exception\n";
    ASSERTNOTREACHED();
```

SysHalt()

- 呼叫SysHalt() 後，隨即跳入Interrupt.cc中的Halt()。

Interrupt::Halt()

- 在Halt() 函數中，print 現在kernel 的states。
- 最後delete kernel終止程式。

```
Interrupt::Halt()
{
    cout << "Machine halting!\n\n";
    cout << "This is halt\n";
    kernel->stats->Print();
    delete kernel;      // Never returns.
}
```

2. SC_Create

ExceptionHandler()

- 同SC_Halt()一樣, SC_Create 也是從Run()到呼叫RaiseException()再到ExceptionHandler()，根據Exception Type及 Syscall Type呼叫SysCreate()。
- 將Argument 1（filename），從Register 4中讀出，傳入SysCreate()中。

```
val = kernel->machine->ReadRegister(4);
{
char *filename = &(kernel->machine->mainMemory[val]);
```

SysCreate()

- 呼叫SysCreate()後，跳入filesys.h的Create()函數，並將filename傳入

```
int SysCreate(char *filename)
{
        // return value
        // 1: success
        // 0: failed
        return kernel->fileSystem->Create(filename);
}
```

Create()

- Create()函數會呼叫sysdep.h中的OpenForWrite()函數

```
bool Create(char *name) {
    int fileDescriptor = OpenForWrite(name);

    if (fileDescriptor == -1) return FALSE;
    Close(fileDescriptor);
    return TRUE;
}
```

- 而OpenForWrite()函數則是呼叫C的原生函數Open()，來create 一個新的file

```
int
OpenForWrite(char *name)
{
    int fd = open(name, O_RDWR|O_CREAT|O_TRUNC, 0666);

    ASSERT(fd >= 0);
    return fd;
}
```

3. SC_PrintInt

ExceptionHandler()

- 這邊Handle SyscallException的方式跟前面呼叫system call的方式相同，只是system call的type不同（SC_PrintInt）。
- 從register 4讀出要print的integer，並把其當作參數傳入ksyscall.h中的SysPrintInt()函數內。

SysPrintInt()

- 此函數的功用是用來呼叫synchconsole.cc中的PutInt()函數，並準備進行asynchronized I/O。

SynchConsoleOutput::PutChar()、PutInt()

- 我們可以先用PutChar()函數了解asynchronized I/O的運作方式，因為PutInt()函數只是將要print的integer分成single digit，然後一個一個digit利用PutChar()函數的概念輸出。
- PutChar()一開始會先利用class Lock中所定義的Acquire()函數先去取得writer，如果沒有閒置的writer的話則需要等待。

- 取得writer之後會去呼叫console.cc中的ConsoleOutput::PutChar()函數。
- 這邊稍微提到一下，ConsoleOutput::PutChar()函數會登記一個interrupt。此interrupt會在print完要print的character之後呼叫登記此interrupt物件的CallBack()函數。
- waitFor->P()函數會用來等待前一個要print的character print完之後再繼續執行，在Semaphore::P()函數的內容中也可以發現有一個while loop在重複執行直到Semaphore是available的狀態。
- 從waitFor->P()函數回來之後，代表說print的工作完成了，接下來會利用class Lock中定義的Release()函數把lock住的writer釋放掉，讓其他需要writer的thread可以來使用。

ConsoleOutput::PutChar()

- 一開始會先確認此ConsoleOutput的物件是否處於busy狀態，如果不是處於busy狀態則繼續執行。
- 透過sysdep.cc中的WriteFile()函數將參數character寫入指定的file裡面或stdout，並把ConsoleOutput的狀態設為busy。
- 透過interrupt.cc中的Interrupt::Schedule()函數schedule一個interrupt，此interrupt會在output完指定的character之後呼叫登記此interrupt物件的CallBack()函數。

Interrupt::Schedule()

- Schedule()函數會傳入三種參數，分別為要呼叫CallBack()函數的物件指標（CallBackObj* toCall）、動作的執行時間（int fromNow）以及產生interrupt的硬體設備（IntType type）。
- 變數when儲存動作預計的完成時間（interrupt預計要被呼叫的時間）。
- SortedList<PendingInterrupt*>* pending可以用來儲存正在pending的interrupt，Schedule()函數會根據class PendingInterrupt來建立一個PendingInterrupt的物件（利用參數toCall、when以及type來建立），並把建立好的物件insert進去pending list，而其會在未來預計要被呼叫的時間點被呼叫執行。

Machine::Run()

- 當schedule好interrupt的執行時間，就只要等待其執行時間到。
- Run()函數在執行完OneInstruction()函數之後就會呼叫在interrupt.cc中的Interrupt::OneTick()函數。

Interrupt::OneTick()

- OneTick()函數的主要功能是讓NachOS可以模擬時間往前的行為，透過呼叫OneTick()函數可以讓系統時間往前一個時刻。而根據處理器不同的status，OneTick()函數的時間變化量可能會有所不同。
- 除了讓時間往前以外，OneTick()函數在讓時間往前之後會呼叫同在interrupt.cc中的Interrupt::CheckIfDue()函數來確認說在移動過後的時間點有沒有任何interrupt的預計呼叫時間到了，如果到了就要被handle。
- OneTick()函數同時也可以處理Context switch的情況。

Interrupt::CheckIfDue()

- 此函數是用來確認在當前時間點有沒有任何在pending list裡面的interrupt需要被handle，如果有就依序handle並回傳true，沒有則回傳false。
- 在Handle interrupt的時候，會呼叫當初建立PendingInterrupt物件時傳入參數toCall的CallBack()函數，在本例中會是ConsoleOutput::CallBack()函數。

ConsoleOutput::CallBack()

- ConsoleOutput::CallBack()函數會將output設備的狀態從busy轉為idle，因為已經執行完print的工作了。
- ConsoleOutput::CallBack()函數中還有一個CallBack()函數，是用來連接SynchConsoleOutput中的CallBack()函數。

SynchConsoleOutput::CallBack()

- 此函數會呼叫waitFor->V()函數，waifFor->V()函數可以將當初在waitFor->P()函數中被put sleep的thread轉為ready狀態，讓其可以繼續執行。
- waitFor->V()函數會將value加1，因此原本在waitFor->P()函數中重複執行的while loop就會跳出來，然後return function到SynchConsoleOutput::PutChar()或PutInt()函數內，接著繼續執行之後的statement。如果是PutChar()函數的話就會呼叫Lock::Release()函數釋放掉被lock住的writer，而PutInt()函數就會繼續print下一個digit或是一樣release掉被lock住的writer。

Verification

- **fileIO_test1**

```
[os22team71@localhost test]$ ../build.linux/nachos -e fileIO_test1
fileIO_test1
Success on creating file1.test
Machine halting!

This is halt
Ticks: total 954, idle 0, system 130, user 824
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
```

- **fileIO_test2**

```
[os22team71@localhost test]$ ../build.linux/nachos -e fileIO_test2
fileIO_test2
Passed! ^_^
Machine halting!

This is halt
Ticks: total 815, idle 0, system 120, user 695
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
```

Implementation Details

- **Start.S**

  為Open, Read, Write, Close創建 syscall stubs

```
        .globl Open
        .ent    Open
Open:
        addiu $2,$0,SC_Open
        syscall
        j       $31
        .end Open
```

- **Syscall.h**

  取消define syscall 代碼的註解

```
#define SC_Open         6       // MP_1
#define SC_Read         7       //
#define SC_Write        8       //
#define SC_Seek         9
#define SC_Close        10      //
```

- **filesys.h**

  在class FileSystem中建立四個functions: OpenAFile, WriteFile, ReadFile, CloseFile.

  OpenAFile:

  首先呼叫OpenForReadWrite() 將回傳值存入int filedescriptor 中, 確認執行成功, 否則回傳-1, 接著用for loop檢查OpenFileTable中是否有指向NULL的entry, 如果有則創建並初始化(將filedescriptor 存入int file 中)一個新的OpenFile物件, 並將entry 指向新的OpenFile物件, 並回傳此entry的index,而如果沒有空entry則表示達到開啟檔案的上限, 回傳 -1表示開啟檔案失敗。

```
OpenFileId OpenAFile(char *name) {
    int fileDescriptor = OpenForReadWrite(name, FALSE);
    if(fileDescriptor == -1) return -1;
    for(int i=0; i<20; i++){
            if(OpenFileTable[i] == NULL){
                    OpenFileTable[i] = new OpenFile(fileDescriptor);
                    return i;
            }
    }
    return -1;
}
```

  WriteFile:

  確認輸入的Id在有效範圍內後, 呼叫OpenFile物件的內建函數Write() 將其回傳值(寫入了多少的bytes) 存入int sz中, 確認sz 與要寫入的size 大小一致, 否則為寫入失敗回傳-1, 成功則回傳sz 也就是實際寫入大小。

```
int WriteFile(char *buffer, int size, OpenFileId id){
    if(id<0 || id>19 || OpenFileTable[id] == NULL) return -1;
    int sz = OpenFileTable[id]->Write(buffer, size);
    if(size != sz) return -1;
    return sz;
}
```

  ReadFile:

  確認輸入的Id在有效範圍內後, 呼叫OpenFile物件的內建函數Read() 將其回傳值(讀取了多少的bytes) 存入int sz中, 確認sz 與讀取的size 大小一致, 否則為讀取失敗回傳-1, 成功則回傳sz 也就是實際讀取大小 。

```
int ReadFile(char *buffer, int size, OpenFileId id){
    if(id<0 || id>19 || OpenFileTable[id] == NULL) return -1;
    int sz = OpenFileTable[id]->Read(buffer, size);
    if(size != sz) return -1;
    return sz;
}
```

CloseFile:

確認輸入的Id在有效範圍內後, delete掉此entry指向的OpenFile物件, 並將此entry指向NULL, 成功關閉檔案回傳1。

```
int CloseFile(OpenFileId id){
  if(id<0 || id>19 || OpenFileTable[id] == NULL) return -1;
  delete OpenFileTable[id];
  OpenFileTable[id] = NULL;
  return 1;
}
```

- **ksyscall.h**

  創建 SysOpen, SysWrite, SysRead, SysClose 四個functions, 並分別呼叫 kernel->fileSystem中我們建立的四個functions: OpenAFile, WriteFile, ReadFile, CloseFile。

```
OpenFileId SysOpen(char *name)
{
        return kernel->fileSystem->OpenAFile(name);
}
int SysClose(int id)
{
        return kernel->fileSystem->CloseFile(id);
}
int SysWrite(char *buffer, int size, int id)
{
        return kernel->fileSystem->WriteFile(buffer, size, id);
}
int SysRead(char *buffer, int size, int id) {
        return kernel->fileSystem->ReadFile(buffer, size, id);
}
```

- **exception.cc**

  在exceptionHandler中syscall cases內新加入四個case: SC_Open, SC_Write, SC_Read, SC_Close

  SC_Open:

  將儲在Reg 4 中的filename參數讀出, 呼叫ksyscall 中的SysOpen, 傳入filename, 將SysOpen的回傳值存入Reg 2中, 調整PreVPCRrg, PCReg, NextPCReg後回傳。

```
case SC_Open:
            val = kernel->machine->ReadRegister(4);
            filename = &(kernel->machine->mainMemory[val]);  // filename  declare before switch
            id = SysOpen(filename);
            kernel->machine->WriteRegister(2, id);
            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);
            return;
            ASSERTNOTREACHED();
break;
```

SC_Write:

將儲在Reg 4, 5, 6 中的buffer, size, id 參數讀出, 呼叫ksyscall 中的SysWrite, 傳入參數, 將SysWrite的回傳值存入Reg 2中, 調整PreVPCRrg, PCReg, NextPCReg後回傳。

```
case SC_Write:
            val = kernel->machine->ReadRegister(4);
            buffer = &(kernel->machine->mainMemory[val]);
            size =  kernel->machine->ReadRegister(5);
            id = kernel->machine->ReadRegister(6);

            size = SysWrite(buffer, size, id);
            kernel->machine->WriteRegister(2, size); // result size store in reg2
            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);
            return;
            ASSERTNOTREACHED();
```

SC_Read:

將儲在Reg 4, 5, 6 中的buffer, size, id 參數讀出, 呼叫ksyscall 中的SysRead, 傳入參數, 將SysRead的回傳值存入Reg 2中, 調整PreVPCRrg, PCReg, NextPCReg後回傳。

```
case SC_Read:
            val = kernel->machine->ReadRegister(4);
            buffer = &(kernel->machine->mainMemory[val]);
            size =  kernel->machine->ReadRegister(5);
            id = kernel->machine->ReadRegister(6);

            size = SysRead(buffer, size, id);
            kernel->machine->WriteRegister(2, size); // result size store in reg2
            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);
            return;
            ASSERTNOTREACHED();
break;
```

SC_Close:

將儲在Reg 4 中的id參數讀出, 呼叫ksyscall 中的SysClose, 傳入id, 將SysClose的回傳值存入Reg 2中, 調整PreVPCRrg, PCReg, NextPCReg後回傳。

```
case SC_Close:
            id = kernel->machine->ReadRegister(4);
            kernel->machine->WriteRegister(2, SysClose(id)); // result(-1 or 1) store in reg2
            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);
            return;
            ASSERTNOTREACHED();
break;
```

Difficulties

在這次assignment裡面, 我們花費比較多的時間在trace code上面。因為NachOS的程式碼較複雜且繁複, 所以我們花了大部分時間在理解程式碼執行的流程以及相關的class及define定義。而在I/O system call的implementation方面, 雖然花費的時間不算多, 但我們也花了一點時間在trace file operations的functions以及熟悉呼叫system call的流程, 最後才正確地實作出了四個I/O system call的functions。