Operating System HW 1 – System Call

Group 18
106061146 陳兆廷 106000147 沈永聖 106061132 黃友廷

1. Code tracing

a. Halt()

i. Machine::Run()

當 NachOS 開始運行時會去執行 Machine::Run()，去模擬在 NachOS 上執行 user program。setStatus(UserMode)在啟動時會先將 OS 的設定成 user mode，接著進入 for 迴圈去呼叫 Machine::OneInstruction()去從已經被編譯成 binary file 的 user program 中讀取 instruction 並執行。

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

    if (debug->IsEnabled('m')) {
        cout << "Starting program in thread: " << kernel->currentThread->getName();
        cout << ", at time: " << kernel->stats->totalTicks << "\n";
    }
    kernel->interrupt->setStatus(UserMode);
    for (;;) {
        OneInstruction(instr);
        kernel->interrupt->OneTick();
        if (singleStep && (runUntilTime <= kernel->stats->totalTicks))
            Debugger();
    }
}
```

ii. Machine::OneInstruction()

先由 ReadMem()去從記憶體內將 binary file 以 instruction 為單位 raw 內，接著使用 instr->Decoder()去對 raw 進行解碼，以利接下來的步驟進行判讀。

```
void Machine::OneInstruction(Instruction *instr)
{
    int raw;
    int nextLoadReg = 0;
    int nextLoadValue = 0;  // record delayed load operation, to apply
                            // in the future

    // Fetch instruction
    if (!ReadMem(registers[PCReg], 4, &raw))
    return;         // exception occurred
    instr->value = raw;
    instr->Decode();      Jhao-Ting Chen, 2 days ago • update
```

由 switch 去判讀這個 instruction 的 opcode，因為 Halt()是一個 system call，所以會進入 OP_SYSCALL 並且執行 RaiseException()。

```
    // Execute the instruction (cf. Kane's book)
    switch (instr->opCode) {

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

iii. Machine::RaiseException()

為了執行 Exception，需要先將 OS 設定成 System Mode，然後呼叫 ExceptionHandler()去分辨是哪種 exception 需要處理，執行完後再將 OS 改回 User Mode。

```
void Machine::RaiseException(ExceptionType which, int badVAddr)      You, s
{
    DEBUG(dbgMach, "Exception: " << exceptionNames[which]);

    registers[BadVAddrReg] = badVAddr;
    DelayedLoad(0, 0);          // finish anything in progress
    kernel->interrupt->setStatus(SystemMode);
//  cout << "entering system mode...\n";
    ExceptionHandler(which);        // interrupts are enabled at this point
    kernel->interrupt->setStatus(UserMode);
//  cout << "entering user mode...\n";
}
```

iv. ExceptionHandler()

區分是哪種 exception 要執行並且去呼叫對應涵式來完成，而這裡是進行 Halt()這個 system call，所以會去呼叫 kernel->interrupt->Halt()。

```
void ExceptionHandler(ExceptionType which)
{
    int type = kernel->machine->ReadRegister(2);
    int val, status;
       Jhao-Ting Chen, 2 days ago • update
    switch (which)
    {
    case SyscallException:
        switch (type)
        {
        case SC_Halt:
            DEBUG(dbgAddr, "Shutdown, initiated by user program.\n");
            kernel->interrupt->Halt();
            break;
```

v. Halt()

使用 kernel->stats->Print()去將一些系統資訊印出，接著就會將這個 kernal 刪除讓 NachOS 關閉。

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

b. Create()

i. Machine::Run()

ii. Machine::OneInstruction()

iii. Machine::RaiseException()

前三步驟與執行 Halt()時相同。

iv. ExceptionHandler()

首先區分是哪種 exception 要執行並且去呼叫對應 function 來完成，而這裡是進行 Create 這個 system call。

創建檔案時需要檔名，需要透過讀取 register 取得當時儲存的的檔案名稱；但是由於檔案名稱長度不固定，不能僅透過固定的大小的 register 傳遞資料，因此在這邊儲存的是檔案名稱儲存的"地址"，當透過 ReadRigster(4)取得地址之後，OS 還必須將虛擬地址轉成實體地址才能夠運作因此透過：

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

完成轉換的動作。最後當檔案創建結束之後，我們必須將創建的結果(成功與否)寫回 register 告知使用者結果。如下：

```
kernel->machine->WriteRegisSter(2, (int)status);
```

```
case SC_Create:
    // DEBUG(dbgSys, "Start Create.\n");
    val = kernel->machine->ReadRegister(4);
    {
        char *filename = &(kernel->machine->mainMemory[val]);
        status = kernel->fileSystem->Create(filename);
        kernel->machine->WriteRegisSter(2, (int)status);
    }
```

v. FileSystem::Create()

由於 Exception 只執行簡單的指令，而詳細的執行方法則定義在別的地方，這邊就是詳細執行 Create()的 function。從這邊可以看到當創建成功時會回傳 TRUE，失敗則

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

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

為 FALSE。

vi. Machine::OneInstruction():

```
        Jhao-Ting Chen, a week ago • update
// Compute next pc, but don't install in case there's an error or branch.
int pcAfter = registers[NextPCReg] + 4;
registers[PCReg] = registers[NextPCReg];
registers[NextPCReg] = pcAfter;
```

在這個作業中我們是使用的是 MIPS 架構，每個 instruction 大小為 4 byte，所以在 OneInstruction()裡在計算下一個 program counter(pcAfter) 時就是加 4。當該 instruction 被執行完畢後，它就會更新 program counter 讓 program counter 指向下一個 instruction。

2. System Call Implementation

透過觀察 Halt()、Create()，我們已經很熟悉要執行 system call 必定會經過以下步驟

i. machine/mipssim.cc --- Machine::Run()

ii. machine/mipssim.cc --- Machine::OneInstruction()

iii. machine/machine.cc --- Machine::RaiseException()

iv. userprog/exception.cc --- ExceptionHandler()

接著在 ExceptionHandler()當中會根據 system call 的種類做出不同的動作。

```
switch (which)
{
case SyscallException:
    switch (type)
    {
    case SC_Halt:
        DEBUG(dbgAddr, "Shutdown, initiated by user program.\n");
        kernel->interrupt->Halt();
        break;
```

以 Halt()為例，當 type 為 SC_HALT 時就會跳轉到 SC_HALT 的 case 執行功能，但是這邊基本上只會呼叫 function，而 function 的定義則是會定義在別的地方，以保持 exception 的簡短。

接著我們能夠模仿 Halt()運行的方式來執行我們的 4 個 function，新增額外的 4 個 case: SC_Open、SC_Write、SC_Read、SC_Close 並在裡面呼叫要執行的功能。

userprog/syscall.h : 定義 SC_Open、SC_Write、SC_Read、SC_Close

關於 system call 的類別的定義都寫在 syscall.h 當中，因此我們必須新增 4 個類別才能正確執行。

test/start.s : 在 register $2 中寫入 type

當我們在 fileTest2.c 執行我們的 system call 的時候，會根據我們所使用的 function 跳到其所對應的 label 位置執行相關組語，並且在這時候將 system call 的”type”寫入 register $2 當中，以 Open 為範例，剩下三個雷同。

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

接著，我們在呼叫 function 的時候會有其用到的 arguments 會依照順序分別放在 register $3、register $4 當中。

此外當我們在 ExceptionHandler()傳遞 arguments 的時候要注意兩種情況

1. 傳遞的 argument 為數值而非 address:

   可以藉由直接讀取 register 的方式獲取並使用。

```
int size = kernel->machine->ReadRegister(5);
```

2. 傳遞的 arguments 為 address:

由於這個位址為虛擬位址，因此我們必須將其轉為實體位址之後才能正確使用。

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

最後再將運行完畢的結果寫為 register 2 即可。(以 OpenAFile()為例)

```
status = kernel->fileSystem->OpenAFile(filename);
kernel->machine->WriteRegister(2, (int)status);
```

filesys/filesys.h

在 ExceptionHandler()當中我們並沒有詳細描述如何執行四個 function，因此我們必須額外定義。在觀察 **Create()**之後，我們可以知道有關於檔案的運作與管理是藉由 filesys.h 中的 FileSystem 這個 class 來做管理與操作，因此我們必須將

1. int OpenAFile(char *name)
2. int WriteFile(char *buffer, int size)
3. int ReadFile(char *buffer, int size)
4. int CloseFile()

等四個 function 定義在這個地方。首先透過觀察 FileSystem::Open()可以發現 OpenFile 這個 class 被定義為類似一個物件，它可以說是”file”本身，因此透過這四個 function 來對 OpenFile 的指標” filePtr”進行操作就可以達到管理一個檔案的目的。

int OpenAFile(char *name)

我們可以藉由 FileSystem::Open()來達成這個目的，當檔案成功開啟的時候會回傳 OpenFile 的 address，若是開啟失敗則會回傳“NULL”，因此透過判斷回傳是否為”NULL”就可以知道開啟的成功與否，再根據於此回傳狀態為”-1”或是”1”，並將回傳的 address 存在 OpenFile 的指標當中。

int WriteFile(char *buffer, int size)

OpenFile 這個 class 有定義對於”單一”個檔案寫入的操作，因此我們可以透過這個 function 就能完成寫入的動作並藉由回傳的值來決定寫入的成功與否。

```
int numsWritten = filePtr->Write(buffer, size);
```

Buffer 為要寫入的內容，size 則是寫入的大小。

int ReadFile(char *buffer, int size)

OpenFile 這個 class 有定義對於”單一”個檔案讀取的操作，因此我們可以透過這個 function 就能完成寫入的動作並藉由回傳的值來決定讀取的成功與否。

```
int numsRead = filePtr->Read(buffer, size);
```

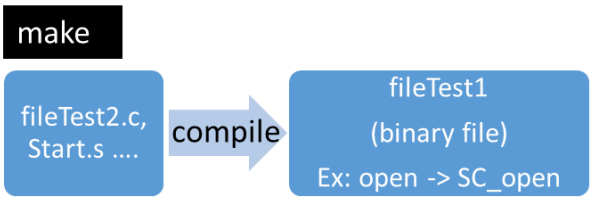
Buffer 為要讀取的內容，size 則是讀取的大小。

int CloseFile()

在這邊我們利用 lib/sysdep.cc 當中的 Close()來進行關閉的動作

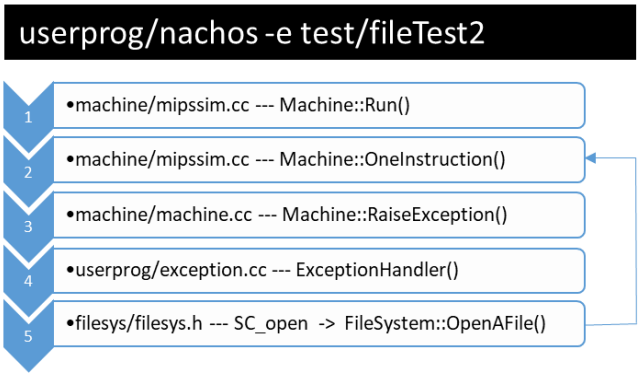
3. 作業執行流程

a.



使用 makefile 將 fileTest.c 和 Sart.s compile 成 MIPS 架構的 binary file。

b.



使用 nachos 去執行被編譯完成的 binary file (fileTest2)，會依序進行上圖所示流程。

四、小組分工

陳兆廷: coding

沈永聖、黃友廷:報告撰寫