Operating systems
MP1

Team 67

108062128 林育丞

108062135 呂佳恩

分工表

組員	內容
林育丞	報告製作與實作 Part II - 2
呂佳恩	Trace Code 與實作 Part II - 2

(a)、(b) Trace Code, explanations of system calls

(1) SC_Halt

(i) **mipssim.cc**

Machine::Run ( )

在UserMode下，不斷地執行以下兩個行為：

OneInstruction()： 處理 *User-level program*

OneTick()： 推進模擬的時間，或是處理可能到來 Pending Interrupt。

Machine::OneInstruction ( )

1. 讀取 registers[PCreg] 存的資料並解析成 Opcode。
2. 接著以龐大的 *switch-case* 架構，將 instr->Opcode 做解析，並執行對應的行為

在本題的情況下，instr->Opcode 為 OP_SYSCALL，其所對應的行為是呼叫 RaiseException()，並傳入 SyscallException。

(ii) **machine.cc**

Machine::RaiseException (ExceptionType which, int badVAddr)

在執行 *User-level program* 的時候碰到錯誤(或是例外情形)，就會呼叫這個函式。此時會記錄出錯的 *virtual address* (參數badVAddr)，接著將模式改為 SystemMode (也就是 *Kernel Mode*)，之後將錯誤名稱的參數which傳給 ExceptionHandler()。

在 ExceptionHandler() 執行完畢之後，會切換回 UserMode。

在本題的情況下，錯誤名稱的參數內容即是 SyscallException。

而出錯的 *address* 為 0 (因為這個 *Exception* 是 *System Call*，並非錯誤，所以為0)。

(iii) **exeception.cc**

ExceptionHanler (ExceptionType which)

解析傳入的Exception Type，如是 SyscallException，則根據 register[2] 內容的 Type，去執行相應的處理。

在本題的情況下，Type 為 SC_Halt，而相應處理則是呼叫 SysHalt()。

(iv) **ksyscall.h**

SysHalt ( )

SysHalt( )是作為 *kernel Interface for system calls*

定義遇到不同 *system call* 該找誰處理

在本題的情況下，是呼叫 kernel->interrupt->Halt() 。

(v) **interrupt.cc**

Interrupt::Halt ( )

呼叫 kernel->stats->Print()，輸出當前資訊 (如 totalTicks)，

並刪除 kernel (停止這個 kernel 的運作)。

(2) SC_Create

(i) **exception.cc**

ExceptionHandler (ExceptionType which)

解析傳入的Exception Type，如果是 SyscallException，則根據register[2]的內容(Type)，去執行相應的處理。

在本題的情況下，Type 為 SC_Create，而相應的處理如下：

1. 首先讀取 register[4] 的內容 (檔案名稱在主記憶體中的位置)，
2. 接著呼叫 SysCreate(filename)，並將執行結果存回 register[2]，
3. 最後推進 *PC counter*，以免無限地重複執行這個 Exception。

(ii) **ksyscall.h**

SysCreate (char *filename)

kernel Interface for system calls

在本題的情況下，是呼叫 kernel->fileSystem->Create(filename)，並回傳成功與否。

(iii) **filesys.h**

FileSystem::Create (char *name, int initialSize)

使用非 *Stub* 的 *File System*。

首先建立一個 Directory，並從根目錄開始，去查詢欲新增的檔案名稱 (參數name) 是否已經存在，如果存在就回傳 *FALSE*。

否則，會在判斷是否有足夠儲存空間之後，取得一塊參數initialSize 大小的空間，然後建立一個新的檔案，並回傳 *TRUE*。

(3) SC_PrintInt

(i) **exception.cc**

ExceptionHandler (ExceptionType which)

解析傳入的 Exception Type，如果是 SyscallException ，則根據 register[2] 的內容(Type)，去執行相應的處理。

在本題的情況下，Type 為 SC_PrintInt，而相應的處理如下 :
首先讀取 register[4] 的內容 (欲輸出的數字)，
接著呼叫 SysPrintInt(val)，將 r4 傳入，
最後推進 *PC counter*，以免無限地重複執行這個 *Exception*。

(ii) **ksyscall.h**

SysPrintInt (int val)

kernel Interface for system calls

在本題的情況下，是呼叫 kernel->synchConsoleOut->PutInt(val)

(iii) **synchconsole.cc**

SynchConsoleOutput::PutInt (int value)

先以 sprint 將欲輸出的參數 value 格式化，並存入 str 變數中。
使用 lock->Acquire()，使得只有當前這個 *Thread* 可以執行 *Output*。
然後不斷執行 PutChar()，將 str 裡面的字元一個一個傳入
並且呼叫 waitFor->P()，等待 waitFor->V() 被呼叫。
直到當前字元為結束字元 *'\0'* 為止。
使用 lock->Release()，讓 *Output* 可再次被其他 *Thread* 使用。

SynchConsoleOutput::PutChar (char ch)

執行 lock->Acquire()。
執行 consoleOutput->PutChar()，傳入剛剛傳入的字元，
並且執行 waitFor->P( )，等待 *CallBack function* 呼叫 waitFor->V()。
執行 lock->Release()。

(iv) **console.cc**

ConsoleOutput::PutChar (char ch)

呼叫 WriteFile()，將參數 ch 放到輸出模擬之中。

並以變數 putBusy 來確保同時只會有一個字元被輸出。

之後 *Schedule* 一個 100 ticks 以後的 *Interrupt*，

作為讀寫一個字元的耗時。

在本題的情況下，這個 *Thread* 會進入睡眠模式。

如果沒有其他 *Thread* 要執行，則 *Interrupt* 會進入 *Idle* 狀態，

並跳至下一個 *Interrupt* 到來的時間點 (也就是 100 ticks 之後)。

(v) **interrupt.cc**

Interrupt::Schedule (CallBackObj *toCall, int fromNow, IntType type)

用上一個函式傳來的所有資訊，建立 PendingInterrupt 的 *Instance*，

並將這個 *Instance* 排入 *pending List* 之中，等待其執行的時機到來。

(vi) **mipssim.cc**

Machine::Run ()

執行 OneInstruction() 來處理其他可以執行的指令。

使用 OneTick() 來推進時間。

(vii) **interrupt.cc**

Machine::OneTick ( )

依據現在的是 SystemMode 或是 UserMode，去增加各自的 Tick 數。

並且在停止接收 *Interrupt* 之後，使用 CheckIfDue() 檢查是否有即將到來的 *Interrupt*。

Interrupt::CheckIfDue (bool advancedClock)

檢查是否有已經到來的 Pending Interrupt，如果有，則執行其 CallBack 函數。

如果當前的 *Interrupt* 是處於 *Idle* 狀態，則參數 advanceClock 會是 *TRUE*，此時如果有尚未到來的 Pending Interrupt 等待執行，則會直接將 *Ticks* 跳至該 Pending Interrupt 的時間。

(viii) **console.cc**

ConsoleOutput::CallBack ( )

Pending Interrupt 到來後，會執行這個 CallBack 函式。

將變數 putBusy 設為 *FALSE*，並記錄已經寫入幾個字元。

呼叫下一層的 CallBack。

在本題的情況下，之前預約的 Pending Interrupt 預計會在 100 ticks 之後才會到來。

(ix) **synchconsole.cc**

SynchConsoleOutput::CallBack ( )

呼叫 waitFor->V( )，以解除之前 *Thread* 的睡眠狀態。

在本題的情況下，這個 *Thread* 會接著執行 consoleOutput->PutChar()，並且再次睡眠，然後再次等待解除睡眠，直到輸出完畢。

(c) Implementation

我們主要的想法，是先將能用的資源一步步取得，再去寫底層實作的部分。

1. 在 syscall.h 中，將定義好的 SC_Open、SC_Read 等取消註解。
2. 在 exception.cc 中，依據其他已經寫好的 *Case*，來新增我們需要的四種 *Case*，並依據所需的參數，去存取 registers 的值。
3. 在 ksycall.h 中，模仿其他已經寫好的 *Interface*，將四個函式介面和其所需的參數寫好。
4. 接著在 Start.S 中，依據其他 *Exception* 的寫法，新增四個 *Exception*。

完成了簡單的部分，接著就是實作函式內容。大部分的改動都在 filesys.h 裡面。而因為我們被要求使用 *Stub File System*，所以不需要去改動 filesys.cc。

1. OpenAFile (char *name)

   這部分本來 *spec* 的第二點有規定不可使用 sysdep.h 當中的函數來存取檔案，因此我們使用同一個 *Class* 底下的 Open( )函式，來實做這部分。

   為了要回傳 OpenFileId，讓我們遇到了很大困難。因為 Open( )函式是回傳 OpenFile *object*，而其 Id 是一個 *private* 的欄位，因此無法直接讀取他。

   最後我們的解決辦法，是使用一個 *for* 迴圈去尋找 OpenFileTable 當中，目前空閒的欄位。

```
OpenFileId OpenAFile(char *name) {
    index = 20;
    for(OpenFileId i = 0; i < 20; i++){
      if(OpenFileTable[i] == NULL){
        index = i;
        break;
      }
    }
    if(index < 0 || index >= 20)
    {
        return -1;
    }
    OpenFile *obj = Open(name);
    if (obj == NULL) return -1;
    OpenFileTable[index] = obj;
    return index;
}
```

   初始化將 index 設為 20，這樣未尋找到時就會進入下方判斷式中，return -1。同時我們也需要判斷找到的位置是否超過開啟的限制，最後將改動 OpenFileTable

2. WriteFile, ReadFile

   這兩個函數實作方面比較相似

```
int WriteFile(char *buffer, int size, OpenFileId id){
    if (OpenFileTable[id] == NULL || size < 0 || id < 0 || id >=20) return -1;
    return OpenFileTable[id]->Write(buffer, size);
}

int ReadFile(char *buffer, int size, OpenFileId id){
    if (OpenFileTable[id] == NULL|| size < 0 || id < 0 || id >=20)  return -1;

    return OpenFileTable[id]->Read(buffer, size);
}
```

   在判斷完 id 是否 *valid* 後，接著判斷 size 為正常的輸入，再呼叫相對應函數。

3. CloseFile

   首先判斷 id 是否 *valid*，

   再用 *delete* 關鍵字，去觸發 OpenFile *class* 的 *destructor*。

   這個 *destructor* 會為我們呼叫底層的 Close 函式。

   最後我們再把對應的 OpenFileTable 欄位清掉。

```
int CloseFile(OpenFileId id){
    if (OpenFileTable[id] == NULL|| id < 0 || id >=20)  return -1;

    delete OpenFileTable[id];
    OpenFileTable[id] = NULL;
    return 1;
}
```

(d) What difficulties did you encounter when implementing this assignment?

Tracing the code at first was quite complicated, since there are multiple files linking to each other. However, after spending a lot of time, we slowly began to understand how the system worked.

Implementing was hard to begin, due to us not knowing where to start. We then opened up the fileIO_test1 and understand how the functions were being called at the begging, we then followed the steps and the understanding we had from tracing the code.

Initially, the second point of the spec requiring us not using functions from the sysdep.h file made it hard for us to implement Open. However, we overcame the obstacle.

(e) Any feedback you would like to let us know.

It would be nice to know earlier that we could use functions from sysdep.h.

We spend multiple hours trying to solve the problem. After solving it, we then received an email saying that we could use the functions. It would be nice to know that we could use the functions in the first place.

We learned a lot from this project and have a deeper understanding of how the system calls work!