NACHOS-4.0

MP1

第 21 組

106062173 吳東弦(一起 trace code，負責整理 report)

106062271 郭子豪(一起 trace code，負責實作 coding)

一、 Trace code

（一） SC_Halt

1. Machine::Run()

當機器開機（執行 main）後，作業系統會被載入（實體化 kernel），接著在 Run()中會跑一個無窮迴圈，不斷執行 OneInstruction(instr)和 OneTick()，前者就像在監聽是否有 instruction 要被執行，後者則持續計算經過的時間。

2. Machine::OneInstruction()

在 OneInstruction 中首先會看是否有指令要被執行（ReadMem），如果有，就會去解讀這個指令要做什麼（decode），接著再根據指令的 opcode，執行相對應的動作。比如在執行 Halt 函式時，addiu 會把後兩個 register 的值加到前面的暫存器，j 會跳到暫存器所指向的位址，而 syscall 則呼叫 RaiseException，準備進行模式切換。

3. Machine::RaiseException()

執行 Halt 時，原本是在 UserMode 中，在 RaiseException 中，由於要執行系統呼叫（透過 ExceptionHandler），會先切換到 SystemMode，等執行完後，再切換回 Usermode（透過 setStatus）。

4. ExceptionHandler()

由於 MIPS 中$v0 暫存器，也就是$2 暫存器，其中一個作用為存放系統呼叫的類型，因此，在 ExceptionHandler 中，會依照傳進來的參數與$2 暫存器的值，來決定要執行哪一種系統呼叫，以本題 Halt 為例，根據$2 暫存器的值要執行的是 SC_Halt，所以會去呼叫 SysHalt。

5. SysHalt()

在 SysHalt 中會去執行作業系統中 Halt 的 interrupt 服務。

6. Interrupt::Halt()

在 Halt 中，最後會將作業系統移出記憶體（delete kernel），模擬虛擬機關機的樣子。

（二） SC_Create

1. ExceptionHandler()

在 ExceptiopHandler 中，根據$2 暫存器的值要執行的是 SC_Create。由於 MIPS 預設將函數參數依序放在$4~$7 暫存器，所以先從$4 暫存器將檔名存入主記憶體，再作為參數傳入 SysCreate。接著，又因為$2 暫存器另一個作用為存放函數的返回值，因此，當建檔成功後回傳的布林值的會讀入$2 暫存器。最後，再將 PC 暫存器遞進。

2. SysCreate()

在 SysCreate 中會去執行作業系統中 fileSystem 的 Create 服務。

3. FileSystem::Create()

在 Create 中，呼叫了 OpenForWrite，並利用 c 語言函式庫的 open 進行開檔，由於我們只要創建檔案，開檔後就直接 close 了。

（三） SC_PrintInt

1. ExceptionHandler()

在 ExceptiopHandler 中，根據$2 暫存器的值要執行的是 SC_PrintInt。先從$4 暫存器將要印出來的數字存入主記憶體，並作為參數傳入 SysPrintInt，最後，再將 PC 暫存器遞進。

2. SysPrintInt()

在 SysPrintInt 中會去執行作業系統中 synchConsoleOut 的 PutInt 服務。

3. SynchConsoleOutput::PutInt() （同 PutChar()）

（1） lock->Acquire()

當要印出數字時，如果同時有其執行緒也同時需要此 I/O，會導致交錯印出，產生錯誤的結果，因此，必須指定當前執行緒（lockholder），不讓別的執

行緒進入，而上鎖（P）、解鎖（V）則是由 Semaphore（在此為 semaphore）負責。

在 semaphore 的 P 中，會先讓當前執行緒執行（value = = 1），然後將其鎖上，避免其他執行緒進入（value = 0）。如果 ready queue 有別的執行緒也可能用到此 I/O（value = = 0），semaphore 會將其轉至 waiting queue 並使其 sleep，避免這段期間這些執行緒回到 ready queue。

接著，如果 ready queue 有其他與此 I/O 無關的執行緒，就會讓 CPU 繼續 Run（asynchronized I/O），否則如果 ready queue 為空，沒有任何執行緒的話，CPU 就會 ilde。

（2） do { PutChar && waitFor->P() } while ()

接著進入 do-while 迴圈，將字元一個一個印出來，以完成整個數字的輸出。在這個執行緒內，每個輸出都是一個動作，為了避免字元交錯印出，同樣需要一個 Semaphore（在此為 waitFor）控制，也就是 waitFor 的 P，而 P 的執行過程與前項相同。

（3） lock->Release()

當整個數字印完後，就可以將當前執行緒釋放（lockholder），接著，在 semaphore 的 V 中，會將其他 sleep 中的執行緒喚醒，放回 ready queue。

4. ConsoleOutput::PutChar()

PutChar 中，模擬將字元輸出到 I/O 設備的情況（WriteFile），將 putBusy 設為 TRUE，表示正在 I/O 中，並將輸出後要發出的中斷加入排程（Schedule），

也就是對 pending 插入 toOccur 指標所指向的 callOnInterrupt。

5. Machine::Run()

如同第一題所示，Run 中會跑無窮迴圈，不斷執行 OneInstruction(instr)和 OneTick()，從 semaphore 的 P 和 waitFor 的 P 可以知道，這時其實 I/O 仍然在進行中，此即 asynchronized 的特性。

6. Interrupt::OneTick()

OneTick 的呼叫，表示時間的推移，每一個 Tick 可看成時間的最小單位。單純執行一個使用者的指令時，會遞進一個 UserTick（1Tick），若是執行系統呼叫或是中斷，則會遞進一個 SystemTick（10Tick）。

7. Interrupt::CheckIfDue()

在 CheckIfDue 中，會去檢查 pending 是否有中斷訊號（callOnInterrupt），也就是剛剛在 PutChar 所做的排程（schedule），如果有的話，代表已經做完一次 I/O，準備執行接下來的 CallBack。另外在這裡可以發現，在整份 code 中，CheckIfDue 會出現在兩個地方，其一為 Idle，其一為 Run，也就是不論 CPU 在執行或是在閒置中，都持續在偵測是否有中斷訊號，這也是 asynchronized 的特性。

8. ConsoleOutput::CallBack()

這個 CallBack 將 putBusy 設為 FALSE，表示 I/O 已經結束，接著呼叫下一個 CallBack。

9. SynchConsoleOutput::CallBack()

這個 CallBack 呼叫了 waitFor 的 P，也就是表示這個字元已經印完了，要去喚醒其他 sleep 的字元，準備進行輸出。

二、 Implement four I/O system calls in NachOS

（一） define

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

這裡將 Macro 的註解取消，ExceptionHandler 才抓得到這些 case。

（二） ExceptionHandler

```
case SC_Open:
    val = kernel->machine->ReadRegister(4);

    {
        char *filename = &(kernel->machine->mainMemory[val]);
        fileID = SysOpen(filename);
        kernel->machine->WriteRegister(2, (int) fileID);
    }

    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;

case SC_Read:
    val = kernel->machine->ReadRegister(4);

    {
        char *buffer = &(kernel->machine->mainMemory[val]);
        int size = kernel->machine->ReadRegister(5);
        OpenFileId id = kernel->machine->ReadRegister(6);
        numChar = SysRead(buffer, size, id);
        kernel->machine->WriteRegister(2, (int) numChar);
    }

    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;
```

```
case SC_Write:
    val = kernel->machine->ReadRegister(4);

    {
        char *buffer = &(kernel->machine->mainMemory[val]);
        int size = kernel->machine->ReadRegister(5);
        OpenFileId id = kernel->machine->ReadRegister(6);
        numChar = SysWrite(buffer, size, id);
        kernel->machine->WriteRegister(2, (int) numChar);
    }

    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;
case SC_Close:
    OpenFileId id = kernel->machine->ReadRegister(4);

    {
        fileID = SysClose(id);
        kernel->machine->WriteRegister(2, (int) fileID);
    }

    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;
```

從$2 取得對應中斷訊號的 case 後，會依序從$4、$5、$6 取得對應的參數，接著呼叫對應的 SysCall，執行完後，將 PC 暫存器遞進。

（三） SysCall

```
OpenFileId SysOpen(char * filename) {
    return kernel->fileSystem->OpenAFile(filename);
}

int SysRead(char * buffer, int size, OpenFileId id) {
    return kernel->fileSystem->ReadFile(buffer, size, id);
}

int SysWrite(char * buffer, int size, OpenFileId id) {
    return kernel->fileSystem->WriteFile(buffer, size, id);
}

int SysClose(OpenFileId id) {
    return kernel->fileSystem->CloseFile(id);
}
```

接著 SysCall 會去呼叫 kernel 對應的實作。

（四） FileSystem

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

int WriteFile(char *buffer, int size, OpenFileId id) {
    if( (id < 20 && id >= 0 && size >= 0) && fileDescriptorTable[id] != NULL) {
        return fileDescriptorTable[id]->Write(buffer,size);
    }
}

int ReadFile(char *buffer, int size, OpenFileId id) {
    if( (id < 20 && id >= 0 && size >= 0) && fileDescriptorTable[id] != NULL) {
        return fileDescriptorTable[id]->Read(buffer,size);
    }
}

int CloseFile(OpenFileId id) {
    if(id < 20 && id >= 0 && fileDescriptorTable[id] != NULL) {
        delete fileDescriptorTable[id];
        fileDescriptorTable[id] = NULL;
        return 1;
    }
}
```

（1） OpenAFile

由於 fileDescripterTable 最大只有 20，所以必須先檢查 FDT 是否有空位，如果滿了就無法開檔，即使有空位，還要再進入 OpenForReadWrite 檢查這個檔案是否能被開啟，比如 filename 是否存在，如果不存在一樣無法開檔，都檢查完確定能開後，就會在 FDT 下一個空位開啟。

（2） ReadFile

由於 fileDescripterTable 最大只有 20，所以必須先檢查要讀取的檔案 id 是否在 FDT 範圍內，接著檔案必須開啟後才能讀取，也就是還要檢查 FDT 對應位置是否為 NULL，都檢查完後就可以讀取檔案了。

（3） WriteFile

由於 fileDescripterTable 最大只有 20，所以必須先檢查要寫入的檔案 id 是否在 FDT 範圍內，接著檔案必須開啟後才能寫入，也就是還要檢查 FDT 對應位置是否為 NULL，都檢查完後就可以寫入檔案了。

（4） CloseFile

由於 fileDescripterTable 最大只有 20，所以必須先檢查要關閉的檔案 id 是否在 FDT 範圍內，接著檔案必須在開啟狀態下才能關閉，也就是還要檢查 FDT 對應位置是否為 NULL，都檢查完後就可以關閉檔案了。

三、 Summary & FeedBack

這次作業由於整份專案非常大，要在許多檔案跳來跳去，再加上有些函數在不同 class 下名稱很相似，甚至一樣，導致經常迷路，經過多方嘗試，學到了在 vscode 如何快速跳轉到想去的位置，以及在終端下透過 grep 查詢，需要的函數定義、呼叫在哪個檔案。

其中 trace 第三題我們認為是本次作業最難的部分，首先得了解在 MIPS 的規範下，$2 與$4~7 暫存器有什麼特定的作用，接著理解真正的作業系統中，lock 與 semaphore 如何透過 P、V 控制 threads 執行的順序與安全性，最後是執行中斷訊號時，兩次 callback 如何被呼叫。