

NachOS – MP1

System Call

宋體淮, R09921135, Electrical Engineering

1. Trace code

1.1. Trace the SC_Halt system call to understand the implementation of a system call. (Sample code: halt.c)

Machine::run() :

- 模擬機器開始執行，設定 user mode，並且執行 OneInstruction() 來模擬 CPU 的指令執行過程。

```
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); // 需要Syscall時轉換為KernelMode

    for (;;) {
        DEBUG(dbgTraCode, "In Machine::Run(), into OneInstruction " << "==" Tick " << kernel->stats->totalTicks << " ==");
        OneInstruction(instr); // 模擬CPU的指令執行過程
    }
}
```

Machine::OneInstruction() :

- 從 program counter register 讀出指令，並透過解碼的動作從 opCode 來判斷指令類型。
- 當發現指令類型為 system call 時，就會進入 RaiseException() 來處理，拋出一個 SyscallException 類型的異常。

```
case OP_SYSCALL: // 呼叫System call
    DEBUG(dbgTraCode, "In Machine::OneInstruction, RaiseException(SyscallException, 0), " << kernel->stats->totalTicks);
    RaiseException(SyscallException, 0); // 發出System csll異常
    return;
```

Machine::RaiseException() :

- 從 user mode 轉換至 kernel mode，才可執行 privileged instruction。
- 此時的 which 為 SyscallException 類型的異常，故 ExceptionHandler() 就會針對這類型的異常來做出相對應的處理，等

到處理完之後再轉換回 user mode。

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

    registers[BadVAddrReg] = badVAddr;

    DelayedLoad(0, 0);          // finish anything in progress

    kernel->interrupt->setStatus(SystemMode); // 從User mode轉換成Kernel mode

    ExceptionHandler(which);    // interrupts are enabled at this point
                                // 異常處理

    kernel->interrupt->setStatus(UserMode); // 處理完後再轉換回User mode
}
```

ExceptionHandler() :

- 首先判斷現在是何種異常類型，對此例子而言為 SyscallException。
- 再從 register 2 讀出是何種 system call，以現在來說就是 SC_Halt，接著進入到 SysHalt()。

```
ExceptionHandler(ExceptionType which)
{
    char ch;
    int val;
    int type = kernel->machine->ReadRegister(2); // 從Register 2讀取是何種System call類型
    int status, exit, threadID, programID, fileID, numChar;
    DEBUG(dbgSys, "Received Exception " << which << " type: " << type << "\n");
    DEBUG(dbgTraCode, "In ExceptionHandler(), Received Exception " << which << " type: " << type << ", " << kernel->stats->totalTicks);

    switch (which) {
    case SyscallException:

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

SysHalt() :

- ksyscall.h 裡宣告了許多 system call 函式，並且會呼叫其他地方來實作。
- 可知 Halt() 在 interrupt.cc 裡被實作。

```
void SysHalt()
{
    kernel->interrupt->Halt();
}
```

Interrupt::Halt() :

- 該指令會將整個 kernel 給刪除掉。

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

1.2. Trace the SC_Create system call to understand the basic operations and data structure in a file system. (Sample code: createFile.c)

- SC_Create 在進入到 ExceptionHandler() 之前的程序和 SC_Halt 相同，故從此處接續 trace code。

ExceptionHandler() :

- 先從 register 4 中讀取 system call 參數，此例子為檔案的位址。並且讓 filename 指標指向該位址。
- SC_Create 的內容會在 SysCreate() 中被更具體定義。
- 完成之後要更新 program counter，往前 4 個 byte(一條指令 32bit)。

```
case SC_Create:
{
    val = kernel->machine->ReadRegister(4); // 從Register 4讀取System call參數(File name)
    char *filename = &(kernel->machine->mainMemory[val]);
    status = SysCreate(filename); // SC_Create被實做的地方
    kernel->machine->WriteRegister(2, (int) status);
    // 更新Program counter
    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;
}
```

SysCreate() :

- 可知會在 fileSystem.cc 中的 Create() 函數被實作。

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

FileSystem::Create() :

- 由於這次是用 stub file system 不是 real file system，所以 filesys.cc 並不會被執行到(因為有 #ifndef FILESYS_STUB)。
- 所以我們只要看 filesys.h 即可，並且可知 OpenForWrite() 只是原生的 C library 而已。

```
class FileSystem {
public:
    FileSystem() {
        for (int i = 0; i < 20; i++) fileDescriptorTable[i] = NULL;
    }

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

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

1.3. Trace the SC_PrintInt system call to understand how NachOS implements asynchronous I/O using Callback functions and register schedule events.

(Sample code: add.c)

- SC_PrintInt 在進入到 ExceptionHandler() 之前的程序和 SC_Halt 相同，故從此處接續 trace code。

ExceptionHandler() :

- 從 register 4 中讀取 system call 的參數。
- SC_PrintInt 會在 SysPrintInt() 中被更具體定義。
- 完成之後要更新 program counter。

```

case SC_PrintInt:
{
    DEBUG(dbgSys, "Print Int\n");
    val = kernel->machine->ReadRegister(4);
    DEBUG(dbgTraCode, "In ExceptionHandler(), into SysPrintInt, " << kernel->stats->totalTicks);
    SysPrintInt(val);    // SC_PrintInt被實做的地方
    DEBUG(dbgTraCode, "In ExceptionHandler(), return from SysPrintInt, " << kernel->stats->totalTicks);
    // Set Program Counter
    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;
}

```

SysPrintInt() :

- 可知會在 synchConsoleOut.cc 中的 PutInt() 函數中被實作。

```

void SysPrintInt(int val)
{
    DEBUG(dbgTraCode, "In ksyscall.h:SysPrintInt, into synchConsoleOut->PutInt, " << kernel->stats->totalTicks);
    kernel->synchConsoleOut->PutInt(val);
    DEBUG(dbgTraCode, "In ksyscall.h:SysPrintInt, return from synchConsoleOut->PutInt, " << kernel->stats->totalTicks);
}

```

SynchConsoleOutput::PutInt() :

- 首先用 sprintf 將 value 存到字元陣列 str，變成字元型態(value 是存在 register 4 中的 system call 參數值)。
- 藉由 lock->Acquire() 鎖定物件達成 synchronization。只有取得鎖定的 thread 才可以要求寫入到 console display 的動作，所以在該 thread 還未印完所有字元之前，其他 thread 必須等待，直到有機會取得鎖定。
- 將 str 字元陣列裡的字元，一個一個丟入 consoleOut.cc 裡的 PutChar()，待會會看到印完一個字元之後會先到一個 pending list 裡面去等，所以不會一次跑完整個迴圈。
- 做完 synchronization 後，用 lock->Release() 解除鎖定，讓其他物件有機會取得資源。
- 這裡使用 waitFor->P() 代表佔用了一個 semaphore 的資源，防止 consoleOutput 的 resource 被無限量的用完。

```

SynchConsoleOutput::PutInt(int value)
{
    char str[15];
    int idx=0;
    //sprintf(str, "%d\n", value); the true one
    sprintf(str, "%d\n", value); //simply for trace code
    lock->Acquire();
    do{
        DEBUG(dbgTraCode, "In SynchConsoleOutput::PutChar, into consoleOutput->PutChar, " << kernel->stats->totalTicks);
        consoleOutput->PutChar(str[idx]);
        DEBUG(dbgTraCode, "In SynchConsoleOutput::PutChar, return from consoleOutput->PutChar, " << kernel->stats->totalTicks);
        idx++;

        DEBUG(dbgTraCode, "In SynchConsoleOutput::PutChar, into waitFor->P(), " << kernel->stats->totalTicks);
        waitFor->P();
        DEBUG(dbgTraCode, "In SynchConsoleOutput::PutChar, return from waitFor->P(), " << kernel->stats->totalTicks);
    } while (str[idx] != '\0');
    lock->Release();
}

```

ConsoleOutput::PutChar() :

- 藉由原生的 C library 的 WriteFile() 來將一個字元 ch 寫入到 writeFileNo (writeFileNo 預設是 stdout, 可參考 ConsoleOutput 被建構時的樣子), 也是在這裡真正把字元印出來。
- 將 putBusy 的狀態改成 True, 讓其他事情不能一起做
- 進入 interrupt.cc 裡的 Schedule(), 向 CPU 發出 interrupt (猜測是因為還有其他要印的字元, 剛剛只印了一個而已), 安排預定被 CPU 執行的時間。
- this 的功能就像是 python 語法中的 self 一樣。

```

ConsoleOutput::PutChar(char ch)
{
    ASSERT(putBusy == FALSE);
    WriteFile(writeFileNo, &ch, sizeof(char));
    putBusy = TRUE;
    kernel->interrupt->Schedule(this, ConsoleTime, ConsoleWriteInt);
}

```

Interrupt::Schedule() :

- toCall 是一個 function pointer, 當 interrupt 要發生時會 callback 的物件, 在此例子是整個 ConsoleOutput 物件。
- fromNow 是指距離此刻 (totalTicks), interrupt 要發生的時間。
- ConsoleWriteInt 是觸發這個 interrupt 的硬體。
- 會計算 interrupt 會發生的絕對時間 when, 然後在 PendingInterrupt 這個 list 裡插入要被執行的 interrupt。


```

Interrupt::Schedule(CallBackObj *toCall, int fromNow, IntType type)
{
    int when = kernel->stats->totalTicks + fromNow;
    PendingInterrupt *toOccur = new PendingInterrupt(toCall, when, type);

    DEBUG(dbgInt, "Scheduling interrupt handler the " << intTypeNames[type] << " at time = " << when);
    ASSERT(fromNow > 0);

    pending->Insert(toOccur); // Register interrupt callback function in pending queue
}

```

Machine::Run() :

- 接著回到 CPU 的執行階段，當安排好 interrupt 被執行的時間，就只要等待 CPU 再回來執行它。
- 執行完 OneInstruction() 後，會進到 interrupt.cc 裡的 OneTick() 函數。

```

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); // 需要Syscall時轉換為KernelMode

    for (;;) {
        DEBUG(dbgTraCode, "In Machine::Run(), into OneInstruction " << "== Tick " << kernel->stats->totalTicks << " ==");

        OneInstruction(instr); // 模擬CPU的指令執行過程

        DEBUG(dbgTraCode, "In Machine::Run(), return from OneInstruction " << "== Tick " << kernel->stats->totalTicks << " ==");
        DEBUG(dbgTraCode, "In Machine::Run(), into OneTick " << "== Tick " << kernel->stats->totalTicks << " ==");

        kernel->interrupt->OneTick();

        DEBUG(dbgTraCode, "In Machine::Run(), return from OneTick " << "== Tick " << kernel->stats->totalTicks << " ==");
        if (singleStep && (runUntilTime <= kernel->stats->totalTicks))
            Debugger();
    }
}

```

Interrupt::OneTick() :

- 此函式會讓系統的時刻往前，就是在模擬時間往前走，可以看到 totalTicks、systemTicks 或是 userTicks 都被更新。
- 並且會在 CheckIfDue() 中去檢查是否有 pending 的 interrupt 到達要被執行的時間了。
- 也是在這裡做到 Round Robin 的 CPU 排程，當 yieldOnReturn 為 True 時，會釋放目前的 thread，執行下一個 thread。

```

Interrupt::OneTick()
{
    MachineStatus oldStatus = status;
    Statistics *stats = kernel->stats;

    // advance simulated time
    if (status == SystemMode) {
        stats->totalTicks += SystemTick;
        stats->systemTicks += SystemTick;
    } else {
        stats->totalTicks += UserTick;
        stats->userTicks += UserTick;
    }
    DEBUG(dbgInt, "== Tick " << stats->totalTicks << " ==");

    // check any pending interrupts are now ready to fire
    ChangeLevel(IntOn, IntOff); // first, turn off interrupts
    // (interrupt handlers run with
    // interrupts disabled)
    CheckIfDue(FALSE); // check for pending interrupts
    ChangeLevel(IntOff, IntOn); // re-enable interrupts
    if (yieldOnReturn) { // if the timer device handler asked
        // for a context switch, ok to do it now
        yieldOnReturn = FALSE;
        status = SystemMode; // yield is a kernel routine
        kernel->currentThread->Yield();
        status = oldStatus;
    }
}

```

Interrupt::CheckIfDue() :

- 可以看到當 pending list 裡面不為空時，也就是有其他 thread 要求 interrupt(此例子為剛剛才印了一個字元的 thread 還沒印完)，且到了它該執行的時間 (when 小於 totalTicks)，或者可以說這個 thread 的 time slice 還沒結束(因預設是用 Round Robin 的 CPU 排程)，那他就會去執行 next->callOnInterrupt->CallBack()。
- next 是原本 pending list 的 Front() (第一個物件)。
- 因前面 interrupt->Schedule 函式在包裝 PendingInterrupt 物件並插入 pending list 時，有將 ConsoleOutput 整個物件放進去，所以 next->callOnInterrupt->CallBack() 就是呼叫 ConsoleOutput->CallBack()。


```

inHandler = TRUE;

do {
    next = pending->RemoveFront();    // pull interrupt off list // 1910010[J]: Pull interrupt from pending queue
    DEBUG(dbgTraCode, "In Interrupt::CheckIfDue, into callOnInterrupt->CallBack, " << stats->totalTicks);

    next->callOnInterrupt->CallBack(); // call the interrupt handler
    // call interrupt service routine (callback function)

    DEBUG(dbgTraCode, "In Interrupt::CheckIfDue, return from callOnInterrupt->CallBack, " << stats->totalTicks);
    delete next;
} while ( !pending->IsEmpty() && (pending->Front()->when <= stats->totalTicks) );

inHandler = FALSE;
return TRUE;

```

ConsoleOutput::CallBack() :

- 這個 CallBack() 的目的是告訴 kernel 又可以多印一個字元到 console display 上了。
- 前面在 PutChar() 時將 putBusy 設為 True，在 CallBack() 把 putBusy 設回 FALSE 才可以讓它再去做下一次的 PutChar()。
- callWhenDone 是 ConsoleOutput 物件的成員，在下面會解釋所以 callWhenDone 到底是甚麼。

```

ConsoleOutput::CallBack()
{
    DEBUG(dbgTraCode, "In ConsoleOutput::CallBack(), " << kernel->stats->totalTicks);
    putBusy = FALSE;
    kernel->stats->numConsoleCharsWritten++;
    callWhenDone->CallBack();
}

```

- 在一開始創建一個 kernel 時，會初始化 SyncConsoleOutput 物件，而該物件在初始化時又會將 outputFile (預設是 stdout) 和 this (就是自己) 當作參數傳給 ConsoleOutput 的建構子。

```

SynchConsoleOutput::SynchConsoleOutput(char *outputFile)
{
    consoleOutput = new ConsoleOutput(outputFile, this);
    lock = new Lock("console out");
    waitFor = new Semaphore("console out", 0);
}

```

- 而 ConsoleOutput 在建構時，會讓 callWhenDone = toCall，所以可以知道其實 callWhenDone 就是 SyncConsoleOutput 物件。
- 所以回到 ConsoleOutput::CallBack()，其實 callWhenDone->CallBack() 就是 SyncConsoleOutput->Call()。

```

ConsoleOutput::ConsoleOutput(char *writeFile, CallbackObj *toCall)
{
    if (writeFile == NULL)
        writeFileNo = 1;           // display = stdout
    else
        writeFileNo = OpenForWrite(writeFile);

    callWhenDone = toCall;
    putBusy = FALSE;
}

```

SyncConsoleOutput::Callback() :

- waitfor->V() 是代表因使用完資源，釋放出一個 semaphore 的值。
- 所以這時又會回到 SynchConsoleOutput::PutInt() 的迴圈中，繼續剛剛還沒完成的印字元的任務，然後重複上面的整個流程。

```

SynchConsoleOutput::Callback()
{
    DEBUG(dbgTraCode, "In SynchConsoleOutput::Callback(), " << kernel->stats->totalTicks);
    waitfor->V();
}

```

2. Implement four I/O system calls: Open, Read, Write, Close.

syscall.h :

- 在檔案中先定義這四種 system call 的編號。

```

#define SC_Open      5
#define SC_Read      6
#define SC_Write     7
#define SC_Close     8

```

- 並且宣告四種 system call 在外部程式使用時的規格。

```

OpenFileId Open(char *name, int mode);

/* Write "size" bytes from "buffer" to the open file.
 * Return the number of bytes actually read on success.
 * On failure, a negative error code is returned.
 */
int Write(char *buffer, int size, OpenFileId id);

/* Read "size" bytes from the open file into "buffer".
 * Return the number of bytes actually read -- if the open file isn't
 * long enough, or if it is an I/O device, and there aren't enough
 * characters to read, return whatever is available (for I/O devices,
 * you should always wait until you can return at least one character).
 */
int Read(char *buffer, int size, OpenFileId id);

/* Close the file, we're done reading and writing to it.
 * Return 1 on success, negative error code on failure
 */
int Close(OpenFileId id);

```

start.s :

- 接著修改 start.s 組合語言檔案，參考其他 system call 代碼複製了 4 份 code 來修改 SC 代碼。
- 這份組語會看的到外部程式用到了哪種 system call 函式(因有.global)，並將該函式的 system call 編號放進 register 2 裡頭，之後 exception handler 會從 register 2 裡讀取內容。

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

.globl Write
.ent    Write
Write:
    addiu $2,$0,SC_Write
    syscall
    j      $31
.end    Write

.globl Read
.ent    Read
Read:
    addiu $2,$0,SC_Read
    syscall
    j      $31
.end    Read

.globl Close
.ent    Close
Close:
    addiu $2,$0,SC_Close
    syscall
    j      $31
.end    Close
```

exception.c :

- 接著進到 ExceptionHandler() 函式，會先將 register 2 的 system call 編號給讀出來傳給 type，並根據 type 的數值(剛剛在 syscall.h 中所定義的)來執行不同的 system call。

```
ExceptionHandler(ExceptionType which)
{
    char ch;
    int val;
    int type = kernel->machine->ReadRegister(2);
    int status, exit, threadID, programID, fileID, numChar;

    DEBUG(dbgSys, "Received Exception " << which << " type: " << type << "\n");

    cout << "which: " << which << endl;
    switch (which) {
    case SyscallException:
        switch(type) {
        case SC_Halt:
            DEBUG(dbgSys, "Shutdown, initiated by user program.\n");
```

- 以下新增四種 Case：SC_Open、SC_Read、SC_Write、SC_Close：

SC_Open :

- 與 SC_Create 的做法類似，首先從 register 4 取得存放檔案的位址，再去 mainMemory[val] 中拿到 filename 的指標。
- 接著 kernel->fileSystem->OpenAFile 來達成 open 的功能，並將回傳值寫入 register 2。
- 最後將 program counter register 更新並往前 4 個 byte。

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

    status = kernel->fileSystem->OpenAFile(filename);

    kernel->machine->WriteRegister(2, (int) status);
    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_Read :

- 一樣從 register 4 拿到 buffer 的位址，並藉由 buffer 指標指向該位址。
- numChar 和 fileId 是第二、三個參數，所以會依序存在 register 5, register 6 之中(可參考 start.s 內的說明)。
- 接著利用 kernel->fileSystem->ReadFile 來達到 read 的功能。
- 回傳值是「總共讀取到幾個 character」，存入 register 2。
- 最後一樣更新 program counter。

```
case SC_Read:
{
    val = kernel->machine->ReadRegister(4);
    //cout << "val = " << val << endl;
    char *buffer = &(kernel->machine->mainMemory[val]);
    //cout << "buffer = " << buffer << endl;
    numChar = kernel->machine->ReadRegister(5);
    fileId = kernel->machine->ReadRegister(6);
    //cout << "fileID = " << fileId << endl;

    status = kernel->fileSystem->ReadFile(buffer, numChar, fileId);

    kernel->machine->WriteRegister(2, (int) status);
    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 :

- 和 SC_Read 的程序幾乎一樣，差別在於是要將 buffer(已經有值的暫存空間) 內的 numChar 個 character 寫入檔案。
- 利用 kernel->fileSystem->WriteAFile 來達到 write 的功能。
- 回傳值一樣是「總共寫入到幾個 character」，存入 register 2 。
- 最後更新 program counter 值。

```
case SC_Write:
{
    val = kernel->machine->ReadRegister(4);
    //cout << "val = " << val << endl;
    char *buffer = &(kernel->machine->mainMemory[val]);
    //cout << "buffer = " << buffer << endl;
    numChar = kernel->machine->ReadRegister(5);
    fileID = kernel->machine->ReadRegister(6);
    //cout << "fileID = " << fileID << endl;

    status = kernel->fileSystem->WriteAFile(buffer, numChar, fileID);

    kernel->machine->WriteRegister(2, (int) status);
    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_Open :

- 利用 kernel->fileSystem->CloseFile 來達到 close 的功能。
- CloseFile 只需要一個參數，所以直接從 register 4 拿到那個 fileID 的值，傳入 CloseFile。
- 一樣將回傳值存入 register 2，並更新 program counter。

```
case SC_Close:
{
    fileID = kernel->machine->ReadRegister(4);
    //cout << "fileID = " << fileID << endl;

    status = kernel->fileSystem->CloseFile(fileID);

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

filesys.h :

- 在這邊可以知道，其實最後就只是呼叫了 sysdep.c 裡面的 OpenForWrite、WriteFile、Read、Close 函式而已。

```

OpenFileId OpenAFile(char *name)
{
    int fileDescriptor = OpenForReadWrite(name, FALSE);
    return fileDescriptor;
}

int WriteAFile(char *buffer, int size, OpenFileId id){
    WriteFile(id, buffer, size);
    return size;
}

int ReadFile(char *buffer, int size, OpenFileId id){
    Read(id, buffer, size);
    return size;
}

int CloseFile(OpenFileId id){
    int ret = Close(id);
    return ret >= 0 ? 1: -1;
}

```

sysdep.c :

- 從最上面的#include<sys/file.h>等等，可看出這邊只是用了原生的 C library 來幫我們做真正的 system call 而已，這也是為什麼本次作業屬於 stub file system。
- 以下是拿支援 OpenAFile 的函式 OpenForReadWrite() 來舉例。真正做 system call 的是 C library 的 open。

```

int
OpenForReadWrite(char *name, bool crashOnError)
{
    int fd = open(name, O_RDWR, 0);

    ASSERT(!crashOnError || fd >= 0);
    return fd;
}

```

Result :

- 先執行 fileIO_test1 做開與寫的動作，可看見 Success 的成功訊息。
- 再執行 fileIO_test2 做讀的動作，可看見 pass 的成功訊息。

```

henry@henry-VirtualBox:~/naches/mp1/naches-4.0_mp1/code/test$ ../userprog/naches -e fileIO_test1
Total threads number is 1
Thread fileIO_test1 is executing.
Success on creating file1.test
Machine halting!

Ticks: total 467, idle 0, system 70, user 397
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
henry@henry-VirtualBox:~/naches/mp1/naches-4.0_mp1/code/test$ ../userprog/naches -e fileIO_test2
Total threads number is 1
Thread fileIO_test2 is executing.
Passed! ^ ^
Machine halting!

Ticks: total 399, idle 0, system 60, user 339
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

```