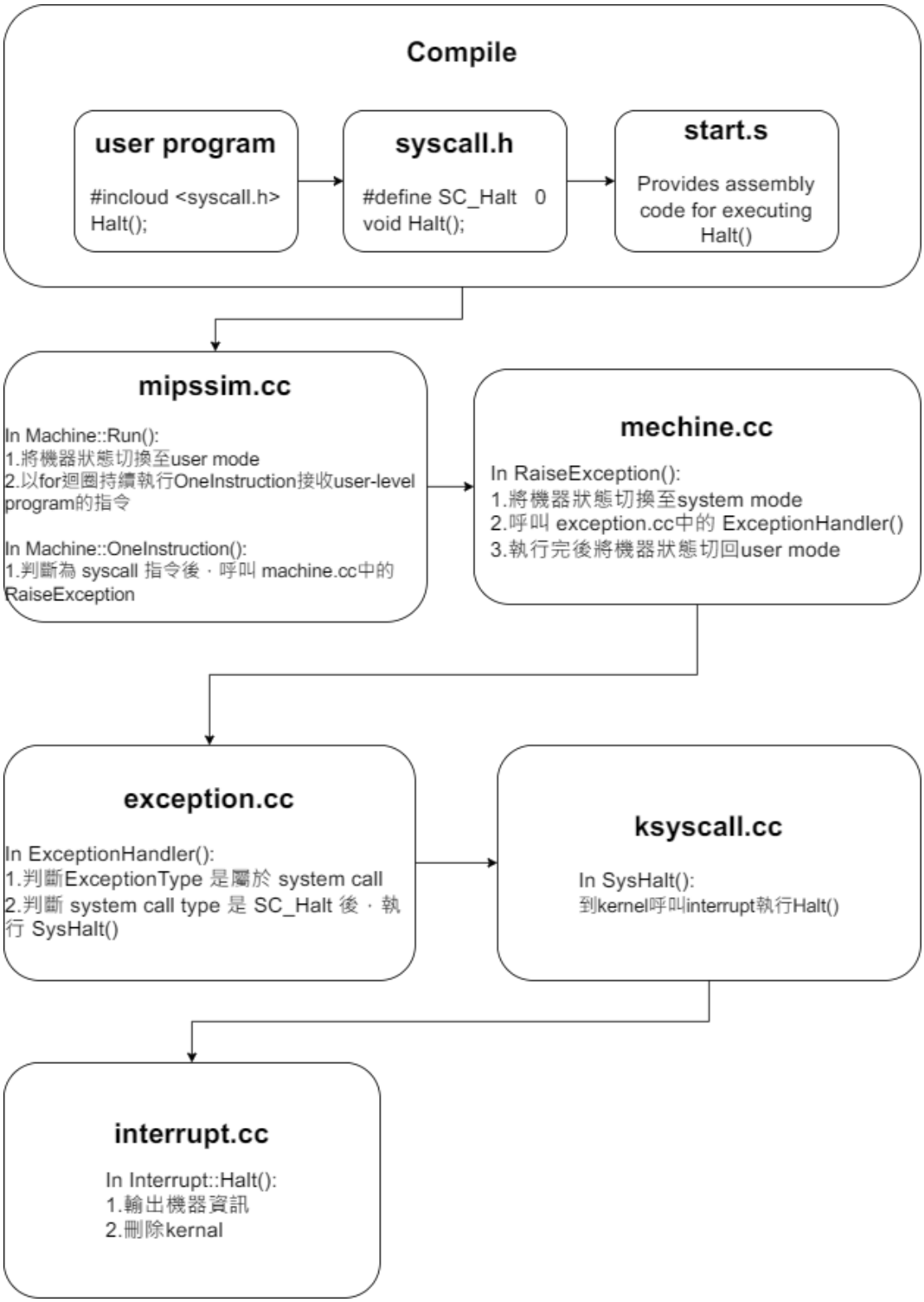
312505017 馮凡哲

312511057 林胤汶

- **Halt():**

A. Flow chart:



B. Tracing details:

- syscall.h 中將 SC_Halt 定義成 0，並宣告 Halt()。

```
#define SC_Halt         0
void Halt();
```

- system call 被 userprogram 呼叫時，NachOS 會執行與 system call 相對應的 stub，stub 被定義在 start.s 中。

```
        .globl Halt
        .ent    Halt
Halt:
        addiu $2,$0,SC_Halt
        syscall
        j       $31
        .end Halt
```

1. .globl Halt :用來標出 Halt 讓 linker 可以看見。
2. .ent Halt :開始執行 Halt。
3. add immediate unsigned 在此相當於將 SC_Halt 存入 #2 register。
4. 執行 system call 指令。
5. 返回到 #31 register 存放的地址處，該地址為用戶程序。
6. 結束 Halt。

- userprogram 啟動時 , kernal 會呼叫 mipssim.cc 中的 Machine::Run() , Run() 用來模擬程式啟動時 ， kernel 呼叫執行 user-level program 的情況。

```
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();
    }
}
```

- kernel->interrupt->setStatus(UserMode) 將 machine 的 status 設定為 UserMode。
- Run() 中的迴圈 for(;;) ，每一回都會呼叫 Machine::OneInstruction(Instruction *instr) , 用來執行一個來自 user-level 的指令。

```
void
Machine::OneInstruction(Instruction *instr)
{
    ...

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

    ...
}
```

- 當有 syscall 指令時，會呼叫 machine.cc 中的 RaiseException(SyscallException,0)，並傳入 SyscallException。

```
void
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);
    ExceptionHandler(which);            // interrupts are
enabled at this point
    kernel->interrupt->setStatus(UserMode);
}
```

- kernel->interrupt->setStatus(SystemMode) 將 status 設定為 SystemMode，接著呼叫 exception.cc 中的 ExceptionHandler() 並傳入 SyscallException，最後再由 kernel->interrupt->setStatus(UserMode) 將 status 設定為 UserMode。

```
void
ExceptionHandler(ExceptionType which)
{
    int type = kernel->machine->ReadRegister(2);
        int val;
    int status, exit, threadID, programID;
        DEBUG(dbgSys, "Received Exception " << which << " type:
" << type << "\n");
    switch (which) {
    case SyscallException:
        switch(type) {
        case SC_Halt:
                        DEBUG(dbgSys, "Shutdown, initiated by
user program.\n");
                        SysHalt();
                       cout<<"in exception\n";
                        ASSERTNOTREACHED();
                        break;
    ...
}
```

- int type = kernel->machine->ReadRegister(2) 將 #2 register 的值傳入 type。
- 接著判斷 ExceptionType 是屬於 system call，再判斷 system call 的 type 是 SC_Halt 並執行 SysHalt()。

- SysHalt 被定義在 ksyscall.h 中。

```
#ifndef __USERPROG_KSYSCALL_H__
#define __USERPROG_KSYSCALL_H__

#include "kernel.h"

#include "synchconsole.h"

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

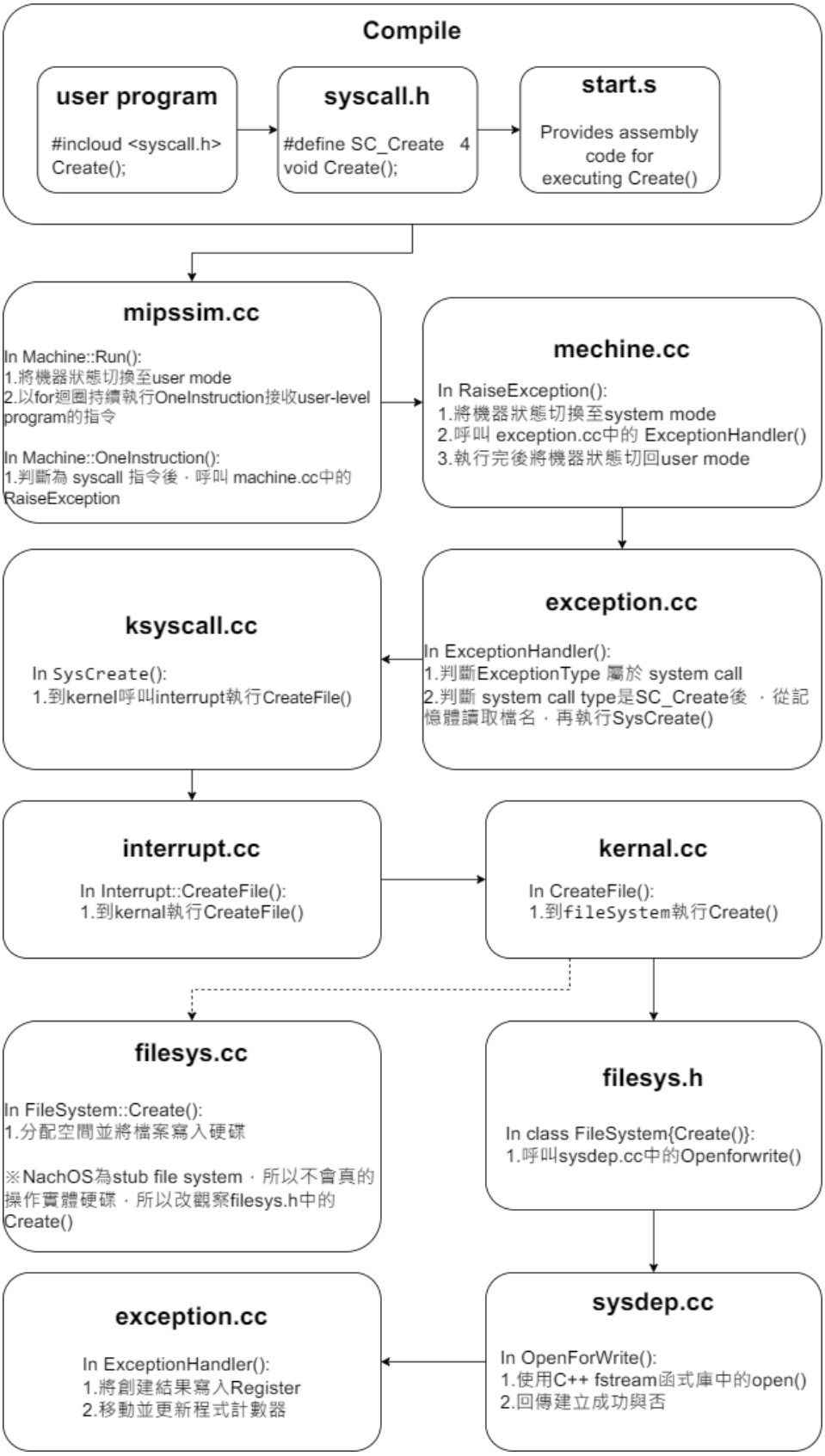
- 接著執行位於 interrupt.cc 的 Halt()。

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

- delete kernel 刪除了 kernal , 所以程式會終止。

- **Create():**

A. Flow chart:



B. Tracing details:

- syscall.h 中將 SC_Create 定義成 4，並宣告 Create()。

```
#define SC_Create       4


int Create(char *name);
```

- Create 被 userprogram 呼叫時，NachOS 會執行與 Create 相對應的 stub，同樣被定義在 start.s 中。

```
        .globl Create
        .ent    Create
Create:
        addiu $2,$0,SC_Create
        syscall
        j       $31
        .end Create
```

1. .globl Create :用來標出 Create 讓 linker 可以看見。
2. .ent Create :開始執行 Create。
3. add immediate unsigned 在此相當於將 SC_Create 存入 #2 register。
4. 執行 system call 指令。
5. 返回到 #31 register 存放的地址處，該地址為 userprogram。
6. 結束 Create。

- 與 Halt 相同，kernal 會呼叫 mipssim.cc 中的 Machine::Run(),其中的 kernel->interrupt->setStatus(UserMode) 會將 machine 的 status 設定為 UserMode，而 Run() 中的迴圈 for(;;) ，則會呼叫 Machine::OneInstruction(Instruction *instr)。
- OneInstruction(Instruction *instr) 則會再呼叫 machine.cc 中的 RaiseException(SyscallException,0)。
- RaiseException(SyscallException,0) 會先切換成 SystemMode，再呼叫 exception.cc 中的 ExceptionHandler()，結束後再切回 UserMode。

```
void
ExceptionHandler(ExceptionType which)
{
    int type = kernel->machine->ReadRegister(2);
        int val;
    int status, exit, threadID, programID;
        DEBUG(dbgSys, "Received Exception " << which << " type:
" << type << "\n");
    switch (which) {
    case SyscallException:
        switch(type) {
        ...
        case SC_Create:
                        val = kernel->machine->ReadRegister(4);
                        {
                        char *filename = &(kernel->machine-
>mainMemory[val]);
                        //cout << filename << endl;
                        status = SysCreate(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;
    ...
}
```

- 先將 #2 register 的值傳入 type，再判斷 ExceptionType，最後判斷 system call 的 type。
- val = kernel->machine->ReadRegister(4) 讀取 #4 register 的值並存入 val。
- char *filename = &(kernel->machine->mainMemory[val]) 宣告一個指標變數 filename 指向 mainMemory[val] 的記憶體位址。
- 接著執行 SysCreate(filename)，回傳值存入 status ; SysCreate() 定義在 ksyscall.h 中。

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

- 回傳值是位於 interrupt.cc 的 CreateFile()。

```
int
Interrupt::CreateFile(char *filename)
{
    return kernel->CreateFile(filename);
}
```

- 執行位於 kernal.cc 的 CreateFile()。

```
int Kernel::CreateFile(char *filename)
{
        return fileSystem->Create(filename);
}
```

- 再來看 filesys.cc 裡面，有這行程式碼。

```
#ifndef FILESYS_STUB
```

- 因為我們是使用 stub file system, 所以只須執行 filesys.h。

```
#ifdef FILESYS_STUB

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;
        }
...
```

- 主要是在呼叫 sysdep.cc 中的 OpenForWrite。

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

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

- 其中的 open() 就是 C++ fstream 函式庫的函式。
- 回到 ExceptionHandler(), 可以看到執行完 SysCreate(filename) 後，進行了一連串 kernel->machine->WriteRegister() 指令，這個指令位於 machine.cc 內。

```
void
Machine::WriteRegister(int num, int value)
{
    ASSERT((num >= 0) && (num < NumTotalRegs));
    registers[num] = value;
}
```

- 主要是將 program counter 設置在 register 中。

- **Makefile**

  - 首先看 Makefile.dep，裡面主要定義了:

1. GCCDIR(GCC 編譯器位址)
2. LDFLAGS(Linker 的選項)
3. ASFLAGS(Assembler 的選項)
4. COFF2NOFF(coff2noff program 的路徑)

```
GCCDIR = ../../usr/local/nachos/bin/decstation-ultrix-
LDFLAGS = -T script -N
ASFLAGS = -mips2
COFF2NOFF = ../../coff2noff/coff2noff.x86Linux
```

  - 再看 Makefile。

```
include Makefile.dep

CC = $(GCCDIR)gcc
AS = $(GCCDIR)as
LD = $(GCCDIR)ld

INCDIR =-I../userprog -I../lib
CFLAGS = -G 0 -c $(INCDIR) -B../../usr/local/nachos/lib/gcc-lib
/decstation-ultrix/2.95.2/ -B../../usr/local/nachos/decstation-
ultrix/bin/
```

  - 首先指定 CC 、 AS 、 LD 的位址，分別是:
    …/…/usr/local/nachos/bin/decstation-ultrix-gcc
    …/…/usr/local/nachos/bin/decstation-ultrix-as
    …/…/usr/local/nachos/bin/decstation-ultrix-ld
  - INCDIR 主要是在添加.h 檔的搜索路徑，CFLAGS 則是 C Complier 的選項。

- 接著以 halt 為例，觀察編譯情形。

```
start.o: start.S ../userprog/syscall.h
        $(CC) $(CFLAGS) $(ASFLAGS) -c start.S

halt.o: halt.c
        $(CC) $(CFLAGS) -c halt.c
halt: halt.o start.o
        $(LD) $(LDFLAGS) start.o halt.o -o halt.coff
        $(COFF2NOFF) halt.coff halt
```

- 先對 start.S 進行 assemble 處理，但不進行 linking，最後生成 start.o。
- 再對 halt.c 進行 preprocessing 、 compilation 、 assemble，產生 halt.o。
- 最後將 halt.o 和 start.o 進行 linking 得到 halt.coff，再轉換成 halt。

- **PrintInt system call implementation:**
  - 在 syscall.h 中進行以下定義:

```
void SysPrintInt(int val)
...
{
  kernel->synchConsoleOut->PutInt(val);
}
```

  - 在 exception.cc 中新增新的 case 處理 PrintInt

```
void
ExceptionHandler(ExceptionType which)
{
...
        case SC_PrintInt:
                DEBUG(dbgSys, "Print Int\n");
                val = kernel->machine->ReadRegister(4);

                SysPrintInt(val);

                kernel->machine->WriteRegister(PrevPCReg, kerne
l->machine->ReadRegister(PCReg));
                kernel->machine->WriteRegister(PCReg, kernel->m
achine->ReadRegister(PCReg) + 4);
                kernel->machine->WriteRegister(NextPCReg, kerne
l->machine->ReadRegister(PCReg)+4);
                return;
                ASSERTNOTREACHED();
                        break;
...
}
```

  - 在 ksyscall.h 中指定呼叫的函式

```
void SysPrintInt(int val)
{
  kernel->synchConsoleOut->PutInt(val);
}
```

- 在 synchconsole.h 新增定義

```
void PutInt(int value);
```

- 在 synchconsole.cc 以 sprintf 將數字寫入 str 並使用 do while 將數字以字元的方式逐個輸出，技術則是呼叫 numConsoleCharsWritten++

```
void
SynchConsoleOutput::PutInt(int value)
{
    char str[15];
    int idx=0;

    sprintf(str, "%d\n\0", value);
    lock->Acquire();
    kernel->stats->numConsoleCharsWritten++;
    do{
        consoleOutput->PutChar(str[idx]);
        idx++;
        waitFor->P();
    } while (str[idx] != '\0');
    lock->Release();
}
```

- **File I/O system call implementation:**
- 參考其他 systemcall，在 exception.cc 中新增 4 個 case:
  1. SC_Open
  2. SC_Write
  3. SC_Read
  4. SC_Close

```
case SC_Open:
    {
    DEBUG(dbgSys, "Open\n");
    //cout<<"open test"<<"\n";
    val = kernel->machine->ReadRegister(4);
    char *filename = &(kernel->machine->mainMemory[val]);
    DEBUG(dbgSys, "Filename " << filename << "\n");
    status = SysOpen(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;
    }
```

```
case SC_Write:
    DEBUG(dbgSys, "Write\n");
    val = kernel->machine->ReadRegister(4);
    buffer = &(kernel->machine->mainMemory[val]);
    DEBUG(dbgSys, "Buffer " << buffer << "\n");
    numChar = kernel->machine->ReadRegister(5);
    fileID = kernel->machine->ReadRegister(6);
    DEBUG(dbgSys, "fileID " << fileID << "\n");
    status = SysWrite(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;
```

```
case SC_Read:
    DEBUG(dbgSys, "Read\n");
    val = kernel->machine->ReadRegister(4);
    buffer = &(kernel->machine->mainMemory[val]);
    DEBUG(dbgSys, "Buffer " << buffer << "\n");
    numChar = kernel->machine->ReadRegister(5);
    fileID = kernel->machine->ReadRegister(6);
    DEBUG(dbgSys, "fileID " << fileID << "\n");
    status = SysRead(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;
```

```
case SC_Close:
    {
    DEBUG(dbgSys, "Close\n");
    //cout<<"close test"<<"\n";
    fileID = kernel->machine->ReadRegister(4);
    //cout<< "fileID : "<< fileID << "\n";
    DEBUG(dbgSys, "fileID " << fileID << "\n");
    status = SysClose(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;
    }
```

- 裡面分別呼叫了:
  1. SysOpen()
  2. SysWrite()
  3. SysRead()
  4. SysClose()
- 我們在 ksyscall.h 定義了這些函式:

```
OpenFileId SysOpen(char *name)
{
  return kernel->fileSystem->OpenAFile(name);
}
int SysWrite(char *buffer, int size, OpenFileId id)
{
  return kernel->fileSystem->Write_File(buffer, size, id);
}

int SysRead(char *buffer, int size, OpenFileId id)
{
  return kernel->fileSystem->Read_File(buffer, size, id);
}
int SysClose(OpenFileId id)
{
  return kernel->fileSystem->CloseFile(id);
}
```

- 回傳值:
  1. fileSystem->OpenAFile()
  2. fileSystem->Write_File()
  3. fileSystem->Read_File()
  4. fileSystem->CloseFile()

- 定義 filesys.h:

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

    return fileDescriptor;
}

    int Write_File(char *buffer, int size, OpenFileId id){
            if(size <= 0){return -1;}
            int write_count = WriteFile(id, buffer, size);
            //return -1;
            if(write_count == size){
                    return size;
            }else{
                    return -1;
            }
    }

    int Read_File(char *buffer, int size, OpenFileId id){
            if(size <= 0){return -1;}
            int read_count = Read(id, buffer, size);
            //return -1;
            if(read_count == size){
                    return size;
            }else{
                    return -1;
            }
    }

    int CloseFile(OpenFileId id){
    int ret = Close(id);

            if (ret >= 0){return 1;}
            return 0;
}
```

- 呼叫以下函式，並定義在 sysdep.cc 中:
  1. OpenForReadWrite()
  2. WriteFile
  3. Read
  4. Close

```
int
OpenForReadWrite(char *name, bool crashOnError)
{
    int fd = open(name, O_RDWR, 0);
    FileCounter = FileCounter + 1;
    //cout << "openFileCounter : " << FileCounter<<"\n";
    if ( FileCounter > 20 ) {
        FileCounter = FileCounter - 1;
        //cout << " Failed "<<"openFileCounter : " << FileCounter<<"\n";
        return -1 ;}
    if ( fd < 0 ) { return -1 ;}
    ASSERT(!crashOnError || fd >= 0);

    return fd;
}
```

- 我們宣告了 FileCounter，用來計算開啟檔案數 ，因為還要考慮 userprogram 本身 ，所以初始值設為 - 1
- 每開啟一個檔案就會+1，當檔案數超過 20，則會回傳 - 1

```
int
WriteFile(int fd, char *buffer, int nBytes)
{
    int retVal = write(fd, buffer, nBytes);
    return retVal;
    ASSERT(retVal == nBytes);
}


int
Read(int fd, char *buffer, int nBytes)
{
    int retVal = read(fd, buffer, nBytes);
    //cout<< "retVal : "<<retVal << "\n";
    return retVal;
    ASSERT(retVal == nBytes);
}
```

- 由於是 stub file system 所以最底層直接呼叫 c++函式庫的 read 及 write。

```
int
Close(int fd)
{
    int retVal = close(fd);
    FileCounter = FileCounter - 1;

    if (retVal < 0) {
        FileCounter = FileCounter + 1;

        return -1;}

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

- 當檔案關閉成功，則 FileCounter - 1