MP1_report_37

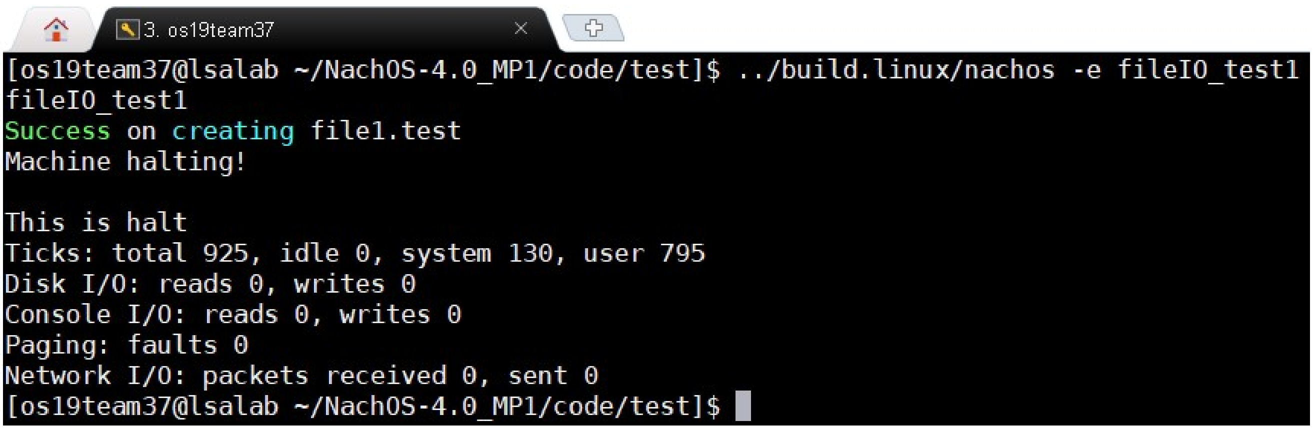
Team Member & Contributions

- ## 資應碩二 **107065522** 陳子潔
- ## 數學大四 **105021127** 徐迺茜

工作項目	分工
Trace Code	陳子潔 & 徐迺茜
文件撰寫(Part I)	徐迺茜
文件撰寫(Part II)	陳子潔
功能實作(Open, Write, Read)	陳子潔
功能實作(Close)	徐迺茜
功能測試&Debug	陳子潔&徐迺茜

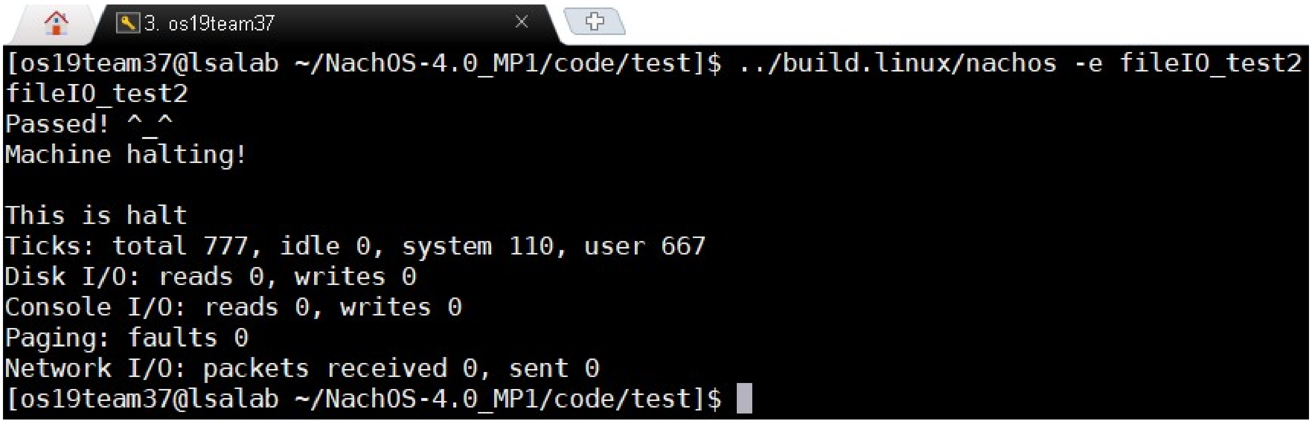
Verification:

1. First use the command “.../build.linux/nachos -e fileIO_test1” to write a file.
    - Result of fileIO_test1:



```
[os19team37@lsalab ~/NachOS-4.0_MP1/code/test]$ ../build.linux/nachos -e fileIO_test1
fileIO_test1
Success on creating file1.test
Machine halting!

This is halt
Ticks: total 925, idle 0, system 130, user 795
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
[os19team37@lsalab ~/NachOS-4.0_MP1/code/test]$
```

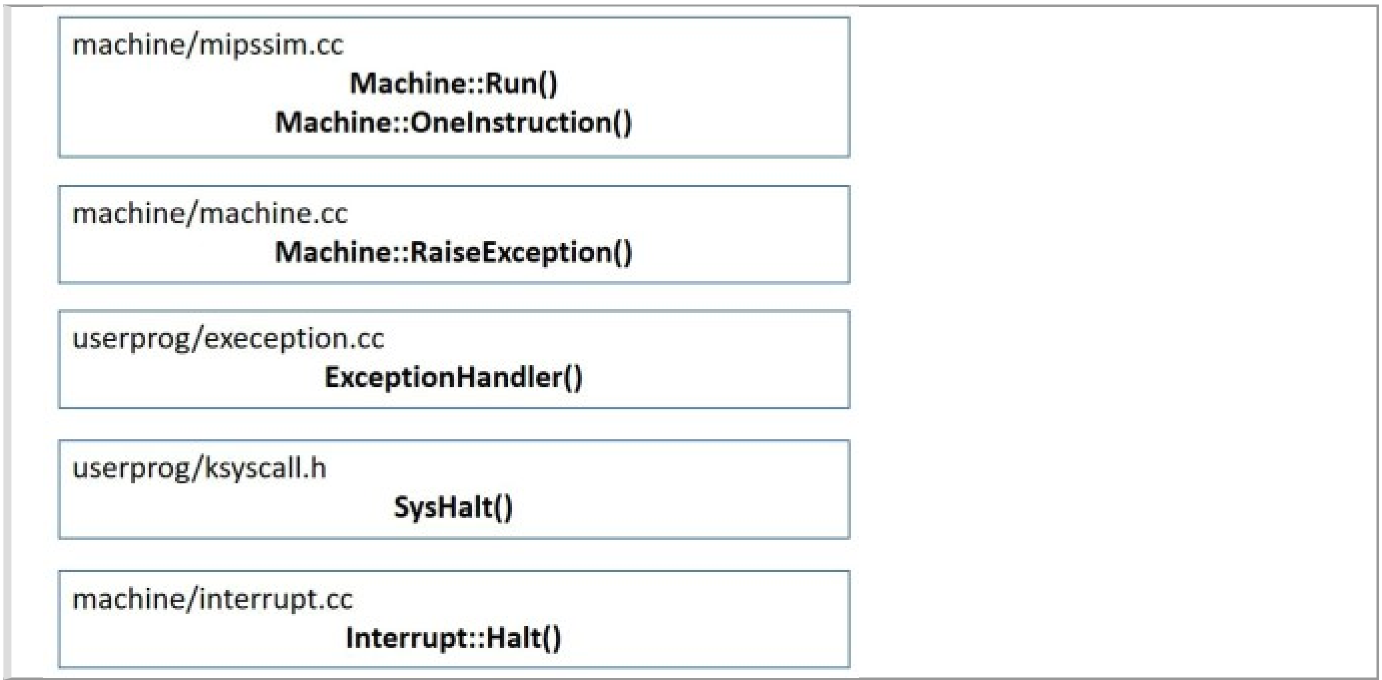
2. Then use the command “.../build.linux/nachos -e fileIO_test2” to read the file
    - Result of fileIO_test2:



```
[os19team37@lsalab ~/NachOS-4.0_MP1/code/test]$ ../build.linux/nachos -e fileIO_test2
fileIO_test2
Passed! ^_^
Machine halting!

This is halt
Ticks: total 777, idle 0, system 110, user 667
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
[os19team37@lsalab ~/NachOS-4.0_MP1/code/test]$
```

Explain how system calls work in NachOS as requested in Part II-1.

(a) SC_Halt



- 當User mode 調用system call 接口時，Nachos 會執行與此system call 相對應的stub
- system call 的stub定義在start.s中，而SC_Halt對應到的stub為:

start.s

```
.globl Halt  # 聲明為外部函數
.ent   Halt  # Halt 開始執行
Halt:
    addiu $2,$0,SC_Halt # 將system call 呼叫caae num 存在r2
    syscall             # 所有system call的參數會自動存在r4,r5,r6,r7
    j   $31
.end Halt
```

- 這裡主要完成3件事:
  1. 將system call的type寫入2號register (在此是SC_Halt，而SC_Halt在system.h中被定義成是0)
  2. 執行system call指令
  3. 返回到31號register存放的地址處，該地址為用戶程序

Machine::Run()

- 當系統執行syscall指令時會丟到mipssim.cc的Machine::Run()，程式碼如下:

```
Instruction *instr = new Instruction;       #storage for decoded instructio

if (debug->IsEnabled('m')) {
    cout << "Starting program in thread: " << kernel->currentThread->getNam
cout << ", at time: " << kernel->stats->totalTicks << "\n";
}
kernel->interrupt->setStatus(UserMode);
#Program平常是跑在UserMode下面，需要Syscall時會轉換為KernelMode

for (;;) {

    OneInstruction(instr); #instr被傳入此函數，觀察此函數

kernel->interrupt->OneTick();

if (singleStep && (runUntilTime <= kernel->stats->totalTicks))
    Debugger();
}
```

- 處理器會將捕獲到的syscall指令丟到OneInstruction()執行

Machine::OneInstruction()

- 在mipssim.cc的OneInstruction()函數是模擬CPU的逐條指令執行過程，進入的代碼如下:

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

- 可以看到處理器發現syscall指令時，會調用RaiseException(SyscallException,0)拋出一個SyscallException異常

RaiseException()

- 進入到位於machine.cc的異常處理函數RaiseException()，可以發現將SyscallException傳入了ExceptionHandler()函數中

- 另外，kernal->interrupt->setStatus(SystemMode)這行程式碼，代表了此時從User Mode轉變為 Kernal Mode
- 而kernal->interrupt->setStatus(UserMode)代表了ExceptionHander()執行完後，要從Kernal Mode轉回User Mode

```
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);#從User Mode轉變到Kernal Mode
    ExceptionHandler(which);          # interrupts are enabled at this poin
    kernel->interrupt->setStatus(UserMode);  #從Kernal Mode轉回User Mode
}
```

ExceptionHandler()

- 進到exception.cc中，ExceptionHandler()函數的程式碼如下:

```
void
ExceptionHandler(ExceptionType which)
{
char ch;
int val;
int type = kernel->machine->ReadRegister(2);
#從r2取出system call type 存進 type (在此type是SC_Halt)

int status, exit, threadID, programID, fileID, numChar;

switch (which) {     #判斷是system call或是其他Exception Type
case SyscallException:

  switch(type) {     #判斷system call是甚麼type，並執行system call 要處理的事情
  case SC_Halt:
    {
        DEBUG(dbgSys, "Shutdown, initiated by user program.\n");
        SysHalt();  #具體處理在ksyscall.h中的SysHalt()
        cout<<"in exception\n";

        ASSERTNOTREACHED();
        #相當於ASSERT(False)功能，Print message and dump core，在debug.h中

      break;
```

```
        break;
    }
```

- 可以看到此函數首先判斷傳入函數的Exception Type屬於哪一種
- Exception Type定義於Machine.h中:

```
enum ExceptionType {
    NoException,
    SyscallException,
    PageFaultException,
    ReadOnlyException,
    BusErrorException,
    AddressErrorException,
    OverflowException,
    IllegalInstrException,
     NumExceptionTypes
};
```

- 若是SyscallException則會根據從2號register取出的system call 的類型(存進type裡)判斷
- 並執行該system call type要處理的事情
- 而SC_Halt的具體處理會在SysHalt()

SysHalt()

- 進到位於ksyscall.h的SysHalt()函數
- 可以看到要跳到位於interrupt.cc的Halt()函數

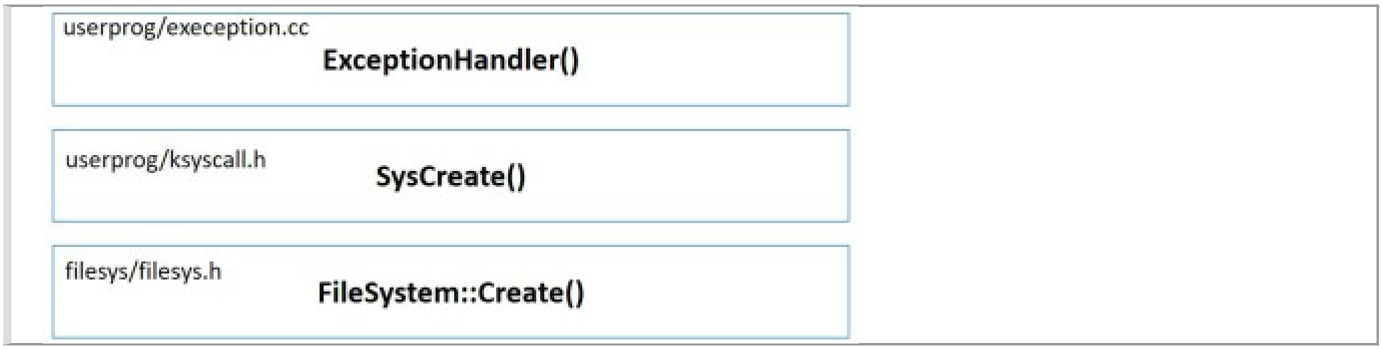
```
void SysHalt()
{
  kernel->interrupt->Halt();
}
```

Halt()

- 進到位於interrupt.cc的Halt()函數，可以看到delete kernel
- 因為kernel連結了所有的程式及函數，所以刪除了kernel程式就停止了。

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

(b) SC_Create



- SC_Create在ExceptionHandler()函數前的運作方式和SC_Halt一樣
- 因此我們從ExceptionHandler()函數開始trace code

ExceptionHandler()

- 位在exception.cc裡的ExceptionHandler()函數程式碼如下:

```
void
ExceptionHandler(ExceptionType which)
{
char ch;
int val;
int type = kernel->machine->ReadRegister(2);
#從r2取出system call type 存進 type (在此type是SC_Create)

int status, exit, threadID, programID, fileID, numChar;
// 判斷是system call或是其他Exception Type
switch (which) {
case SyscallException:
// 判斷system call 是甚麼type，並執行system call 要處理的事情
  switch(type) {
      case SC_Create:
    {
        val = kernel->machine->ReadRegister(4);
        // 從4號register取出此system call的參數值

        char *filename = &(kernel->machine->mainMemory[val]);
        // cout << filename << endl;
        // 具體處理在ksyscall.h中的SysCreate()
        status = SysCreate(filename);

        kernel->machine->WriteRegister(2, (int) status);
        // 將status寫入registers[2]中

/* set previous programm counter (debugging only)*/
kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCR
kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg)
kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCR
        return;
        ASSERTNOTREACHED();
    // 相當於ASSERT(False)功能，Print message and dump core，在debug.h中
    break;
    }
```

- 首先判斷傳入函數的Exception Type
- 若是system call 則會根據從2號register取出的system call的類型(存進type裡)，判斷並執行對應system call type要處理的事情
- 在此可以看到SC_Create的具體處理會在SysCreate()
- 在做完SysCreate()之後，可以看到程式去machine.cc執行WriteRegister()，此函數程式碼如下:

```
void
Machine::WriteRegister(int num, int value)
{
    ASSERT((num >= 0) && (num < NumTotalRegs));
    #位於debug.h，若不符合括弧內的條件，則Print message and dump core

    registers[num] = value; #將value寫入regisers[num]中
}
```

- 可以看到WriteRegister()的用途是將value寫入user program register中，在ExceptionHandler()中主要是被用來設置先前的program counter，單純debug用

SysCreate()

- 進到位於ksyscall.h的SysCraete()函數，可以看到要跳到位於fileSystem.cc的Create()函數

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

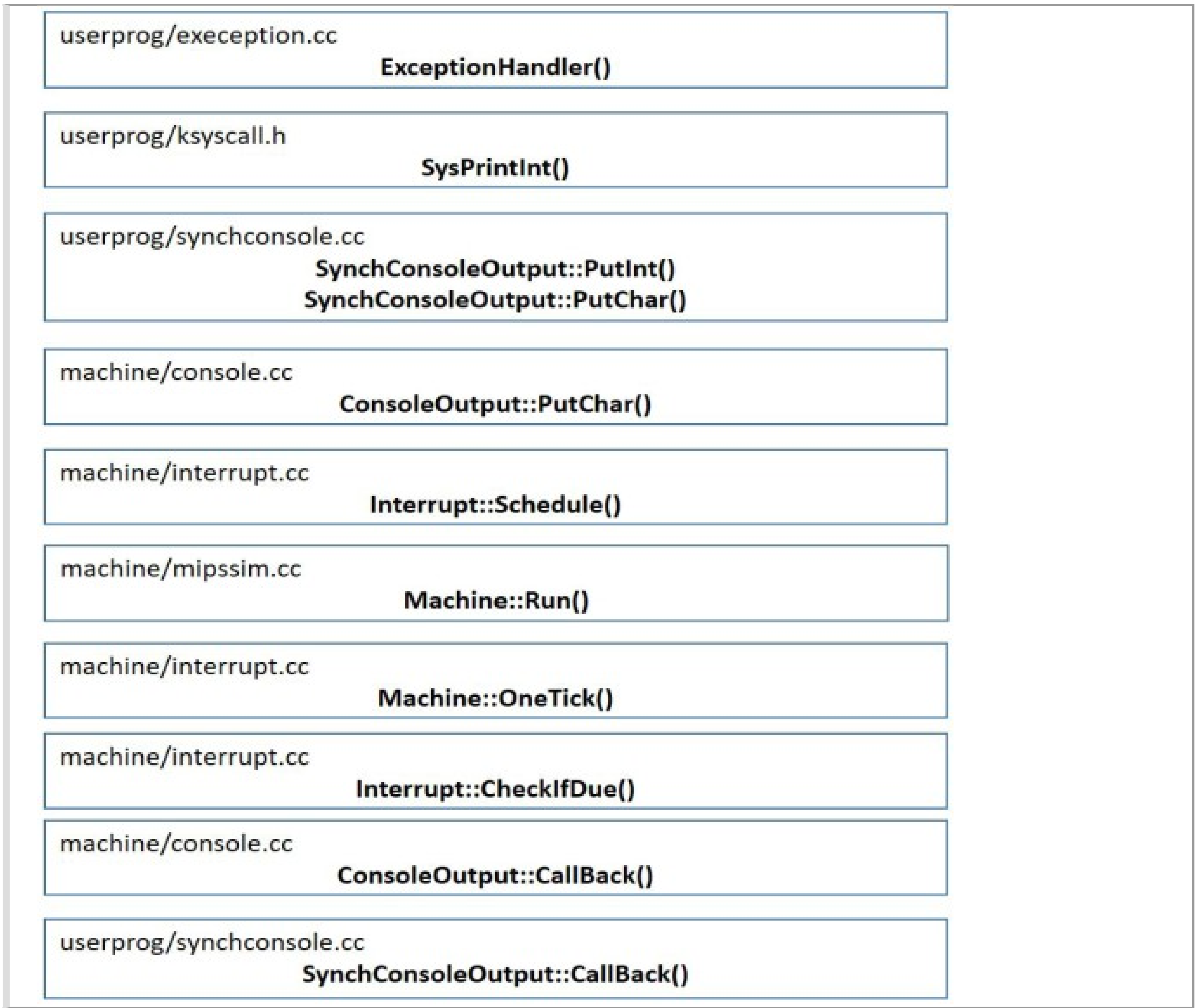
FileSystem::Create()

- 因為FILESYS_STUB已經被define了 (本次作業使用stub file system)
- 故只需追蹤filesys.h這個檔案的程式碼即可

```
#ifdef FILESYS_STUB
// Temporarily implement file system calls as
// calls to UNIX, until the real file system
// implementation is available

typedef int OpenFileId;

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;
    }
```

- 注意上面Line 15呼叫了sysdep.c裡面的OpenForWrite函式
- 而此函式其實就是再呼叫一次C原生的stdlib裡面的open函式
- 故稱之為stub file system

(c) SC_PrintInt



- 同樣的SC_PrintInt在ExceptionHandler()函數前的運作方式和SC_Halt一樣
- 因此我們從ExceptionHandler()函數開始trace code

ExceptionHandler()

- 進到exception.cc中，ExceptionHandler()函數的程式碼如下:

```
void
ExceptionHandler(ExceptionType which)
{
    char ch;
    int val;
    int type = kernel->machine->ReadRegister(2);
    #從r2取出system call type 存進 type (在此type是SC_PrintInt)

    int status, exit, threadID, programID, fileID, numChar;

    switch (which) {  #判斷是system call或是其他Exception Type
    case SyscallException:
      switch(type) {  #判斷system call是甚麼type，並執行system call 要處理的事忄
            case SC_PrintInt:
        {
                val = kernel->machine->ReadRegister(4);
            #從4號register取出此system call的參數值

                SysPrintInt(val); #具體處理在ksyscall.h中的SysPrintInt()

 // Set Program Counter
 kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PC
 kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg)
 kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PC
                return;
                ASSERTNOTREACHED();
            #相當於ASSERT(False)功能，Print message and dump core，在debug.h中

          break;
        }
            case SC_MSG:
        {
```

- 首先判斷傳入函數的Exception Type
- 若是system call 則會根據從2號register取出的system call的類型(存進type裡)，判斷並執行對應system call type要處理的事情
- 在此可以看到SC_PrintInt的具體處理會在SysPrintInt()
- 在做完SysPrintInt()之後，可以看到程式去machine.cc執行WriteRegister()，在此是被用來設置先前的program counter，單純debug用，詳細說明可見SC_Create的ExceptionHandler()

SysPrintInt()

- 進到位於ksyscall.h的SysPrintInt()函數，可以看到要跳到位於synchConsoleOut.cc的PutInt()函數

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

SynchConsoleOutput::PutInt()

- 進到synchconsole.cc中，SynchConsoleOutput::PutInt()函數的程式碼如下:

```
void
SynchConsoleOutput::PutInt(int value)
{
    char str[15];
    int idx=0;
    //sprintf(str, "%d\n\0", value);  the true one
    sprintf(str, "%d\n\0", value); //simply for trace code
    lock->Acquire(); #鎖定物件，開始執行同步化
    do{

        consoleOutput->PutChar(str[idx]);
        #一個一個字元丟入consoleOutput.cc裡的PutChar()函數

       idx++; #換下一個字元執行

        waitFor->P(); #wait for EOF or a char to be available.

    } while (str[idx] != '\0');
    lock->Release(); #執行完同步化，解除鎖定
}
```

- 首先用sprint將value存到str，變成字元型態(從ExceptionHandler()可以知道，value是從4號register取出此system call的參數值)
- 接著利用lock->Acquire()鎖定物件，開始執行同步化。只有取得鎖定的執行緒才可以進入同步區，未取得鎖定的執行緒則必須等待，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()
- 執行完同步化後，用lock->Release()解除鎖定，讓其他物件有機會取得鎖定

SynchConsoleOutput::PutChar()

```
void
SynchConsoleOutput::PutChar(char ch)
{
    lock->Acquire();
    consoleOutput->PutChar(ch);
    waitFor->P();
    lock->Release();
}
```

- 這個函數和上一個SynchConsoleOutput::PutInt()的差別是傳入的參數直接是一個字元，其他功能都和SynchConsoleOutput::PutInt()一樣

ConsoleOutput::PutChar()

- 在SynchConsoleOutput::PutInt()進行同步化時，從4號register取出的system call參數值會進入console.cc裡的PutChar()，以下是它的程式碼:

```
void
ConsoleOutput::PutChar(char ch)
{
    ASSERT(putBusy == FALSE);
    #debug，如果括號內的條件為假，則打印一條消息並轉存核心(core)

    WriteFile(writeFileNo, &ch, sizeof(char)); #將數據寫入一個文件

    putBusy = TRUE; #不能有其他事一起做

    kernel->interrupt->Schedule(this, ConsoleTime, ConsoleWriteInt);
    #進入interrupt.cc裡的Schedule()
}
```

- 首先利用WriteFile()，將數據寫入一個文件中
- 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()，安排程式預定被CPU執行的時間

Interrupt::Schedule()

- 位在interrupt.cc裡的Schedule()函數程式碼如下:

```
void
Interrupt::Schedule(CallBackObj *toCall, int fromNow, IntType type)
{
    int when = kernel->stats->totalTicks + fromNow;
    #現在的時間+多久後要發生interrupt的時間

    PendingInterrupt *toOccur = new PendingInterrupt(toCall, when, type);
    #Initialize a hardware device interrupt that
    #is to be scheduled to occur in the near future

    ASSERT(fromNow > 0);

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

- "toCall"是interrupt時要被執行的對象
- "fromnow"是指在模擬時間內interrupt發生的時間
- "type"是產生interrupt的硬體設備
- 這個函數先記錄了interrupt何時要被執行，然後在PendingInterrupt List裡插入要被執行的interrupt

Machine::Run()

- 當安排好CPU執行時間，就只要等待CPU來執行它，執行的程式會在mipssim.cc裡的Machine::Run()，以下是它的程式碼:

```
Instruction *instr = new Instruction;       #storage for decoded instructio

if (debug->IsEnabled('m')) {
    cout << "Starting program in thread: " << kernel->currentThread->getNam
    cout << ", at time: " << kernel->stats->totalTicks << "\n";
}
    kernel->interrupt->setStatus(UserMode);
    #Program平常是跑在UserMode下面，需要Syscall時會轉換為KernelMode
    for (;;) {

        OneInstruction(instr); #instr被傳入此函數，模擬CPU逐一執行

        kernel->interrupt->OneTick();

        if (singleStep && (runUntilTime <= kernel->stats->totalTicks))
            Debugger();
    }
```

- 在SC_Halt那裡有提到，OneInstruction()模擬CPU逐一執行任務的功能

- 在執行完OneInstruction()後，會進到interrupt.cc裡的OneTick()函數

Interrupt::OneTick()

- 位在interrupt.cc裡的Interrupt::OneTick()函數程式碼如下:

```
void
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(); #釋放kernel目前的thread

        status = oldStatus;
    #執行READY Queue(即對於在主記憶體內的所有行程
    #而且就緒等待執行的行程是保存在此就緒佇列)第一個行程
    }
}
```

- 透過程式碼可以知道OneTick()有以下幾種功能:
  1. 會讓系統時間往前一個時刻，來模擬時間往前的行為
  2. 這個函數能夠設定中斷狀態，並釋放目前Thread，然後執行下一個Thread
- 整體來說，NachOS interrupt controller模擬一個時鐘，這個時鐘從NachOS啟動時開始計數(ticks)，作為NachOS的系統時間。當NachOS模擬的CPU執行完成一條指令，ticlks=ticks+1，當中斷狀態從disabled轉到enabled，ticks+10。而此函數就是在模擬時鐘走一個時刻。

Interrupt::CheckIfDue()

- 位在interrupt.cc裡的Interrupt::CheckIfDue()函數程式碼如下:

```
bool
Interrupt::CheckIfDue(bool advanceClock)
{
    PendingInterrupt *next;
    Statistics *stats = kernel->stats;

    ASSERT(level == IntOff);          // interrupts need to be disabled,

    // to invoke an interrupt handler
    if (debug->IsEnabled(dbgInt)) {
    DumpState();
    }

    if (pending->IsEmpty()) {         // no pending interrupts
      return FALSE;
    }

    next = pending->Front();

    if (next->when > stats->totalTicks) {
        if (!advanceClock) {          // not time yet
            return FALSE;
        }
        else {     // advance the clock to next interrupt
            stats->idleTicks += (next->when - stats->totalTicks);
            stats->totalTicks = next->when;
// UDelay(1000L); // rcgood - to stop nachos from spinning.
              }
    }

    if (kernel->machine != NULL) {
        kernel->machine->DelayedLoad(0, 0);
    }


    inHandler = TRUE;

    do {
        next = pending->RemoveFront();
        // pull interrupt off list
        // Pull interrupt from pending queue


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

              delete next;
    } while ( !pending->IsEmpty() && (pending->Front()->when <= stats->tota

    inHandler = FALSE;
    return TRUE;
}
```

- 此函數的目的是檢查全部的interrupts是否有如預期的發生，並且解決
- 當所有interrupts解決完，回傳TRUE

ConsoleOutput::CallBack()

- 位在console.cc裡的ConsoleOutput::CallBack()函數程式碼如下:

```
void
ConsoleOutput::CallBack()
{
    putBusy = FALSE;
    kernel->stats->numConsoleCharsWritten++;
    callWhenDone->CallBack();
}
```

- 當下一個字元可以輸出到顯示器時，模擬器將調用此函數

SynchConsoleOutput::CallBack()

- 位在synchconsole.cc裡的ConsoleOutput::CallBack()函數程式碼如下:

```
void
SynchConsoleOutput::CallBack()
{
    waitFor->V();
}
```

- 如果可以安全的發送下一個字元，調用interrupt，並送到顯示器

Explain your implementation as requested in Part II-2.

- 根據投影片的Hints，可得知
    1. Hint1: Files to be modified are
        - [x] test/start.S
        - [x] userprog/syscall.h
        - [x] userprog/exception.cc
        - [x] userprog/ksyscall.h
        - [x] filesys/filesys.h
    2. Hint2: You can use the file operations defined in lib/sysdep.cc
- 以下依序解釋實作內容

syscall.h

- 首先進到此檔案，將以下四行程式碼的註解拿掉
  - #define SC_Open 6
  - #define SC_Read 7
  - #define SC_Write 8
  - #define SC_Close 10

start.S

- 接著修改start.s，依樣畫葫蘆地複製了4份Code來修改SC代碼

```
        .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
```

- 根據[17]，.globl與.ent關鍵字分別為外部檔案的進入點以及方便Debug的marks
- 而從test/Makefile看來，NachOS應該是透過將start.S與syscall.h以及user program聯結起來(linking?)來達到呼叫system call的

```
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
```

- 每個case對應到的代碼都是在syscall.h所定義的
- 前面的starts.s會從 $2 裡面讀取暫存器的值，並呼叫對應的syscall
- 而Register 4, 5, 6, 7則依序儲存4個參數($a0, $a1, $a2, $a3)

exception.c

- 接著進到ExceptionHandler(ExceptionType which)此函式

```
int type = kernel->machine->ReadRegister(2);
```

- 此行將剛剛start.s所寫入的SC代碼讀出來

```
1  switch (which) {
2     case SyscallException:
3
4     switch(type) {
```

- 以上第一行which用來判斷Exception的Type(詳細的定義參見machine.h)
- 本次作業的ExceptionType都是SystemCall
- 於是根據type的數值，來執行不同的Case
- 依序新增四個case

```
case SC_Open:
{
    DEBUG(dbgSys, "Open\n");
// 檔案操作可參考上面的SC_CREATE case
    val = kernel->machine->ReadRegister(4);
    char *filename = &(kernel->machine->mainMemory[val]);
    DEBUG(dbgSys, "Filename " << filename << "\n");
// 191012[J]: systemcall細節其實是在ksyscall裡實作
    status = SysOpen(filename);
    kernel->machine->WriteRegister(2, (int) status);

// 191012[J]: 每一個功能結束後都要 Set Program Counter。之後要依序return, asser
// set previous programm counter (debugging only)
    kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister
    kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCR
    kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister
    return;
    ASSERTNOTREACHED();
    break;
}

case SC_Write:
{
    DEBUG(dbgSys, "Write\n");
    val = kernel->machine->ReadRegister(4);
    char *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);

// 191012[J]: 每一個功能結束後都要 Set Program Counter。之後要依序return, asser
// set previous programm counter (debugging only)
    kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister
    kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCR
    kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister
    return;
    ASSERTNOTREACHED();
    break;
}

case SC_Read:
{
    DEBUG(dbgSys, "Read\n");
    val = kernel->machine->ReadRegister(4);
    char *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);
```

```
// set previous programm counter (debugging only)
    kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister
    kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCR
    kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister
    return;
    ASSERTNOTREACHED();
    break;
}

case SC_Close:
{
    DEBUG(dbgSys, "Close\n");
    fileID = kernel->machine->ReadRegister(4);
    DEBUG(dbgSys, "fileID " << fileID << "\n");
    status = SysClose(fileID);
    kernel->machine->WriteRegister(2, (int) status);
// set previous programm counter (debugging only)
    kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister
    kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCR
    kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister
    return;
    ASSERTNOTREACHED();
    break;
}
```

- 簡單來說，參數都是儲存在Register4, 5, 6, 7裡面
- 可能是地址、也可能是value (size, FileID...,etc.)
- 根據不同的函式規格要求，我們用不同的方法來取值
- 以Open為例

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

- 首先到$4取值 (這邊是存filename在記憶體中的Address)

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

- 由於*filename是一個指標，指派一個儲存filename的地址給他，用以模擬從記憶體Load值的動作

```
status = SysOpen(filename);
```

- status拿來儲存SysCall的執行狀況，例: 成功為1，失敗為-1。其餘Case也依此類推

```
kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCR
kernel->machine->WriteRegister(PCReg, kernel->machine-ReadRegister(PCReg) +
kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCR
```

- 最後執行完畢後，記得將Programming Counter的值 +4 唷~

ksyscall.h

- 接續前面的ExceptionHandeler
- 其實他在裡面只是呼叫了ksyscall.h的SysOpen()而已
- 透過這樣子的間接呼叫，來對Kernel進行操作

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

int SysWrite(char *buffer, int size, OpenFileId id)
{
  return kernel->fileSystem->WriteFile0(buffer, size, id);
}

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

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

- 而真正的實作細節其實又定義在filesys.h

filesys.h

- 這邊僅以WriteFile0為例子，其餘依此類推

```
int WriteFile0(char *buffer, int size, OpenFileId id){
  if(size <= 0){return -1;}
  WriteFile(id, buffer, size);
  return size;
}
```

- 因為跟sysdep.c裡的WriteFile有命名衝突，而我不太會用namespcae
- 故這邊取名為WriteFile0
- 其實從這邊可以看到，就只是呼叫sysdep.c裡面的"WriteFile" function而已

sysdep.c

- 從最上面的 #include <stdlib.h> 可看出
- 其實這邊也只是呼叫原生的C Library而已
- stub file system由此而來!

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

- 接下來到終端機的test目錄下指令:

```
4. os19team37
[os19team37@lsalab ~/NachOS-4.0_MP1/code/test]$ ../build.linux/nachos -e fileIO_test1
fileIO_test1
Success on creating file1.test
Machine halting!

This is halt
Ticks: total 925, idle 0, system 130, user 795
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
[os19team37@lsalab ~/NachOS-4.0_MP1/code/test]$ ../build.linux/nachos -e fileIO_test2
fileIO_test2
Passed! ^_^
Machine halting!

This is halt
Ticks: total 777, idle 0, system 110, user 667
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
[os19team37@lsalab ~/NachOS-4.0_MP1/code/test]$
```

- 完成!

What difficulties did you encounter when implementing this assignment?

徐迺茜

NachOS的程式碼非常多，除了要了解程式碼在寫甚麼以外，還要知道他在作業系統裡是負責甚麼功能。有的時候一個函數裡又包含很多其他函數，常常只是為了瞭解一個函數功能就開了很多檔案，花很多時間。另外電腦結構跟作業系統運作的不熟悉也影響我做這次作業的效率，加深作業難度。

陳子潔

從打開程式碼開始就遇到挑戰了，面對茫茫的代碼，只好一個一個點開來試圖理解
花了很多時間後覺得卡住了，又上網Google找了許多教學文件以及Reference。
勉強搞懂，到了要實作的時候，又遇到很多莫名奇妙的Bug，MobaXterm上的編輯器不太好用，常常發生忘記加中括號的錯誤，或者打錯字...等等
而且每次修改完都要重新cd, make clean, make, 呼叫nachos -e XXX
不過最後完成後覺得滿有成就感的啦!

Reference

[1]. [教學]C/C++ Callback Function 用法範例/內含Function Pointer 用法/ (http://dangerlover9403.pixnet.net/blog/post/83880061-%5B%E6%95%99%E5%AD%B8%5Dc-c++-callback-function-%E7%94%A8%E6%B3%95-%E7%AF%84%E4%BE%8B-(%E5%85%A7%E5%90%ABfunctio))
[2]. 何謂callback function (https://note1.pixnet.net/blog/post/10131726-%E4%BD%95%E8%AC%82callback-function)
[3]. 中斷和例外 (https://www.csie.ntu.edu.tw/~wcchen/asm98/asm/proj/b85506061/chap4/overview.html)
[4]. 10510周志遠教授作業系統_第5A講 Nachos解說/Ch2:OS Structure(Nachos解說) (https://www.youtube.com/watch?v=9vrPlp_f5A0&list=PLS0SUwlYe8czigQPzgJTH2rJtwm0LXvDX&index=15&t=0s)
[5]. What is the difference between interrupt and system call? (https://www.quora.com/What-is-the-difference-between-interrupt-and-system-call-1)
[6.] nachos系統調用實現Write、Read、Exec、Join (重要!!) (https://www.twblogs.net/a/5b88b67e2b71775d1cddf529)
[7.] [Nachos 4.0] Nachos System Call Implementation Sample (重要!!) (http://puremonkey2010.blogspot.com/2013/05/nachos-40-nachos-system-call.html)
[8.] cdave1/nachos (https://github.com/cdave1/nachos)
[9.] cygwin下交叉编译nachos-4.1 (https://bbs.csdn.net/topics/340086129)
[10.] 變數、函式可視範圍（static 與 extern） (https://openhome.cc/Gossip/CGossip/Scope.html)
[11.] Get Last Modified Date of File in Linux (https://superuser.com/questions/737247/get-last-modified-date-of-file-in-linux/737248)
[12.] Error: Jump to case label (坑啊!!!) (https://stackoverflow.com/questions/5685471/error-jump-to-case-label)
[13.] ERROR：expected initializer before “int” (奇怪的坑) (https://blog.csdn.net/qq_29985391/article/details/78606294)
[14.] Method stub (https://en.wikipedia.org/wiki/Method_stub)
[15.] C語言assert的用法 (https://www.itread01.com/content/1545290502.html)
[16.] Linux 系統程式設計 - fd 及 open()、close() 系統呼叫 (https://blog.jaycetyle.com/2018/12/linux-fd-open-close/)
[17.] Why do we use .globl main in MIPS assembly language? (https://stackoverflow.com/questions/33062405/why-do-we-use-globl-main-in-mips-assembly-language)
[18.] C語言的條件編譯#if， #elif， #else， #endif、#ifdef， #ifndef (https://codertw.com/%E7%A8%8B%E5%BC%8F%E8%AA%9E%E8%A8%80/81826/)