# OS MP1 Report

Team 27 : 112062232賴允中、111070070林櫂松

## 分工表

| 組員 | Trace Code | Report編寫 | Implementation |
|---|---|---|---|
| 賴允中 | **Y** | **Y** | **Y** |
| 林櫂松 | **Y** | **Y** | **Y** |

# 1. Trace Code

(a) SC_Halt (No arguments to pass)

- **machine/mipssim.cc => Machine::Run()**

此函數模擬CPU不斷讀取指令的動作。

```
kernel->interrupt->setStatus(UserMode);
```

設定讀新指令時的初始狀態為user mode。

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

無限迴圈，持續使用 OneInstruction() 來處理讀入的指令。

```
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();
```

處理完指令return後，呼叫 OneTick() 將CPU的時鐘週期更新一次，並檢查有沒有要打斷去執行其他程式的要求。

- **machine/mipssim.cc => Machine::OneInstruction()**

此函數可fetch一條user-level的指令，解碼並執行，如果遇到exception或interrupt，就呼叫 RaiseException() 來處理，然後return回Run() 。

```
// Fetch instruction
if (!ReadMem(registers[PCReg], 4, &raw))
    return;  // exception occurred
instr->value = raw;
instr->Decode();
```

有可能會呼叫 RaiseException() 的有以下幾種指令：OP_ADD, OP_ADDI, OP_LHU, OP_LW, OP_SUB, OP_SYSCALL, OP_UNIMP。

```
case OP_SYSCALL:
    DEBUG(dbgTraCode, "In Machine::OneInstruction, RaiseException(SyscallException, 0), " << kernel->stats->totalTicks);
    RaiseException(SyscallException, 0);
    return;
```

這裡只關注其中會引起interrupt的OP_SYSCALL。對 RaiseException() 傳入syscall的ExceptionType，處理完後return。

- **machine/machine.cc => Machine::RaiseException()**

當user program引發了syscall或發生exception時，呼叫此函數。

```
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);
}
```

把造成exception的地址存入reg[BadVAddrReg]，並把delayed operation都做完。接著切換到kernel mode，才能透過OS使用ISR（都是特權指令）。然後看呼叫此函數時傳入的ExceptionType是哪種，傳進 ExceptionHandler() 做對應的處理，處理完就切換回user mode繼續執行user program。

- **userprog/exception.cc => ExceptionHandler()**

此函數處理user program引發的syscall或exception。

```
enum ExceptionType { NoException,           // Everything ok!
                     SyscallException,      // A program executed a system call.
                     PageFaultException,    // No valid translation found
                     ReadOnlyException,     // Write attempted to page marked
                                            // "read-only"
                     BusErrorException,     // Translation resulted in an
                                            // invalid physical address
                     AddressErrorException, // Unaligned reference or one that
                                            // was beyond the end of the
                                            // address space
                     OverflowException,     // Integer overflow in add or sub.
                     IllegalInstrException, // Unimplemented or reserved instr.

                     NumExceptionTypes
};
```

在第一層switch case根據傳入的ExceptionType做選擇，ExceptionType被定義在machine.h裡。switch case中，目前除了syscall以外的都還沒被實作。

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

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

進入SyscallException的case後，第二層switch case會看存在reg2（r2被專門用來存放system call）的值決定是哪種system call，這裡以SC_Halt為主要觀察對象，halt的功能是中止OS的運行。

- **userprog/ksyscall.h => SysHalt()**

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

呼叫位於interrupt.cc的 Halt() 。ksyscall.h負責連結system call和kernel間的呼叫。

- **machine/interrupt.cc => Interrupt::Halt()**

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

kernel pointer連接了所有OS控制的部分，delete kernel也就相當於shutdown OS，並在shutdown前印出系統的資訊。

(b) SC_Create (Pass parameters by reference)

- **userprog/exception.cc => ExceptionHandler()**

```
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;
```

這次關注SC_Create的部分。ExceptionHandler() 前面的運作方式和SC_Halt一樣，先從reg2存的值決定是哪種type的system call，然後進對應的case。先去main memory取得文件名，它被放在memory的位置是存在reg4中（r4專門存放argument 1）。接著呼叫 SysCreate() ，把filename用**pass by reference**的方式傳入。

做完 SysCreate() 後，把回傳的值存在status，並把status寫進reg2 ( The result of the system call, if any, must be put back into r2 )。最後在return前，要把PC加上4，否則會一直在同個system call困住。

- **userprog/ksyscall.h => SysCreate()**

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

SysCreate() 函數是file system的interface，去呼叫在filesys.h裡的Create() 函數。

- **filesys/filesys.h => FileSystem::Create()**

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

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

此函數用於創建新檔案。由於目前使用的file system為stub，這裡呼叫的OpenForWrite() 其實實作上只是呼叫了c底層已寫好的 open() 。當創建

失敗時，代表file已經存在，open() 回傳-1，這裡就會return false，創建成功的話則return true。

(c) SC_PrintInt (Pass parameters by value)

- **userprog/exception.cc => ExceptionHandler()**

```
case SC_PrintInt:
    DEBUG(dbgSys, "Print Int\n");
    val = kernel->machine->ReadRegister(4);
    DEBUG(dbgTraCode, "In ExceptionHandler(), into SysPrintInt, " <<
    kernel->stats->totalTicks);
    SysPrintInt(val);
    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;
```

這次關注SC_PrintInt的部分。ExceptionHandler() 前段的運作方式和前述一樣。將在reg4（argument1）讀到的值存到val裡，並呼叫 SysPrint() 函數，將參數val傳入。一樣在return前要把PC加上4，否則會一直在同個system call困住。

- **userprog/ksyscall.h => SysPrintInt()**

```
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);
}
```

SysPrintInt() 函數去呼叫在 synchconsole.h 裡的 PutInt() 函數，並將val參數傳入。

- **userprog/synchconsole.cc =>SynchConsoleOutput::PutInt()**

```
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();
    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 form  waitFor->P(), " <<
        kernel->stats->totalTicks);
    } while (str[idx] != '\0');
    lock->Release();
}
```

此函數首先用 sprintf() 把value格式化變成字串後，存到str裡。接著，透過 lock->Acquire() 嘗試取得lock，才能開始執行do-while的部分，這樣做可以確保不會有不同的thread同時要做output的問題發生。接下來將str裡的字元依序傳入 PutChar() 等待列印，而為了避免一個字元還沒結束輸出，下一個字元就被傳入， waitFor->P() 會等待 PutChar() 結束後取得其釋放的lock。每個字元的輸出，會經過各一次的 P()、V()。最後輸出完畢，利用 lock->Release() 將lock釋放給其他threads使用。

- **userprog/synchconsole.cc => SynchConsoleOutput::PutChar()**

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

此函數和上面的 PutInt() 功能幾乎一樣，差異只在傳入的參數是一個字元，不用經過格式化，也不需要迴圈。

- **machine/console.cc => ConsoleOutput::PutChar()**

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

此函數在做把字元一個一個印在模擬的display上。

ASSERT(putBusy == False) 用意是確保沒有其他輸出在執行，然後即可透過 WriteFile() 把字元寫到display上。接著，將putBusy設成True，代表output正在執行中，不能有其他output operations一起執行。最後，透過 Schedule() 函數，把這個Console Write的interrupt安排在距離現在再過 ConsoleTime 的時間後被CPU收到。

- **machine/interrupt.cc => Interrupt::Schedule()**

```
void 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);
}
```

此函數在進行排程的動作，創建一個toOccur的物件來記錄interrupt的事項。”toCall”是interrupt發生時要執行的物件，”when”是未來的什麼時候要執行，”type”是產生interrupt的硬體，(fromNow>0)可以確保interrupt發生的時間在未來，並把這個toOccur插入pending list。

- **machine/mipssim.cc => Machine::Run()**

在SC_Halt中已做過分析。安排好interrupt後，會透過 OneTick() 去檢查pending list中是否有interrupt已經到了該執行的時間。

- **machine/interrupt.cc => Interrupt::OneTick()**

```
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();
        status = oldStatus;
    }
}
```

這個函數主要有兩個目的：第一個是讓系統前進一個時刻，第二個是檢查有沒有interrupt要被執行。在檢查有無interrupt要執行時，要先把interrupt disable掉，再呼叫 CheckIfDue() 來執行在等待中的interrupt，確保interrupt handler執行時不會被其他interrupt打斷。直到pending interrupt都被解決後，才把重新enable interrupt。

這個函數的最後還會檢查是否有context switch的需求，如果有，先把status設定成system mode(kernel mode)，才有權限做調度，調度完要把status恢復為原本的狀態。

- **machine/interrupt.cc => Interrupt::CheckIfDue()**

```
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.
        }
    }

    DEBUG(dbgInt, "Invoking interrupt handler for the ");
    DEBUG(dbgInt, intTypeNames[next->type] << " at time " << next->when);

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

    inHandler = TRUE;
    do {
        next = pending->RemoveFront();  // pull interrupt off list
        DEBUG(dbgTraCode, "In Interrupt::CheckIfDue, into callOnInterrupt->CallBack, " << stats->totalTicks);
        next->callOnInterrupt->CallBack();  // call the interrupt handler
        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;
}
```

這個函數用以檢查pending list中是否有已經到期該被執行的interrupt。首先，ASSERT(level == IntOff)用以確保interrupt已被disable。檢查list過程中，如果list是空的便直接return false；如果list非空則檢查最前面的interrupt事件。若其還沒到該觸發的時間，則依據傳入的參數advanceClock決定要快轉時間到下個interrupt的發生點，或是不動作直接return。（OneTick()中預設對此函數傳入FALSE，代表不快轉。）

若其已到期，則把inHandler設為True，依序處理所有到期的interrupt，並呼叫相對應的interrupt handler。由於前面執行 Interrupt::Schedule() 時有傳入ConsoleOutput的物件，所以會呼叫 ConsoleOutput::CallBack() 。直到全部該處理的interrupt被解決後，把inHandler改回False並回傳True。

- **machine/console.cc => ConsoleOutput::CallBack()**

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

當一個字元完成輸出，輪到下個字元輸出時，會呼叫此函數。

將putBusy設為False代表write operation執行完畢，可以繼續output了，並把輸出的字數+1記錄到kernel。

因為SynchConsoleOutput物件在建構時，會對ConsoleOutput的callWhenDone傳入自己的位置(this)。所以這裡的 CallBack() 函數，實際上就是SynchConsoleOutput::CallBack()。

- **userprog/synchconsole.cc => SynchConsoleOutput::CallBack()**

```
void SynchConsoleOutput::CallBack() {
    DEBUG(dbgTraCode, "In SynchConsoleOutput::CallBack(), " << kernel->stats->totalTicks);
    waitFor->V();
}
```

透過 V() 釋放lock，允許下一個字元被送到display output。

## (d) MakeFile

Makefile檔在大型的專案中很常見，因為其可以一次進行大量檔案的編譯，也可以針對不同的檔案下不同的編譯參數。

```
############################################################
# Makefile.dep contains all machine-dependent definitions
# If you are trying to build coff2noff somewhere outside
# of the MFCF environment, you will almost certainly want
# to visit and edit Makefile.dep before doing so
############################################################

include Makefile.dep
```

先include makefile需要用到的變數。Makefile.dep是針對不同的機器，包含作業系統和硬體設備的不同所定義的設定，有編譯器路徑、選項等等。

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

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

在makefile.dep中，GCCDIR定義了gcc toolchain的路徑。

以上makefile片段中，指定CC為GCCDIR下的gcc編譯器，指定AS為GCCDIR下的assembler（組譯器），指定LD為GCCDIR下的linker（負責連結.o檔和他們用到的library file）。

INCDIR是include directory，就是library的路徑。

CFLAGS是compile flags，即用gcc編譯時使用的參數，所以makefile的方便之處就在這，可以把編譯大量文件的重複工作和複雜的編譯參數整合成一套自動化流程。

```
ifeq ($(hosttype),unknown)
PROGRAMS = unknownhost
else
# change this if you create a new test program!
PROGRAMS = add halt createFile LotOfAdd
endif
```

如果hosttype（作業系統的類型）未知，make會執行unknownhost case，其實就是輸出錯誤訊息然後終止。若hosttype已知，則會將後面的case們（add, halt, createFile等等）指派給PROGRAMS。

我們以 make halt 為例子來觀察一隻nachos program是如何被編譯的。

```
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
```

首先halt.c被編譯成halt.o。第一行的halt.c代表它是halt.o的相依檔案，makefile會檢查halt.c是否有被修改過，決定需不需要重新編譯。接著，把start.o和halt.o link起來並輸出成halt.coff。注意 start.o：

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

使用start.S的意義，是因為不想將整個C library都link進來，只留下user program會用到的部分。它會引導程式從main開始，當需要syscall時把syscall type和需要的argument存進r2, r4, r5, r6, r7中並協助呼叫。

最後把 halt.coff 編譯成nachos執行檔 (NOFF格式) halt。

```
clean:
    $(RM) -f *.o *.ii
    $(RM) -f *.coff

distclean: clean
    $(RM) -f $(PROGRAMS)
```

這是刪除檔案的兩個指令。第一個是clean，它會刪除在compile過程中的中間檔，包含.o檔、.ii檔、.coff檔；第二個是distclean，它除了會先執行clean指令外，還會刪除生成的NOFF執行檔，像是add、halt等等，可以刪除得更乾淨。

## 2. Implementation

(a) userprog/syscall.h

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

原本SC_Open, SC_Read, SC_Write, SC_Close被註解掉了，為了讓start.S可以讀取到這些syscall對應的參數值，要去掉註解。

(b) test/start.S

```
    .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
```

比照其他種syscall，加上這四種。把system call type放入reg[2]，ExceptionHandler() 會讀取reg[2]的值，來判斷exception的type。

(c) userprog/exception.cc

- **SC_Open**

```
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;
```

按spec描述，Open() 只會傳遞一個參數name，它在memory中的位置會被儲存在r4裡。所以先去r4取其位置，再去main memory中讀取其地址，用傳址的方式傳進 SysOpen()。如果檔案開啟成功，SysOpen() 回傳該檔案的OpenFileId，否則回傳 -1。最後，syscall的回傳值要被存回r2中，然後更新PC。

- **SC_Write**

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

按spec描述，Write() 會傳遞三個參數，各自被放在r4, r5, r6中。其中buffer一樣透過傳址的方式傳入 SysWrite() ，numChar (size) 和 fileID則

是直接傳值。如果寫入成功，SysWrite() 回傳寫入的總字元數，否則回傳-1。回傳值一樣存回r2然後更新PC。

- **SC_Read**

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

按spec描述，Read() 會傳遞三個參數，各自被放在r4, r5, r6中。其中buffer一樣透過傳址的方式傳入 SysRead() ，numChar (size) 和 fileID則是直接傳值。如果讀取成功，SysRead() 回傳讀入的總字元數，否則回傳-1。回傳值一樣要存回r2然後更新PC。

- **SC_Close**

```
case SC_Close:
    fileID = kernel->machine->ReadRegister(4);
    {
        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;
```

按spec描述，Close() 只會傳遞一個參數 fileID，存放在r4中。直接將其傳值給 SysClose() ，如果檔案關閉成功就回傳1，否則回傳-1。回傳值一樣要存回r2然後更新PC。

(d) userprog/ksyscall.h

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

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

將 SysOpen() 的註解拿掉，並仿造其格式製造另外三種函數。前面trace code時提過ksyscall.h的功能就是負責連接system call到kernel底下的組件去做對應的操作，這裡是連結到fileSystem。

(e) filesys/filesys.h

- **FileSystem::OpenAFile()**

```
OpenFileId OpenAFile(char *name) {
    int idx = 0;
    while (OpenFileTable[idx] != NULL) {
        if(fname[idx] == name)  return -1;
        idx++;
    }
    if(idx >= 20) return -1;
    else {
        int fileDescriptor = OpenForReadWrite(name, FALSE);
        if (fileDescriptor == -1) return -1;

        OpenFileTable[idx] = new OpenFile(fileDescriptor);
        fname[idx] = name;
        return idx;
    }
}
```

```
OpenFile *OpenFileTable[20];
//array of opened file's filename, need this to handle duplicate opening case
char* fname[21];
```

SysOpen() 呼叫此函數後，這裡用迴圈在OpenFileTable中尋找是否還有空位可以開啟檔案。為了處理同一檔案被重複開啟的狀況，我們在FileSystem class底下新增一個 fname 字串陣列，用來依序記錄被開啟過的檔案的名字。如果idx超過了可開啟檔案的數量限制(20)，或是在fname中發現該檔案已被開啟過，則視為開啟失敗，直接回傳-1。

反之則代表table中有空位，嘗試呼叫 OpenForReadWrite() 透過傳入的name去開啟檔案。如果開啟失敗，它會回傳-1給fd，那 OpenAFile() 也會跟著回傳-1。開啟成功的話，就用fd建立一個OpenFile物件，放入table中，最後回傳此檔案在table中的index作為OpenFileId。

- **FileSystem::WriteFile()**

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

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

SysWrite() 呼叫此函數後，先別斷Write失敗的條件，當OpenFileId不在table index範圍裡，或是在OpenFileTable裡ID對應的entry沒有東西，都會回傳-1。反之，我們就使用OpenFile物件的 Write() ，此函式會負責將buffer中的內容寫入file，並回傳寫入的字數。

***OpenFile::Write()**

```
int Write(char *from, int numBytes) {
    int numWritten = WriteAt(from, numBytes, currentOffset);
    currentOffset += numWritten;
    return numWritten;
}
```

- **FileSystem::ReadFile()**

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

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

SysRead() 呼叫此函數後，先別斷Read失敗的條件，當OpenFileId不在table index範圍裡，或是在OpenFileTable裡ID對應的entry沒有東西，都會回傳-1。反之，我們就使用OpenFile物件的 Read() ，此函式會負責將file的內容讀出並寫進buffer，並回傳讀取的字數。

***OpenFile::Read()**

```
int Read(char *into, int numBytes) {
    int numRead = ReadAt(into, numBytes, currentOffset);
    currentOffset += numRead;
    return numRead;
}
```

- **FileSystem::CloseFile()**

```
int CloseFile(OpenFileId id) //file讀寫完了，把它從OpenFileTable移除掉
{
    if(id < 0 || id >= 20 || !OpenFileTable[id]) return -1;

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

SysClose() 呼叫此函數後，當OpenFileId不在table index範圍裡，或是在OpenFileTable裡ID對應的entry沒有東西，都會回傳-1。反之，就釋放該ID在OpenFileTable裡的記憶體，並且把OpenFileTable[id]指標設成NULL以避免存取到無效的指標，成功關閉檔案後回傳1。

- **Test Result**

1. fileIO_test1:

```
[os24team27@localhost test]$ ../build.linux/nachos -e fileIO_test1
fileIO_test1
Success on creating file1.test
```

2. fileIO_test2:

```
[os24team27@localhost test]$ ../build.linux/nachos -e fileIO_test2
fileIO_test2
Passed! ^_^
```

# 3. Difficulties we encounter when implementing this assignment

賴允中：

第一次接觸OS的核心程式，雖然只是模擬出來的OS，其環環相扣的各種資料結構和函數呼叫一樣十分複雜，光是跟著作業的spec一步步看都相當費神，這過程中我也修正了很多上課時沒完全理解或是搞錯的細節。在此讚嘆nachos非常完整的註解，和非常棒的coding style。我卡關最久的地方在於理解start.S的作用，在trace SC_Halt的時候，看halt.c明明呼叫了 Halt() ，也可以運作，它在syscall.h中卻根本沒有被實作啊？這困擾了我很久，感謝討論區熱心回覆的同學跟助教詳盡的解決了我的疑惑。然後實作部分雖然有遇到一點點小問題，不過很快就解決了，感謝組員的認真參與和配合。

後記：雖然很早就做完了，但因為助教說要處理同個檔案重複開啟的部分，所以禮拜六又花了一點點時間來做。完工後隔一兩小時組員開工作站檢查卻發現檔案全部變回原始狀態，就是剛從github clone下來的樣子！我趕緊把原本在local寫好的部分貼上去，卻發現怎麼跑都不對，著急的東改西改了一個多小時才發現，exception.cc裡的case都忘記要return了.....幸好有很快檢查到，否則最後交屍體的話真的會哭死。

林櫂松：

首先遇到最大的困難就是要去trace很多檔案，有時候雖然只是trace一個簡單的function，但它包含許多function，因此要開很多檔案，一直跳來跳去，有時候看到會眼睛花掉，遇到很多不懂的地方也上網找了許多資料來理解，也透過作業找出自己上課理解錯誤的地方。在基本的連server和linux的指令也遇到很多不懂，謝謝組員耐心教導和提供reference。最後就是實作時，在實作 OpenAFile() 時，沒注意到不能呼叫上面給user program使用的 Open() 函數來創建OpenFile物件，而直接在實作裡呼叫，test結果一直fail，幸好組員很快的幫我找到問題。如果今天是自己一組，應該會卡很久都debug不出來吧。之後還有更多挑戰，希望都可以迎刃而解。