MP1: SYSTEM CALL

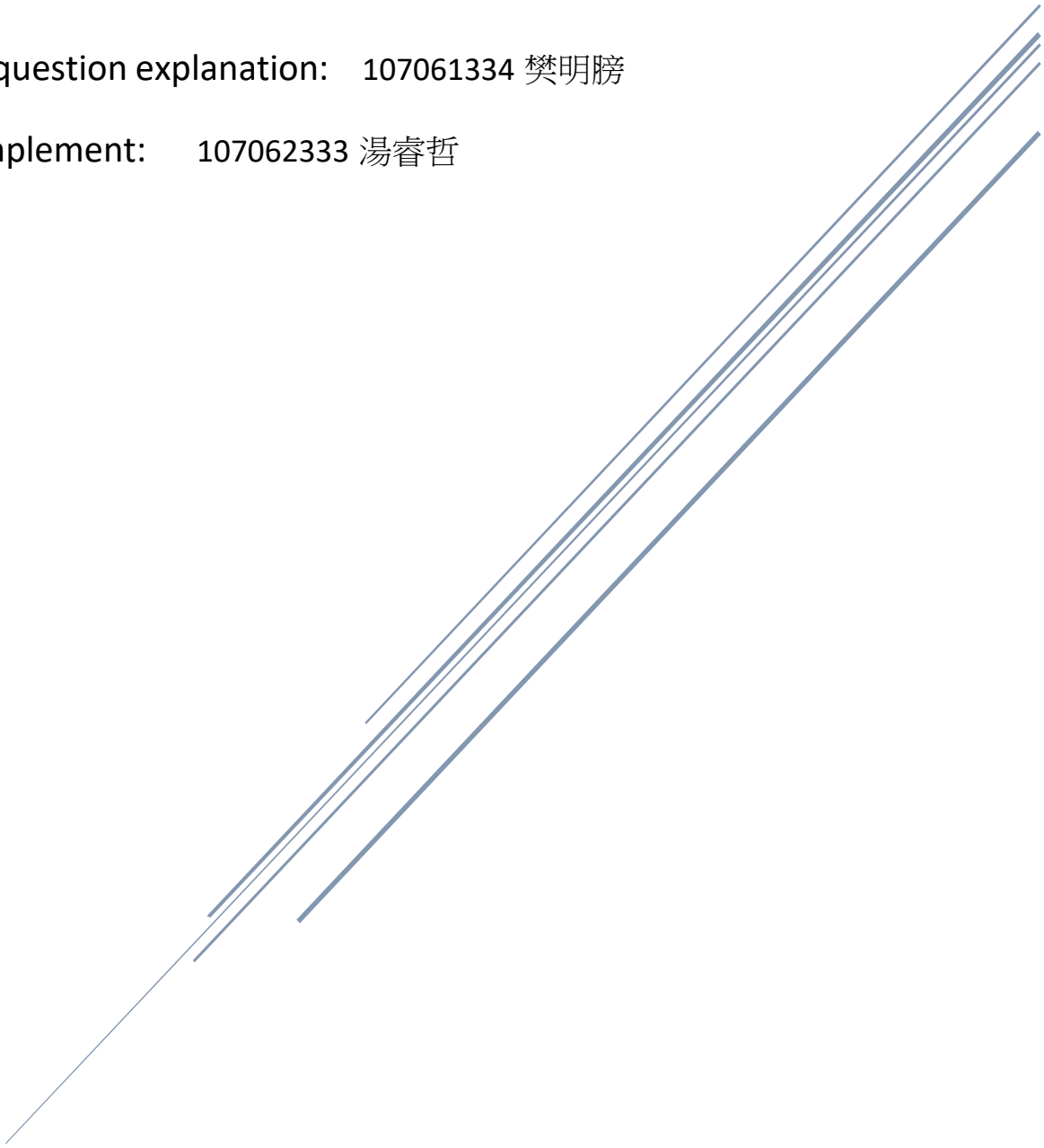
Team 21

Operating System

(a)Team List & Contibutions:

Report question explanation: 107061334 樊明膀

Code Implement: 107062333 湯睿哲



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

➢ SC_HALT

Machine/mipssim.cc

Machine::Run()

就是要 run 一個 user 的指令

```
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);
```

創造需要 decode 的 instruction 的 Storage，中間的 debug 不用理他，debug 主要是告訴你說，你現在執行順序中的 ins 是哪一個，然後時間是怎麼樣
Kernel ->interrupt->setStatus(Usermode)kernel 用 interrupt 把 mode 設成 usermode 避免之後在執行程式的時候，使用到了 kernel 才可以用的指令，保護電腦，使 user 不能隨意的更改一些東西。

```
for (;;) {
DEBUG(dbgTraCode, "In Machine::Run(), into O
    OneInstruction(instr);
DEBUG(dbgTraCode, "In Machine::Run(), return

DEBUG(dbgTraCode, "In Machine::Run(), into O
kernel->interrupt->OneTick();
DEBUG(dbgTraCode, "In Machine::Run(), return
if (singleStep && (runUntilTime <= kernel->s
    Debugger();
}
```

在 for 迴圈裡面 call instruction 用 OneInstruction()使他一直往下面執行，然後算 oneTick()這個跟 call back function 的時間有關，會影響 instruction 的排序問題，後面會再解釋

Machine::OneInstruction

就是在處理一個普通的 user 的指令
但如果中間有發生 interrupt 的情形的話，就會啟動到 interrupt handler，如果要回到這個 function 的話一樣是要經過 run()這個 function 避免出錯

還有就是如果有 interrupt 在之前還是會把變數那些的好好的存好在 mem or reg，而在下次回來的時候，就回 restart 上次的 instruction，並取得正確的數值。

```
#ifdef SIM_FIX
    int byte;       // described in Kane for LWL,LWR,...
#endif
```

這個會跟 mips 的 instruction 的讀法有關等等，用 SIM_FiX 這個 flag 是否 def 去決定讀法做修正，後面也常常會有類似的方法

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

從 mem 取得 instruction

Instr->Decode() 是 decode the binary representation of the instruction

```
unsigned int value; // binary representation of the instruction

char opCode;     // Type of instruction.  This is NOT the same as the
                 // opcode field from the instruction: see defs in mips.h
char rs, rt, rd; // Three registers from instruction.
int extra;       // Immediate or target or shamt field or offset.
                 // Immediates are sign-extended.
```

Instr 的結構是這個樣子　value 記下 raw 的值，也就是 memory 對應到的 pcreg 的值，然後解析之後確認是什麼指令(opcode) reg 的位置在哪(rs,rt,rd)或是 immd or 額外的 bit(extra)

```
switch (instr->opCode) {
```

根據 opcode 去做相對應的動作(case)，動作裡面也些也會因為 SIM_FIX 而讀寫的方式不太一樣

```
// Do any delayed load operation
DelayedLoad(nextLoadReg, nextLoadValue);

// Advance program counters.
registers[PrevPCReg] = registers[PCReg];    // for debugging, in case we
                     // are jumping into lala-land
registers[PCReg] = registers[NextPCReg];
registers[NextPCReg] = pcAfter;
```

Delayedload 跟 interrupt 或處理 exception 那些有關係

然後下面就是要將跑過的指令替換掉了 存下跑過的原因是為了 debug 然後把下一個要跑的，替換過來

Machine/machine.cc

Machine::RaiseException(ExceptionType which, int badVAddr)

User program 有 system call 或 exception 的時候會需要進入到 kernel 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);
    ExceptionHandler(which);        // interrupts are enabled at this point
    kernel->interrupt->setStatus(UserMode);
}
```

Which 是引起 exception 的原因
badVaddr 是引起 exception 的 virtual address
記好之後，先暫停一下 progress 然後進入到 system mode 也就是 kernel mode
這樣的話就可以啟動 ExceptionHandler 處理完之後回到 Usermode 回到 user program

Userprog/exeception.cc

ExceptionHandler(ExceptionType which)

```
//  For system calls, the following is the calling convention:
//
//  system call code -- r2
//      arg1 -- r4
//      arg2 -- r5
//      arg3 -- r6
//      arg4 -- r7
//
//  The result of the system call, if any, must be put back into r2.
```

上圖可以知道說，kernel mechine 的 reg 對應到的位置跟它的功用，r2 最一開始是 systemcall 的種類，如果有要返回的值的話記得存回 r2，其他是 argument，這邊也應該是對應到了上面 instruction 解讀時給她安排的 register 位置

```
ExceptionHandler(ExceptionType which)
{
    char ch;
    int val;
    int type = kernel->machine->ReadRegister(2);
```

像 reg 2 就是可以知道說 systemcall exception 的 type

```
switch (which) {
case SyscallException:
switch(type) {
```

判斷是 systemcall 之後，去判斷他的 type 作相對應的動作
在動作裡面會有

```
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);
```

這一連串的動作是因為目前的 pcreg 在 systemcall 的地方，因為 handler 處理好了這個指令，所以要往下一個 pcreg 前進。

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

裡面有兩個動作沒有做以上的動作，分別是 SC_MSG、SC_Halt，因為他們在裡面 call 了 SysHalt ()

Userprog/ksyscall.cc

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

Syshalt 就是要 call interrupt 的 Halt()

Machine/Interrupt.cc

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

就是直接把機器給停止了，直接刪掉就不會動了

➢ SC_Create

Userprog/exeception.cc

因為前面有介紹 exeception.cc 了所以這邊主要講 SC_Create

```
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;
```

Reg4 這個 argument 主要是記錄一個 memory 的位置裡面對應了一個 filename
接者 call SysCreate 去創建一個 file
這個函數會回傳創建是成功還是失敗，用 status 去紀錄
接著寫回到 reg2 告訴結果是成功還是失敗
最後調整 program counter 的位置，因為這個 instruction exception 處理好了

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

Userprog/ksyscall.cc

```
bool Create(char *name, int initialSize);
                // Create a file (UNIX creat)
```

filesys/filesys.h

➢ SC_PRintInt

Userprog/exeception.cc

因為前面有介紹 exeception.cc 了所以這邊主要講 SC_Printint

```
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;
```

reg4 這個 argument 主要是記錄要 print 出來的東西，把這個給 val，這邊主要是 int (val 的型態)
之後 call SysPrintInt()去讓他 print
後面一樣去修改 PCReg

```
void SysPrintInt(int val)
{
  DEBUG(dbgTraCode, "In ksyscall.h:SysPrintInt
  kernel->synchConsoleOut->PutInt(val);
  DEBUG(dbgTraCode, "In ksyscall.h:SysPrintInt
}
```

Userprog/ksyscall.cc 裡面的 SysPrintInt 去 call userprog/synchconsole.cc 裡面的 PutInt

call userprog/synchconsole.cc

```
SynchConsoleOutput::PutInt(int value)
{
    char str[15];
    int idx=0;
    //sprintf(str, "%d\n\0", value);  the
    sprintf(str, "%d\n\0", value); //simpl
    lock->Acquire();
    do{
    DEBUG(dbgTraCode, "In SynchConsoleOutp
        consoleOutput->PutChar(str[idx]);
    DEBUG(dbgTraCode, "In SynchConsoleOutp
    idx++;

    DEBUG(dbgTraCode, "In SynchConsoleOutp
        waitFor->P();
    DEBUG(dbgTraCode, "In SynchConsoleOutp
    } while (str[idx] != '\0');
    lock->Release();
}
```

str 先設置好超過這個 int 極限(2147483647)的長度>10

sprintf 這個只是 print 出來比較好 debug 用的

```
void Lock::Acquire()
{
    semaphore->P();
    lockHolder = kernel->currentThread;
}
```

threads/synch.cc

lock 不用太去在意，主要是不想要 thread 跑下去，直到我下面的 release

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

上面用 do while 迴圈直到字都 Putchar 出去從 idx0 到空白字眼'\0'

waitFor->P()就是在等 putchar 的時間

Putchar 一樣先 lock 然後裡面 call 了 console.cc 的 Putchar()

Machine/console.cc

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

Assert 去判斷 put 的狀態是不是 busy 的如果是的話就不能進行，

```
bool putBusy;          // Is a PutChar operation in progress?
                       // If so, you can't do another one!
```

WriteFile unix file 去模擬 display 的 function

```
"writeFile" -- UNIX file simulating the display (NULL -> use stdout)
```

後面用 interrupt->schedule(call back function) 去做排程

```
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);
}
```

首先先計算出要發生的時間 (when=now+from now)
然後去設計好 interrupt 的時間把跟要發生的事還有事哪個 hardware type 發出 interrupt 先包裝好，之後方便 pending
Assert 那邊確認好，是之後要發生的事
最後就是做 pending 將他放在排好的 list 裡面

```
DEBUG(dbgTraCode, "In Machine::Run(), I
kernel->interrupt->OneTick();
```

回到 mipssim 這邊 ，裡面的 run()裡的 for 迴圈，call 了一個 oneTick

Machine/interrupt.cc

Interrupt::OneTick()

兩種情況會啟動這個 onetick，一個是 user instruction 執行的時候，另一個是 interrupt 又遇到 interrupt

這個功能主要是確認是不是有 pending interrupt 需要執行

```
advance simulated time
 if (status == SystemMode) {
     stats->totalTicks += SystemTick;
 stats->systemTicks += SystemTick;
 } else {
 stats->totalTicks += UserTick;
 stats->userTicks += UserTick;
 }
```

確認好時間，用 mode 判別

```
 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;
 }
```

Changelevel 是因為避免有在 handler 遇到 interrupt 又插入的情況

yieldOnReturn 是應對可能 handler 的一些要求改變了一些 context

中間的 checkifDue 下面解說

Interrupt::CheckIfDue(bool advanceClock)

這個功能主要是拿來判斷說有沒有排程中的 interrupt 需要發生的，有的話就讓它發生

```
"advanceClock" -- if TRUE, there is nothing in the ready queue,
    so we should simply advance the clock to when the next
    pending interrupt would occur (if any).
```

advanceClock ，如果是的話就是現在沒有東西 ready，而我們要提前 clock 讓下一個 interrupt 發生，但 OneTick 裡面的 advanceClock==false

```
  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.
  }
  }
```

CheckIfDue 裡面去判斷說時間的條件有沒有滿足，以及有沒有前面說的 advanceClock 的要求，如果時間沒到的話 就會 return False

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

看一下 machine 有沒有在忙

```
inHandler = TRUE;
do {
    next = pending->RemoveFront();     // pull interrupt off list
    DEBUG(dbgTraCode, "In Interrupt::CheckIfDue, into callOnInterrupt->CallBack,
    next->callOnInterrupt->CallBack();// call the interrupt handler
    DEBUG(dbgTraCode, "In Interrupt::CheckIfDue, return from callOnInterrupt->Cal
delete next;
} while (!pending->IsEmpty()
        && (pending->Front()->when <= stats->totalTicks));
inHandler = FALSE;
return TRUE;
```

說明正在處理 ishandler=true 等做完在設成 false，然後中間 do while 處理第一筆後，如果後面的時間一樣>=when(interrupt 要發生的時間)的話就繼續做，while 裡面的動作就是一一的清空 pending 裡面的名單，然後 call 對應的 interrupt handler。

這邊因為是要 print out 所以會是對應到 output

Machine/console.cc

ConsoleOutput::CallBack() 在可以display char 的時候 call 這個 function

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

這邊將前面 Putchar 裡拉起來的 putbusy 使他變成了 false

kernel->stats->numConsoleCharsWritten++; 寫過的字加一不是用重要，統計用的

```
CallBackObj *callWhenDone;    // Interrupt handler to call when
    // the next char can be put
```

callWhenDone ->Callback() 因 為 SynchConsoleInput : public CallBackObj
所以連到下面的 callback() function

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

當 handler 覺得下一個字可以被 display 的時候，就會 call 這個 function，然後 waitFor callback 這樣

(c) Explain your implementation as requested in Part II-2.

已知整個 monitor 的流程為:
從 usermode 的 C++ code 編譯產生組語(in start.s)並藉由 system_call (in system_call.h)媒介進入 kernel，並藉由 kernel mode exception (in exception.cc)接收 usermode 傳入的 interrupt(i.e. system_call) 並使的 machine 做出對應的動作。

1. Start.s:

而在 Start.s 裡所做的事情類似於系統 API，目的只是要模擬組語幫助 usermode 的 system_call 進入 kernel 並藉由每一次對應的 stub id 讓 kernel system 知道 user 欲直行的指令，因此每種指令於 Start.s 的 implement 都大同小異。
以 Open 為例:

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

.global (Instruction)作為設置一全域標籤作為協助 system_call 的參照，亦即宣告一個外部函數
而.ent (Instrucution)是說該 Instruction 的 entry 位置，亦即該函數的開始處
有了前置宣告後，接下來要將 system_call 想要傳的 instruction stub 放進一

register (在這我們使用$2)並帶入 kernel。

2. 進入 kernel:

成功進入 kernel 後，接下來由 exception.cc 來處理收到 system_call 所帶來的 stub id 後要做出的行為。這次作業有 4 個行為需要 implements:

a. SC_Open:

```
/*========================= 10/15 edit ============================*/
        case SC_Open:
            val = kernel->machine->ReadRegister(4);
            {
            char *filename = &(kernel->machine->mainMemory[val]);
            fileID = SysOpen(filename);
            //cout<<"open "<<fileID-5<<" file"<<"\n";////拿來debug用 for exceed 20 files
            kernel->machine->WriteRegister(2,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;
```

參考定義好的 SC_Create，可以推知傳入的參數應該是放在$4 的位置，也就是檔案存放於 main memory 索引處，將從 kerneal 呼叫 SysOpen 來開啟檔案，並將開檔後回傳的 id 寫入 register。而 SysOpen 會藉 kernel 進入 filesystem 呼叫 OpenAFile()來讀取檔案訊息，並回傳一 OpenFileId 作為該檔案開啟後的識別碼。於 filesys.h 的 implement 如下:

```
OpenFileId OpenAFile(char *name) {
    int fileDescriptor = OpenForReadWrite(name,FALSE);///id start from 6
    if(fileDescriptor==-1 || fileDescriptor>25)return -1; ///invalid or
    OpenFileTable[fileDescriptor-6] = new OpenFile(fileDescriptor);
    return fileDescriptor;
}
```

依據取得的 OpenFileId(從 6 開始)，-1 代表開檔失敗，而超過 25 則代表開啟第 21 個檔案了(作業規定最多只能開 20 個檔案)。並利用提供的 Table 來記錄對應的 index 的資料。最後回傳該檔案的 id 回去給 kernel exception 端。
最後將檔案是別 id 寫入 register。完成該 Open() 該指令的流程。

b. SC_Close:

```
case SC_Close:
    val = kernel->machine->ReadRegister(4);
    {
    status = kernel->fileSystem->CloseFile(val);
    kernel->machine->WriteRegister(2,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;
```

先從$4 取得欲關閉的檔案他所對應的 id，並使 kernel 進入 filesystem 呼叫

```
int CloseFile(OpenFileId id){////
    if(OpenFileTable[id-6] == NULL){
        return -1;
    }
    else{
        OpenFileTable[id-6] = NULL;
        return 1;
    }
}
```

CloseFile()，於 filesys.h 的 implement 如下:

將 OpenFiletable[index]拉成 NULL 並回傳 1 表示關檔成功。

如果該資料未被open則return-1

C. SC_Write:

```
case SC_Write:
    val = kernel->machine->ReadRegister(4);
    fileID = kernel->machine->ReadRegister(6);
    {
    char* buffer = &kernel->machine->mainMemory[val];
    int size = kernel->machine->ReadRegister(5);
    numChar = kernel->fileSystem->WriteFile(buffer,size,fileID);
    kernel->machine->WriteRegister(2,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;
```

先從$4 取得欲寫入的內容放在 maim memory 的位置 index，從$6 取得欲寫入檔案的 id，從$5 取的內容的大小 size。得到所有資訊後，使 kernel 進入 filesystem 呼叫 WriteFile()，於 filesys.h implement 如下:

```
int WriteFile(char *buffer, int size, OpenFileId id){
    OpenFile* file = new OpenFile(id);
    int len = file->Length();
    int num = file->WriteAt(buffer,size,len);
    if(num)return 1;
    else return -1;
}
```

先建立一個檔案型別的 variable，並取得其 len，隨後呼叫 WriteAt()去將傳進來的欲寫入內容(buffer)寫入該檔案中，並從回傳的質來得知是否寫入成功，並再回傳至 kernel 的 exception 端。

d. SC_read:

```
case SC_Read:
    val = kernel->machine->ReadRegister(4);
    fileID = kernel->machine->ReadRegister(6);
    {
    char* buffer = &kernel->machine->mainMemory[val];
    int size = kernel->machine->ReadRegister(5);
    numChar = kernel->fileSystem->ReadFile(buffer,size,fileID);
    kernel->machine->WriteRegister(2,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;
```

先從$4 得知欲讀取資訊他於 main memory 的 index 位置，從$6 取得欲讀取檔案的 id，從$5 取得內容大小 size，隨後帶著這些資訊使 kernel 進入 filesystem 呼叫 ReadFile()，於 filesys.h implement 如下:

```
int ReadFile(char *buffer, int size, OpenFileId id){
    //OpenFile* file = new OpenFile(id); //spec 規定要用 OpenFileTable
    int num = OpenFileTable[id-6]->Read(buffer,size);
    if(num)return num;
    else return -1;
}
```

將傳入的欲讀取內容(buffer)、內容大小 size 以及檔案是別 id，利用對應 table 中的檔案呼叫 Read()取得內容，並回傳其值，-1代表錯誤。並將核對結果傳回至 kernel exception 端。

必要步驟 PC+4:

```
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;
```

每次做完一個 instruction 後(即執行完一次 system_call)，都要記得將 program counter + 4 準備 fetch 下一條 instruction。
如此才算完成一次 kernel mode 的 exception handle case，才能交由 machine Run。

(d) What difficulties did you encounter when implementing this assignment?

1. 一開始拿到這份作業時，完全不知道該從何下手，花了很多時間觀察已經 define 好的內容以及嘗試去了解整個模擬 OS 架構，加上 spec 上有的提示，才慢慢地知道該去哪 implements 以及通過 test smaple。
2. 這次我覺得作業難的不適 code implements，而是 report 第一大題問題回答，需要花比想像中多的時間去 trace 去了解，整個架構對我們剛學習 OS 的人來說是頗為複雜的，需要花時間欲吸收。
3.在做 debug 想要了解處理 exception 的時候，打入 -e halt -d u ，u 對應到了 system call，其中的 SC_halt 本來就會有停止的動作，可是他會出現一句 unable to open file halt，對於這個的意思不是很懂 不清楚他的 halt 是什麼東西。

```
[os20team21@lsalab ~/NachOS-4.0_MP1/code]$ build.linux/nachos -e halt -d u
halt
Unable to open file halt
```