MP1_report

109070025 林泓錩

1 Trace Code

1.1 SC_Halt:

a. **Machine::Run():**

- **Purpose:**

當程式開始後，會由 Kernel 呼叫在 NachOS 上模擬 user-level program 的執行。

- **Detail:**

```
void Machine::Run() {
  Instruction *instr = new Instruction;
```

開始後會先給一個儲存空間 (storage) 給 instruction 儲存。

```
if (debug->IsEnabled('m')) {
```

在這之後 if 條件句裡會檢查在 debug.h 中 Debug::IsEnabled 定義的'm' 是否已經被 enable。

```
kernel->interrupt->setStatus(UserMode);
enum MachineStatus {IdleMode, SystemMode, UserMode};
```

接著會把 interrupt.h 中 MachineStatus 定義的 status (分成 idleMode, SystemMode, UserMode 三種) 設成 UserMode。 之後會跑

```
for (;;) {
  DEBUG(dbgTraCode, "In Machine::Run(), into OneInstruction "
OneInstruction(instr); kernel->interrupt->OneTick();
```

for(; ;) 的無窮迴圈，在回圈內有幾個 Debugger 測試失敗的話會回傳訊息，此外回圈內還有跑 OneInstruction() 和 OneTick() 兩個函式。

b. **Machine::OneInstruction():**

- **Purpose:**

執行從 user-level program 獲得的單一指令。

- **Detail:**

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

```
void Instruction::Decode() {
  OpInfo *opPtr;
```

先用 if 條件句判斷 ReadMem() 函數是否讀取到東西，沒有則是 expection occurred 會跳出函式，若有則 ReadMem() 會讀取 register[PCReg] 中的 Instruction 存到 raw 中，然後呼叫 Decode() 函式獲得 opCode, rs, rt 及 rd，用來判斷要執行哪一類型的 instruction。

```
if (debug->IsEnabled('m')) {
int pcAfter = registers[NextPCReg] + 4;
```

接著一樣會經過一個 Debug::IsEnabled 進行確認，之後便會將下個要執行的 instruction 地址存放在 pcAfter 裡，但因會有 error 或 branch 的可能性所以不會直接載入。

```
switch (instr->opCode) {
registers[PrevPCReg] = registers[PCReg];

registers[PCReg] = registers[NextPCReg];
registers[NextPCReg] = pcAfter;
```

再來會進入 switch 裡來執行 instruction，如果是一般的指令 (ex: OP_ADD, OP_ADDI, OP_ADDU... 等) 就存取，然後跳出 switch，跳出 switch 後會將之前算好的 pcAfter 寫入 registers[NextPCReg] 中。

```
case OP_SYSCALL:
  DEBUG(dbgTraCode, "In Machine::OneInstruction,
  RaiseException(SyscallException, 0);
  return;
```

而在 switch 中若偵測到 OP_SYSCALL(ex: SC_Halt) 後會呼叫 RaiseExpection(SyscallExpection, 0)。

c. **Machine::RaiseExpection():**

- **Purpose:**

將 Status 從 UserModde 轉變為 KernelMode，因為使用者的程式觸發 system call 或有 excpetion 發生 (ex: 地址轉移失敗)。

- **Detail:**

```
Machine::RaiseException(ExceptionType which, int badVAddr)
```

呼叫函式後必須傳入兩個參數，第一個是 exceptionNames[which]，which 表示是哪種型態的 kernel trap(ex:"no exception", "syscall",etc.)，第二個則是 registers[BadVAddrReg]，會把 badVAddr 放進 register[39](BadVAddrReg 被定義為 39)，而 badVAddr 則是該 trap 的 virtual address。

```
DelayedLoad(0, 0);
```

兩個參數傳入後會呼叫 DelayedLoad() (在 mipssim.cc 中) 模擬 delay load 產生的影響。

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

並將原來的 usermode 轉到 systemmode 之後呼叫 ExceptionHandler(which) 去處理 exception，等結束後再將 mode 切回 usermode 表示 exception 處理完成。

```
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
};
```

因為在 SC_HALT 中，偵測到了 system_call 所以呼叫了 ExceptionHandler() 並傳入 SyscallException(ExceptionType 的一種，定義在 machine.h)

d. **ExpectionHandler():**

- **Purpose:**

進入 Nachos kernel 的入口，會被使用者執行程式呼叫，或在有 system call, addressing exception 或 arithmetic exception 時被呼叫。

- **Detail:**

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

如果是 system call 的話，會先將 r2 中的 system call code 傳入 type 參數中。

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

接著進入到 switch 迴圈根據 which 與 type 的類型執行不同的函式，在 SC_Halt 中，switch 裡將會執行 SysHalt() 函式。

e. **SysHalt():**

- **Purpose:**

ksyscall.h 是一個介面 (kernel interface) 給 systemcalls，SysHalt() 為其中之一。

- **Detail:**

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

呼叫 kernel->interrupt 去執行裡面的 Halt()。

f. **Interrupt::Halt():**

- **Purpose:**

乾淨地停止 (Shut Down) Nachos, 並輸出 performance 的統計資料.

- **Detail:**

```
kernel->stats->Print();
```

呼叫 kernel->stats 執行裡面的 Print()，此函示會印出 totalTicks, idleTicks, systemTicks, userTicks 等的時間資訊, 並印出 Disk I/O

reads,writes；Console I/O reads, writes；Paging faults；Network I/O packets recieved, sent 等資訊。

```
delete kernel;  // Never returns.
```

之後 delete 掉 kernel，且無法挽回 (never returns)。

1.2 SC_Create:

a. **ExceptionHandler():**

- **Purpose:**

進入 Nachos kernel 的入口，會被使用者執行程式呼叫，或在有 system call, addressing exception 或 arithmetic exception 時被呼叫。

- **Detail:**

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

與前一部分 SC_Halt 在這裡的過程很相似，但在 type 中偵測到 system call code 是 SC_Create 後，會從 r4 得到 arg1 並存入 val 中。

```
char *filename = &(kernel->machine->mainMemory[val]);
// cout << filename << endl;
status = SysCreate(filename);
kernel->machine->WriteRegister(2, (int)status);
```

接著使用 val 進入 Machine::mainMemory 裡面獲取 filename，然後呼叫 SysCreate(filename)，並將回傳值記錄在 r2 中。

```
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);
```

接著將 pc+4(以避免重複執行同一個 system call) 後 return。

b. **SysCreate():**

- **Purpose:**

與前面 SysHalt() 一樣，SysCreate() 存於 ksyscall.h 是一個介面 (kernel interface) 給 systemcalls。

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

- **Detail:**

呼叫 kernel->fileSystem 去執行其中的 Create(filename)，會透過 return value 判斷是否成功創建檔案，且此 value 會回傳到 ExceptionHandler 的 status 裡。

c. **FileSystem::Create():**

- **Purpose:**

在 NachOS 檔案系統中建立檔案。

- **Detail:**

```
directory = new Directory(NumDirEntries);
directory->FetchFrom(directoryFile);
```

```
void
Directory::FetchFrom(OpenFile *file)
{
    (void) file->ReadAt((char *)table, tableSize * sizeof(DirectoryEntry), 0);
}
```

要創建檔案會先建立一個 directory，接著用 FetchFrom 函式從 disk 中讀取 directory 的內容。

```
if (directory->Find(name) != -1)
  success = FALSE;          // file is already in directory
```

再來用 directory::find() 去判斷該檔案是否已經存在 directory 了，若已存在則 (name!=-1) 會回傳 false。,

```
else {
    freeMap = new PersistentBitmap(freeMapFile,NumSectors);
    sector = freeMap->FindAndSet(); // find a sector to hold the file header
    if (sector == -1)
        success = FALSE;        // no free block for file header
    else if (!directory->Add(name, sector))
        success = FALSE;    // no space in directory
```

若該檔案不存在，則判斷第二個條件：有沒有 file header 的空間，而我們需要用 PersistentBitmap to initialize a persistent bitmap，並使用 freeMap->FindAndSet() 去尋找有沒有 sector 可以寫入 file header，若回傳-1 代表沒有 free block 可以寫入 file header 了，此外還會使用!directory->Add(name, sector) 檢查 directory 還有沒有更多空間可以放入 filenames 。

```
else {
        hdr = new FileHeader;
    if (!hdr->Allocate(freeMap, initialSize))
            success = FALSE;    // no space on disk for data
    else {
        success = TRUE;
    // everthing worked, flush all changes back to disk
            hdr->WriteBack(sector);
            directory->WriteBack(directoryFile);
            freeMap->WriteBack(freeMapFile);
    }
```

若以上情況都沒有發生問題，就可以創建 file header 並利用 Allocate(freeMap, initialSize) 檢查 disk 有沒有空間可以容納 file，沒有就會回傳 False，有空間的話就會紀錄 header, directory, freeMap 到 disk 中並 delete 掉剛剛創建 header, freeMap, directory 最後並回傳 Success=true 代表創建成功。

※ 檔案已經存在；沒有 file header 的空間；沒有 directory 的空間；disk 沒有空間都會造成檔案無法創建。

1.3 SC_PrintInt:

a. **ExceptionHandler():**

- **Purpose:**

進入 Nachos kernel 的入口，會被使用者執行程式呼叫，或在有 system call, addressing exception 或 arithmetic exception 時被呼叫。

- **Detail:**

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

與前一部分 SC_Create 在這裡的過程很像，在 type 中偵測到 system call code 是 SC_PrintInt 後，會從 r4 得到 arg1 並存入 val 中，接著將 val 傳入 SysPrintInt()。

```
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;
```

最後將 pc+4(以避免重複執行同一個 system call) 後 return。

b. **SysPrintInt():**

- **Purpose:**

與前面 SysCreate() 一樣，SysPrintInt() 存於 ksyscall.h 是一個介面 (kernel interface) 給 systemcalls。

- **Detail:**

```
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);
}
```

呼叫 kernel->sysnchConsoleOut 去執行其中的 PutInt()，會將 ExceptionHandler 傳進來的 value 傳入 PutInt()。

c. **SynchConsoleOutput::PutInt():**

- **Purpose:**

將 ExceptionHandler 傳進來的 int value 透過 sprintf() 函式存入 str 中 (char 型態)，尾端會加上
0 作為字串結束判斷點。。

- **Detail:**

```
sprintf(str, "%d\n\0", value); //simply for trace code
lock->Acquire();
```

將 int value 存入 str 後，會先透過 lock->Acquire() 去確保 lock 是 free 的，也將 lock 設為 busy。- **Detail:**

```
consoleOutput->PutChar(str[idx]);
} while (str[idx] != '\0');
lock->Release();
```

接著在 do-while 迴圈內，會將 str 內的每個字元傳入 ConsoleOutput::PutChar() to simulated display，直到遇到//0 字串結束，完成後會再 lock->Release()，將 lock 恢復為 free。

d. **SynchConsoleOutput::PutChar():**

- **Purpose:**

將字元寫入去 console display，如果必要的話要等待 lock 變 free。

- **Detail:** - **Detail:**

```
lock->Acquire();
consoleOutput->PutChar(ch);
waitFor->P();
lock->Release();
```

一樣先透過 lock->Acquire() 去確保 lock 是 free 的，也將 lock 設為 busy，做完 Putchar() 都要執行 wait->P() 去等待 callBack()，完成後會再 lock->Release()，將 lock 恢復為 free。

e. **ConsoleOutput::PutChar():**

- **Purpose:**

將字元寫入去 console display，schedule 一個 interrupt 在未來發生，然後 return。

- **Detail:**

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

先查看 putBusy 狀態是否為 False，去確認是否有 PutChar operation 在進行中，如果是的話會發出警告，接著會將字元寫入 writeFileNo 的檔案，然後再將 putBusy 狀態設為 True，代表有 PutChar operation 在進行中，最後呼叫 kernel->interrupt->Schedule()，將目前的時間、執行所需時間、指令種類傳入。

f. **Interrupt::Schedule():**

- **Purpose:** 安排 CPU 需在時間到達"now+when" 時去被 interrupted。

- **Detail:**

```
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);
}
```

*toCall 是當 interrupt 發生時需呼叫的物件的指標，when 會計算此 interrupt 發生的絕對時間 = kernel->stats->totalTicks + fromNow，意即目前的 ticks + 未來指令所需時間，之後將此指令需呼叫的函示、執行的絕對時間、執行指令種類，包成一個 PendingInterrupt 的 class，放進 *toOccur。

g. **Machine::Run():**

- **Purpose:**

當程式開始後，會由 Kernel 呼叫在 NachOS 上模擬 user-level program 的執行。

- **Detail:**

```
void Machine::Run() {
  Instruction *instr = new Instruction;
```

開始後會先給一個儲存空間 (storage) 給 instruction 儲存。

```
if (debug->IsEnabled('m')) {
```

在這之後 if 條件句裡會檢查在 debug.h 中 Debug::IsEnabled 定義的'm' 是否已經被 enable。

```
kernel->interrupt->setStatus(UserMode);
enum MachineStatus {IdleMode, SystemMode, UserMode};
```

接著會把 interrupt.h 中 MachineStatus 定義的 status (分成 idleMode, SystemMode, UserMode 三種) 設成 UserMode。 之後會跑

```
for (;;) {
  DEBUG(dbgTraCode, "In Machine::Run(), into OneInstruction "
OneInstruction(instr); kernel->interrupt->OneTick();
```

for(; ;) 的無窮迴圈，在回圈內有幾個 Debugger 測試失敗的話

會回傳訊息，此外回圈內還有跑 OneInstruction() 和 OneTick() 兩個函式。

h. **Interrupt::OneTick():**

- **Purpose:**

預先模擬時間且檢查是否有待辦的 interrupt 需要被呼叫。有兩種情形 OneTick() 會被呼叫: interrupts are re-enabled；a user instruction is executed。

- **Detail:**

```
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;
}
```

先將原先的 status 存在 oldStatus 中，依據當下的 status (UserMode 或 SystemMode)，增加一單位時間的 Userticks 或 Systemticks 記錄在 Statistics 的 class 裡。

```
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;
```

接著判斷是否有需要處理的 interrupt 透過 ChangeLevel() 先 turn off interrupt，確保待會若有需處理 interrupt，不會有其他的 interrupt 同時執行，再呼叫 CheckIfDue(FALSE)，確認有無等待處理的 interrupts，執行完後再透過 ChangeLevel() turn on interrupt，允許其他 interrupt 的執行，再來若 timer device handler 要求要執行 context switch，則將 status 切換到 SystemMode 讓 kernel 呼叫 Thread Yield()，檢查是否有其它 Thread 正在執行，完成後再將 status 設回之前存下的 oldStatus。

i. **Interrupt::CheckIfDue():**

- **Purpose:**

檢查是否有任何 interrupts 被安排去執行，如果有的話，fire them off 然後回傳 TRUE。

- **Detail:**

```
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.
  }
}
```

確認是否 level==IntOff，然後檢查 pending 裡是否有東西，有的話查看 pending 裡的第一個 interrupt，若它的"when" 大於目前的 (totalTicks) 代表 pending queue 裡面所有的 interrupt 目前都不會執行，因為時間還沒到。

```
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));
```

若 interrupt 的"when" 到了，會進入 do-while 迴圈對 pending 中

的 interrupts 呼叫 callback()，直到目前的時間內尚不會發生任何 interrupt (pending 為 empty 或 interrupt 時間大於目前時間)。

j. **ConsoleOutput::CallBack():**

- **Purpose:**

模擬器會呼叫它，當下一個字元可以輸出到 display 時。

- **Detail:**

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

將 putBusy=FALSE，呼叫 kernel->stats 完成一次 numConsoleCharsWritten++，再做 callwhenDone->CallBack()。

k. **SynchConsoleOutput::CallBack():**

- **Purpose:**

當可以安全發送下一個字元時，interrupt handler 的命令可以發送到 display。

- **Detail:** - **Detail:**

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

呼叫 waitFor->V()，增加 semaphore value 的值，必要時呼叫 waiter。

2 Implement

2.1 test/start.S

根據以下提示，仿照 PrintInt、Create... 等指令將 Open；Close；Read；Write 四個指令寫入 r2 裡面。

```
System call stubs:
 Assembly language assist to make system calls to the Nachos kernel.
 There is one stub per system call, that places the code for the
 system call into register r2, and leaves the arguments to the
 system call alone (in other words, arg1 is in r4, arg2 is
 in r5, arg3 is in r6, arg4 is in r7)

 The return value is in r2. This follows the standard C calling
 convention on the MIPS.
```

```
/* TODO (Open): Add SC_Open system call stubs, you can imitate existing system calls. */
/* TODO (Write): Add SC_Write system call stubs, you can imitate existing system calls. */
/* TODO (Read): Add SC_Read system call stubs, you can imitate existing system calls. */
/* TODO (Close): Add SC_Close system call stubs, you can imitate existing system calls. */
```

結果如下圖:

```
/* TODO (Open): Add SC_Open system call stubs, you can imitate existing system calls. */
    .globl Open
    .ent    Open
Create:
    addiu $2,$0,SC_Open
    syscall
    j   $31
    .end Open
/* TODO (Write): Add SC_Write system call stubs, you can imitate existing system calls. */
    .globl Write
    .ent    Write
Create:
    addiu $2,$0,SC_Write
    syscall
    j   $31
    .end Write
/* TODO (Read): Add SC_Read system call stubs, you can imitate existing system calls. */
    .globl Read
    .ent    Read
Create:
    addiu $2,$0,SC_Read
    syscall
    j   $31
    .end Read
/* TODO (Close): Add SC_Close system call stubs, you can imitate existing system calls. */
    .globl Close
    .ent    Close
Create:
    addiu $2,$0,SC_Close
    syscall
    j   $31
    .end Close
```

2.2 userprog/syscall.h

接著將 syscall.h 的註解移除，讓 stubs 能告訴 kernel 哪個 system call 被要求執行，下圖是移除前及移除後:

```
// TODO (Open): define SC_Open (uncomment)
// TODO (Write): define SC_Write (uncomment)
// TODO (Read): define SC_Read (uncomment)
// TODO (Close): define SC_Close (uncomment)
//#define SC_Open   6
//#define SC_Read   7
//#define SC_Write  8
#define SC_Seek         9
//#define SC_Close  10
```

```
// TODO (Open): define SC_Open (uncomment)
// TODO (Write): define SC_Write (uncomment)
// TODO (Read): define SC_Read (uncomment)
// TODO (Close): define SC_Close (uncomment)
#define SC_Open     6
#define SC_Read     7
#define SC_Write    8
#define SC_Seek     9
#define SC_Close    10
```

2.3 userprog/ksyscall.h

接著到 ksyscal.h 裡把函式做為 interface。

```
// TODO (Open): Finish kernel interface for system call (Open).
// OpenFileId SysOpen(char *name) {}
OpenFileId SysOpen(char *name)
{
  return kernel->fileSystem->OpenAFile(name);
}

// TODO (Read): Finish kernel interface for system call (Read).
// int SysRead(char *buffer, int size, OpenFileId id) {}
int SysRead(char *buffer, int size, OpenFileId id)
{
  return kernel->fileSystem->ReadFile(buffer, size, id);
}

// TODO (Write): Finish kernel interface for system call (Write).
// int SysWrite(char *buffer, int size, OpenFileId id) {}
int SysWrite(char *buffer, int size, OpenFileId id)
{
  return kernel->fileSystem->WriteFile(buffer, size, id);
}

// TODO (Close): Finish kernel interface for system call (Close).
// int SysClose(OpenFileId id) {}
int SysClose(OpenFileId id)
{
  return kernel->fileSystem->CloseFile(id);
}
```

2.4 userprog/exception.cc

藉由觀察 create，可以發現他從 r4 拿出值存到 val 中

system call code – r2 //arg1 – r4 //arg2 – r5 //arg3 – r6 //arg4 – r7 所以函式需要的變數會依序存放在 r4,r5,r6，所以把所有變數拿出來之後轉成各自對應的參數結束後傳進 function 裡，並且在完成 system call 之後寫回 r2 並把 pc+4

```
// TODO (Open): Add SC_Open case to let the kernel able to handle this system call. You can imitate SC_Create case.
case SC_Open:
  val = kernel->machine->ReadRegister(4);
  {
    DEBUG(dbgSys, "Open "<<kernel->machine->ReadRegister(4)<<"+"<<kernel->machine->ReadRegister(5)<< "\n");
    char *filename = &(kernel->machine->mainMemory[val]);
    // cout << filename << endl;
    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;
// TODO (Write): Add SC_Write case to let the kernel able to handle this system call. You can imitate SC_Add case.
case SC_Write:
  val = kernel->machine->ReadRegister(4);
  {
    DEBUG(dbgSys, "Write "<<kernel->machine->ReadRegister(4)<<"+"<<kernel->machine->ReadRegister(5)<< "\n");
    char *buffer = &(kernel->machine->mainMemory[val]);
    size = kernel->machine->ReadRegister(5);
    id = kernel->machine->ReadRegister(6);

    status = SysWrite(buffer, size, id);
    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;
```

```
// TODO (Read): Add SC_Read case to let the kernel able to handle this system call. You can imitate SC_Add case.
case SC_Read:
  val = kernel->machine->ReadRegister(4);
  {
    DEBUG(dbgSys, "Read "<<kernel->machine->ReadRegister(4)<<"+"<<kernel->machine->ReadRegister(5)<< "\n");
    char *buffer = &(kernel->machine->mainMemory[val]);
    size = kernel->machine->ReadRegister(5);
    id = kernel->machine->ReadRegister(6);

    status = SysRead(buffer, size, id);
    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;
// TODO (Close): Add SC_Close case to let the kernel able to handle this system call. You can imitate SC_Create case.
case SC_Close:
  val = kernel->machine->ReadRegister(4);
  {
    //char *filename = &(kernel->machine->mainMemory[val]);
    // cout << filename << endl;
    status = SysClose(val);
    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;
```

2.5 filesys/filesys.h

Open: Openfiletable 是在存取 openfileid 的位置因此在 open 函式裡先假設要存取的位置為-1 看 openfiletable 有沒有可以存取的位置，若 openfiletable 已滿，fileDescriptor=-1，return -1 代表已滿了。如果有位置就可以呼叫 OpenForReadWrite 函式並傳入要打開的 name，存在 fileDescriptor 中。

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

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

    if (fileDescriptor == -1) return FALSE;
    Close(fileDescriptor);
    return TRUE;
  }
  // The OpenFile function is used for open user program  [userprog/addrspace.cc]
  OpenFile *Open(char *name) {
    int fileDescriptor = OpenForReadWrite(name, FALSE);
    if (fileDescriptor == -1) return NULL;
    return new OpenFile(fileDescriptor);
  }

  // The OpenAFile function is used for kernel open system call
  /* TODO (Open)
    1) If the file is not exist or OpenFileTable is full, return -1
    2) Otherwise, find the empty table to place the new created OpenFile and return its index.
  */
  // OpenFileId OpenAFile(char *name) {}
  OpenFileId OpenAFile(char *name)
  {
    int fileDescriptor = OpenForReadWrite(name,FALSE);
    if (fileDescriptor == -1) {
      return -1;
    }
    for(int i = 0;i < 20; i++)
    {
      if(OpenFileTable[i] == NULL)
      {
        OpenFileTable[i] = new OpenFile(fileDescriptor);
        return i;
      }
    }
    return -1;
  }
```

Write: 先從 id 判斷存在 Openfiletable 中的 openfile 是否可以開啟 (所以要滿足前面提到的兩個條件 id 位置正確跟檔案不等於 NULL)，如果失敗回傳-1，如果成功就可以呼叫 openfile.h 裡面的 write 函式成功讀寫並回傳讀寫入的字元數目。

```
int WriteFile(char *buffer, int size, OpenFileId id)
{
  if( id<20 && id>=0 )
  {
    DEBUG(dbgSys, "Writing Start\n");
    OpenFile* fileDescriptor = OpenFileTable[id];
    if(needop==NULL) return -1;
    else
    DEBUG(dbgSys, "Writing Completed\n");
    return  fileDescriptor->Write(buffer, size);
  }else return -1;
}
```

Read: 先從 id 判斷存在 Openfiletable 中的 openfile 是否可以開啟 (所以要滿足前面提到的兩個條件 id 位置正確跟檔案不等於 NULL)，如果失敗回傳-1，如果成功就可以呼叫 openfile.h 裡面的 Read 函式成功讀寫並回傳讀寫入的字元數目。

```
int ReadFile(char *buffer, int size, OpenFileId id)
{
  if( id<20 && id>=0 )
  {
    DEBUG(dbgSys, "Reading Start\n");
    OpenFile* fileDescriptor = OpenFileTable[id];
    if(needop==NULL) return -1;
    else
    DEBUG(dbgSys, "Reading Completed\n");
    return  fileDescriptor->Read(buffer, size);
  }else return -1;
}
```

Close: 先判斷 id 位置是否在 0-19 之間與檔案不等於 NULL，如果否回傳-1，如果是就可以 delete 掉 OpenFileTable[id] 並回傳 1。

```
int CloseFile(OpenFileId id)
{
  if(id < 0 || id >= 20 || OpenFileTable[id] == NULL){
    return -1;
  }
  else
  {
    DEBUG(dbgSys, "Closing Completed!\n");
    delete OpenFileTable[id];
    OpenFileTable[id] = NULL;
    return 1;
  }
}
```

3 Difficults & FeedBack

(1) 第一個遇到的問題與作業內容沒有甚麼太大關係，就是連線 VPN 時，發現用學校的公用 Wi-Fi 會無法連接，原因好像是因為有防火牆會阻擋，幸好有可行的網路，所以不構成甚麼問題。

(2) 一開始 trace code 時，真的有點不知道從何下手，尤其看到 Machine::Instruction() 那麼長時，真的有點不知道如何是好，看了很久才了解到大概流程，才開始上手。

(3) Implement 感謝助教有標示出 TODOS，不然要自己找的話真的會耗超多時間，但再修改 exception.cc 及 filesys.h 時 debug 也花了很久時間，希望可以的話能再多一點細部提示，謝謝。