Cover Page

Team61

Member List

> 108070039 羅稚涵
> 108070038 簡志宇

Member Contribution

> Trace code 羅稚涵, 簡志宇
> Part I report 羅稚涵
> Implementation 簡志宇
> Part II report 簡志宇

Part II-1

- SC_Halt()
  - machine/mipssim.cc: Machine::Run()

    這個 function 模擬 processor 在user mode運作的過程: 持續不斷地讀入一個指令。

    - 將 mode 設成 user mode

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

    - 呼叫此函式以處理 instruction

      ```
      OneInstruction(instr);
      ```

    ```
    void
    Machine::Run()
    {
        Instruction *instr = new Instruction;  // storage for decoded instruction

        if (debug->IsEnabled('m')) {
            cout << "Starting program in thread: " << kernel->currentThread->getNam
    e();
            cout << ", at time: " << kernel->stats->totalTicks << "\n";
        }
        kernel->interrupt->setStatus(UserMode);
        for (;;) {
          OneInstruction(instr);

          kernel->interrupt->OneTick();

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

  - machine/mipssim.cc: Machine::Oneinstruction()

    這個 function 模擬 cpu 讀入 instruction (from user program) 後執行的過程

Fetch an instruction and decode it

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

當 instruction 為 OP_SYSCALL 時, 呼叫 RaiseException() 以處理 system call

```
 switch (instr->opCode) {
    .
    .
    .
    .
    case OP_SYSCALL:
      DEBUG(dbgTraCode, "In Machine::OneInstruction, RaiseException(SyscallExcep
tion, 0), " << kernel->stats->totalTicks);
      RaiseException(SyscallException, 0);
    return;
    .
    .
    .
    .
}
```

- machine/machine.cc: Machine::RaiseException()

  這個 function 在 user program 呼叫 system call 或是有 exception 發生時負責將 mode 轉到 kernel mode, 並呼叫 ExceptionHandler() 來處理。處理完之後再將 mode 設回 user mode

  ex. 前面當instruction 為 OP_SYSCALL 呼叫時傳入 ExceptionType = SyscallException (type 定義於 machine.h 中)以進入kernel mode 準備處理system call

```
void
Machine::RaiseException(ExceptionType which, int badVAddr)
{
    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);
}
```

- userprog/exception.cc: ExceptionHandler()

這個 function 處理由 user program 所產生的 exception 或 systemcall (以SC_Halt為例子)

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

- userprog/ksyscall.h: SysHalt()

  這個 function 為 kernel 裡處理 system call halt 的接口, 會再呼叫到位於其他地方(interrupt::Halt())的 service routine

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

- machine/interrupt.cc: Interrupt::Halt()

  這個 function 處理 system call halt, 執行之後直接 shutdown NachOS

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

- (b)

  Machine::OneInstruction()會呼叫Machine::RasieException()執行system call。接下來由Machine::RasieException()執行相對應的system call來呼叫在NachOS kernel裡的ExceptionHandler(), 執行完回到user program的Machine::OneInstruction。

- SC_Create
  - userprog/exception.cc: ExceptionHandler()

對應到的type為SC_Create，會先取得一個register，將MIPS machine自動存入的參數取出，再將其physical memory的指標位置賦值給filename，然後交給SysCreate()，並將SysCreate()回傳給status的值寫入register2。接下來將各個program counter加4，是避免持續執行相同instruction，加4是因為每個記憶體內存資料量大小是1 Byte，完整的32Bits指令會分別存在4個記憶體位址中，所以每次program counter都是加4。

```
case SC_Create:
val = kernel->machine->ReadRegister(4);
{
  char *filename = &(kernel->machine->mainMemory[val]);
  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;
```

- userprog/ksyscall.h: SysCreate()

  由這個function呼叫Create()，當回傳給SysCreate()是True時，會再回傳1給SC_Crate的status，反之則是回傳0給SC_Create裡的status。

```
int SysCreate(char *filename)
{
  return kernel->fileSystem->Create(filename);
}
```

- filesys/filesys.h: FileSystem::Create()

  此function裡面的OpenForWrite()在創造的檔案已經存在或是創檔失敗時，fileDescriptor會是-1，回傳到SysCreate()會是False，若為新創的檔案回傳到SysCreate()為True。

```
bool Create(char *name)
{
  int fileDescriptor = OpenForWrite(name);
  if (fileDescriptor == -1) return FALSE;
  Close(fileDescriptor);
  return TRUE;
}
```

- (b)

  Machine::OneInstruction()會呼叫Machine::RasieException()執行system call。接下來由Machine::RasieException()執行相對應的system call來呼叫在NachOS kernel裡的ExceptionHandler()。進到kernel後，再經由ksyscall.h做為interface來進入到file system裡面。最後，filesys.h就會將指令傳入file sysetem裡。

- SC_PrintInt
  - userprog/exception.cc: ExceptionHandler()

    進入到該funciton後會呼叫SysPrintInt()。SC_PrintInt後面步驟與SC_Create相似就不贅述

```
case SC_PrintInt:
    DEBUG(dbgSys, "Print Int\n");
    val=kernel->machine->ReadRegister(4);
    DEBUG(dbgTraCode, "In ExceptionHandler(), into SysPrintInt, " << kernel->st
ats->totalTicks);
    SysPrintInt(val);
    DEBUG(dbgTraCode, "In ExceptionHandler(), return from SysPrintInt, " << ker
nel->stats->totalTicks);
    kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCR
eg));
    kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg)
 + 4);
    kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCR
eg)+4);
    return;
    ASSERTNOTREACHED();
    break;
```

  - userprog/ksyscall.h: SysPrintInt()

    該function是要呼叫SynchConsoleOutput::PutInt()，傳入 val(要print 的數字)，作為處理 system call PrintInt的接口

```
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->PutI
nt, " << kernel->stats->totalTicks);
}
```

  - userprog/synchconsole.cc: SynchConsoleOutput::PutInt()

    userprog/synchconsole.cc: SynchConsoleOutput::PutChar()

lock->Acquire()為確認lock是否為busy，再進入下面的do-while迴圈裡。當有輸入一個字元時，會先確認此字元是否能進來還是需要再等待，如果能進來idx就會增加一，watiFor->P()是用來觀察下個字元是否能進來，最後直到輸入空字元才會跳出迴圈。結束迴圈後，會將lock設為free。

```
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->PutCha
r, " << 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(), " << ke
rnel->stats->totalTicks);
  waitFor->P();
  DEBUG(dbgTraCode, "In SynchConsoleOutput::PutChar, return form  waitFor->P(),
" << kernel->stats->totalTicks);
  } while (str[idx] != '\\0');
  lock->Release();
}
```

- machine/console.cc: ConsoleOutput::PutChar()

  模擬 hardware device controller，一開始先確認PutChar是否busy，如果沒有就會進入WriteFile()，並在未來安排 interrupt (要寫回去file)。

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

- machine/interrupt.cc: Interrupt::Schedule()

  這裡的when是用來記錄什麼時間interrupt要介入，接著會產生hardware interrupt (toOccur)將其安排到該發生的時間，確認fromNow > 0後，

則會insert interrupt。其中 toCall 是連到 callback function，當做完一連串的動作後會 call 回去。

```
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);
}
```

- machine/mipssim.cc: Machine::Run()

  前面的 interrupt 被 schedule 好之後就讓 machine 繼續執行 user program，直到系統時間到達 interrupt 須被處理的時刻再進行處理，而這個過程由下面的 OneTick() 實現。

- machine/interrupt.cc: Interrupt::OneTick()

  OneTick()主要目的是設定interrupt狀態。一開始會先將現在 machine 狀態儲存下來。再來，OneTick()會先分辨現在的狀態是 kernel mode 還是 user mode，並計算TotoalTicks的時間和當時狀態下的Ticks。再來呼叫 ChangeLevel(IntOn, IntOff)， 因為 interrupt handler 要處理 interrupts 時要將 interrupt (其他的) disable 掉，並且在處理完之後 enable。

  接下來，會呼叫 CheckIfDue() 處理 interrupts，傳入的參數 為 FALSE 代表處理過程中不會推進時間。

  最後，當 yieldOnReturn 為True時，也就是可以context switch時，yieldOnReturn 會被設定為False，之後kernel目前執行的thread會被釋放掉，換執行在ready中第一個 thread。

```
void
Interrupt::OneTick()
{
  MachineStatus oldStatus = status;
  Statistics *stats = kernel->stats;
  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) {
    yieldOnReturn = FALSE;
    status = SystemMode;
    kernel->currentThread->Yield();
    status = oldStatus;
  }
}
```

- machine/interrupt.cc: Interrupt::CheckIfDue()

  CheckIfDue 是要確認規劃好的 interrupt 是否有如期發生。下面的迴圈是當pending 的interrupt list 不是空的話，會先將 list 中 的 interrupt 依序執行，直到 interrupt 被預定的執行時間大於現在時間，其中呼叫 next->callOnInterrupt->CallBack() 是要讓 interrupt handler 處理。跳出迴圈後，會將 interrupt handler 設為False，最後回傳 True 表示到期的 interrupt 均已完成處理。

```
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->Call
Back, " << stats->totalTicks);
        next->callOnInterrupt->CallBack();// call the interrupt handler
        DEBUG(dbgTraCode, "In Interrupt::CheckIfDue, return from callOnInterrup
t->CallBack, " << stats->totalTicks);
        delete next;
    } while (!pending->IsEmpty()
        && (pending->Front()->when <= stats->totalTicks));
    inHandler = FALSE;
    return TRUE;
}
```

- machine/console.cc: ConsoleOutput::CallBack()

  CallBack()是要initialize硬體，一開始會先將 putBusy 設為 FALSE，確保 output 時能正常執行。下一行的 numConsoleCharsWritten記錄將顯示的字元，最後則再呼叫 callWhenDone->CallBack() (回到SynchConsoleOutput object)來告知什麼時候能將下個字元放入。

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

- userprog/synchconsole.cc: SynchConsoleOutput::CallBack()

  CallBack()是用來接收由 ConsoleOutput 傳回來的訊息，告知現在可以傳入下一個新的字元

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

- (b)

  Machine::OneInstruction()會呼叫Machine::RasieException()執行system call。接下來由Machine::RasieException()執行相對應的system call來呼叫在NachOS kernel裡的ExceptionHandler()。進到kernel後，再經由ksyscall.h裡的SysPrintInt來呼叫真正的system call。接著靠sychnconsole.cc來同步進入到hardware device。然後由console.cc模擬由port到console device的路徑。再來的interrupt.cc只是一個hardware device的模擬器，所以與電腦是不同步的，這時就會呼叫Interrupt::Schedule()來安排interrupt的出現時候。然後又會回去Machine::Run()再來一次，並判斷是否有interrupt進來，最後確認無誤時候就會display在螢幕上。

Part II-2 report

- 程式執行流程
  - user program –> syscall.h –> start.S –> exception.cc –> ksyscall.h –> filesys.h

    (–>sysdep.h)
- 細部實作解說
  - user program (此部分無實作)
    - user program 執行程式時有進行檔案操作這種會呼叫到system call的動作，就會呼叫 syscall.h 裡的 system call 函式 (API), 例如: Write(), Read() 等.
    - ex. fileIO_test1.c

```
int success = Create("file1.test");
```

```
if (success != 1) MSG("Failed on creating file");
```

```
fid = Open("file1.test");
```

```
int count = Write(test + i, 1, fid);
```

```
success = Close(fid)
```

```
Halt();
```

  - syscall.h
    - 由 user program 呼叫這個檔案裡定義的 API, 以呼叫 ksyscall.h 裡的函式(底層真正的 system call 函式)

- 將幾個被註解掉的 SC_...(system call 代碼) 打開使其生效, 這些是用來連接 start.S 的stub

```
// to make SC of open/read/write/close executable
#define SC_Open   6
#define SC_Read   7
#define SC_Write  8
#define SC_Close  10
```

- 如此一來就能透過 exception handler (in exception.cc)來辨別是哪種 system call 在呼叫相對應的system call函式

◦ start.S (以 Open 為例子, 其他類似)

- 透過此 stub 將對應的 system call 所需資料放入 register(r2 for SC, r4 ~ r7 for parameters) 中

```
  .globl Open // 表示 Open 為 global 函式
  .ent   Open // 進入 Open
Open:
  addiu $2,$0,SC_Open // 將 SC_Open 放入 r2 以供之後讀取資料使用
  syscall // 呼叫 syscall 指令
  j $31 // 跳回 r31 儲存的位址(原本的位址)
  .end Open // 結束 Open
```

◦ exception.cc (以 Open 為例子, 其他類似)

- 用 exception handler 來辨識何種 SC, 並且將資料從記憶體讀出來, 再呼叫 ksyscall.h 的函式
- ExceptionHandler(ExceptionType which)

1. 從 r2 讀出何種 SC
2. 從 r4 ~ r7(視何種SC而定) 讀出參數
3. 呼叫 ksyscall.h的函式
4. 將回傳值放入 r2
5. program counter 加 4 (往後一個指令)

p.s. which 是用來辨別何種 exception, 這次實作部分與此無關, 且這次的 exception 都是 SyscallException

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

```
...
case SC_Open: // 1.Open

    val = kernel->machine->ReadRegister(4); //2.
    {
        char *filename = &(kernel->machine->mainMemory[val]);
        fileID = SysOpen(filename); // 3.
        kernel->machine->WriteRegister(2, fileID); //4.
    }
    // 5.
    kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegist
er(PCReg));
    kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(P
CReg) + 4);
    kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegist
er(PCReg)+4);
    return;
    ASSERTNOTREACHED();
break;
```

- ksyscall.h
  - 定義各種 system call 函式(與 file operation 有關), 呼叫對應的 service routine (in filsesys.h)

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

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

- filesys.h
  - file operation 的 service routine (用 OpenFile object 實作, 維護 OpenFileTable 讓OpenFile 同時不超過20個)
  - 這邊的 service routine 會呼叫 sysdep.h 的函式, 其用 C++ library 呼叫 linux 的 system call 模擬真正的 filesystem 的檔案操作
  - OpenAFile

```
OpenFileId OpenAFile(char *name) {
  OpenFile *fileOpened = Open(name);
  if(fileOpened == NULL)
      return -1;
  int id;
  for(id = 0; id < 20; id++){
      if(OpenFileTable[id] == NULL){
    OpenFileTable[id] = fileOpened;
    return id;
    //break;
      }
  }
  if(id > 19 || id < 0) return -1;
}
```

- `WriteToFile`

```
int WriteToFile(char *buffer, int size, OpenFileId id){
  if(id > 19 || id < 0)
      return -1;
  OpenFile *fileOpened = OpenFileTable[id];
  if(fileOpened == NULL)
      return -1;
  return fileOpened->Write(buffer, size);
}
```

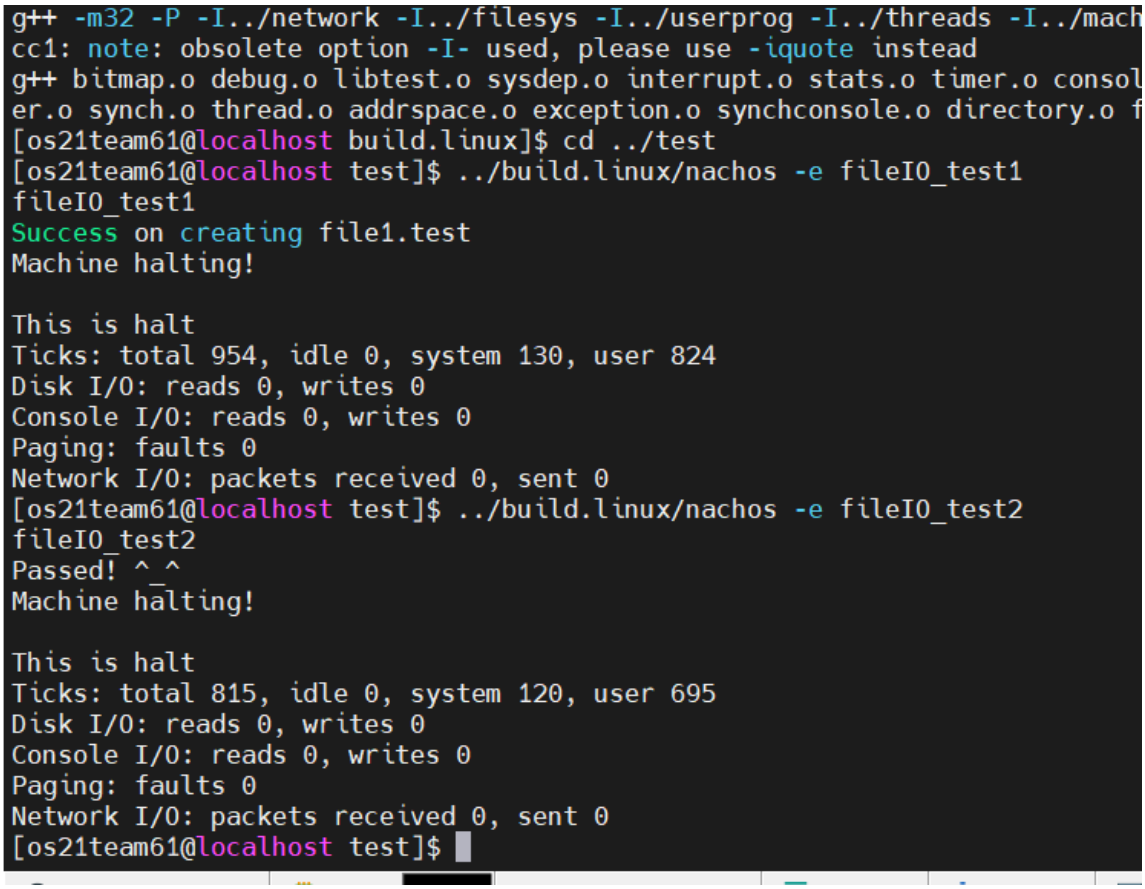
- `ReadFile`

```
int ReadFile(char *buffer, int size, OpenFileId id){
  if(id > 19 || id < 0)
      return -1;
  OpenFile *fileOpened = OpenFileTable[id];
  if(fileOpened == NULL)
      return -1;
  return fileOpened->Read(buffer, size);
}
```

- `CloseFile`

```
int CloseFile(OpenFileId id){
  OpenFile *fileOpened = OpenFileTable[id];
  if(fileOpened == NULL)
      return -1;
  delete fileOpened;
  OpenFileTable[id] = NULL;
  return 1;
}
```

    - sysdep.h(非實作)
        - OpenForReadWrite, Read, Wrtie, Close 等
- 測試結果



Difficulties & Feedback

羅稕涵

這是我第一次接觸到NachOS，trace code的過程只覺得code沒有那麼直覺，無法馬上理解出寫這行code的意思是什麼，而且還需要不停的去尋找這個function又呼叫了哪個function，所以在第一次的trace code是花了滿多時間的，尤其是在SC_Create那塊花的時間最多。再來，遇到的問題就是在不同的function call間的意思代表什麼也花了不少的時間去理解。

簡志宇

一開始面對如大海一般綿密的程式碼，簡直讓我喘不過氣，不知從何處下手是好。不過萬事起頭難，當我克服心理障礙開始 trace 第一部分的 code 的時候就倒吃甘蔗，漸入佳境。但這樣的好景並不常見，在第三個部分的 SysPrintInt 我又碰到困難了，一堆 callback function 兜過來兜過去，搞得我暈頭轉向，老實說這大概是這次 MP 最難的part了吧。

至於實作部分，原本我是直接使用 sysdep.h 的函式實作各項功能，後來再改成用 OpenFile obiect 的版本。起初剛改完的時候，不論我怎麼測試 OpenAFile 回傳 0 時就會導致 "Failed on opening file"。之後試了許多方法、改了好幾種寫法都找不到bug。直到最後我重新make fileIO_test1 跟 fileIO_test2 ，問題才迎刃而解。

這次作業頗為繁雜，不過也讓我搞懂了 NachOS 大致上的運作原理。