作業系統 - MP1 – System Call

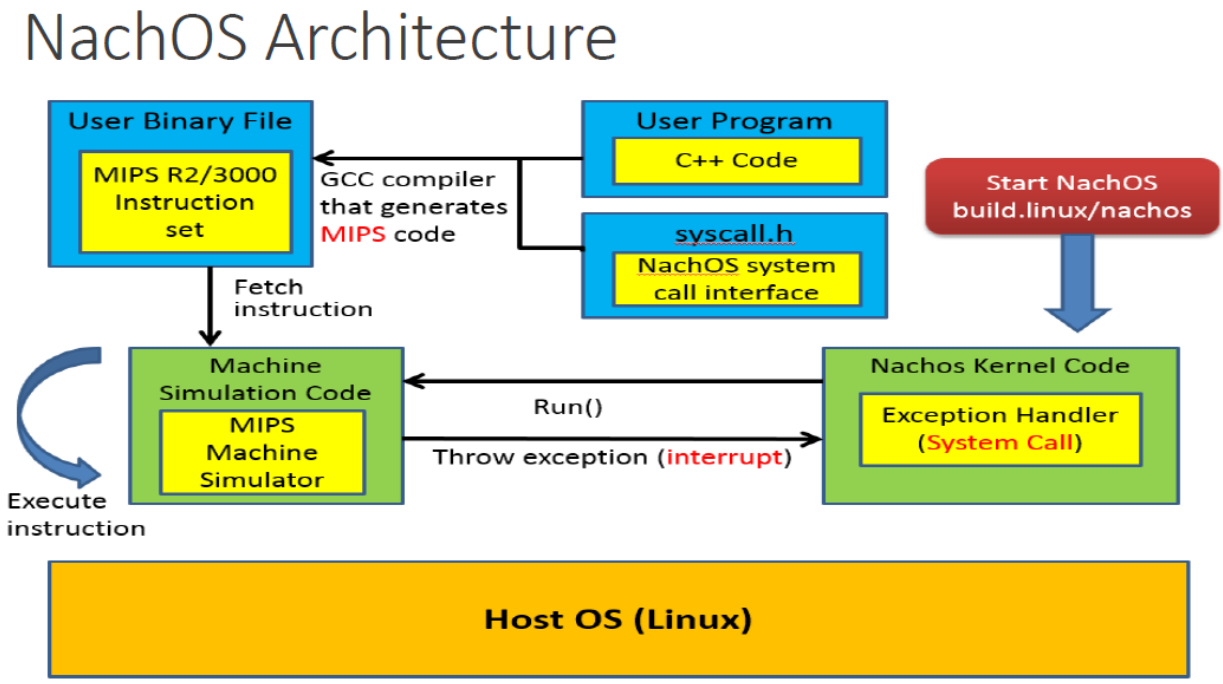
學號:1102003S

姓名:陳觀宇

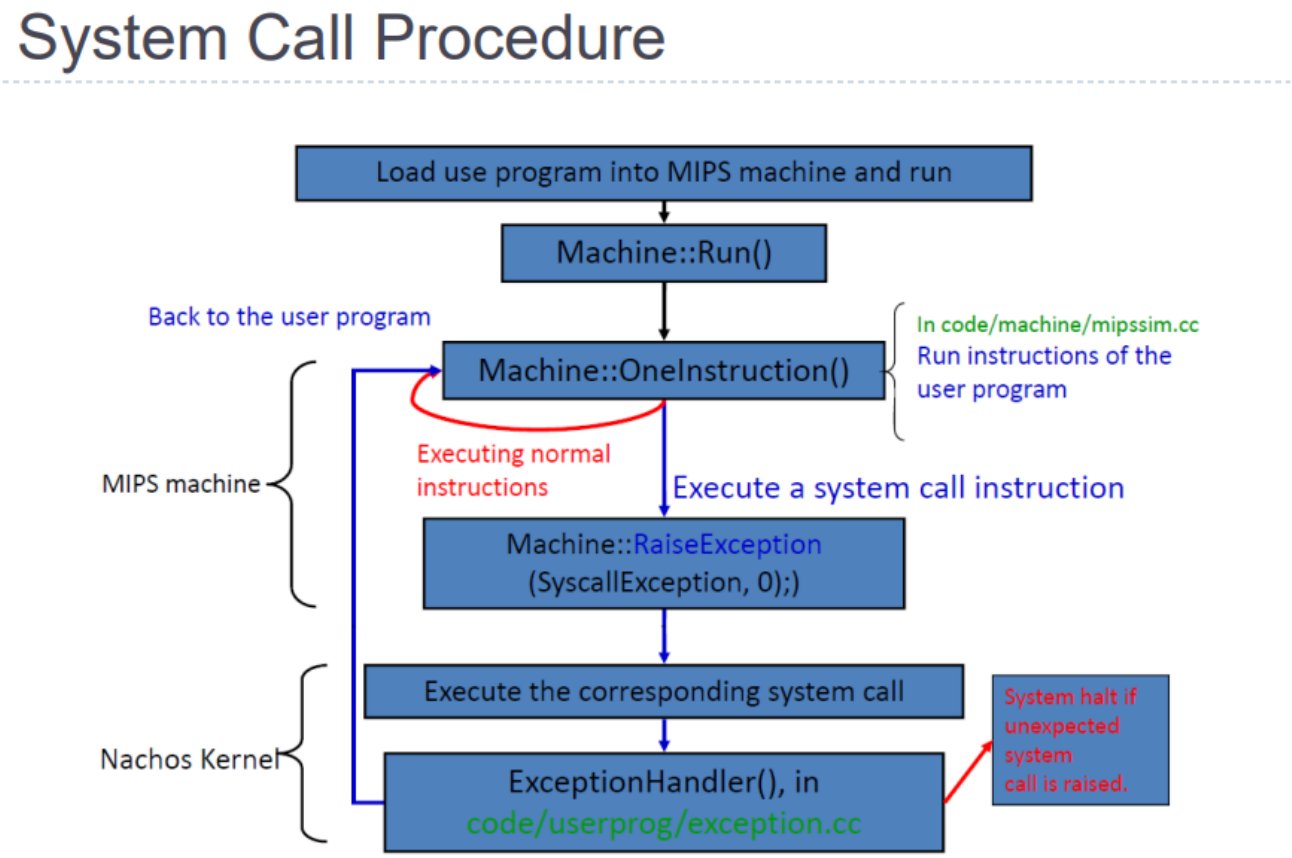
HackMD連結: Link

Part 1 Trace Code

NachOS 架構圖:



system call 執行流程:



- 1.Machine::Run()→NACHOS的虛擬機器，模擬MIPS CPU執行user program編譯後的assembly code。
- 2.Machine::OneInstruction()→把user program decode成MIPS指令，並判斷出此指令為system call，並呼叫fucntion處理system call異常。
- 3.Machine::RaiseException(SyscallException,0)→由於是user program執行需要請kernel協助執行system call，因此將mode從user mode切換成system mode(kernel mode)，並呼叫ExceptionHandler處理system call異常，待處理完system call之後再將mode從system mode(kernel mode)切換回user mode。
- 4.ExceptionHandler()→判斷system call的類別，呼叫kernel執行，完成後再返回Machine::OneInstruction()，繼續執行下一個指令。

(1) System Call : SC_Halt (Sample code: halt.c)

halt.c

```
#include "syscall.h"

int
main()
{
    Halt();
    //呼叫Halt函數，介面定義在syscall.h
    //實作定義在start.S(組合語言)
    /* not reached */
}
```

- 1.user program(c code)透過NachOS的System Call Interface(syscall.h)呼叫system call對應的stub，接著透過NashOS的compiler將c code轉成MIPS Binary Code，透過NachOS Kernel處理指令與模擬的Machine執行指令。
- 2.根據halt.c文件定義，此user program呼叫Halt()，而此功能會影響NachOS執行，因此需要會觸發interrupt，使程式程序trap to kernel，故屬於system call。，

syscall.h <僅擷取部分程式碼>

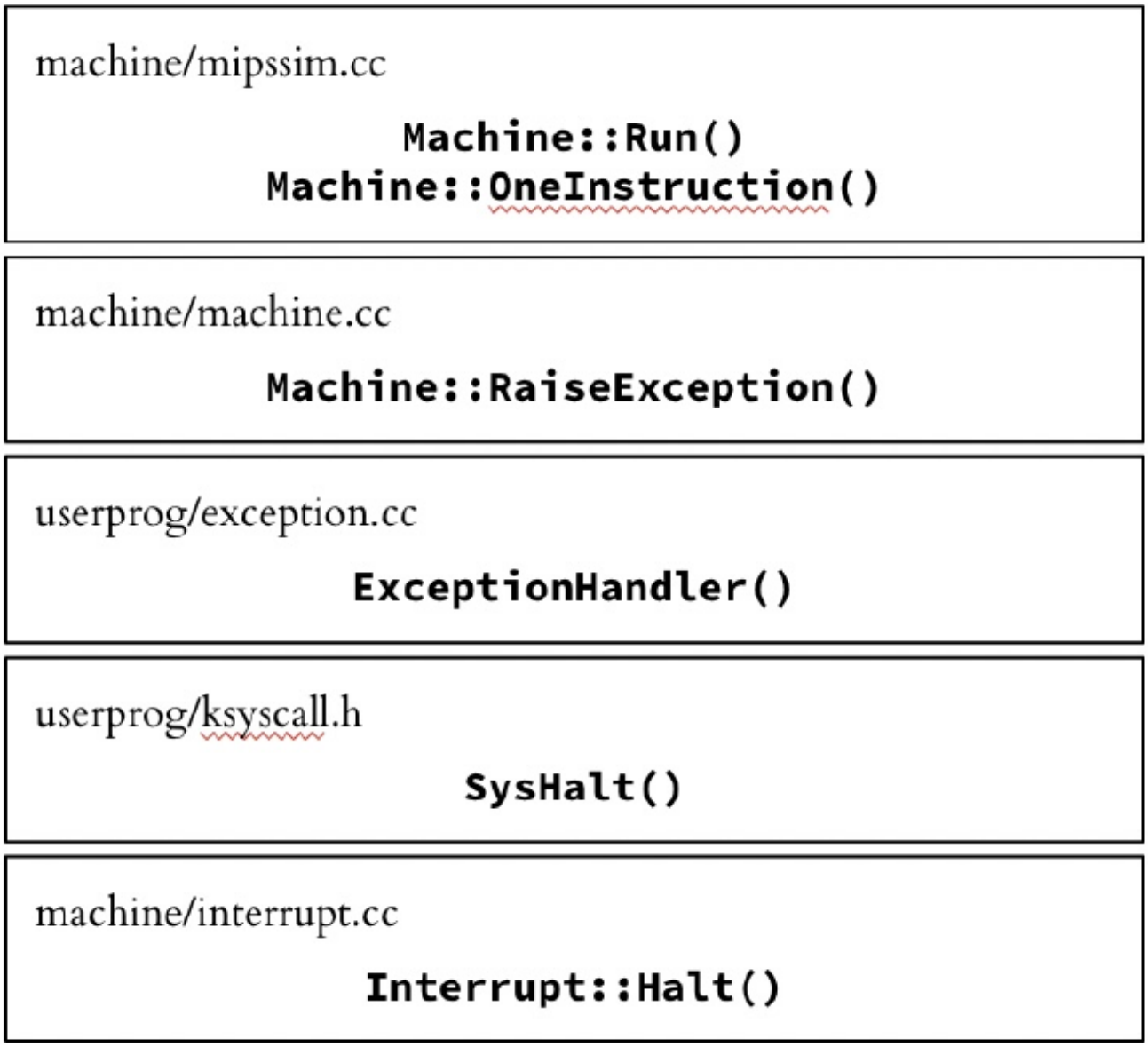
```
#define SC_Halt     0
//SC_Halt 對應的system call stub number為0

/* Stop Nachos, and print out performance stats */
void Halt();
```

- syscall.h定義了Halt function介面與SC_Halt的stub code(0)

start.S <僅擷取部分程式碼>

```
    .globl Halt
    //宣告“Halt”為全域變數
    .ent    Halt
    //設定“Halt”函數的起始點
Halt:
    //“Halt”函數調用將從此位置開始
    addiu $2,$0,SC_Halt
    //將system call SC_Halt對應的stub code(0)存入r2，
    //stub code定義在syscall.h
    syscall
    //呼叫system call
    //由於Halt()函數沒有傳入參數，
    //因此不將參數寫入r4(arg1)、r5(arg2)、r6(arg3)、r7(arg4)、
    j   $31
    //執行完system call後,返回到RetAddrReg r31(user program)
    .end Halt
    //“Halt”函數的終止點
```

- 1.根據start.S文件說明，此assembly language(為了system call執行效率，因此用組合語言撰寫)可以輔助NachOS kernel執行system calls，當呼叫system call時會根據system call類別(stub code)進行對應處理。
- 2.根據system call Halt函數的stub定義，會依序完成以下事情:
    - (1) 將system call SC_Halt對應的stub code(0)存入r2。
    - (2) 呼叫system call。
    - (3) 執行完system call後,返回到RetAddrReg r31(user program address)。



mipssim.cc Machine::Run() <僅擷取部分程式碼>

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

  kernel->interrupt->setStatus(UserMode);
    // 將mode設定為UserMode
  for (;;) {
    // 無窮迴圈
    OneInstruction(instr);
    // 模擬MIPS CPU逐行執行指令

    kernel->interrupt->OneTick();
    // 確認是否有其他interrupt已經到達預定執行時間，需要優先處理
  }
}
```

- 1.根據mipssim.cc文件說明，Machine::Run()是模擬Nashos執行user-level program，當program啟動時，會由kernel呼叫Machine::Run()執行，且此函數不會返回(CPU不斷讀取指令執行)。
- 2.根據Machine::Run()的定義，會依序完成以下事情:
    - (1) 建立Instruction class pointer instr紀錄instruction的物件記憶體位置。
    - (2) 將Mode設定為UserMode。
    - (3) 進入無窮迴圈，透過OneInstruction(instr)function傳入instruction，模擬MIPS CPU逐行執行指令。
    - (4) 利用OneTick()確認是否有其他interrupt已經到達預定執行時間，需要優先處理。

mipssim.cc Machine::OneInstruction() <僅擷取部分程式碼>

```
void Machine::OneInstruction(Instruction *instr) {

  int raw;
  int nextLoadReg = 0;
  int nextLoadValue = 0;
  // record delayed load operation, to apply
  // in the future

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

  // Compute next pc,
  // but don't install in case there's an error or branch.
  int pcAfter = registers[NextPCReg] + 4;
  int sum, diff, tmp, value;
  unsigned int rs, rt, imm;

  // Execute the instruction (cf. Kane's book)
  switch (instr->opCode) {
    //利用opCode判斷指令類別
    case OP_SYSCALL:
    //若指令為system call
      RaiseException(SyscallException, 0);
    //引發例外處理
      return;
  }
 // Now we have successfully executed the instruction.

  // 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;
}
```

- 1.根據mipssim.cc文件說明，Machine::OneInstruction()是執行來自user-level program的instruction。
- 2.根據Machine::OneInstruction()的定義，會依序完成以下事情:
  - (1) 完成instruction fetch跟instruction decode，找出instruction對應的opcode。
  - (2) 利用instruction opcode判斷出此instruction為system call，呼叫RaiseException(SyscallException, 0)引發例外處理。
  - (3) 如果指令能順利完成，則最後會更新program counter到下一個指令的memory address位置，讓模擬MIPS CPU的虛擬machine能繼續執行下一個指令。

machine.h <僅擷取部分程式碼>

```
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
};
```

- ExceptionType定義在machine.h，若Excetption類別為system call，其對應的ExceptionType為SyscallException。

machine.cc Machine::RaiseException() <僅擷取部分程式碼>

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

```
    registers[BadVAddrReg] = badVAddr;
    //從register取出發生trap的virtual address(badVAddr)
    DelayedLoad(0, 0);
    // finish anything in progress
    // 進行延遲載入
    kernel->interrupt->setStatus(SystemMode);
    //將Mode設定為SystemMode
    ExceptionHandler(which);
    // interrupts are enabled at this point
    kernel->interrupt->setStatus(UserMode);
    //將Mode設定為UserMode
}
```

- 1.根據machine.cc文件說明，當user program需要調用system call或是某些exception發生時(例如address translation failed)，就會呼叫Machine::RaiseException()，轉移系統控制權從user mode到NachOS kernel。
- 2.function傳入參數:
  - (1) which: 造成kernel trap的原因，而which的型別為ExceptionType，其定義在Machine.h，由於此引發Exception是system call，因此傳入which參數為SyscallException。
  - (2) badVaddr: 發生trap的virtual address，由於c code main呼叫system call引發trap，因此根據start.S文件說明，此程式會loaded at location 0 (也就是r0)，故傳入dadVaddr參數為0。
- 3.根據Machine::RaiseException()的定義，會依序完成以下事情:
  - (1) 取出從register取出發生trap的virtual address(badVAddr)
  - (2) 根據mipssim.cc文件說明，當trap to kernel之前需要進行delay load延遲載入，加速指令執行。
  - (3) 切換Mode從UserMode變成SystemMode，將控制權交給NachOS kernel。
  - (4) 呼叫ExceptionHandler(which)，根據Exception類別which進行例外處理。
  - (5) 完成例外處理後，再次切換Mode從SystemMode變成UserMode，將控制權返還給User Program。

exception.cc ExceptionHandler() <僅擷取部分程式碼>

```
void ExceptionHandler(ExceptionType which) {
  char ch;
  int val;
  int type = kernel->machine->ReadRegister(2);
    //從register r2取出system call code存入type
  int status, exit, threadID, programID, fileID, numChar;
  switch (which) {
  // 進行which判斷
    case SyscallException:
    // 當which為SyscallException
      switch (type) {
      // 進行type判斷
        case SC_Halt:
        // 當type為SC_Halt
          SysHalt();
          // 呼叫SysHalt()，其實作定義在ksyscall.h
          cout << "in exception\n";
          ASSERTNOTREACHED();
```

```
            // degbug用
            break;
        }
        break;
    }
}
```

- 1.根據exception.cc文件說明，ExceptionHandler()為進入NachOS kernel的進入點。當user program需要kernel協助執行syscall或是產生addressing exception或 arithmetic exception時，則會呼叫ExceptionHandler()進行處理。
- 2.function傳入參數:
    - (1) which: 造成kernel trap的原因，而which的型別為ExceptionType，其定義在Machine.h，由於此引發Exception是system call，因此傳入which參數為SyscallException。
- 3.根據ExceptionHandler()的定義，會依序完成以下事情:
    - (1) 讀取register r2的資料(對應system call stub code)，作為type參數。根據system call stub定義，因此type為SC_Halt對應的stub code(0)。因此system call SC_Halt的type參數是由user program將參數寫入registers，再透過kernel讀取對應registers獲得。(pass parameters in registers)
    - (2) 利用傳入參數which判斷是哪一種Exception，因為指令為system call，因此which為SyscallException。
    - (3) 第一層switch case判斷which會對應到SyscallException。
    - (4) 第二層switch case判斷type會對應到SC_Halt。
    - (5) 呼叫SysHalt()進行system call SC_Halt。

ksyscall.h sysHalt()

```
void SysHalt()
{
  kernel->interrupt->Halt();
    //透過kernel的interrupt呼叫Halt()
}
```

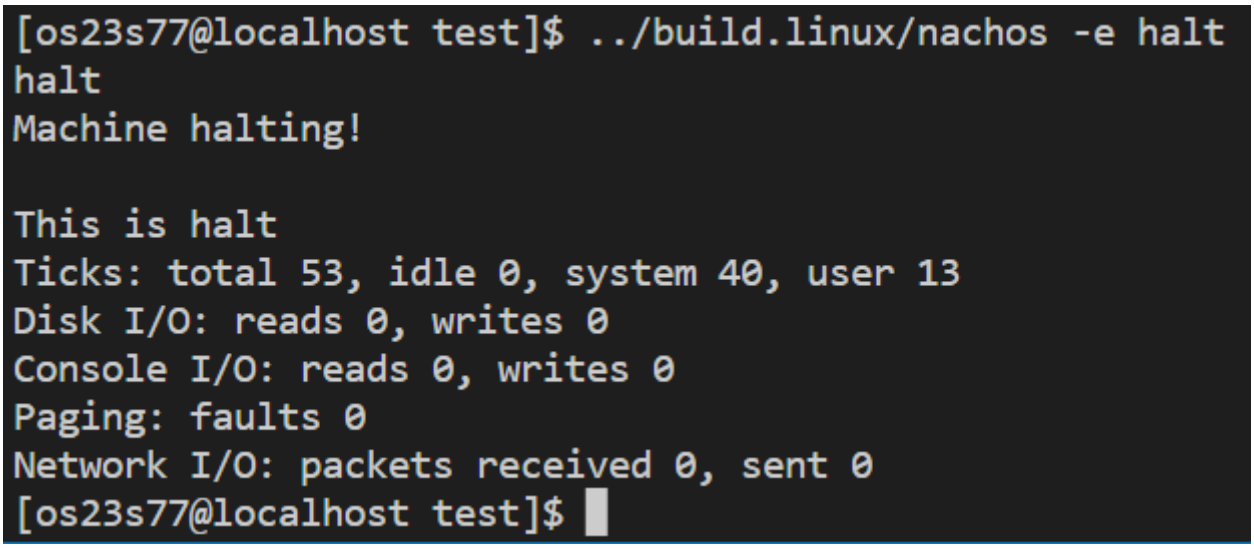
- 1.根據ksyscall.h文件說明，ksyscall.h為kernel執行system call的interface。
- 2.根據SysHalt()的定義，會透過透過kernel的interrupt呼叫Halt()進行system call SC_Halt處理。

interrupt.cc Halt()

```
void Interrupt::Halt() {
  cout << "Machine halting!\n\n";
  cout << "This is halt\n";
  kernel->stats->Print();
  // 透過kernel的stats統計呼叫print印出kernel performance數據
  delete kernel;
  // Never returns.
}
```

- 1.根據interrupt.cc文件說明，Interrupt::Halt()是用來關閉NachOS，並印出performance數據
- 2.根據Interrupt::Halt()的定義，會依序完成以下事情:
  - (1) 透過kernel的stats呼叫print()印出kernel performance數據。
  - (2) 釋放kernel物件的記憶體，因為kernel為NachOS啟動時核心，用來處理啟動主線程、創建線程調度表、中斷處理模塊、CPU、控制台、文件系統、中斷等NachOS功能，因此一旦kernel物件記憶體被釋放，就無法再透過kernel執行相關NachOS功能，等同於NachOS系統被關閉。

Sample code: halt.c 執行結果



(2) System Call : SC_Create (Sample code: createFile.c)

createFile.c

```
#include "syscall.h"

int main(void)
{
    int success= Create("file0.test");
    if (success != 1) MSG("Failed on creating file");
    MSG("Success on creating file0.test");
    Halt();
}
```

- 1.根據createFile.c文件，此user program呼叫Create()、MSG()、Halt()三個function，而Create()建立檔案需要用到file system、MSG()顯示訊息到command line上需要調用I/O資源、Halt()關閉kernel會影響OS運作，因此上述三個fuction皆會觸發interrupt，使程式程序trap to kernel，故屬於system call。
- 2.本段trace Code專注在Create()後續的code path。
- 3.根據createFile.c的定義，會依序完成以下事情:
  - (1) 使用Create()建立"file0.test"檔案，並回傳數值存入success變數。Create檔案成功，則success=1；Create檔案失敗則success!=1。
  - (2) 透過success數值，利用MSG()在command line顯示對應訊息。若Create檔案成功，則顯示"Success on creating file0.test"；若Create檔案失敗，則顯示"Failed on creating file"。
  - (3) 呼叫Halt()，釋放Kernel物件記憶體，停止NachOS系統。

syscall.h <僅擷取部分程式碼>

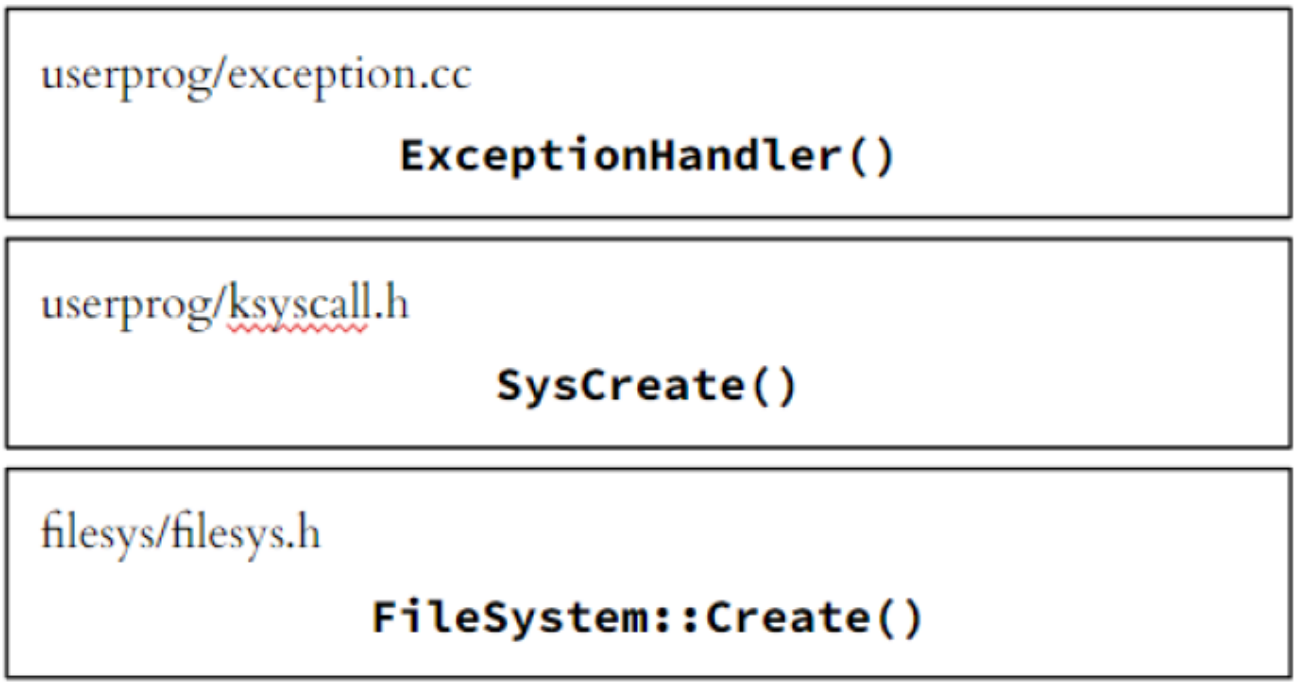
```
#define SC_Create   4
//SC_Create 對應的system call stub number為4

/* Create a Nachos file, with name "name" */
/* Note: Create does not open the file.   */
/* Return 1 on success, negative error code on failure */
int Create(char *name);
```

- syscall.h定義了Create function介面與SC_Create的stub code(4)

start.S <僅擷取部分程式碼>

```
    .globl Create
    //宣告“Create”為全域變數
    .ent    Create
    //設定“Create”函數的起始點
Create:
    //“Create”函數調用將從此位置開始
    addiu $2,$0,SC_Create
    //將system call SC_Create對應的stub code(4)存入r2，
    //stub code定義在syscall.h
    syscall
    //呼叫system call，
    //將Create()函數參數name放入r4(arg1)
    //r4為name字串開頭記憶體位置對應在mainMemory的index數值
    j   $31
    //執行完system call後,返回到RetAddrReg r31(user program)
    .end Create
    //“Create”函數的終止點
```

- 1.根據system call Create函數的stub定義，會依序完成以下事情:
  - (1) 將system call SC_Create對應的stub code(4)存入r2。
  - (2) 呼叫system call。
  - (3) 執行完system call後,返回到RetAddrReg r31(user program address)。



exception.cc ExceptionHandler() <僅擷取部分程式碼>

```
void ExceptionHandler(ExceptionType which) {
  char ch;
  int val;
  int type = kernel->machine->ReadRegister(2);
  //從register r2取出system call code存入type
  int status, exit, threadID, programID, fileID, numChar;
  switch (which) {
  // 進行which判斷
    case SyscallException:
    // 當which為SyscallException
      switch (type) {
      // 進行type判斷
        case SC_Create:
        // 當type為SC_Create
          val = kernel->machine->ReadRegister(4);
          // 從register r4取出arg1
          // (file name對應的mainMemory開頭index)存入val
          {
            char *filename = &(kernel->machine->mainMemory[val]);
            // 利用val當作index存取mainMemory陣列
            // 並用filename紀錄對應檔案名稱在mainMemory[val]的開頭記憶體位置
            // cout << filename << endl;
            status = SysCreate(filename);
            // 呼叫SysCreate傳入filename進行檔案建立
            // 將回傳值存入status
            kernel->machine->WriteRegister(2, (int)status);
            // 把SysCreate回傳結果status寫回register r2
          }
          // 更新program counter register數值
```

```
            kernel->machine->WriteRegister(PrevPCReg, kernel->machine-
>ReadRegister(PCReg));
            // 將前一個指令的pc counter更新為現在指令的pc counter
            kernel->machine->WriteRegister(PCReg, kernel->machine-
>ReadRegister(PCReg) + 4);
            // 將現在指令的pc counter更新為下一個指令的pc counter
            // 因此pc'=pc+4
            kernel->machine->WriteRegister(NextPCReg, kernel->machine-
>ReadRegister(PCReg) + 4);
            // 將下一個指令的pc counter更新為下下一個指令的pc counter
            // 因此pc''=pc'+4
            return;
            ASSERTNOTREACHED();
            // debug用
            break;
        }
        break;
    }
}
```

- 1.由於SC_Create與SC_Halt皆為system call，因此前面執行流程如Machine::Run()、Machine::OneInstruction()、Machine::RaiseException()完全相同，因此從ExceptionHandler()接續進行code trace。
- 2.根據ExceptionHandler()的定義，會依序完成以下事情:
    - (1) 讀取register r2的資料(對應system call stub code)，作為type參數。根據system call stub定義，因此type為SC_Create對應的stub code(4)。因此system call SC_Create的type參數是由user program將參數寫入registers，再透過kernel讀取對應registers獲得。(pass parameters in registers)
    - (2) 利用傳入參數which判斷是哪一種Exception，因為指令為system call，因此which為SyscallException。
    - (3) 第一層switch case判斷which會對應到SyscallException。
    - (4) 第二層switch case判斷type會對應到SC_Create。
    - (5) 讀取register r4的資料(Create("file0.test")之中傳入一個參數為arg1，會轉換儲存在register r4，對應filename字串開頭記憶體位置)，作為val參數(對應儲存在phsical memory mainMeory的字串開頭index)。因此system call SC_Create的val參數是由user program將資料寫入mainMemory與對應資料的開頭記憶體位置在register，再透過kernel讀取對應registers獲得mainMemory access data的index。(store the parameters in a table in memory, and the table address is passed as a parameter in a register)
    - (6) 利用val當作index讀取mainMemory資料(檔案名稱字串開頭字元)，再利用filename紀錄字串開頭字元的記憶體位置。
    - (7) 呼叫SysCreate傳入filename進行檔案建立，並將回傳值存入status。status數值表示檔案是否建立成功。
    - (8) 將SysCreate回傳值status數值寫入register r2，作為system call SC_Create的回傳結果。
    - (9) 更新program counter register數值，個別對PrevPCReg、PCReg、NextPCReg數值做+4，代表指向個別對應的下一個指令的記憶體位置。(PC=PC+4)

ksyscall.h SysCreate()

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

- 1.根據SysCreate()的定義，會透過透過kernel的filesSystem呼叫Create(filename)進行system call SC_Create處理。

filesys.h FileSystem::Create() <僅擷取部分程式碼>

```
#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++) OpenFileTable[i] = NULL;
    // 將已開啟檔案清單OpenFileTable的值設定為NULL
  }

  bool Create(char *name) {
    int fileDescriptor = OpenForWrite(name);
    // 利用OpenForWrite(name)傳入name建立檔案
    // 而回傳值利用fileDescriptor變數做為檔案描述子

    if (fileDescriptor == -1) return FALSE;
    // 若fileDescriptor=-1，代表create檔案失敗，則Create()回傳False
    Close(fileDescriptor);
    // 若fileDescriptor!=-1，代表create檔案成功
    // 使用Close(fileDescriptor)關閉對應檔案
    return TRUE;
    // Create()回傳True
  }
  OpenFile *OpenFileTable[20];
  // 建立已開啟檔案清單OpenFileTable，並設定同時能開啟的檔案上限為20個
};
```

- 1.根據filesys.h文件說明，filesys.h定義了Nachos file system的資料結構，以及file system的操作(如create, open, and delete files,stub 介面定義在filesys.h)與已開啟檔案的操作(如read, write,and close,介面定義在openfile.h)。

- 2.由於MP1作業的NachOS kernel沒有進行real version file system實作，因此是採用stub version file system方式進行。stub version僅只是借用native UNIX file system當作NachOS的file system，NachOS本身並沒有實作自己的file system，而是使用c library的定義file operations間接呼叫底層Linux OS的file system協助執行，因此稱為stub version。
- 3.根據Create()的定義，會依序完成以下事情:
    - (1) 利用OpenForWrite()傳入name建立檔案，而回傳值利用fileDescriptor變數做為檔案描述子。OpenForWrite() 實作定義在 sysdep.cc。
    - (2) 若fileDescriptor=-1，代表create檔案失敗，則Create()回傳False。
    - (3) 若fileDescriptor!=-1，代表create檔案成功，則呼叫Close(fileDescriptor)關閉對應檔案，並且Create()回傳True。

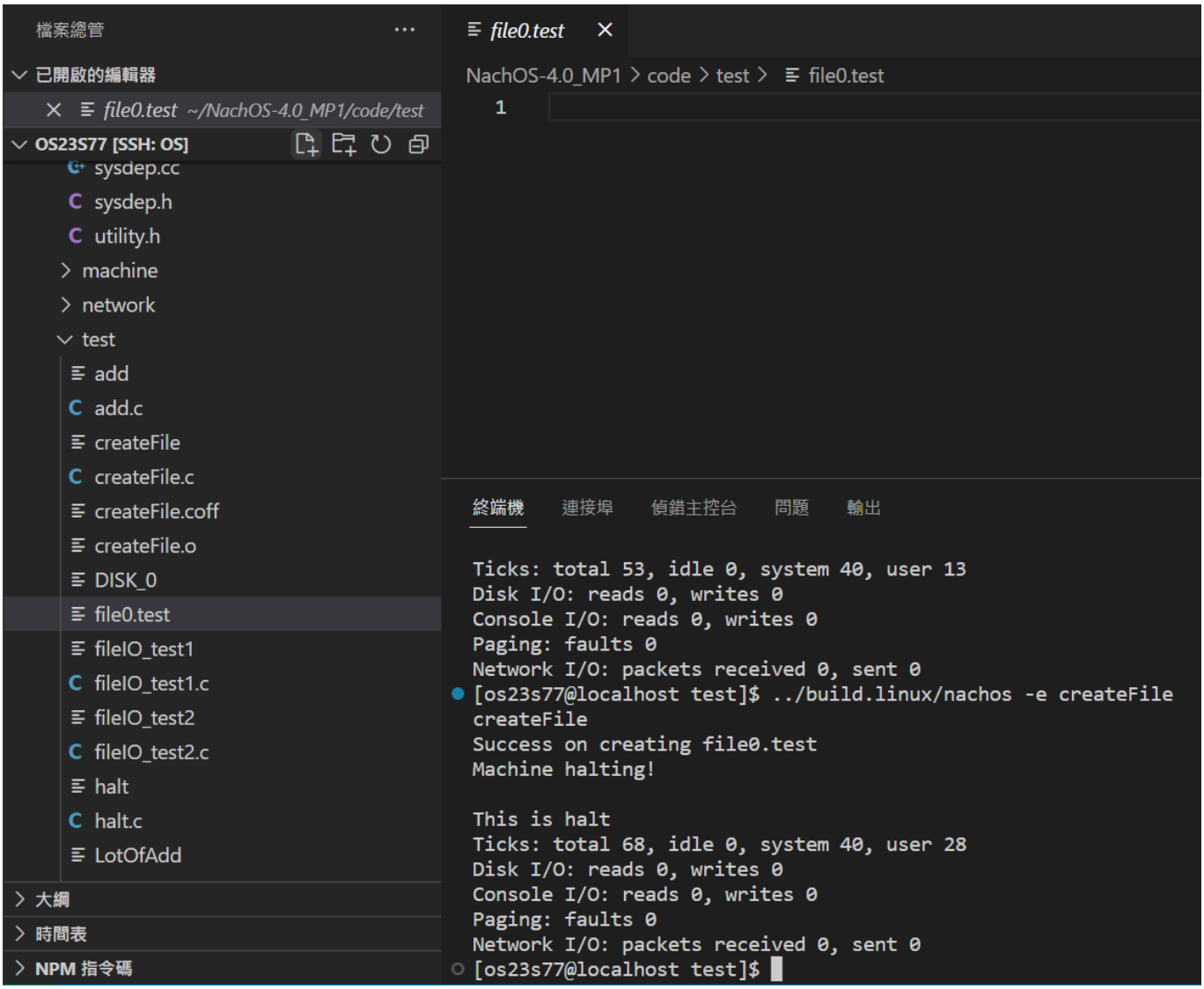
sysdep.cc OpenForWrite() <僅擷取部分程式碼>

```
int
OpenForWrite(char *name)
{
    int fd = open(name, O_RDWR|O_CREAT|O_TRUNC, 0666);
    // 呼叫c library的open()
    // 間接呼叫底層Linux OS的file system協助執行
    // 若檔案名稱不存在，則建立一個新的以此為名稱的檔案(create)
    // 檔案名稱已存在，則清空檔案內容(truncate)

    ASSERT(fd >= 0);
    return fd;
    // 回傳fd(file descriptor)
}
```

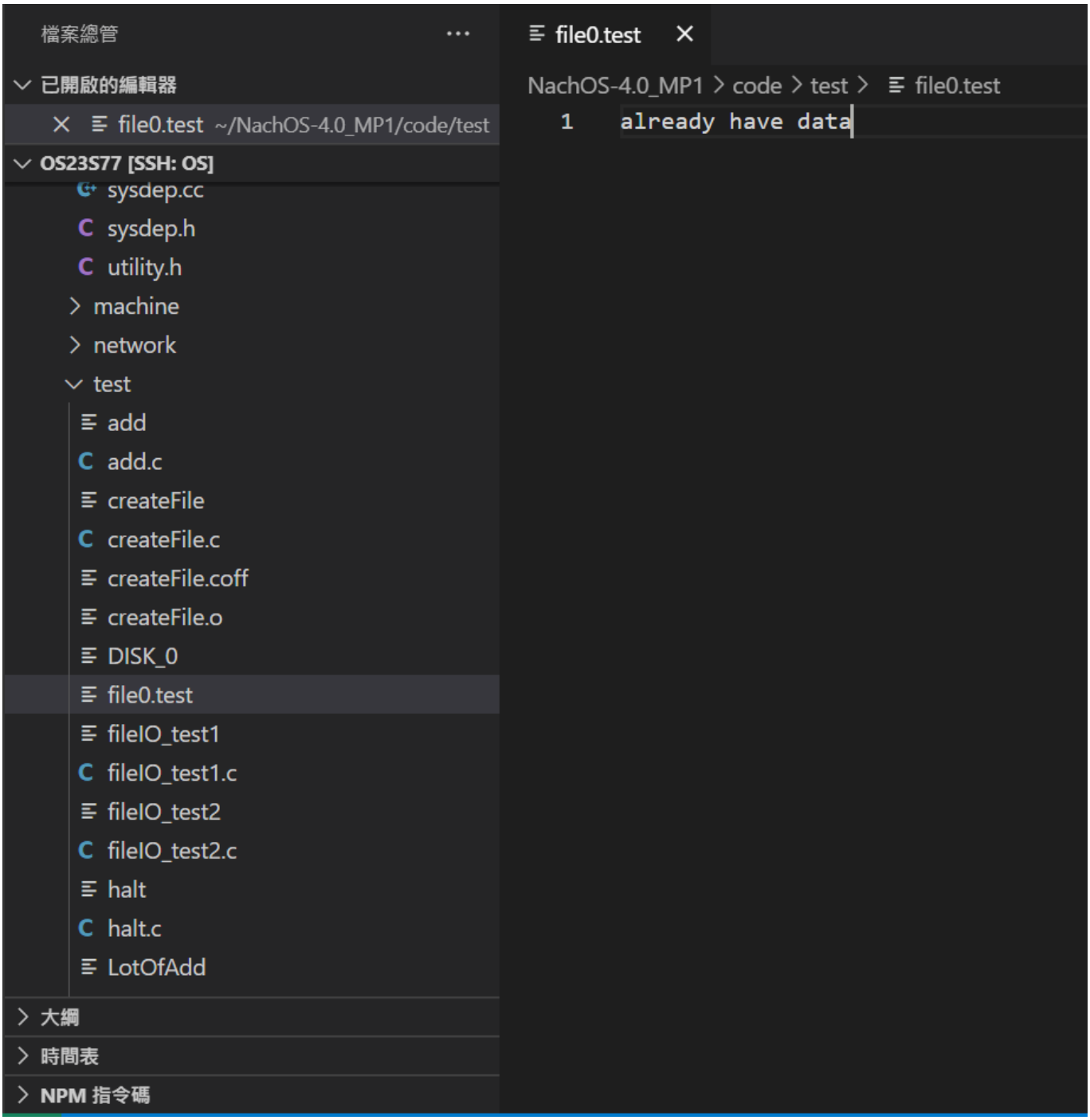
- 1.根據sysdep.cc文件說明，OpenForWrite()可以開啟一個可以寫入的檔案。假如此檔案名稱不存在，則建立一個新的以此為名稱的檔案；假如此檔案名稱已存在，則清空檔案內容，最後回傳fd做為檔案描述子。

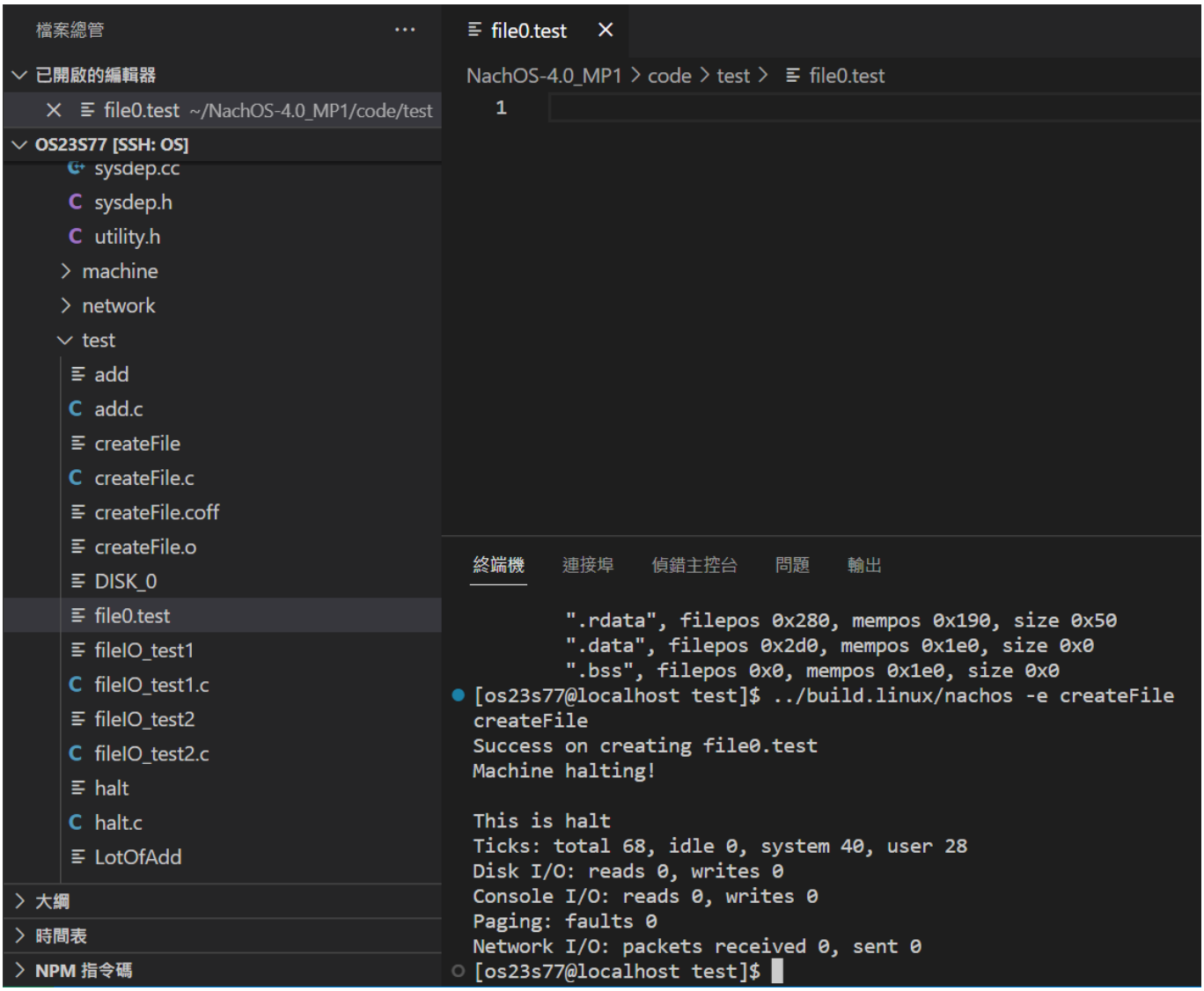
Sample code: createFile.c 執行結果

- Case 1:如果test資料夾內不存在file0.test，則會create一個新的file0.test。



- Case 2:如果test資料夾內已存在file0.test，則會清空file0.test內容。


檔案總管
已開啟的編輯器
file0.test ~/NachOS-4.0_MP1/code/test
OS23S77 [SSH: OS]
sysdep.cc
sysdep.h
utility.h
machine
network
test
add
add.c
createFile
createFile.c
createFile.coff
createFile.o
DISK_0
file0.test
fileIO_test1
fileIO_test1.c
fileIO_test2
fileIO_test2.c
halt
halt.c
LotOfAdd
大綱
時間表
NPM 指令碼
file0.test
NachOS-4.0_MP1 > code > test > file0.test
1 already have data



(3) System Call : SC_PrintInt (Sample code: add.c)

add.c

```
#include "syscall.h"
int
main()
{
  int result;

  result = Add(42, 23);
  PrintInt(result);
  MSG("add~~~~");
  Halt();
  /* not reached */
}
```

- 1.根據add.c文件，此user program呼叫Add()、PrintInt()、MSG()、Halt()四個function，而Add進行加法需要用到寫入以及讀取register數值、PrintInt()與MSG()皆會顯示訊息到command line上需要調用I/O資

源、Halt()關閉kernel會影響OS運作，因此上述四個fuction皆會觸發interrupt，使程式程序trap to kernel，故屬於system call。

- 2.本段trace Code專注在PrintInt()後續的code path。
- 3.根據add.c的定義，會依序完成以下事情:
    - (1) 使用Add()將整數42 & 23相加為整數65，並將65存在result變數。
    - (2) 使用PintInt()將result顯示在command line。
    - (3) 使用MSG()顯示"add~~~~"在command line。
    - (4) 呼叫Halt()，釋放Kernel物件記憶體，停止NachOS系統。

syscall.h <僅擷取部分程式碼>

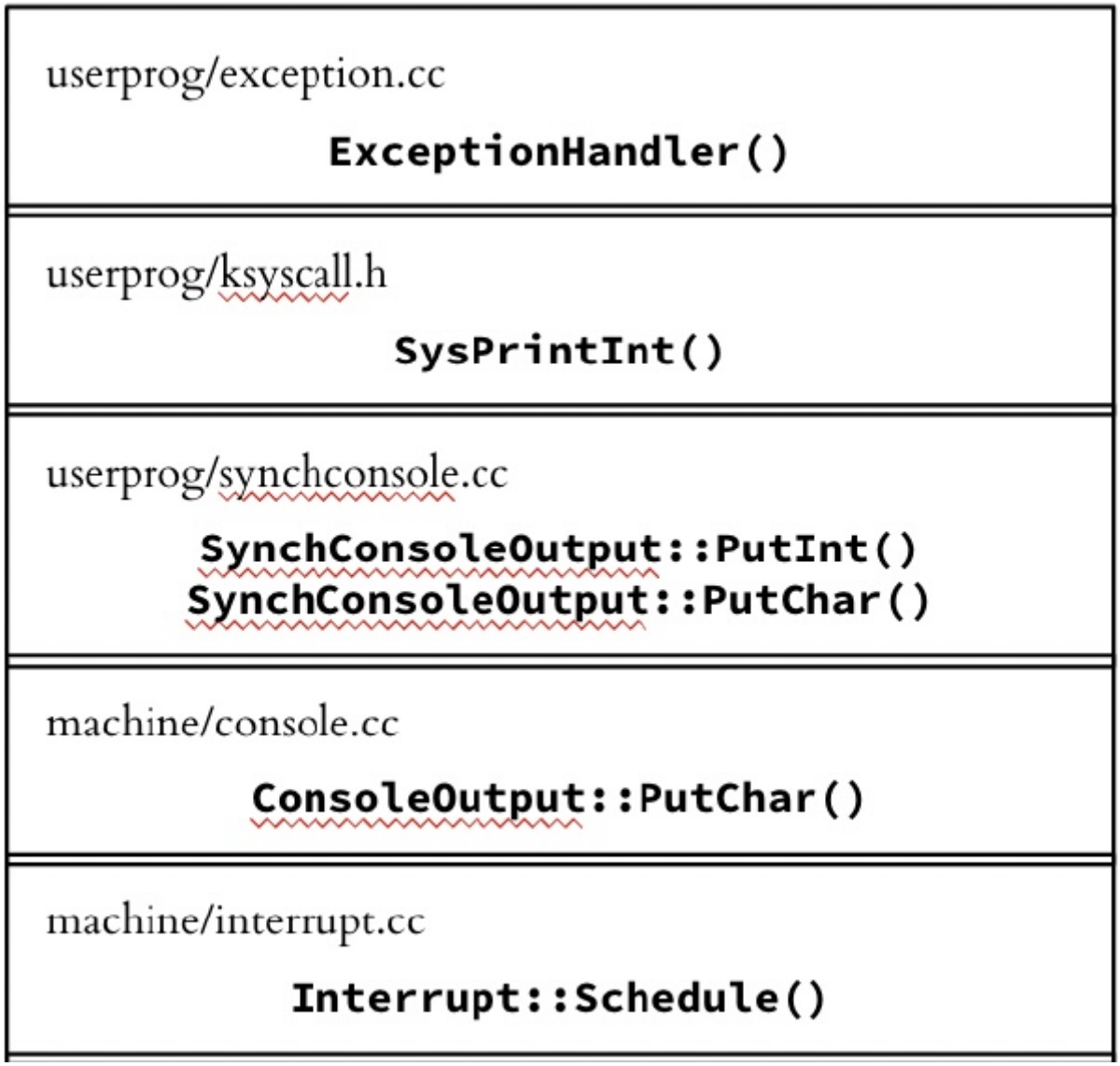
```
#define SC_PrintInt     16
//SC_PrintIn 對應的system call stub number為16

/* Print Integer */
void PrintInt(int number);
```

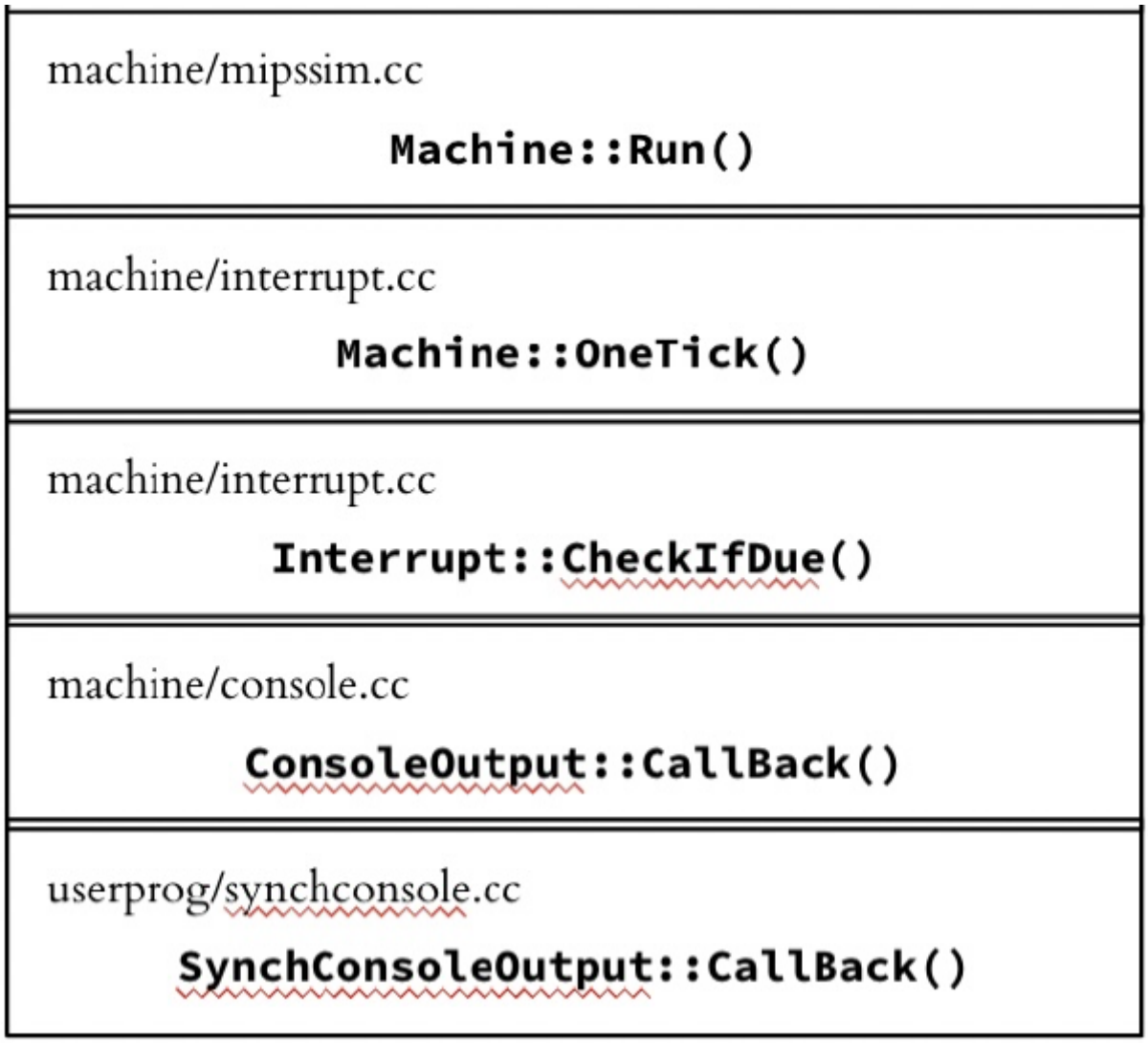
- syscall.h定義了PrintInt()介面與SC_PrintInt的stub code(16)

start.S <僅擷取部分程式碼>

```
    .globl PrintInt
    //宣告“PrintInt”為全域變數
    .ent    PrintInt
    //設定“PrintInt”函數的起始點
PrintInt:
    //“PrintInt”函數調用將從此位置開始
    addiu $2,$0,SC_PrintInt
    //將system call SC_PrintInt對應的stub code(16)存入r2，
    //stub code定義在syscall.h
    syscall
    //呼叫system call，
    //將PrintInt()參數number放入r4(arg1)
    //r4為number數值
    j   $31
    //執行完system call後,返回到RetAddrReg r31(user program)
    .end PrintInt
    //“PrintInt”函數的終止點
```

- 1.根據system call Create函數的stub定義，會依序完成以下事情:
    - (1) 將system call SC_Create對應的stub code(16)存入r2。
    - (2) 呼叫system call。
    - (3) 執行完system call後,返回到RetAddrReg r31(user program address)。


userprog/exception.cc
ExceptionHandler()
userprog/ksyscall.h
SysPrintInt()
userprog/synchconsole.cc
SynchConsoleOutput::PutInt()
SynchConsoleOutput::PutChar()
machine/console.cc
ConsoleOutput::PutChar()
machine/interrupt.cc
Interrupt::Schedule()



exception.cc ExceptionHandler() <僅擷取部分程式碼>

```
void ExceptionHandler(ExceptionType which) {
  char ch;
  int val;
  int type = kernel->machine->ReadRegister(2);
  //從register r2取出system call code存入type
  int status, exit, threadID, programID, fileID, numChar;
  switch (which) {
  // 進行which判斷
    case SyscallException:
    // 當which為SyscallException
      switch (type) {
      // 進行type判斷
        case SC_PrintInt:
        // 當type為SC_PrintInt
          val = kernel->machine->ReadRegister(4);
          // 從register r4取出arg1(number數值)存入val
          SysPrintInt(val);
          // 呼叫SysPrintInt傳入val進行整數顯示在command line上

          // 更新program counter register數值
```

```
            kernel->machine->WriteRegister(PrevPCReg, kernel->machine-
>ReadRegister(PCReg));
            // 將前一個指令的pc counter更新為現在指令的pc counter
            kernel->machine->WriteRegister(PCReg, kernel->machine-
>ReadRegister(PCReg) + 4);
            // 將現在指令的pc counter更新為下一個指令的pc counter
            // 因此pc'=pc+4
            kernel->machine->WriteRegister(NextPCReg, kernel->machine-
>ReadRegister(PCReg) + 4);
            // 將下一個指令的pc counter更新為下下一個指令的pc counter
            // 因此pc''=pc'+4
            return;
            ASSERTNOTREACHED();
            // debug用
            break;
        }
        break;
    }
}
```

- 1.由於SC_PrintInt與SC_Halt皆為system call，因此前面執行流程如Machine::Run()、Machine::OneInstruction()、Machine::RaiseException()完全相同，因此從ExceptionHandler()接續進行code trace。
- 2.根據ExceptionHandler()的定義，會依序完成以下事情:
    - (1) 讀取register r2的資料(對應system call stub code)，作為type參數。根據system call stub定義，因此type為SC_PrintInt對應的stub code(16)。因此system call SC_PrintInt的type參數是由user program將參數寫入registers，再透過kernel讀取對應registers獲得。(pass parameters in registers)
    - (2) 利用傳入參數which判斷是哪一種Exception，因為指令為system call，因此which為SyscallException。
    - (3) 第一層switch case判斷which會對應到SyscallException。
    - (4) 第二層switch case判斷type會對應到SC_PrintInt。
    - (5) 讀取register r4的資料(PrintInt(result)之中傳入一個參數為arg1，會儲存在register r4，對應add加總後的result)，作為val參數。因此system call SC_PrintInt的val參數是由user program將參數寫入registers，再透過kernel讀取對應registers獲得。(pass parameters in registers)
    - (6) 呼叫SysPrintInt傳入val進行整數顯示在command line上
    - (7) 更新program counter register數值，個別對PrevPCReg、PCReg、NextPCReg數值做+4，代表指向個別對應的下一個指令的記憶體位置。(PC=PC+4)

ksyscall.h SysPrintInt() <僅擷取部分程式碼>

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

- 1.根據SysPrintInt()的定義，會透過透過kernel的synchConsoleOut呼叫PutInt(val)進行system call SC_PrintInt處理。

synchconsol.cc SynchConsoleOutput::PutInt() <僅擷取部分程式碼>

```
void
SynchConsoleOutput::PutInt(int value)
{
    char str[15];
    // str陣列用來儲存value數值對應字串
    int idx=0;
    //sprintf(str, "%d\n\0", value);  the true one
    sprintf(str, "%d\n\0", value); //simply for trace code
    // 利用sprintf將value數值用字串方式儲存在str陣列中
    lock->Acquire();
    // 使用lock的Acquire()，會等待lock返回free狀態後，再設定為busy
    // 能鎖定在目前的thread(only one writer at a time)，進行同步化處理
    do{
        consoleOutput->PutChar(str[idx]);
        // 使用consoleOutput的PutChar()傳入字串的對應idx字元
        // 進行後續排程與輸出流程
    idx++;
        // 往字串中下一個字元進行
        waitFor->P();
        // 呼叫waitFor的P()，使後續callback()能通知上一個字元印出的thread已完成
        // 可呼叫下一個字元印出的thread進行處理
        // 達到Output能夠將字串依照順序顯示在command line上的效果
    } while (str[idx] != '\0');
    // while終止條件為遍歷過整個str字串的每個字元一次
    lock->Release();
    // 使用lock的Release()，會把lock設定為free狀態，
    // 接著會喚醒正在等待此lock的thread
}
```

- 1.根據synchconsole.cc文件說明，其定義用來做同步化存取鍵盤與command line顯示硬體設備的例行事項。

- 2.根據synch.h文件說明，可支援thread進行同步化的資料結構有三種:

    - (1) semaphore
    - (2) lock
    - (3) condition variable 可藉由實作出其中一項資料結構，即可以此為基礎完成另外兩項。而在此NachOS版本則是實作出semaphore、並以此為基礎製作了lock與condition variable的操作介面。semaphore class定義了一個非負整數值value以及對應兩個操作operation P()與V()。P()是等待value值>0，接著遞減。V()是遞增，接著喚醒在P()中等待的thread。 lock class則是沿用semaphore class定義的value與function P()與V()，藉此完成lock的function如Acquire()&Release()

- 3.根據SynchConsoleOutput::PutInt()的定義，會依序完成以下事情:

    - (1) 利用sprintf將value數值用字串方式儲存在str陣列中

- (2) 使用lock的Acquire()鎖定在目前的thread，進行同步化處理
- (3) 利用while loop將字元依照index順序傳入consoleOutput->PutChar()。而while loop過程中呼叫waitFor的P()，使後續callback()能通知上一個字元印出的thread已完成。while loop終止條件為整個字串的字元遍歷一遍即可跳出迴圈。
- (4) 使用lock的Release()，解除lock鎖定，接著會喚醒正在等待此lock的thread

synchconsol.cc SynchConsoleOutput::PutChar() <僅擷取部分程式碼>

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

- 根據synchconsole.cc文件說明，SynchConsoleOutput::PutChar()是將一個字元寫入command line顯示，在寫入前需要用lock的Acquire()進行thread鎖定進行同步化，結束後用lock的Release()進行lock解除。而過程之中利用 waitFor->P()為之後的callback()設定一個執行點，以利透過callback()通知kernel已完成目前thread，並呼叫接續的thread進行，達到同步效果。

console.cc ConsoleOutput::PutChar() <僅擷取部分程式碼>

```
void
ConsoleOutput::PutChar(char ch)
{
    ASSERT(putBusy == FALSE);
    // debug用
    WriteFile(writeFileNo, &ch, sizeof(char));
    // 利用WriteFile()傳入寫入檔案id編號、字元對應記憶體位置、字元大小
    putBusy = TRUE;
    // 將putBusy設定為True代表有一個PutChar Progress正在執行
    kernel->interrupt->Schedule(this, ConsoleTime, ConsoleWriteInt);
    // 安排預定地CPU執行排程
}
```

- 根據console.cc文件說明，ConsoleOutput::PutChar()是將一個字元寫入模擬的顯示器上，並且將安排一個interrupt進入排程，以利在未來發生並且返回。

interrupt.cc Interrupt::Schedule() <僅擷取部分程式碼>

```
void Interrupt::Schedule(CallBackObj *toCall, int fromNow, IntType type) {
  int when = kernel->stats->totalTicks + fromNow;
  // when = now + from,計算未來要發生intrrupt的時間點
  PendingInterrupt *toOccur = new PendingInterrupt(toCall, when, type);
```

```
  // PendingInterrupt class pointer toOccur儲存利用PendingInterrupt()產生待發生的
interrupt
  // toCall: interrupt發生要呼叫的物件
  // fromNow: 距離現在多久會發生interrupt
  // type: 哪個硬體設備造成interrupt
  ASSERT(fromNow > 0);
  // debug用
  pending->Insert(toOccur);
  //排入pending的thread駐列內，等待時間到發生interrupt
}
```

- 根據interrupt.cc文件說明，Interrupt::Schedule()是當模擬時間到達now +fromnow時，會使CPU被interrupt打斷，轉而執行設定排程的thread。

Machine::Run() <僅擷取部分程式碼>

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

  kernel->interrupt->setStatus(UserMode);
    // 將mode設定為UserMode
  for (;;) {
    // 無窮迴圈
    OneInstruction(instr);
    // 模擬MIPS CPU逐行執行指令

    kernel->interrupt->OneTick();
    // 確認是否有其他interrupt已經到達預定執行時間，需要優先處理
  }
}
```

- 1.根據mipssim.cc文件說明，Machine::Run()是模擬Nashos執行user-level program，當program啟動時，會由kernel呼叫Machine::Run()執行，且此函數不會返回(CPU不斷讀取指令執行)。
- 2.根據Machine::Run()的定義，會依序完成以下事情:
  - (1) 建立Instruction class pointer instr紀錄instruction的物件記憶體位置。
  - (2) 將Mode設定為UserMode。
  - (3) 進入無窮迴圈，透過OneInstruction(instr)function傳入instruction，模擬MIPS CPU逐行執行指令(包含排程)。
  - (4) 利用OneTick()確認是否有其他interrupt已經到達預定執行時間，需要優先處理。

interrupt.cc Interrupt::OneTick() <僅擷取部分程式碼>

```
void Interrupt::OneTick() {
  MachineStatus oldStatus = status;
  Statistics *stats = kernel->stats;
```

```
  // advance simulated time
  // 根據mode計算對應時間
  if (status == SystemMode) {
    stats->totalTicks += SystemTick;
    stats->systemTicks += SystemTick;
  } else {
    stats->totalTicks += UserTick;
    stats->userTicks += UserTick;
  }

  // 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thread
    status = oldStatus;
  }
}
```

- 根據interrupt.cc文件說明，Interrupt::OneTick()是將計算模擬未來的某個時間點，是否有任何pending interrupt需要被呼叫。從程式碼可得知Interrupt::OneTick()具有模擬時間未來時間的行為，並且根據情況設定interrupt狀態，並釋放目前Thread，然後改成執行下一個Thread。

interrupt.cc Interrupt::CheckIfDue() <僅擷取部分程式碼>

```
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
    next->callOnInterrupt->CallBack();  // call the interrupt handler
    delete next;
  } while (!pending->IsEmpty() && (pending->Front()->when <= stats->totalTicks));
  inHandler = FALSE;
  return TRUE;
}
```

- 根據interrupt.cc文件說明，Interrupt::CheckIfDue()是檢查目前有哪些interrupt被排進排程內準備要發生，若有則優先執行此interrupt。Interrupt::CheckIfDue()若回傳True，代表確實有interrupt handlers被處理。傳入參數advanceClock若為True，代表ready queue內沒有任何interrupt，因此只要繼續推進時間到下一個pending的interrupt準備要發生的時間點，並且呼叫下一個在pending queue的interrupt處理。

console.cc ConsoleOutput::CallBack() <僅擷取部分程式碼>

```
`void
ConsoleInput::CallBack()
{
  char c;
  int readCount;

    ASSERT(incoming == EOF);
    if (!PollFile(readFileNo)) { // nothing to be read
        // schedule the next time to poll for a packet
        kernel->interrupt->Schedule(this, ConsoleTime, ConsoleReadInt);
    } else {
        // otherwise, try to read a character
        readCount = ReadPartial(readFileNo, &c, sizeof(char));
    if (readCount == 0) {
       // this seems to happen at end of file, when the
       // console input is a regular file
       // don't schedule an interrupt, since there will never
       // be any more input
       // just do nothing....
    }
    else {
      // save the character and notify the OS that
      // it is available
      ASSERT(readCount == sizeof(char));
      incoming = c;
```

```
        kernel->stats->numConsoleCharsRead++;
    }
    callWhenAvail->CallBack();
    }
}
```

- 根據console.cc文件說明，ConsoleOutput::CallBack()是模擬CPU machine可從模擬鍵盤讀取一個字元時，會呼叫此函數。首先會先確認此字元是否可被使用，接著當任何人需要此字元時皆會引發callback()進行註冊。

synchconsole.cc SynchConsoleOutput::CallBack()

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

- 根據synchconsole.cc文件說明，SynchConsoleOutput::CallBack()是當可安全傳遞下一個字元到顯示器display時，則Interrupt handler會呼叫此函數通知下一個Interrupt可接續進行。

Sample code: add.c 執行結果

```
add~~~~
Machine halting!

This is halt
Ticks: total 431, idle 300, system 100, user 31
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 3
Paging: faults 0
Network I/O: packets received 0, sent 0
[os23s77@localhost test]$
```

Part 2 Implement four I/O system calls in NachOS

- 根據上面與Trace SC_Create System Call與File System流程，知道需要依序修改以下文件，才能完成4個File System Operation實作:
    - (1) userprog/syscall.h
    - (2) test/start.S
    - (3) userprog/exception.cc
    - (4) userprog/ksyscall.h
    - (5) filesys/filesys.h

- 因為NachOS MP1作業中採用stub file system，因此是藉由呼叫C library function間接透過Linux OS system call協助完成file system 操作，故可調用下面文件的function完成File System Operation實作:
    - lib/sysdep.cc
- 以下按照code執行順序文件進行說明:

syscall.h <僅擷取部分程式碼>

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

- 把syscall.h內的註解拿掉，完成以下四個system call stub code定義:
    - (1) SC_Open = 6
    - (2) SC_Read = 7
    - (3) SC_Write = 8
    - (4) SC_Close = 10

start.S <僅擷取部分程式碼>

```
/* TODO (Open): Add SC_Open system call stubs, you can imitate existing system
calls. */

    .globl Open
    .ent   Open
Open:
    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
Write:
    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
Read:
    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
Close:
    addiu $2,$0,SC_Close
    syscall
    j   $31
    .end Close
```

- 參考其他system call的assmebly語法，依序完成以下system call，讓sytem call stub code寫入register r2, 對應參數能依照順序寫入register r4,r5,r6,r7。
    - (1) open:
        - r2 = SC_Open = 6
        - r4 = file name字串開頭對應mainMemory的index位置
    - (2) Write:
        - r2 = SC_Write = 8
        - r4 = file name字串開頭對應mainMemory的index位置
        - r5 = 寫入字元數量
        - r6 = 對應OpenFile ID號碼
    - (3) Read:
        - r2 = SC_Read = 7
        - r4 = file name字串開頭對應mainMemory的index位置
        - r5 = 寫入字元數量
        - r6 = 對應OpenFile ID號碼
    - (4) Close:
        - r2 = SC_Close = 10
        - r4 = 對應OpenFile ID號碼

exception.cc ExceptionHandler() <僅擷取部分程式碼>

```
void ExceptionHandler(ExceptionType which) {
  char ch;
  int val;
  int type = kernel->machine->ReadRegister(2);
  //從register r2取出system call code存入type
  int status, exit, threadID, programID, fileID, numChar;
```

```
  switch (which) {
    case SyscallException:
      switch (type) {
// 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);
          {
            char *filename = &(kernel->machine->mainMemory[val]);
            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);
          {
            char *buffer = &(kernel->machine->mainMemory[val]);
            int size = kernel->machine->ReadRegister(5);
            int 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);
          {
            char *buffer = &(kernel->machine->mainMemory[val]);
            int size = kernel->machine->ReadRegister(5);
            int 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);
          {
            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;
      }
      break;
  }
}
```

- 類似SC_Create流程，user program傳遞參數給kernel，依照參數的類型，透過register傳遞數值或是對應的mainMemory的index位置，再用區域變數儲存，傳遞參數到對應函數，並透過status回傳執行成功與否。system call執行完畢之後，會更新program counter register數值，讓program counter指向下一個指令。
  - (1) SC_Open:
    - type = r2 = SC_Open (pass parameters in registers)
    - val = r4 = file name字串開頭對應mainMemory的index位置
    - filename = file name字串對應mainMemory的記憶體位置 (store the parameters in a table in memory, and the table address is passed as a parameter in a register)
    - status = SysOpen(filename) = 檔案開啟結果(1:成功,-1失敗) = r2(將結果寫回r2)
  - (2) Write:
    - type = r2 = SC_Write (pass parameters in registers)
    - val = r4 = file name字串開頭對應mainMemory的index位置 (store the parameters in a table in memory, and the table address is passed as a parameter in a register)
    - size = r5 = 寫入字元數量 (pass parameters in registers)
    - id = r6 = 對應OpenFile ID號碼 (pass parameters in registers)
    - status = SysWrite(buffer, size, id) = 檔案寫入結果(1:成功,-1失敗) = r2(將結果寫回r2)
  - (3) Read:
    - type = r2 = SC_Read (pass parameters in registers)

        - val = r4 = file name字串開頭對應mainMemory的index位置 (store the parameters in a table in memory, and the table address is passed as a parameter in a register)
        - size = r5 = 寫入字元數量 (pass parameters in registers)
        - id = r6 = 對應OpenFile ID號碼 (pass parameters in registers)
        - status =SysRead(buffer, size, id) = 檔案讀取結果(1:成功,-1失敗) = r2(將結果寫回r2)
    - (4) Close:
        - type =r2 = SC_Close (pass parameters in registers)
        - id = r4 = 對應OpenFile ID號碼 (pass parameters in registers)
        - status = SysClose(val) = 檔案關閉結果(1:成功,-1失敗) = r2(將結果寫回r2)

ksyscall.h <僅擷取部分程式碼>

```
// TODO (Open): Finish kernel interface for system call (Open).
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)
{
  return kernel->fileSystem->ReadFile(buffer,size,id);
}


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

- 根據ksyscall.h文件定義，此為kernel執行system call的介面，藉由呼叫kernel的fileSystem的對應function，完成open, read, write, close四種需要user program透過kernel代為執行的檔案操作相關system call。

filesys.h <僅擷取部分程式碼>

```
  // 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)
  {
    int fileDescriptor = OpenForReadWrite(name, FALSE);
    // 利用OpenForReadWrite()傳遞name，返回fileDescriptor
    // fileDescriptor >=0代表有成功開啟檔案
    // fileDescriptor = -1代表此檔案名稱不存在
    int full_check = 1;
    // 利用full_check變數檢查OpenFileTable是否填滿
    // full_check = 1預設為填滿
    // full_check = 0代表為未填滿，仍有空位
    int availalbe_index = -1;
    // 利用availalbe_index紀錄OpenFileTable空位的index
    for (int i = 0; i < 20; i++)
    {
      if(OpenFileTable[i]==NULL)
    //掃一輪OpenFileTable，若有發現table值為NULL，代表此位置是空位
      {
        full_check = 0;
        // full_check = 0代表為未填滿，仍有空位
    availalbe_index = i;
    // availalbe_index紀錄OpenFileTable空位的index
        break;
    // 提早跳出迴圈
      }
    }
    if(fileDescriptor == -1 || full_check == 1)
    {
    // 若fileDescriptor == -1代表檔案不存在
    // full_check == 1代表OpenFileTable已填滿
    // 則回傳-1
    return -1;
    }else
    {
        // 若非上述情況，
        // 則在空位處儲存透過fileDescriptor創立一個OpenFile的class pointer
        // 並回傳空位處的index
    OpenFileTable[availalbe_index] = new OpenFile(fileDescriptor);
      return availalbe_index;
    }
  }

  // The WriteFile function is used for kernel write system call
  /* TODO (Write)
    1) If the id is out of range or indicates to a non-exist file, return -1
    2) Otherwise, call OpenFile function to execute write and return the number of
characters.
  */
  int WriteFile(char *buffer, int size, OpenFileId id)
  {
    OpenFile *open_file = OpenFileTable[id];
    // 利用open_file紀錄OpenFileTable[id]的OpenFile class pointer
    if( id >= 20 || id < 0)
    {
    // 假如id超過 OpenFileTable範圍(index:0-19)，則回傳-1
```

```
    return -1;
    }
    if(open_file == NULL)
    {
    // 假如id指向不存在的檔案，則回傳-1
    return -1;
    }
        // 若非上述情況，則呼叫OpenFile的Write function執行檔案寫入
    // 並且回傳實際寫入的字元數量
    return open_file->Write(buffer, size);
  }

  // The ReadFile function is used for kernel read system call
  /* TODO (Read)
    1) If the id is out of range or indicates to a non-exist file, return -1
    2) Otherwise, call OpenFile function to execute read and return the number of
characters.
  */
  int ReadFile(char *buffer, int size, OpenFileId id)
  {
    OpenFile *open_file = OpenFileTable[id];
    // 利用open_file紀錄OpenFileTable[id]的OpenFile class pointer
    if( id >= 20 || id < 0)
    {
    // 假如id超過 OpenFileTable範圍(index:0-19)，則回傳-1
    return -1;
    }
    if(open_file == NULL)
    {
    // 假如id指向不存在的檔案，則回傳-1
    return -1;
    }
        // 若非上述情況，則呼叫OpenFile的Write function執行檔案寫入
    // 並且回傳實際讀取的字元數量
    return open_file->Read(buffer, size);
  }

  // The CloseFile function is used for kernel close system call
  /* TODO (Close)
    1) If the id is out of range or indicates to a non-exist file, return -1
    2) Otherwise, delete the open file and clear its open file table.
  */
  int CloseFile(OpenFileId id)
  {
    OpenFile *open_file = OpenFileTable[id];
    // 利用open_file紀錄OpenFileTable[id]的OpenFile class pointer
    if( id >= 20 || id < 0)
    {
        // 假如id超過 OpenFileTable範圍(index:0-19)，則回傳-1
    return -1;
    }
    if(open_file == NULL)
    {
        // 假如id指向不存在的檔案，則回傳-1
```

```
    return -1;
    }
    // 刪除open_file class pointer
    delete open_file;
    // OpenFileTable[id]設定為NULL，也就是清空對應id的table
    OpenFileTable[id] = NULL;
    // 回傳1
    return 1;
  }
```

- 根據filesys.h文件說明，此文件定義了kernel執行4種file system operation open, read , write, close的實作內容，但其實因為此版本NachOS MP1採用的是stub file system，是透過sysdep.cc定義的funtion，讓NachOS呼叫C libray funtion，而C libray funtion再呼叫Linux OS的file system代為處理。因此並非是透過NachOS的file system處理。
- sysdep.cc相關function擷取如下:

sysdep.cc <僅擷取部分程式碼>

```
//----------------------------------------------------------------------
//  OpenForReadWrite
//  Open a file for reading or writing.
//  Return the file descriptor, or error if it doesn't exist.
//
//  "name" -- file name
//----------------------------------------------------------------------

int
OpenForReadWrite(char *name, bool crashOnError)
{
    int fd = open(name, O_RDWR, 0);

    ASSERT(!crashOnError || fd >= 0);
    return fd;
}

//----------------------------------------------------------------------
//  Read
//  Read characters from an open file.  Abort if read fails.
//----------------------------------------------------------------------

void
Read(int fd, char *buffer, int nBytes)
{
    int retVal = read(fd, buffer, nBytes);
    ASSERT(retVal == nBytes);
}

//----------------------------------------------------------------------
//  WriteFile
//  Write characters to an open file.  Abort if write fails.
```

```
//----------------------------------------------------------------------

void
WriteFile(int fd, char *buffer, int nBytes)
{
    //printf("In sysdep.cc, nBytes: %d\n", nBytes);
    int retVal = write(fd, buffer, nBytes);
    ASSERT(retVal == nBytes);
}
```

Sample code: fileIO_test1.c && fileIO_test2.c 執行結果

- fileIO_test1.c 執行結果

```
終端機  連接埠  偵錯主控台  問題 4  輸出
[os23s77@localhost test]$ ../build.linux/nachos -e fileIO_test1
fileIO_test1
Success on creating file1.test
Machine halting!

This is halt
Ticks: total 954, idle 0, system 130, user 824
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
[os23s77@localhost test]$
```

- fileIO_test2.c 執行結果

```
終端機  連接埠  偵錯主控台  問題 4  輸出
[os23s77@localhost test]$ ../build.linux/nachos -e fileIO_test2
fileIO_test2
Passed! ^_^
Machine halting!

This is halt
Ticks: total 815, idle 0, system 120, user 695
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
[os23s77@localhost test]$
```

Part 3 What difficulties did you encounter when implementing this assignment?

- 本次實作過程中遇到的困難條列如下:
  - (1) 不夠了解NachOS的系統架構，翻查檔案沒有依循架構的概念去尋找，導致花費大量時間再找對應功能的介面與實作放在哪個檔案。若是對於系統架構更為了解，就能更清楚明白哪些功能應該歸屬於那個位置，NachOS的設計必定是遵循著作業系統的主要框架進行的。
  - (2) 不夠了解NachOS的同步問題Synchronization，如非同步I/O處理與callback應用、同步用的資料結構如semaphore與lock的定義與操作。我認為若先了解作業系統的同步章節，應該能對本次作業會有更深刻的了解。