MP1_Report_47

HackMD 好讀版：連結 (https://hackmd.io/@OJo2ruXGShKdpuewtwzZcQ/H1YaWNeZa)

團隊分工表

團隊成員	Trace code	文件撰寫	功能實作
111065542 楊智堯	v	v	v
111065531 郭芳妤	v	v	v

Result

fileIO_test1

```
[os23team47@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
```

fileIO_test2

```
[os23team47@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
```

> -------------------------- Start for the details -------------------------

II-1 (a) Explain the purposes and details of each function call listed in the code path

1. Trace the SC_Halt system call to understand the implementation of a system call.

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

machine/machine.cc
Machine::RaiseException()

userprog/exeception.cc
ExceptionHandler()

userprog/ksyscall.h
SysHalt()

machine/interrupt.cc
Interrupt::Halt()

總結路徑

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

Machine::OneInstruction(Instruction *instr)
{
    // Fetch instruction
    if (!ReadMem(registers[PCReg], 4, &raw))
        return;                 // exception occurred
    instr->value = raw;
    instr->Decode();
    switch (instr->opCode) {
      case OP_SYSCALL:
        RaiseException(SyscallException, 0);
        return;
    }
}

Machine::RaiseException(ExceptionType which, int badVAddr)
{
    kernel->interrupt->setStatus(SystemMode);
    ExceptionHandler(which);
    kernel->interrupt->setStatus(UserMode);
}

void
ExceptionHandler(ExceptionType which)
{
    int type = kernel->machine->ReadRegister(2);
    switch (which) {
    case SyscallException:
        switch(type) {
            case SC_Halt:
                SysHalt();
                cout<<"in exception\n";
                ASSERTNOTREACHED();
            break;
    }
}

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

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

```
}

void
Statistics::Print()
{
cout << "Ticks: total " << totalTicks << ", idle " << idleTicks;
cout << ", system " << systemTicks << ", user " << userTicks <<"\n";
cout << "Disk I/O: reads " << numDiskReads;
cout << ", writes " << numDiskWrites << "\n";
cout << "Console I/O: reads " << numConsoleCharsRead;
cout << ", writes " << numConsoleCharsWritten << "\n";
cout << "Paging: faults " << numPageFaults << "\n";
cout << "Network I/O: packets received " << numPacketsRecvd;
cout << ", sent " << numPacketsSent << "\n";
}
```

以下為各程式細節

Machine::Run()

```
//----------------------------------------------------------------------
// Machine::Run
//      Simulate the execution of a user-level program on Nachos.
//      Called by the kernel when the program starts up; never returns.
//
//      This routine is re-entrant, in that it can be called multiple
//      times concurrently -- one for each thread executing user code.
//----------------------------------------------------------------------
```

目的：模擬 CPU 不停地執行 instruction，直到遇到 Halt 條件停下。 內容主要分成三大部分：

1. 執行迴圈：用無限迴圈 for-loop 不停地執行 instruction。
2. 追蹤和 Debug：追蹤 instruction 執行時間(ticks) 和提供各種 debugging 細節。
3. 狀態調整：將 interrupt 狀態轉成 userMode，提醒機器現在是 user-level 的 instruction 而非 kernel 或 system level。

細節：

1. Allocate 一個已經 decoded instruction 的 memory 空間，供機器執行。

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

2. 如果 debugging mode 設定成 ‘m’，印出正在執行的 thread 跟執行時間。

```
if (debug->IsEnabled('m')) {
    cout << "Starting program in thread: " << kernel->currentThread->getName();
cout << ", at time: " << kernel->stats->totalTicks << "\n";
}
```

3. 將 interrupt 狀態轉成 UserMode，提醒機器現在是 user-level 的 instruction 而非 kernel 或 system level。

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

4. 開始 `for(;;)` 無限執行程式，透過 OneInstruction(instr) 去獲取、解碼，並執行該 instruction， 並且把原本的 ptr 指向同個程式下一個指令。

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

5. OneTick()，用來紀錄系統時間經過一個單位（ticks），並且檢查有沒有要中斷去執行其他程式的請求，有的話會再 call 一次 run 把它做完。沒有的話就會繼續迴圈直到程式執行完。

參照原文:

```
//----------------------------------------------------------------------
// Interrupt::OneTick
//      Advance simulated time and check if there are any pending
//      interrupts to be called.
//
//      Two things can cause OneTick to be called:
//              interrupts are re-enabled
//              a user instruction is executed
//----------------------------------------------------------------------
```

```
DEBUG(dbgTraCode, "In Machine::Run(), into OneTick " << "== Tick " << kernel->stats->totalTicks << " ==");
kernel->interrupt->OneTick();
DEBUG(dbgTraCode, "In Machine::Run(), return from OneTick " << "== Tick " << kernel->stats->totalTicks << " ==");
```

6. 透過兩種方式 `設定 singleStep 為 true 或 設定 runUntilTime 時間` 去暫停程式，讓使用者更好地觀察 OS 運行的流程與狀態。

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

Machine::OneInstruction(Instruction *instr)

目的：主要有三大功能： `獲取(fetch)` 、 `解碼` ，以及 `執行` `指令`

- 從 memory 獲取指令
- 解碼指令以判斷執行哪個 operation
- 執行指令，並更新機器狀態

細節：因程式碼共 583 行，故以功能切分。

1. Instruction Fetch：用 ReadMem 讀取位於 PCReg 位置起算 4 bytes 的指令以 Binary representation 的方式存入 raw 中。

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

2. 計算下一個 PC 位置，執行完指令後更新。

```
int pcAfter = registers[NextPCReg] + 4;
    int sum, diff, tmp, value;
    unsigned int rs, rt, imm;
```

3. 開始執行指令，以 Switch-case 方式判斷 opcode，以下僅以 `OP_ADD` 及 `OP_ADDI` 為例

```c
// Execute the instruction (cf. Kane's book)
switch (instr->opCode) {

  case OP_ADD:
sum = registers[instr->rs] + registers[instr->rt];
if (!((registers[instr->rs] ^ registers[instr->rt]) & SIGN_BIT) &&
    ((registers[instr->rs] ^ sum) & SIGN_BIT)) {
    RaiseException(OverflowException, 0);
    return;
}
registers[instr->rd] = sum;
break;

  case OP_ADDI:
sum = registers[instr->rs] + instr->extra;
if (!((registers[instr->rs] ^ instr->extra) & SIGN_BIT) &&
    ((instr->extra ^ sum) & SIGN_BIT)) {
    RaiseException(OverflowException, 0);
    return;
}
registers[instr->rt] = sum;
break;
```

實際造成 exception 的片段為：

```c
switch (instr->opCode) {
  case OP_SYSCALL:
        RaiseException(SyscallException, 0);
        return;
}
```

4. 用 `DelayedLoad` 執行因 pipeline 或其他原因延後執行的指令。紀錄當前 PC 作為 debug 用途，並更新 PC。

```
// 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;
```

Machine::RaiseException()

目的：

1. Exception handling： 處理在執行指令時遇到的例外狀況，如：invalid memory access, division by zero 或 system call。
2. Trasition to kernel mode：通常發生例外時會交由 OS 協助處理，故需要先將 user mode 轉為 kernel mode 以進行後續動作。

```
//----------------------------------------------------------------------
// Machine::RaiseException
//  Transfer control to the Nachos kernel from user mode, because
//  the user program either invoked a system call, or some exception
//  occured (such as the address translation failed).
//
//  "which" -- the cause of the kernel trap
//  "badVaddr" -- the virtual address causing the trap, if appropriate
//----------------------------------------------------------------------
```

細節：

1. Debeg information:

用 exceptionNames 印出當前發生的 exception event type。

> `DEBUG(dbgMach, "Exception: " << exceptionNames[which]);`

2. 儲存 Faulting address:
   將 virtual address `badVddr` 存入特殊的 register `BadVAddrReg`，以便 exception handler 知道發生問題的地址。

> `registers[BadVAddrReg] = badVAddr;`

3. 完成 Delayed Operation：

在開始處理 exception 之前，確定機器是否有 pending 或 delayed operation 是否皆已完成。

> `DelayedLoad(0, 0);`

4. 切換成 kernel mode：

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

5. 呼叫 Exception Handler：
   正式處理 Exception

> `ExceptionHandler(which);`

6. 回到 Usermode：

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

ExceptionHandler()

目的：針對在 user program 執行時所產生不同的 exceptions 做出對應的行為。其中一個常見的狀況是處理 user program 呼叫的 system call。

細節:

透過 Switch-case 偵測不同的 exception type 做出對應的處理。此處只針對 System call 做出處理，若非 System call 則輸出 `Unexpected user mode exception`

```
switch (which) {
case SyscallException:
switch(type) {…
break;
default:
    cerr << "Unexpected user mode exception " << (int)which << "\n";
    break;
}
```

若是 System call，則以 `switch(type)-case` 判別，並做出對應行為，以 SC-Halt 為例：

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

簡述此處所處理的 System call

- SC_HALT：暫停機器。當 SysHalt() 被呼叫時，機器應當暫停。
- SC_PrintInt：印出存在 machine register 傳給 SysPrintInt(val)的值，再將 program counter 更新到下一個指令。
- SC_MSG：印出 main memory 的 address message string 並暫停機器
- SC_Create：create a file
- SC_Add：從 register fetch 兩個 operands 相加後寫回 register
- SC_Exit：結束現有 Program

SysHalt()

目的：
initiate a complete halt or shutdown of the simulated machine or operating system environment.

細節：

> `kernel->interrupt->Halt();`

Halt()

目的：

1. Shutdown the System： 停止或終止現機器。
2. Print Statistics：在關機之前印出系統的相關數據。

```
//----------------------------------------------------------------------
// Interrupt::Halt
//  Shut down Nachos cleanly, printing out performance statistics.
//----------------------------------------------------------------------
```

細節：

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

1. Print 相關數據

```
//----------------------------------------------------------------------
// Statistics::Print
//  Print performance metrics, when we've finished everything
//  at system shutdown.
//----------------------------------------------------------------------

void
Statistics::Print()
{
    cout << "Ticks: total " << totalTicks << ", idle " << idleTicks;
    cout << ", system " << systemTicks << ", user " << userTicks <<"\n";
    cout << "Disk I/O: reads " << numDiskReads;
    cout << ", writes " << numDiskWrites << "\n";
    cout << "Console I/O: reads " << numConsoleCharsRead;
    cout << ", writes " << numConsoleCharsWritten << "\n";
    cout << "Paging: faults " << numPageFaults << "\n";
    cout << "Network I/O: packets received " << numPacketsRecvd;
    cout << ", sent " << numPacketsSent << "\n";
}
```

2. deletes the main kernel object

2. Trace the SC_Create system call to understand the basic operations and data structure in a file system. (Sample code: createFile.c)

userprog/exeception.cc

ExceptionHandler()

userprog/ksyscall.h

SysCreate()

filesys/filesys.h

FileSystem::Create()

總結路徑

```
void
ExceptionHandler(ExceptionType which)
{
int type = kernel->machine->ReadRegister(2);
switch (which) {
case SyscallException:
switch(type) {
    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(PCRe
    kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) +
    kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCRe
    return;
    ASSERTNOTREACHED();
    break;
}
}

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

bool
FileSystem::Create(char *name, int initialSize)
{
    ...
    return success;
}
```

以下為各程式細節

ExceptionHandler()

目的：針對在 user program 執行時所產生不同的 exceptions 做出對應的行為。其中一個常見的狀況是處理 user program 呼叫的 system call。

細節:

針對 `SC_Create` 介紹，`SC_Create` 是處理新建檔案的 system call。

1. 從 register 讀值

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

2. 得到 filename string 的起始位置

> ```
> char *filename = &(kernel->machine->mainMemory[val]);
> ```

3. 正式呼叫 system call 建立檔案

> ```
> status = SysCreate(filename);
> ```

4. 將結果寫回 register

> Arguments to the system call, when necessary, are passed in MIPS registers r4 through r7 (i.e. the argument registers, a0 ... a3), following the standard C procedure call convention. Function return values, including system call return values, are expected to be in register r2 (v0) on return.

source: A Guide to Nachos 5.0j (https://inst.eecs.berkeley.edu/~cs162/sp07/Nachos/walk/x416.html)

> ```
> kernel->machine->WriteRegister(2, (int) status);
> ```

5. 更新 Program counter 後 return

> ```
> 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;
> ```

SysCreate()

目的：

一個基於模擬 Nachos，處理 user program 建立檔案請求的 interface

細節：

1. Interface with the File System

```
return kernel->fileSystem->Create(filename);
```

2. Return value

- `0: failed`
- `1: Success`

FileSystem::Create()

目的： 透過 Nachos file system 新建檔案

細節：

1. Initialization

定義 Pointers 跟 variables 作為管理 directories、追蹤硬碟 free sectors、handling file headers、以及決定 operation 的 success state。

```
Directory *directory;
PersistentBitmap *freeMap;
FileHeader *hdr;
int sector;
bool success;
```

2. 找到 current directory

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

3. 檢查檔案是否已存在

```
if (directory->Find(name) != -1)
    success = FALSE;
```

4. 如果檔案不存在則需要建立，Allocate Sector for File Header

`PersistentBitmap` 幫忙 track disk 裡 free and occupied sectors

`FindAndSet` 幫忙找到 free sector 並標記成 occupied

> ```
> freeMap = new PersistentBitmap(freeMapFile,NumSectors);
> sector = freeMap->FindAndSet();
> ```

5. 判斷是否有空間建立檔案

> ```
> if (sector == -1)
>   success = FALSE;     // no free block for file header
> else if (!directory->Add(name, sector))
>   success = FALSE;     // no space in directory
>
> else {
>   hdr = new FileHeader;
>   if (!hdr->Allocate(freeMap, initialSize))
>     success = FALSE;   // no space on disk for data
> ```

6. 將更改寫回 disk

- File header 寫到 allocated sector
- directory 更新到 disk
- free space bitmap 更新到 disk

> ```
> hdr->WriteBack(sector);
> directory->WriteBack(directoryFile);
> freeMap->WriteBack(freeMapFile);
> ```

7. Clean up

> ```
> delete hdr;
> delete freeMap;
> delete directory;
> ```

3. Trace the SC_PrintInt system call to understand how NachOS implements asynchronized I/O using CallBack functions and register schedule events. (Sample code: add.c)

```
$ ../build.linux/nachos -d + -e add
```

userprog/exeception.cc
ExceptionHandler()

userprog/ksyscall.h
SysPrintInt()

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

machine/console.cc
ConsoleOutput::PutChar()

machine/interrupt.cc
Interrupt::Schedule()

machine/mipssim.cc
Machine::Run()

machine/interrupt.cc
Machine::OneTick()

machine/interrupt.cc
Interrupt::CheckIfDue()

machine/console.cc
ConsoleOutput::CallBack()

userprog/synchconsole.cc
SynchConsoleOutput::CallBack()

總結路徑

```
void
ExceptionHandler(ExceptionType which)
{
int type = kernel->machine->ReadRegister(2);
switch (which) {
case SyscallException:
switch(type) {
    case SC_PrintInt:
    val=kernel->machine->ReadRegister(4);
    SysPrintInt(val);
    return;
}
}

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

void
SynchConsoleOutput::PutInt(int value)
{
    char str[15];
    int idx=0;
    sprintf(str, "%d\n\0", value);
    lock->Acquire();
    do{
        consoleOutput->PutChar(str[idx]);
        idx++;
        waitFor->P();
    } while (str[idx] != '\0');
    lock->Release();
}

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

void
Interrupt::Schedule(CallBackObj *toCall, int fromNow, IntType type)
{
    int when = kernel->stats->totalTicks + fromNow;
    PendingInterrupt *toOccur = new PendingInterrupt(toCall, when, type);
    ASSERT(fromNow > 0);
    pending->Insert(toOccur);
}

void
Machine::Run()
```

```
{
    kernel->interrupt->OneTick();
}

void
Interrupt::OneTick()
{
  CheckIfDue(FALSE);
}

bool
Interrupt::CheckIfDue(bool advanceClock)
{
  next->callOnInterrupt->CallBack();// call the interrupt handler
}

void
ConsoleOutput::CallBack()
{
    putBusy = FALSE;
    kernel->stats->numConsoleCharsWritten++;
    callWhenDone->CallBack();
}

void
SynchConsoleOutput::CallBack()
{
    waitFor->V();
}
```

以下為各程式細節

ExceptionHandler()

目的: 判別例外處裡

細節:

1. 判別 type 是 SC_PrintInt
2. 將參數 `val` （欲列印的值）從 $4 取出
3. system call `SysPrintInt` 並將 `val` 傳入

```
void
ExceptionHandler(ExceptionType which)
{
int type = kernel->machine->ReadRegister(2);
switch (which) {
case SyscallException:
switch(type) {
    case SC_PrintInt:
    val=kernel->machine->ReadRegister(4);
    SysPrintInt(val);
    return;
}
}
```

SysPrintInt()

目的：作為 user program 跟 NachOS 的 interface，提供的功能為在 sychronized console output 印出 integer value。

細節：

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

SynchConsoleOutput::PutInt()

目的：

1. Synchronized Integer Printing:

在 console 上印出 Integer value，並確保當 multi-thread 要印資料時沒有 concurrency-related issues

細節：

1. String buffer and Formatting:
   預先給定 String buffer 後，將 integer value 轉成 string 並以 string 方式印出。

2. Acquire Lock
   Lock 以避免 race condition

> ```
> lock->Acquire();
> ```

3. Printing Loop
   將字串以一個字元的方式 send to console 等待列印。為避免一個字元尚未結束又傳入下一字元，故使用 `waitFor->P()`

   > ```
   > do{
   >     consoleOutput->PutChar(str[idx]);
   >     idx++;
   >     waitFor->P();
   > } while (str[idx] != '\0');
   > ```

4. Release Lock
   列印完畢，解除 Lock

```
lock->Release();
```

ConsoleOutput::PutChar

目的：

1. Character Output:
   在 console 上印出一個一個字元，
2. Interrupt Scheduling:
   排程一個 Interrupt

細節：

1. 確定是否正在執行其他 write operation

   > ```
   > ASSERT(putBusy == FALSE);
   > ```

2. 印出字元

   > ```
   > WriteFile(writeFileNo, &ch, sizeof(char));
   > ```

3. 設定正在執行 write operation

   > ```
   > putBusy = TRUE;
   > ```

4. 排程 Interrupt

```
kernel->interrupt->Schedule(this, ConsoleTime, ConsoleWriteInt);
```

Interrupt::Schedule

目的：

1. Interrupt Scheduling
   schedule interrupt 並且會在未來的時間點發生
2. Ordered Execution
   將 Interrupt 放進 ordered list based on scheduled times
3. Device Simulation
   模擬 I/O operation, timers interrupt 的行為

細節：

1. 計算 Interrupt 發生時間

```
int when = kernel->stats->totalTicks + fromNow;
```

2. 建立一個新的 Interrupt Object

```
PendingInterrupt *toOccur = new PendingInterrupt(toCall, when, type);
```

3. 確保 Interrupt 發生在未來的時間，並將 interrupt insert to pending list

```
ASSERT(fromNow > 0);
pending->Insert(toOccur);
```

Machine::run()

已 Trace，重點是透過 OneTick() 看是否有待執行的 Interrupt

Interrupt::OneTick()

目的：

1. Time Simulation：
   模擬前進一個模擬時間(one tick)
2. Interrupt Checking：
   檢查在特定時間是否有 interrupt 被觸發
3. Context Switching：
   如果有 interrupt(e.g. from timer)，
   current thread yield its exection

細節：

1. 前進一個模擬時間

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

2. 檢查 Pending Interrupt
   為確保 atomic 執行 operation，故 disabled interrupt 再檢查 pending interrupt，完成後再 enable interrupt。

   > ```
   > ChangeLevel(IntOn, IntOff);
   > CheckIfDue(FALSE);
   > ChangeLevel(IntOff, IntOn);
   > ```

3. Handle Context Switching:
   如果 `yieldOnReturn` 是 `true` ，代表有 context switch 的 request(像是 timer interrupt handler)，執行完再恢復成 oldStatus。

   > ```
   > if (yieldOnReturn) {
   >     yieldOnReturn = FALSE;
   >     status = SystemMode;
   >     kernel->currentThread->Yield();
   >     status = oldStatus;
   > }
   > ```

Interrupt::CheckIfDue()

目的：

1. Interrupt Checking：
   確定 pending interrupt 是否已到期(due)需要執行
2. Interrupt Handling
   如果有 due interrupt，呼叫對應的 interrupt handler
3. Time Advancement (optional)
   如果 ready queue 沒有 thread ，則執行下一個 scheduled interrupt。

細節：

1. 確定是否 disabled interrupt 及是否有 pending interrupts

```
ASSERT(level == IntOff);

if (pending->IsEmpty()) {
    return FALSE;
}
```

2. 若尚未到觸發時間，return `false` 或如果 `advanceClock` 是 `true` 的話代表 queue 內無其他 thread，則將時間設成下一個 interrupt 的時間。

```
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.
}
```

3. Invoking Due Interrupts

處理此時到期的全部 interrupts，先將 interrupt 從 `pending` list 移除，並呼叫對應的 handler。

```
do {
    next = pending->RemoveFront();
    next->callOnInterrupt->CallBack();
    delete next;
} while (!pending->IsEmpty() && (pending->Front()->when <= stats->totalTicks));
```

ConsoleOutput::CallBack()

目的：

1. Notification of Write Completion
   當之前 requested 的 character output 完成的時候會呼叫此 Function，告訴 console ready 好收道下一個字元。

2. Housekeeping and Triggers
   更新 kernel 內 some statistic

細節：

1. Update Busy Status:
   將 console output 是否為 busy processing character write 的變數設為 false。

   > putBusy = FALSE;

2. Update Statistics:
   將輸出字數記錄到 kernel。

   > kernel->stats->numConsoleCharsWritten++;

3. Trigger Further Actions:
   如果有在 console write operation 後需要執行的指令會在此被呼叫。

   > callWhenDone->CallBack();

SynchConsoleOutput::CallBack()

目的： 允許下一個 console write operation 通過

細節：

> waitFor->V();

4. Trace the Makefile in code/test/Makefile to understand how test files are compiled.

目的：
Compiling and linking a set of test programs.

細節：

1. Includes makefile 需要的 variables

```
include Makefile.dep
```

2. 定義 Complier、Assembler 和 linker

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

3. 選擇預編譯的資料夾及 tag

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

4. 定義需要 compile 的program list

```
ifeq ($(hosttype),unknown)
PROGRAMS = unknownhost
else
PROGRAMS = add halt createFile fileIO_test1 fileIO_test2 LotOfAdd
endif
```

5. Compile and Link 各預執行程式
   當我們下 `make` 指令會找到從 `All` 這個 target 開始執行，若無則從第一個 target 開始，以 add 為例，其餘程式同理：
    1. Compile `add.c` 變成 `add.o`
    2. link `add.o` 跟 `start.o` 並輸出成 `add.coff`
    3. 把 `add.coff` 轉成 Nachos 執行檔 `add`

```
all: $(PROGRAMS)

add.o: add.c
	$(CC) $(CFLAGS) -c add.c
add: add.o start.o
	$(LD) $(LDFLAGS) start.o add.o -o add.coff
	$(COFF2NOFF) add.coff add
```

6. 刪除檔案的指令

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

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

II-1 (b) Explain how the arguments of system calls are passed from user program to kernel

程式碼執行部分皆已逐步分析，此處僅講解 system call 如何傳遞參數

1. SC_Halt system call

1. 在 `code/test/start.S` 檔案，把 system call type `SC_Halt` 放入 `$2` ，接下來用在 ExceptionHandler() 的參數 `type` ，作為判斷 exception 型態用。參數則透過 `syscall` 自動存入暫存器。

```
	.globl  Halt  # make it global visible
	.ent     Halt # 開始執行
Halt:
	addiu $2,$0,SC_Halt # 將 system call 存在 r2
	syscall  #system call 參數會自動存在 r4,r5,r6,r7
	j       $31 # return from `Halt`
	.end  Halt # end of a func.
```

2. 執行 Machine::Run()
3. 執行 Machine::OneInstruction()，這邊把 Exception type 傳進去參數 `which` ，這裡即 `SyscallException` 。 `RaiseException()` 再把 `which` 傳入 `ExceptionHandler()`

```
Machine::OneInstruction()
case OP_SYSCALL:
        RaiseException(SyscallException, 0);
        return;

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

4. `ExceptionHandler()` 會看 `$2` 是哪種 exception 並存入 `type` ，此處是 `SC_Halt`

```
void
ExceptionHandler(ExceptionType which)
{
    int type = kernel->machine->ReadRegister(2);
}
```

5. 直接執行 `SysHalt()`

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

6. 後續則可回溯 II-1-a-(1) 查看剩下的流程。

2. SC_Create system call

1. 在 `code/test/start.S` 檔案，把 system call type `SC_Create` 放入 `$2` ，接下來用在 ExceptionHandler() 的參數 `type` ，作為判斷 exception 型態用。參數則透過 `syscall` 自動存入暫存器。

```
        .globl  Create  # make it global visible
        .ent     Create # 開始執行
Create:
        addiu $2,$0,SC_Create # 將 system call 存在 r2
        syscall  #system call 參數會自動存在 r4,r5,r6,r7
        j       $31 # return from `Create`
        .end  Create # end of a func.
```

2. 執行 Machine::Run()
3. 執行 Machine::OneInstruction()，這邊把 Exception type 傳進去參數 `which` ，這裡即 `SyscallException` 。 `RaiseException()` 再把 `which` 傳入 `ExceptionHandler()`

```
Machine::OneInstruction()
case OP_SYSCALL:
        RaiseException(SyscallException, 0);
        return;

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

4. `ExceptionHandler()` 會看 `$2` 是哪種 exception 並存入 `type` ，此處是 `SC_Create` （MIPS 在執行 syscall 前放進去，再看一次 asm code）

```
Create:
        addiu $2,$0,SC_Create
        syscall  #system call 參數會自動存在 r4,r5,r6,r7
        j       $31 # return from `Create`
```

```
void
ExceptionHandler(ExceptionType which)
{
    int type = kernel->machine->ReadRegister(2);
}
```

5. 從第四個 register 拿到參數 `val` ，找到字串位置開頭 `filename` 後丟入 `SC_Create()`

```
void
ExceptionHandler(ExceptionType which)
{
    int type = kernel->machine->ReadRegister(2);
    switch (which) {
    case SyscallException:
            switch(type) {
             case SC_Create:
                val=kernel->machine->ReadRegister(4);
                char *filename = &(kernel->machine->mainMemory[val]);
                status = SysCreate(filename);
                kernel->machine->WriteRegister(2, (int) status);
        }
    }
}
```

6. 後續則可回溯 II-1-a-(2) 查看剩下的流程。

3. SC_PrintInt system call

1. 在 `code/test/start.S` 檔案，把 system call type `SC_PrintInt` 放入 `$2` ，接下來用在 ExceptionHandler() 的參數 `type` ，作為判斷 exception 型態用。。參數則透過 `syscall` 自動存入暫存器。

```
        .globl  PrintInt  # make it global visible
        .ent     PrintInt # 開始執行
PrintInt:
        addiu $2,$0,SC_PrintInt # 將 system call 存在 r2
        syscall  #system call 參數會自動存在 r4,r5,r6,r7
        j       $31 # return from `PrintInt`
        .end  PrintInt # end of a func.
```

2. 執行 Machine::Run()
3. 執行 Machine::OneInstruction()，這邊把 Exception type 傳進去參數 `which` ，這裡即 `SyscallException` 。 `RaiseException()` 再把 `which` 傳入 `ExceptionHandler()`

```
Machine::OneInstruction()
case OP_SYSCALL:
       RaiseException(SyscallException, 0);
       return;

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

4. `ExceptionHandler()` 會看 `$2` 是哪種 exception 並存入 `type` ，此處是 `SC_PrintInt`

```
void
ExceptionHandler(ExceptionType which)
{
    int type = kernel->machine->ReadRegister(2);
}
```

5. 從第四個 register 拿到參數 `val` 並丟入 `SysPrintInt()`

```
void
ExceptionHandler(ExceptionType which)
{
    int type = kernel->machine->ReadRegister(2);
    switch (which) {
    case SyscallException:
           switch(type) {
            case SC_PrintInt:
               val=kernel->machine->ReadRegister(4);
               SysPrintInt(val);
        }
    }
}
```

6. 後續則可回溯 II-1-a-(3) 查看剩下的流程。

II-2

跟著傳參數的起源 II-1-b 流程。

1. 在 `code/test/start.S` 檔案，把 system call type 放入 `$2` ，接下來用在 ExceptionHandler() 的參數 `type` ，作為判斷 exception 型態用。

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

        .globl Write
        .ent    Write
Write:
        addiu $2, $0, SC_Write
        syscall
        j       $31
        .end Write

        .globl Read
        .ent    Read
Read:
        addiu $2, $0, SC_Read
        syscall
        j       $31
        .end Read

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

2. 後續辨別 system call 是透過參數 `tpye` ，可知 `type` 會抓到 $2 的值，Trace 過後發現此值定義在 `userprog/syscall.h`

線索： `start.s`： 最上面有 `#include "syscall.h"`

可看到 `userprog/syscall.h` 已定義好 4 個 syscall type num，取消註解即可。

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

開始進 `userprog/exception.cc` 更改 `ExceptionHandler()` ，模仿 `SC_Halt` 新增 `(a) SC_Open` 、 `(b) SC_Write` 、 `(c) SC_Read` 以及 `(d) SC_Close` 的 case。

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

開始之前線索：在 `test/start.S` 可以看到此描述，已知參數從 r4 開始存放，故有參數需使用則從 r4 起開始取用。

> /* --
>
> - 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.
> - -- */

(a). OpenFileId Open(char *name);

Open a file with the name, and return its corresponding OpenFileId.
Return -1 if fail to open the file.

0. 觀察

最重要 `case(SC_Open)` 裡面的任務是呼叫 `SysOpen` System call，結果發先 SysOpen 已經實現，故只需要在 case 中正確呼叫 `SysOpen` 即可。

In `userprog/ksyscall.h` 的 `SysOpen` 可以觀察到需要一個參數，故從在 case 內需從 `r4` 獲得此參數傳過來 `SysOpen` 。

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

1. 實作 `case(SC_Open)` ，可參考 `SC_Create`

```
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(PCRe
    kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) +
    kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCRe
    return;
    ASSERTNOTREACHED();
    break;
```

2. 進去看 `SysOpen()` ，發現 `OpenAfile` 需要實作，從其它如 `SysCreate()` 找到實現邏輯的位置在 `code/filesys/filesys.h`

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

3. 透過 `Open` 實作 `OpenAFile`

參考 Linux `Open` 流程

> 當程式向核心發起system call open()，核心將會
>
> 1. 允許程序請求
> 2. 建立一個 entry 插入到 file table，並返回 file descriptor
> 3. 程式把 fd 插入 fds。
>   來源：Link (https://wiyi.org/linux-file-descriptor.html)

看到 `filesys.h` 原先 `Open` 的實作僅有處理返回 file descripter。

```
OpenFile* Open(char *name) {
    int fileDescriptor = OpenForReadWrite(name, FALSE);
    if (fileDescriptor == -1) return NULL;
    return new OpenFile(fileDescriptor);
}
```

故我們 `OpenAFile` 實作插入 entry 的部分，實作如下：

```
OpenFileId OpenAFile(char *name) {
    OpenFile* fd = Open(name);
    if(fd == NULL) return -1;

    for(int i = 0; i < 20; i++) {
        if(OpenFileTable[i] == NULL) {
            OpenFileTable[i] = fd;
            return i;
        }
    }
    return -1;
}
```

(b). int Write(char *buffer, int size, OpenFileId id);

Write “size” characters from the buffer into the file, and return the number of characters actually written to the file.
Return -1, if fail to write the file.

0. 觀察

最重要 case(SC_Write) 裡面的任務是呼叫 SysWrite System call，觀察到有三個參數需要使用，個別為 `fileBuffer`、`fileSize`，`fileID`。

1. 實作 `case(SC_Write)`，可參考 `SC_Open`

```
case SC_Write:
val = kernel->machine->ReadRegister(4);
{
    char* fileBuffer = &(kernel->machine->mainMemory[val]);
    int fileSize = kernel->machine->ReadRegister(5);
    int fileID = kernel->machine->ReadRegister(6);
    status = SysWrite(fileBuffer, fileSize, (OpenFileId) fileID);
    kernel->machine->WriteRegister(2, (int) status);
}
kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) + 4);
kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg)+4
return;
ASSERTNOTREACHED();
break;
```

2. 在 `SysWrite` 引導到 `WriteFile`

```
int SysWrite(char *fileBuffer, int fileSize, OpenFileId fileID)
{
    return kernel->fileSystem->WriteFile(fileBuffer, fileSize, fileID);
}
```

3. 在 `filesys/filesys.h` 實作 `WriteFile`

觀察到 `OpenFile* OpenFileTable` 知道 `OpenFileTable` 已實現 `Write` 功能，程式如下，此處會回傳寫入字數。

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

故 `WriteFile` 只需要判別例外條件，並將對應 ID 的 OpenFileTable 呼叫 `Write` function 即可， `WriteFile` 實作如下

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

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

©. int Read(char *buffer, int size, OpenFileId id);

Read “size” characters from the file to the buffer, and return the number of characters actually read from the file.
Return -1, if fail to read the file

1. In `/userprog/exception.cc()` 實作 `SC_Read`

```
case SC_Read:
val = kernel->machine->ReadRegister(4);
{
    char* fileBuffer = &(kernel->machine->mainMemory[val]);
    int fileSize = kernel->machine->ReadRegister(5);
    int fileID = kernel->machine->ReadRegister(6);
    status = SysRead(fileBuffer, fileSize, (OpenFileId) fileID);
    kernel->machine->WriteRegister(2, (int) status);
}
kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) + 4);
kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg)+4
return;
ASSERTNOTREACHED();
break;
```

2. In `userprog/ksyscall.h` 實作 `SysRead`

```
int SysRead(char *fileBuffer, int fileSize, OpenFileId fileID)
{
    return kernel->fileSystem->ReadFile(fileBuffer, fileSize, fileID);
}
```

3. In `filesys/filesys.h` 實作 `ReadFile`

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

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

(d). int Close(OpenFileId id);

Close the file with id.
Return 1 if successfully close the file. Otherwise, return -1.
Need to delete the OpenFile after you close the file
(Can't only set the table content to NULL)

1. In `/userprog/exception.cc()` 實作 `SC_Close`

```
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;
```

2. In `userprog/ksyscall.h` 實作 `SysClose`

```
int SysClose(OpenFileId fileID)
{
    return kernel->fileSystem->CloseFile(fileID);
}
```

3. In `filesys/filesys.h` 實作 `CloseFile`

```
int CloseFile(OpenFileId id){
    if(id < 0 || id >= 20) return -1;
    if(OpenFileTable[id] == NULL) return -1;

    delete OpenFileTable[id];
    OpenFileTable[id] = NULL;

    return 1;
}
```

Result

1. 在 `code/build.linux` 底下執行 `make nachos`
2. 在 `code/test` 底下執行 `make clear` 後執行 `make`
3. 在 `code/test` 底下執行 `../build.linux/nachos -e fileIO_test1`
4. 在 `code/test` 底下執行 `../build.linux/nachos -e fileIO_test2`

fileIO_test1

```
[os23team47@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
```

fileIO_test2

```
[os23team47@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
```

III. Difficulties encounter when implementing this assignment

智堯：

作業大方向不會太困難，主要卡在以下幾個點

- Code 散佈在各資料夾，追蹤較不方便（特感謝 IDE 的追蹤功能）
- header 檔案層層引用，某些特別定義的型別不知道能不能沿用
  如 `typedef OpenFileID int` 這類型的變數可用的 scope 需要層層遞進追蹤
- Makefile 觀念不熟悉
  一開始只有對 `test` 底下做 make，沒有注意到 nachos 也需要做 `make` 的動作，花了些許時間再次 Trace system call 結果發現定義錯問題。

但好在真正實作功能的地方幾乎都已完成，我們只需要實作呼叫 system call 的 API，對第一次上手 Nachos 來說是個很好的入口。

芳妤：

在這次的實作中，我遇到了以下兩個困難：

1. 程式碼和資料結構分散問題

最大的挑戰是程式碼與資料結構分散於不同的檔案中。除了作業系統的本身程式碼外，還有用於模擬硬體的程式碼。儘管已經追踪了整個程式碼，但在實際執行時，我經常需要同時開啟多個檔案頁面進行比對。幸好有 debug message，它讓我可以清楚地看到錯誤的地方和需要修正的檔案。

2. 對硬體的不熟悉

我在開始時並不太了解「軟體模擬硬體」的具體概念，是在實作完成後才有比較清晰的「原來這份 code 是在模擬哪一個硬體行為」。

最後想特別提及一點：在與組員交換意見時，我發現自己的方法雖然能夠通過兩個 testcases，但我忽略了必須 maintain filetable 的部分。僅僅因為測試資料的設計，使得使用 Linux 原生的 fileDescriptorID 作為 table 中的 id 並不會出錯。這個錯誤是在互相檢查程式碼後才發現的，很慶幸還好最後有發現XD

整體來說自己 implement 一次 system call 確實有對於這個階段內 OS 的運作有了更具體的認識！