OS MP1 - HackMD

OS MP1 Report

Team member:

110062204 呂宜嫻: SC_Create report，以及 function Write、Read的撰寫
110062239 侯茹文: SC_Halt、SC_Create report，以及 function OpenFileId、Close的撰寫

(a)SC_Halt

Machine::Run

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

    kernel->interrupt->setStatus(UserMode); //將現在的status設為User mode，需要
system call時才會設為kernal mode

    for (;;) {
        OneInstruction(instr); //進入到OneInstruction() 進行指令的實作

        kernel->interrupt->OneTick(); //模擬時間的前進

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

}
```

> 主要目的為模擬電腦運作並逐行執行指令的行為，並且在進入OneInstruction()後才有指令的獲取及實作。

Machine::OneInstruction()

```
Machine::OneInstruction(Instruction *instr)
{
    if (!ReadMem(registers[PCReg], 4, &raw)) //獲取下一個指令，將其存在raw後，進行
解碼的動作
        return;
    instr->value = raw;
    instr->Decode();

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

```
        ...
        ASSERT(instr->opCode <= MaxOpcode); //確保code在有效的opcode範圍，若沒有
就會終止code
        ...
    }

    int pcAfter = registers[NextPCReg] + 4; //下一行指令位置
    ...

    switch (instr->opCode) { //以下處理opcode，也就是指令會需要對應到的實際動作
        ...
    }

    DelayedLoad(nextLoadReg, nextLoadValue);

    registers[PrevPCReg] = registers[PCReg]; //進行指令順序的設定

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

獲取到現在需要執行的instruction後存入raw，並且在Decode()進行解碼，接下來根據其opcode執行相應的動作。

其中有ASSERT()，後面也會出現；它用來避免一些錯誤發生、又或是到達不應該到達的地方時，會終止程式的運作。

而處理完畢之後，會將pc數進行調整，以備執行下一個指令時使用。

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

而值得一提的是，若有system call或是其他異常現象會使用RaiseException()進行system call，交由os處理這些特殊狀況。

Machine::RaiseException()

```
Machine::RaiseException(ExceptionType which, int badVAddr)
{
    registers[BadVAddrReg] = badVAddr; //紀錄現在有問題的address
    DelayedLoad(0, 0); //(0, 0)取消現在任何再加載的動作 怕會有數據不一的情況發生
    kernel->interrupt->setStatus(SystemMode); //轉為kernal mode
    ExceptionHandler(which);  //處理問題ing
    kernel->interrupt->setStatus(UserMode); //轉回user mode
}
```

因為發生了異常，所以將控制權由user轉為kernal的過程，並要交由ExceptionHandler()去處理狀

況，而在處理之前為了防止數據的混亂以及其他可能的不穩定狀況。

在異常被解決後，又會轉回user mode。

ExceptionHandler()

```
ExceptionHandler(ExceptionType which) //which: 哪種exception
{
    int type = kernel->machine->ReadRegister(2); //讀取其中的type
    switch (which) {
    case SyscallException:
    switch(type) {
            case SC_Halt:
                SysHalt();
                ASSERTNOTREACHED(); //如果halt(終止了)不可能執行到這一行
            break;
           ...
        }
        break;
    }
    ASSERTNOTREACHED();
}
```

這部分在處理各種exception，而我們要關注的SC_Halt也是其中一種，這裡call了SysHalt()繼續處理問題。

SysHalt()

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

跳到位於interrupt.cc的Halt()，進行終止程序的動作。

Interrupt::Halt()

```
Interrupt::Halt()
{
    cout << "Machine halting!\n\n";
    cout << "This is halt\n";
    kernel->stats->Print();
    delete kernel; //把整個kernal刪除
}
```

刪除了kernal，也代表它將整個程式停止了，關閉了NachOS程序。

(b)SC_Creat

ExceptionHandler()

```
case SC_Create:
    val = kernel->machine->ReadRegister(4); //讀取數值(檔案名稱位址)
        {
        char *filename = &(kernel->machine->mainMemory[val]); //存取要生成的檔案名稱
        status = SysCreate(filename); //創建文件 並回傳是否成功
        kernel->machine->WriteRegister(2, (int) status); //將現在的狀態寫入reg2
        }
        kernel->machine->WriteRegister(PrevPCReg,
kernel->machine->ReadRegister(PCReg)); //增加pc值 才能繼續往下讀取指令
        kernel->machine->WriteRegister(PCReg,
kernel->machine->ReadRegister(PCReg) + 4);
        kernel->machine->WriteRegister(NextPCReg,
kernel->machine->ReadRegister(PCReg)+4);
        return;
        ASSERTNOTREACHED();
        break;
```

首先讀取檔案名稱的位址，再去那裏將file name讀取出來；傳入SysCreate()後，會回傳是否已經成功創建了，並記錄到register2 中。

SysCreate()

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

跳到位於ksyscall.h的Create()，並回傳是否成功創建。

FileSystem::Create()

```
bool Create(char *name) {
    int fileDescriptor = OpenForWrite(name); //創建一個資料夾

        if (fileDescriptor == -1) return FALSE; //失敗回傳false
        Close(fileDescriptor); //創建了 關掉資料夾 因為沒有要編輯
        return TRUE;
    }
```

因為這次使用的是stub file system，因此我們只需要關注在stub file的部分。

先用OpenForWrite()開啟一個資料夾，若是沒有已存在的資料夾就會創建一個空的資料夾，而這個動作失敗會回傳false。

而因為我們的指令只有創建資料夾，所以創完之後把資料夾關掉即可。

(c)SC_PrintInt

ExceptionHandler()

```
case SC_PrintInt:
    DEBUG(dbgSys, "Print Int\n");
    val=kernel->machine->ReadRegister(4); // 讀取參數
    DEBUG(dbgTraCode, "In ExceptionHandler(), into SysPrintInt, " <<
kernel->stats->totalTicks);
    SysPrintInt(val);   // 呼叫 System call
    DEBUG(dbgTraCode, "In ExceptionHandler(), return from SysPrintInt, " <<
kernel->stats->totalTicks);
    // Set Program Counter
    kernel->machine->WriteRegister(PrevPCReg,
kernel->machine->ReadRegister(PCReg));
    kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg)
+ 4);
    kernel->machine->WriteRegister(NextPCReg,
kernel->machine->ReadRegister(PCReg)+4);
    return;
    ASSERTNOTREACHED();
    break;
```

處理 PrintInt System Call 的 Exception。
自 r4 讀取 System call 的參數 (user 在呼叫 system call 時設定的)，並呼叫 SysPrintInt(val) 執行 PrintInt System Call

SysPrintInt()

```
void SysPrintInt(int val)
{
    DEBUG(dbgTraCode, "In ksyscall.h:SysPrintInt, into synchConsoleOut->PutInt,
" << kernel->stats->totalTicks);
    kernel->synchConsoleOut->PutInt(val);    // pass val to PutInt()
    DEBUG(dbgTraCode, "In ksyscall.h:SysPrintInt, return from
synchConsoleOut->PutInt, " << kernel->stats->totalTicks);
}
```

在 ExceptionHandler 中的 SC_PrintInt case 被呼叫後，得到參數 val，再將此參數傳至 SyschConsoleOut::PutInt()。

SysConsoleOutput::PutInt()

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

將得到的 integer 轉為適合在 console output 的字串型式 (%d\n\0)，再逐字交由 ConsoleOutput::PutChar() 處理，'\0' 為中止字元。
waitFor->P() 模擬等待 PutChar() 結束的訊號。 (可參考 SysConsoleOutput::CallBack())

SysConsoleOutput::PutChar()

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

原理與 SynchConsoleOutput::PutInt() 相同，但只有在 console output 單一字元 (char)。

ConsoleOutput::PutChar()

```
void ConsoleOutput::PutChar(char ch)
{
    ASSERT(putBusy == FALSE);
    WriteFile(writeFileNo, &ch, sizeof(char));
    putBusy = TRUE;
```

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

在指定檔案(預設為stdout)中，寫入ch，並 schedule 一個 ConsoleWriteInt 的 interrupt。

Interrupt::Schedule()

```
void Interrupt::Schedule(CallBackObj *toCall, int fromNow, IntType type)
{
    int when = kernel->stats->totalTicks + fromNow;    // when to callback
    PendingInterrupt *toOccur = new PendingInterrupt(toCall, when, type);

    DEBUG(dbgInt, "Scheduling interrupt handler the " << intTypeNames[type] <<
" at time = " << when);
    ASSERT(fromNow > 0);

    pending->Insert(toOccur);
}
```

在 ConsoleOutput::PutChar() 執行此 method 後，會將指定的 interrupt 進行排程 (在此為 ConsoleWriteInt)。

Machine::Run()

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

        DEBUG(dbgTraCode, "In Machine::Run(), into OneTick " << "== Tick " <<
kernel->stats->totalTicks << " ==");
        kernel->interrupt->OneTick();    // simulate one tick
        DEBUG(dbgTraCode, "In Machine::Run(), return from OneTick " << "== Tick
" << kernel->stats->totalTicks << " ==");
        if (singleStep && (runUntilTime <= kernel->stats->totalTicks))
                Debugger();
    }
}
```

在進行以上 System call 之後，會回到 Machine::Run() ，接續執行Interrupt::OneTick()。

Interrupt::OneTick()

```
void Interrupt::OneTick()
```

```
{
    MachineStatus oldStatus = status;
    Statistics *stats = kernel->stats;

// advance simulated time
    if (status == SystemMode) {
        stats->totalTicks += SystemTick;
        stats->systemTicks += SystemTick;
    } else {
        stats->totalTicks += UserTick;
        stats->userTicks += UserTick;
    }
    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) {        // if the timer device handler asked
                                // for a context switch, ok to do it now
        yieldOnReturn = FALSE;
        status = SystemMode;            // yield is a kernel routine
        kernel->currentThread->Yield();
        status = oldStatus;
    }
}
```

在 OneTick() 中，會模擬時間的進行，同時也會確認 pending interrupts 是否到了指定的排程 (CheckIfDue(FALSE))。
在確認前，必須關閉 interrupt，結束確認後再開啟 (ChangeLevel())

Interrupt::CheckIfDue()

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

確認是否有 pending interrupts 到期，若沒有，因在 OneTick() 中呼叫時參數為 FALSE，會直接回傳。

如果有應該被執行的 interrupts，便會開始從 pending 的第一個檢查，直到沒有 pending 或 已經沒有到期 interrupts 了。處理 interrupts，即 callback 當初在 schedule 時指定的 Object->CallBack()

(在此 case 為 ConsoleOutput::CallBack())。

ConsoleOutput::CallBack()

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

Interrupt 之後，將 putBusy 設為 FALSE (代表結束 putting char)，並記錄數值。
最後 call back callWhenDone->CallBack() (在此 case 中為 SynchConsoleOutput::CallBack())

SynchConsoleOutput::CallBack()

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

使用 waitFor->V() 模擬 PutChar() 結束的訊號，觸發在 PutInt() 中的 waitFor->P()。

(d) Makefile

make <filename>

編譯指定檔案。需要編譯的檔案在 Makefile 中有做編譯需要的指令定義，因此只要 “make <filename>” 即可編譯。而若有新增需要編譯的檔案，則要在 Makefile 中另行定義。

make clean

清除編譯時產生的檔案。

Our Part

OpenFileId Open(char name);

exception.cc: 首先先在exception.cc增加一個新的SC_Open case，在其中讀取現在filename存的位址。根據位址找到名稱後便將要打開的檔案名稱後將其傳入SysOpen。

```
case SC_Open:
        val = kernel->machine->ReadRegister(4); //讀取位址
        {
                char *filename = &(kernel->machine->mainMemory[val]); //在位址
中找到要打開的文件名稱
                cout << filename << endl;
```

```
                status = SysOpen(filename);
                kernel->machine->WriteRegister(2, (int) status); //將狀態存入
reg2
        }
        kernel->machine->WriteRegister(PrevPCReg,
kernel->machine->ReadRegister(PCReg)); //增加PC
        kernel->machine->WriteRegister(PCReg,
kernel->machine->ReadRegister(PCReg) + 4);
        kernel->machine->WriteRegister(NextPCReg,
kernel->machine->ReadRegister(PCReg)+4);
        return;
        ASSERTNOTREACHED();
        break;
```

ksyscall.h: 在ksyscall.h中增加SysOpen()，並call到OpenAFile 也將file name傳入。

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

filesys.h: 利用for迴圈尋找在OpenFileTable中可以放的位置，將檔案開啟後，便已開啟的檔案紀錄在OpenFileTable中，而若是for迴圈在20個內都找不到可以放的位置，就代表超過可以開的檔案數量了，回傳-1。

```
OpenFileId OpenAFile(char *name) {
        for(int i = 0; i < 20 ; i++){ //找可以放的位置
            if(OpenFileTable[i] == NULL){ //有空位
                int fileDescriptor = OpenForReadWrite(name, FALSE); //開，失敗會
是-1
                if (fileDescriptor == -1) return -1; //失敗了所以回傳-1

                OpenFileTable[i] = new OpenFile(fileDescriptor); //將現在開的放
入OpenFileTable中
                return i; //回傳現在的id
            }
        }
        return -1;//20個位置中沒有空位，超過可以開啟的檔案數量了
    }
```

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

exception.cc**:** 在 ExceptionHandler 中針對 SC_Write system call 的參數做存取，取得欲寫入字串的 address (buffer)、寫入長度 (size)、及寫入的檔案 (fid)，再進行 system call。

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

```
    {
        char *buffer = &(kernel->machine->mainMemory[val]);
        int size = kernel->machine->ReadRegister(5);
        OpenFileId fid = (OpenFileId) kernel->machine->ReadRegister(6);

        int ret = SysWrite(buffer, size, fid);
        kernel->machine->WriteRegister(2, (int)ret);
    }
    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;
```

ksyscall.h: 在 system call 中，利用 kernel 中的 fileSystem 來進行 file writing。

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

filesys.h: 先檢查目標的檔案是否開啟，若無則回傳 -1。再使用 OpenFile::Write() 在目標檔案中 writing，最後回傳成功寫入的文字數。

```
int WriteFile(char *buffer, int size, OpenFileId id){
    OpenFile *openFile = OpenFileTable[id];
    if (!openFile) return -1;
    return openFile->Write(buffer, size);
}
```

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

exception.cc**:** 在 ExceptionHandler 中針對 SC_Read system call 的參數做存取，取得要存入的目標 address (buffer)、要讀取的文字長度 (size)、及要讀取的檔案 (id)，在進行 system call。

```
case SC_Read:
    val = kernel->machine->ReadRegister(4);
    {
        char *buffer = &(kernel->machine->mainMemory[val]);
        int size = kernel->machine->ReadRegister(5);
        OpenFileId fid = (OpenFileId) kernel->machine->ReadRegister(6);

        int ret = SysRead(buffer, size, fid);
```

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

ksyscall.h: 在 system call 中，利用 kernel 中的 fileSystem 來進行 file reading。

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

filesys.h: 先檢查目標的檔案是否開啟，若無則回傳 -1。再使用 OpenFile::Read() 對目標檔案 reading，最後回傳成功讀取的文字數。

```
int ReadFile(char *buffer, int size, OpenFileId id){
    OpenFile *openFile = OpenFileTable[id];
    if (!openFile) return -1;
    return openFile->Read(buffer, size);
}
```

int Close(OpenFileId id);

exception.cc**:** 在其中新增一個case，並讀取要關閉的id值(val)，將其傳入SysClose()中。

```
case SC_Close:
        val = kernel->machine->ReadRegister(4); //讀取要關閉的id
        {
                //char *filename = &(kernel->machine->mainMemory[val]);
                cout << val << endl;
                status = SysClose(val); //將數值傳入SysClose()
                kernel->machine->WriteRegister(2, (int) status); //將狀態記錄到
reg2中
        }
        kernel->machine->WriteRegister(PrevPCReg,
kernel->machine->ReadRegister(PCReg)); //增加PC
        kernel->machine->WriteRegister(PCReg,
kernel->machine->ReadRegister(PCReg) + 4);
        kernel->machine->WriteRegister(NextPCReg,
kernel->machine->ReadRegister(PCReg)+4);
```

```
        return;
        ASSERTNOTREACHED();
        break;
```

ksyscall.h: 在ksyscall.h中增加SysClose()，並call CloseFile 也將file id傳入。

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

filesys.h: 檢查傳入的id檔案是否存在，無則回傳。找到那個檔案後先將OpenFileTable的位址設為NULL，並將該檔案刪除。

```
int CloseFile(OpenFileId id){
    if(id < 0 || id >= 20) return -1; //檢查是否在0-20間
    if(OpenFileTable[id] == NULL) return -1; //檢查這個id是否有檔案

    OpenFile *ClosedFile = OpenFileTable[id]; //找到要被關的檔案
    OpenFileTable[id] = NULL; //將那個id設為NULL

    delete ClosedFile; //Close(ClosedFile); //並將檔案刪除

    return 1; //回傳成功關閉了
    }
```

Problem Encounter

侯茹文: 在寫close的部分時，猶豫要用delete還是close猶豫了很久，最終查了一些相關資料後發現delete應該是沒問題的。

呂宜嫻: 編譯時忘記設定 start.S，導致無法編譯 system call 的部分。