MP1_Report_Team11

Team Member

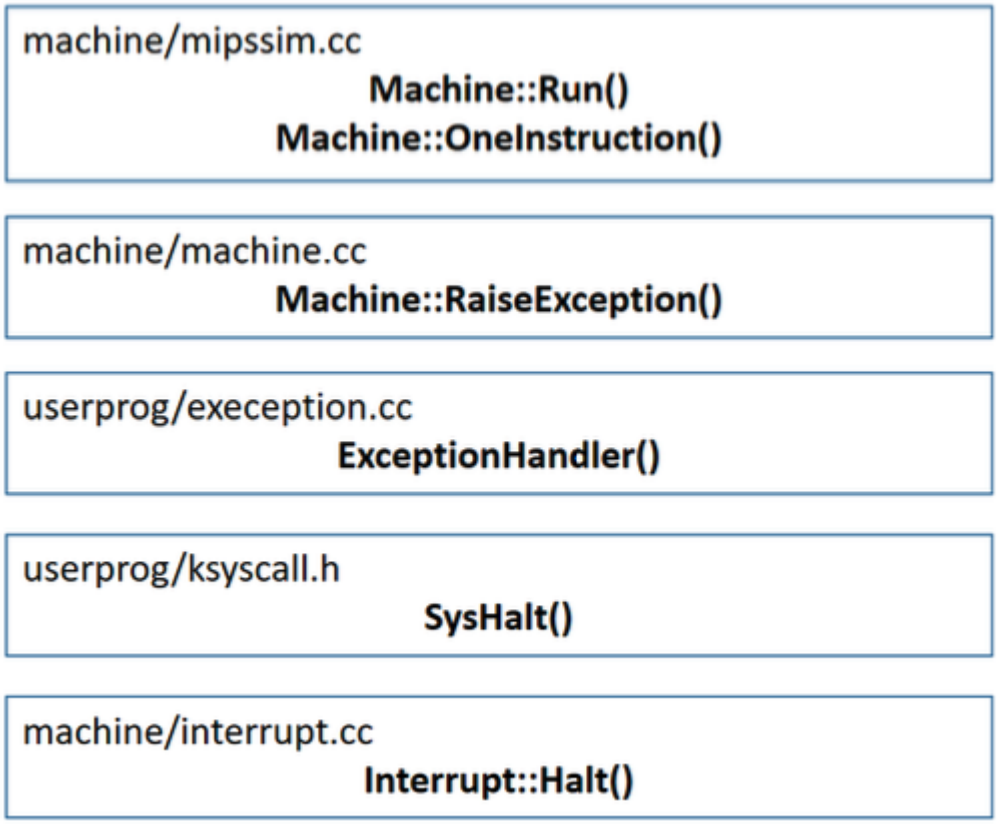
- 資工大三 108062213 顏浩昀
- 電資大三 108061250 吳長錡

Contributions

Work	Member
Trace code (SC_Halt & SC_Create)	顏浩昀&吳長錡
Trace code (SC_PrintInt)	吳長錡
Implement(syscall.h & ksyscall.h & filesys.h)	顏浩昀&吳長錡
Implement(exception. cc & start.S)	顏浩昀
Debug	顏浩昀&吳長錡
report(PartII-1)	吳長錡
report(PartII-2)	顏浩昀
report(remains)	顏浩昀&吳長錡

Part II-1 Trace code

a) SC_Halt



- 當User Mode調用system call的街口時，Nachos會執行相對應的stub
- System call stub is defined in start.S

start.S

```
        .globl Halt
        .ent    Halt
Halt:
        addiu $2,$0,SC_Halt
        syscall
        j       $31
        .end Halt
```

- Write system call type into number 2 register(SC_Halt is 0, defined in system.h)
- Run system call
- Return number 31 register address

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.
//----------------------------------------------------------------------

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

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

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

- 當系統執行syscall指令時會傳到mipssim.cc的此function
- 系統會將接收到的syscall指令傳送到OneInstruction()

Machine::OneInstruction()

```
case OP_SYSCALL:
  DEBUG(dbgTraCode, "In Machine::OneInstruction, RaiseException(SyscallException, 0), " << kernel->stats->totalTicks);
  RaiseException(SyscallException, 0);
  return;
```

- 模擬CPU逐條指令執行過程
- 發現syscall指令時，調用RaiseException拋出SyscallException異常

RaiseException()

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

void
Machine::RaiseException(ExceptionType which, int badVAddr)
{
    DEBUG(dbgMach, "Exception: " << exceptionNames[which]);
    registers[BadVAddrReg] = badVAddr;
    DelayedLoad(0, 0);                  // finish anything in progress
    kernel->interrupt->setStatus(SystemMode);
    ExceptionHandler(which);            // interrupts are enabled at this point
    kernel->interrupt->setStatus(UserMode);
}
```

- 進入到此function中後發現，將exection繼續傳入ExeptionHandler()
- 傳入ExceptionHandler之前，從User Mode轉變為Kernal Mode
- 傳入ExceptionHandler之後，從Kernal Mode轉回User Mode

ExceptionHandler()

```
//----------------------------------------------------------------------
// ExceptionHandler
//      Entry point into the Nachos kernel.  Called when a user program
//      is executing, and either does a syscall, or generates an addressing
//      or arithmetic exception.
//
//      For system calls, the following is the calling convention:
//
//      system call code -- r2
//              arg1 -- r4
//              arg2 -- r5
//              arg3 -- r6
//              arg4 -- r7
//
//      The result of the system call, if any, must be put back into r2.
//
// If you are handling a system call, don't forget to increment the pc
// before returning. (Or else you'll loop making the same system call forever!)
//
//      "which" is the kind of exception.  The list of possible exceptions
//      is in machine.h.
//----------------------------------------------------------------------
void
ExceptionHandler(ExceptionType which)
{
    char ch;
    int val;
    int type = kernel->machine->ReadRegister(2);
    int status, exit, threadID, programID, fileID, numChar;
    DEBUG(dbgSys, "Received Exception " << which << " type: " << type << "\n");
    DEBUG(dbgTraCode, "In ExceptionHandler(), Received Exception " << which << " type: " << type << ", " << kernel->stats->totalTicks);
    switch (which) {
    case SyscallException:
        switch(type) {
            case SC_Halt:
                DEBUG(dbgSys, "Shutdown, initiated by user program.\n");
                SysHalt();
                cout<<"in exception\n";
                ASSERTNOTREACHED();
            break;
            case SC_PrintInt:
                DEBUG(dbgSys, "Print Int\n");
                val=kernel->machine->ReadRegister(4);
                DEBUG(dbgTraCode, "In ExceptionHandler(), into SysPrintInt, " << kernel->stats->totalTicks);
                SysPrintInt(val);
                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;
            case SC_MSG:
                DEBUG(dbgSys, "Message received.\n");
                val = kernel->machine->ReadRegister(4);
                {
                char *msg = &(kernel->machine->mainMemory[val]);
                cout << msg << endl;
                }
                SysHalt();
                ASSERTNOTREACHED();
            break;
            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(PCReg));
                kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) + 4);
                kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg)+4);
                return;
                ASSERTNOTREACHED();
            break;
            case SC_Add:
                DEBUG(dbgSys, "Add " << kernel->machine->ReadRegister(4) << " + " << kernel->machine->ReadRegister(5) << "\n");
```

- 首先判斷Exception Type，Exception Type定義於Machine.h

- 從2號register取出type判斷要處理的事情並執行
- SC_Halt執行SysHalt()

SysHalt()

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

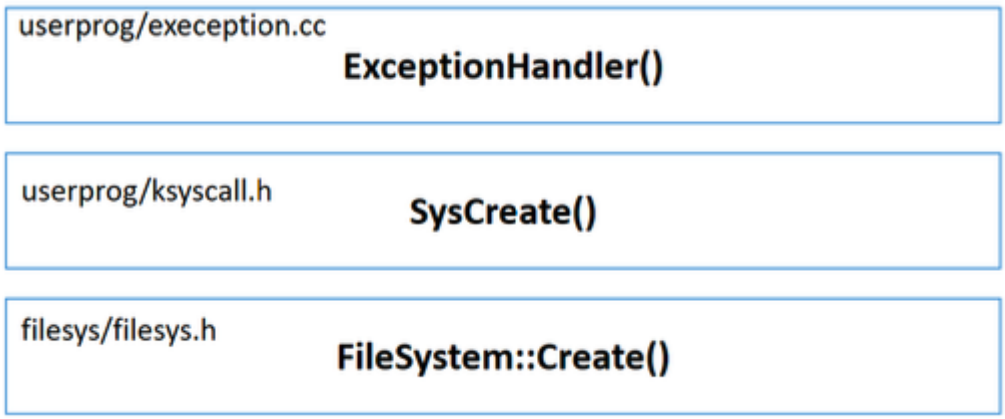
- 執行interrupt.cc的Halt()

Halt()

```
//----------------------------------------------------------------------
// Interrupt::Halt
//      Shut down Nachos cleanly, printing out performance statistics.
//----------------------------------------------------------------------
void
Interrupt::Halt()
{
    cout << "Machine halting!\n\n";
    cout << "This is halt\n";
    kernel->stats->Print();
    delete kernel;      // Never returns.
}
```

- 刪除kernel

b) SC_Create



- ExceptionHandler()前的運作方式都跟SC_Halt一樣
- 因此我們從ExceptionHandler()開始trace

ExceptionHandler()

```
//----------------------------------------------------------------------
// ExceptionHandler
//      Entry point into the Nachos kernel.  Called when a user program
//      is executing, and either does a syscall, or generates an addressing
//      or arithmetic exception.
//
//      For system calls, the following is the calling convention:
//
//      system call code -- r2
//              arg1 -- r4
//              arg2 -- r5
//              arg3 -- r6
//              arg4 -- r7
//
//      The result of the system call, if any, must be put back into r2.
//
// If you are handling a system call, don't forget to increment the pc
// before returning. (Or else you'll loop making the same system call forever!)
//
//      "which" is the kind of exception.  The list of possible exceptions
//      is in machine.h.
//----------------------------------------------------------------------
void
ExceptionHandler(ExceptionType which)
{
    char ch;
    int val;
    int type = kernel->machine->ReadRegister(2);
    int status, exit, threadID, programID, fileID, numChar;
    DEBUG(dbgSys, "Received Exception " << which << " type: " << type << "\n");
    DEBUG(dbgTraCode, "In ExceptionHandler(), Received Exception " << which << " type: " << type << ", " << kernel->stats->totalTicks);
    switch (which) {
    case SyscallException:
        switch(type) {
            case SC_Halt:
                DEBUG(dbgSys, "Shutdown, initiated by user program.\n");
                SysHalt();
                cout<<"in exception\n";
                ASSERTNOTREACHED();
            break;
            case SC_PrintInt:
                DEBUG(dbgSys, "Print Int\n");
                val=kernel->machine->ReadRegister(4);
                DEBUG(dbgTraCode, "In ExceptionHandler(), into SysPrintInt, " << kernel->stats->totalTicks);
                SysPrintInt(val);
                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;
            case SC_MSG:
                DEBUG(dbgSys, "Message received.\n");
                val = kernel->machine->ReadRegister(4);
                {
                char *msg = &(kernel->machine->mainMemory[val]);
                cout << msg << endl;
                }
                SysHalt();
                ASSERTNOTREACHED();
            break;
            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(PCReg));
                kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) + 4);
                kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg)+4);
                return;
                ASSERTNOTREACHED();
            break;
            case SC_Add:
                DEBUG(dbgSys, "Add " << kernel->machine->ReadRegister(4) << " + " << kernel->machine->ReadRegister(5) << "\n");
```

- 首先判斷Exception Type，Exception Type定義於Machine.h
- 從2號register取出type判斷要處理的事情並執行
- SC_Create執行SysCreate()
- WriteRegister()將value寫入user progrom register
- 在ExceptionHandler中用來設置先前的program counter，單純debug用

SysCreate()

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

- 執行fileSystem.cc的Create()

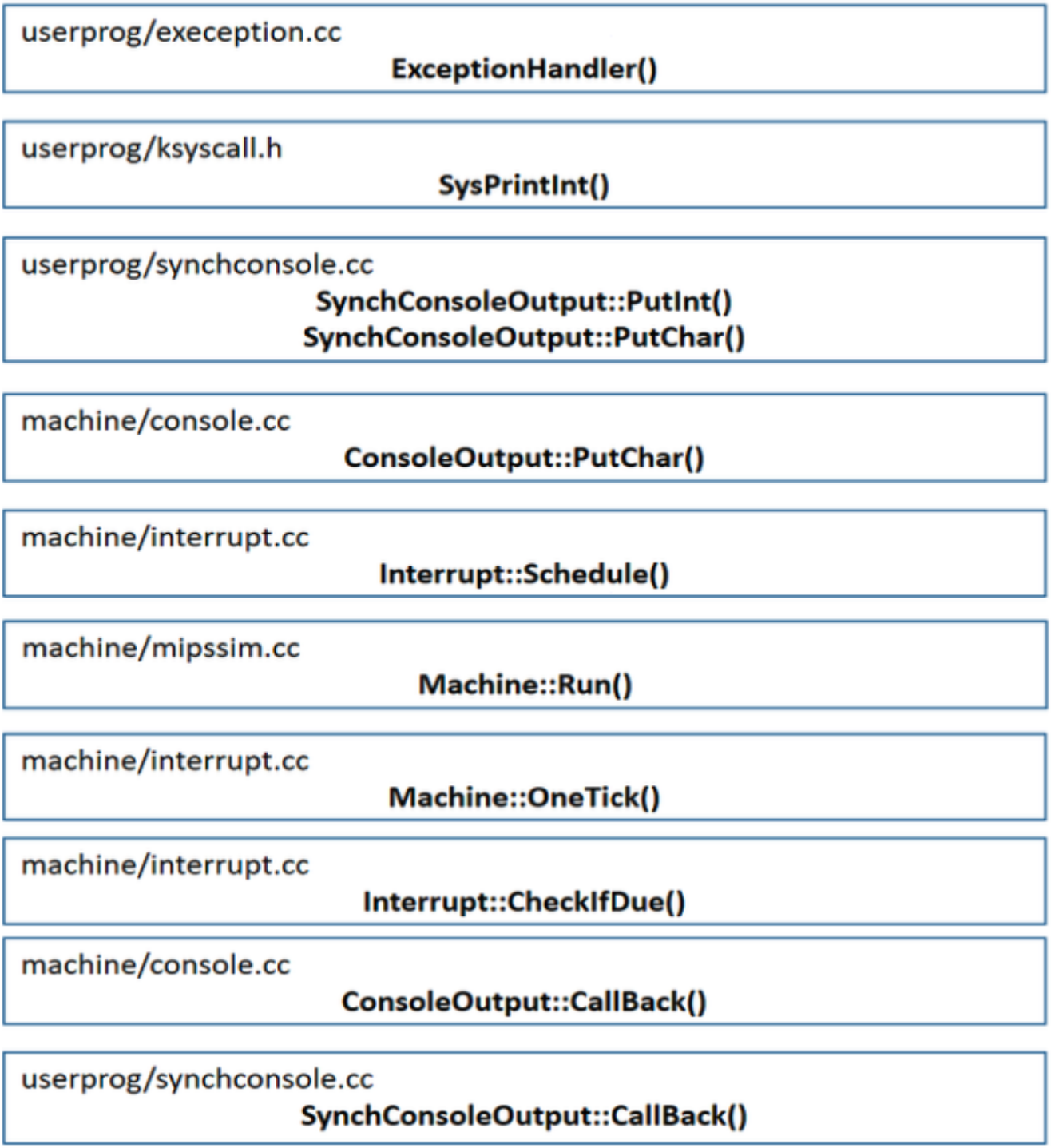
FileSystem::Create()

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

    if (fileDescriptor == -1) return FALSE;
    Close(fileDescriptor);
    return TRUE;
}
```

- OpenForWrite是sysdep.c裡面的函式
- 是呼叫C原生的open()
- 故稱之為stub file system

c) SC_PrintInt



- ExceptionHandler()前的運作方式都跟SC_Halt一樣
- 因此我們從ExceptionHandler()開始trace

ExceptionHandler()

```
//----------------------------------------------------------------------
// ExceptionHandler
//      Entry point into the Nachos kernel.  Called when a user program
//      is executing, and either does a syscall, or generates an addressing
//      or arithmetic exception.
//
//      For system calls, the following is the calling convention:
//
//      system call code -- r2
//              arg1 -- r4
//              arg2 -- r5
//              arg3 -- r6
//              arg4 -- r7
//
//      The result of the system call, if any, must be put back into r2.
//
// If you are handling a system call, don't forget to increment the pc
// before returning. (Or else you'll loop making the same system call forever!)
//
//      "which" is the kind of exception.  The list of possible exceptions
//      is in machine.h.
//----------------------------------------------------------------------
void
ExceptionHandler(ExceptionType which)
{
    char ch;
    int val;
    int type = kernel->machine->ReadRegister(2);
    int status, exit, threadID, programID, fileID, numChar;
    DEBUG(dbgSys, "Received Exception " << which << " type: " << type << "\n");
    DEBUG(dbgTraCode, "In ExceptionHandler(), Received Exception " << which << " type: " << type << ", " << kernel->stats->totalTicks);
    switch (which) {
    case SyscallException:
        switch(type) {
            case SC_Halt:
                DEBUG(dbgSys, "Shutdown, initiated by user program.\n");
                SysHalt();
                cout<<"in exception\n";
                ASSERTNOTREACHED();
            break;
            case SC_PrintInt:
                DEBUG(dbgSys, "Print Int\n");
                val=kernel->machine->ReadRegister(4);
                DEBUG(dbgTraCode, "In ExceptionHandler(), into SysPrintInt, " << kernel->stats->totalTicks);
                SysPrintInt(val);
                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;
            case SC_MSG:
                DEBUG(dbgSys, "Message received.\n");
                val = kernel->machine->ReadRegister(4);
                {
                char *msg = &(kernel->machine->mainMemory[val]);
                cout << msg << endl;
                }
                SysHalt();
                ASSERTNOTREACHED();
            break;
            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(PCReg));
                kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) + 4);
                kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg)+4);
                return;
                ASSERTNOTREACHED();
            break;
            case SC_Add:
                DEBUG(dbgSys, "Add " << kernel->machine->ReadRegister(4) << " + " << kernel->machine->ReadRegister(5) << "\n");
```

- 首先判斷Exception Type，Exception Type定義於Machine.h
- 從2號register取出type判斷要處理的事情並執行
- SC_PrintInt執行SysPrintInt()
- WriteRegister()將value寫入user progrom register
- 在ExceptionHandler中用來設置先前的program counter，單純debug用

SysPrintInt()

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

- 執行synchConsoleOutput::PutInt()

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

- 首先使用sprint將value存到str，變成字元型態
- 利用lock->Aquire()鎖定物件，執行同步化。只有取得鎖定的執行緒才可以進入同步區，未取得鎖定的執行緒必須等待，直到有機會取得鎖定。
- 將str字元陣列一個一個丟入PutChar()
- 執行玩同步化後，lock->Release()解除鎖定
- waitFor->P()讓後面還沒作用的字元先等待

SynchConsoleOutput::PutChar()

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

- 除了傳入的參數是一個字元其他都跟PutInt()一樣

ConsoleOutput::PutChar()

```
//----------------------------------------------------------------------
// ConsoleOutput::PutChar()
//      Write a character to the simulated display, schedule an interrupt
//      to occur in the future, and return.
//----------------------------------------------------------------------

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

- 在Print進行同步化時，從4號register取出的system call參數值會進入console.cc裡的PutChar()
- 首先利用WriteFile()，將數據寫入一個文件中
- 將putBusy的狀態改成True，讓其他事情不能一起做
- 進入interrypt.cc的Schedule()，安排程式預定被CPU執行的時間

Interrupt::Schedule()

```
//----------------------------------------------------------------------
// Interrupt::Schedule
//      Arrange for the CPU to be interrupted when simulated time
//      reaches "now + when".
//
//      Implementation: just put it on a sorted list.
//
//      NOTE: the Nachos kernel should not call this routine directly.
//      Instead, it is only called by the hardware device simulators.
//
//      "toCall" is the object to call when the interrupt occurs
//      "fromNow" is how far in the future (in simulated time) the
//               interrupt is to occur
//      "type" is the hardware device that generated the interrupt
//----------------------------------------------------------------------
void
Interrupt::Schedule(CallBackObj *toCall, int fromNow, IntType type)
{
    int when = kernel->stats->totalTicks + fromNow;
    PendingInterrupt *toOccur = new PendingInterrupt(toCall, when, type);

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

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

- toCall是interrupt時要被執行的對象
- fromnow是指在模擬時間內interrupt發生的時間
- type是產生interrupt的硬體設備
- 這個函數先記錄了interrupt何時要被執行，然後在PendingInterrupt List裡插入要被執行的interrupt

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.
//----------------------------------------------------------------------

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

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

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

- 在SC_Halt裡面有提到，OneInstruction()模擬CPU逐一執行任務的功能
- 在執行完OneIntruction()後，會進入OneTick()函數

Machine::OneTick()

```
//----------------------------------------------------------------------
// 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
//----------------------------------------------------------------------
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;
    }
}
```

- 讓系統時間往前一個時刻，來模擬時間往前的行為
- 這個函數能夠設定中斷狀態，並釋放目前Thread，然後執行下一個Thread
- NachOS interrupt controller模擬一個時鐘，這個時鐘從NachOS啟動時開始計數，作為NachOS的系統時間。當NachOS模擬的CPU執行完成一條指令，ticks = ticks + 1，當中斷狀態從disabled轉到enable，ticks + 10。而此函數就是在模擬時鐘走一格時刻

Interrupt::CheckIfDue()

```
//----------------------------------------------------------------------
// Interrupt::CheckIfDue
//      Check if any interrupts are scheduled to occur, and if so,
//      fire them off.
//
// Returns:
//      TRUE, if we fired off any interrupt handlers
// Params:
//      "advanceClock" -- if TRUE, there is nothing in the ready queue,
//              so we should simply advance the clock to when the next
//              pending interrupt would occur (if any).
//----------------------------------------------------------------------
bool
Interrupt::CheckIfDue(bool advanceClock)
{
    PendingInterrupt *next;
    Statistics *stats = kernel->stats;

    ASSERT(level == IntOff);            // interrupts need to be disabled,
                                        // to invoke an interrupt handler
    if (debug->IsEnabled(dbgInt)) {
        DumpState();
    }
    if (pending->IsEmpty()) {           // no pending interrupts
        return FALSE;
    }
    next = pending->Front();

    if (next->when > stats->totalTicks) {
        if (!advanceClock) {            // not time yet
            return FALSE;
        }
        else {                  // advance the clock to next interrupt
            stats->idleTicks += (next->when - stats->totalTicks);
            stats->totalTicks = next->when;
            // UDelay(1000L); // rcgood - to stop nachos from spinning.
        }
    }

    DEBUG(dbgInt, "Invoking interrupt handler for the ");
    DEBUG(dbgInt, intTypeNames[next->type] << " at time " << next->when);

    if (kernel->machine != NULL) {
        kernel->machine->DelayedLoad(0, 0);
    }

    inHandler = TRUE;
    do {
        next = pending->RemoveFront();    // pull interrupt off list
                DEBUG(dbgTraCode, "In Interrupt::CheckIfDue, into callOnInterrupt->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;
}
```

- 此函數目的是檢查全部interrupts是否有如預期的發生，並且解決
- 當所有interrupts解決完成後，回傳True

ConsoleOutput::CallBack()

```
//----------------------------------------------------------------------
// ConsoleInput::CallBack()
//      Simulator calls this when a character may be available to be
//      read in from the simulated keyboard (eg, the user typed something).
//
//      First check to make sure character is available.
//      Then invoke the "callBack" registered by whoever wants the character.
//----------------------------------------------------------------------

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();
    }
}
```

- 當下一個字元可以輸出的顯示器時，模擬器將調用此函數

SynchConsoleOutput::CallBack()

```
//----------------------------------------------------------------------
// SynchConsoleOutput::CallBack
//      Interrupt handler called when it's safe to send the next
//      character can be sent to the display.
//----------------------------------------------------------------------

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

- 如果可以安全地發送下一個字元，調用interrupt，並送到顯示器

Part II-2 Implement

userprog/Syscall.h

觀察此檔案發現nachos已經都幫我們define好nachos kernel需要支援的operation，我們只需要將下面程式碼的註解拿掉即可（影響system call type）。

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

test/start.S

參考原先的code，在start.S檔案中新增下列程式碼，讓open, write, read, close可以順利執行。
start.S檔案定義了system call的實作方式，實際作法可見之前Trace code部分。

```
        .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
```

userprog/exception.cc

當執行machine::RaiseException()後，會進入exception.c的ExceptionHandler()，在ExceptionType = syscall的case下新增SC_Open、SC_Write、SC_Read、SC_Close四個case。

- 下方程式碼重要名詞意義

> Register(2): system code type
> Register(4): arg1（function的第一個變數）
> Register(5): arg2（function的第二個變數）
> Register(6): arg3（function的第三個變數）

1. **SC_Open實作：** 先讀到register(4)的值設為val(filename在mainMemory中的位置)，然後將整個address存於filename，接著執行SysOpen，然後透過fileID儲存id(error時為-1)，再把結果寫回Resgister(2)。

```
case SC_Open:
    //DEBUG(dbgSys, "exception SC_Open begin \n");
    val = kernel->machine->ReadRegister(4);
    filename = &(kernel->machine->mainMemory[val]);
    fileID = SysOpen(filename);
    kernel->machine->WriteRegister(2, (int)fileID);
```

在完成後去更新Programming Counter，已完成這一次的system call，繼續執行下一條指令（每一次system call完成皆會執行，後面不再重複敘述）。

```
    {
    kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegiste
    kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PC
    kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegiste
    }
    //DEBUG(dbgSys, "exception SC_Open end \n");
    return;
    ASSERTNOTREACHED();
    break;
```

2. **SC_Read實作：** 先讀到register(4)的值設為val(filename在mainMemory中的位置)，然後將整個address存於filename，接著執行SysRead，然後透過numChar儲存回傳的size(error時為-1)，再把結果寫回Resgister(2)。

```
case SC_Read:
        val = kernel->machine->ReadRegister(4);
        filename = &(kernel->machine->mainMemory[val]);
        numChar = SysRead(filename, kernel->machine->ReadRegister(5), kernel->m
        kernel->machine->WriteRegister(2, (int)numChar);
        {
        kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister
        kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCR
        kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister
        }
        return;
        ASSERTNOTREACHED();
        break;
```

3. **SC_Write實作：** 先讀到register(4)的值設為val(buf在mainMemory中的位置)，然後將整個address存於buf，接著執行SysWrite，然後透過numChar儲存回傳的size(error時為-1)，再把結果寫回Resgister(2)。

```
case SC_Write:
        //DEBUG(dbgSys, "exception SC_Write begin \n");
        val = kernel->machine->ReadRegister(4);
        buf = &(kernel->machine->mainMemory[val]);
        numChar = SysWrite(buf, kernel->machine->ReadRegister(5), kernel->machi
        kernel->machine->WriteRegister(2, (int)numChar);
        {
        kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister
        kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCR
        kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister
        }
        //DEBUG(dbgSys, "exception SC_Write end \n");
        return;
        ASSERTNOTREACHED();
        break;
```

4. **SC_Close實作：** 先讀到register(4)的值設為val(要close的id)，然後將整個address存於buf，接著執行SysWrite，然後透過status儲存回傳值，(success為1，error時為-1)，再把結果寫回Resgister(2)。

```
case SC_Close:
        val=kernel->machine->ReadRegister(4);
        status = SysClose(val);
        kernel->machine->WriteRegister(2, status);
        {
        kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister
        kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCR
        kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister
        }
        return;
        ASSERTNOTREACHED();
        break;
```

userprog/ksyscall.h

這部分是exception.c中會呼叫的function，只需要新增下下幾行code，進一步去使用fileSystem的function即可。

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

int SysWrite(char *buffer, int size, OpenFileId id)
{
  return kernel->fileSystem->WriteFile0(buffer, size, id);
}

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

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

filesys/filesys.h

1. **OpenAFile()**： 透過呼叫同為filesys當中的Open()先取得型態為OpenFile的資料，如果因為檔案名稱錯誤沒能成功打開檔案，指標openFile會是NULL，也就會return -1達到判斷error。如果Open成功取得檔案則會進入for-loop判斷開啟的檔案有沒有超過上限。判斷的方式就是透過OpenFileTable，從i =[0]~[19]找尋仍為NULL的的位置將openFile放入然後回傳i便是OpenFileId；若是i =[0]~[19]都已經有值，代表已經達到檔案開啟上限（20個）則離開for-loop並回傳-1。

```
OpenFileId OpenAFile(char *name) {
        OpenFile *openFile = Open(name);
        if (openFile == NULL) return -1;
        for (int i = 0; i < 20; i++) {
                if (OpenFileTable[i] == NULL) {
                        OpenFileTable[i] = openFile;
                        return i;
                }
        }
        return -1;
}
```

2. **WriteFile()**： 首先先透過OpenFileTable[id]取得要write的檔案，如果沒有這份檔案就會return -1。如果有檔案則會透過Openfile的Write()，將buffer及指定的size寫入file中，並回傳Write()的回傳值（寫入的nByte）。

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

3. **ReadFile()**： 同樣透過OpenFileTable[id]取得要read的檔案，如果沒有這份檔案則return -1，如果有則透過Openfile的Read()從檔案裡的資料傳指定size至buffer中並回傳Read()的回傳值（讀到的nByte）

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

4. **CloseFile()**： 先透過OpenFileTable[id]判斷有沒有開著要close的檔案，如果沒有則return -1。如果有則delete OpenFileTable[id]然後再將其設為NULL，確保下一次還可以open在這個id位置，並在完成後return 1。

```
int CloseFile(OpenFileId id) {
    if (OpenFileTable[id] == NULL) return -1;
        else {
                delete OpenFileTable[id];
                OpenFileTable[id] = NULL; //add (10.21 1:32)
                return 1;
        }
}
```

Difficulties that we encounter

在這次作業中主要花費時間在Trace code上，尤其在眾多資料夾與檔案中，需要一些時間去記得哪一份檔案會在哪一個資料夾中。另外當程式裡call到其他檔案的function時，也容易搞不清楚該去哪個檔案尋找，這是我們在Trace code時遇到的第一個困難。
在Trace過程中我們遇到比較困難的部分是schedule及callback的部分，花費不少時間才理解模擬時間計算的方式。

在Implement部分的coding其實難度不高，大部分都是瞭解後照著附近相同類型function撰寫即可，有些甚至只需要移除註解。但我們遇上的困難主要是「搞錯測試資料夾」及「Debug」。因為make指令會在build.linux資料夾使用，後面測試指令又包含.../build.linux，結果誤以為是要在build/linux資料夾運作，導致根本找不到我們要運作的檔案（檔案在test資料夾），不斷發生

Unable to open file的錯誤。但上面的困難照理來說很快就能依照錯誤訊息找出錯誤，但就因為Nachos的Debug會根據machine、system call、interrupt等等有不同分類，一開始根本不知道錯誤訊息是從哪裡來的，因此又花了時間理解Debug的運作，也終於成功找出錯誤所在。

另外因為對vim的使用很不熟悉，而且不像vscode等編輯器可能會偵錯、自動添加括號等輔助，雖然這次都沒有出問題，但覺得很容易不小心有很微小的輸入錯誤又找不到bug。因此我們希望能在下一次的作業把code用git的方式拿出並用熟悉的編輯器編輯，期待能增加我們的撰寫效率。