(1)SC_Halt:

Machine::Run()

```
void Machine::Run() {
  Instruction *instr = new Instruction;  // storage for decoded instruction
  if (debug->IsEnabled('m')) {          //debug.cc 如果m有在Debug message裡
    cout << "Starting program in thread: " << kernel->currentThread->getName();
    cout << ", at time: " << kernel->stats->totalTicks << "\n";
  }
  kernel->interrupt->setStatus(UserMode); //user program執行 使用usermode interrupt.h IdleMode, SystemMode, UserMode
  for (;;) {
    DEBUG(dbgTraCode, "In Machine::Run(), into OneInstruction "               //debug.h  dbgTraCode = 'c' 如果這個flag enable時候 就印出後面的訊息
                      << "== Tick " << kernel->stats->totalTicks << " ==");
    OneInstruction(instr);                                                    //執行解碼完的instruction
    DEBUG(dbgTraCode, "In Machine::Run(), return from OneInstruction  "
                      << "== Tick " << kernel->stats->totalTicks << " ==");

    DEBUG(dbgTraCode, "In Machine::Run(), into OneTick "                      //debug.h  dbgTraCode = 'c' 如果這個flag enable時候 就印出後面的訊息
                      << "== Tick " << kernel->stats->totalTicks << " ==");
    kernel->interrupt->OneTick();                                             //執行OneTick
    DEBUG(dbgTraCode, "In Machine::Run(), return from OneTick "
                      << "== Tick " << kernel->stats->totalTicks << " ==");
    if (singleStep && (runUntilTime <= kernel->stats->totalTicks))
      Debugger();
  }
}
```

把解碼後的 instruction 如 ADD, BEQ 丟進 OneInstruction(instr)

做運算

Machine::OneInstruction()

模擬 CPU 執行各個指令, 其中還有 Exception 的 opcode, 遇到異常會

丟出 RaiseException 訊息

Machine:RaiseException()

```
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); //轉成kernel mode
    ExceptionHandler(which);        // interrupts are enabled at this point
    kernel->interrupt->setStatus(UserMode); //轉回user mode
}
```

當發生 Exception, 印出訊息, 進入 Kernel Mode, 解決完之後跳回 User Mode

ExceptionHandler()

```
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: " <<
switch (which) {
  case SyscallException:  //看是哪個System call
    switch (type) {
      case SC_Halt:
        DEBUG(dbgSys, "Shutdown, initiated by user program.\n");
        SysHalt();
        cout << "in exception\n";
        ASSERTNOTREACHED();
        break;
```

System call 的值在 Reg2, 以 type 儲存起來, 接著根據這個 type 值看是哪種 system call, 此處是 SC_Halt.

SysHalt()

進入到 **ksyscall.h**,系統的 **system call** 都寫在這了

```
>machine->ReadRegister(2);
threadID, programID, fileID, numChar;
eived Exception " << which << " type: " << type << "
"In ExceptionHandler(), Received Exception " << whic
 void SysHalt()
 {
   kernel->interrupt->Halt();
 }
```

此實作的 **system call** 寫在 **interupt.cc** 的 **Halt()**裡了

Halt()

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

系統 **print** 出訊息後,把整個 **kernel** 結束掉,程式就停止了

(2)SC_Create

ExceptionHandler()

```
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;
```

把要創立的 **file** 名字放入到 **val** 變數(原先儲存在 **Reg4),SysCreate** 詳細在 **ksyscall.h,**並且把 **status** 更新到 **Reg2(system call** 的值),最後更新 **program counter** 的值.

SysCreate()

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

此 **function** 在 **ksyscall.h,System call** 的實作在 **filesys.h**

FileSystem::Create()

```
bool Create(char *name) {
  int fileDescriptor = OpenForWrite(name); //在lib的sysdep.cc

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

再往下追蹤 **OpenForWrite** 可發現在 **lib** 的 **sysdep.cc,**可以看到會呼叫 **Unix** 系統的 **open() system call,**也就是 **stub** 檔案的來由.最後 **Create(),**根據回傳的 **fileDescriptor** 建立檔案,成功為 **1,**失敗為**-1**

(3)SC_PrintInt

ExceptionHandler():

```
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;
```

一樣會判斷是甚麼樣的 system call 類型,此處是 PrintInt,讀出傳入的參數進入 SysPrintInt()

SysPrintInt():

進到 **ksyscall.h** 看到必須進入 **synchConsoleOut.cc**

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

SynchConsoleOutput::PutInt() &&

SynchConsoleOutput::PutChar()

```
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();
}
```

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

使用 **lock,**只有獲得鎖的 **process** 可以進入同步區,其他 **process** 必須等待,

將輸入的整數轉成 **string** 並在最後加入換行以及終止字元,

consoleOutput->PutChar()主要是控制硬體輸出在螢幕上,

waitFor->P() 讓後面還沒做完的字元先等待

ConsoleOutput::PutChar():

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

寫入 **file,**當寫入時候 **flag** 為 **True,**此時不能處理其他事情.

Interrupt::Schedule()

```
void Interrupt::Schedule(CallBackObj *toCall, int fromNow, IntType type) {
  int when = kernel->stats->totalTicks + fromNow;
  //  arg1 is the object to call when the interrupt occurs
  //  arg2 is when (in simulated time) the interrupt is to occur
  //  arg3 is the hardware device that generated the interrupt
  PendingInterrupt *toOccur = new PendingInterrupt(toCall, when, type);
  //Initialize a hardware device interrupt that is to be scheduled
  //  to occur in the near future.
  DEBUG(dbgInt, "Scheduling interrupt handler the " << intTypeNames[type] << " at time =
  ASSERT(fromNow > 0);

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

在 **Interrupt.cc** 底下,**tocall** 是 **interrupt** 執行的對象,**when** 是指 **interrupt** 發生的時間,**type** 是指產生 **interrupt** 的硬體設備,最後把此 **interrupt** 放入 **pending queue** 裡.

Machine::Run()

OneInstrucion 會把解碼好的 **instruction** 放入 **MIP** 模擬 **CPU** 執行指令

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(); // Relinquish the CPU if any
                                    // other thread is runnable
    status = oldStatus;
  }
}
```

目的是要模擬系統往前一個時刻

檢查是否有等待中斷：程式碼會關閉中斷（**turn off interrupts**），然後檢查是否有等待的中斷需要處理。如果有，就會開啟中斷（**re-enable interrupts**）以處理中斷。

如果當前的 **Thread** 用完了 **time slice,kernel** 會釋放現在的 **Thread** 並調度新的 **Thread**

Interrupt::CheckIfDue()

檢查所有 **pending queue** 裡的 **interrupt** 是否全部解決了**,**解決完 **return True.**

ConsoleOutput::CallBack()

當下一個字元可以輸出到 **Monitor** 上了時候使用這個 **function**

SynchConsoleOutput::CallBack()

如果可以安全的發送下一個字元**(semaphore on),**調用 **interrupt,**並送到 **monitor**

PartII:

Syscall.h:

- 先到 syscall.h 把#define SC_Open 6 等等的註解拿掉

start.S

user program call Open()的 **API** 時會發出 **system call** 給 **kernel(SC_Open)**

重複做 **4** 次

```
Add:
    addiu $2,$0,SC_Add
    syscall
    j   $31
    .end Add

    .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

    .globl Exit
    .ent    Exit
/* 以上為修改部分 */
```

Exception.cc

```
case SC_Open:
  DEBUG(dbgSys, "File Open.\n");
  val = kernel->machine->ReadRegister(4); //arg1
  {
    char *filename = &(kernel->machine->mainMemory[val]);
    DEBUG(dbgSys, "File name :"<<filename<<"\n");
    // cout << filename << endl;
    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;
case SC_Write:
  DEBUG(dbgSys, "File Writing...\n");
  val = kernel->machine->ReadRegister(4); //arg1
  numChar = kernel->machine->ReadRegister(5); //arg2

  fileID = kernel->machine->ReadRegister(6); //arg3
  DEBUG(dbgSys, "file ID:"<<fileID<<"\n");
  {
    char *buffer = &(kernel->machine->mainMemory[val]);
    DEBUG(dbgSys, "Buffer:"<<buffer<<"\n");
    // cout << filename << endl;
    //numChar,fileID在函數先定義好了
    status = SysWrite(buffer,numChar,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;
```

```
case SC_Read:
  DEBUG(dbgSys, "File Reading...\n");
  val = kernel->machine->ReadRegister(4); //arg1
  numChar = kernel->machine->ReadRegister(5); //arg2

  fileID = kernel->machine->ReadRegister(6); //arg3
  DEBUG(dbgSys, "file ID:"<<fileID<<"\n");
  {
    char *buffer = &(kernel->machine->mainMemory[val]);
    DEBUG(dbgSys, "Buffer:"<<buffer<<"\n");
    // cout << filename << endl;
    //numChar,fileID在函數先定義好了
    status = SysRead(buffer,numChar,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;
case SC_Close:
  DEBUG(dbgSys, "File Closing...\n");
  fileID = kernel->machine->ReadRegister(4); //arg1
  DEBUG(dbgSys, "file ID:"<<fileID<<"\n");

  status = SysClose(fileID);
  DEBUG(dbgSys, "status:"<<status<<"\n");
  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;
```

我們根據 **function** 傳入的參數,**arg1,arg2,arg3,**去接收對應的值,如 **fileID,numChar.**

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

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

這兩行會去 **memory** 把 **buffer** 的起始位置抓出來

ksyscall.h

負責 system call,在本次作業主要針對檔案做操作,實作細節在

filesys.h.

filesys.h

```
int ReadFile(char *buffer, int size, OpenFileId id) {
    if(size<=0) return -1;
    Read(id,buffer,size);
    return size;
}


"File Closing...\n");
l->machine->ReadRegister(4); //arg1
"file ID:"<<fileID<<"\n");
```

大部分跟上圖的 ReadFile 一樣撰寫邏輯

其中 Read()在 sysdep.c 檔裡.

進去檔案可看到是 Unix 的 read() system call.

成果

```
Received Exception 1 type: 100
Message received.

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
[os23s71@localhost test]$
```

What difficulties did you encounter when implementing this assignment?

蠻開心可以看到以前學過的作業系統以及計算機組織,在實際的程式裡出現,過程有一些函式觀念還是比較模糊,相信之後的學習可以幫助我釐清觀念.