CS342301: Operating System

MP1: System Call

Team31

111060013 劉祐廷 111060019 黃子恩

	劉祐廷	黃子恩
Contribution		

1. Trace Code

I. SC_Halt

A. Machine::Run()

這個 function 負責模擬 NachOS 中的 CPU 執行程序的過程，他會先使用 kernel->interrupt->setStatus(UserMode);，將模式設定成 usermode，接著用後面的無限迴圈 for(;;){}模擬 CPU 執行時一直去 fetch instruction 的動作，在迴圈裡面會先呼叫 OneInstruction()來執行 instruction，在每次 OneInstruction 執行之後會呼叫 OneTick()來檢查有無 interrupt 發生並處理。

B. Machine::OneInstruction()

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

先去讀取 registers[PCReg]裡面的那個 instruction，讀不到則 return，有讀到的話就做 decode。

```
// Execute the instruction (cf. Kane's book)
switch (instr->opCode) {
    case OP_ADD:
        sum = registers[instr->rs] + register
        if (!((registers[instr->rs] ^ registe
```

接著使用 opCode 取找尋這個指令所代表的動作。

```
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;
    }
```

以 OP_ADD 這個動作為例，若遇到 exception 的狀況發生，會把錯誤的代碼當參數傳給 RaiseException()處理，沒有 exception 發生的話就把各自的操作做完。

C. Machine::RaiseException()

```
registers[BadVAddrReg] = badVAddr;
```

把遇到問題的 virtual address 放進專門放 bad virtual address 的 register 裡面。

```
kernel->interrupt->setStatus(SystemMode);
ExceptionHandler(which);  // interrupts a
kernel->interrupt->setStatus(UserMode);
```

切到 system mode 執行 ExceptionHandler()，執行完成後切回 usermode。

D. ExceptionHandler()

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

從 start.S 裡面可以發現，這些 exception 的編碼都被放在了 r2 這個記憶體裡面。

```
int type = kernel->machine->ReadRegister(2);
```

因此可以透過讀取 r2 的值去獲取 system call code。（system call code 存放在 r2）

```
switch (which) {
    case SyscallException:
```

先檢查這個 system call code 是否存在。

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

接著依照 system call code 執行不同的動作，由於傳入的 system call code 為 SC_Halt，因此會執行這段程式碼去呼叫 SysHalt()。

E. SysHalt()

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

呼叫定義在 kernel 可接受的 interrupt 操作裡的 Halt()

F. Interrupt::Halt()

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

解構整個 kernel 釋放資源，停止整個 system。

II. SC_Create

A. ExceptionHandler()

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

從 r4（通常是 arg1）讀取欲新增檔案檔名字串的位址。

```
{
    char *filename = &(kernel->machine->mainMemory[val]);
    // cout << filename << endl;
    status = SysCreate(filename);
    kernel->machine->WriteRegister(2, (int)status);
}
```

把 filename 提取出來傳給 SysCreate()來創建檔案，並把要 return 的值寫入 r2。

```
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);
```

更新 PC 準備執行下一行指令。

B. SysCreate()

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

呼叫 fileSystem 裡面的 Create()，並把 filename 傳給他。

C. FileSystem::Create()

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

獲取對應的 directory。

```
if (directory->Find(name) != -1)
    success = FALSE;  // file is already in directory
```

確認檔案是否存在，若存在則回傳 false。

```
else {
    freeMap = new PersistentBitmap(freeMapFile, NumSectors);
    sector = freeMap->FindAndSet();  // find a sector to hold the file header
    if (sector == -1)
        success = FALSE;  // no free block for file header
    else if (!directory->Add(name, sector))
        success = FALSE;  // no space in directory
```

檢查是否有足夠的空間創建檔案，若不夠則回傳 false。

```
else {
    hdr = new FileHeader;
    if (!hdr->Allocate(freeMap, initialSize))
        success = FALSE;  // no space on disk for data
```

嘗試 allocate 空間給這個檔案，若空間不夠則回傳 false。

```
else {
    success = TRUE;
    // everthing worked, flush all changes back to disk
    hdr->WriteBack(sector);
    directory->WriteBack(directoryFile);
    freeMap->WriteBack(freeMapFile);
}
```

成功創建檔案，將變更後的資料寫回 disk。

```
            delete hdr;
        }
        delete freeMap;
    }
    delete directory;
```

解構這些指標以釋放記憶體資源。

III.SC_PrintInt

A. ExceptionHandler()

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

從 start.S 裡面可以發現，這些 exception 的編碼都被放在了 r2 這個記憶體裡面。

```
int type = kernel->machine->ReadRegister(2);
```

因此可以透過讀取 r2 的值去獲取 system call code。（system call code 存放在 r2）

```
switch (which) {
    case SyscallException:
```

先檢查這個 system call code 是否存在。

```
case SC_PrintInt:
    DEBUG(dbgSys, "Print Int\n");
    val = kernel->machine->ReadRegister(4);
    DEBUG(dbgTraCode, "In ExceptionHandler()
    SysPrintInt(val);
    DEBUG(dbgTraCode, "In ExceptionHandler()
```

接著依照 system call code 執行不同的動作，由於傳入的 system call code 為 SC_PrintInt，因此會執行這段程式碼去呼叫，首先會去 r4（arg1）讀取要輸出的東西存在 val，接著把 val 當參數傳給 SysPrintInt()。

```
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);
```

更新 PC 準備執行下一行指令。

B. SysPrintInt()

```
DEBUG(dbgTraCode, "In ksyscall.h:SysPr
kernel->synchConsoleOut->PutInt(val);
DEBUG(dbgTraCode, "In ksyscall.h:SysPr
```

把要輸出的東西（val）傳給已經定義在 kernel 的 synchConsoleOut 裡面的 PutInt()。

C. SynchConsoleOutput::PutInt()

```
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, in
    consoleOutput->PutChar(str[idx]);
    DEBUG(dbgTraCode, "In SynchConsoleOutput::PutChar, re
    idx++;

    DEBUG(dbgTraCode, "In SynchConsoleOutput::PutChar, in
    waitFor->P();
    DEBUG(dbgTraCode, "In SynchConsoleOutput::PutChar, re
} while (str[idx] != '\0');
lock->Release();
```

把要輸出的數字轉換成字串，並用迴圈一個字元一個字元傳給 PutChar()。

- lock->Acquire();
  這行 code 在執行共享資源操作之前 acquire lock，確保當前的 process 在進入 critical section 之前，獨占對該資源的訪問權。這樣可以避免其他 process 同時訪問 consoleOutput，導致輸出的數據混亂。當 lock 被某個 process 持有時，其他嘗試 acquire lock 的 process 會被阻塞，直到 lock 被 release。
- lock->Release();
  當這段操作完成後 release lock，允許其他 process 進入 critical section。這樣能確保在一次完整的輸出操作結束後，其他 process 可以安全地執行自己的輸出操作。

D. SynchConsoleOutput::PutChar()

比起 SynchConsoleOutput::PutInt()，只是少了將數字轉換成字串的過程，剩下部分大致上一樣。

E. ConsoleOutput::PutChar()

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

putBusy 表示 console output 是否忙碌。他是 FALSE 時表示 console is idle，此時才能開始新的輸出。如果他是 TRUE，表示正在進行輸出，則這次呼叫應該被阻止，這樣可以避免同時輸出多個字符。

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

模擬將 ch 輸出到 console output 中。

```
putBusy = TRUE;
```

把 putBusy 設為 TRUE 表示 console output 忙碌中，可以避免同時輸出多個字符。

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

呼叫 Schedule()以在輸出完成之後 raise up interrupt 表示輸出完成。

F. Interrupt::Schedule()

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

計算這個 interrupt 要被觸發的時間

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

DEBUG(dbgInt, "Scheduling interrupt handler the " << intTypeNames[typ
ASSERT(fromNow > 0);
   dasbd72, 9 months ago • MP1 init
pending->Insert(toOccur);
```

建構一個 PendingInterrupt，檢查 fromNow 是否大於 0（是否為未來會發生的事件），如果是的話就把這個 PendingInterrupt 放入 sorted 的 list 裡面等待觸發。

G. Machine::Run()

```
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 << " ==");
```

Run()會在每次呼叫完 OneInstruction()後呼叫 OneTick()，目的是在每次執行完一行指令時去檢查有沒有 interrupt 發生，並更新時間的 tick。

H. Machine::OneTick()

```
MachineStatus oldStatus = status;
```

保存目前的狀態，以便恢復數據時使用。

```
Statistics *stats = kernel->stats;
```

獲取 statistic 的指標，以便更新 tick。

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

依照目前的模式去更新 tick。

```
// 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
```

呼叫 ChangeLevel()來關掉 interrupt，這樣才能讓 interrupt handler 執行，再來呼叫 CheckIfDue()檢查是否有 PendingInterrupt 到期需要被 raise up 的，最後再呼叫 ChangeLevel()重新開啟 interrupt。

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

檢查 timer 是否有請求 context switch，有的話就直接切換過去，由於 yield 是 kernel 負責的，因此要先切換到 system mode，然後讓 CPU 切換工作，切換完後要把 status 恢復以繼續執行。

I. Interrupt::CheckIfDue()

```
ASSERT(level == IntOff);  // interrupts need to be disabled,
                          // to invoke an interrupt handler
```

確保 interrupt 有被關掉。

```
if (pending->IsEmpty()) {  // no pending interrupts
    return FALSE;
}
```

如果沒有 interrupt 在等待，則直接 return FALSE。

```
next = pending->Front();
```

讀取最快要執行的那個 interrupt。

```
if (next->when > stats->totalTicks) {
```

若還不到 interrupt 要發生的時間。

```
if (!advanceClock) {  // not time yet
    return FALSE;
```

若還沒到觸發時間則 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 spi
}
```

advanceClock 是 true 的時候表示目前來沒有準備好其他的 process，因此 system 可以把時間推進到這個 interrupt，這也會增加 idle ticks，所以也要更新 idleTicks。

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

若有 machine 的模擬器存在，則需要執行 delay load，這是 kernel 跟 hardware 之間的一個步驟。

```
inHandler = TRUE;
do {
    next = pending->RemoveFront();  // pull interrupt off list
    DEBUG(dbgTraCode, "In Interrupt::CheckIfDue, into callOnInterrupt->CallBack
    next->callOnInterrupt->CallBack();  // call the interrupt handler
    DEBUG(dbgTraCode, "In Interrupt::CheckIfDue, return from callOnInterrupt->C
    delete next;
} while (!pending->IsEmpty() && (pending->Front()->when <= stats->totalTicks));
inHandler = FALSE;
```

把 inHandler 設成 TRUE，表示正在處理 interrupt，使用 while loop 檢查 pending 中的 interrupt 是否到期，到期的話就 remove 掉並按照 next 的 interrupt type 呼叫 callOnInterrupt 的 CallBack()來處理 interrupt，最後在 interrupt 處理完之後，要把 inHandler 改回 FALSE，表示 interrupt 處理完成。

J. ConsoleOutput::CallBack()

```
putBusy = FALSE;
kernel->stats->numConsoleCharsWritten++;
callWhenDone->CallBack();
```

當一個字元輸出完成，putBusy 會被設成 FALAE，表示可以輸出下一個字元了，更新成功輸出的字元數量，接著呼叫 callWhenDone 的 CallBack()，通知系統可以輸出下一個字了。

K. SynchConsoleOutput::CallBack()

```
void SynchConsoleOutput::CallBack() {
    DEBUG(dbgTraCode, "In SynchConsol
    waitFor->V();
}
```

當輸出完成後，用 waitfor 的 V()通知系統解除 lock，可以換下一個 process 輸出了。比起 ConsoleOutput::CallBack()多了一個 lock 的機制，可以確保 multiprocess 下輸出的正確性。

IV. Makefiles

```
start.o: start.S ../userprog/syscall.h
    $(CC) $(CFLAGS) $(ASFLAGS) -c start.S

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

以 halt 舉例說明，當在終端機輸入 make halt 時，會跳到 halt 這個 label 這裡，檢查冒號後面的檔案，依序遞迴進去將每個檔案做好。

```
CFLAGS = -g -G 0 -c $(INCDIR) -B/usr/bin/
```
…

CFLAGS 是個變數，存放一些指令用來告訴編譯器在編譯過程中使用哪些選項。

- -g：告訴 compiler 產生調試信息
- -G 0：-G 通常用於 MIPS 架構的 compiler，0 表示放入 global pointer area (GPA)中的變數的大小，0 表示不做任何優化。
- -c：告訴 compiler 只編譯 source files，不進行 link 的動作，這會產生.o 檔。
- $(INCDIR)：這是一個變數，存放標頭檔的路徑，當編譯器遇到#include 時會查找這個路徑。
- -B/usr/bin/…：-B 會讓後面的路徑變成二進制的，讓 compiler 可以從這個路徑中尋找 linker 之類的工具。

```
CC = $(GCCDIR)gcc
```

CC 這個變數表示 gcc 這個程式(compiler)的路徑。

```
LD = $(GCCDIR)ld
```

LD 這個變數表示 ld 這個程式(linker)的路徑。

```
$(CC) $(CFLAGS) -c halt.c
```

這行負責把 halt.c 做成 halt.o。

```
halt: halt.o start.o
    $(LD) $(LDFLAGS) start.o halt.o -o halt.coff
    $(COFF2NOFF) halt.coff halt
```

這一段程式碼負責將 halt.o 和 start.o 鏈接起來，做成 halt.coff，.coff 檔為 mips 可以執行的檔案，但由於 linux 的架構是 x86，所以要再轉成.noff 變成可以在模擬器上面執行 mips 的檔案。

2. **Implementation of I/O System Calls in NachOS**
3. **Difficulties**
4. **Feedback**