Operating System

MP1: System Call

Team 41 分工

PartII-1 SC_Halt	郭逸洪
PartII-1 SC_Create	郭逸洪
PartII-1 SC_PrintInt	黃啓恆
PartII-1 Makefile	郭逸洪
PartII-2 Open	郭逸洪&黃啓恆
PartII-2 Write	郭逸洪&黃啓恆
PartII-2 Read	郭逸洪&黃啓恆
PartII-2 Close	郭逸洪&黃啓恆

Contents

Part II–1 Trace Code

(a) SC_Halt

(i) machine/mipssim.cc

void Machine::Run()

- 切換到UserMode
- 在無窮迴圈中呼叫Machine::OneInstruction，在執行完指令後若未exit，呼叫Interrupt::OneTick()使kernel的時間前進SystemTick（10）或UserTick（1），取決於當下是在UserMode或SystemMode

(ii) machine/mipssim.cc

void Machine::OneInstruction(Instruction *instr)

- 在program counter紀錄的記憶體位址讀取4 Bytes的指令，若失敗則直接返回，否則繼續執行
- 執行指令解碼
- 先將PC+4但不寫回，避免執行時發現是分支指令或有錯誤發生
- 根據指令的opCode找到對應的case，halt程式編譯後產生的組合語言如下，程式主要任務來自最後兩行，將r2暫存器設置為0後，呼叫SYSCALL，進到Machine::RaiseException

```
At PC = 0       JAL 68
At PC = 4       SLL r0,r0,0
At PC = 272     ADDIU r29,r29,-24
At PC = 276     SW r31,20(r29)
At PC = 280     SW r30,16(r29)
At PC = 284     ADDU r30,r29,r0
At PC = 288     JAL 65
At PC = 292     SLL r0,r0,0
At PC = 260     JR r0,r31
At PC = 264     SLL r0,r0,0
At PC = 296     JAL 5
At PC = 300     SLL r0,r0,0
At PC = 20      ADDIU r2,r0,0
At PC = 24      SYSCALL
```

(iii) machine/machine.cc

void Machine::RaiseException(ExceptionType which, int badVAddr)

- SYSCALL將BadVAddrReg暫存器的值設為0
- 切換到SystemMode
- 呼叫ExceptionHandler

(iv) userprog/exception.cc

void ExceptionHandler(ExceptionType which)

- 將r2暫存器的值（目前為0）放到type變數
- 進入which為SyscallException且type為SC_Halt（值為0）的case
- 呼叫SysHalt

(v) userprog/ksyscall.h

void SysHalt()

- 呼叫kernel interrupt的Halt方法

(vi) machine/interrupt.cc

void Interrupt::Halt()

- 印出停止訊息及kernel狀態統計
- 刪除kernel
- 在kernel解構式中刪除各項物件、資源
- exit 0

(b) SC_Create

(i) userprog/exception.cc

void ExceptionHandler(ExceptionType which)

- createFile程式主要由以下組合語言執行檔案創建，呼叫SYSCALL時r2暫存器值為4、r4暫存器值為400，故ExceptionHandler中的type變數為4、val變數為400

```
At PC = 296     LUI r4,0
At PC = 300     ADDIU r4,r4,400
At PC = 304     JAL 37
At PC = 308     SLL r0,r0,0
At PC = 148     ADDIU r2,r0,4
At PC = 152     SYSCALL
```

- 進入which為SyscallException且type為SC_Create（值為4）的case
- 從記憶體位置為400處讀取檔案名稱
- 呼叫SysCreate並傳入檔案名稱，回傳是否成功的狀態
- SysCreate結束後，將狀態值寫回r2暫存器，此值將影響印出訊息的分支
- 更新PC並返回

(ii) userprog/ksyscall.h

int SysCreate(char *filename)

- 呼叫kernel fileSystem的Create方法

(iii) filesys/filesys.h

bool Create(char *name)

- 呼叫sysdep.cc的OpenForWrite
- 若OpenForWrite回傳的file descriptor為-1則回傳失敗
- 否則呼叫sysdep.cc的Close關閉該fd
- 回傳成功

(iv) lib/sysdep.cc

int OpenForWrite(char *name)

- 呼叫standard libraries的open在host UNIX的檔案系統上打開檔案，若檔案不存在則創建，若存在則截切原文件
- 回傳file descriptor

(v) lib/sysdep.cc

int Close(int fd)

- 呼叫standard libraries的close關閉fd

(c) SC_PrintInt

(i) userprog/exception.cc

void ExceptionHandler(ExceptionType which)

- add程式主要由以下組合語言執行PrintInt，呼叫SYSCALL時r2暫存器值為16、r4暫存器為要印出的值，故ExceptionHandler中的type變數為16、val變數為先前add的計算結果65

```
At PC = 76      JR r0,r31
At PC = 80      SLL r0,r0,0
At PC = 312     SW r2,16(r30)
At PC = 316     LW r4,16(r30)
At PC = 320     JAL 9
At PC = 324     SLL r0,r0,0
At PC = 36      ADDIU r2,r0,16
At PC = 40      SYSCALL
```

- 進入which為SyscallException且type為SC_PrintInt（值為16）的case
- 將r4暫存器的值放到val中並將其傳入SysPrintInt()中

(ii) userprog/ksyscall.h

void SysPrintInt(int val)

- 呼叫SynchConsoleOutput::PutInt，並傳入val

(iii) userprog/synchconsole.cc

void SynchConsoleOutput::PutInt(int value)

- 為了簡化實作，這邊在印出前先將int轉換為字串（字元陣列），並加上換行符號，此時字串實質內容為"65\n"
- 將SynchConsoleOutput的lock鎖上，同時間只允許一個執行緒持有該鎖，避免output錯誤，其餘執行緒只能停在此處等持續該鎖的執行緒釋放該鎖
- 接下來在迴圈中呼叫ConsoleOutput::PutChar直到印完要輸出的字元陣列，由以下add程式輸出的除錯訊息可知，在同一個時刻下（此除錯訊息是在totalTicks為73時），Nachos模擬的IO裝置在ConsoleOutput::PutChar中schedule一個100 ticks後要執行的interrupt，且程式呼叫SynchConsoleOutput::waitFor的P()方法，禁用interrupt並將add這個執行緒轉為sleep狀態，直到100 ticks後，由模擬IO裝置觸發的callback將其喚醒

```
In SynchConsoleOutput::PutChar, into
consoleOutput->PutChar, 73
6Scheduling interrupt handler the console write at time
= 173
In SynchConsoleOutput::PutChar, return from
consoleOutput->PutChar, 73
In SynchConsoleOutput::PutChar, into waitFor->P(), 73
In Semaphore::P(), 73
        interrupts: on -> off
Sleeping thread: add
```

- 100 ticks後（此除錯訊息在totalTicks為173時）SynchConsoleOutput::waitFor的V()因callback而被呼叫，Nachos將add的執行緒放到ready list，之後被CPU執行，成功輸出一個字元，並重新開啟interrupt

```
Invoking interrupt handler for the
console write at time 173
In Interrupt::CheckIfDue, into
callOnInterrupt->CallBack, 173
In ConsoleOutput::CallBack(), 173
In SynchConsoleOutput::CallBack(), 173
In Semaphore::V(), 173
        interrupts: off -> off
Putting thread on ready list: add
        interrupts: off -> off
In Interrupt::CheckIfDue, return from
callOnInterrupt->CallBack, 173
In Interrupt::Idle, return true from CheckIfDue, 173
```

```
Switching from: add to: add
Now in thread: add
        interrupts: off -> on
```

(iv) machine/console.cc

void ConsoleOutput::PutChar(char ch)

- 模擬的IO裝置判斷putBusy，確保沒有其他的寫入操作正在執行
- 若沒有在執行中就可以進行寫入file
- 將putBusy設成TRUE
- 排程一個100 ticks（ConsoleTime）後的ConsoleWriteInt，該interrupt執行完後會呼叫ConsoleOutput::CallBack()

(v) machine/interrupt.cc

void Interrupt::Schedule(CallBackObj *toCall, int fromNow, IntType type)

- 宣告when這個變數用以表示該 interrupt應該發生的時間點
- 宣告一個PendingInterrupt物件toOccur，這個物件代表即將發生的interrupt，記錄著發生interrput時需要回呼的對象、時間及interrupt的類型
- 確fromNow大於0，因為若等於0就表示當下立刻就要執行中斷，但我們並不知道當前是否有中斷正在執行，以此避免衝突
- 將toOccur插入pending這個SortedList中並等待至該中斷應發生的時刻

(vi) machine/mipssim.cc

void Machine::Run()

- 在無窮迴圈中執行Machine::OneInstruction，執行指令後若未exit，則呼叫Interrupt::OneTick()

(vii) machine/interrupt.cc

void Interrupt::OneTick()

- 使kernel的時間前進SystemTick（10）或UserTick（1），取決於當下是在UserMode或SystemMode
- 為了要去檢查是否當前有需要執行的interrupts，先將interrupt關掉，避免處理interrupts時又有其他interrupts產生
- 進入CheckIfDue去檢查是否目前有需要執行的的pending interrupt
- 檢查完後再透過ChangeLevel(IntOn, IntOff)重新打開interrupts
- 如果目前可以進行context switch（yieldOnReturn為TRUE），則將yieldOnReturn設為FALSE，進入SystemMode後呼叫kernel->currentThread->Yield()，當此執行緒又可以執行時，則恢Interrupt::status

(viii) machine/interrupt.cc

bool Interrupt::CheckIfDue(bool advanceClock)

- 確認目前處於關閉interrupt的狀態
- 若有設定interrupt除錯旗標，將呼叫DumpState()印出目前pending interrupts
- 透過next這個指標指向在pending interrupts中最接近目前時間的interrput，如果next的發生時間還沒達到，使用advanceClock判斷目前需不需要將totakTicks往後移到next指向的interrupt的發生時間，若不需要，則回傳FALSE，表示沒有任何的interrupt handler被啟動
- 若next的發生時間已經到達，且kernel->machine未被移除，則呼叫Machine::DelayedLoad模擬硬體的delayed load effect
- 將inHandler設成TRUE表示正在處理interrupt
- 取出pending interrupts中最前面的interrupt，並呼叫對應的interrupt handler（callback function）進行處理，直到pending interrupts為空或是下一個interrupt發生時間大於目前的時間
- 做完後將inHandler設為FALSE表示處理完interrupts
- 回傳TRUE表示在呼叫本次CheckIfDue中至少處理一個interrupt

(ix) machine/console.cc

void ConsoleOutput::CallBack()

- 寫入動作結束並將putBusy設為FALSE
- 將console印出字元數統計量+1
- 呼叫SynchConsoleOutput::CallBack()

(x) userprog/synchconsole.cc

void SynchConsoleOutput::CallBack()

- 呼叫SynchConsoleOutput::waitFor的V()方法
- 禁用interrupt
- 若等待SynchConsoleOutput::waitFor的thread queue不為空，則取出最前面的thread，將其設為READY狀態，從以下add程式輸出的除錯訊息可以看到add執行緒被改回READY狀態，隨時準備再被CPU執行

```
In ConsoleOutput::CallBack(), 283
In SynchConsoleOutput::CallBack(), 283
In Semaphore::V(), 283
        interrupts: off -> off
Putting thread on ready list: add
        interrupts: off -> off
```

- 恢復interrupt level

(d) Makefile (code/test/Makefile)

```
include Makefile.dep

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

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/
```

- Makefile.dep包含所有依賴機器的定義
- CC、AS、LD分別指Compiler、Assembler及Linker
- INCDIR指出Header檔的目錄
- CFLAGS中–g表示保留除錯訊息，–c表示生成.o檔而不經由Linker連結成可執行檔，–B表示必須包含這些目錄下的binary檔或library[1]

```
ifeq ($(hosttype),unknown)
PROGRAMS = unknownhost
else
# change this if you create a new test program!
#PROGRAMS = add halt shell matmult sort segments test1 test2 a
PROGRAMS = add halt createFile fileIO_test1 fileIO_test2
LotOfAdd
endif

all: $(PROGRAMS)
```

- 根據host機器類型定義PROGRAMS變量
- 定義Makefile預設目標all的依賴為PROGRAMS變量中的內容

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

- 定義start.o目標依賴於start.S組合語言檔及syscall.h標頭檔
- 定義Compiler如何將start.S組合語言檔編譯為start.o檔，其中–c表示生成.o檔而不經由Linker連結成可執行檔

```
halt.o: halt.c
    $(CC) $(CFLAGS) -c halt.c
halt: halt.o start.o
```

[1] https://gcc.gnu.org/onlinedocs/gcc/Option-Index.html

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

- 定義halt.o由halt.c編譯而成
- 定義可執行檔halt依賴於halt.o及start.o
- 定義Linker將halt.o及start.o連結為halt.coff檔，又因為Nachos預設啟動程序（startup routine）位於0，所以第一個被傳入Linker的.o檔必須為start.o
- 使用COFF2NOFF將halt.coff轉為noff格式的Nachos可執行檔halt
- 其餘程式編譯步驟與halt相似，都是先編譯為.o檔，由Linker連結為.coff檔，再轉換成noff格式檔

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

- make clean將會清除這些編譯生成的臨時檔

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

- make distclean將會清除這些Nachos可執行檔

```
unknownhost:
    @echo Host type could not be determined.
    @echo make is terminating.
    @echo If you are on an MFCF machine, contact the instructor
to report this problem
    @echo Otherwise, edit Makefile.dep and try again.
```

- 若host機器類型未知，印出終止訊息

Part II–2 I/O System Calls Implementation

我們在userprog/syscall.h定義SC_Open、SC_Read、SC_Write及SC_Close等新的system calls的列舉值，在test/start.S以組合語言定義對應的system call stubs，最後我們在userprog/exception.cc新增對應上述system calls的cases，以上system calls的操作十分相似，大致分為以下幾個步驟：

1. 讀入所需的arguments（存放在4至7號暫存器）
2. 呼叫對應的system call
3. 將system call回傳的結果寫回2號暫存器
4. 更新program counter

接下來將針對每個system call說明其中的實作細節

(a) OpenFileId Open(char *name)

(i) SysOpen實作主要依靠FileSystem::OpenAFile方法，呼叫時傳入檔案名稱

```
OpenFileId OpenAFile(char *name)
{
    OpenFileId id;
    for (id = 0; id < 20; ++id)
    {
        if (OpenFileTable[id] == NULL)
        {
            break;
        }
    }
    // exceed the opened file limit
    if (id == 20)
    {
        return -1;
    }
    // call UNIX open and check
    int fileDescriptor = OpenForReadWrite(name, FALSE);
    if (fileDescriptor == -1)
    {
        return -1;
    }
    OpenFileTable[id] = new OpenFile(fileDescriptor);
    return id;
}
```

(ii) 先檢查OpenFileTable是否有可用的entry，若無則回傳–1

(iii) 透過sysdep.cc的OpenForReadWrite方法，在host的UNIX機器上以讀寫模式打開檔案並回傳file descriptor[2]

(iv) 檢查file descriptor是否為-1，若是則開檔失敗，回傳-1

(v) 若file descriptor不是-1，開檔成功，建立新的OpenFile entry並放入OpenFileTable中

(vi) 回傳該OpenFile entry在OpenFileTable中的索引值

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

(i) SysWrite實作主要依靠FileSystem::WriteFile_方法，呼叫時傳入寫入來源buffer、寫入大小及欲寫入的OpenFileId

```
int WriteFile_(char *buffer, int size, OpenFileId id)
{
    if (buffer != NULL
        && size >= 0
        && is_valid_file_id(id))
    {
        return OpenFileTable[id]->Write(buffer, size);
    }
    return -1;
}

bool is_valid_file_id(OpenFileId id)
{
    return id >= 0 && id < 20 && OpenFileTable[id] != NULL;
}
```

(ii) 檢查buffer、OpenFileId及寫入大小，OpenFileId需要在合法範圍內，且OpenFileTable確實已經開啟該檔，否則返回-1表示非法存取

(iii) 針對該OpenFile entry呼叫OpenFile::Write方法，透過sysdep.cc的Lseek[3]及WriteFile[4]方法將檔案寫入host的UNIX機器，此操作後，OpenFile的currentOffset將被更新

(iv) 成功返回寫入的大小

[2] https://man7.org/linux/man-pages/man2/open.2.html
[3] https://man7.org/linux/man-pages/man2/lseek.2.html
[4] https://man7.org/linux/man-pages/man2/write.2.html

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

(i) SysRead實作主要依靠FileSystem::ReadFile方法，呼叫時傳入讀取目的buffer、讀取大小及欲讀取的OpenFileId

```
int ReadFile(char *buffer, int size, OpenFileId id)
{
    if (buffer != NULL
        && size >= 0
        && is_valid_file_id(id))
    {
        return OpenFileTable[id]->Read(buffer, size);
    }
    return -1;
}
```

(ii) 檢查buffer、OpenFileId及讀取大小，OpenFileId需要在合法範圍內，且OpenFileTable確實已經開啟該檔，否則返回–1表示非法存取

(iii) 針對該OpenFile entry呼叫OpenFile::Read方法，透過sysdep.cc的Lseek及ReadPartial[5]方法從host的UNIX機器讀取檔案，並將讀取內容寫入目的buffer中，此操作後，OpenFile的currentOffset將被更新

(iv) 成功返回讀取的大小

(d) int Close(OpenFileId id)

(i) SysClose實作主要依靠FileSystem::CloseFile方法，呼叫時傳入欲關閉的OpenFileId

```
int CloseFile(OpenFileId id)
{
    if (is_valid_file_id(id))
    {
        delete OpenFileTable[id];
        OpenFileTable[id] = NULL;
        return 1;
    }
    return -1;
}
```

(ii) 同樣先檢查OpenFileId是否在合法範圍內，且OpenFileTable確實已經開啟該檔，否則返回–1表示非法存取

[5] https://man7.org/linux/man-pages/man2/read.2.html

(iii) 釋放該OpenFileId在OpenFileTable中所占用的entry，OpenFile的解構式呼叫sysdep.cc的Close[6]方法，在host的UNIX機器上關閉檔案

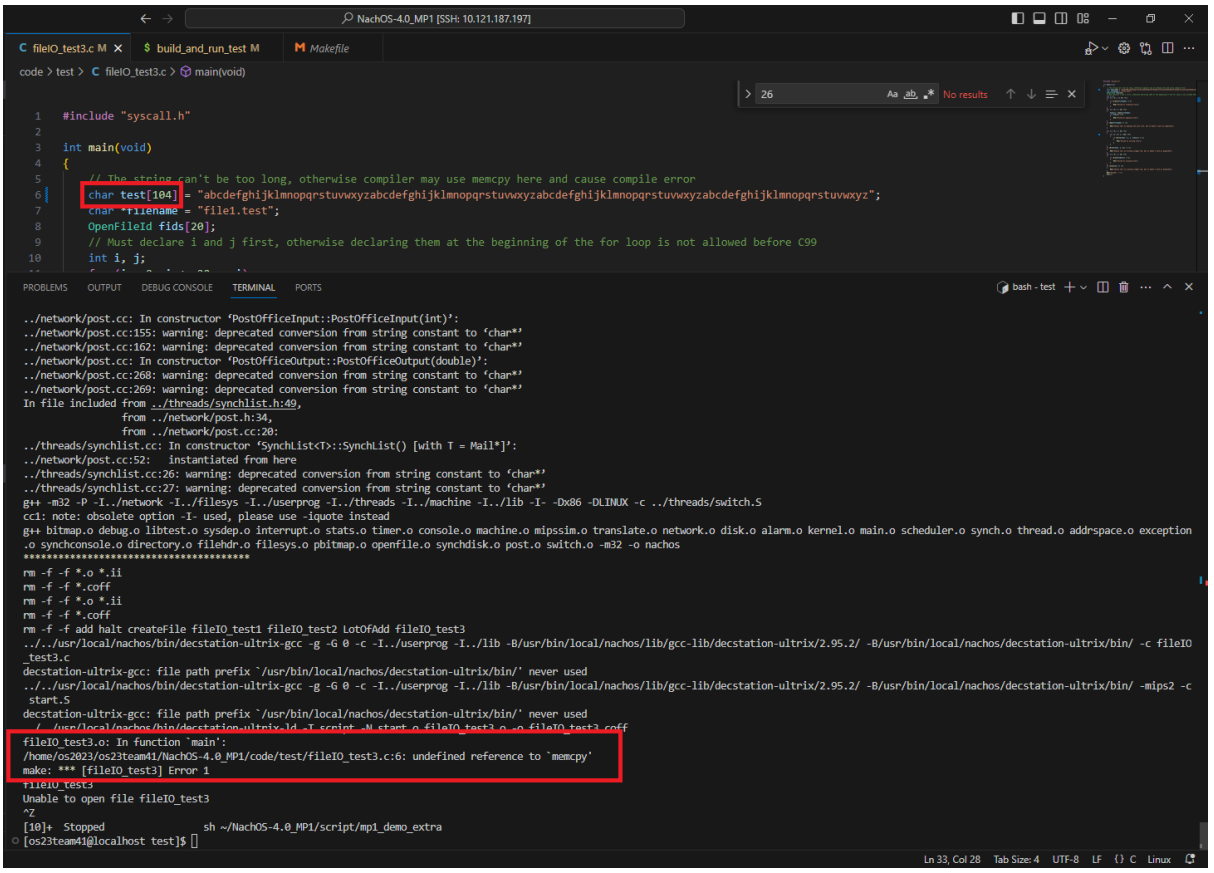
(iv) 將該OpenFileId在OpenFileTable的位置設為NULL

(v) 返回1表示成功

[6] https://man7.org/linux/man-pages/man2/close.2.html

Difficulties encountered

- 在撰寫新的測試案例時，發現若是將IO test的字串長度改成原有案例的4倍（104個字元），編譯時會出現如下圖所示的undefined reference to 'memcpy'錯誤，似乎gcc在編譯時，有機會新增對memcpy的調用



- 在撰寫新的測試案例時，發現若是在同一程式對同一個OpenFileId寫入後再讀取會出現錯誤，經過除錯後發現是同一個OpenFile物件的currentOffset在寫入後被更新了，導致後續也從這個currentOffset的位置開始讀取，因為目前沒有能指定從哪個位置開始讀取或是能重置OpenFile currentOffset的syscall，只能在寫入後先將檔案關閉，再次打開進行讀取

Feedback

- 建議compile時可以指定–std選項，或是提醒同學撰寫測試程式時不要用過新的C語言語法，我在撰寫測試程式時，花了一些時間除錯才發現for loop開頭宣告變數是C99之後才允許的[7]

[7] https://stackoverflow.com/questions/1287863/c-for-loop-int-initial-declaration