CS342301: Operating System

MP1: System Call

- Group6 -

I. Team members and contributions

Trace code

Working items	Members
1. SC_Halt	許安嫰
2. SC_Create	許安嫰
3. SC_PrintInt	詹詠絜
4. Makefile	詹詠絜

Implementation

Working items	Members: Coding & Report
1. OpenFileId OpenAFile(char *name);	許安嫰
2. int Write(char *buffer, int size, OpenFileId id);	詹詠絜
3. int Read(char *buffer, int size, OpenFileId id);	詹詠絜
4. int Close(OpenFileId id);	許安嫰

II. Trace code

(a.) **`SC_Halt`** system call –

When a user program calls the system call `SC_Halt`, NachOS will find the corresponding system call stub in start.S. It then stores the system call code into register `r2` and processes the `syscall` instruction.

```
45        .globl Halt
46        .ent    Halt
47    Halt:
48        addiu $2,$0,SC_Halt
49        syscall
50        j   $31
51        .end Halt
```

machine/mipssim.cc

Machine::Run()

Machine::OneInstruction()

When a program starts up, the kernel will call `Machine::Run()`.

```
54  void Machine::Run() {
55      Instruction *instr = new Instruction;  // storage for decoded instruction
56      if (debug->IsEnabled('m')) {
57          cout << "Starting program in thread: " << kernel->currentThread->getName();
58          cout << ", at time: " << kernel->stats->totalTicks << "\n";
59      }
60      kernel->interrupt->setStatus(UserMode);
```

`L60` Transfer control to the user mode from the kernel mode.

```
61      for (;;) {
62          DEBUG(dbgTraCode, "In Machine::Run(), into OneInstruction "
63                              << "== Tick " << kernel->stats->totalTicks << " ==");
64          OneInstruction(instr);
65          DEBUG(dbgTraCode, "In Machine::Run(), return from OneInstruction  "
66                              << "== Tick " << kernel->stats->totalTicks << " ==");
67
68          DEBUG(dbgTraCode, "In Machine::Run(), into OneTick "
69                              << "== Tick " << kernel->stats->totalTicks << " ==");
70          kernel->interrupt->OneTick();
71          DEBUG(dbgTraCode, "In Machine::Run(), return from OneTick "
72                              << "== Tick " << kernel->stats->totalTicks << " ==");
73          if (singleStep && (runUntilTime <= kernel->stats->totalTicks))
74              Debugger();
75      }
```

`L61` Create an infinite loop.

L64 Execute instructions by calling `Machine::OneInstruction()`.

L70 Advance the simulated time.

machine/mipssim.cc

Machine::Run()

Machine::OneInstruction()

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

L133、L136 Fetch and decode the instruction.

```
154     // Execute the instruction (cf. Kane's book)
155     switch (instr->opCode) {
```

L155 Execute the instruction (which is `OP_SYSCALL` in this case).

```
663           case OP_SYSCALL:
664               DEBUG(dbgTraCode, "In Machine::OneInstruction, RaiseException(SyscallEx
665               RaiseException(SyscallException, 0);
666               return;
```

L665 Call `Machine::RaiseException()`, and pass the exception type `SyscallException` as argument.

machine/machine.cc

Machine::RaiseException()

```
97  void Machine::RaiseException(ExceptionType which, int badVAddr) {
98      DEBUG(dbgMach, "Exception: " << exceptionNames[which]);
99      registers[BadVAddrReg] = badVAddr;
100     DelayedLoad(0, 0);  // finish anything in progress
101     kernel->interrupt->setStatus(SystemMode);
102     ExceptionHandler(which);  // interrupts are enabled at this point
103     kernel->interrupt->setStatus(UserMode);
104 }
```

L101 Transfer control to the kernel mode.

L102 Call `ExceptionHandler()`.

L103 Switch back to user mode.

userprog/exeception.cc

ExceptionHandler()

```
60                  case SC_Halt:
61                      DEBUG(dbgSys, "Shutdown, initiated by user program.\n");
62                      SysHalt();
63                      cout << "in exception\n";
64                      ASSERTNOTREACHED();
65                      break;
```

L62 Go to the system call interface by calling `SysHalt()`.

userprog/ksyscall.h

SysHalt()

```
17  void SysHalt() {
18      kernel->interrupt->Halt();
19  }
```

L18 Call `Interrupt::Halt()` to shut down NachOS.

machine/interrupt.cc

Interrupt::Halt()

```
228 void Interrupt::Halt() {
229 #ifndef NO_HALT_STAT
230     cout << "Machine halting!\n\n";
231     cout << "This is halt\n";
232     kernel->stats->Print();
233 #endif
234     delete kernel;  // Never returns.
235 }
```

L234 Delete the kernel to shut down NachOS.

(b.) **`SC_Create`** system call –

Similar to `SC_Halt`, when a program calls the system call `SC_Create`, the system call stub in start.S will store its system call code into register `r2`.

```
        .globl Create
        .ent    Create
Create:
        addiu $2,$0,SC_Create
        syscall
        j   $31
        .end Create
```

userprog/exeception.cc

ExceptionHandler()

```
89              case SC_Create:
90                  val = kernel->machine->ReadRegister(4);
91                  {
92                      char *filename = &(kernel->machine->mainMemory[val]);
93                      // cout << filename << endl;
```

`L90` Read the address stored in register `r4`.

`L92` Obtain the file name from main memory and store its base address in pointer `filename`.

```
94                      status = SysCreate(filename);
95                      kernel->machine->WriteRegister(2, (int)status);
96                  }
```

`L94` Go to the system call interface by calling `SysCreate()`, and pass the file name as argument.

`L95` Store its return value into register `r2`.

```
97                  kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
98                  kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) + 4);
99                  kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg) + 4);
100                 return;
101                 ASSERTNOTREACHED();
102                 break;
```

`L97-99` Update PC.

userprog/ksyscall.h

SysCreate()

```
31  int SysCreate(char *filename) {
32      // return value
33      // 1: success
34      // 0: failed
35      return kernel->fileSystem->Create(filename);
36  }
```

`L35` Go to the file system by calling `FileSystem::Create()`.

filesys/filesys.h

FileSystem::Create()

```
52      bool Create(char *name) {
53          int fileDescriptor = OpenForWrite(name);
54
55          if (fileDescriptor == -1)
56              return FALSE;
57          Close(fileDescriptor);
58          return TRUE;
59      }
```

`L53` Call `OpenForWrite()`.

`[補充]` `FileDescripter`: It is a nonnegative integer number that uniquely represents an opened file for the process (They are bound to a process ID).

`L55-56` If failed to open, return `FALSE`.

`L57-58` Otherwise, call `Close()` and return `TRUE`.

(c.) **`SC_PrintInt`** system call –

How NachOS implements asynchronized I/O using CallBack functions and register schedule events?

userprog/exeception.cc

ExceptionHandler()

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

L53 將 system call code (stored in r2) 存入 type.

```
66  case SC_PrintInt:
67      DEBUG(dbgSys, "Print Int\n");
68      val = kernel->machine->ReadRegister(4);
69      DEBUG(dbgTraCode, "In ExceptionHandler(), into SysPrintInt, " <<
70      SysPrintInt(val);
```

L66 如果 type == SC_PrintInt，則...

L68 將 arg1 (stored in r4) 存入 val

由此可知，user program 中的參數是透過 register 傳遞

L70 呼叫 SysPrintInt()並傳入參數 val

userprog/ksyscall.h

SysPrintInt()

```
21  void SysPrintInt(int val) {
22      DEBUG(dbgTraCode, "In ksyscall.h:SysPrintInt, into synchConsole
23      kernel->synchConsoleOut->PutInt(val);
24      DEBUG(dbgTraCode, "In ksyscall.h:SysPrintInt, return from synch
25  }
```

L23 呼叫 PutInt()並傳入參數 val

userprog/synchconsole.cc

SynchConsoleOutput::PutInt()
SynchConsoleOutput::PutChar()

```
100 void SynchConsoleOutput::PutInt(int value) {
101     char str[15];
102     int idx = 0;
103     // sprintf(str, "%d\n\0", value);  the true one
104     sprintf(str, "%d\n\0", value);  // simply for trace code
105     lock->Acquire();
106     do {
107         DEBUG(dbgTraCode, "In SynchConsoleOutput::PutChar, into consoleOut
108         consoleOutput->PutChar(str[idx]);
109         DEBUG(dbgTraCode, "In SynchConsoleOutput::PutChar, return from con
110         idx++;
111 
112         DEBUG(dbgTraCode, "In SynchConsoleOutput::PutChar, into waitFor->P
113         waitFor->P();
114         DEBUG(dbgTraCode, "In SynchConsoleOutput::PutChar, return form  wa
115     } while (str[idx] != '\0');
116     lock->Release();
117 }
```

L104 將型別原來是整數的 val，透過 sprintf()將其格式化輸出成字串，並存入 str.

L105、L116 利用 Lock 的機制來確保資源一次只給一個 thread 運行

L106-L115 do-while loop 中，str 的每個字元都會呼叫 PutChar()，並等待該字元的輸出完成後，才會繼續處理下一個字元

L113 waitFor 是一種 semaphore，會使用一個數字來表示「允須多少程式同時執行 CS(Critical Section)」，根據 threads/synch.h 裡的定義，P()透過整數變數 value(允許同時執行的 threads 的數量)判斷當下是否可以輸出字元(value>0)，準備輸出前，再將 value-1

```
93 void SynchConsoleOutput::PutChar(char ch) {
94     lock->Acquire();
95     consoleOutput->PutChar(ch);
96     waitFor->P();
97     lock->Release();
98 }
```

L93-L98 處理單一字元的輸出，其中呼叫的 consoleOutput->PutChar()是將該字元 ch 實際寫入螢幕或其他裝置

[觀念]

「同步」在 SynchConsoleOutput::PutInt()與 SynchConsoleOutput::PutChar()的目的：

- 讓資源在多個 threads 之間被正確地使用，避免競爭條件(Race Condition)
- 保證輸出順序無誤，特別是在涉及多個 threads 輸出字元的情況下，讓結果能夠按預期的順序顯示出來

machine/console.cc

ConsoleOutput::PutChar()

```
154 void ConsoleOutput::PutChar(char ch) {
155     ASSERT(putBusy == FALSE);
156     WriteFile(writeFileNo, &ch, sizeof(char));
157     putBusy = TRUE;
158     kernel->interrupt->Schedule(this, ConsoleTime, ConsoleWriteInt);
159 }
```

L156 確認目前的輸出操作不處於忙碌狀態後，呼叫 WriteFile()將字元寫入可以模擬顯示器的檔案 writeFileNo

L158 使用 Schedule()設置一個 timer，當指定的時間到達時會觸發指定的中斷處理程序 ConsoleWriteInt 並直接返回，不會等待該字元的輸出實際完成

[觀念] 這種非同步操作允許程式在等待設備輸出的同時繼續執行其他操作

machine/interrupt.cc

Interrupt::Schedule()

```
289 void Interrupt::Schedule(CallBackObj *toCall, int fromNow, IntType type) {
290     int when = kernel->stats->totalTicks + fromNow;
291     PendingInterrupt *toOccur = new PendingInterrupt(toCall, when, type);
292 
293     DEBUG(dbgInt, "Scheduling interrupt handler the " << intTypeNames[type]
294     ASSERT(fromNow > 0);
295 
296     pending->Insert(toOccur);
297 }
```

L290 計算何時 trigger interrupt:

When = totalTicks(系統目前運作的總時間) + fromNow(多少 ticks 後開始處理中斷)

`L291` 建立一個待處理的 interrupt object `toOccur`，同時宣告它的類型與觸發時刻，再將它加入 sorted pending list 中，等待未來被觸發

machine/mipssim.cc

Machine::Run()

```
54  void Machine::Run() {
55      Instruction *instr = new Instruction;  // storage for decoded instruction
56      if (debug->IsEnabled('m')) {
57          cout << "Starting program in thread: " << kernel->currentThread->getNa
58          cout << ", at time: " << kernel->stats->totalTicks << "\n";
59      }
60      kernel->interrupt->setStatus(UserMode);
```

`L55` 創建一個新的 instruction object `instr`，用來存解碼後的 MIPS 指令

`L60` 從 kernel mode 切換到 user mode，系統已準備好執行 user program

```
61      for (;;) {
62          DEBUG(dbgTraCode, "In Machine::Run(), into OneInstruction "
63                      << "== Tick " << kernel->stats->totalTi
64          OneInstruction(instr);
65          DEBUG(dbgTraCode, "In Machine::Run(), return from OneInstruct
66                      << "== Tick " << kernel->stats->totalTi
67
68          DEBUG(dbgTraCode, "In Machine::Run(), into OneTick "
69                      << "== Tick " << kernel->stats->totalTi
70          kernel->interrupt->OneTick();
71          DEBUG(dbgTraCode, "In Machine::Run(), return from OneTick "
72                      << "== Tick " << kernel->stats->totalTi
```

`L64` 進入無限 for loop 後，呼叫 `OneInstruction()`，此時系統會從記憶體中讀取一條指令並執行它 (user level)

`L70` 呼叫 `OneTick()`讓系統時間前進一個 Tick

machine/interrupt.cc

Machine::OneTick()

```
159     // check any pending interrupts are now ready to fire
160     ChangeLevel(IntOn, IntOff);  // first, turn off interrupts
161                                  // (interrupt handlers run with
162                                  // interrupts disabled)
163     CheckIfDue(FALSE);           // check for pending interrupts
164     ChangeLevel(IntOff, IntOn);  // re-enable interrupts
```

`L160` 呼叫 `ChangeLevel(IntOn, IntOff)` 暫時 disable interrupt，將第二個參數(current state)設為 `IntOff(0)`，避免受到其他 interrupts 干擾

`L163` 呼叫 `CheckIfDue()`檢查 ready queue 中是否有準備好的 interrupt 可以處理，將參數設為 `FALSE` 表示如果還沒到要處理 interrupt 的時間，不會主動推進時間

`L164` 再次呼叫 `changeLevel(IntOff, IntOn)`允許接受其他 interrupt

```
165     if (yieldOnReturn) {          // if the timer device handler asked
166                                   // for a context switch, ok to do it now
167         yieldOnReturn = FALSE;
168         status = SystemMode;  // yield is a kernel routine
169         kernel->currentThread->Yield();
170         status = oldStatus;
171     }
```

`L169` `Yield` 函數透過 scheduler 尋找下一個 thread，並執行它(context switch)

`L170` Reload old interrupt status.

machine/interrupt.cc
Interrupt::CheckIfDue()

```
343     do {
344         next = pending->RemoveFront();  // pull interrupt off list
345         DEBUG(dbgTraCode, "In Interrupt::CheckIfDue, into callOnInterrupt->CallBac
346         next->callOnInterrupt->CallBack();  // call the interrupt handler
347         DEBUG(dbgTraCode, "In Interrupt::CheckIfDue, return from callOnInterrupt->
348         delete next;
349     } while (!pending->IsEmpty() && (pending->Front()->when <= stats->totalTicks))
```

`L344` 等待當初安排執行 interrupt 的時間(`when`)到，將 sorted pending list 裡的第一個 interrupt 存入 pending interrupt object `next`

`L346` 呼叫 `CallBack()`，這個函數本身是 interrupt handler，也就是負責完成 `ConsoleWriteInt` 這個 interrupt type，這時會回到當初處理 I/O 的地方並輸出一個字元

machine/console.cc

ConsoleOutput::CallBack()

```
141 void ConsoleOutput::CallBack() {
142     DEBUG(dbgTraCode, "In ConsoleOutput::CallBack(), " << kernel->stats->
143     putBusy = FALSE;
144     kernel->stats->numConsoleCharsWritten++;
145     callWhenDone->CallBack();
146 }
```

`L143` 表示設備不再忙碌，已結束 `PutChar` operation 且能夠準備處理下一個字元的輸出

`L145` 通知 kernel 中斷已完成，表示該字元已經成功輸出到螢幕上，再呼叫 `SynchConsoleOutput::CallBack()`

userprog/synchconsole.cc

SynchConsoleOutput::CallBack()

```
125 void SynchConsoleOutput::CallBack() {
126     DEBUG(dbgTraCode, "In SynchConsoleOutput::CallBack(), " << kernel
127     waitFor->V();
128 }
```

`L127` 呼叫 `waitFor` 的 `V()`，將 `value+1`，透過傳送 signal 讓等待中的 thread 可以繼續執行

(d). Trace the **Makefile**: 以 `halt.c` 進行說明

```
120 start.o: start.S ../userprog/syscall.h
121     $(CC) $(CFLAGS) $(ASFLAGS) -c start.S
122 
123 halt.o: halt.c
124     $(CC) $(CFLAGS) -c halt.c
125 halt: halt.o start.o
126     $(LD) $(LDFLAGS) start.o halt.o -o halt.coff
127     $(COFF2NOFF) halt.coff halt
```

`L120`、`L121`

1. 在當前 `test` 目錄中找到 `start.S`，它負責初始化 user program 的運行環境、定義 system calls，讓 processes 可以與 `NachOS` 內核進行互動。
2. 當兩的 dependencies (`start.S`, `syscall.h`) 都找到後，依照 compiler 和 assembler 的

選項生成 **start.o**

L123 在當前 test 目錄中找到 halt.c

L124 依照 compiler 的選項生成 **halt.o**

L125 兩個 dependencies (**start.o**, **halt.o**) 皆滿足，可以開始進行真正的 command

L126 依照 linker 的選項生成 halt.coff

L127 將 coff 格式轉換為 noff 格式，兩者是不同的二進制文件格式，具體用於不同的系統（e.g., NachOS），最後生成 halt 的可執行檔

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

每次修改程式後都需要 make clean 的原因是為了避免沒有成功編譯後的檔案，主要移除所有中間檔案，包含.o/.ii/.coff 檔

III. Implementation

1. userprog/syscall.h

```
27    #define SC_Open 6
28    #define SC_Read 7
29    #define SC_Write 8
30    #define SC_Seek 9
31    #define SC_Close 10
```

`L27`、`L28`、`L29`、`L31` 刪除這四個 system call code 的註解，讓 kernel 知道 user program 當前需要執行哪一種 system call

2. test/start.S

新增 `Open, Read, Write, Close` 共四個 system call stubs，格式皆相同，下方以 `Open` 進行說明

```
165    //---------------- MP1 ----------------//
166        .globl Open
167        .ent    Open
168    Open:
169        addiu $2,$0,SC_Open
170        syscall
171        j   $31
172        .end Open
```

`L166` 將 `Open` 設為全域的

`L167` 表示 `Open` 是一個可呼叫的函數(entry point)

`L169` 將 `Open` 的 system call code 存入 `r2`

`L170` 通知系統執行 `Open` 的操作

`L171` 當函數執行完成後，返回當初呼叫它的地方，return address 存在 `r31`

3. userprog/exception.cc

新增 `SC_Open, SC_Write, SC_Read, SC_Close` 的 cases 在 `ExceptionHandler()`之下

(a.) `SC_Open`

```
139     case SC_Open:
140         // Load arguments from user program
141         val = kernel->machine->ReadRegister(4);
142 
143         // Process SysOpen Systemcall
144         filename = &(kernel->machine->mainMemory[val]);
145         //cout << "filename: " << filename << endl;
146         fileID = SysOpen(filename);
147         kernel->machine->WriteRegister(2, (int)fileID);
148 
149         // Set program counter
150         kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
151         kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) + 4);
152         kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg) + 4);
153         return;
154         ASSERTNOTREACHED();
155         break;
```

`L141` Read the address stored in register `r4`.

`L144` Obtain the `filename` from main memory.

`L146` Call `SysOpen()` and pass the `filename` as argument.

`L147` Store the returned `fileID` into register `r2` .

`L150-152` Update PC.

(b.) SC_Write

```
135 case SC_Write:
136     // Load arguments from user program
137     val = kernel->machine->ReadRegister(4);
138     numChar = kernel->machine->ReadRegister(5);
139     fileID = kernel->machine->ReadRegister(6);
140 
141     // Process SysWrite Systemcall
142     char *buffer = &(kernel->machine->mainMemory[val]);
143     int numWritten = SysWrite(buffer, numChar, fileID);
144 
145     // Prepare Result
146     kernel->machine->WriteRegister(2, (int)numWritten);
147 
148     // Set program counter
149     kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
150     kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) + 4);
151     kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg) +
152     return;
153     ASSERTNOTREACHED();
154     break;
```

`L137-L139` 分別從 r4, r5, r6 中找到 val(寫入資料的存放記憶體位置)、numChar(寫入多少 bytes)以及 fileID(預計寫入的檔案編號)

`L142` 取址之後要寫入資料的記憶體空間，將起始位置存入指標 buffer

`L143` 呼叫 SysWrite()，將所需資訊(buffer, numChar, fileID)傳送到 system call interface

`L146` 將回傳的實際寫入字元數 numWritten 存入 r2

`L149-L151` 更新 PrevPCReg, PCReg 和 NextPCReg 裡存的 program counter，才會繼續執行 user progam 中的下個指令

(c.) SC_Read

```
156 case SC_Read:
157     // Load arguments from user program
158     val = kernel->machine->ReadRegister(4);
159     numChar = kernel->machine->ReadRegister(5);
160     fileID = kernel->machine->ReadRegister(6);
161 
162     // Process SysWrite Systemcall
163     char *buffer = &(kernel->machine->mainMemory[val]);
164     int numRead = SysRead(buffer, numChar, fileID);
165 
166     // Prepare Result
167     kernel->machine->WriteRegister(2, (int)numRead);
168 
169     // Set program counter
170     kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
171     kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) + 4);
172     kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg) +
173     return;
174     ASSERTNOTREACHED();
175     break;
```

L158-L160 分別從 r4, r5, r6 中找到 val(讀取資料將要存放的記憶體位置)、numChar(讀取多少 bytes)以及 fileID(預計讀取的檔案編號)

L163 取址之後要存放資料的記憶體空間，將起始位置存入指標 buffer

L164 呼叫 SysRead()，將所需資訊(buffer, numChar, fileID)傳送到 system call interface

L167 將回傳的實際讀取字元數 numRead 存入 r2

L149-L151 更新 PrevPCReg, PCReg 和 NextPCReg 裡存的 program counter，才會繼續執行 user progam 中的下個指令

(d.) SC_Close

```
199     case SC_Close:
200         fileID = kernel->machine->ReadRegister(4);
201         //cout << "fileID: " << fileID << endl;
202 
203         // Process SysClose Systemcall
204         status = SysClose(fileID);
205         kernel->machine->WriteRegister(2, (int)status);
206 
207         // Set program counter
208         kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
209         kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) + 4);
210         kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg) + 4);
211         return;
212         ASSERTNOTREACHED();
213         break;
```

`L200` Read the `fileID` stored in register `r4` .

`L146` Call `SysClose()` and pass the `fileID` as argument.

`L147` Store the returned `status` into register `r2` .

`L208-210` Update PC.

4. userprog/ksyscall.h: 作為 kernel 和 system call 之間的互動介面

(a.) SysOpen

```
47  OpenFileId SysOpen(char *name){
48      OpenFileId fileID = kernel->fileSystem->OpenAFile(name);
49      return fileID;
50  }
```

`L48-L49` 呼叫 filesys.h 定義下的 `OpenAFile()`，傳遞所需資訊後，等待其回傳 `fileID` 再回傳至上層

(b.) SysWrite

```
47  int SysWrite(char *buffer, int size, OpenFileId id){
48      int numWritten = kernel->fileSystem->WriteFile(buffer, size, id);
49      return numWritten;
50  }
```

`L48-L49` 呼叫 filesys.h 定義下的 `WriteFile()`，傳遞所需資訊後，等待回傳的實際寫入字元數再回傳至上層

(c.) SysRead

```
52 int SysRead(char *buffer, int size, OpenFileId id){
53     int numRead = kernel->fileSystem->ReadFile(buffer, size, id);
54     return numRead;
55 }
```

`L53-L54` 呼叫 filesys.h 定義下的 `ReadFile()`，傳遞所需資訊後，等待回傳的實際讀取字元數再回傳至上層

(d.) SysClose

```
62 int SysClose(OpenFileId id){
63     int status = kernel->fileSystem->CloseFile(id);
64     return status;
65 }
```

`L63-L64` 呼叫 filesys.h 定義下的 `CloseFile()`，傳遞所需資訊後，等待其回傳狀態再回傳至上層

5. filesys/filesys.h

(a.) OpenAFile

```
70     OpenFileId OpenAFile(char *name) {
71         OpenFile *file = Open(name);
72 
73         // Check if the file exist
74         if(file == NULL) return -1;
75 
76         for(int i=0; i<20; i++){
77             if(OpenFileTable[i] == NULL){
78                 // Check if the file is already opened
79                 for(int j=0; j<20; j++){
80                     if(!strcmp(name, OpenFileName[j])) return -1;
81                 }
82 
83                 OpenFileTable[i] = file;
84                 strcpy(OpenFileName[i], name);
85                 return i+1; // return fileID
86             }
87         }
88         // Exceed the opened file limit
89         delete file;
90         return -1;
91     }
```

`L71` 呼叫 `FileSystem::Open()`，並將檔案名傳入

L74 若檔案沒被成功開啟，則回傳-1

L76 看是否還有位子開新檔案

L77-81 若有空位則先檢查該檔案目前是否已經有被開啟，有的話則回傳-1

L83-85 在 OpenFileID 中放入被開啟的檔案，並在 OpenFileName(定義於 L122，如下圖)中記錄其檔案名，再將 fileID 回傳

```
122        char OpenFileName[20][1024];
```

L89-L90 達到開啟檔案的上限，因此刪除剛剛的 file，再回傳-1.

(b.)WriteFile

```
73      int WriteFile(char *buffer, int size, OpenFileId id){
74          // Check if the target file and the size are valid
75          if(size < 0 || id < 1 || id > 20 || OpenFileTable[id-1] == NULL) return -1;
76          int numWritten = OpenFileTable[id-1]->Write(buffer, size);
77          return numWritten;
78      }
```

L75 檢查參數是否都合法：size 不可為負、檔案 id 介於 1-20、指定檔案能被成功開啟，若其中一項不符合，則回傳-1

L76 呼叫定義在 openfile.h 下的 Write()和 WriteAt()函數，接續前一次呼叫寫到的位置繼續寫入字元，最後回傳實際寫入的字元數

(c.)ReadFile

```
100     int ReadFile(char *buffer, int size, OpenFileId id){
101         // Check if the target file and the size are valid
102         if(size < 0 || id < 1 || id > 20 || OpenFileTable[id-1] == NULL) return -1;
103         int numRead = OpenFileTable[id-1]->Read(buffer, size);
104
105         // Check if read file with no error
106         if(numRead < 0) return -1;
107         return numRead;
108     }
```

L102 檢查參數是否都合法：size 不可為負、檔案 id 介於 1-20、指定檔案能被成功開啟，若其中一項不符合，則回傳-1

L103 呼叫 openfile.h 下相關函數:Read()→ReadAt()→ReadPartial()，接續前一次呼叫讀到的位置繼續讀取最多的可讀字元，最後將實際讀取的字元數存入 numRead

`L106` 根據 lib/sysdep.cc 的 `read()`函數定義(如下圖)，當讀取失敗時會回傳-1，所以這裡需要再次檢查最終是否讀取成功，再回傳 `numRead`

```
ssize_t read(int __fd, void *__buf, size_t __nbytes)

Read NBYTES into BUF from FD. Return the
number read, -1 for errors or 0 for EOF.
```

(d.)`CloseFile`

```
110     int CloseFile(OpenFileId id){
111         // Check if the target file is valid
112         if(id < 1 || id > 20 || OpenFileTable[id-1] == NULL) return -1;
113         delete OpenFileTable[id-1];
114         OpenFileTable[id-1] = NULL;
115         memset(OpenFileName[id-1], '\0', 1024);
116         return 1;
117     }
```

`L112` 檢查參數是否都合法：檔案 `id` 介於 `1-20`、指定檔案能被成功開啟，若其中一項不符合，則回傳-1

`L113` 刪除指向的 `OpenFile` 物件(其 destructor 會呼叫 lib/sysdep.cc 的 `Close()`函數)

`L114-116` 將 `OpenFileTable[id-1]`的值設為 `NULL`，並從 `OpenFileName` 中刪除此檔名，再回傳 1

IV. Difficulties

- 也許因為 NachOS 是運行在 VM 中的模擬系統，有些函數內的實作方式和課本不同，比較難直接連想到同一個觀念。以”context switch”為例，典型的步驟需要 save/reload PCB，可是 NachOS 中定義的 `Yield()`函數則是以 Interrupt state 紀錄當下 thread 的狀態。
- 在處理 duplicate file opening 的部分初期並未想到如何有效解決這個問題，因此參考了討論區其他同學的建議。

V. Feedback

- 在學期開始前，我們對於 OS 是完全陌生的，只知道它是軟硬體之間溝通的橋樑，透過這份作業，讓我們實際觀察 system call 實作的完整流程，加深在課堂上吸收的觀念的印象，針對不太熟悉的技術，像是 Semaphore 和 lock 機制，我們也盡量在短時間內理解他

們的作用以及原理，期待在未來的學習中可以更全面地理解 OS 的各個核心功能!

- 雖然作業的 spec 有說要從哪邊開始 trace code，但因為與以往單純從 main()開始看的習慣不同，這讓我們一開始非常措手不及。不過，在一層一層看過後，有慢慢掌握了 NachOS 的架構，對 system call 也有更深入的理解了!

VI. References

1. **Semaphore**
   Semaphore - iT 邦幫忙::一起幫忙解決難題，拯救 IT 人的一天 (ithome.com.tw)

2. **Makefile**
   [Day13] Makefile 介紹 - iT 邦幫忙::一起幫忙解決難題，拯救 IT 人的一天 (ithome.com.tw)