Operating System
MP1 Report

組別：25

陳凱揚：Trace code、Implement、Report

簡佩如：Trace code、Implement、Report

1. Trace code

(a) SC_Halt

(1) /test/halt.c -> /userprog/syscall.h -> /test/start.S

在第 13 行中 include 了 syscall.h，所以當執行到了第 18 行的 Halt()時（圖 1-1），會進入 syscall.h 找到 Halt()的宣告，接著就會在同在 tset 資料夾下的 start.S 裡找到 Halt()所對應的組語。在第 48 行（圖 1-2）時，addiu 指令會將後面兩個值相加並存入第一個值的 register（不會 overflow），由於我們將 r0 的值隨時保持為 0，因此也就相當於將 SC_Halt 存入 r2。接著下一個指令為 system call instruction，最後再回到儲存 return address 的 r31。

```
13 #include "syscall.h"
14
15 int
16 main()
17 {
18     Halt();
19     /* not reached */
20 }
```

▲ 圖 1-1

```
47 Halt:
48          addiu $2,$0,SC_Halt
49          syscall
50          j       $31
51          .end Halt
52
53          .globl  PrintInt
54          .ent    PrintInt
```

▲圖 1-2

(2) /machine/mipssim.cc

在 Nachos 上，有個模擬 MIPS 架構的 CPU，會執行 Run()來處理指令，其中會有個無限迴圈，不停的執行第 68 行（圖 1-3）的 OneInstrction()。而在進入迴圈前，第 65 行會先將 kernel mode 設為 UserMode，以執行接下來的 user instruction。

進到 OneInstruction()後，會先去 memory 取得下一個要執行的指令，並 Decode()得到指令的資訊（圖 1-4），接著根據不同的指令對應到不同的做法（圖 1-5），而 SC_Halt 是一個 system call，會對應到 OP_SYSCALL，並執行第 677 行（圖 1-6）的 RaiseException()。

```
56 void
57 Machine::Run()
58 {
59     Instruction *instr = new Instruction;
60
61     if (debug->IsEnabled('m')) {
62         cout << "Starting program in thread
63         cout << ", at time: " << kernel->st
64     }
65     kernel->interrupt->setStatus(UserMode);
66     for (;;) {
67         DEBUG(dbgTraCode, "In Machine::Run(
68         OneInstruction(instr);
69         DEBUG(dbgTraCode, "In Machine::Run(
70
71         DEBUG(dbgTraCode, "In Machine::Run(
72         kernel->interrupt->OneTick();
73         DEBUG(dbgTraCode, "In Machine::Run(
74         if (singleStep && (runUntilTime <=
75                 Debugger();
76     }
77 }
```

▲ 圖 1-3

```
125 void
126 Machine::OneInstruction(Instruction *instr)
127 {
128 #ifdef SIM_FIX
129     int byte;       // described in Kane for
130 #endif
131
132     int raw;
133     int nextLoadReg = 0;
134     int nextLoadValue = 0;      // record del
135                                 // in the fut
136
137     // Fetch instruction
138     if (!ReadMem(registers[PCReg], 4, &raw))
139         return;                 // exception
140     instr->value = raw;
141     instr->Decode();
```

▲ 圖 1-4

```
158
159     // Execute the instruction
160     switch (instr->opCode) {
161
162       case OP_ADD:
163         sum = registers[instr->
164         if (!((registers[instr-
165             ((registers[instr->
166             RaiseException(Over
167             return;
168         }
169         registers[instr->rd] =
170         break;
```

▲ 圖 1-5

```
675       case OP_SYSCALL:
676         DEBUG(dbgTraCode, "In Machine::OneIns
677         RaiseException(SyscallException, 0);
678         return;
```

▲ 圖 1-6

(3) /machine/machine.cc

在 RaiseException()裡，在進到 107 行（圖 1-7）的 ExceptionHandler()前，因為即將要執行 system call 或是 exception，所以會先將 kernel mode 從 UserMode 轉換成 SystemMode，而執行完後會設回 UserMode。

```
100 void
101 Machine::RaiseException(ExceptionType which, int badVAddr)
102 {
103     DEBUG(dbgMach, "Exception: " << exceptionNames[which]);
104     registers[BadVAddrReg] = badVAddr;
105     DelayedLoad(0, 0);                  // finish anything i
106     kernel->interrupt->setStatus(SystemMode);
107     ExceptionHandler(which);            // interrupts are ena
108     kernel->interrupt->setStatus(UserMode);
109 }
```

▲ 圖 1-7

(4) /userprog/exception.cc

進到 ExceptionHandler()之後，會先到 r2 找到步驟 1 存起來的 type，再根據 which 和 type，執行對應的處理方法。因此在第 62 行（圖 1-8）可以找到 SC_Halt，並執行 SysHalt()。

```
50 void
51 ExceptionHandler(ExceptionType which)
52 {
53     char ch;
54     int val;
55     int type = kernel->machine->ReadRegister(2);
56     int status, exit, threadID, programID, fileID, numChar;
57     DEBUG(dbgSys, "Received Exception " << which << " type: " << type <<
58     DEBUG(dbgTraCode, "In ExceptionHandler(), Received Exception " << whi
59     switch (which) {
60     case SyscallException:
61         switch(type) {
62             case SC_Halt:
63                 DEBUG(dbgSys, "Shutdown, initiated by user program.\n");
64                 SysHalt();
65                 cout<<"in exception\n";
66                 ASSERTNOTREACHED();
67             break;
```

▲ 圖 1-8

(5) /userprog/ksyscall.h

接著會進到 ksyscall.h 裡找到 SysHalt()，並執行第 21 行（圖 1-9）。

```
19 void SysHalt()
20 {
21   kernel->interrupt->Halt();
22 }
```

▲ 圖 1-9

(6) /machine/interrupt.cc

最後會執行 Halt()這個 interrupt，在程式結束前 Print()印出一些統計的資訊（圖 1-10），並 delete kernel，程式就結束了。

```
236 void
237 Interrupt::Halt()
238 {
239     cout << "Machine halting!\n\n";
240     cout << "This is halt\n";
241     kernel->stats->Print();
242     delete kernel;      // Never returns.
243 }
```

▲ 圖 1-10

(b) SC_Create

(1) /userprog/exception.cc

如同 SC_Halt，SC_Create 這個 system call 也會進到 ExceptionHandler()中（圖 1-11）。由於他含有一個參數，因此會先去 r4 找出這個參數(即"指向 filename 的指標")，透過指標從 kernel 的 mainMemory 得到字串所在位置，並傳回第一個字元的位址給 filename。接著就以 filename 為參數去執行 ksyscall.h 裡的 SysCreate()，並回傳執行後的 return 值，將這個值存回 r2 裡。最後需要更新 Program counter，將現在的 PCReg 寫入 PrevPCReg、PCReg 和 NextPCReg 各+4，使下次執行時可以得到下一個 instruction。

```
 91             case SC_Create:
 92                 val = kernel->machine->ReadRegister(4);
 93                 {
 94                 char *filename = &(kernel->machine->mainMemory[val]);
 95                 //cout << filename << endl;
 96                 status = SysCreate(filename);
 97                 kernel->machine->WriteRegister(2, (int) status);
 98                 }
 99                 kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
100                 kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) + 4);
101                 kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg)+4);
102                 return;
103                 ASSERTNOTREACHED();
104             break;
```

▲ 圖 1-11

(2) /userprog/ksyscall.h

在這邊會執行第 41 行（圖 1-12），kernel 裡面的 fileSystem 提供的 Create()，如果成功即回傳 1，失敗則回傳 0。

```
36 int SysCreate(char *filename)
37 {
38         // return value
39         // 1: success
40         // 0: failed
41         return kernel->fileSystem->Create(filename);
42 }
```

▲ 圖 1-12

(3) /filesys/filesys.h -> /lib/sysdep.cc

而上一步驟會進到 filesys.h 裡的 Create()，並執行第 53 行（圖 1-13）裡中位於 sysdep.cc 裡的 OpenForWrite()（圖 1-14），這是一個使用 unix 的 open()作為開啟檔案的方式，並會回傳檔案是否成功開啟，-1 代表失敗，接著呼叫同樣位於 sysdep.cc 裡的 Close()（圖 1-15），使用 unix 的 close()來關閉檔案，並回傳 TRUE。

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

▲ 圖 1-13

```
307 int
308 OpenForWrite(char *name)
309 {
310     int fd = open(name, O_RDWR|O_CREAT|O_TRUNC, 0666);
311
312     ASSERT(fd >= 0);
313     return fd;
314 }
```

▲ 圖 1-14

```
405 int
406 Close(int fd)
407 {
408     int retVal = close(fd);
409     ASSERT(retVal >= 0);
410     return retVal;
411 }
```

▲ 圖 1-15

(c) SC_PrintInt

(1) /userprog/exception.cc

這次進到 ExceptionHandler()，會來到 SC_PrintInt 的地方（圖 1-16），同樣會先去 r4 取出第一個參數，而這裡與 SC_Create 不同的是不需要 return 值，因此不用將結果存進 r2，只要執行 ksyscall.h 裡的 SysPrintInt()，並更新 Program counter。

```
68              case SC_PrintInt:
69                  DEBUG(dbgSys, "Print Int\n");
70                  val=kernel->machine->ReadRegister(4);
71                  DEBUG(dbgTraCode, "In ExceptionHandler(), into SysPrintInt, " << kernel->stats->to
72                  SysPrintInt(val);
73                  DEBUG(dbgTraCode, "In ExceptionHandler(), return from SysPrintInt, " << kernel->st
74                  // Set Program Counter
75                  kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
76                  kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) + 4);
77                  kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg)+4);
78                  return;
79                  ASSERTNOTREACHED();
80              break;
```

▲ 圖 1-16

(2) /userprog/ksyscall.h

這裡會執行第 27 行（圖 1-17）kernel 裡的 synchConsoleOut 的 PutInt()。

```
24 void SysPrintInt(int val)
25 {
26   DEBUG(dbgTraCode, "In ksyscall.h:SysPr
27   kernel->synchConsoleOut->PutInt(val);
28   DEBUG(dbgTraCode, "In ksyscall.h:SysPr
29 }
```

▲ 圖 1-17

(3) /userprog/synchconsole.cc

進到 PutInt()後，會先將要輸出的數字轉換成字串的形式，並加上'\n'（圖 1-18），由於 IO 機器一次只能執行一個動作，印完一個才能印下一個，因此一開始會先執行 lock->Acquire()，等待獲取輸出權後，將其鎖起來，並且只有現在這個 thread 可以解鎖，確保輸出過程不被截斷，直到所有要輸出的東西印完後，才會呼叫 lock->Release()解鎖。

在輸出過程，會使用 consoleOutput 的 PutChar()，一個一個將字元印出，而每印一個字元後，會呼叫 waitFor->P()，waitFor 是一個 semaphore，有兩種功能，P()會等待直到 value > 0，再 value--，V()則是會叫起一個等待中的 thread，再 value++。在這邊的功能是等待前一個字元輸出完成後，再去執行輸出下一個字元。

```
109 void
110 SynchConsoleOutput::PutInt(int value)
111 {
112     char str[15];
113     int idx=0;
114     //sprintf(str, "%d\n\0", value);  the t
115     sprintf(str, "%d\n\0", value); //simply
116     lock->Acquire();
117     do{
118         DEBUG(dbgTraCode, "In SynchConsoleO
119         consoleOutput->PutChar(str[idx]);
120         DEBUG(dbgTraCode, "In SynchConsoleO
121         idx++;
122 
123         DEBUG(dbgTraCode, "In SynchConsoleO
124         waitFor->P();
125         DEBUG(dbgTraCode, "In SynchConsoleO
126     } while (str[idx] != '\0');
127     lock->Release();
128 }
```

▲ 圖 1-18

(4) /machine/console.cc -> /lib/sysdep.cc

在這裡會使用 sysdep.cc 裡的 WriteFile()（圖 1-20），使用 unix 的 write()，來將字元寫入檔案中。接著在第 173 行（圖 1-19）會呼叫 kernel 裡的 interrupt 的 Schedule()，第二個參數的 ConsoleTime 為 IO 機器讀或寫一個字元所需的時間，也就是排程一個 interrupt 在未來 IO 結束時，使 CPU 知道可以重新回到這邊繼續執行。

```
167 void
168 ConsoleOutput::PutChar(char ch)
169 {
170     ASSERT(putBusy == FALSE);
171     WriteFile(writeFileNo, &ch, sizeof(char));
172     putBusy = TRUE;
173     kernel->interrupt->Schedule(this, ConsoleTime, ConsoleWriteInt);
174 }
```

▲ 圖 1-19

```
363 void
364 WriteFile(int fd, char *buffer, int nBytes)
365 {
366     //printf("In sysdep.cc, nBytes: %d\n", nBytes);
367         int retVal = write(fd, buffer, nBytes);
368     ASSERT(retVal == nBytes);
369 }
```

▲ 圖 1-20

(5) /machine/interrupt.cc

Schedule()裡會 new 一個 PendingInterrupt，並確認 fromNow > 0，確保這個 interrupt 發生的時間點在未來，接著加進 pending 裡依照時間順序做排程（圖 1-21）。

```
297 void
298 Interrupt::Schedule(CallBackObj *toCall, int fromNow, IntType type)
299 {
300     int when = kernel->stats->totalTicks + fromNow;
301     PendingInterrupt *toOccur = new PendingInterrupt(toCall, when, type);
302
303     DEBUG(dbgInt, "Scheduling interrupt handler the " << intTypeNames[type
304     ASSERT(fromNow > 0);
305
306     pending->Insert(toOccur);
307 }
```

▲ 圖 1-21

(6) /machine/mipssim.cc

在 Run()的第 72 行（圖 1-22），每次執行完一個 user instruction，都會呼叫 interrupt 的 OneTick()，來增加 simulated time 並檢查是否有 interrupt 需要執行。

```
56 void
57 Machine::Run()
58 {
59     Instruction *instr = new Instruction;
60
61     if (debug->IsEnabled('m')) {
62         cout << "Starting program in thread
63         cout << ", at time: " << kernel->st
64     }
65     kernel->interrupt->setStatus(UserMode);
66     for (;;) {
67         DEBUG(dbgTraCode, "In Machine::Run(
68         OneInstruction(instr);
69         DEBUG(dbgTraCode, "In Machine::Run(
70
71         DEBUG(dbgTraCode, "In Machine::Run(
72         kernel->interrupt->OneTick();
73         DEBUG(dbgTraCode, "In Machine::Run(
74         if (singleStep && (runUntilTime <=
75                 Debugger();
76     }
77 }
```

▲ 圖 1-22

(7) /machine/interrupt.cc

在 OneTick()裡（圖 1-23），首先會先增加 simulated time，接著由於要檢查目前是否有 interrupt 在等待執行，因此必須先 disable interrupt，並在 CheckIfDue()中檢查並執行 interrupt 後，重新 enable interrupt。最後的 yieldOnReturn 則是在檢查是否需要 context switch，如果需要則會呼叫 currentThread 的 Yield()來做轉換。

```
148 Interrupt::OneTick()
149 {
150     MachineStatus oldStatus = status;
151     Statistics *stats = kernel->stats;
152
153 // advance simulated time
154     if (status == SystemMode) {
155         stats->totalTicks += SystemTick;
156         stats->systemTicks += SystemTick;
157     } else {
158         stats->totalTicks += UserTick;
159         stats->userTicks += UserTick;
160     }
161     DEBUG(dbgInt, "== Tick " << stats->totalTicks << " ==");
162
163 // check any pending interrupts are now ready to fire
164     ChangeLevel(IntOn, IntOff); // first, turn off interrupts
165                                 // (interrupt handlers run with
166                                 // interrupts disabled)
167     CheckIfDue(FALSE);          // check for pending interrupts
168     ChangeLevel(IntOff, IntOn); // re-enable interrupts
169     if (yieldOnReturn) {        // if the timer device handler asked
170                                 // for a context switch, ok to do it now
171         yieldOnReturn = FALSE;
172         status = SystemMode;            // yield is a kernel routine
173         kernel->currentThread->Yield();
174         status = oldStatus;
175     }
176 }
```

▲ 圖 1-23

(8) /machine/interrupt.cc

在 CheckIfDue()裡，會先檢查是否有待執行的 interrupt，且此 interrupt 的時間點必須小於等於目前的時間（圖 1-25），而在這裡因為剛剛傳進的 advanceClock = FALSE，因此若下一個 interrupt 的時間點大於目前的時間，會直接 return FALSE，代表沒有處理任何 interrupt。

接著會有一個 do-while 迴圈不停地執行 interrupt 並刪除，直到所有目前時間點以前的 interrupt 都被執行完畢，而在第 359 行（圖 1-24）處，這個 interrput 會呼叫 CallBack()回到之前執行 IO 的地方繼續執行。最後迴圈執行結束後，會 return TRUE，代表有 interrupt 被處理掉了。

```
355     inHandler = TRUE;
356     do {
357         next = pending->RemoveFront();    // pull interrupt off list
358                 DEBUG(dbgTraCode, "In Interrupt::CheckIfDue, into ca
359         next->callOnInterrupt->CallBack();// call the interrupt hand
360                 DEBUG(dbgTraCode, "In Interrupt::CheckIfDue, return
361         delete next;
362     } while (!pending->IsEmpty()
363                 && (pending->Front()->when <= stats->totalTicks));
364     inHandler = FALSE;
365     return TRUE;
```

▲ 圖 1-24

```
321 bool
322 Interrupt::CheckIfDue(bool advanceClock)
323 {
324     PendingInterrupt *next;
325     Statistics *stats = kernel->stats;
326
327     ASSERT(level == IntOff);           // interrupts need to be disabled,
328                                        // to invoke an interrupt handler
329     if (debug->IsEnabled(dbgInt)) {
330         DumpState();
331     }
332     if (pending->IsEmpty()) {          // no pending interrupts
333         return FALSE;
334     }
335     next = pending->Front();
336
337     if (next->when > stats->totalTicks) {
338         if (!advanceClock) {           // not time yet
339             return FALSE;
340         }
341         else {                 // advance the clock to next interrupt
342             stats->idleTicks += (next->when - stats->totalTicks);
343             stats->totalTicks = next->when;
344             // UDelay(1000L); // rcgood - to stop nachos from spinning.
345         }
346     }
```

▲ 圖 1-25

(9) /machine/console.cc

回到剛剛輸出的地方後，因為上一個字元已經輸出完畢，因此將 putBusy 設為 FALSE（圖 1-26），並呼叫 callWhenDone 的 CallBack()告訴他輸出已經結束，可以叫下一個字元準備輸出了。

```
152 void
153 ConsoleOutput::CallBack()
154 {
155         DEBUG(dbgTraCode, "In ConsoleOutput:
156     putBusy = FALSE;
157     kernel->stats->numConsoleCharsWritten++;
158     callWhenDone->CallBack();
159 }
```

▲ 圖 1-26

(10) /userprog/synchconsole.cc

回到這邊後，就會呼叫 waitFor 這個 semaphore 的 V()，將 val++，並叫醒等待中的下一個 thread，使其可以進行輸出，也就是使圖 1-18 第 124 行的 waitFor()->P()能夠繼續往下執行。

```
136 void
137 SynchConsoleOutput::CallBack()
138 {
139     DEBUG(dbgTraCode, "In Synch(
140     waitFor->V();
141 }
```

▲ 圖 1-27

2. Implement

(a) /test/syscall.h

在 syscall.h 裡，我更改了第 28、29、30、32 行（圖 2-1），將註解掉的部分刪去，define 這些 system call 的指令，而 Open()、Read()、Write()、Close()等函式，下方已經有先宣告了，所以不需要再自己加上。

```
27 // ********** MP1 ********** //
28 #define SC_Open 6
29 #define SC_Read 7
30 #define SC_Write        8
31 #define SC_Seek         9
32 #define SC_Close        10
33 // ********** MP1 ********** //
```

▲ 圖 2-1

(b) /userprog/syscall.h

在 start.S 裡，我更改的程式碼位於第 165 行至第 197 行（圖 2-2），加上我們要實作的這四個 system call，使他們被呼叫時，有對應的組語可使 nachos 執行，而這邊每個寫法都與其他 system call 相同，皆將對應到的指令存進 r2 中，加上一個 syscall 指令，最後回到儲存 return address 的位置。

```
165 // *************** MP1 *************** //
166         .global Open
167         .ent Open
168 Open:
169         addiu $2, $0, SC_Open
170         syscall
171         j       $31
172         .end Open
173
174         .global Read
175         .ent Read
176 Read:
177         addiu $2, $0, SC_Read
178         syscall
179         j       $31
180         .end Read
181
182         .global Write
183         .ent Write
184 Write:
185         addiu $2, $0, SC_Write
186         syscall
187         j       $31
188         .end Write
189
190         .global Close
191         .ent Close
192 Close:
193         addiu $2, $0, SC_Close
194         syscall
195         j       $31
196         .end Close
197 // *************** MP1 *************** //
```

▲ 圖 2-2

(c) /userprog/exception.cc

在 exception.cc 裡，我更改的程式碼位於第 135 行至第 191 行，使程式進到 ExceptionHandler()後，能依照 which 和 type 找到並執行 SC_Open（圖 2-3）、SC_Write（圖 2-4）、SC_Read（圖 2-5）、SC_Close（圖 2-6）這些 system call 的處理方式。

(1) SC_Open

首先先從$4 裡讀出第一個參數值，由於第一個參數的 type 為 char*，也就是一個字串的第一個字元的位址，因此需要以 val 去 mainMemory 找出這個位址並回傳給 filename，接著就能呼叫 kernel interface 裡的 SysOpen()（位於 ksyscall.h）來執行這個 system call，並將回傳的值存回 r2 中。最後需要更新 Program counter，將 PrevPCReg、PCReg、NextPCReg 的位置都+4，使下次可以執行新的 instruction。

```
135         // ****************** MP1 ****************** //
136         case SC_Open:
137                 val = kernel->machine->ReadRegister(4);
138                 {
139                 char *filename = &(kernel->machine->mainMemory[val]);
140                 status = SysOpen(filename);
141                 kernel->machine->WriteRegister(2, (int)status);
142                 }
143                 kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
144                 kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg)+4);
145                 kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg)+4);
146                 return;
147                 ASSERTNOTREACHED();
148                 break;
```

▲ 圖 2-3

(2) SC_Write

由於這個指令包含三個參數，因此依序從 r4、r5、r6 找到 val、numChar、fileID 等參數，同樣的需要去 mainMemory 找到第一個參數 buffer 的位址，接著呼叫 SysWrite()執行，將回傳值存入 r2，最後更新 Program counter。

```
149         case SC_Write:
150                 val = kernel->machine->ReadRegister(4);
151                 numChar = kernel->machine->ReadRegister(5);
152                 fileID = kernel->machine->ReadRegister(6);
153                 {
154                 char *buffer = &(kernel->machine->mainMemory[val]);
155                 status = SysWrite(buffer, numChar, fileID);
156                 kernel->machine->WriteRegister(2, (int)status);
157                 }
158                 kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
159                 kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg)+4);
160                 kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg)+4);
161                 return;
162                 ASSERTNOTREACHED();
163                 break;
```

▲ 圖 2-4

(3) SC_Read

與 SC_Write 幾乎相同，參數也完全相同，只差在呼叫的是 SysRead()。

```
164         case SC_Read:
165                 val = kernel->machine->ReadRegister(4);
166                 numChar = kernel->machine->ReadRegister(5);
167                 fileID = kernel->machine->ReadRegister(6);
168                 {
169                 char *buffer = &(kernel->machine->mainMemory[val]);
170                 status = SysRead(buffer, numChar, fileID);
171                 kernel->machine->WriteRegister(2, (int)status);
172                 }
173                 kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
174                 kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg)+4);
175                 kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg)+4);
176                 return;
177                 ASSERTNOTREACHED();
178                 break;
```

▲ 圖 2-5

(4) SC_Close

同樣的讀取第一個參數，並呼叫 SysClose()，其他步驟皆相同。

```
179         case SC_Close:
180                 fileID = kernel->machine->ReadRegister(4);
181                 {
182                 status = SysClose(fileID);
183                 kernel->machine->WriteRegister(2, (int)status);
184                 }
185                 kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));
186                 kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg)+4);
187                 kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg)+4);
188                 return;
189                 ASSERTNOTREACHED();
190                 break;
191                 // ******************** MP1 ******************** //
```

▲ 圖 2-6

(d) /userprog/ksyscall.h

在 ksyscall.h 裡，我更改的程式碼位於第 46 行至第 66 行（圖 2-7），這裡只是一個 interface，直接呼叫 fileSystem 裡對應的函式，並回傳其回傳值。

```
46 // *************** MP1 *************** //
47 OpenFileId SysOpen(char *name)
48 {
49   return kernel->fileSystem->OpenAFile(name);
50 }
51
52 int SysWrite(char *buffer, int size, OpenFileId id)
53 {
54   return kernel->fileSystem->WriteFile(buffer, size, id);
55 }
56
57 int SysRead(char *buffer, int size, OpenFileId id)
58 {
59   return kernel->fileSystem->ReadFile(buffer, size, id);
60 }
61
62 int SysClose(OpenFileId id)
63 {
64   return kernel->fileSystem->CloseFile(id);
65 }
66 // *************** MP1 *************** //
```

▲ 圖 2-7

(e) /filesys/filesys.h

在 filesys.h 裡，我更改的程式碼位於第 68 行至第 95 行，包括 OpenAFile()（圖 2-8）、WriteFile()（圖 2-9）、ReadFile()（圖 2-10）、CloseFile()（圖 2-11），這裡就是經過層層 interface 後呼叫到最後實作的地方，是 fileSystem 所提供的函式，而這個 fileSystem 實際上底層是呼叫 unix 的功能來執行關於檔案的操作，並以 OpenFileTable 來儲存開啟的檔案，同時開啟上限為 20 個檔案。

(1) OpenAFile()

首先以 fileSystem 提供的 Open()來以 name 開啟檔案，Open()會利用 sysdep.h 裡的 OpenForReadWrite()來開啟檔案，如果成功開啟，Open()會 new 一個儲存此檔案位置的 OpenFile 物件，並回傳此物件，若失敗則會回傳 NULL。

因此若得到回傳值是 NULL 時，就直接回傳 -1，代表檔案開啟失敗，若成功開啟，就去 OpenFileTable 找一個未使用的位置，放進這個開啟的檔案，並回傳 OpenFileId（需要加 1，因為 id 不得為 0）。如果發現整個 table 都沒有空位，代表檔案開啟已到上限，這時 delete 掉剛剛 new 出來的 OpenFile 物件，並回傳 -1。

```
68      // ********************* MP1 ********************* //
69      OpenFileId OpenAFile(char *name) {
70          OpenFile *file = Open(name);
71          if(file == NULL) return -1;
72          for(int i = 0; i < 20; i++){
73              if(OpenFileTable[i] == NULL){
74                  OpenFileTable[i] = file;
75                  return i+1;
76              }
77          }
78          delete file;
79          return -1;
80      }
```

▲ 圖 2-8

(2) WriteFile()

首先先檢查不合法的情況，包括 id 錯誤、size 為負數、指向的 OpenFile 是 NULL，發生這些情況就會回傳 -1，代表寫入失敗。接著會呼叫 OpenFile 物件裡提供的 Write()，底層使用了 sysdep.h 裡 unix 的 write()功能，最後會回傳實際寫入的字元數。

```
81      int WriteFile(char *buffer, int size, OpenFileId id){
82          if(id<1 || id>20 || size<0 || OpenFileTable[id-1]==NULL) return -1;
83          return OpenFileTable[id-1]->Write(buffer, size);
84      }
```

▲ 圖 2-9

(3) ReadFile()

與 WriteFile()一樣，先檢查不合法的情況，發生時就回傳 -1。接著呼叫 OpenFile 物件裡提供的 Read()，並回傳實際讀到的字元數。

```
85      int ReadFile(char *buffer, int size, OpenFileId id){
86          if(id<1 || id>20 || size<0 || OpenFileTable[id-1]==NULL) return -1;
87          return OpenFileTable[id-1]->Read(buffer, size);
88      }
```

▲ 圖 2-10

(4) CloseFile()

同樣先檢查不合法的情況，發生時就回傳 -1。接著 delete 掉指向的 OpenFile 物件，並將原先指向這個物件的 OpenFileTable[id-1]設成 NULL，最後回傳 1，表示成功關閉檔案。

```
89      int CloseFile(OpenFileId id){
90          if(id<1 || id>20 || OpenFileTable[id-1]==NULL) return -1;
91          delete OpenFileTable[id-1];
92          OpenFileTable[id-1] = NULL;
93          return 1;
94      }
95      // ********************* MP1 ********************* //
```

▲ 圖 2-11

3. Difficulties and Feedback

在 trace code 時，由於一個類別經常包含另一個類別的物件，如果想要徹底了解整個類別在做甚麼的話，經常都需要一層一層的往下了解其他類別，有時候開了太多層，就會不小心迷失自我，而且有些類別是後面章節的內容，像是 Lock、Semaphore，就會需要不少時間來理解他們，我覺得這是在 trace 時遇到最大的問題，也因為是第一次接觸大型系統的程式碼，感覺蠻難很快上手，大部分時間都花在了解整個架構和摸索。

而 Implement 的部分我覺得沒有遇到太多問題，要歸功於前一部份的引導讓我了解呼叫 system call 的完整流程，接著只要跟著這些流程，就可以一步一步的完成缺少的部分，實作出自己的 system call。