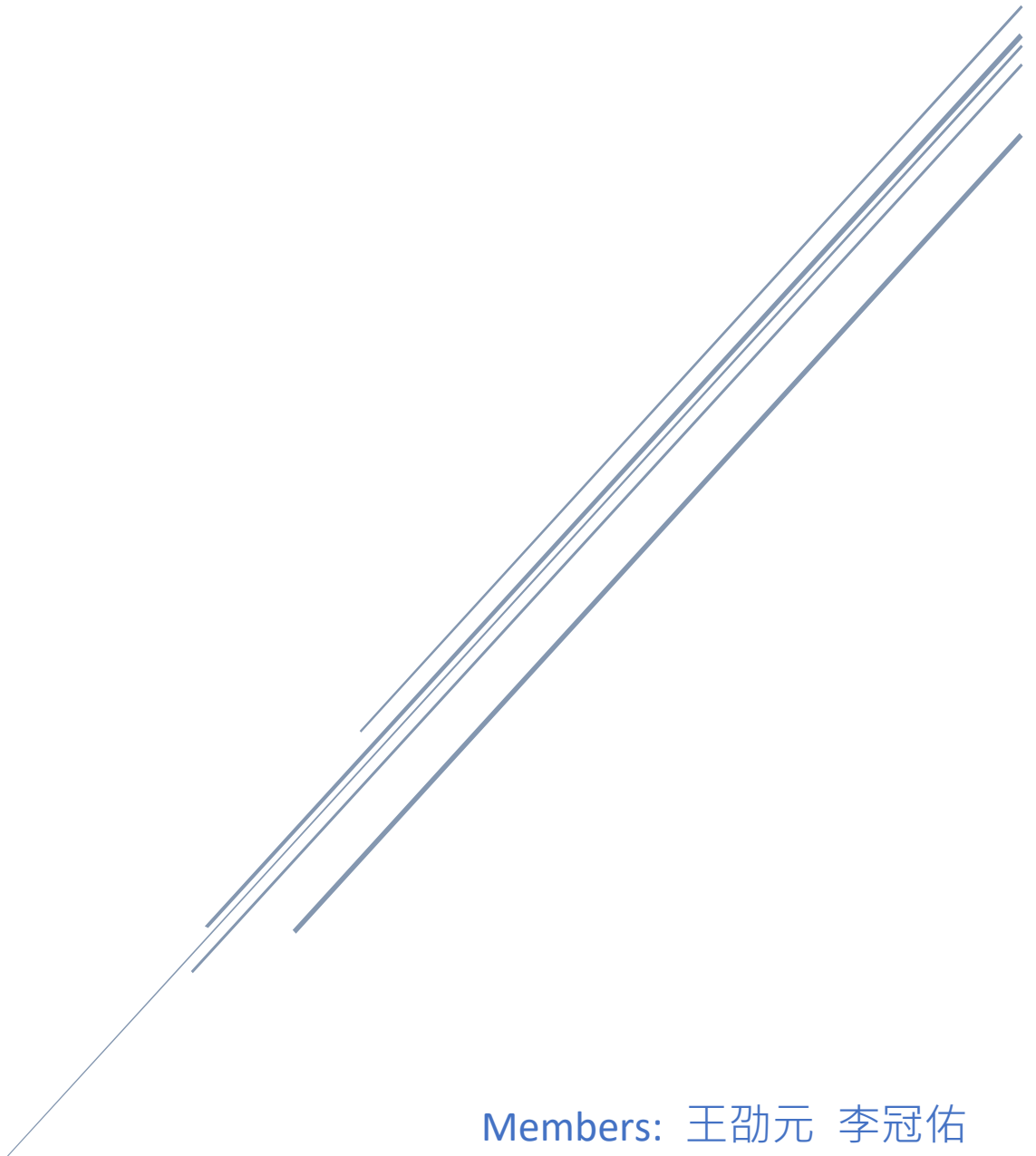


# MP1:SYSTEM CALL

## Report



Members: 王劭元 李冠佑

Contribution: 50 % 50 %

# How system calls work in NachOS in Part II-1

## Part 1: Trace code

- (a) Explain the purposes and details of each function call listed in the code path above.

### 1. *SC\_Halt*

#### ***Machine::Run():***

主要有兩個部分，OneInstruction 和 OneTick，會用一個無窮迴圈不停地跑，不斷的去讀取 instruction 和記錄時間。模擬使用者 program 的執行。

#### ***Machine::OneInstruction():***

負責去讀取使用者下一個 instruction，在一開始，會先從 memory 讀取下一個 instruction，並把它 decode，接著就去執行它。透過 switch case 的方式看 opcode 是什麼，來判斷這次要做什麼。在這個例子中，我們會跑到 OP\_SYSCALL 裡面，並呼叫 RaiseException 的函式。

#### ***Machine::RaiseException(which, badVAddr):***

傳進來的 which 是代表為什麼會導致這次 exception，而 badVAddr 則是存放導致 trap 的 virtual address。

一開始會把 badVAddr 放到存放 badVAddr 的 Register 裡，而

Delayedload(0, 0)是若上面有要延後 Load 進來的東西的話，在處理

Exception 前先 load 進來，用(0, 0)只是因為 register0 本來就會是 0，並且在 Delayedload 這個函式的最後也有確保 register0 會是 0。最後呼叫 ExceptionHandler 來處理 Exception。

#### ***ExceptionHandler(which):***

which 會告知這次的 exception 是什麼造成的，然後再從 register2 中取出告知這次準確的 exception 是什麼並用 type 接起來，像這次就會拿到 which=SyscallException, type = SC\_Halt。接著他去呼叫 SysHalt。

#### ***SysHalt():***

這個 function 單純只有去呼叫 interrupt 下的 function Halt()。這樣就可以達到讓 Nachos Halt 住的功能

#### ***Interrupt::Halt():***

這個 function 中，他會先去把 kernel 的 status 印出來後，就直接把 kernel delete 掉，達到 SC\_Halt 要做的事情。

## ***2. SC\_Create:***

#### ***ExceptionHandler(which):***

前面跟上面介紹的一樣，只是這次 switch case 會進到的 case 為 SC\_Create。首先先從 argument register 4 裡面拿到存放 filename 的 memory address，接著從 memory 讀取出來。接著就去呼叫 SysCreate，並用 status 記錄回傳的狀態，得到狀態後把它寫進

register2，最後設定 program counter 接下來的值。

### ***SysCreate(filename):***

這個 function 是去呼叫 filesystem 下的 create，並 return 有沒有成功 create。

### ***FileSystem::Create(name):***

在 *Create()* 裡面，我們使用 sysdep.cc 裡面的

*OpenForWrite(name)*，若要 Open 的 filename 不存在，就會幫忙直接 create 一個，此外，他會回傳一個 file descriptor 回來，若為-1，代表 create 失敗，*Create()* 就會直接 return FALSE 回去。接著由於我們只需要 create 一個檔案，還不需要把它打開，因此下面再借助 sysdep.cc 的 *Close(fd)* 把檔案關掉，fd 為剛剛 *OpenForWrite()* 回傳的 file descriptor。最後沒有錯誤的話就 return TRUE 回去。

## ***3.SC\_PrintInt***

### ***ExceptionHandler()***

當需要執行 system call 時，將會呼叫 *ExceptionHandler()*，而 system call code 會存於 r2，它會告訴我們現在要做的是哪一個 system call，在這裡依題意我們要執行的 system call 是 *SC\_PrintInt*，我們所要 print 的 argument 會存於 r4，接著就可以呼

叫 `SysPrintIn`，並將 `r4` 裡的 `argument` 當作參數傳入，在做完 `system call` 後，需要將 `system call` 的 `result` 存在 `r2`，也要做 `pc` 的 `increment`，避免一直重複同個 `system call`，而最後的 `ASSERTNOTREACH` 是倘若這次 `system call` 沒有正常 `return` 的話，就會 `assertion fail`

### ***SysPrintInt()***

`SysPrintInt()`是作為 `SC_PrintInt` `system call` 的 `kernal` `interface`，它會去 `call` `synchronize` 的 `consoleOutput`，呼叫 `synchConsoleOut` 裡的 `PutInt` 並將剛剛的 `argument` 傳進去

### ***synchConsoleOut::PutInt()***

它先將傳入的 `argument` 轉為 `string`，然後跑迴圈將 `string` 裡的 `char` 一個一個傳到 `consoleOutput->PutChar()`裡，並運用 `lock->aquire` 來控制讓一次只能有一個 `writer`，直到整個 `string` 做完後才會 `release` 讓其他 `writer` 去做，除此之外，每次回圈都有一個 `waitFor->P`，這個的目的是要在呼叫 `consoleOutput->PutChar()` 後，會讓 `thread sleep`，禁止進行其他 `thread`，直到獲得 `call back` 後才會能繼續執行

### ***synchConsoleOut::PutChar()***

在這個函式裡他會 call consoleOutput0->PutChar()，寫一個 character 到 console display，這裡和 PutInt 一樣會用 lock->acquire 來控制 writer，並用 waitfor->P 去等 call back

### ***ConsoleOutput::PutChar()***

這裡會先用 putBusy 來判斷現在是否有在寫其他 character，如果有得話就 abort，沒有的話就繼續做下去，將 character display 完後，它就會在未來 schedule 一個 interrupt，然後就 return

### ***Interrupt::Schedule()***

在 Schedule 裡頭，他會 new 一個 pendingInterrupt，並將這個 interrupt 之後會發生的時間，哪個 hardware device 產生的以及發生時要 call 哪個 object 記錄起來，然後判斷發生的時間是否在現在的時間之後，是的話就會將這個 interrupt insert 到 pending 的 list 裡面，等待 interrupt 時間到達

### ***Machine::Run()***

這個函式作用是執行 instruction，它有一個無窮迴圈會去執行每個 instruction，將 instruction 傳到 OneInstruction()裡去 decode，並且，每執行一個 instruction 他都會 call OneTick()，去看現在是在什麼 mode 以更新 simulated time

### ***Interrupt::OneTick()***

首先，它會去更新 simulated time，若是 userMode 就 +=1，若是 SystemMode 則是 +=10，更新完 simulated time 後，再來就是要去檢查是否有 interrupt 要執行，所以它會先 disabled interrupt，接著 call CheckIfDue() 去看是否有 interrupt 要 handle，之後再將 interrupt enable，如果這時 timer device handler 有要求 context switch 的話，就會切回 system mode 去 switch thread，結束後再切回原本的 mode

### ***Interrupt::CheckIfDue()***

它會先檢查有沒有把 interrupt disabled，如果沒有就 abort，有的話就繼續往下進行，再來，會到 pending 的 list 裡面看有沒有 interrupt 要 handle，如果沒有的話就 return false 回去，有的話就會將 list 裡最前面的 interrupt 取出來，接著就會看這個 interrupt 能不能做時間的快進，若 Ready queue 裡沒有其他 process(也就是 AdvanceClock == TRUE)，他就會將 simulated time 改為這個 interrupt 本來 schedule 發生的時間，反之就 return false，最後，就是要來 fire 這個 interrupt，它會將這個 interrupt 從 list 最前面 remove 掉，做完後就會去 call interrupt handler 去 call back，再來會去看看現在 pending list 裡面是否有其他 interrupt，若有且這些 interrupt 時間小於等於現在時間，就會繼續去 fire 這些 interrupt，直到沒有 interrupt 或 interrupt 時間還沒到才會結束回圈

### ***ConsoleOutput::CallBack()***

這是一個 call back function，在 interrupt fire 後，這裡就可以將 putBusy 改回 false，代表可以去寫下一個 character，記錄將寫過的 character 加一，最後再 call back 回去

### ***synchConsoleOut::CallBack()***

這裡代表已經等到在 waiting 的 call back 了，因此就可以設 waitFor->V，允許去執行其他的 thread

(b) Explain how the arguments of system calls are passed from user program to kernel in each of the above use cases.

## ***1. SC\_Halt***

由於這個函式只是要讓系統停掉，不需要任何變數來告訴系統，只需層層呼叫函示下去即可，所有在 ExceptionHandler 內，就沒有看到他有傳遞 arguments

## ***2. SC\_Create***

從 ExceptionHandler 可以看到，有一個 int 的變數 val 會從 register4 拿值出來後，去 mainMemory[val]的位置拿出一個 filename 的 address 出來。

這邊 val 取出來的值是一個 index，代表我們要去拿取的東西，現在被存在 mainMemory 得第幾個 index，而接下來就用變數。



char\* filename 到 memory 的第 val 個 index 中，取出我們命名的 filename 的「位置」，接著就以 pass by reference 的方式，繼續向下傳遞到最後，讓 sysdep.cc 中的 *OpenForWrite()* 可以用這個 address 幫我們創建檔案

### 3. *SC\_PrintInt*

SC\_PrintInt 的 argument 會存在 r4 裡面，可以直接從 register 裡面拿出來存在 value 裡面，不需要用 address 從 MainMemory 裡拿，接下來就拿 val 用 pass by value 的方法傳入 SysPrintInt 這個 kernal interface 裡，一層層傳遞到 sydep.cc 的 writeFile 中去寫檔案

## Part 2 Implementation

### (a) *OpenFileId Open(char \*name)*

首先，在 *ExceptionHandler* 的地方，多加上 SC\_Open 這個在 syscall.h 內有定義好的 case。進到這個 case 後，先從 Register4 拿出存放 filename 的 address，接著再用 readmemory 的方式，讀取欲打開的 filename。接著就呼叫一個我們寫的 system call *Open(filename)*，並且會讓他回傳 status，執行完會把 status 記錄在 register2 中。最後再處理 program counter。

我們在 ksyscall.h 中定義了一個會回傳 OpenfileId 的 Open 函數，在這個函數裡，他會直接回傳 *kernel->fileSystem->OpenAFile(filename)* 的回傳值。

接著在 filesys.h 中的 FILESYS\_STUB 中，完 *OpenAFile(filename)*。

首先，先讓要回傳的 id 值為-1，接著用迴圈去檢查 OpenFileTable 還有沒有空位，有空位就會讓 id = i。當迴圈結束若 id 仍為-1，代表現在能打開的 file 數量已經到上限了，所以就會 return -1 回去。

若 id 不為-1，代表我們可打開這個 file 並賦予他我們給予的 id 值，但由於要 new 一個 Openfile 的物件，我們需要真正的 file descriptor，我們選擇透過 sysdep.cc 中 *OpenforReadWrite(filename, crashOnError)*的回傳值來得到。

*OpenForReadWrite* 的參數 crashOnError 由於我們不希望他一打不開就 crash 掉，故設定 False。此外，此函數中是使用 linux 的 open 來實踐打開一個 file，linux 的 open 若沒成功打開 file 的話，會回傳一個-1 值。

所以在 *OpenAFile* 中，拿來接 *OpenforReadWrite* 的變數為-1 的話，代表沒有成功打開檔案，故我們 *OpenAFile* 的 function 也會直接 return -1。

若成功打開，我們便以拿到的 file descriptor，new 出一個新的 Openfile 在 OpenFileTable[id]的位置。最後 return id 回去讓 user 使用。這邊可以先特別注意到，這邊 Openfile 在 new 的時候，會把真正的 file descriptor 存到 private 變數 file 底下。

### **(b) int Close(OpenFileId id)**

首先，在 *ExceptionHandler* 的地方，多加上 SC\_Close 這個在 syscall.h 內有定義好的 case。進到這個 case 後，我們先從 register4 中讀取想要關掉的 OpenFileId。接著就呼叫一個我們寫的 system call *Close(OpenFileId)*，並且會讓他回傳 status，執行完會把 status 記錄在

register2 中。最後再處理 program counter。

我們在 ksyscall.h 中直接回傳 *kernel->filesystem->CloseFile(id)* 的值。

接著在 filesys.h 中的 FILESYS\_STUB 中，完成

*CloseFile(OpenFileId)*。首先先判斷 id 是否在 0-19 之間且

OpenFileTable[id]!=NULL，若其中一項不符合就直接 return -1 回去。

接著我們想使用 sysdep.cc 中的 Close function 來幫我們關掉

Openfile，由於這個 function 需要的是真正的 file descriptor，因此我們需要拿到在 Open 時有提到的，Openfile 裡面的 file (存真正的 file descriptor)，但由於他是 private 的，因此我們在 Openfile 的 class 裡再加一個 public function *int Getfd()*，讓他回傳 Openfile 的 file 回來。我

```
int success = Close(OpenFileTable[id]->Getfd());
```

們就會像下圖一樣來 Close file。

且 sysdep.cc 裡的 *close* 是用 linux 的 close 來關掉的，當回傳 0 代表成功關檔，回傳 -1 代表關檔失敗。於是我們就判斷 success 若  $\geq 0$  就回傳 1， $\text{success} < 0$  則回傳 -1 (這次题目的要求)。

### ***(c)Write/Read(char \*buffer, int size, OpenFileId id)***

首先，先將 SC\_Write 和 SC\_Read 的 assembly code 寫在 Start.S 裡，他會將 system call 的 code 和後來 return 回來的 value 都存在 r2，這些 assembly code 可以讓我們在不拉進整個 C library 的情況下，也可以 make system call 到 Nachos 的 kernal

```
.end Write
```

```
.end Read
```

下一步就是要 define 我們的 system call code，我們會將 write 和 read 都 define 在 sysall.h 中，他會透過剛剛的 assembly code 告訴 kernal 我們現在是要做什麼 system code

```
#define SC_Write 8
```

再來，就是到 kyscall.h 這個 kernal 的 interface 裡去定義我們的 SysWrite 和 SysRead，在這兩個 function 裡會分別去 call filsystem 的 Writefile 和 ReadFile，引數是由 exception.cc 傳入的 buffer，size 和 id，作完後會將結果直接 return 回去

在 filsystem 的 Writefile 和 ReadFile 裡，首先我們要去 handle 錯誤的狀況，我們要判斷這個 id 是否有超過 20 的 file limit，若超過就 return -1，再來要判斷這個 id 在 OpenFileTable 裡是否存在，若不存在一樣 return -1，以上兩項都正確後才可以做下面的 Write 和 Read 的部分，我們會利用傳入的 id 從 OpenFileTable 上找到我們要 write 或 read 的檔案，若讀寫正確的話，它會返回寫了或讀了多少個 character，反之則返回 0，所以如果接收到 0 我們就 return -1 回去，反之就 return 讀或寫的 character 數量回去

```
return result;
```

```
return result;
```

```
}
```

接下來，我們就可以去 exception.cc 裡增加 SC\_Read 和 SC\_Write 兩種 type 的 exception，再來我們需要拿到三種參數，buffer 可以透過 r4 從

mainMemory 已拿到，size 和 id 則分別直接從 r5，r6 裡拿，然後就可以去呼叫 SysWrite 或 Sysread 了，並將 buffer，size，id 當作引述傳進去，return 回來後要將結果存回 r2 裡，並 increment pc

```
int id = (int)kernel->machine->ReadRegister(6);
```

### Part 3 Difficulties encountered in the homework

1. 這是我們第一次需要 Trace 這麼大型的 code，之前程設的 project 要 trace 的也大概也不到這個的一半。並且在一開始對於一個作業系統的運作沒有那麼熟悉，對於 Code 要去哪裡找也沒什麼概念，所以我們在一開始的時候真的蠻徬徨的，不太知道到底從哪裡著手。如果沒有作業資訊的流程要我們一步一步 trace 下去的話，可能要看好久才知道要從哪裡開始。
2. 再來就是自己 implement 的時候，也找了好久要從哪裡開始做，也不太敢隨便直接動 code，很怕他整組壞掉，我們也找了一兩天才發現其實在 filesys.h 的檔案裡面有註解，類似提示我們是要從這邊開始的意思，我們才從這個線索開始慢慢推回去，然後也看到 syscall.h 裡面有 SC\_Open 等等都被註解起來，才完全大概瞭解這次作業的 implement 要從何開始。
3. 再來就是 debug 的時候，因為我們一開始就只有把我們的邏輯寫進去，但當然不可能一次就對，所以我們又完全不知道是哪裡出錯了，我們也是想了好久才想到，但我們覺得如果可以好好利用輸出一些 debug 訊息的話應該可以更快知道錯在哪，所以下次會試著多寫一點 For debug 的 code。

## Part 4 Feedback

這次作業剛開始的時候，發現單純從老師投影片理解的 `system call` 過程，與自己親自去慢慢 `trace code`，找出 `function` 和 `function` 之間的關聯性的感覺真的差很多，真正 `trace code` 起來才發現之前的觀念都太表面了，對於整個 `system call` 怎麼在 `code` 上呈現很不熟悉，以至於剛開始 `trace` 的時候速度很慢，要花比較多時間去理解一些 `function` 的作用是什麼，若沒有這個 `function` 會發生什麼事之類的，但在 `trace` 完一個 `system call` 後，會發現自己對這整個過程又更加了解了，之後 `trace` 的速度也會變快，我覺得這個作業很有意義，因為它可以幫助我們，要求我們，去把一些之前模糊的觀念弄的更清楚，我們也必須要弄清楚才能去寫之後的 `code`，不知不覺中就學到了蠻多東西的，謝謝辛苦的助教和老師