Operating System

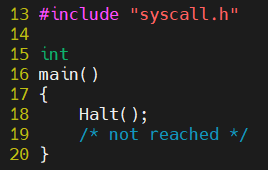
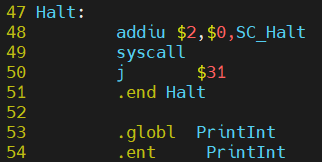
MP1 Report

組別：25

陳凱揚：Trace code、Implement、Report  
簡佩如：Trace code、Implement、Report

1. Trace code
2. SC\_Halt
3. /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。

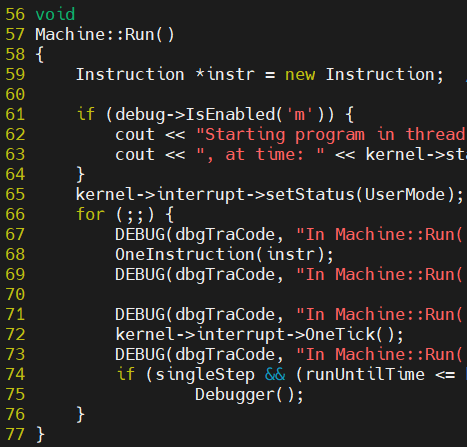
 

▲ 圖1-1 ▲圖1-2

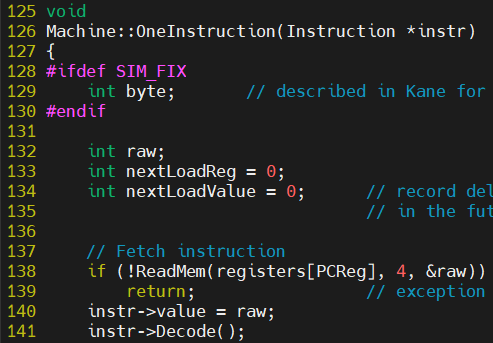
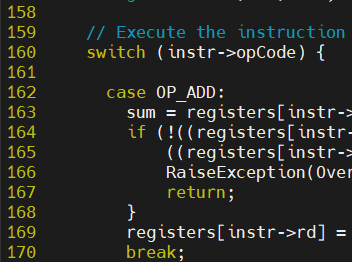
1. /machine/mipssim.cc

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。

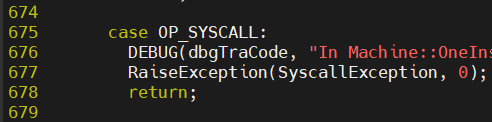
進到OneInstruction()後，會先去memory取得下一個要執行的指令，並Decode()得到指令的資訊（圖1-4），接著根據不同的指令對應到不同的做法（圖1-5），而SC\_Halt是一個system call，會對應到OP\_SYSCALL，並執行第677行（圖1-6）的RaiseException()。



▲ 圖1-3

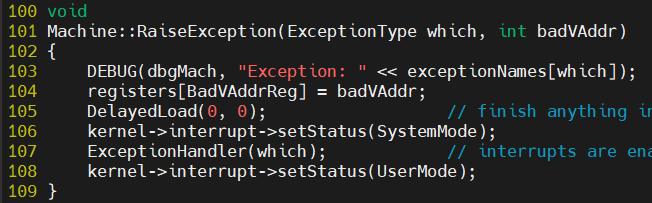
▲ 圖1-4 ▲ 圖1-5



▲ 圖1-6

1. /machine/machine.cc

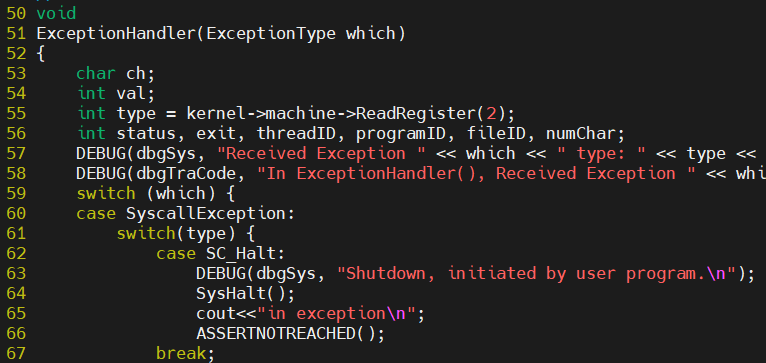
在RaiseException()裡，在進到107行（圖1-7）的ExceptionHandler()前，因為即將要執行system call或是exception，所以會先將kernel mode從UserMode轉換成SystemMode，而執行完後會設回UserMode。



▲ 圖1-7

1. /userprog/exception.cc

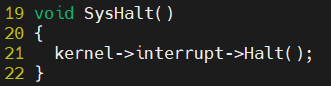
進到ExceptionHandler()之後，會先到r2找到步驟1存起來的type，再根據which和type，執行對應的處理方法。因此在第62行（圖1-8）可以找到SC\_Halt，並執行SysHalt()。



▲ 圖1-8

1. /userprog/ksyscall.h

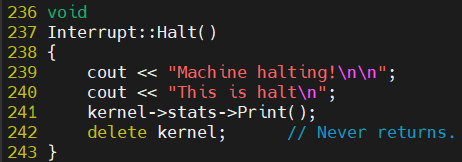
接著會進到ksyscall.h裡找到SysHalt()，並執行第21行（圖1-9）。



▲ 圖1-9

1. /machine/interrupt.cc

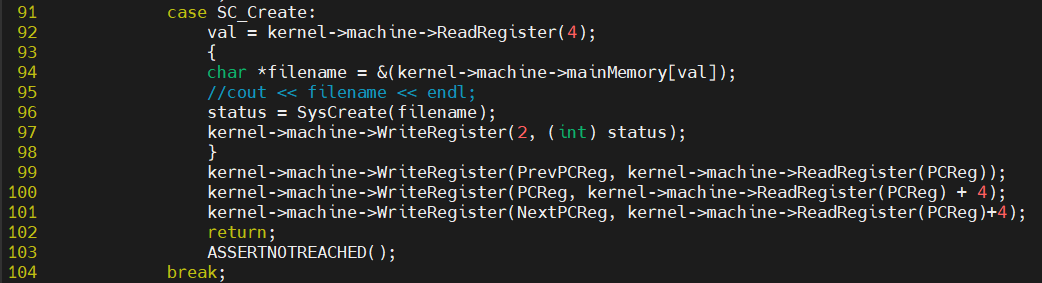
最後會執行Halt()這個interrupt，在程式結束前Print()印出一些統計的資訊（圖1-10），並delete kernel，程式就結束了。



▲ 圖1-10

1. SC\_Create
   1. /userprog/exception.cc

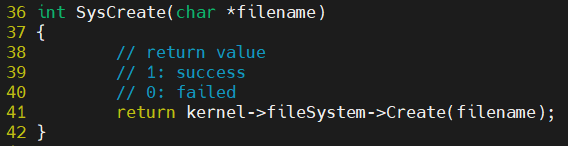
如同SC\_Halt，SC\_Create這個system call也會進到ExceptionHandler()中（圖1-11）。由於他含有一個參數，因此會先去r4找出這個參數(即”指向filename的指標”)，透過指標從kernel的mainMemory得到字串所在位置，並傳回第一個字元的位址給filename。接著就以filename為參數去執行ksyscall.h裡的SysCreate()，並回傳執行後的return值，將這個值存回r2裡。最後需要更新Program counter，將現在的PCReg寫入PrevPCReg、PCReg和NextPCReg各+4，使下次執行時可以得到下一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



▲ 圖1-11

* 1. /userprog/ksyscall.h

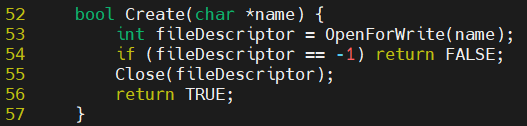
在這邊會執行第41行（圖1-12），kernel裡面的fileSystem提供的Create()，如果成功即回傳1，失敗則回傳0。



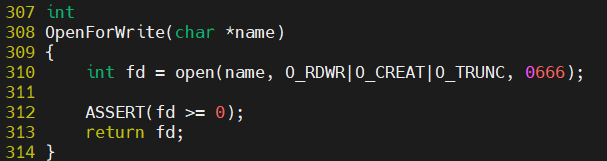
▲ 圖1-12

* 1. /filesys/filesys.h -> /lib/sysdep.cc

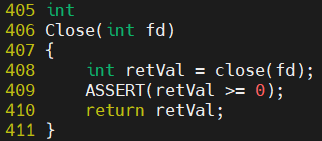
而上一步驟會進到filesys.h裡的Create()，並執行第53行（圖1-13）裡中位於sysdep.cc裡的OpenForWrite()（圖1-14），這是一個使用unix的open()作為開啟檔案的方式，並會回傳檔案是否成功開啟，-1代表失敗，接著呼叫同樣位於sysdep.cc裡的Close()（圖1-15），使用unix的close()來關閉檔案，並回傳TRUE。



▲ 圖1-13



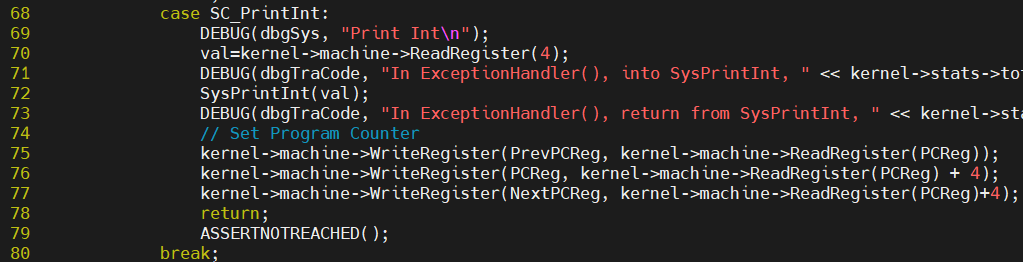
▲ 圖1-14



▲ 圖1-15

1. SC\_PrintInt
   1. /userprog/exception.cc

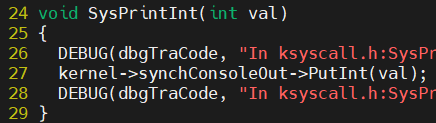
這次進到ExceptionHandler()，會來到SC\_PrintInt的地方（圖1-16），同樣會先去r4取出第一個參數，而這裡與SC\_Create不同的是不需要return值，因此不用將結果存進r2，只要執行ksyscall.h裡的SysPrintInt()，並更新Program counter。



▲ 圖1-16

* 1. /userprog/ksyscall.h

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

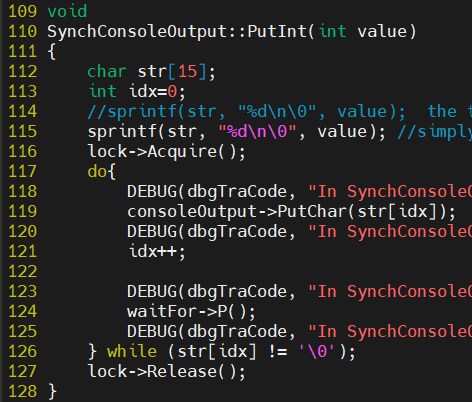


▲ 圖1-17

* 1. /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。

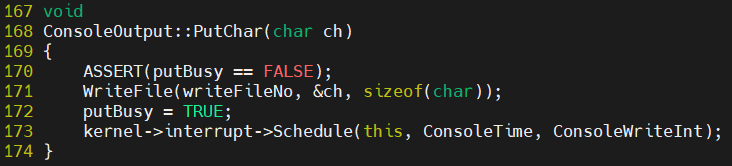
在輸出過程，會使用consoleOutput的PutChar()，一個一個將字元印出，而每印一個字元後，會呼叫waitFor->P()，waitFor是一個semaphore，有兩種功能，P()會等待直到value > 0，再value--，V()則是會叫起一個等待中的thread，再value++。在這邊的功能是等待前一個字元輸出完成後，再去執行輸出下一個字元。



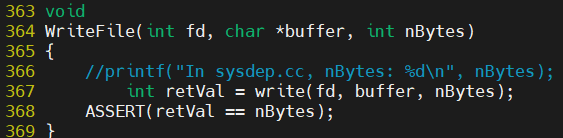
▲ 圖1-18

* 1. /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。



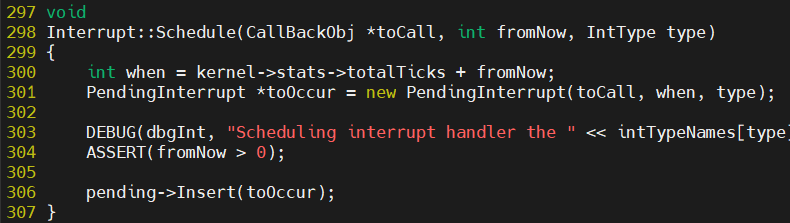
▲ 圖1-19



▲ 圖1-20

* 1. /machine/interrupt.cc

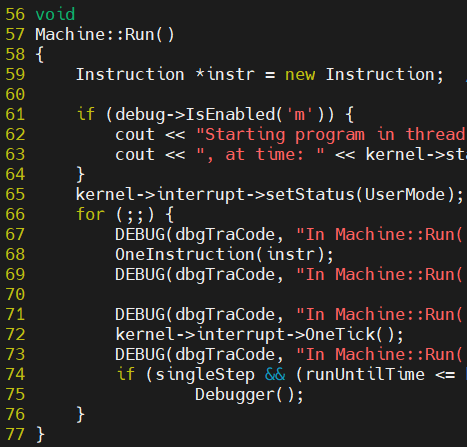
Schedule()裡會new一個PendingInterrupt，並確認fromNow > 0，確保這個interrupt發生的時間點在未來，接著加進pending裡依照時間順序做排程（圖1-21）。



▲ 圖1-21

* 1. /machine/mipssim.cc

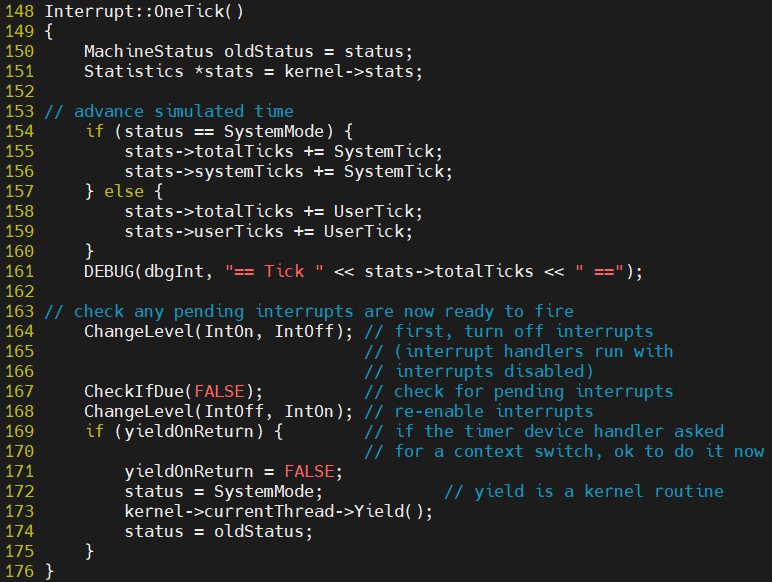
在Run()的第72行（圖1-22），每次執行完一個user instruction，都會呼叫interrupt的OneTick()，來增加simulated time並檢查是否有interrupt需要執行。



▲ 圖1-22

* 1. /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。

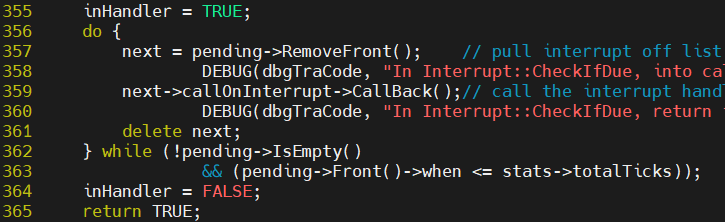


▲ 圖1-23

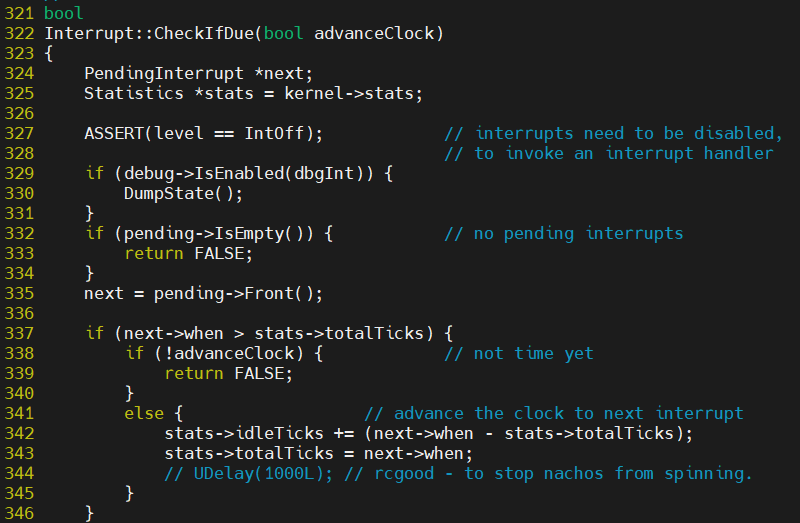
* 1. /machine/interrupt.cc

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。



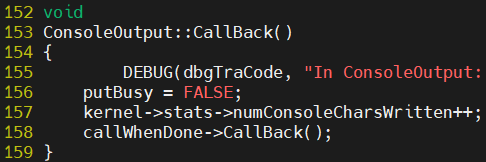
▲ 圖1-24



▲ 圖1-25

* 1. /machine/console.cc

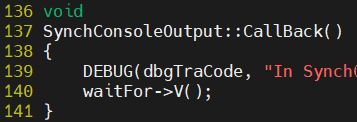
回到剛剛輸出的地方後，因為上一個字元已經輸出完畢，因此將putBusy設為FALSE（圖1-26），並呼叫callWhenDone的CallBack()告訴他輸出已經結束，可以叫下一個字元準備輸出了。



▲ 圖1-26

* 1. /userprog/synchconsole.cc

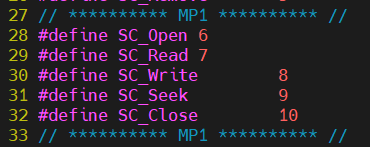
回到這邊後，就會呼叫waitFor這個semaphore的V()，將val++，並叫醒等待中的下一個thread，使其可以進行輸出，也就是使圖1-18第124行的waitFor()->P()能夠繼續往下執行。



▲ 圖1-27

1. Implement
2. /test/syscall.h

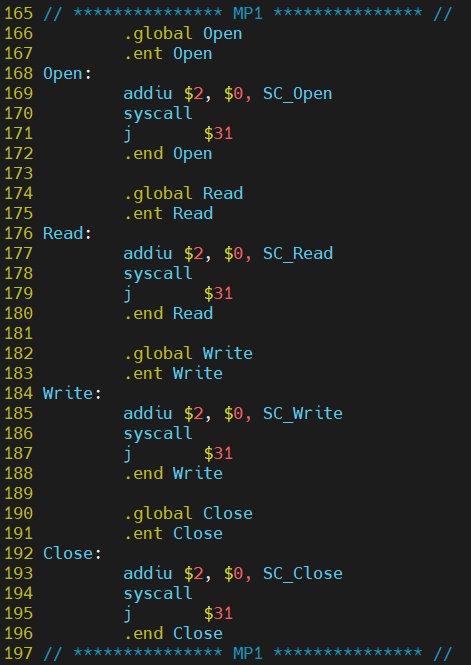
在syscall.h裡，我更改了第28、29、30、32行（圖2-1），將註解掉的部分刪去，define這些system call的指令，而Open()、Read()、Write()、Close()等函式，下方已經有先宣告了，所以不需要再自己加上。



▲ 圖2-1

1. /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。



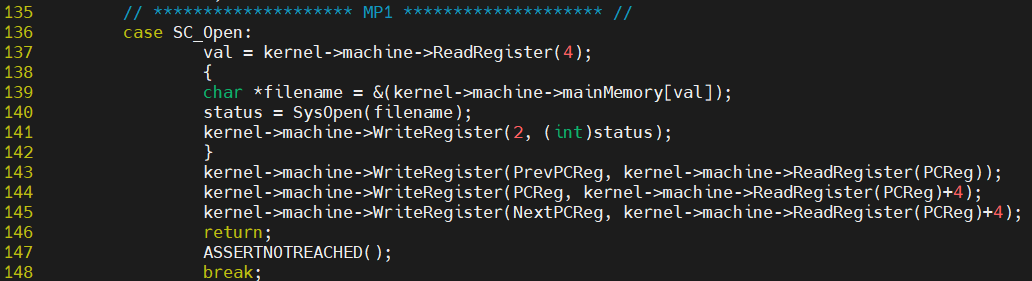
▲ 圖2-2

1. /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

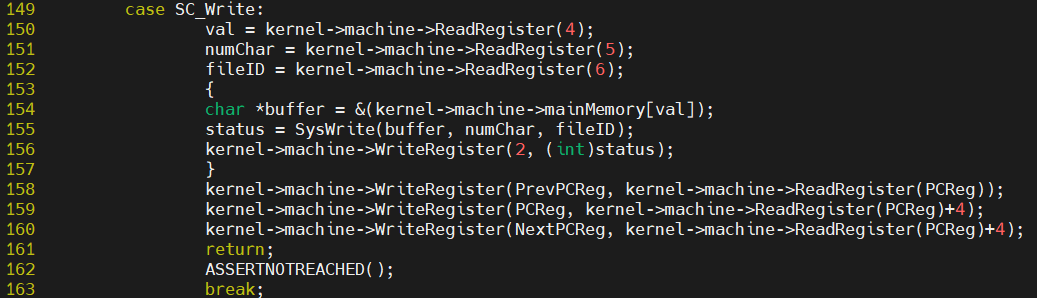
首先先從$4裡讀出第一個參數值，由於第一個參數的type為char\*，也就是一個字串的第一個字元的位址，因此需要以val去mainMemory找出這個位址並回傳給filename，接著就能呼叫kernel interface裡的SysOpen()（位於ksyscall.h）來執行這個system call，並將回傳的值存回r2中。最後需要更新Program counter，將PrevPCReg、PCReg、NextPCReg的位置都+4，使下次可以執行新的instruction。



▲ 圖2-3

* 1. SC\_Write

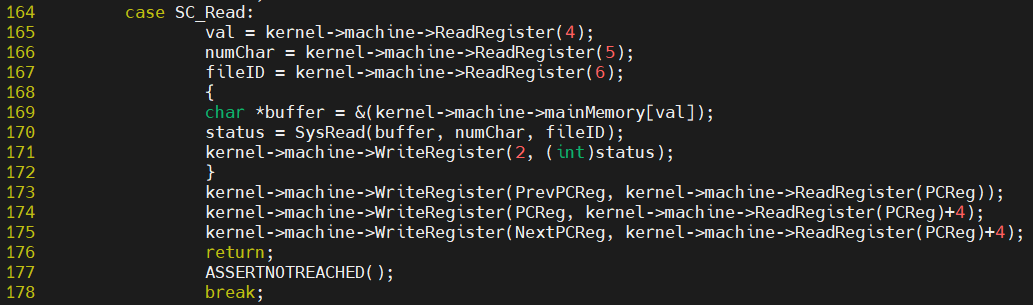
由於這個指令包含三個參數，因此依序從r4、r5、r6找到val、numChar、fileID等參數，同樣的需要去mainMemory找到第一個參數buffer的位址，接著呼叫SysWrite()執行，將回傳值存入r2，最後更新Program counter。



▲ 圖2-4

* 1. SC\_Read

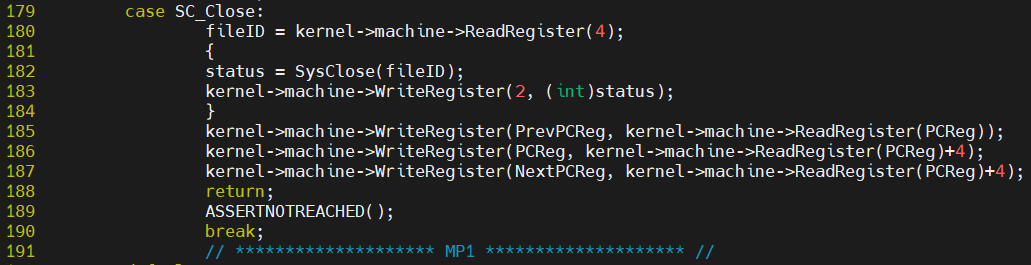
與SC\_Write幾乎相同，參數也完全相同，只差在呼叫的是SysRead()。



▲ 圖2-5

* 1. SC\_Close

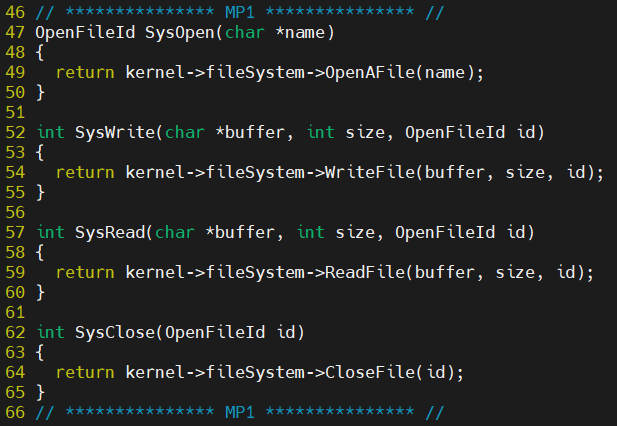
同樣的讀取第一個參數，並呼叫SysClose()，其他步驟皆相同。



▲ 圖2-6

1. /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。



▲ 圖2-7

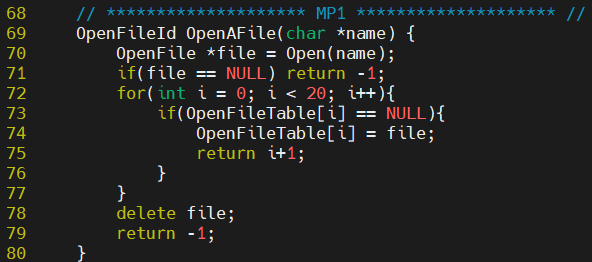
1. /filesys/filesys.h

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。

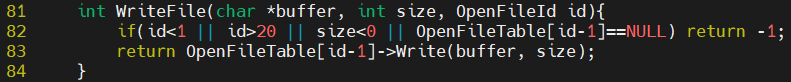
因此若得到回傳值是NULL時，就直接回傳 -1，代表檔案開啟失敗，若成功開啟，就去OpenFileTable找一個未使用的位置，放進這個開啟的檔案，並回傳OpenFileId（需要加1，因為id不得為0）。如果發現整個table都沒有空位，代表檔案開啟已到上限，這時delete掉剛剛new出來的OpenFile物件，並回傳 -1。



▲ 圖2-8

* 1. WriteFile()

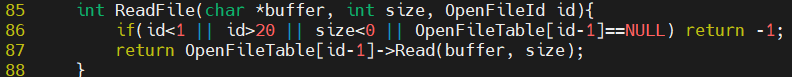
首先先檢查不合法的情況，包括id錯誤、size為負數、指向的OpenFile是NULL，發生這些情況就會回傳 -1，代表寫入失敗。接著會呼叫OpenFile物件裡提供的Write()，底層使用了sysdep.h裡unix的write()功能，最後會回傳實際寫入的字元數。



▲ 圖2-9

* 1. ReadFile()

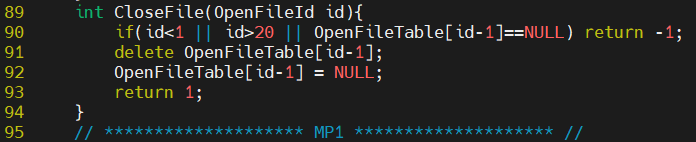
與WriteFile()一樣，先檢查不合法的情況，發生時就回傳 -1。接著呼叫OpenFile物件裡提供的Read()，並回傳實際讀到的字元數。



▲ 圖2-10

* 1. 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。



▲ 圖2-11

1. 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。