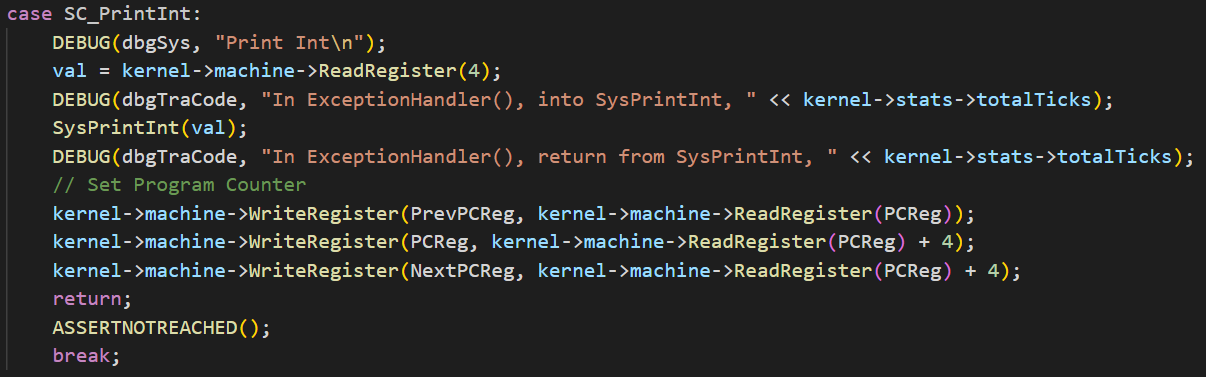
作業系統

組別:8

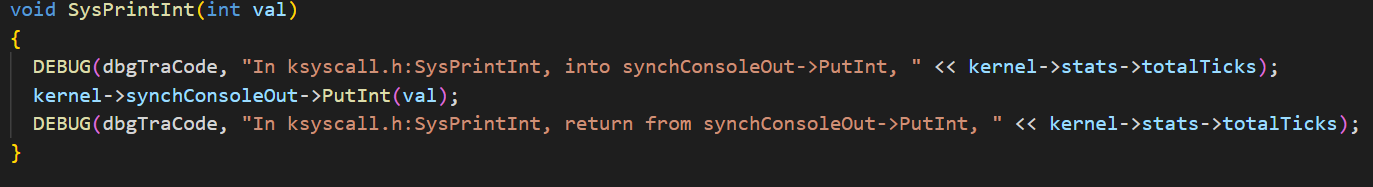
組員:洪聖祥、張紘齊

(c) Trace SC\_PrintInt

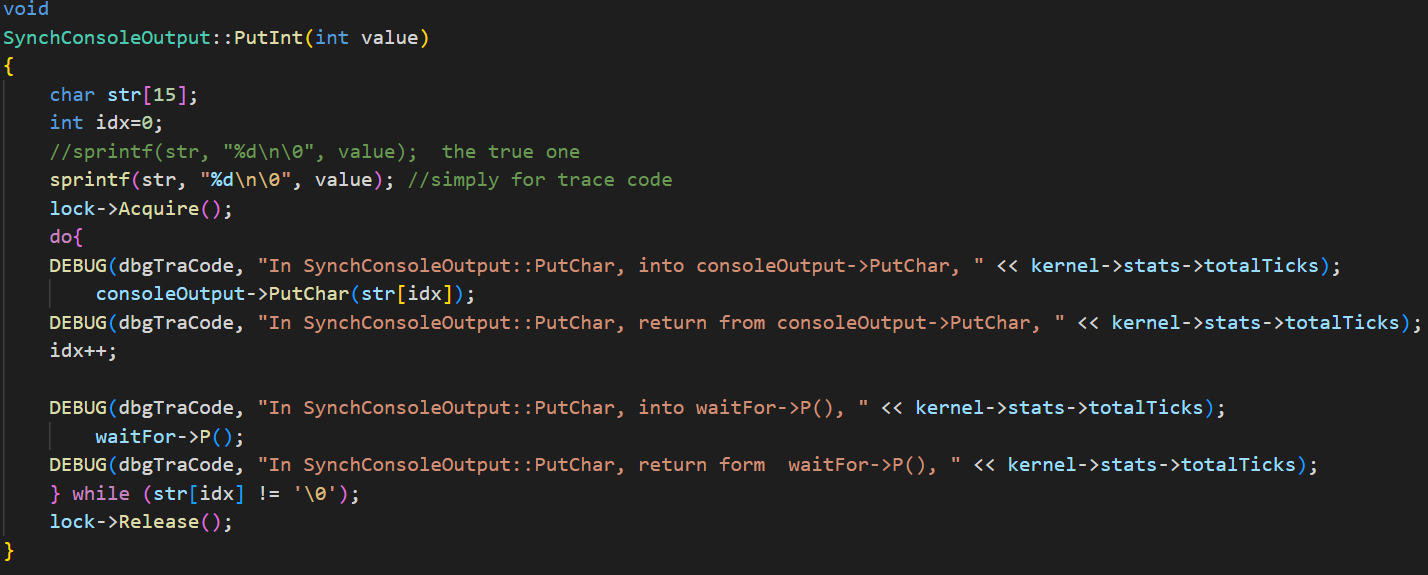
在exception.cc，跟SC\_Halt、SC\_Create一樣，判斷是哪一個exception type，並執行對應exception type 要做的事。SC\_PrintInt 讀r4 register (只有一個參數，存在r4 register中)，讀出來的是要印出來的integer。接著把讀出來的值放進SysPrintInt(val)執行。最後更新PC register。



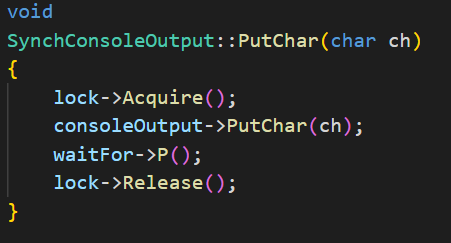
在ksyscall.h，SysPrintInt()執行kernel->sychConsoleOut->PutInt(val)



在synchconsole.cc，PutInt(int value)先建立一個char array str，並用sprintf把integer value 轉成 character array 存進str。接著取得 SynchConsoleOutput class的lock。Lock是一種同步化方法，假設有2個程式要access 同一個resource(可能是memory，或是這個例子是console)，第一個程式更改resource 的data，但第二個程式沒有讀到更新後的resource，這樣導致data inconsistency。為了避免這樣的情況發生，resource會有一個lock，程式要取得lock才能對resource進行更改。更改完後要release lock，這樣第二個程式才能正確讀到更新後的值。(ref: <https://web.mit.edu/6.005/www/fa15/classes/23-locks/>)。取得SynchConsoleOutpu的lock後，用while loop一個一個把character 印出來，執行consoleOutput->PutChar(str[idx])，waitFor->P()是用來讓後面還沒作用的字元先等待。最後才 release lock。



SynchConsoleOutput::PutChar(char ch)跟PutInt(int value)不一樣的地方是指印出一個字母。

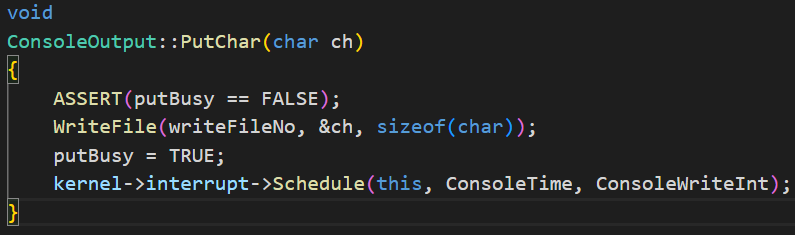


在console.cc 中，PutChar(char ch) 先將字元寫入stdout，用debug功能得知writeFileNo=1。而writeFileNo 1是stdout。(file descriptor 0 1 2 是stdin, stdout and stderr)

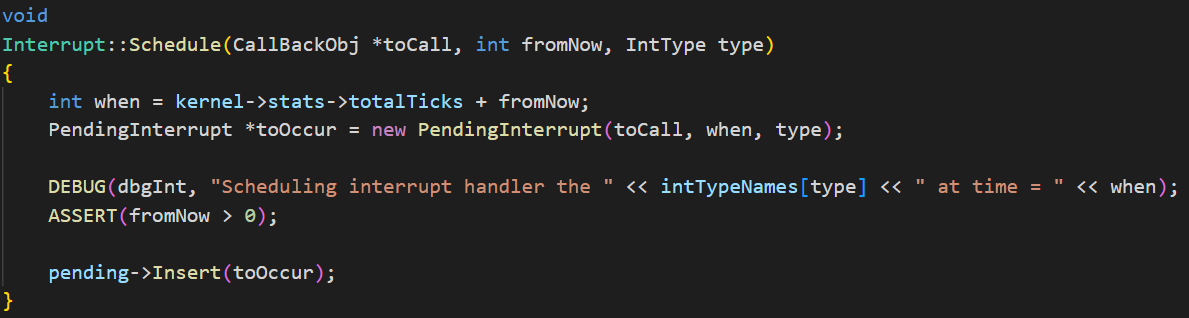
這裡DISK\_0模擬顯示器



將putBusy改成TRUE，讓其他事情不能做。進入interrupt.cc 中的Schedule，安排程式被CPU執行。Interrupt handler 要被預排所以進入kernel->interrupt->Schedule()



在interrupt.cc 中，Schedule()安排callback object要何時被執行。when是現在時間再加上ConsoleTime，得出interrupt handler要執行的時間點。並把它放進一個sorted list(依時間排列，模擬依序執行interrupt handler)。



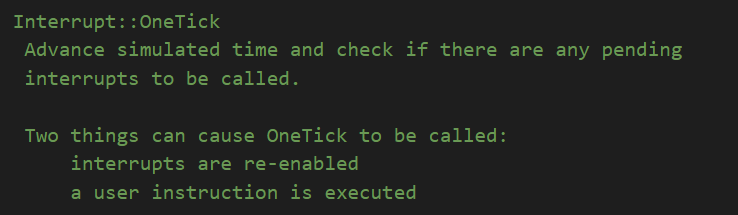
從stats.h中可以知道ConsoleTime是100，模擬字元顯示在顯示器所需的時間。



在mipssim.cc中，跟在SC\_Halt解釋一樣，用一個for loop依時間一一執行每一個instruction，在執行完OneInstruction()後，會進到interrupt.cc裡的OneTick()函數

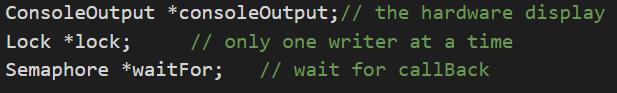


在interrupt.cc 中，OneTick()會讓系統時間往前一個時刻，來模擬時間往前的行為。剛剛說ConsoleTime = 100，所以可以跳到現在時間+100。從OneTick()註解中可以知道2種情況OneTick()被執行，一是interrupt 重新啟用(off->on)或是instruction被執行。

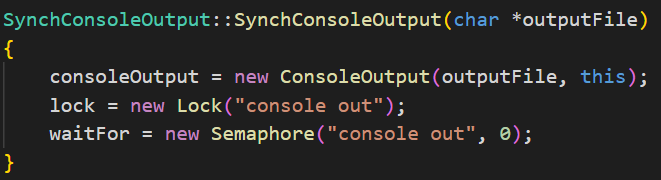


回到synchconsole.cc，PutInt()執行完console->Putchar()後執行waitFor->p()。這個waitFor是一個叫semaphore的class。Semaphore 其實也是一個同步化的機制。簡單來說semaphore是一個counter，當counter == 0時，則thread 等待直到counter > 0。

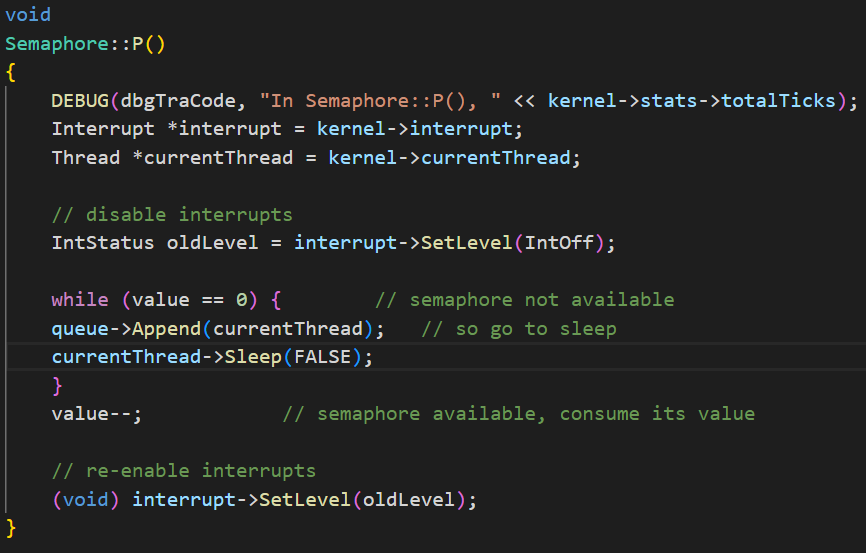


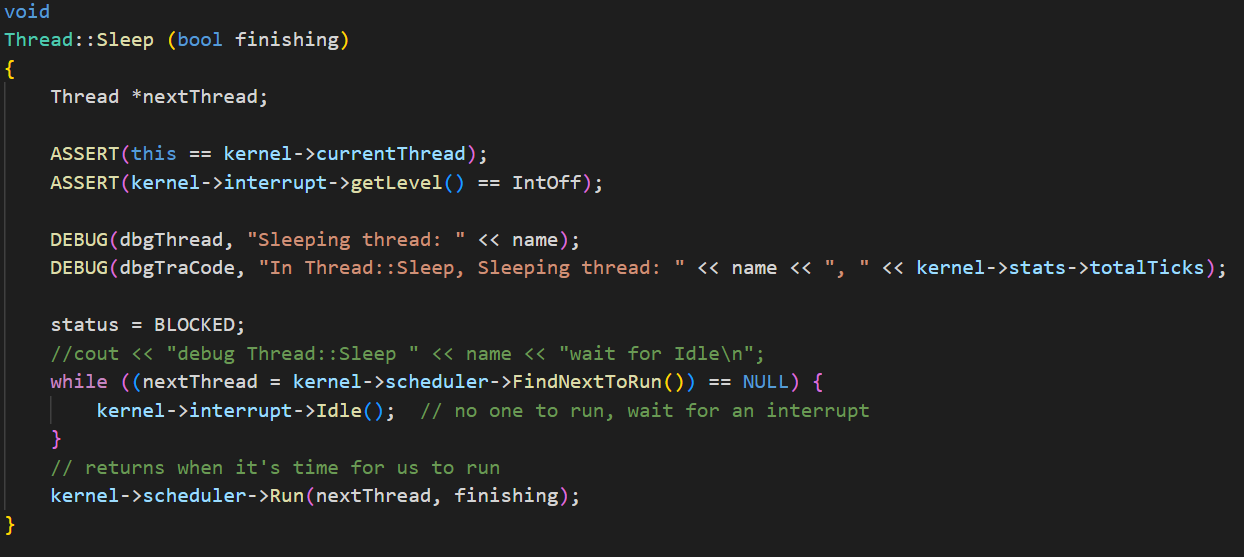


而可以從SynchConsoleOutput的constructor得知，waitFor semaphore初始值是0。

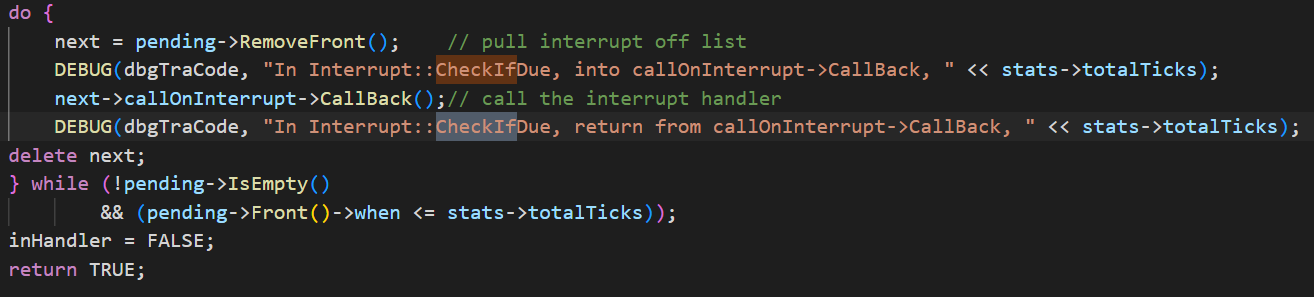


在synch.cc中Semaphore::P()，會停用interrupt，如果value == 0，會把現在的thread push到queue，並讓current thread sleep。

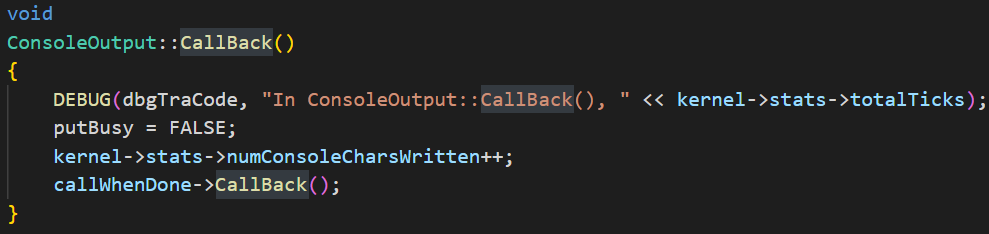
在thread.cc Thread::Sleep()，如果current thread 因為synchronization 機制(semaphore、lock) 被阻擋，其他thread會wake up這個thread然後回到ready queue。



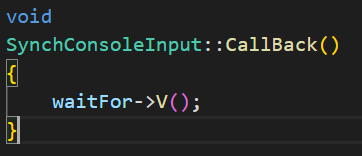
從Interrupt::Idle()會檢查有沒有預排的interrupt handler，這個function是CheckIfDue()。如果確定sortedlist中有預排的interrupt handler，用do while loop 一個個把interrupt handler pop出來，然後執行interrupt handler

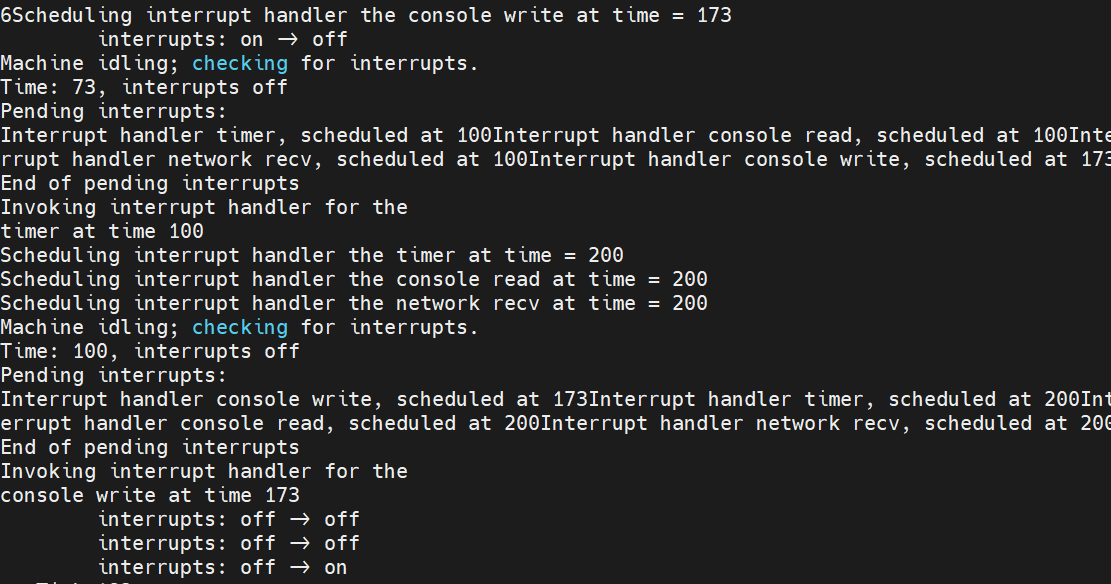


接著當下一個字元可以輸出到顯示器時，putBusy更新成FALSE。並呼叫callWhenDone->CallBack()



在Synchconsole.cc，CallBack()呼叫waitFor->V()。表示semaphore的值+1。讓thread ready to run。

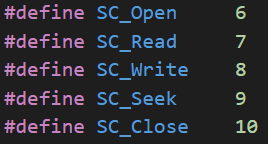




Part II-2: Implementation four I/O system calls in NachOS

syscall.h

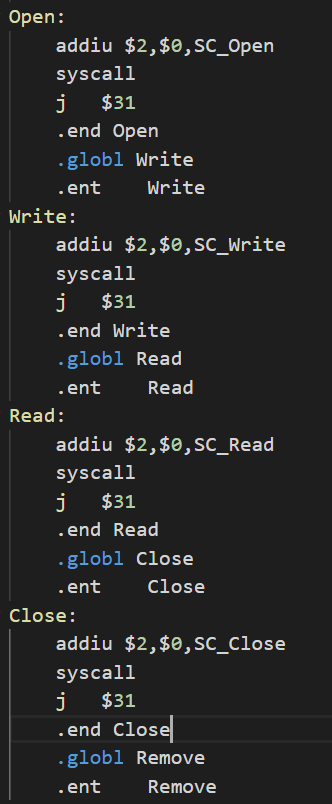
先把 SC\_Open 、 SC\_Read、SC\_Write、SC\_Close 四行註解拿掉。



start.S

照著之前的assmebly code 寫Open、Write、Read、Close。

因為在exception.cc 中ExceptionHandler 會讀取r2 register 來得知exception type，addiu $2,$0, SC\_Open 是把 SC\_Open 這個 system call exception type 放進 r2 register。

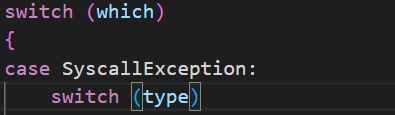


exception.cc

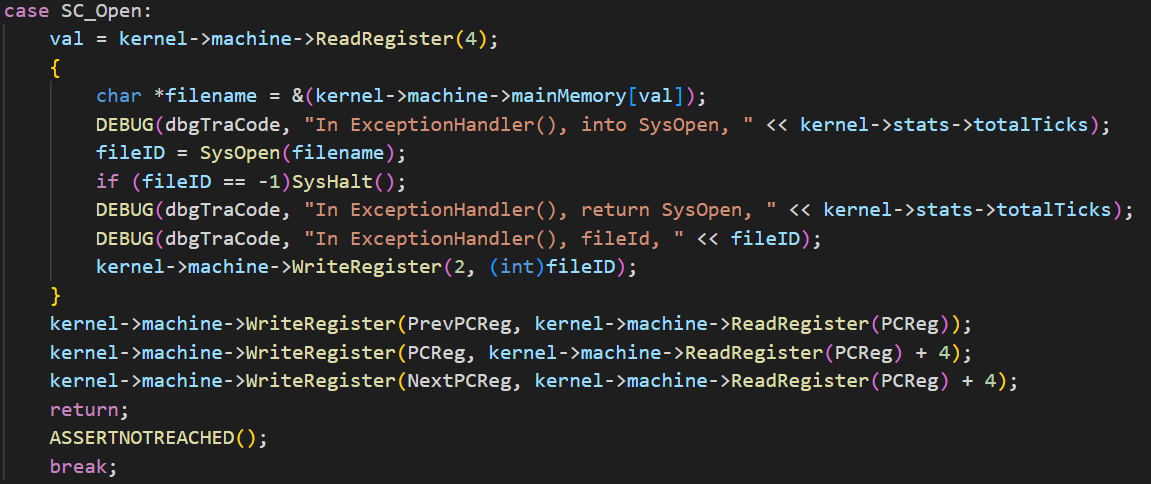
如同上面所講的，ExceptionHandler會讀取 r2 register。在 start.S中已經把Open、Read、Write、Close四個system call type放進r2 register。這裡就會讀取是哪一個type。



並且用switch case 判斷是哪一種exception，並且exception是哪一種type。

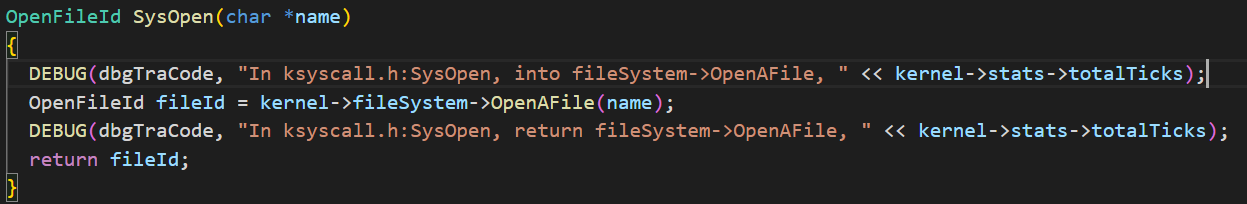


接著寫type是SC\_Open、SC\_Read、SC\_Write、SC\_Close這四種狀況所需要做的事。system call 的參數都是依序存在r4、r5、r6、r7中。以SC\_Open為例，因為Open()這個system call只會傳入一個參數(一個 string 或者說char array)。所以只有讀取r4 register，讀取內容是記憶體位置。接著得到參數的記憶體位置。並呼叫SysOpen(filename)。

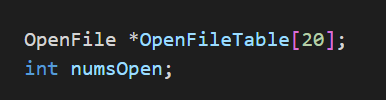


1. Implement Open a file

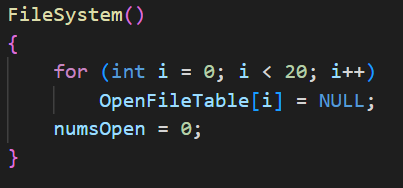
SysOpen 呼叫 kernel->fileSystem->OpenAFile(name) 傳回 openFileId。



因為Spec上要求開啟檔案的上限是20個，所以多寫一個變數numsOpen紀錄有多少個file open。



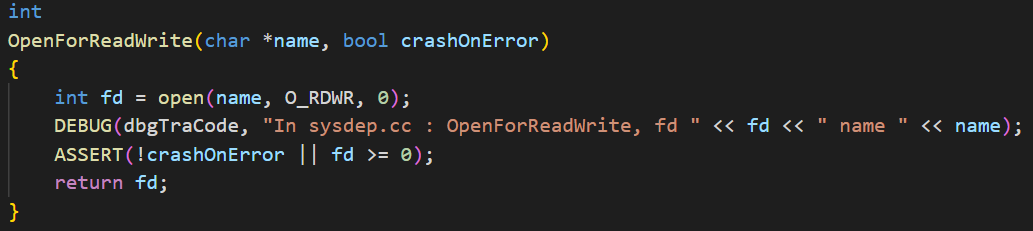
並且在FileSystem constructor初始化numsOpen為0。



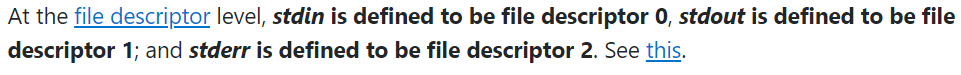
OpenAFile 用 OpenForReadWrite實作。OpenForReadWrite會回傳file descriptor。如果開啟檔案的數量超過20個或是無法開啟檔案，會回傳

-1。Spec上要求用OpenFileTable 的 entry number當作 OpenFileId，而且file descriptor從6開始。如果開啟5個檔案，file descriptor 有 6 7 8 9 10。如果關閉 8 file descriptor，再開啟一個檔案，開啟的file descriptor是8而不是11。也就是說file descriptor是根據最小的值分配的。因此，return 的 OpenFileId是 file descriptor -6。接下來就要instantiate OpenFile class 並存在OpenFileTable中。所以用OpenFileId access OpenFileTable會得到 OpenFile class，再class 中會存真正的file descriptor。

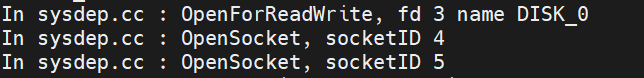




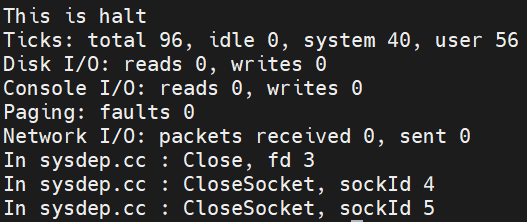
上面說的file descriptor從6開始其實有點不正確。其實file descriptor是從3開始: (從stackoverflow 查到)



因此我用debug的功能從把sysdep.cc中OpenForReadWrite、OpenForWrite、Close、OpenSocket、CloseSocket印出來file descriptor。因為這些function中包含了open() close() socket()會回傳file descriptor。把這些file descriptor 的值印出來。一開始會打開DISK\_0這個檔案，file descriptor 是3。Socket也會return file descriptor 4 5。

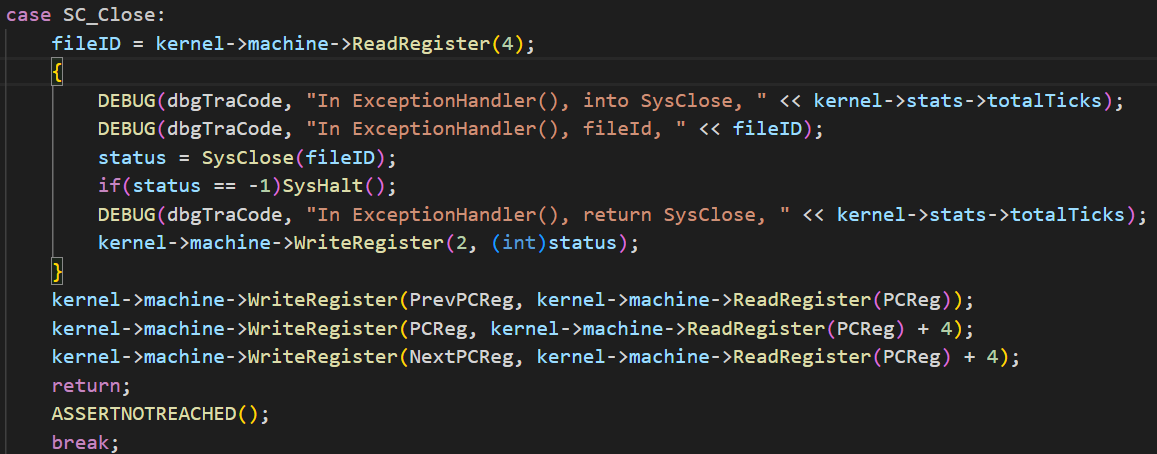


在程式執行完machine halt之後才會close file descriptor 3 4 5。所以可以安心的在system call，file descriptor從6開始。

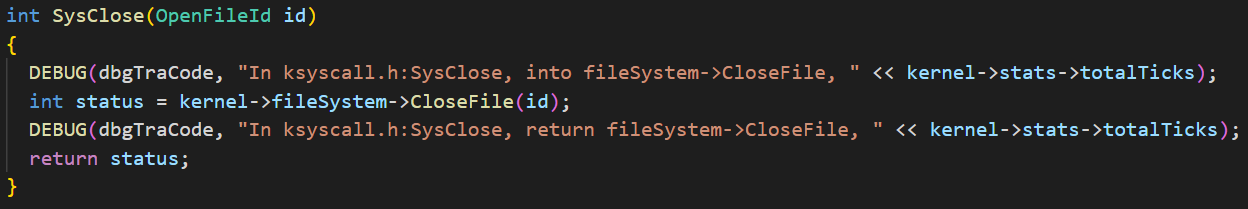


4. Implementation of closing a file

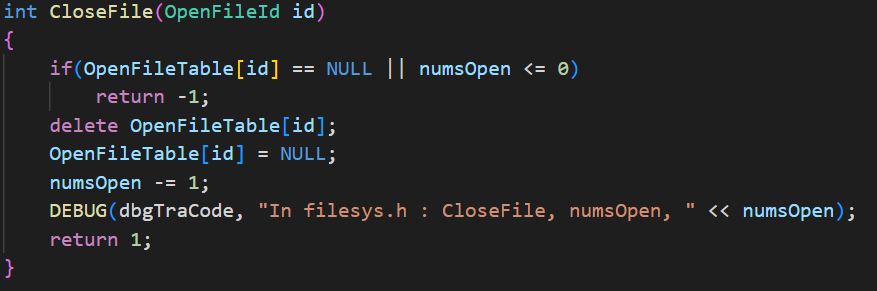
System call Close() 跟Open() 很類似，先從r4 register 讀取參數。參數是Open() 回傳的 OpenFileId。接著呼叫SysClose()。



在ksyscall.h中 SysClose()呼叫kernel->filesystem->CloseFile(id)



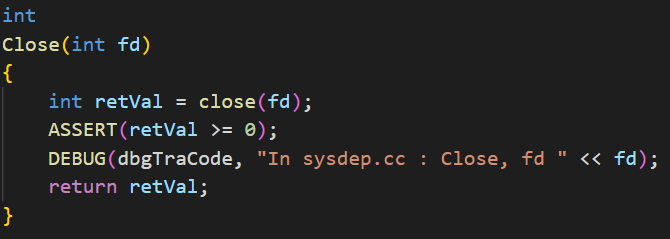
在filesys.h中，CloseFile(id) function會先檢查是否這個OpenFileId 是開啟的檔案，如果不是已經開啟的檔案回傳-1。檢查方法是在OpenFileTable中看這個id是否是NULL(表示OpenFileId沒有對應的OpenFile class)。也會檢查開啟的檔案數量，小於0也會回傳-1。接著delete OpenFileTable[id] 指到的指標。最後numsOpen-=1，更新開啟檔案數量-1。



在delete OpenFileTable[id] 所指到的指標，其實會呼叫OpenFile的destructor。在destructor中會呼叫Close(file)，file是真正的file descriptor不是OpenFileId。所以在filesys.h中CloseFile()中不需要再呼叫sysdep.h Close()，不然會呼叫2次，導致錯誤。



sysdep.h Close() 呼叫 sys/file.h中的close()。



Difficulty:

洪聖祥:

我覺得最難的部分是trace code第三題，有些觀念是老師還沒有教過的，像是synchronization的機制。所以必須上網查甚麼是lock、semaphore。而且有時候執行的順序跟我的想法對不上，還好有debug功能，讓我可以清楚知道哪一行執行後執行哪一個涵式。