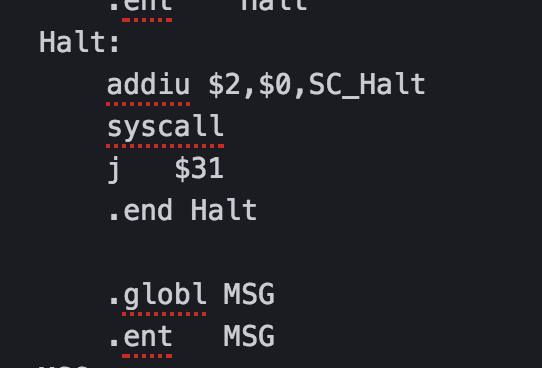
1. ***Trace system call***

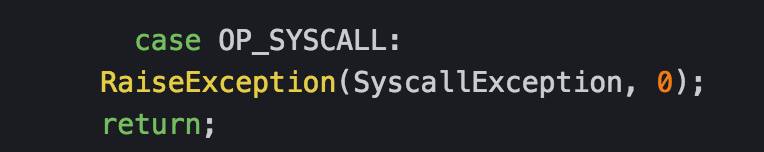
a. Halt(): 最凶狠的system call

以testcase : halt.c為例，當我們include”syscall.h”之後，我們就可以使用syscall中halt()這個function。當這個c檔在compile的時候遇到halt()會轉換成start.S中以下這段組合語言：



如此Mips Machine Simulator跑到這行instruction─ SC\_Halt時會先將這個代碼會存在$2裡面，接下來執行syscall。

接下來看到mipssim.cc，這個檔案會模擬cpu，在Run()這個function中有一個無線迴圈不斷地執行OneInstruction()，而OneInstruction()就是依照PC\_counter去nachos的memory中拿一行instruction出來進行decode，如果出來的opCode符合下面這個case的話：



就會RaiseException()，並傳入SyscallException這個ExceptionType。

RaiseException()這個function是在machine.cc中，首先它會把mode換成SystemMode(也就是kernel mode)，執行ExceptionHandler()並傳入ExceptionType(這邊是SyscallException)，執行完ExceptionHandler()後會將mode換回UserMode。

ExceptionHandler()會在Exception.cc中，首先他會從$2中讀出type(這邊是SC\_Halt)，接下來看傳入的ExceptionType，如果是SyscallException的話就去看type是否符合其中一項case，在這邊符合：



並執行其中的SysHalt()。

SysHalt()在ksyscall.h裡面：



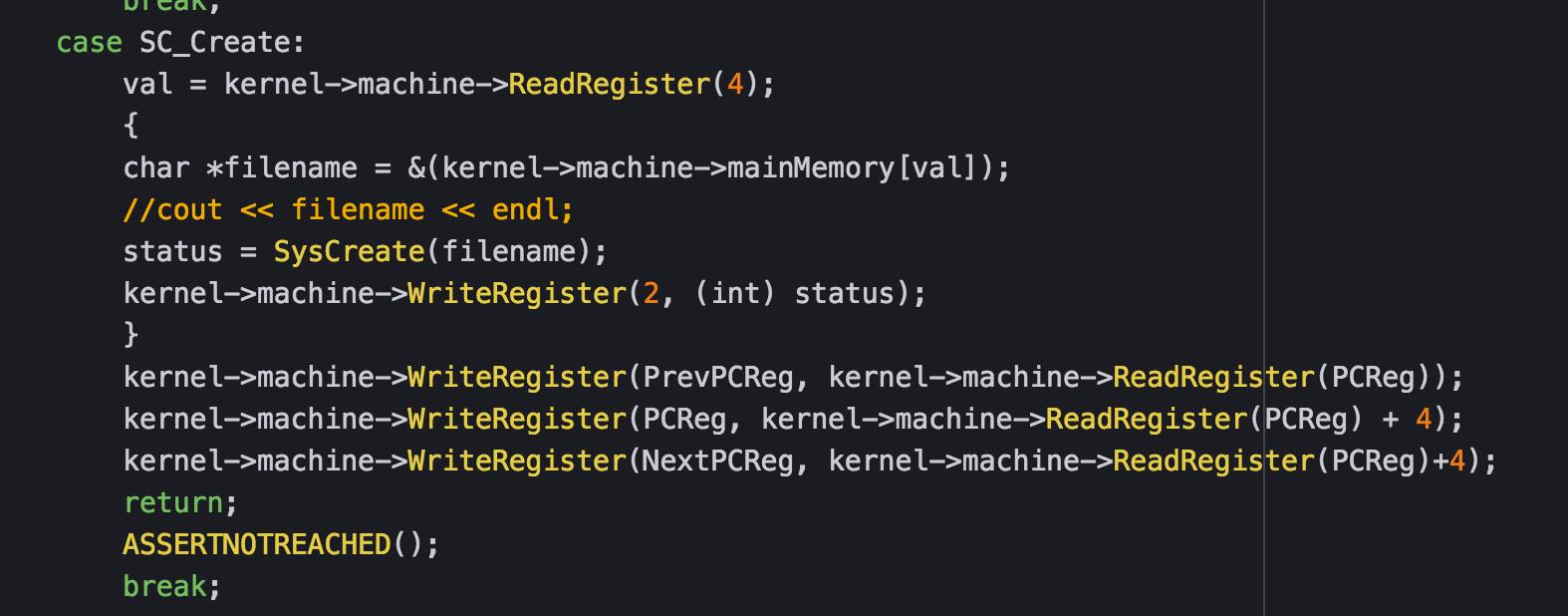
而裡頭就是執行kernel裡的interrupt物件的Halt()，實作細節在interrupt.cc中：



這裡會把kernel delete掉，也就是結束nachos 作業系統的運作。

b. Create(): 檔案之母

前面的流程跟Halt()一樣，不同點在於exception.cc在看syscallException Type時，會match到下面這個case：

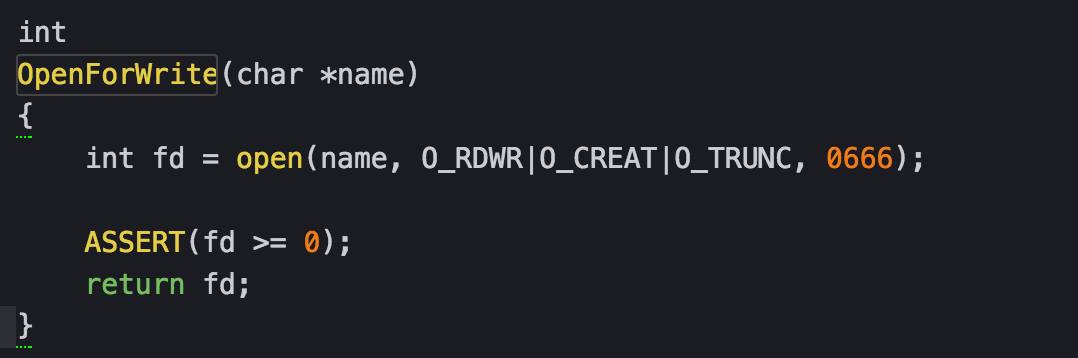


首先會從$4讀出Create()傳入的參數(指向檔名字串的指標)，該指標紀錄的是在 Nachos 的mainMemory的位址，我們還須要把他轉成這個字串在底層linux系統中真正的位址，並將他給filename，再呼叫SysCreate()並傳入filename。

接下來的流程跟Halt()大同小異，並一路傳到kernel.cc。在kernel.cc中會呼叫fileSystem的Create()。在filesys.h中：



在Create()中會呼叫在sysdep.cc中的OpenForWrite()：



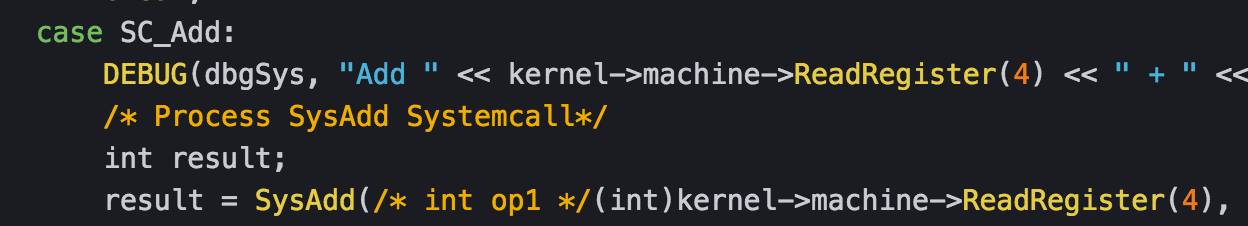
OpenForWrite()會呼叫底層Linux的open()並回傳fileDescriptor。而上一層的Creata()會依據fileDescriptor來回傳表示是否有create成功的布林值並呼叫Close()來關閉檔案，Close()會使用到sysdep.cc中底層Linux的close()。

而Create()回傳的布林值會一路回傳到exception.cc，然後會將這回傳的布林值寫入$2，最後將現在的PC寫入register[Prev

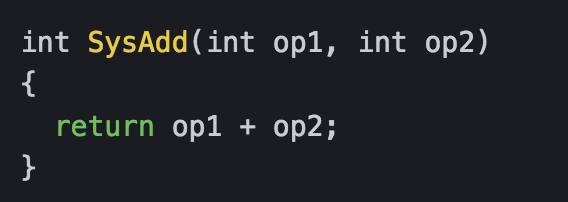
PCreg]，現在的PC+4並寫入register[PCreg]， PC再+4寫入register[NextPCreg]，最後再一路回傳到mipssim.cc繼續抓下一個instruction。

c. Add(): 媽的智障

前面流程都跟前兩個一樣，不同點也是在exception.cc，會match到下面這個case：



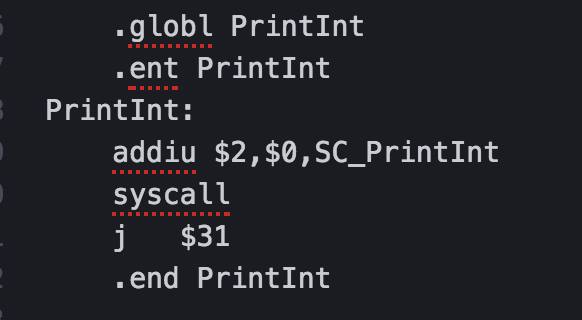
而其中會call SysAdd()，再看到SysAdd()所在的ksyscall.h：



Return之後就跟Create()一樣，將結果寫在$2，並maintain PC的資料。

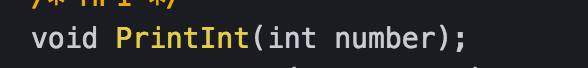
1. ***Report***
2. PrintInt():

在Start.S中要加入下面這段code：



並在syscall.h中宣告：

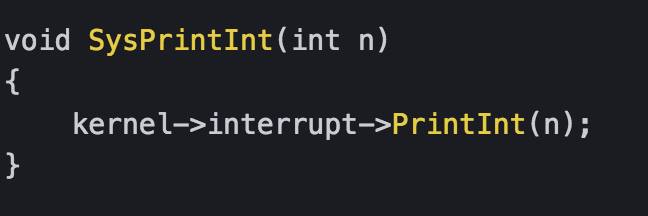




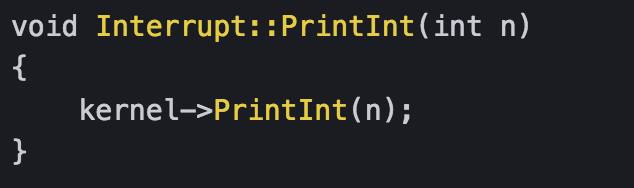
流程跟前面的system call一樣，不同點一樣是從exception.cc開始，match的case如下：



首先從$4讀出傳入的參數，這參數就是要印出的int，再呼叫在ksyscall.h創立的SysPrintInt()並傳入這個int：



再到Interrupt.cc：



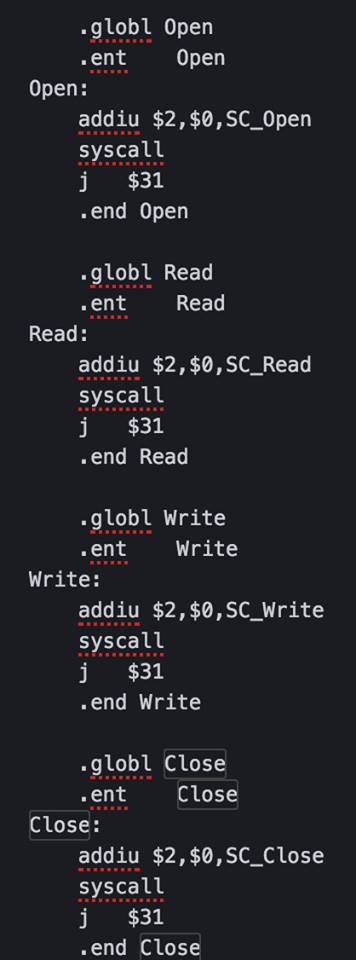
最後到kernel.cc中創立PrintInt()：



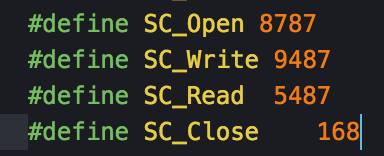
這邊將傳傳入的數字依照位數一個一個取出，並使用synchConsoleOut的PutChar()將一個一個數字印出到console上，比較須注意的正負數及0也有作處理。之後會一路回到exception.cc，並如同前面的system call，接下來會maintain PC的資料，最後再回到mipssim.cc抓取下一個instruction。

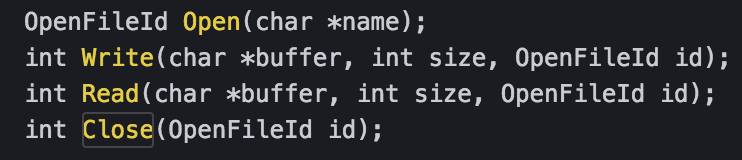
1. Open() , Read() , Write() , Close():

一樣在start.S要加入:



syscall.h也要宣告:

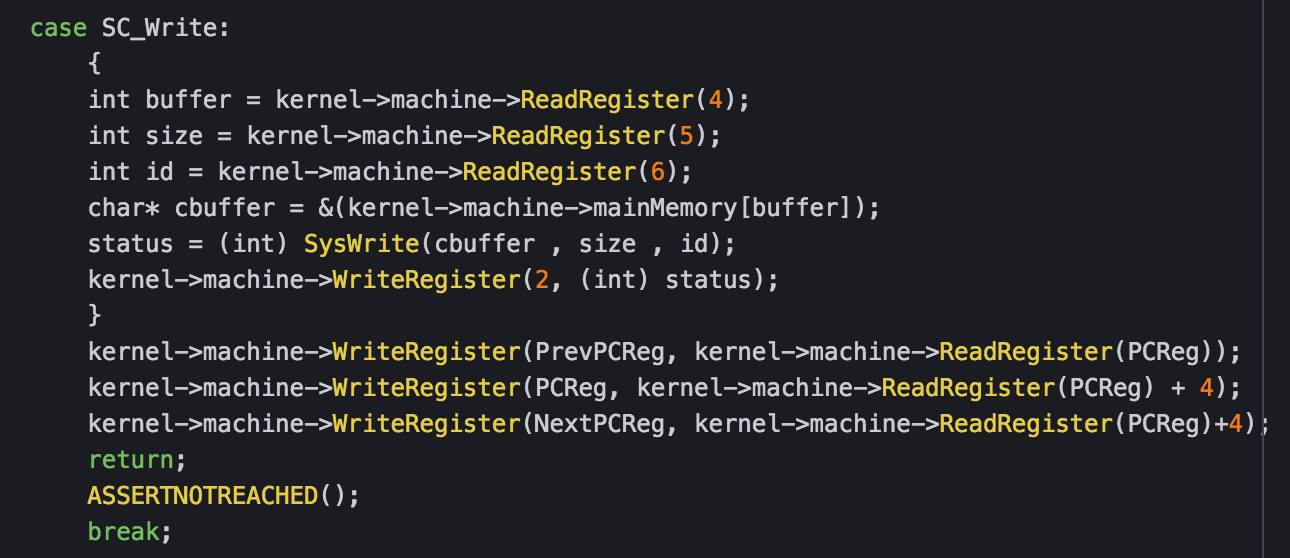




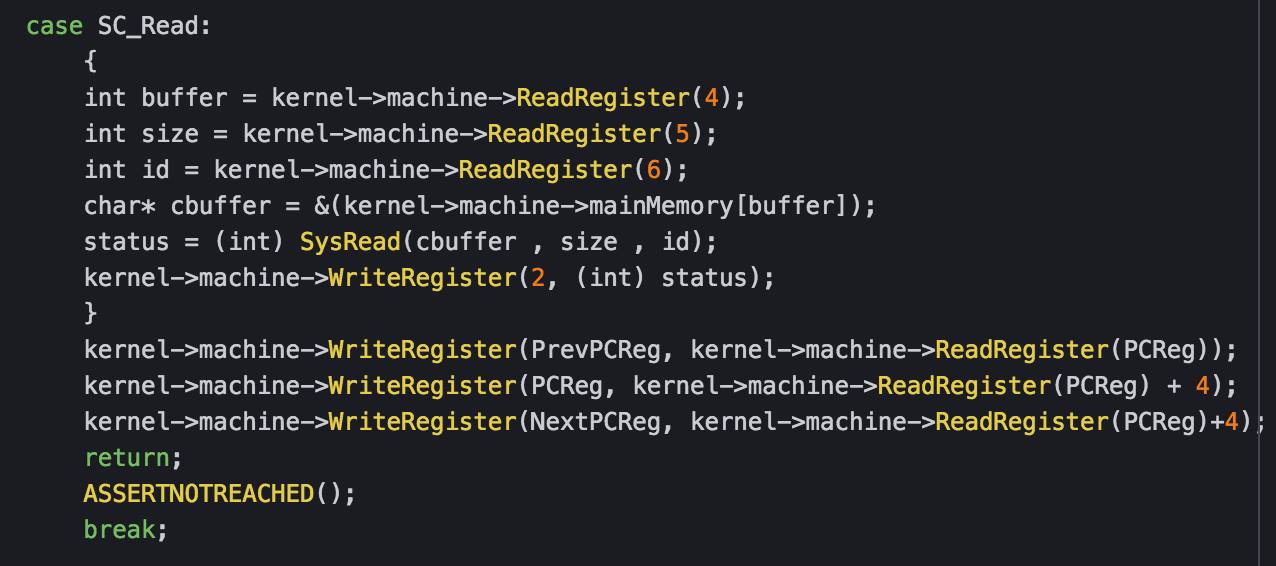
抓取到instruction的流程跟前面的system call一樣，不同點一樣是從exception.cc開始，match的case如下：



Open()和Create()這邊一樣，從$4讀指向檔名的指標，去nachOS的main memory取出該檔名的在底層linux系統中的位址，呼叫SysOpen()並傳入此位址，當return回來以後，將OpenFileID寫入$2，並maintain PC的資料。



Write()多了兩個參數，一樣取得到寫入的字串在底層linux的位址，並從$5, $6取得字串長度及Openfile的id，呼叫SysWrite()並傳入這三筆資料，return回來後得到寫入該檔的byte數，將之寫入$2，最後maintain PC 資料。

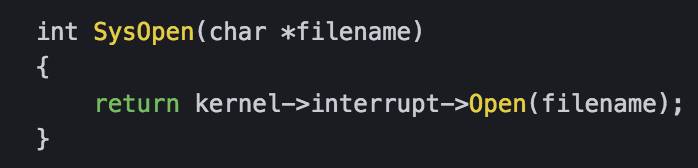


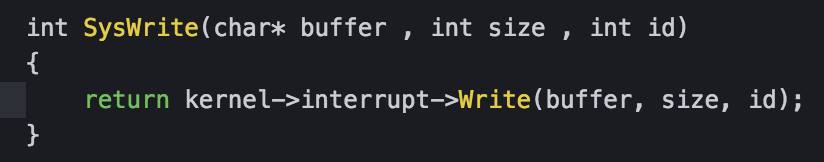
Read()和Write()一樣，不同的是從$4得到的指標是要將read到的字串放入的位址，呼叫SysRead()並傳述這三筆參數，return回來後同Write()。

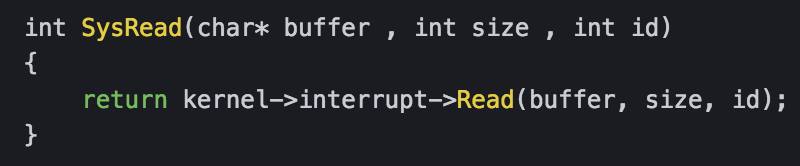


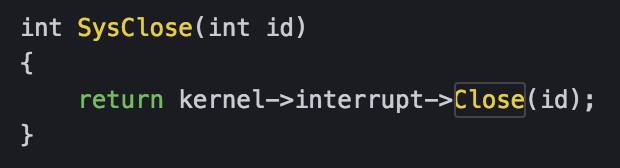
Close()稍微不同，從$4讀出要關閉檔案的OpenFileID，並呼叫SysClose()傳入id，return 回來後將是否關閉成功的狀態寫入$2，最後maintain PC的資料。

接下來看到ksyscall.h:

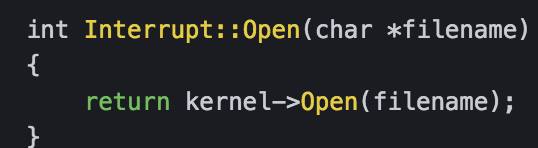


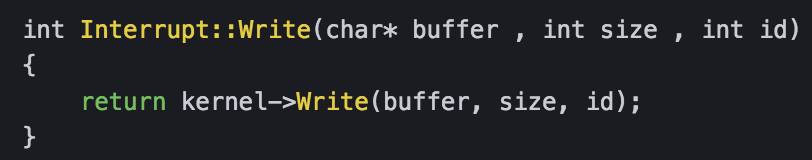


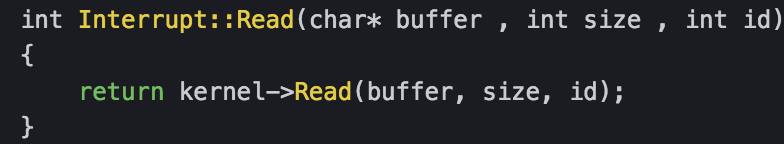




再來會跑到interrupt.cc：

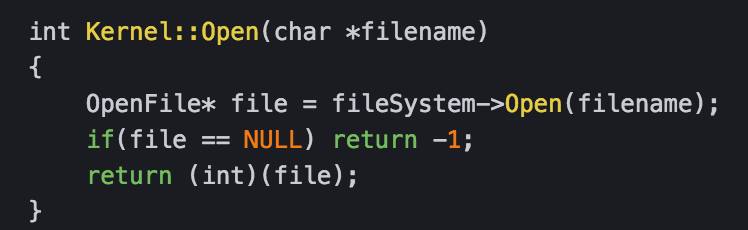




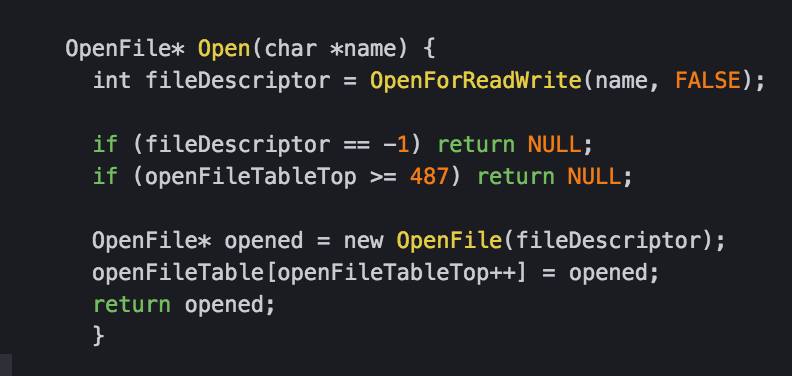


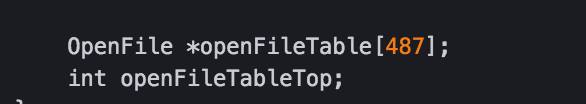


最後到kernel.cc，這邊比較複雜，四個function會分開說明，首先是Open()：



這邊直接使用fileSystem的Open()，他會回傳一個OpenFile物件的指標，我們將該指標儲存位址當作OpenFileID回傳，若fileSytem Open失敗則回傳-1，接下來看到filesys.h。





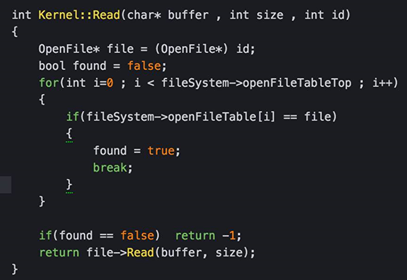
這邊OpenForReadWrite()會在sysdep.cc中呼叫底層linux的open()，得到的file descriptor會記錄在新建立的OpenFile物件中，最後回傳該物件指標，而這邊我使用一個openFileTable來記錄該物件指標，代表該檔案有成功開啟，方便之後的Read() , Write(), Close()進行操作，到這邊Open()就結束了。

接下來看到Write()，一樣從kernel.cc開始：



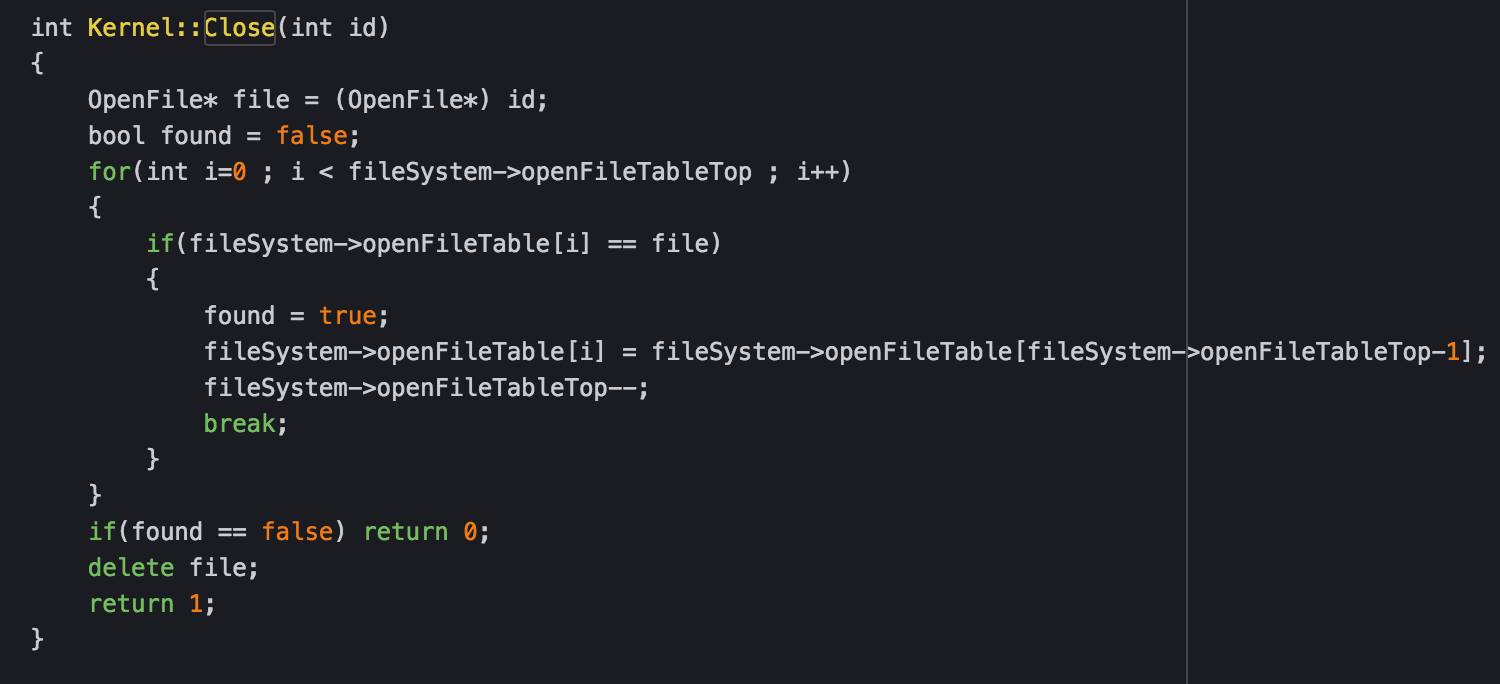
首先將傳入的id轉型成OpenFile的指標，並去openFileTable中看是否有記錄這個指標，若沒有則代表該檔案沒有成功開啟，回傳-1，若有找到則呼叫該OpenFile物件裡的Write()並傳入字串及長度，openfile.h中定義的這個函式會一路call到底層linux的write()並回傳寫入了多少bytes，然後一路回傳到exception.cc。

下一個是Read()，回到kernel.cc：



跟Write()一樣先將id的類型轉成OpenFile指標，然後去看看是否有成功開啟，沒有的話回傳-1，有的話呼叫該物件的Read()並傳入要read到的字串及大小，這函式一樣在openfile.h中定義，會一路call到底層linux的read()並回傳寫入了多少bytes，並一路回傳到exception.cc。

最後一個Close()，回到kernel.cc：



首先將傳入的id轉型成OpenFile指標，一樣去table檢查是否有成功開啟了，沒有的話就回傳0代表關閉失敗，如果有的話就將該OpenFile物件delete掉，並將這個指標從table中刪除，代表已經不是open的狀態，並回傳1回到exception.cc，值得注意的是delete OpenFile物件時，會啟動openfile.h中的解構式，這邊會一路call到底層linux的close()。