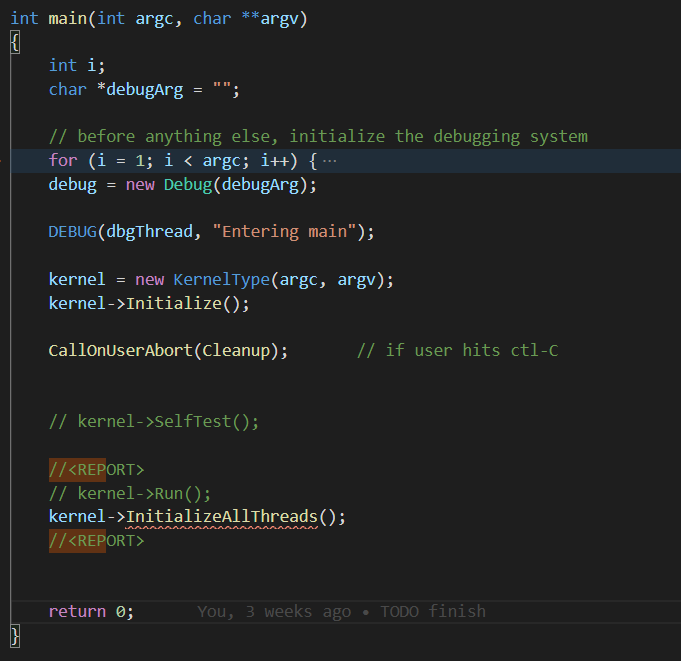
Operating System HW－CPU Scheduling

Group 18

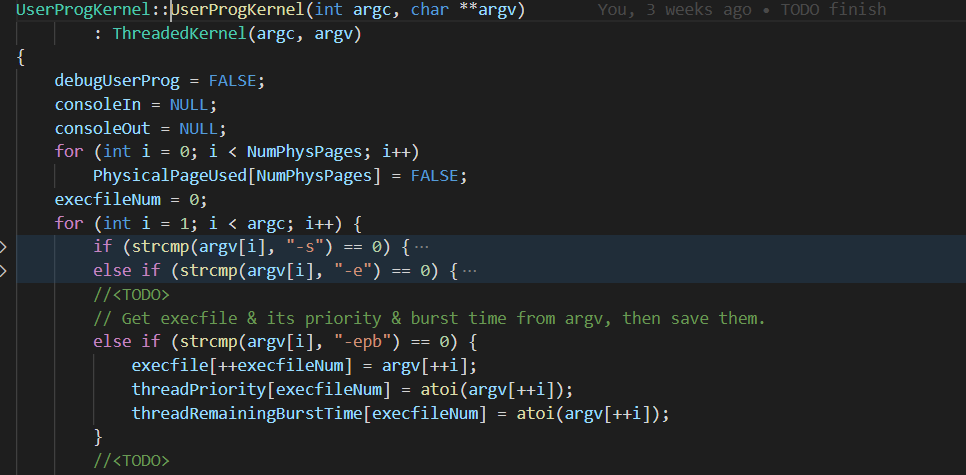
106061132 黃友廷 106061146 陳兆廷

1. Code Tracing and Implementation
   1. New→Ready

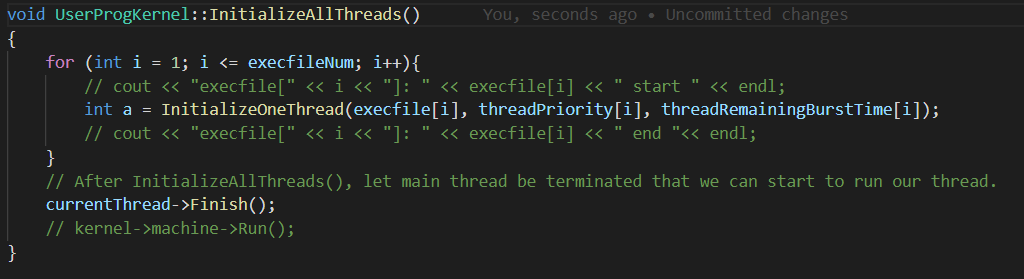
這個步驟在模擬有個新的Thread要進入Ready的階段。因此要創一個新的thread並將這個thread放到ready queue等待scheduler去選擇。



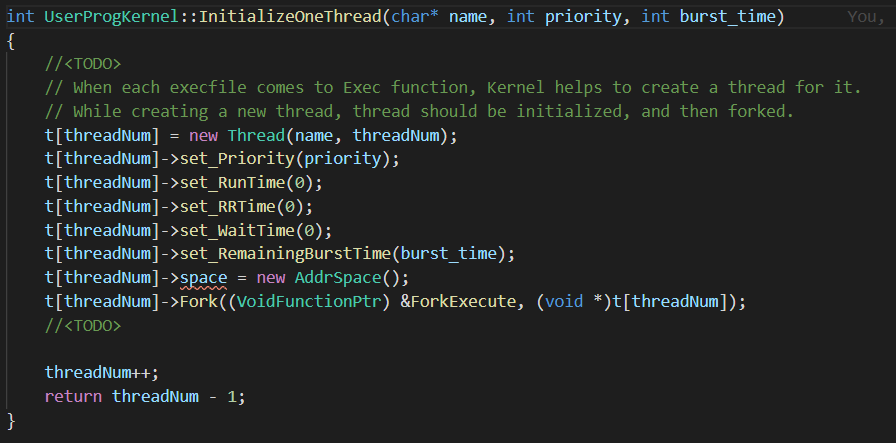
這份作業的main()是在threads/main.cc裡面，而這個main()在第一個for loop會把debug argument記錄到debugArg這個list裡面，然後就會使用KernelType()這個建構子new一個kernel出來，最後面在使用kernel->InitializeAllThreads()對各個thread進行初始化。



在這次作業我們所使用的KernelType是UserProgKernel，而這個建構子會把argv[]的輸入存到thread的對應資訊裡面，exefile[]是負責存各個thread所需要執行的file，threadPriority[]則是存thread的初始優先度，threadRemainingBurstTime[]則是存一開始預測所需的CPU burst time，execfileNum是存總共有幾個threadfile需要被執行。

* UserProgKernel::InitializeAllThreads()

第一個for loop會去使用UserProgKernel::InitializeOneThread()創造各一個thread給每一個要被執行的file。在這次作業中就是hw2\_test1.c, hw2\_test2.c。currentThread->Finish()會將現在正在執行的main thread停止並且刪除，去切換成在Ready queue中應該被執行的thread。

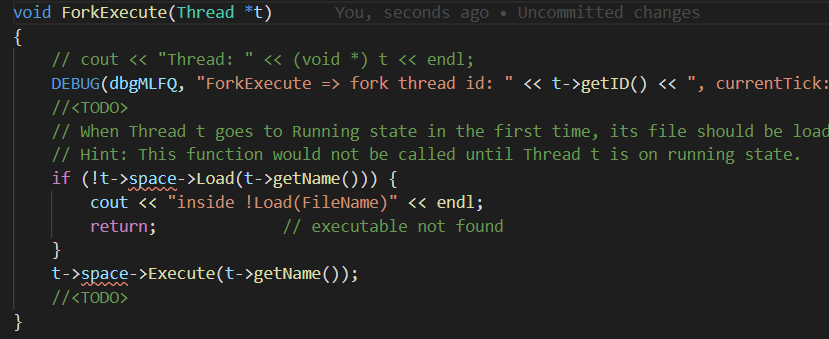
* + UserProgKernel:: InitializeOneThread()

為了實作出一個Multilevel Feedback Queue Scheduler並有不同的scheduling algorithm，一個thread會有ID、Priority、RunTime、RRTime、WaitTime、RemainBurstTime等資訊，並在此函式中定義。

|  |  |
| --- | --- |
| ID | 根據thread被加入的先後順序去設定ID |
| Priority | 決定thread要進入哪一個level的ready queue |
| RunTime | thread在CPU執行了多少時間，初始化為0 |
| RRTime | thread在Run Robin下在CPU執行了多少時間，初始化為0 |
| WaitTime | thread在其ready queue中等待了多少時間，初始化為0 |
| RemainBurstTime | thread距離預測的CPU burst time還剩下多少時間，初始化為預測值 |

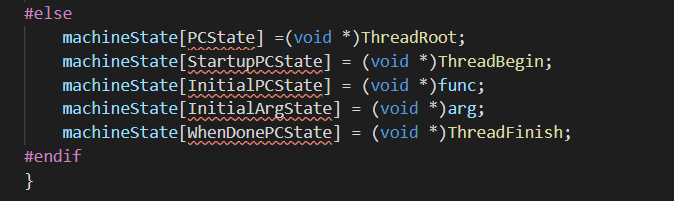
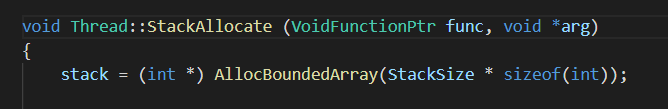
Thread::Fork()會去進行Fork產生一個子程序，這個子程序會去呼叫執行ForkExecute這個function，t[threadNum]則是ForkExecute會用到的argument。

* + - Thread::Fork()

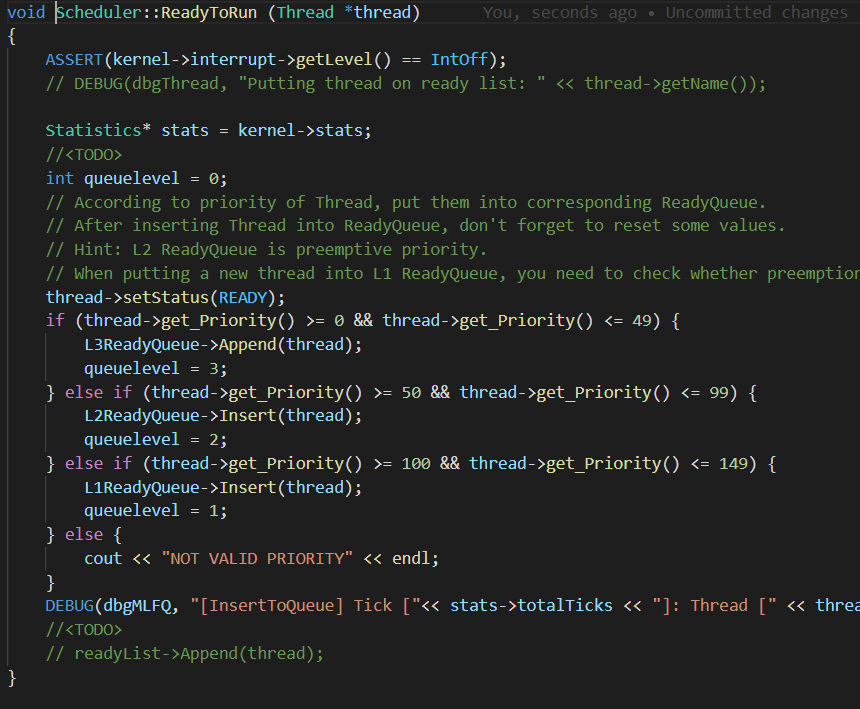
呼叫StackAllocate()去allocate一個execute stack去儲存func和arg，然後使用SetLevel()先將interrupt設為disable的狀態，接著呼叫ReadyToRun把thread放到適當的ready queue，最後再將interrupt改回原本的狀態。

ForkExecute就是先使用space->Load()把對應檔案打開並且讀入，再使用space->Execute()去執行這個檔案。

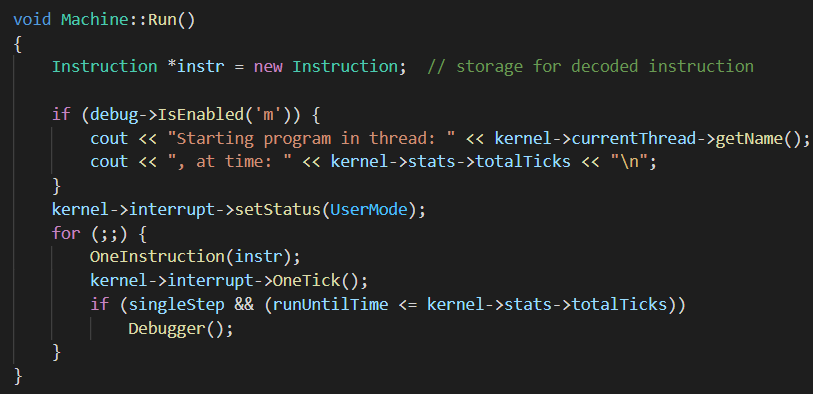
* + - * Thread::StackAllocate()



StackAllocate主要是先用AllocBoundedArray去allocate一個bounded array給thread的stack使用，然後再將各個registers (machineState[])去賦予所需要的數值。

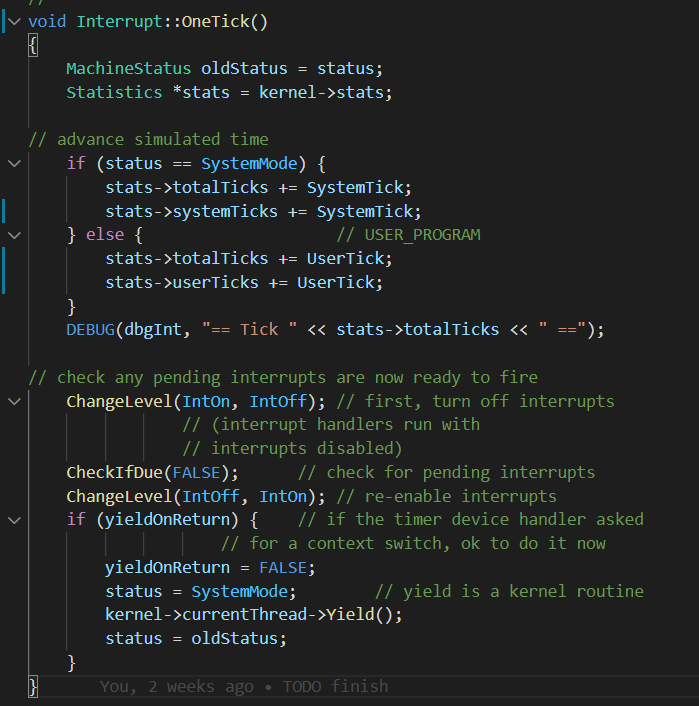
* + - * Scheduler::ReadyToRun()

ReadyToRun就是將傳進來的thread的Status設定為READY，並且依據他的priority去放到對應的ready queue裡等待scheduler去選擇並執行。

* 1. Running→Ready
* Machine::Run()

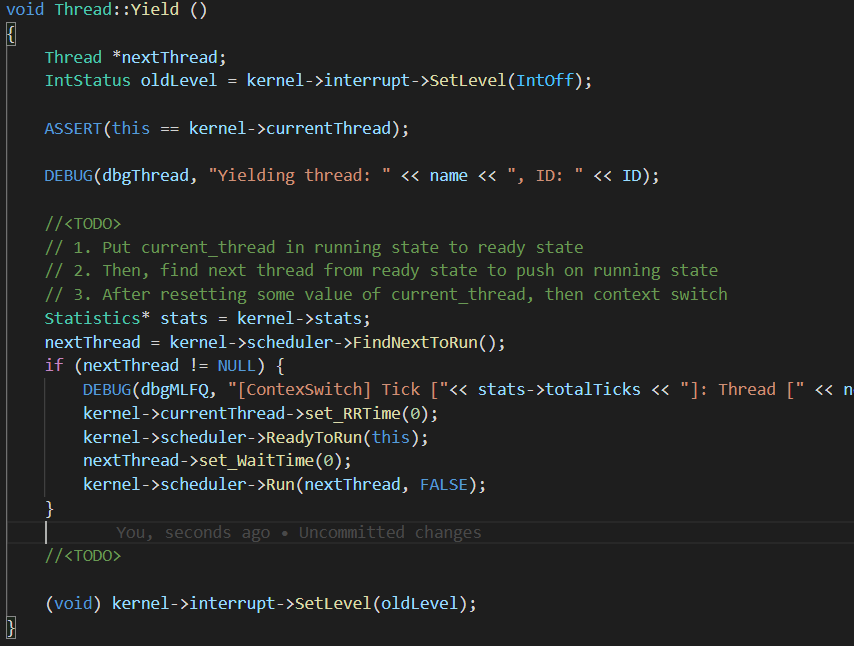
功能是去模擬執行user的program在NachOS上，setStatus(UserMode)會先將OS的設定成user mode，接著進入for loop去呼叫OneInstruction()去從已經被compile成binary file的User program中讀取instruction並執行。Interrupt->OneTick()則是會去檢查是否有interrupt被呼叫。

* + Interrupt::OneTick()



第一個if else主要是去計算目前的tick數量，幫助我們去了解各個事件的發生時間。再來會先將interrupt的狀態設定為disable，接著CheckIfDue()檢查是否有interrupt需要被執行，最後再將interrupt的狀態改回enable。最後面的if則是去判斷目前這個thread需不需要被換成其他在ready queue等待的thread，如果需要的話，就會使用currentThread->Yield()去進行thread的交換。而這裡的yeildOneReturn是由Alarm::CallBack()去決定的，如果thread需要進行交換就會把yeildOneReturn設為true，否則yeildOneReturn就是false。

* + - Thread::Yield()



先用Scheduler::FindNextToRun()去從Ready queue找出下一個要被執行的thread，如果下一個thread存在就會把目前thread的RRTime歸0，接著使用Scheduler::ReadyToRun()去把目前的thread放進適當的ready queue裡。然後把下一個要被執行的thread的WaitTime歸0，並且使用scheduler->Run()進行context switch，把CPU中執行的thread換成下一個thread。

* + - * Scheduler::FindNextToRun()

Scheduler::FindNextToRun()的實作就是依照L1、L2、L3的順序，取出queue裡最高順位的Thread執行回傳。

* + - * Scheduler::ReadyToRun()

已於(1)解釋。

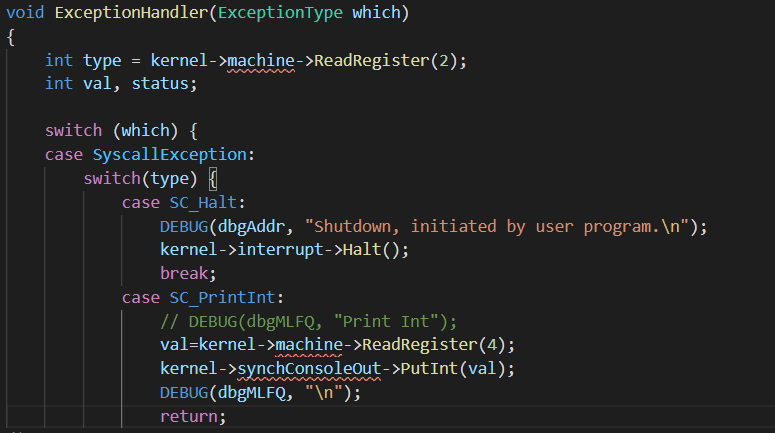
* + - * Scheduler::Run()

將於(6)連同Switch解釋。

* 1. Running→Waiting

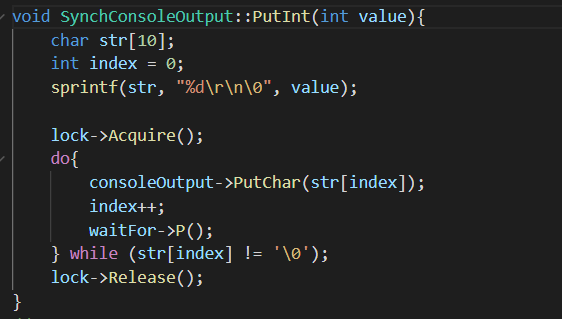
當目前的thread需要I/O時就會進入waiting state，等待I/O完成後才會回到ready queue等待scheduler去選擇。而這次作業的testfile裡面都有用到PrintInt()這個function去印數字在terminal上，因此會需要用到I/O。而PrintInt()實際上是用system call的方式去實做，所以所觸發的途徑會與HW1類似。

* ExceptionHandler()



因為argument是從register 4開始往後存，所以第一個argument就是存在register 4，因此會先使用ReadRegister(4)去把register的數值讀出來。接著使用PutInt()這個function去完成這個system call。

* + SynchConsoleOutput::PutInt()

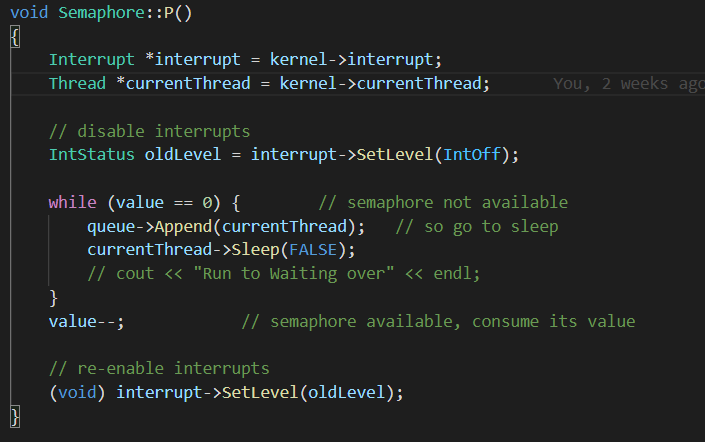


PutInt()一開始會先用sprintf將要印出的數字轉換成完整要在terminal上顯示的string，然後使用lock->Acquire和lock->Release() (這兩個function是Atomical)去確保同時只會有一個thread在terminal上印訊息。

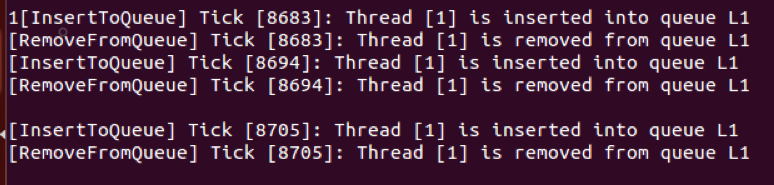
* + - SynchConsoleOutput::PutChar()

PutInt()使用while loop則是使用PutChar()去把char一個一個印出來，waitFor->P()則是為了確保輸出在terminal時I/O有完整執行完才會繼續印下一個char。

* + - * Semaphore::P()



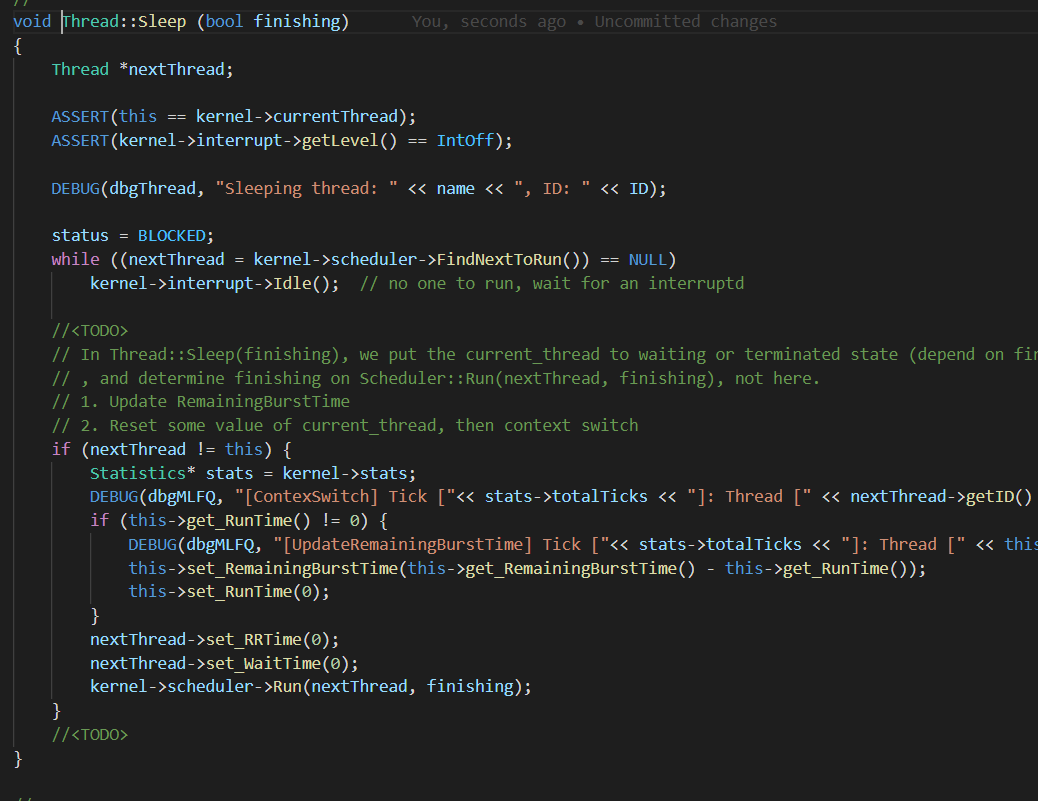
當value = 0時就是代表現在semaphore是not availabe，所以會將currentThread放到I/O waiting queue裡面等待，並且使用thread->sleep()去找出下一個要執行的thread，進行交換。而當value>0時就會減掉一個value，繼續進行。



這邊可以注意到印一次數字會需要印三個char(%d, \r, \n)，這也是為甚麼印一次完整的數字會讓thread進入Ready queue三次。

* + - * + SynchList<T>::Append()

在這裡將currentThread用Append()加入ready queue代表即將被wait的Thread要被加入回到ready queue了。

* + - * + Thread::Sleep()

整體與thread->Yeild()很像，依樣需要先用FindNextToRun()去找出下一個要被執行的thread，並且去更新RemainingBurstTime、RunTime、RRTime、WaitTIme，最後在使用scheduler->Run()去進行context switch以及執行nextThread。唯一不同的是不需要將currentThread放到ready queue中，因為我們在Semaphore->P()時就已經將他放在I/O waiting queue裡了。

Scheduler::ReadyToRun()

已於(1)解釋。

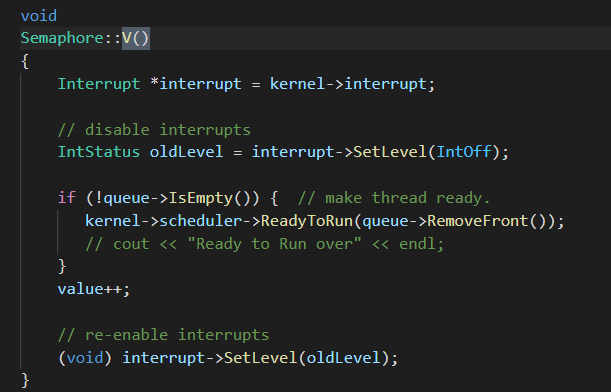
Scheduler::Run()

將於(6)連同Switch解釋。

* 1. Waiting→Ready

這個步驟在模擬Thread從waiting進入ready queue的過程。

* Semaphore::V()



Semaphore::V()與P()做相反的事情。當ready queue還不是空的，會利用ReadyToRun呼叫ready queue的下一位執行，若是空的就將value加上一單位。相同於P()，也會將Interrupt關閉再開啟。

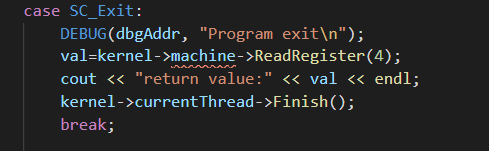
* + Scheduler::ReadyToRun()

已於(1)解釋。

* 1. Running→Terminated

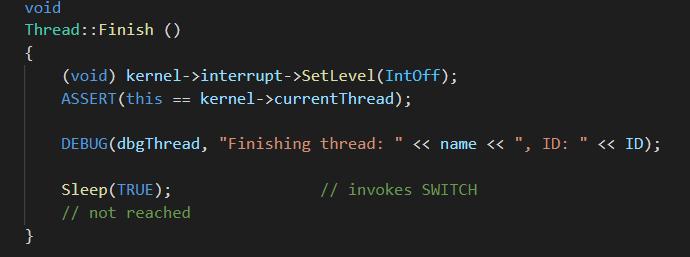
這個步驟在模擬一個外來指令終止了一個Thread，將它從Running state直接Terminated。

* ExceptionHandler()



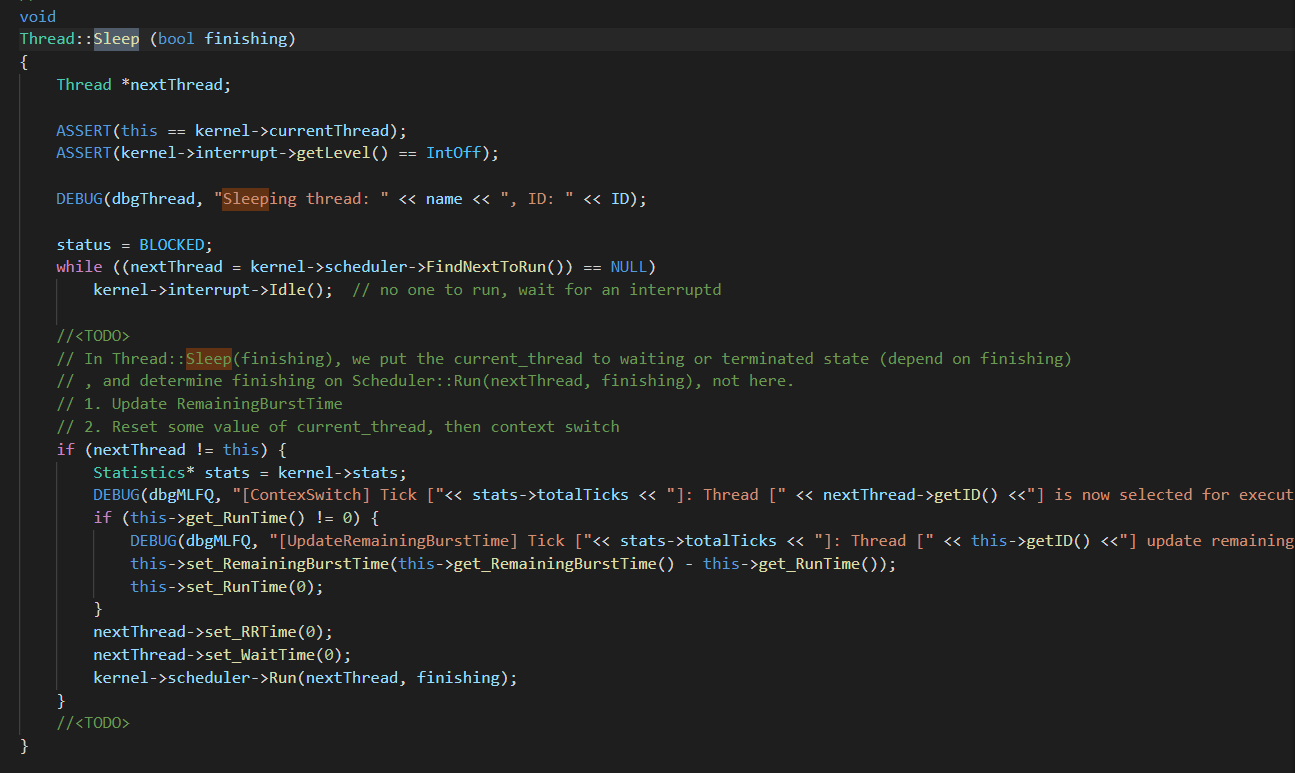
啟動這個終止程序的是一個System Call。藉由SC\_Exit，將Thread直接變為Finish()的狀態。

* + Thread::Finish()



Thread::Finish()會將這個Thread進入Sleep。

* + - Thread::Sleep()



首先要先把Interrupt關掉，接著利用前面提到的FindNextToRun()去儲存下一個要執行的Thread。如果下個Thread是其他的Thread，就要有一系列的時間交接跟設定。先將這個Thread總結一下，把剩餘的時間計算好，然後歸零；接著初始下個Thread的RoundRobin時間值跟等待的時間，最後執行下個Thread。

* + - * Scheduler::FindNextToRun()

已在(2)解釋。

* + - * Scheduler::Run()

在(6)隨著SWITCH解釋。

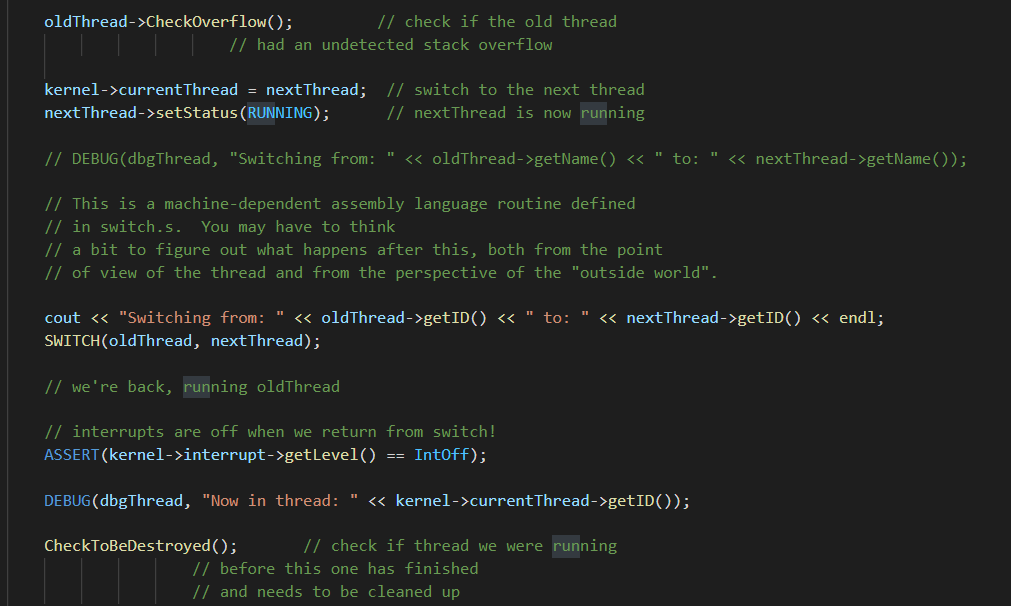
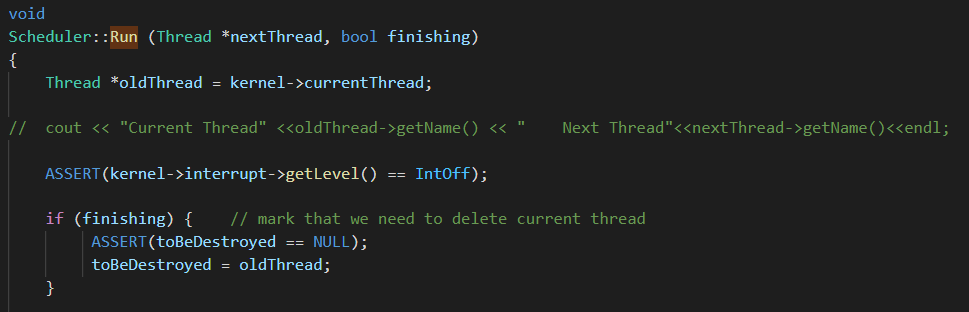
* 1. Ready→Running

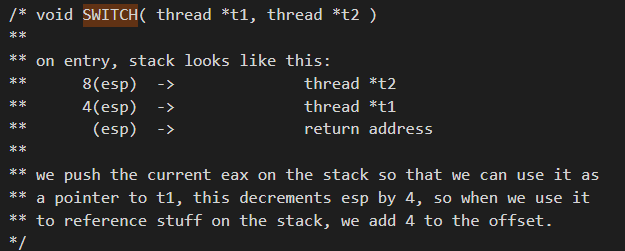
這個步驟在模擬Thread在Ready queue中被選出後執行。首先如同上個步驟中的Finish()/Sleep，當這個Thread因為事情而結束，輪到下一個Thread執行時，會經過Scheduler::FindNextToRun()挑選出下一個要執行的Thread，挑選完後用Scheduler::Run()執行。

* Scheduler::FindNextToRun()

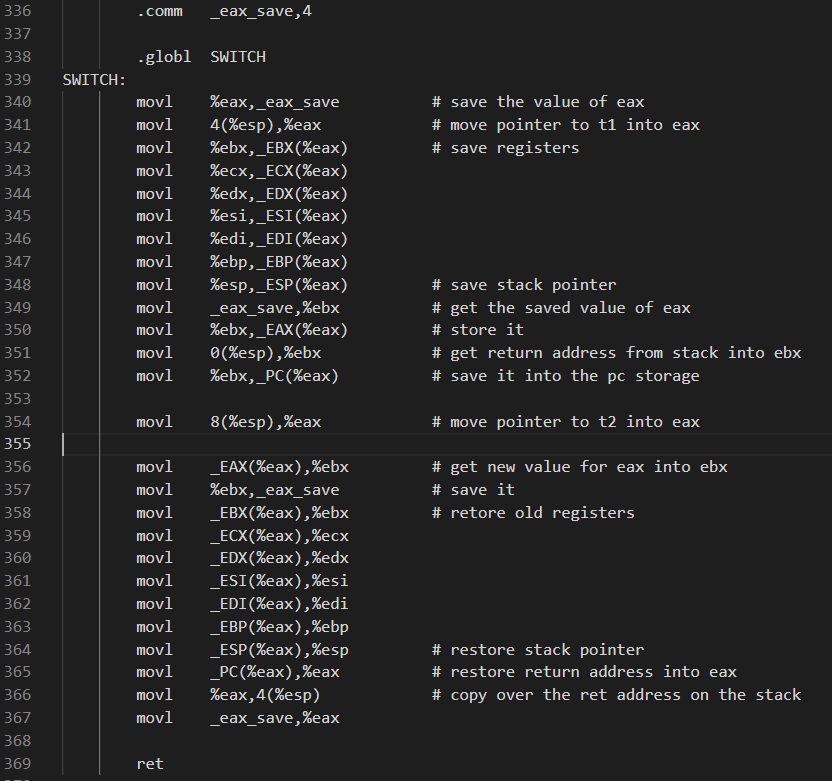
已於(2)解釋。

* + Scheduler::Run()



假設這個Thread已經該結束了，就把它刪除掉(toBeDestroyed)。接著將新的Thread變為現在的Thread(kernel->currentThread)並更新狀態成Running。進入SWITCH以組合語言執行Switch的工作。

在SWITCH一開始，t2是存在8(esp)，t1存在4(esp)，而(esp)為return的位址。ESP為Stack Pointer。



340、341行在把原本放在reg eax的值取出存在\_eax\_save後，把t1放入eax。

342～348、350是將所有reg依照eax的位置存好，也就是t1的位置。

349將原本eax的值，也就是\_eax\_save存入ebx。

351、352把return address存入ebx中，並存在PC storage裡。

354將t2放入eax。

356、357把新eax的值放入ebx，並存在\_eax\_save。

358～364把t2的register value存回原本的register。

365把原本存在PC的return address放回eax。

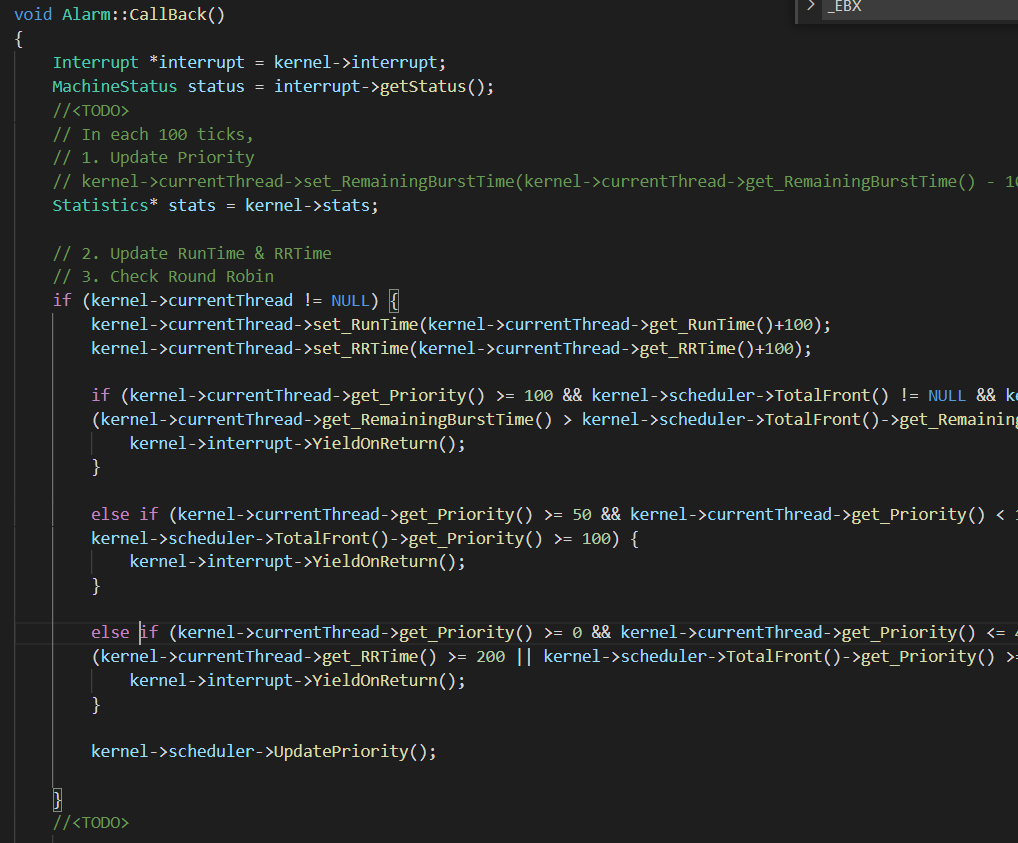
366把eax指向4(%esp)，也就是把原本return address存在stack的位置(%esp)往上移了4 bytes。

367把原本eax的值放回eax。

* 1. Update priority

在每個Thread執行後，需要有個計時器將正在執行的Thread的priority、timer等做更新，也就是整個scheduler最重要的一個部分，若沒有這個更新，每一步都都會是一模一樣的。

* + Alarm::CallBack()



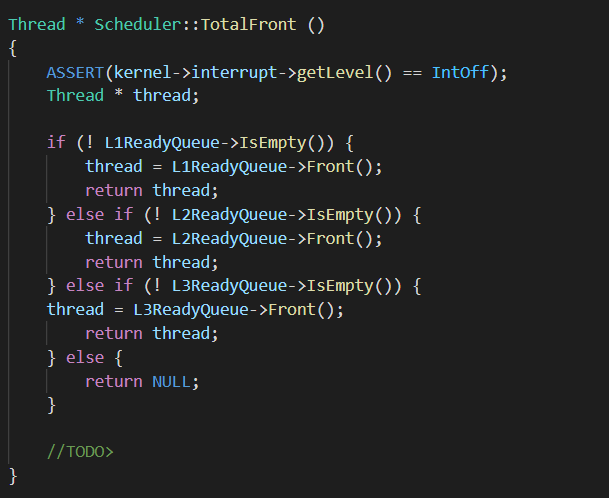
CallBack()是每100 ticks就會執行一次，所以當程式每次進入這個function時都要把currentThread的RunTime和RRTime都加100，以此來計算currentThread在CPU執行了多久。接著就是去檢查各個ready queue有沒有存在需要與currentThread進行context switch的thread，需要進行context switch的條件有以下這幾個。

* + 在較高priority的ready queue中存在thread (Level 1 > Level 2 > Level 3)
  + 在Level 1 ready queue時則是比較thread的RemainingBurstTime，越短的thread有越高的順位被CPU執行
  + 在Level 3 ready queue時則是採取RoundRobin，所以只要Level 3的thread執行滿200 ticks就要進行context switch，換其他在Level 3 ready queue等待的thread執行。

這裡implement的方式是只要符合特定的條件，就會去呼叫interrupt->YeildOneReturn()，去讓yeildOneReturn這個flag改為true，讓currentThread可以由Running state變為Ready state。

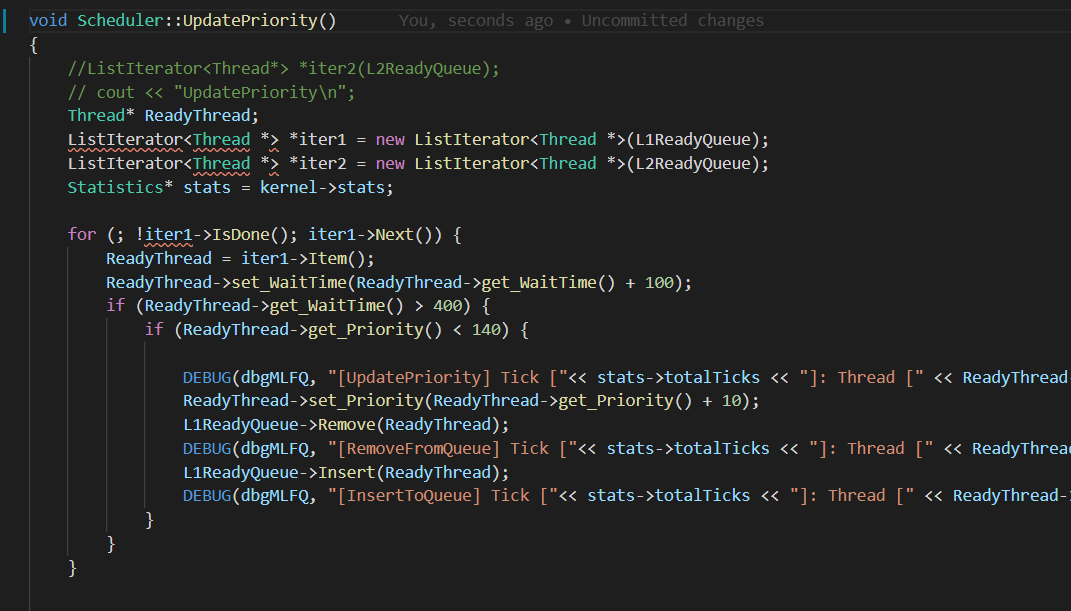
最後面呼叫UpdatePriority() 對各個level的ready queue裡thread的priority進行aging。

* + Scheduler::TotalFront ()



TotalFront()就會從ready queue裡面選出優先度最高的thread進行回傳，優先度排列是L1 > L2 > L3，所以較高優先度的ready queue有thread時，就會優先回傳。

* + Scheduler::UpdatePriority()



這個部分就是依序去檢查各個level的ready queue裡thread的WaitTime有沒有超過400 ticks，如果有就需要把priority提高10，並且重新sorting讓ready queue裡面是依照特定順序排列。做法就是使用SortedList的Insert去達成，因為我們有去定義不同level ready queue所需要用到的compare，而insert就會使用這個compare去讓ready queue能夠按照特定順序去排列。

1. instruction execution
   1. 這個作業是將一個要被執行的檔案(ex: hw2\_test1)當作一個process去讓NachOS執行，而每個process並不會再分成多個thread，因此可以把一個要被執行的檔案直接當作一個thread。

userprog/nachos -epb <execute file> <p1> <bt1> -epb <execute file> <p2> <bt2> -d z

execute file : 需要NachOS去執行的file

p1, p2 : 對應thread的初始priority

bt1, bt2 : 對應thread的預測CPU burst time

-d z : 選擇所需要顯示的DEBUG message

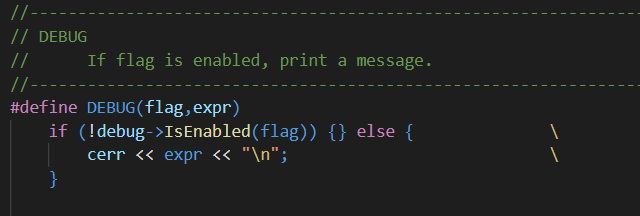
ex:

userprog/nachos -epb test/hw2\_test1 40 5000 -epb test/hw2\_test2 80 4000 -d z

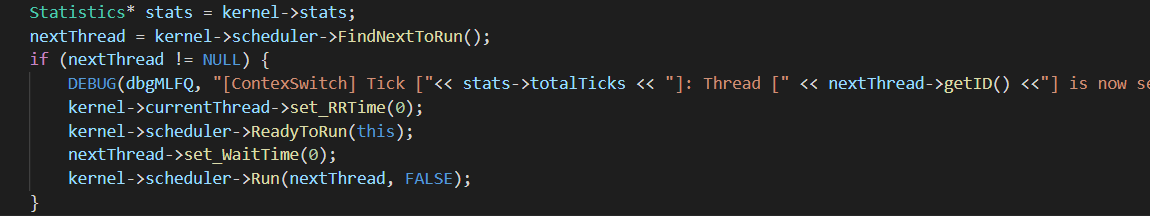
* 1. debug.h:



定義debug flag，讓我們能夠去選擇在執行NachOS時要顯示那些debug message。



定義DEBUG這個macro function，我們會使用這個macro function在NachOS code裡面適當位置印出助教所要求的message。這個作業所使用的debug flag是dbgMLFQ，所以使用的方式就會是下圖這樣子。



1. 小組分工

106061132 黃友廷：Coding、Report

106061146 陳兆廷：Report

106000147 沈永聖（已退選）：Coding、Report