**CS342301: Operating System**

**MP3: CPU scheduling**

組別: 8

組員:

109062315洪聖祥

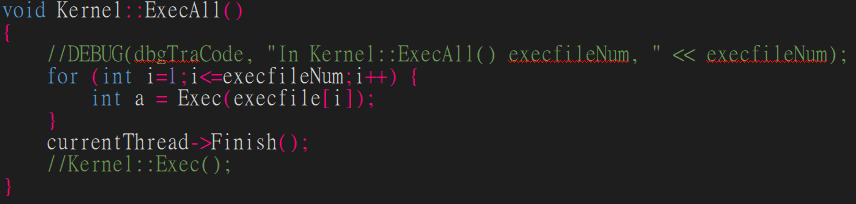
109062314張紘齊

|  |  |
| --- | --- |
| 項目 | 分工 |
| Report |  |
| Switch.s | 洪聖祥 |
| 剩餘Trace code | 張紘齊 |
| Implementation |  |
| Add priority to thread | 洪聖祥 |
| 3 level feedback queue | 洪聖祥 |
| Add command line argument -ep | 洪聖祥 |
| Add a debug flag z and use DEGUG(‘z’ , expr) | 洪聖祥 |
| Aging mechanism | 張紘齊 |
| preemption | 張紘齊 |
| Debug | 洪聖祥、張紘齊 |

**Part I: Trace Code**

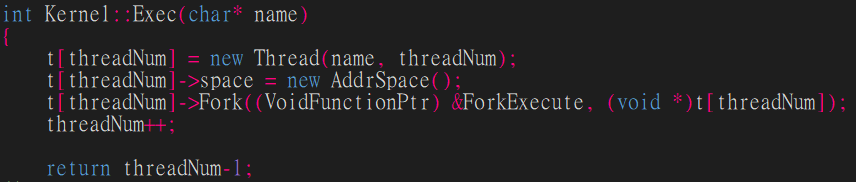
1-1. New -> Ready

Kernel:: ExecAll()



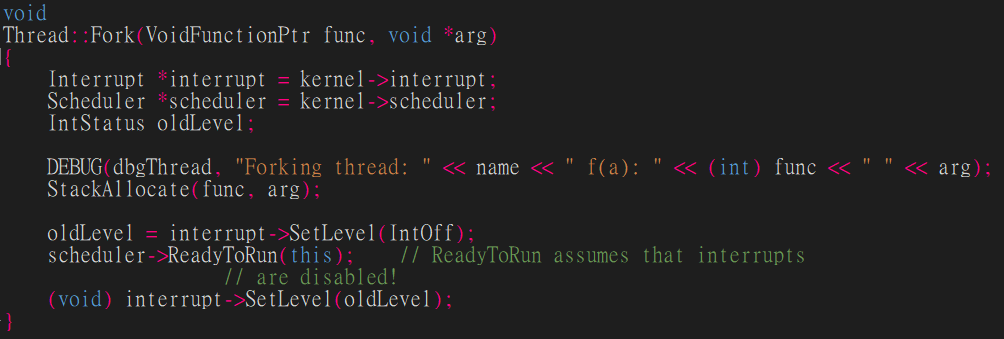
根據input instruction檔案的數量去跑for loop，執行Exec(file name)。最後再將main Thread執行Finish，讓它與其他檔案做context switch。

Kernel:: Exec(char\*)



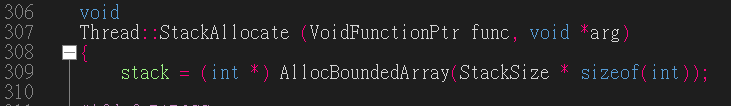
替該檔案新增一個Thread物件，並透過new AddrSpace給予空間，以及透過Fork對該Thread進行前處理

Thread:: Fork(VoidFunctionPtr, void\*)

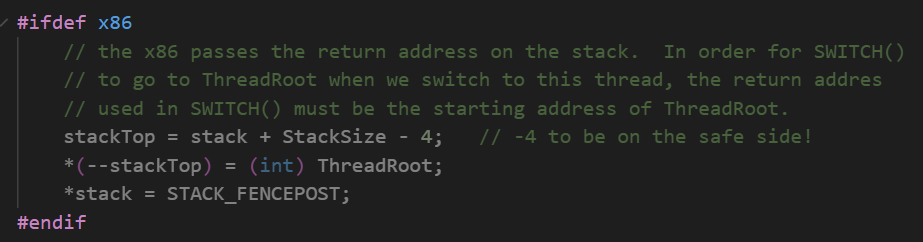


這裡會執行StackAllocate分配stack給該Thread，接著透過ReadyToRun將該Thread放入Ready Queue中

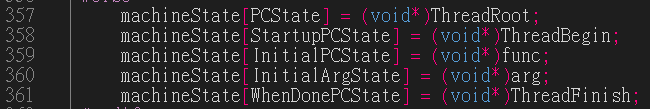
Thread:: StackAllocate(VoidFunctionPtr, void\*)



首先配置一塊連續的記憶體給stack，讓thread能存放temporary data

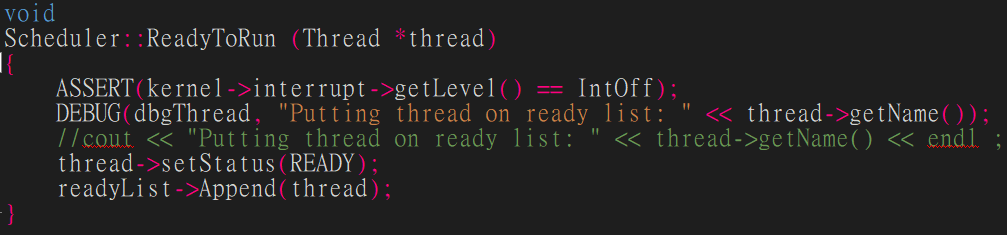


接著設定current stack pointer (stackTop) 的位置、放好ThreadRoot、將Bottom of the stack (\*stack) 設為STACK\_FENCEPOST來判斷stack overflows



最後設定registers的值，讓context switch發生時能夠切換到當前的thread

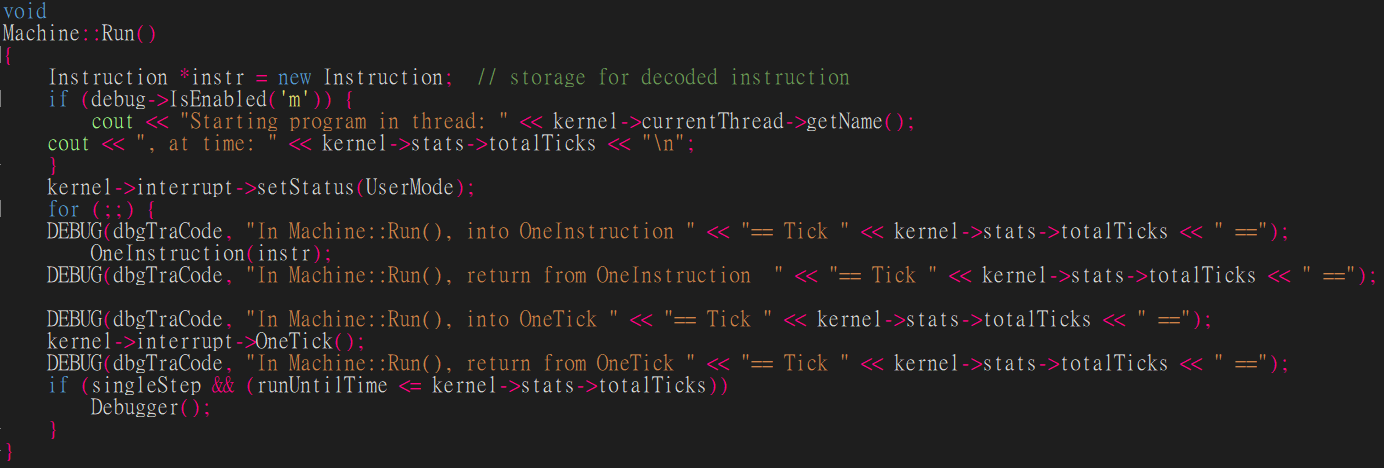
Scheduler:: ReadyToRun(Thread\*)



將該Thread放入Ready Queue中

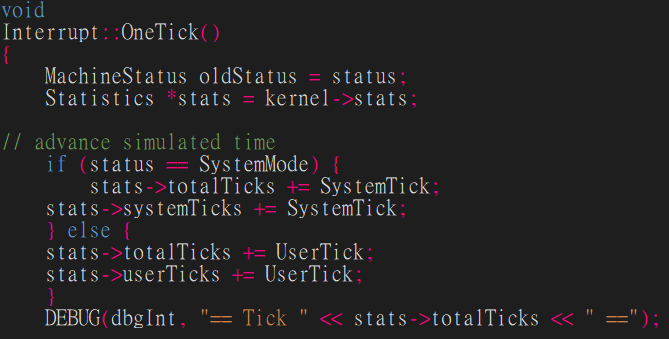
1-2. Running -> Ready

Machine:: Run()

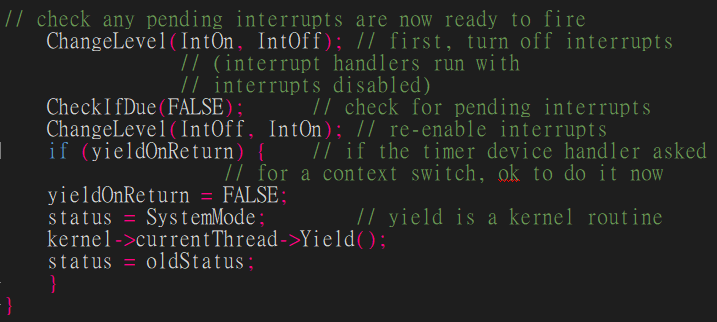


主要就是透過for loop，將每個指令透過OneInstruction解析和執行，並在結束該指令後進入OneTick

Interrupt:: OneTick()

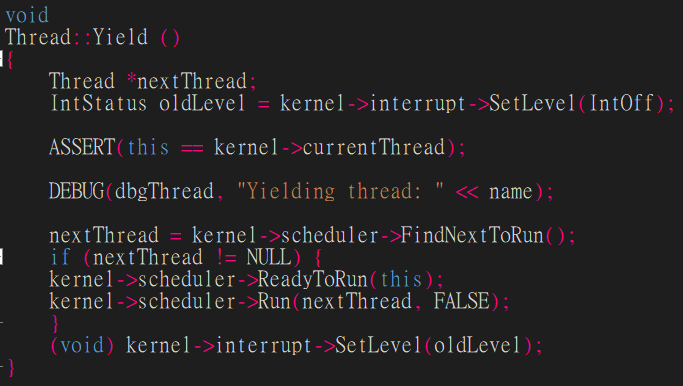


前半部分是在跟更新SystemTick和UserTick



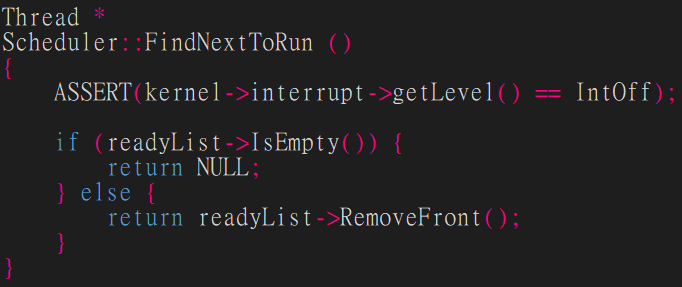
後半部分，透過CheckIfDue來判斷pending Interrupts是否要被執行，接著如果yieldOnReturn為true，也就是該執行time slice，就呼叫Yield將current Thread釋放CPU資源

Thread:: Yield()



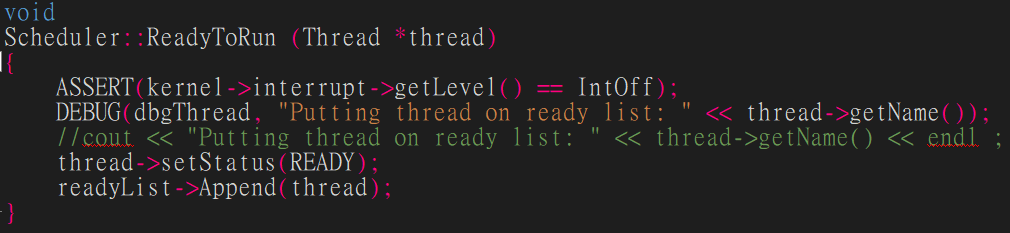
呼叫FindNextToRun取得Ready Queue中的下一個Thread，並透過ReadyToRun將自己放回Ready Queue，最後在透過Run進行context switch

Scheduler:: FindNextToRun()



判斷Ready Queue中是否還有其他Thread，沒有的話回傳Null，有的話回傳最前面的Thread

Scheduler:: ReadyToRun(Thread\*)



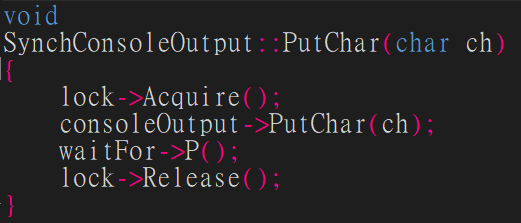
將該Thread放回Ready Queue中

Scheduler:: Run(Thread\*, bool)

主要就是進行context switch，並在執行前將oldThread資料保存，執行後將newThread資料讀取出來

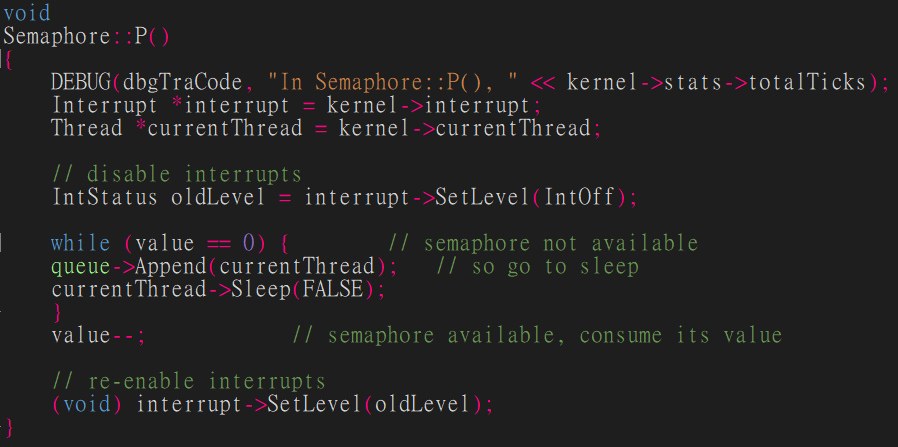
1-3. Running -> Waiting

SynchConsoleOutput:: PutChar(char)



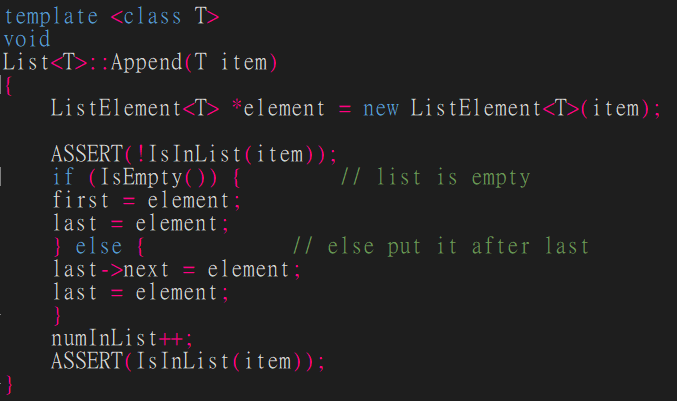
替該Thread取得lock，接著執行I/O，然後呼叫waitFor->P()等待該Thread執行完成，最後再釋放lock

Semaphore:: P()



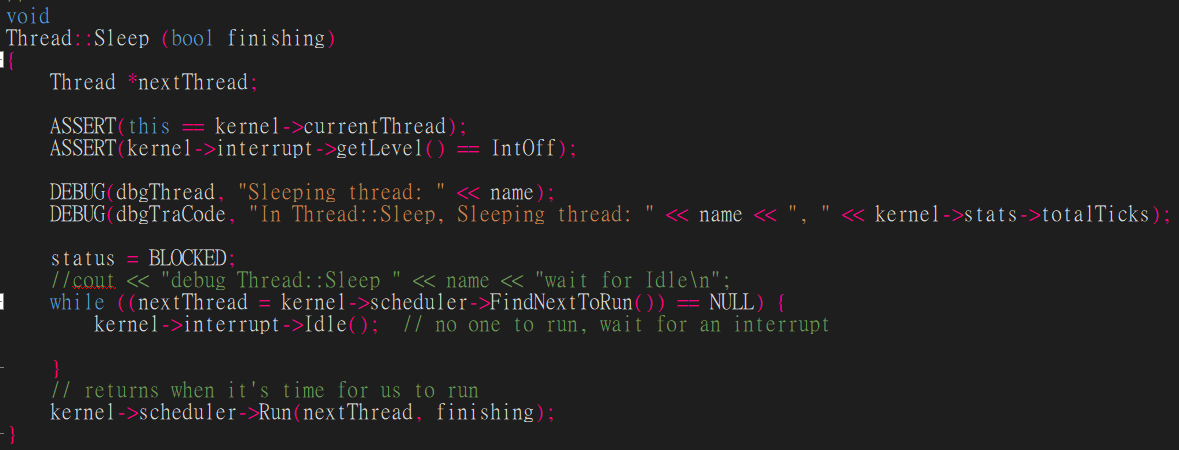
假如當前value為零，也就是I/O還沒做完，則將它放入Semaphore 的queue中等待，並使該Thread呼叫Sleep，讓他進入waiting state。當它做完時，則將value--

List<T>:: Append(T)



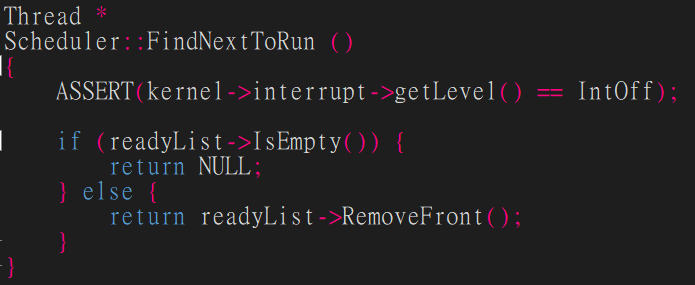
就是Linked list的Append實作

Thread:: Sleep(bool)



將該Thread status設為BLOCKED，並透過FindNextToRun找出Ready Queue中的下一個Thread，接著將該Thread與current Thread透過Run進行context switch

Scheduler:: FindNextToRun()



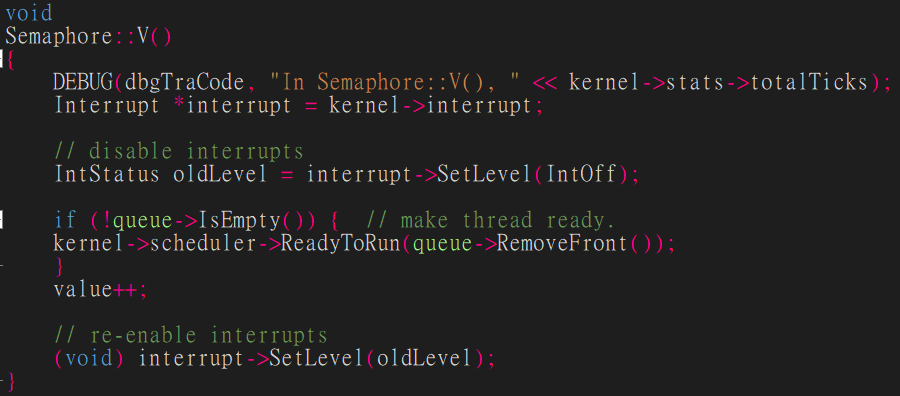
判斷Ready Queue中是否還有其他Thread，沒有的話回傳Null，有的話回傳最前面的Thread

Scheduler:: Run(Thread\*, bool)

主要就是進行context switch，並在執行前將oldThread資料保存，執行後將newThread資料讀取出來

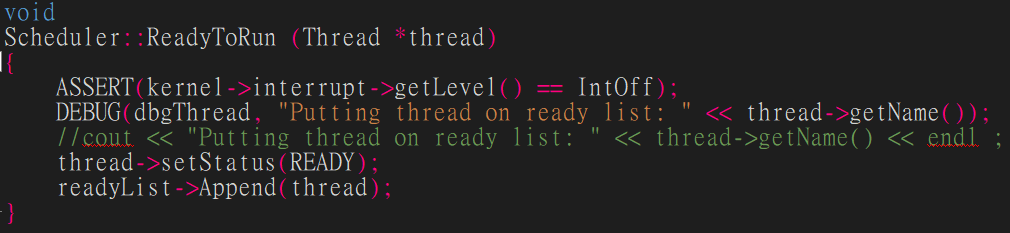
1-4. Waiting -> Ready

Semaphore:: V()



會呼叫到Semaphore V，代表說現在I/O執行完成，因此將先前在Semaphore P中被放入Semaphore queue的Thread，放回Ready List中，並將value++

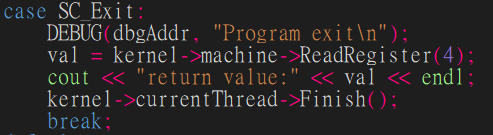
Scheduler:: ReadyToRun(Thread\*)



將thread的status切換成READY，然後把此thread放入ready queue中

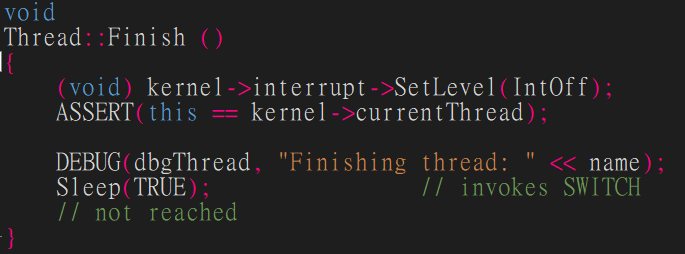
1-5. Running -> Terminated

ExeceptionHandler(ExeceptionType) case SC\_Exit



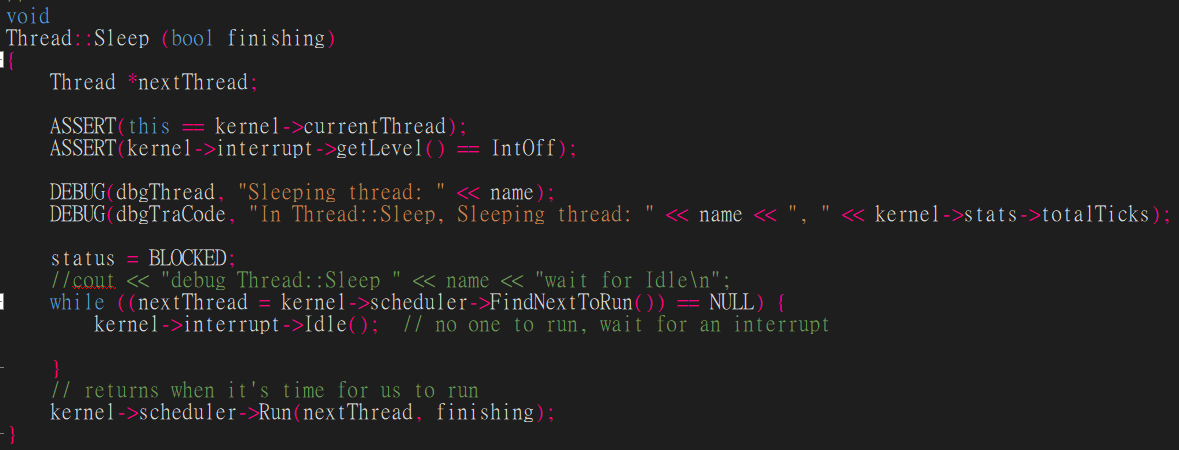
取得Register 4的值將之印出，接著讓current Thread執行 Finish

Thread:: Finish()



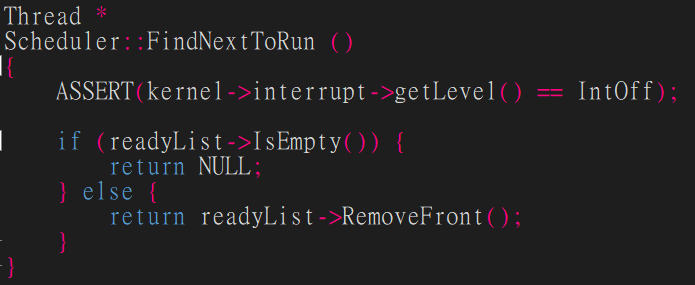
呼叫Sleep，並注意parameter的值為true，因為該Thread將不需再被執行

Thread:: Sleep(bool)



將該Thread status設為BLOCKED，並透過FindNextToRun找出Ready Queue中的下一個Thread，接著將該Thread與current Thread透過Run進行context switch

Scheduler:: FindNextToRun()

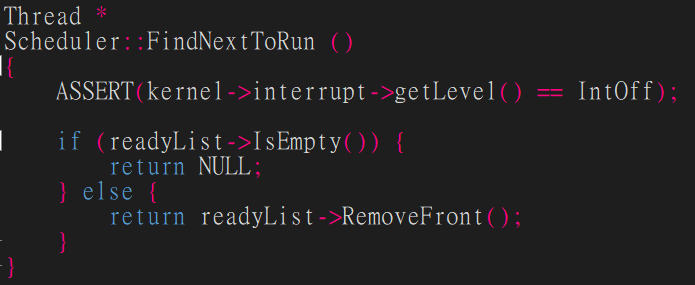


判斷Ready Queue中是否還有其他Thread，沒有的話回傳Null，有的話回傳最前面的Thread

Scheduler:: Run(Thread\*, bool)

主要就是進行context switch，並在執行前將oldThread資料保存，執行後將newThread資料讀取出來

1-6. Ready -> Running



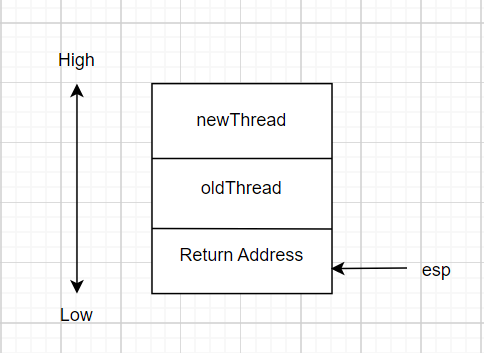
判斷Ready Queue中是否還有其他Thread，沒有的話回傳Null，有的話回傳最前面的Thread

Scheduler:: Run(Thread\*, bool)

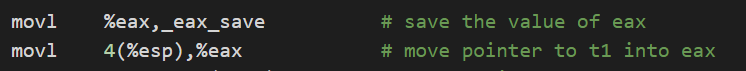
主要就是進行context switch，並在執行前將oldThread資料保存，執行後將newThread資料讀取出來

SWITCH(Thread\*, Thread\*)

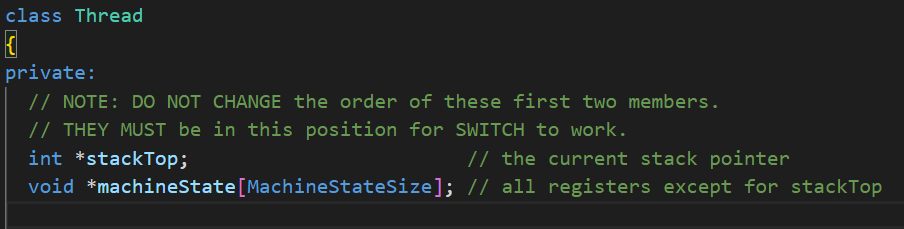
首先呼叫SWITCH(oldThread, newThread) 記憶體的分布如下圖:



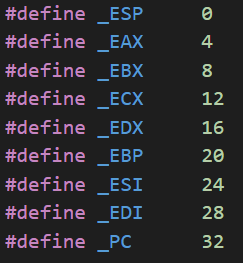
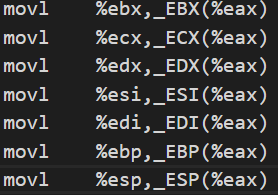
先將 eax的值存到eax\_save，並將esp+4的值存到eax。



從記憶體的分布圖來看，現在eax的值存的是oldThread的記憶體位置。

我們可以從下圖class Thread中發現oldThread的記憶體位置也是stackTop的記憶體位置。

從switch.s 和 switch.h中對應，其實就是將general purpose register以及stack pointer和base pointer存到oldThread的stackTop 以及machineState中。至於為什麼都是4的倍數，這是因為x86 是32 bit，也就是4byte，記憶體也是byte-address，所以移到下一個記憶體位置要+4。

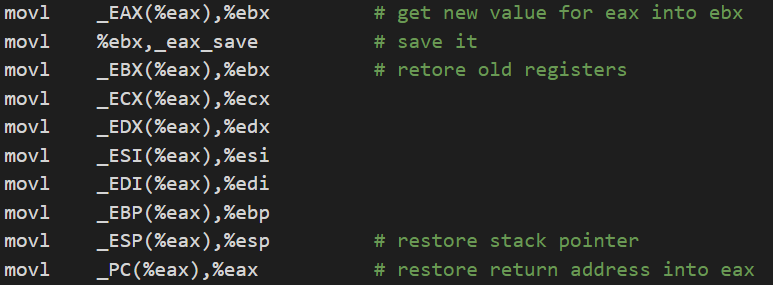


以 movl %ebx,\_EBX(%eax)為例，將ebx的值存到eax指到的記憶體位置+\_EBP/4中。而現在eax指到的記憶體位置是oldThread的記憶體位置(stackTop)。eax+4從Thread class 發現是machineState array，所以存到machineState[0]。其他register也是類似方法存到machineState array的其他entry。值得注意的是，\_ESP 是0 ，所以esp (stack pointer)的值是存到stackTop而不是machineState array。

接下來直覺想，因為要做context switch既然register的值都被存起來，下一步就是把新的thread中被存起來的值重新放到register中。

所以先把esp指到的記憶體位置+8(下圖)，也就是newThread的記憶體位置。

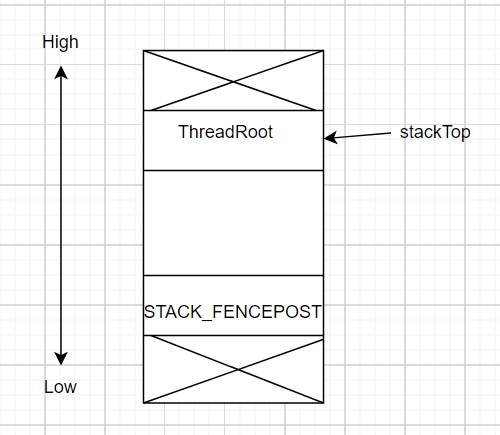
跟剛才解釋的邏輯相似，只是現在要做的事不是存取register的值，而是把newThread的 stackTop和machineState array中的值重新恢復到register中。其中，取出\_ESP(%eax)是取出newThread的stackTop而不是machineState entry的值。並將newThread的PC存到eax中。

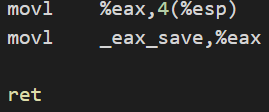


我們可以從下圖看到newThread的PC指到的是ThreadRoot。

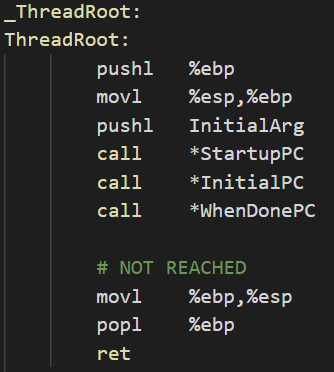


接下來將eax的值(現在是newThread的PC)存到esp指到的位置+4，最後把eax的值恢復。這裡是存到esp指到的位置+4是因為ret這個assembly code會return 回去esp指到的address並執行address的下一個指令。從下圖來看stackTop指到ThreadRoot，如果不+4，return 回去不會執行ThreadRoot。+4就可以避免這個問題。

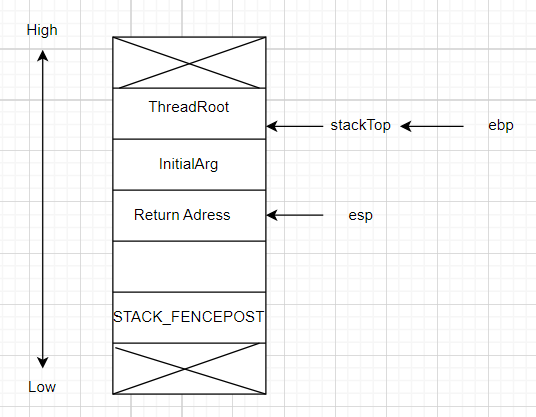




現在return 回去esp指到的位置並執行esp-4，也就是ThreadRoot。執行以下指令。



執行完後的記憶體分布圖:



StartupPC 從switch.h中知道是%ecx ，而前面知道ecx 從的值是machineState[StartupPCState]。從Thread::StackAllocate知道machineState[StartupPCState] = ThreadBegin





接下來執行ThreadBegin



接下來的assembly code也跟上面的邏輯一樣，只是執行不同涵式



InitialPC從switch.h中知道是%esi，而前面知道esi的值是machineState[InitialPCState]。從Thread::StackAllocate知道machineState[InitialPCState] = (void\*)func





func知道是ForkExecute(Thread\* t)，但有一個參數，所以回到剛剛的記憶體分布圖看到參數是InitialArg，從Thread::StackAllocate知道machineState[InitialArgState] = (void\*)arg，接下來就執行ForkExecute(Thread\* t)

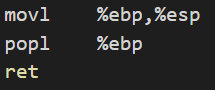
最後WhenDonePC知道是%edi。edi存的是machineState[WhenDonePCState]。

從Thread::StackAllocate知道machineState[WhenDonePCState] = ThreadFinish

最後執行ThreadFinish()



最後pop stack回到return address



**(depends on the previous process state, e.g., [New,Running,Waiting]→Ready)**

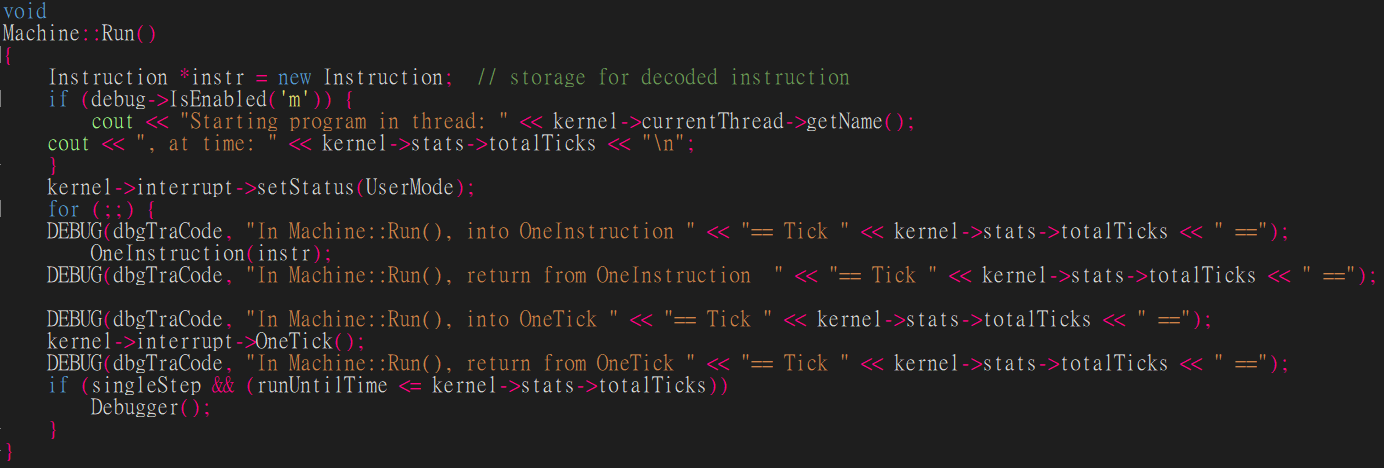
進到Ready Queue的Thread，它原本的state有以下3種可能:

1. New，也就是一開始替每個檔案新增一個Thread，接著放入Ready Queue

2. Running，因為Alarm call back，要進行time slice

3. Waiting，收到interrupt，得知I/O已做完，可以回到Ready Queue

Machine:: Run()



執行完context switch後，回到Machine Run繼續執行剩下的指令

**Part II: Implementation**

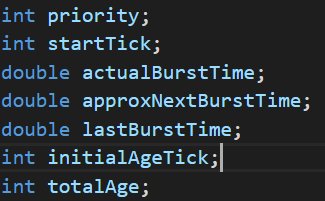
定義class scheduler新的attribute



readyList: pointer to array of pointers to List of pointers to Thread

先定義class Thread新的attribute以及method

Attribute:



priority: 打指令時要告訴OS哪一個thread的priority是多少

startTick: thread進入running state當下的tick是多少，來計算actual burst time

actualBurstTime: 字面上的意思，thread從進到running state到進入waiting state的total tick

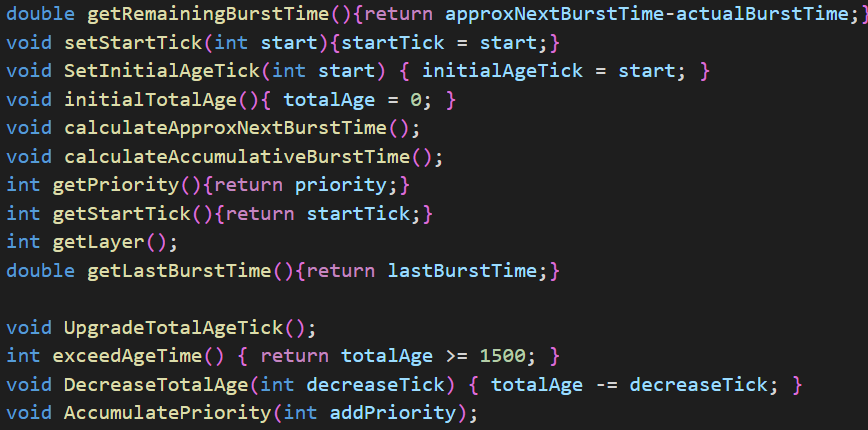
approxNextBurstTime: 當進入waiting state，要重新計算next approximate burst time

lastBurstTime: 等於actualBurstTime，我先會將actualBurstTime重置為0，但DEBUG之後會用到，所以先存起來

initialAgeTick: 進入到ready queue的當前tick

totolAge: 在ready queue 等了多久tick

Method:



getRemainingBurstTime(): 回傳 remaining approximate burst time

setStartTick(int start): 將startTick設為start

setInitialAgeTick(int start): 將initialAgeTick設為start

initalTotalAge(): 將 totalAge重置為0

calculateApproxNextBurstTime(): 在thread從running結束要進到waiting前，計算 approxNextBurstTime，並將lastBurstTime 設為actualBurstTime最後將actualBurstTime重置為0。

calculateAccumulativeBurstTime(): interrupt發生時，thread會從running 進到ready，但是CPU burst還沒做完所以先累加目前已經執行的burst time。

getPriority(): 回傳priority

getStartTick(): 回傳startTick

getLayer(): 回傳目前這個thread是在哪一個level的ready queue

getLastBurstTime(): 回傳 lastBurstTime

UpgradeTotalAgeTick(): 更新totalAgeTick

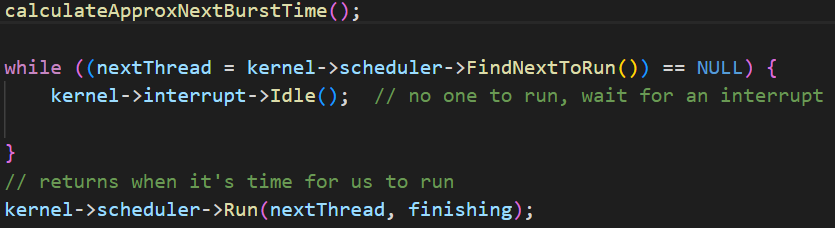
exceedAgeTime(): 回傳是否totalAge >= 1500

DecreaseTotalAge(int decreaseTick): totalAge -= decreaseTick

AccumulatePrioriry(int addPriority): priority += addPriority

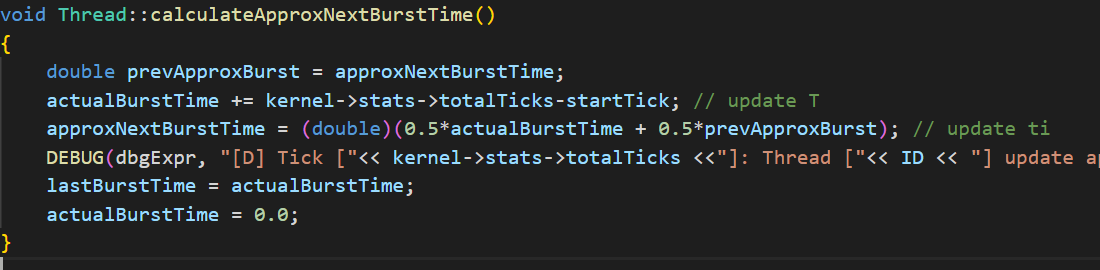
Running->Waiting

Thread::Sleep(bool)



在current thread進入waiting 之前，先計算next approximate burst time。計算完之後找從ready queue找下一個要執行的thread。最後執行nextThread。

Thread::calculateApproxNextBurstTime()



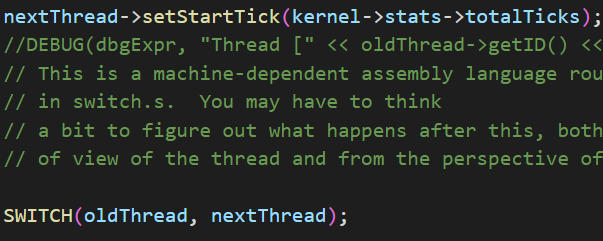
actualBurstTime可以用現在的totalTicks減掉startTick累加，用累加的是因為thread有可能因為interrupt從running回到ready，ready回到running，running到waiting，但屬於同一個CPU burst。用公式計算approxNextBurstTime。將lastBurstTime設為actualBurstTime。最後將actualBurstTime設為0。

Scheduler::FindNextToRun() (下圖以L1 queue 為例)



簡單來說就是先看L1 queue有沒有thread，沒有再判斷L2 queue，依此類推。如果L1 queue有thread，判斷 approximate remaining burst time哪一個最小，當作下一個要執行的thread。L2判斷priority，L3直接從list最前面拿出。

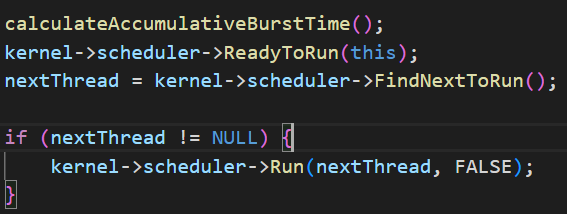
Scheduler::Run()



現在的thread要從running到waiting，下一個thread要進入running。進入running前要先設開始的tick為多少。

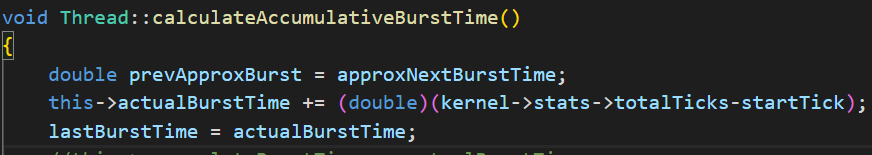
Running->Ready

Thread::Yield()



計算actualBurstTime，計算完後將current thread 放進ready queue。先放進ready queue是因為比較是整個ready queue做比較，沒放進ready queue就沒比較current thread。接著選出下一個要執行的thread。

Thread::calculateAccumulativeBurstTime()



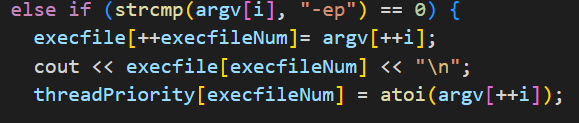
跟上面 calculateApproxBurstTime一樣只是現在不用計算approximate burst time，因為還沒有進入waiting，現在只是進入ready，還沒執行一個完整的CPU burst就被interrupt。

定義class Kernel新的attribute



threadPriority[10]: array of int，來記錄每一個thread對應的priority是多少

Kernel::Kernel(int argc, char \*\*argv)



將每一個thread 的priority放到threadPriority array中。

Kernel::Exec(char\* name)

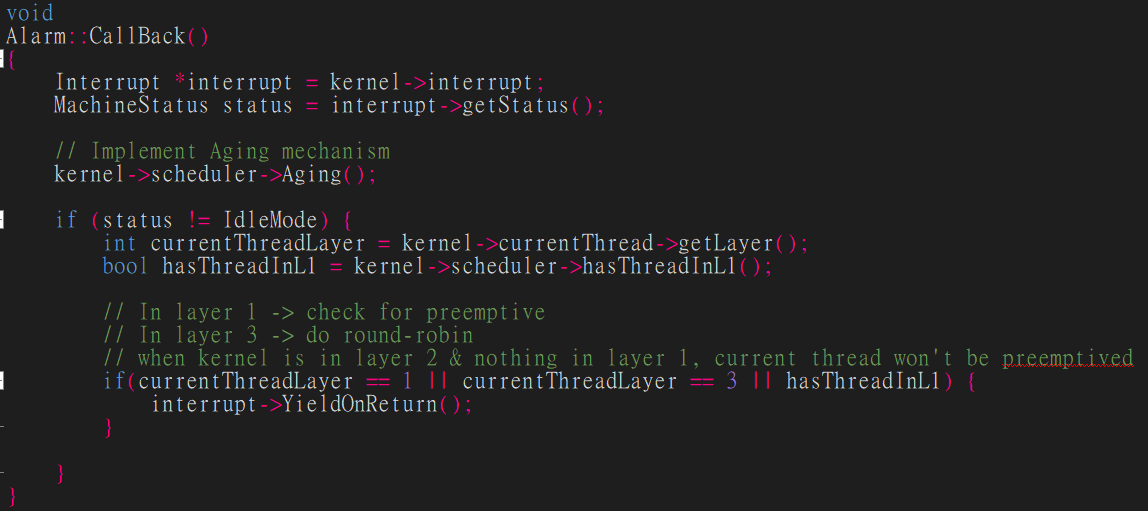


Thread constructor多一個參數把thread priority 讀進去

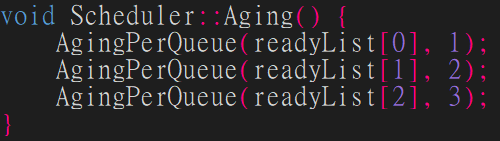
* Aging mechanism ( priority updating ) & preemption

首先介紹Aging的機制。每次執行Alarm的call back function時，也就是經過了100 TimerTicks，就呼叫scheduler裡實作的Aging function

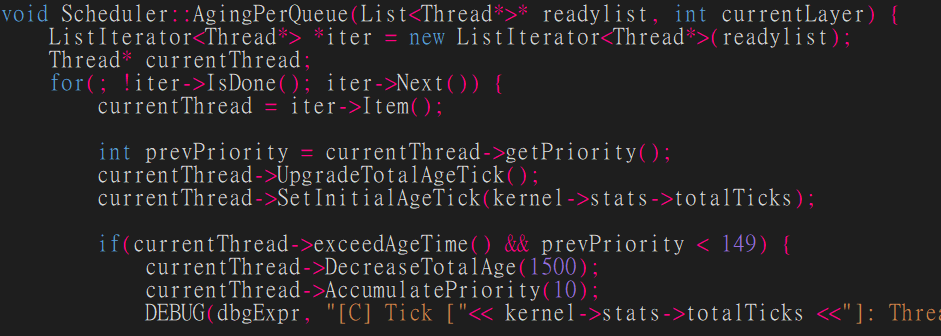
**Alarm.cc**



**Scheduler.cc**



這個函式會替multilevel feedback queue的每一層都執行Aging mechanism，也就是呼叫scheduler裡實作的AgingPerQueue function

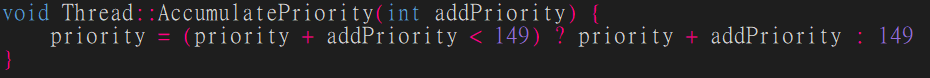


首先是上半部分，它會替當前的Ready Queue建一個ListIterator，方便透過for loop去遍歷裡頭的每一個Thread。接著就是進for loop，不斷的執行直到所有Thread都被執行到: 首先它取得當前的priority，並更新totalAge以及將initialAgeTick更新為當前時間，接著判斷

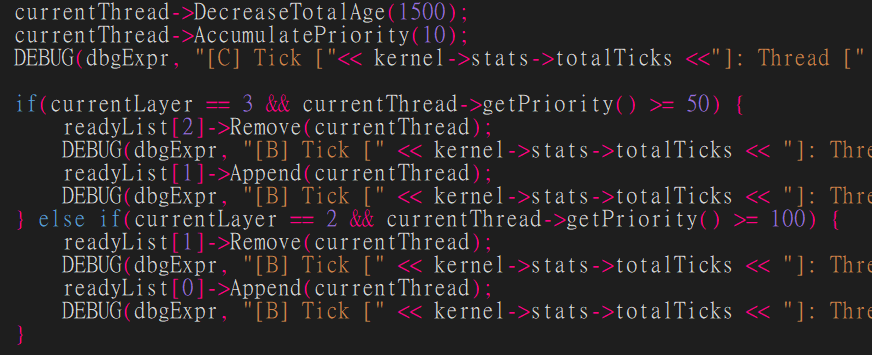
1. totalAge的值是否>=1500 (在Ready Queue中等待至少1500 ticks

2. priority是否<149 (如果>=149就沒必要更新了

假如以上條件皆成立，則將TotalAge – 1500，並增加Priority，實作如下



也就是priority += 10，但如果超過149，則最高只會是149

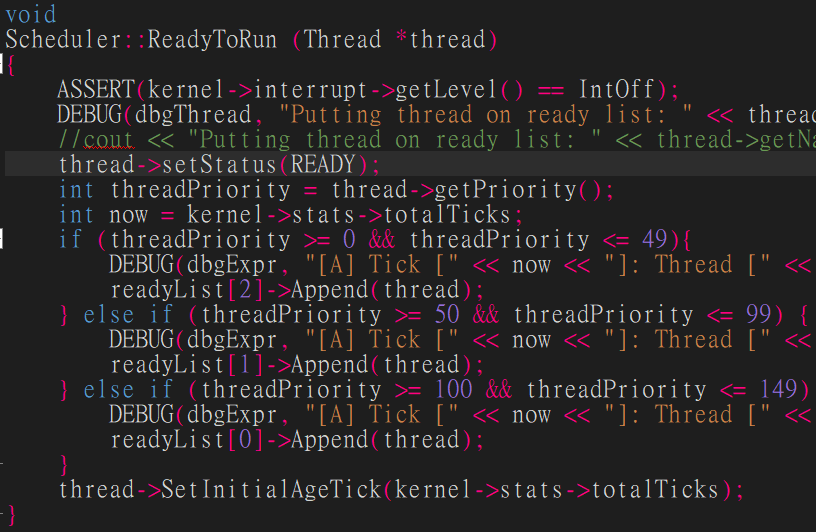


再來是下半部分，這裡是在判斷更新priority完後，是否有需要將Thread移到上一層的Ready Queue中，也就是

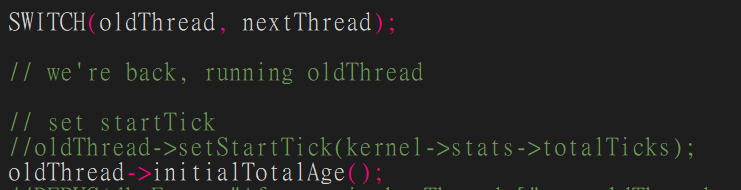
1. 假如當前在L3，若priority >= 50，則將此Thread從L3移出放入L2

2. 假如當前在L2，若priority >= 100，則將此Thread從L2移出放入L1

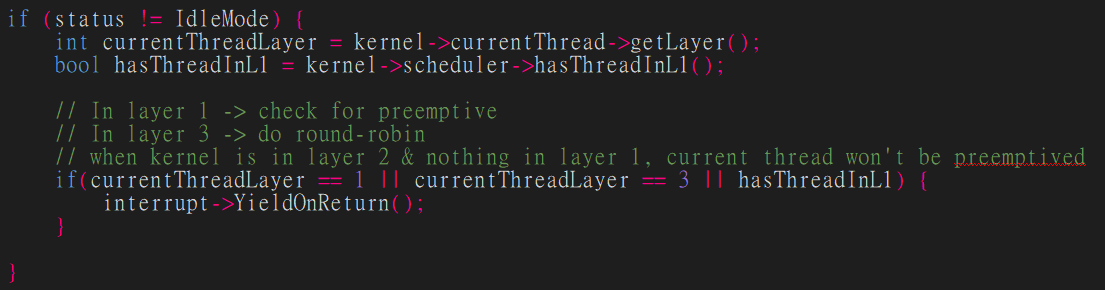
再來是小細節的部分，由於Thread可能不是在100 ticks的倍數放入Ready Queue，因此在ReadyToRun時，也就是要放入Ready Queue時，要將initialAgeTick更新為當前的值



還有就是Thread要進Running state時，要將totalAge歸零，這部分實做在context switch的正下方



最後是Preemption的實作，它在Alarm call function的下面部分



首先取得current Thread所屬的layer level，以及判斷當前L1是否有Thread待在裡頭。

接下來就是判斷是否該執行interrupt->YieldOnReturn:

1. 假如當前Thread屬於L1，則必須將其放回L1中，看有沒有其他L1中的Thread的Remaining burst time比它更小

2. 假如當前Thread屬於L2，若L1為空的，則可繼續執行下去，因為此層的scheduler algorithm為non-preemptive priority，也就是說，若有Thread在L1，則需要被preempted

3. 假如當前Thread屬於L3，由於此層的scheduler algorithm為round-robin，且time quantum = TimerTicks = 100，因此不管怎樣都需要被preempted