

2021 OS MP3

TEAM15

108062217 傅詠軒

108062203江浩辰

Contribution:

江浩辰50%

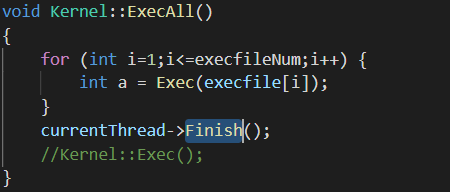
傅詠軒50%

**Trace code:**

**1-1. New→Ready**

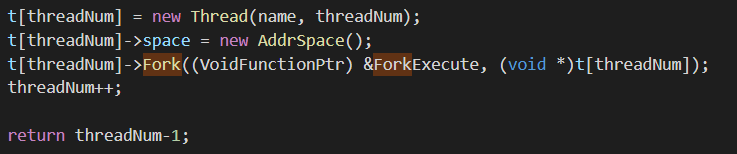
**Kernel::ExecAll**

main.cc最後會呼叫kernel->ExecAll()來執行execfile array中要執行的user program，進到ExecAll()中，用for loop呼叫Exec()來一一執行所有user programs，待全部執行完畢後，會呼叫Thread::finish()表示main thread執行完所有execfiles了。



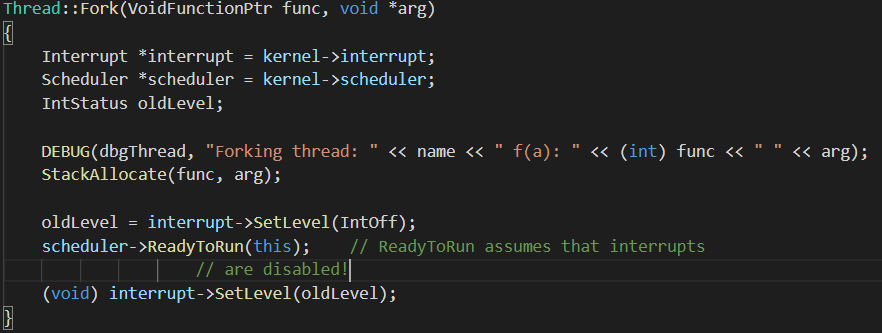
**Kernel::Exec**

ExecAll呼叫Exec後，在這邊透過以下function來allocate一個new thread object、allocate user address space給這個thread，以及call Thread::Fork()來對這個thread做一些設定，最後更新threadnum。



**Thread::Fork**

呼叫StackAllocate()來allocate一個stack、初始化stack，設定好machinestate，將interrupt設為disabled，呼叫Scheduler->ReadyToRun(this) 將此thread放入ready queue中，最後將interrupt設成原本的狀態。

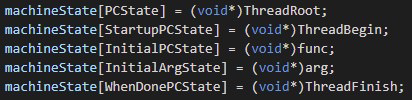


**Thread::StackAllocate**

allocate一個stack並初始化。



並將machinestate設定好，包含root，開始結束要call的function。



**Scheduler::ReadyToRun**

將thread的狀態設定為Ready，但還沒Running。

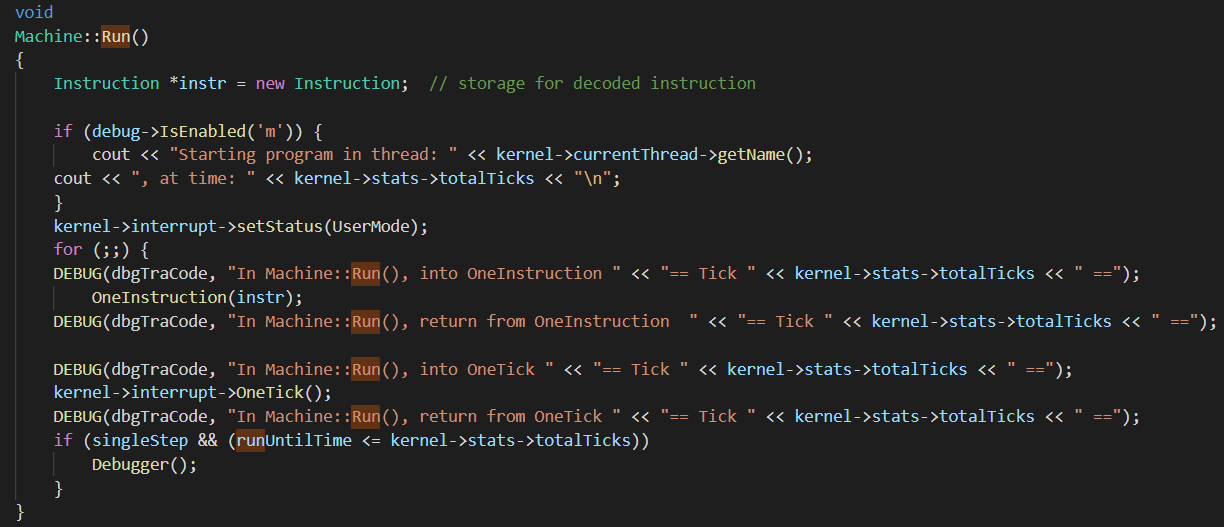
並將此thread加入ready queue之中，等待被執行。



**1-2. Running→Ready**

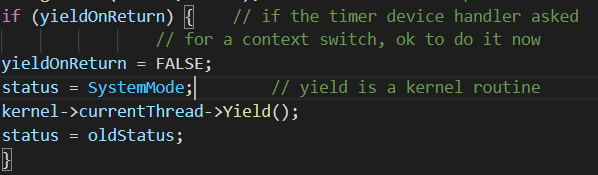
**Machine::Run()**

Run()的過程中會進行無限迴圈重複呼叫 Machine::OneInstruction()執行一行user-level的instruction、Interrupt:OneTick()以及一些 debug 的指令。

****

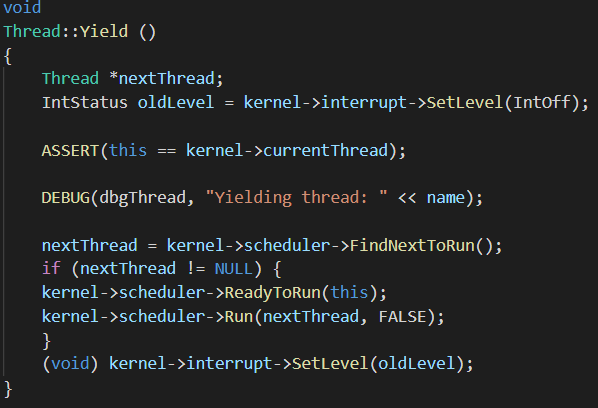
**Interrupt:OneTick()**

會advance simulated time並且檢查有沒有pending的interrupt。接著透過check ”yieldOnReturn” 來看timer有沒有請求context switch，由於此時thread還可以繼續running，但被timer打斷，所以此時timer ask的context switch是hardware interrupt，會把thread放回ready queue中，會先flip mode to kernel然後呼叫Thread::Yield()來完成。



**Thread::Yield()**

先disable interrupt，接著呼叫Scheduler::FindNextToRun() 選出ready queue中的process來執行，如果ready queue沒有其他process了，直接return，會繼續執行原本process，如果有找到就呼叫Scheduler::ReadyToRun()把原本的process加到ready queue的尾巴，之後呼叫Scheduler::Run() 來執行新選中的程式，最後把interrupt enable。



**Scheduler::FindNextToRun()**

如果ready queue是空的，return NULL，如果有thread，則return queue中最前方的thread，並把他從ready queue移出。

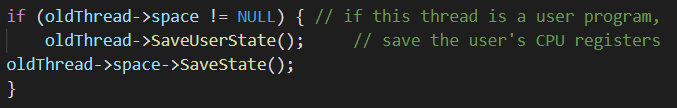
**Scheduler::ReadyToRun**

將原本thread的狀態設定為Ready。並將原本的thread加入ready queue的尾端，等待被執行。

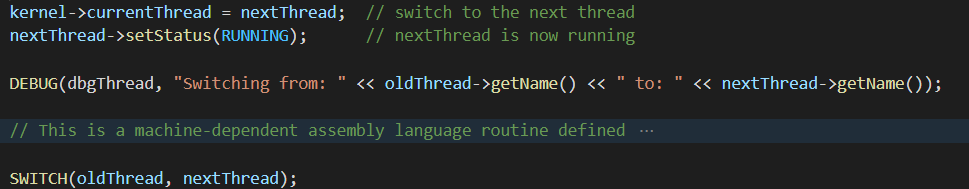


**Scheduler::Run()**

把原本的thread的state記錄下來(save the user's CPU registers)。



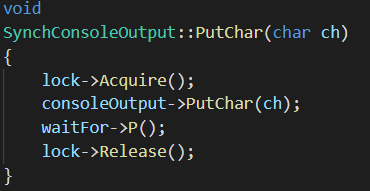
接著把kernel的current thread指向被選中的thread，並把狀態設成running。接著呼叫SWITCH來context switch。



**1-3. Running→Waiting**

**SynchConsoleOutput::PutChar(char)**

SynchConsoleOutput::PutChar()會呼叫ConsoleOutput->PutChar(ch)把character print到display裝置。先呼叫lock.acquire()上鎖確保一次只有一個writer，再呼叫ConsoleOutput->PutChar(char)將欲輸出的數值印出來，在output尚未完成之前(還沒callback前)透過Semaphore::P()使得process sleep並等在waiting queue內，等到callback呼叫Semaphore::V()了，P()就不會卡在while loop執行到底後再利用lock.Release()解鎖。



**Semaphore::P()**

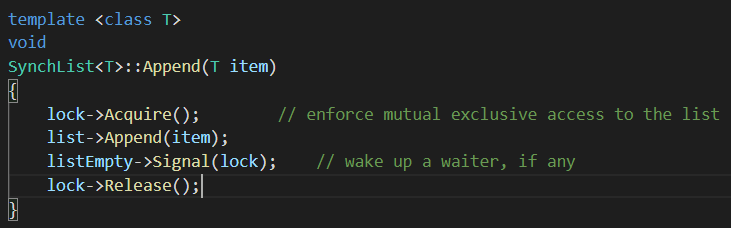
透過semaphore來執行同步化，由於output一開始所create的semaphore object的計數器(value) = 0，所以在PutChar內呼叫的P()會使得output卡在P()內的while loop，直到output做完後的callback function呼叫V()把value++後才會從while loop break，之後才會呼叫lock->Release()把output讓出來，否則在做完output之前不會release lock。

首先會把interrupt disable，之後進到while loop直到計數器(value)不等於0，在while內會把此等待output完成的process append到queue中，並呼叫Thread::Sleep()來釋放CPU，因為此時此process從running到waiting state，直到做完呼叫V()後，減計數器1，並把interrupt設成原本狀態。



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

利用append()將item放入指定list的最尾端，且透過lock避免有多個processes同時對list進行access。這裡是提供Semaphore::P()方法將currentThread放入waiting queue之中。



**Thread::Sleep(bool)**

Sleep 被呼叫有兩種情況：1. 目前的thread執行完畢，傳入的finishing為true，2. 目前的thread被block住（semaphore, lock等等），此時我們的狀況是第二種，因為semaphore同步化所以被block住。

我們會讓目前的thread釋放CPU資源，並且透過scheduler來找到新的thread來執行（FindNextToRun()），達成context switch。

如果在ready queue中找不到可以執行的thread時，CPU處於Idle狀態，等待新的I/O interrupt（這樣才會有新的thread可以 run），如果有可以執行的thread則透過呼叫Scheduler::Run()執行被選中thread。

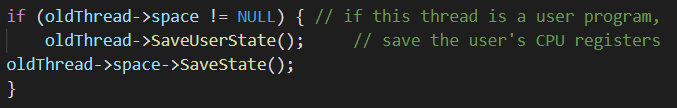


**Scheduler::FindNextToRun()**

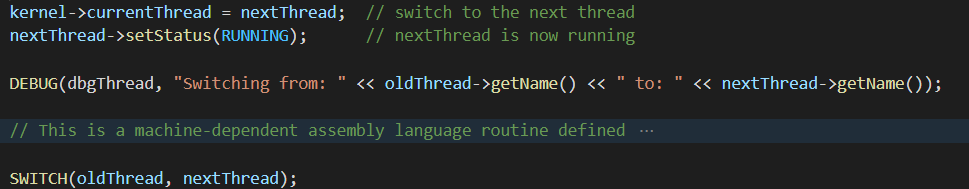
如果ready queue是空的，return NULL，如果有thread，則return queue中最前方的thread，並把他從ready queue移出。

**Scheduler::Run()**

把原本的thread的state記錄下來(save the user's CPU registers)。



****接著把kernel的current thread指向被選中的thread，並把狀態設成running。接著呼叫SWITCH來context switch。



**1-4. Waiting→Ready**

**Semaphore::V()**

當output結束後的callback function會呼叫Semaphore::V()，先把interrupt關掉，之後檢查waiting queue內有沒有thread（此處會有因為前面的P()有append），把waiting queue內的thread放回ready queue中（by call scheduler::ReadyToRun()），並把計數器++使得P()可以執行完並能夠release lock，最後把interrupt handler設回原本狀態。



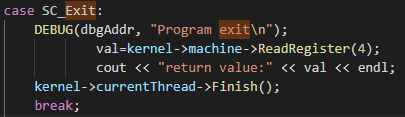
**Scheduler::ReadyToRun**

將原本thread的狀態設定為Ready。並將原本的thread加入ready queue的尾端，等待被執行。



**1.5 Running->Terminated**

**ExceptionHandler(ExceptionType) case SC\_Exit**

****

當一個thread即將執行結束時，會raise一個exception。

ExceptionHandler先判斷為SystemCallExcepton，再由type得知為SC\_EXIT。

ExceptionHandler會先將我們要return的值從register r4中取出並輸出，再呼叫kernel->currentThread->Finish()來引發之後的context switch。

**Thread::Finish()**

呼叫Sleep(True)讓系統知道currentThread是已經執行完成的狀態。

**Thread::Sleep(bool)**

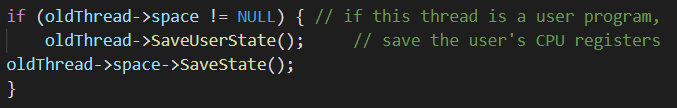
由於上面所傳入的finishing為true，所以可以知道currentThread是已完成的狀態，接著再利用kernel->scheduler->FindNextToRun()找到下一個待執行的thread，再呼叫kernel->scheduler->Run()來觸發context switch。

**Scheduler::FindNextToRun()**

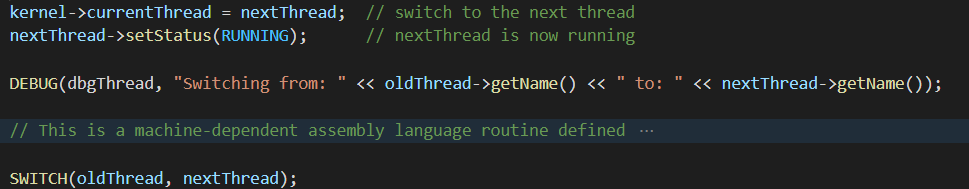
如果ready queue是空的，return NULL，如果有thread，則return queue中最前方的thread，並把他從ready queue移出。

**Scheduler::Run(Thread\*, bool)**

把原本的thread的state記錄下來(save the user's CPU registers)。



接著把kernel的current thread指向被選中的thread，並把狀態設成running。接著呼叫SWITCH來context switch。

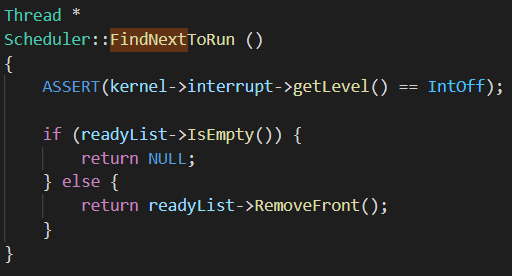


**1-6. Ready→Running**

**Scheduler::FindNextToRun()**

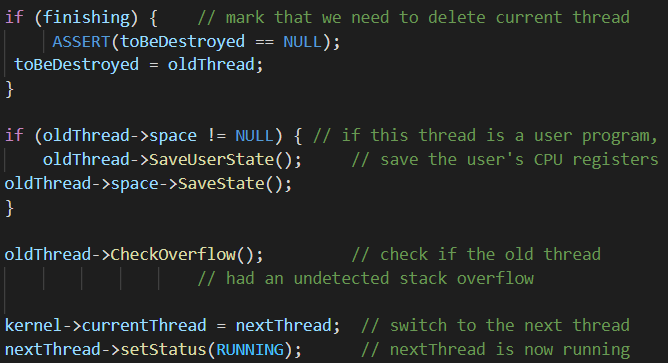
從ready queue中找到下一個執行的thread。

如果ready queue是空的，return NULL，如果有thread，則return queue中最前方的thread，並把他從ready queue移出。

****

**Scheduler::Run(Thread\*, bool)**

**根據傳入的finishing決定舊的thread是不是執行完畢需要deallocate，之後將**原本的thread的state記錄下來(save the user's CPU registers)，接著將kernel-> currentthread指向新選中的thread，並將新選中的thread設成running。



接著呼叫SWITCH()進行context switch。

**SWITCH(Thread\*, Thread\*)**

直接由assembly code寫成的context switch routine。

一開始stack上長得像這樣：

8(esp) -> thread \*t2 (新thread的address)

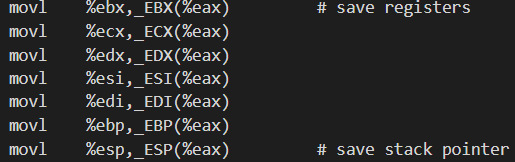
****4(esp) -> thread \*t1 (舊thread的address)

(esp) -> return address

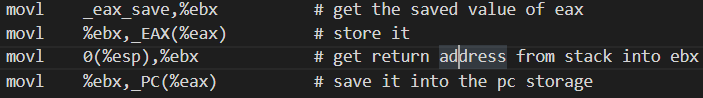
第一行先把目前thread的ptr address存到variable eax\_save，之後把old thread的address move到eax方便把registers存下來。



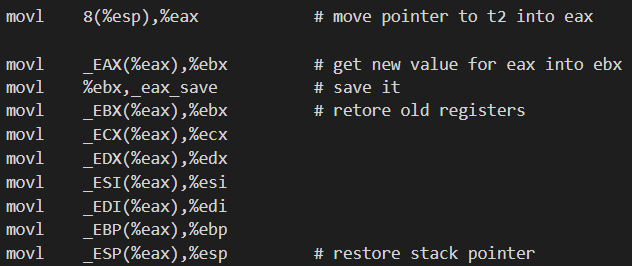
之後把registers的值存到old thread指定的儲存位置內，並存取stack ptr。



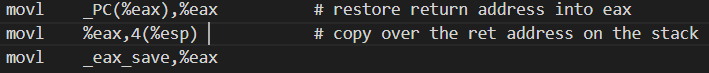
把eax\_save內的值存到old thread指定的EAX位置。之後把return address存到old thread的PC中用以告知之後從這開始執行。



把new thread address移到eax中，把eax中的值存到eax\_save，並把new thread所存下的registers放到相對應的regsiters中，並存取stack ptr。



最後把PC放到stack上4(esp)的位置，代表SWITCH結束後會從這邊開始執行→執行new thread。並把new thread eax的值放到register eax中。



**for loop in Machine::Run()**

接著前一個thread(old thread)會回到ready state，然後在Run()內的OneInstruction()則會開始執行new thread的code。

**Implementation**

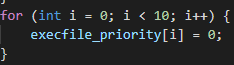
**kernel.h**

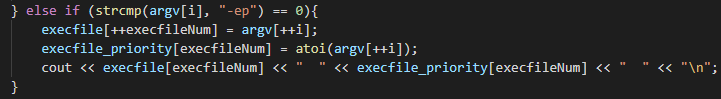
新增execfile\_priority[]用來存execfile的priority。



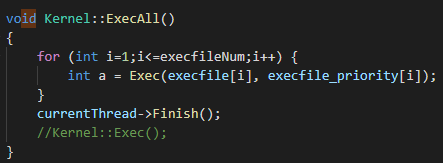
**kernel.cc**

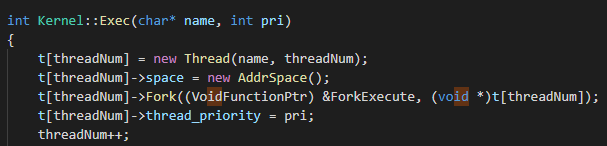
初始化execfile\_priority，並在Kernel::Kernel()中新增command line “-ep”。



****

在Exec()中多加一個priority的Input，並將其指派到新的thread中。





**Scheduler.h**

新增三個ready queue到scheduler中，分別是L1, L2, L3。

因為L1, L2需要依據burst time跟priority排順序，所以使用SortedList。

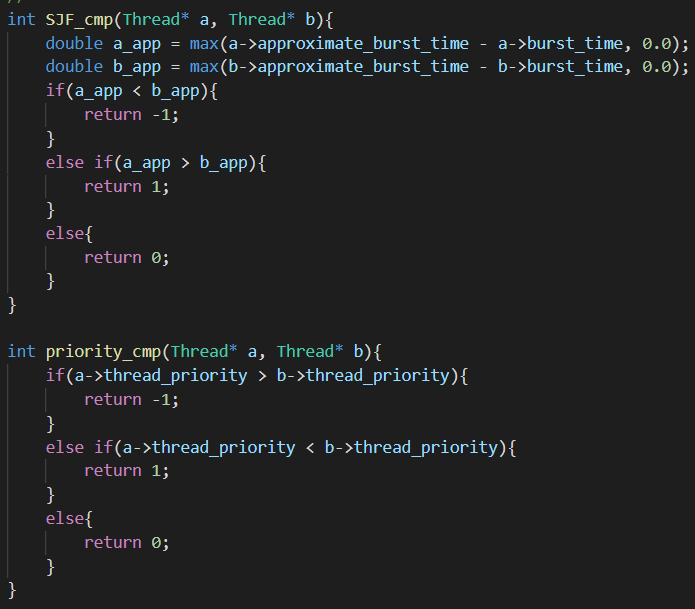


**Scheduler.cc**

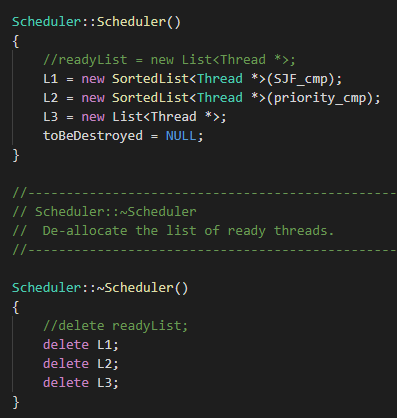
新增L1, L2的compare funtion。

L1 : remaining burst time 小的在前面。

L2 : priority大的在前面。



在建構子與解構子新增三個ready queue的宣告與刪除。



****因為現在變成了multilevel queue，所以在跑ReadyToRun()時必須依照priority的大小放入相對應的queue中，並設置開始等待進入running的時間(for aging)。



由於優先順序從高到低分別是L1, L2, L3，在FindNextToRun()中需要從上到下尋找，當L1為空時才會到L2找，L2為空時才會到L3找。

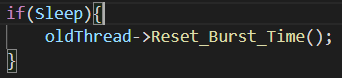
並在找到時，reset waiting\_time = 0，這是記錄此thread在ready state等了多久，用來實現aging。



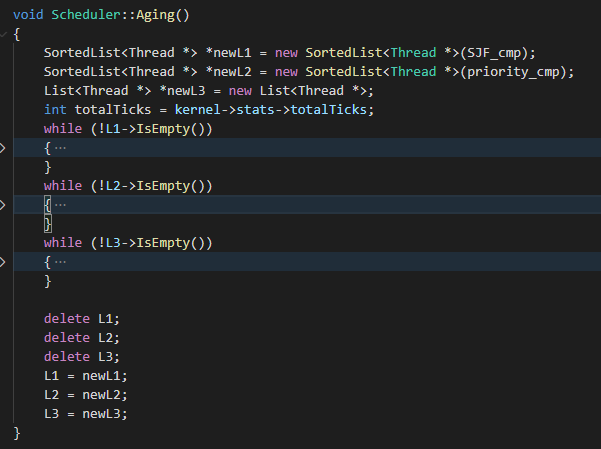
在scheduler::Run() set被選出來下個要跑的thread的start\_time，代表進入running的起始tick，方便之後計算burst time。

並且如果是因為Sleep，也就是I/O造成thread進入waiting的context switch，就把burst time設為0，代表進到下一個burst cycle。





****Aging() 我們的做法是創建另外的newL1, newL2, newL3，更新完內容後再讓L1,L2,L3 = newL1,newL2,newL3。



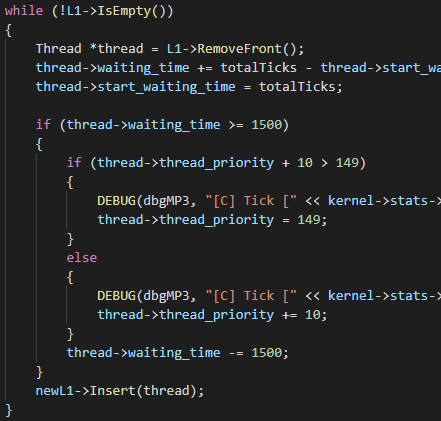
L1的部分，抓出所有thread來更新priority。

start\_waiting\_time紀錄目前距離上次aging或未aging過，開始等待的時間。

用totalTicks(也就是當前時間)減掉start\_waiting\_time就會等於總共等待了多久時間，並將這個時間加到waiting\_time中。

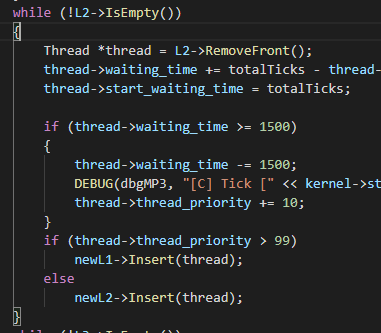
當waiting\_time >= 1500時，必須執行aging，priority增加10，但因為這是L1，加完可能出現超過149的情況，所以必須在此判斷。

最後如果aging的話waiting\_time需減掉1500。

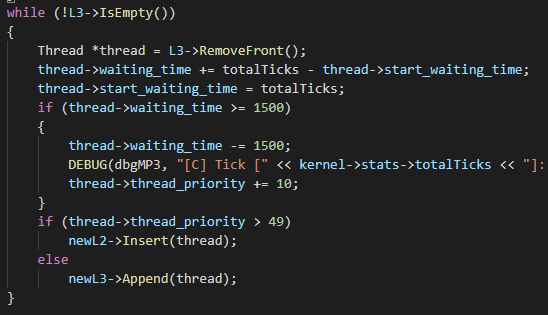


****L2的部分，照L1的方法，判斷是否需要aging，即priority是否要加10。

並判斷Thread的priority+10之後會不會超過99，如果超過就要被搬到L1。

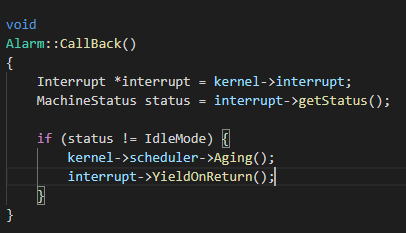


L3的部分，一樣要多判斷是否需要被搬到L2。



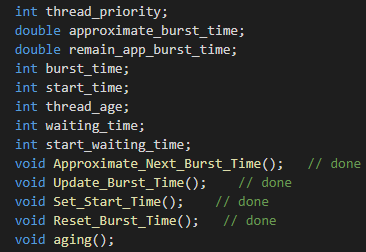
**Alarm.cc**

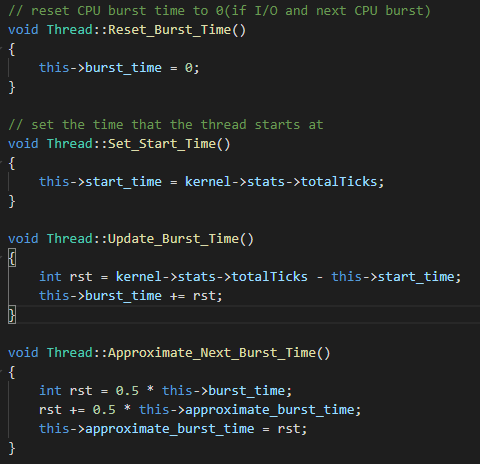
在Alarm.cc中的Alarm::CallBack()呼叫kernel->scheduler->aging()，並執行interrupt->YieldonReturn來呼叫Yield()。



**Thread.h**

新增所需的property 以及function。



****

**Thread.cc**

在Thread::Yield()更新burst time，並將目前執行的thread放回ready queue中，再從上而下從queue中尋找下一個要執行的thread，來達成在timer interrupt之後進行preempt的行為。

值得注意的是我們必須判斷目前正在執行的thread是否在L2，如果是的話，如果有新的thread加入L2且priority比較小，會導致L2的thread被L2的thread preempt，所以要多加判斷式來避免這種情況。



在Thread::Sleep()，由於代表I/O進入waiting，更新burst time以及approximated burst time。

