Operating System

MP3 Report

組別：25

陳凱揚：Trace code、Implement、Report  
簡佩如：Trace code、Implement、Report

1. Trace code
   1. New 🡪 Ready
2. Kernel::ExecAll()、Kernel::Exec(char\*)

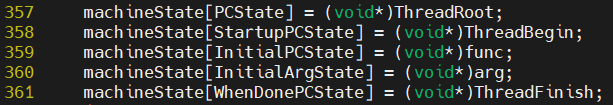
ExecAll()會將execfile裡的每個執行檔名稱作為參數，分別執行Exec()。接著在Exec()裡就會建立新的thread，並分配addrspace、呼叫Fork()。

1. Thread::Fork(VoidFunctionPtr, void\*)

在Fork()裡，會先以StackAllocate()分配一塊stack給這個thread，接著就以ReadyToRun()將這個thread加進ready queue裡。

1. Thread::StackAllocate(VoidFunctionPtr, void\*)

在StackAllocate()裡，會先給stack一塊新的記憶體，再根據不同CPU架構（這邊是x86架構）對stackTop的位置有所調整，最後初始化machineState，其中func為剛剛傳進來的ForkExecute，arg為t[threadNum] （圖1-1）。



▲ 圖1-1

1. Scheduler::ReadyToRun(Thread\*)

在ReadyToRun()裡，會將thread狀態設為ready，並加進ready queue裡。

* 1. Running 🡪Ready

1. Machine::Run()、Interrupt::OneTick()

在Run()裡有個無限迴圈不斷執行user的程式，並且呼叫OneTick()。在OneTick()裡首先會增加模擬的時間，接著檢查有沒有hardware interrupt要執行，最後檢查timer是否要求context switch，如果需要則會呼叫Yield()。

1. Thread::Yield()

在Yield()裡，會呼叫FindNextToRun()尋找下一個在ready state的thread來執行，再呼叫ReadyToRun()將現在這個thread重新加回ready queue裡面，最後呼叫Run()執行context switch。

1. Scheduler::FindNextToRun()、Scheduler::ReadyToRun(Thread\*)

在FindNextToRun()裡會根據schedule的方法尋找下一個執行的thread並回傳。而ReadyToRun()會把thread加進ready queue裡。

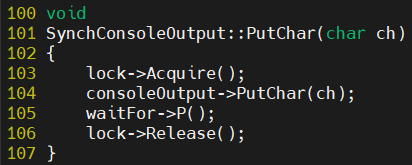
1. Scheduler::Run(Thread\*, bool)

在Run()裡的功能為執行context switch，首先會記錄現在的thread是否結束，接著如果這個thread是user program，就會儲存user state，再來就能以SWITCH()進行context switch，最後當程式又再次回來後，會檢查是否結束這個thread，並檢查是否需要還原user state。

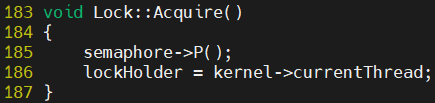
* 1. Running 🡪Waiting

1. SynchConsoleOutput::PutChar(char)

在PutChar()裡，會先執行lock->Acuquire()（圖1-2），其中除了做P()外，也將lockHolder設為現在的thread（圖1-3），也就是只有他能解開lock，接著呼叫PutChar()輸出字元，並排程一個hardware interrupt在未來，最後呼叫P()。



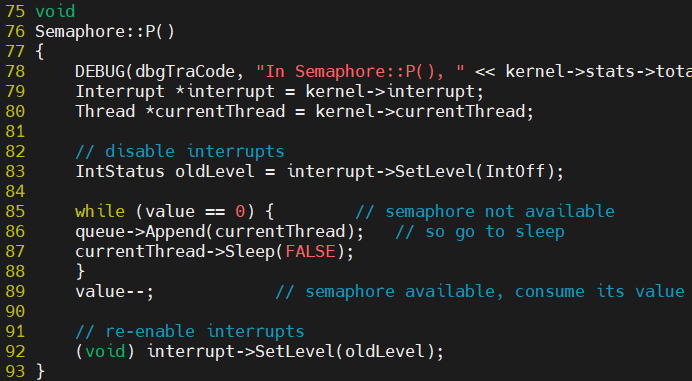
▲ 圖1-2



▲ 圖1-3

1. Semaphore::P()

在P()裡，如果value大於0，需要value--，代表消耗掉可用的資源；如果value為0，則會呼叫Append()將這個thread加入等待的queue裡，並呼叫Sleep()，參數為FALSE代表進入waiting state（圖1-4）。



▲ 圖1-4

1. List<T>::Append(T)

在Append()裡會將傳進的thread\*存到list的最後面。

1. Thread::Sleep(bool)

在Sleep()裡會將狀態設為blocked，直到有其他thread來叫醒他，並將他放回ready queue裡re-scheduled。而此時若沒有其他thread在ready queue裡，就會呼叫Idle()表示現在CPU在閒置中，直到找到下一個thread並呼叫Run()來進行context switch。

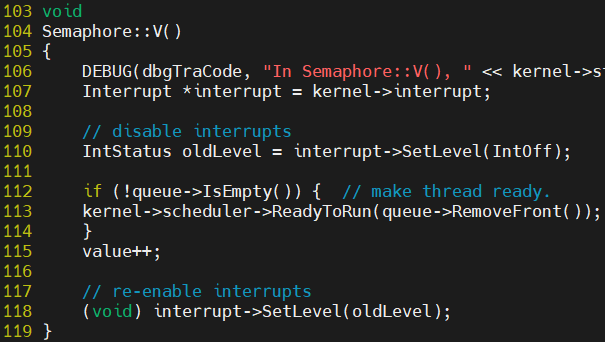
1. Scheduler::FindNextToRun()、Scheduler::Run(Thread\*, bool)

FindNextToRun()會找到下一個thread來執行，Run()會進行context switch。

* 1. Waiting 🡪Ready

1. Semaphore::V()

在V()裡，會從queue裡叫醒一個等待中的thread，並呼叫ReadyToRun()，讓他重回ready state，再value++，代表釋放出資源（圖1-5）。



▲ 圖1-5

1. Scheduler::ReadyToRun(Thread\*)

ReadyToRun()會將傳進的thread加進ready queue裡。

* 1. Running 🡪Terminated

1. ExceptionHandler(ExceptionType)

當user program呼叫Exit()這個system call時，會進到ExceptionHandler裡的SC\_Exit，此時會呼叫Finish()來結束目前這個thread。

1. Thread::Finish()

在Finish()裡會呼叫Sleep()，參數為TRUE代表thread結束。

1. Thread::Sleep(bool)

在Sleep()裡會呼叫FindNextToRun()去尋找下一個thread，再呼叫Run()進行context switch，並傳入剛剛的TRUE，讓Run()知道待會要結束這個thread。

1. Scheduler::FindNextToRun()、Scheduler::Run(Thread\*, bool)

FindNextToRun()會找到下一個thread來執行，Run()會進行context switch。

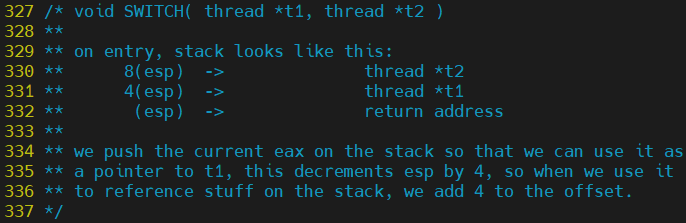
* 1. Ready 🡪Running

1. Scheduler::FindNextToRun()、Scheduler::Run(Thread\*, bool)

FindNextToRun()會找到下一個thread來執行，Run()會呼叫SWITCH()來進行context switch。

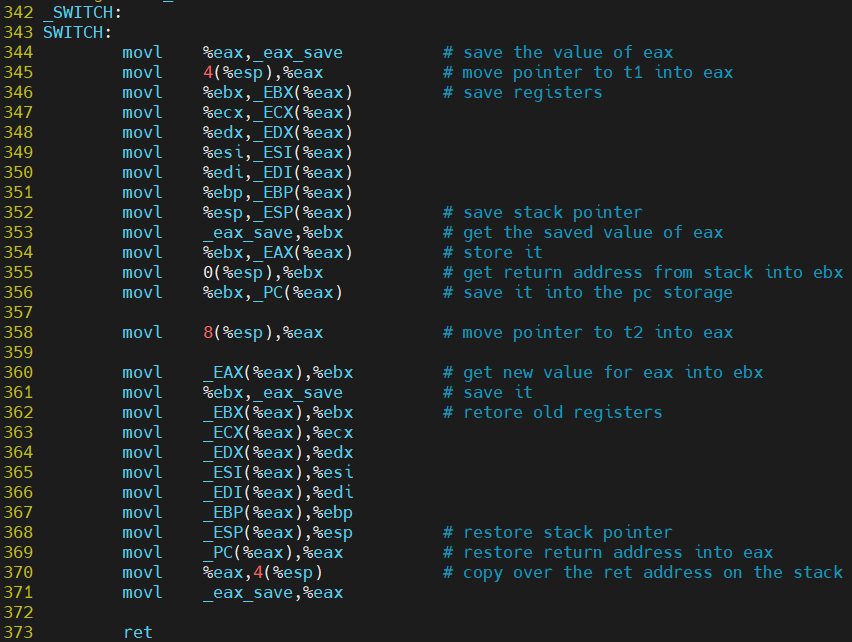
1. SWITCH(Thread\*, Thread\*)

由圖1-6可知目前esp指向的記憶體存著return address，esp+4指向的記憶體存著oldThread的address，esp+8指向的記憶體存著nextThread的address。



▲ 圖1-6

下圖1-7，首先在第344行會將oldThread放在eax的資料存進\_eax\_save；第345行會將oldThread的位址存進eax，方便接下來的第346~352行以eax為基準，分別將ebx、ecx、edx、esi、edi、ebp、esp等register儲存的舊資料存回oldThread的register裡；第353~354行將剛剛儲存起來的eax資料也存回oldThread的register裡；第355行~356行將return address存進oldThread的register裡，讓程式之後能夠回到oldThread目前的中斷點；第358行將nextThread的位址存進eax中；第360~368行將存於nextThread的register的資料存進CPU的register中；第369~370行將之前nextThread的中斷點存到esp+4中，使程式在這個函數結束後，就能開始執行nextThread；最後371~373行恢復eax的值並ret。



▲ 圖1-7

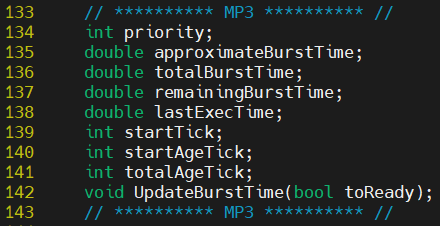
1. Depends on the previous process state

在SWITCH()結束後，程式會從上次這個thread的中斷點開始繼續執行。

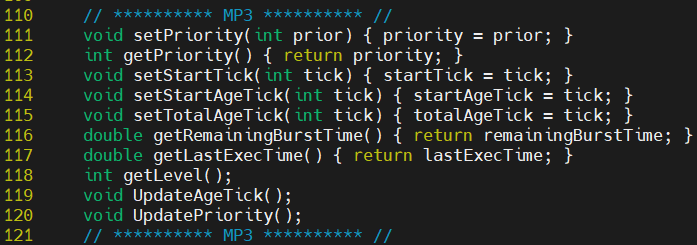
1. Implement
   1. Multilevel feedback queue
2. /threads/thread.h

下圖2-1為新增於Thread的private variable和function，其中lastExecTime記錄上次使用CPU的時間，供debug使用；startTick記錄進入running state的時間點；startAgeTick記錄上次更新totalAgeTick的時間點；UpdateBurstTime()用於更新approximate和remaining burst time，參數toReady在runnung to ready時設為TRUE，在running to waiting時設為FALSE。

下圖2-2則為新增的public function，其中getLvel()會根據priority回傳目前所在ready queue的level，UpdateAgeTick()和UpdatePriority()在timer到期時，會被呼叫來更新age和priority。



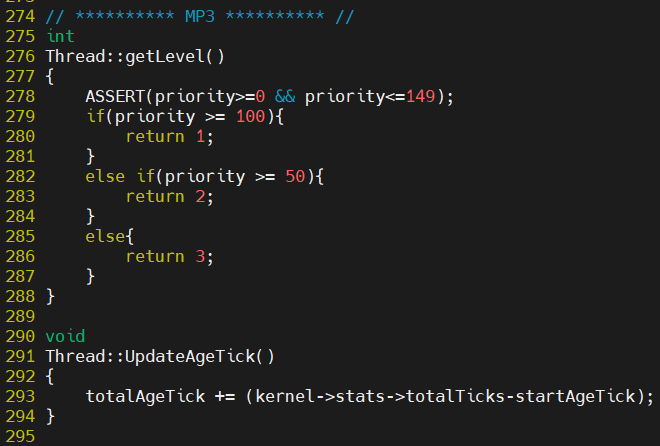
▲ 圖2-1



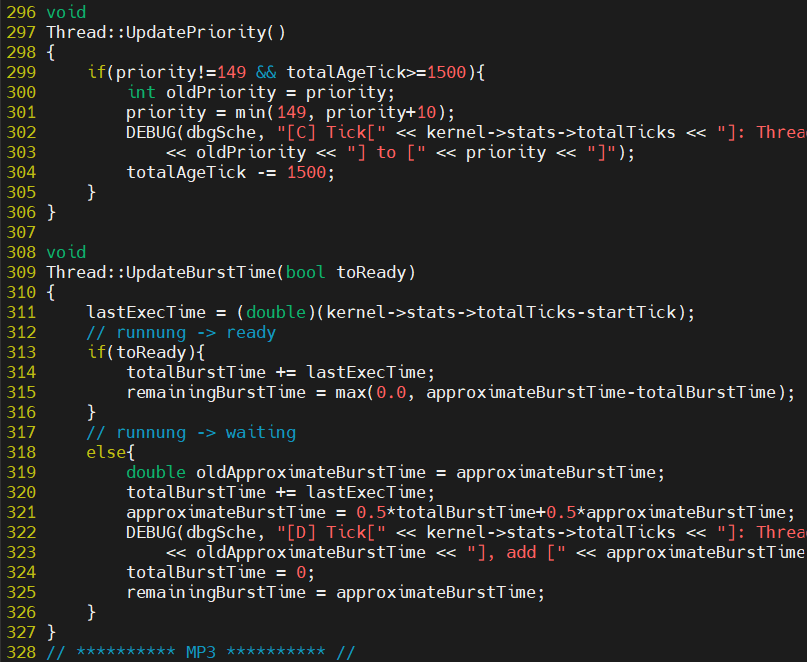
▲ 圖2-2

1. /threads/thread.cc

圖2-3為getLevel()和UpdateAgeTick()的實作，圖2-4則為UpdatePriority()和UpdateBurstTime()的實作，其中UpdatePriority()中的priority最大只會加到149；在UpdateBurstTime()中會根據toReady有兩種不同的做法，當TRUE的時候，只會更新burst time，並計算remaining burst time（最少為0），不會更新approximate burst time；當FALSE時（進入wating state），就會計算新的approximate burst time，同時將remaining burst time設為新的估計時間。

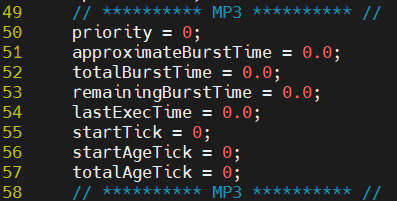


▲ 圖2-3

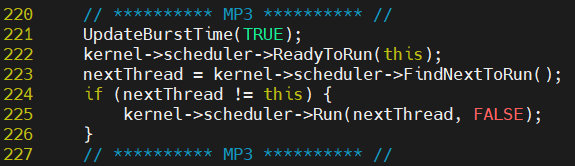


▲ 圖2-4

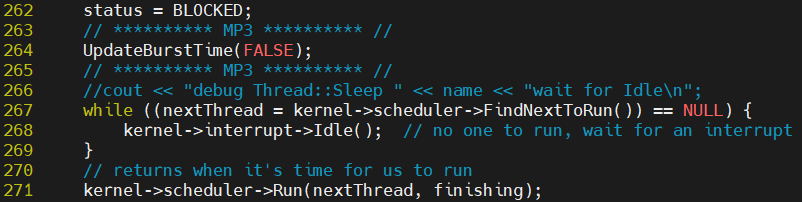
下圖2-5為在constrctor中會將這些變數初始化為0。圖2-6為在Yield()中，會先呼叫UpdateBurstTime()，並將現在這個thread也加入ready queue裡，再去尋找下一個要執行的thread，如果還是同一個的話，就不用呼叫Run()來進行context switch，直接繼續執行下去。圖2-7為在Sleep()中，會先呼叫UpdateBurstTime()，再去尋找下一個要執行的thread。



▲ 圖2-5



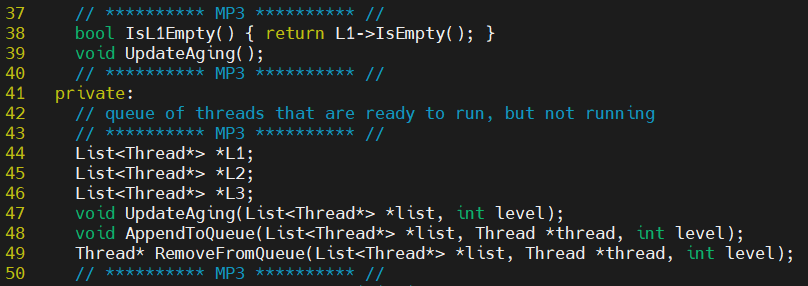
▲ 圖2-6



▲ 圖2-7

1. /threads/scheduler.h

圖2-8中，前半部為新增於Scheduler中的public function，後半部則為private的variable和function，其中UpdateAging()會在timer到期時被呼叫。

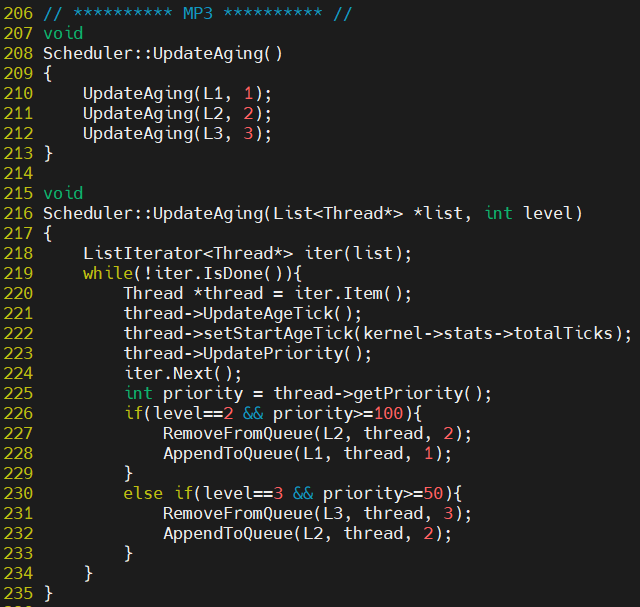


▲ 圖2-8

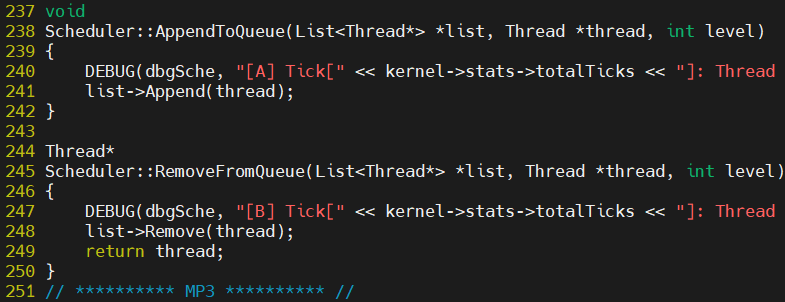
1. /threads/scheduler.cc

下圖2-9為UpdateAging()的實作，內容為將每個ready queue裡面的每個thread都更新age和priority，因此會使用ListIterator走過所有thread，而在更新後，會檢查L2的和L3的thread是否需要提升至上一層ready queue裡。

下圖2-10為AppendToQueue()和RemoveFromQueue()的實作，內容會根據傳入的List來Append或Remove傳入的的thread。

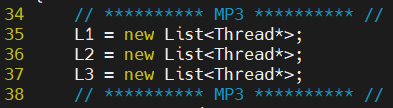
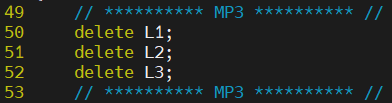


▲ 圖2-9

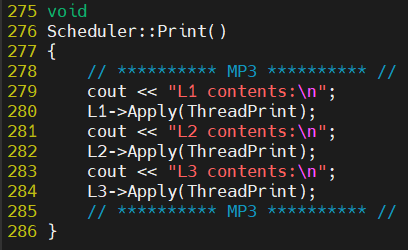


▲ 圖2-10

下圖2-11、圖2-12分別為在constrctor、destroctor中new及delete掉L1、L2、L3。圖2-13則為在Print()分別輸出3個ready queue的訊息。

▲ 圖2-11 ▲ 圖2-12

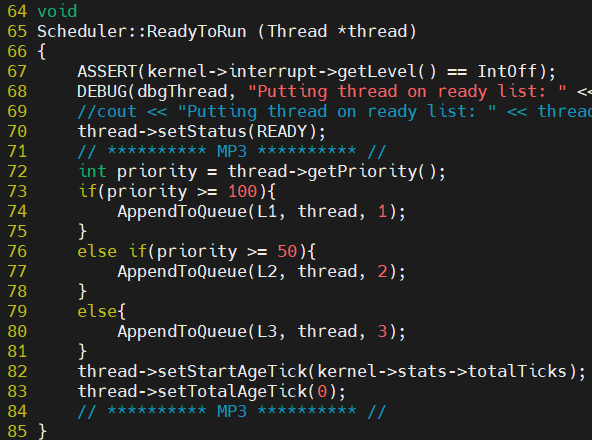


▲ 圖2-13

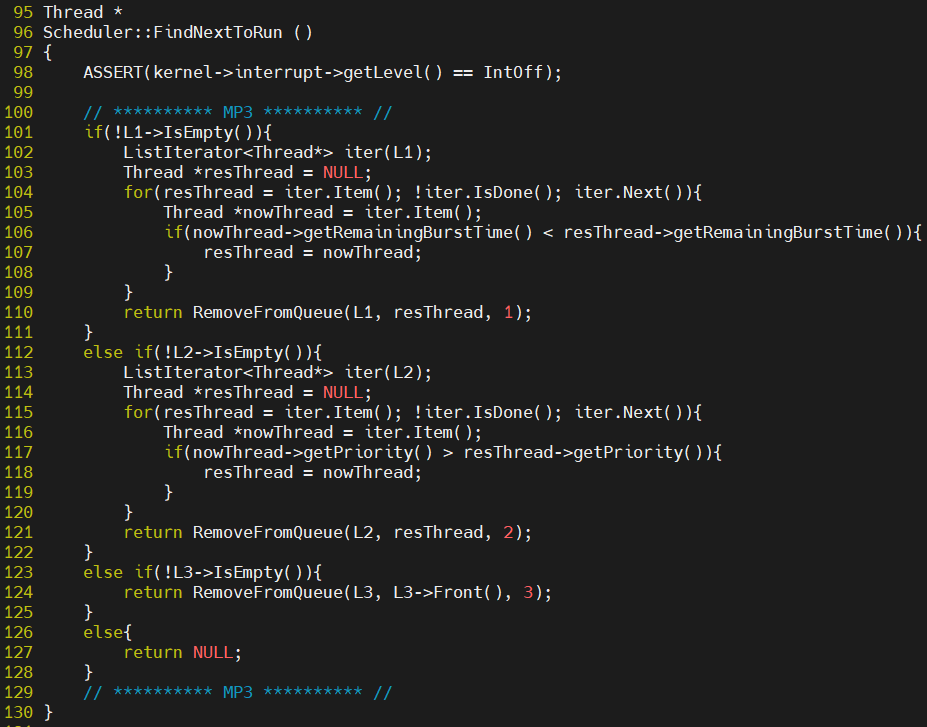
下圖2-14為在ReadyToRun()中，會根據thread的priority來決定要把thread加進哪一個ready queue裡，並將startAgeTick設為現在的時間點、totalAgeTick設為0。

下圖2-15為在FindNextToRun()中，會根據L1、L2、L3的順序去找出下一個要執行的thread，在L1裡會尋找remaining burst time最小的thread、在L2裡會尋找priority最大的thread、在L3以FCFS選擇排在最前面的thread。

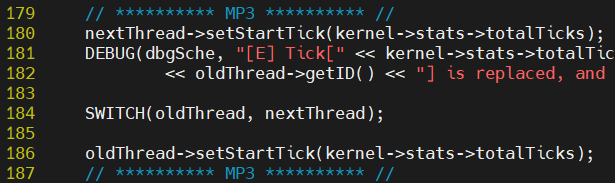
下圖2-16為在Run()中，在SWITCH前後會分別將新舊thread的start tick設為目前的時間點。



▲ 圖2-14



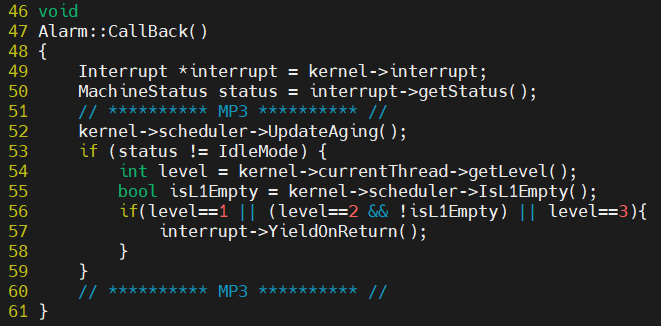
▲ 圖2-15



▲ 圖2-16

1. /threads/alarm.cc

timer每次到期時，會進入下圖2-17的CallBack()裡，在這邊會先呼叫UpdateAging()來更新所有在ready state的thread的age和priority，接著如果目前的thread在L1或是在L2且L1不為空或是在L3，則會呼叫YieldOnReturn()，來讓其他thread能夠preempt。

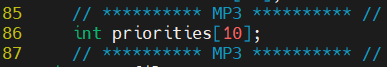


▲ 圖2-17

* 1. Command line argument “-ep”

1. /threads/kernel.h

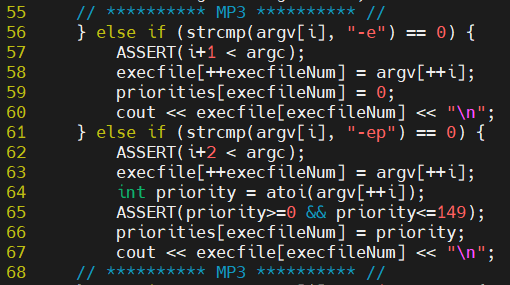
下圖2-18在Exec()裡增加了一個新的參數priority，而圖2-19增加了一個新的private variable，用於儲存使用者輸入的priority。

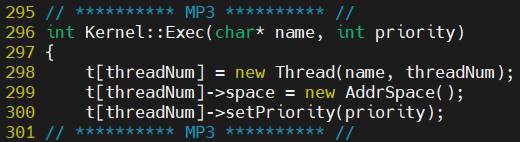
▲ 圖2-18 ▲ 圖2-19

1. /threads/kernel.cc

下圖2-20為在construcotr中，當參數為”-e”時，將priority設為0；當參數為”-ep”時，將priority設為使用者輸入的數字，並檢查此數字必須介於0-149。下圖2-21為在Exec()裡會新增setPriority()的動作。



▲ 圖2-20



▲ 圖2-21

* 1. Dubugging flag z

1. /lib/debug.h

下圖2-22為在debug.h中加上這個flag，讓使用者能以”-d z”開啟功能，而以下為此5種debug訊息所在位置的function。

[A]：Scheduler::AppendToQueue()

[B]：Scheduler::RemoverFromQueue()

[C]：Thread::UpdatePriority()

[D]：Thread::UpdateBurstTime()

[E]：Scheduler::Run()



▲ 圖2-22

1. Difficulties and Feedback

在這次作業中，我覺得在implement的部分比起前兩次作業難度增加了不少，不僅要知道哪些檔案的函式需要修改，也需要額外添加許多變數和函式，其中也有很多小細節都是在實作時才會發現，像是在running to ready時，必須先更新approximate burst time，並加回ready queue，才去尋找下一個thread，因此有可能又是相同的thread繼續執行下去。此外，在debug的部分也蠻困難的，雖然有了自己加上debug訊息，但每100tick就可能會發生preempt，又經常是同一個thread連續搶到CPU，訊息就會不斷顯示移出又插入ready queue的不重要訊息。而在trace code的部分，很多函式之前就已經都認識了，所以輕鬆了不少，唯一比較需要注意的只有SWITCH()中的組合語言。雖然這次作業相對來說比較重，但透過實作也更加了解了CPU schedule的方法和不同演算法的細節差異，也複習到了簡單的組合語言語法。