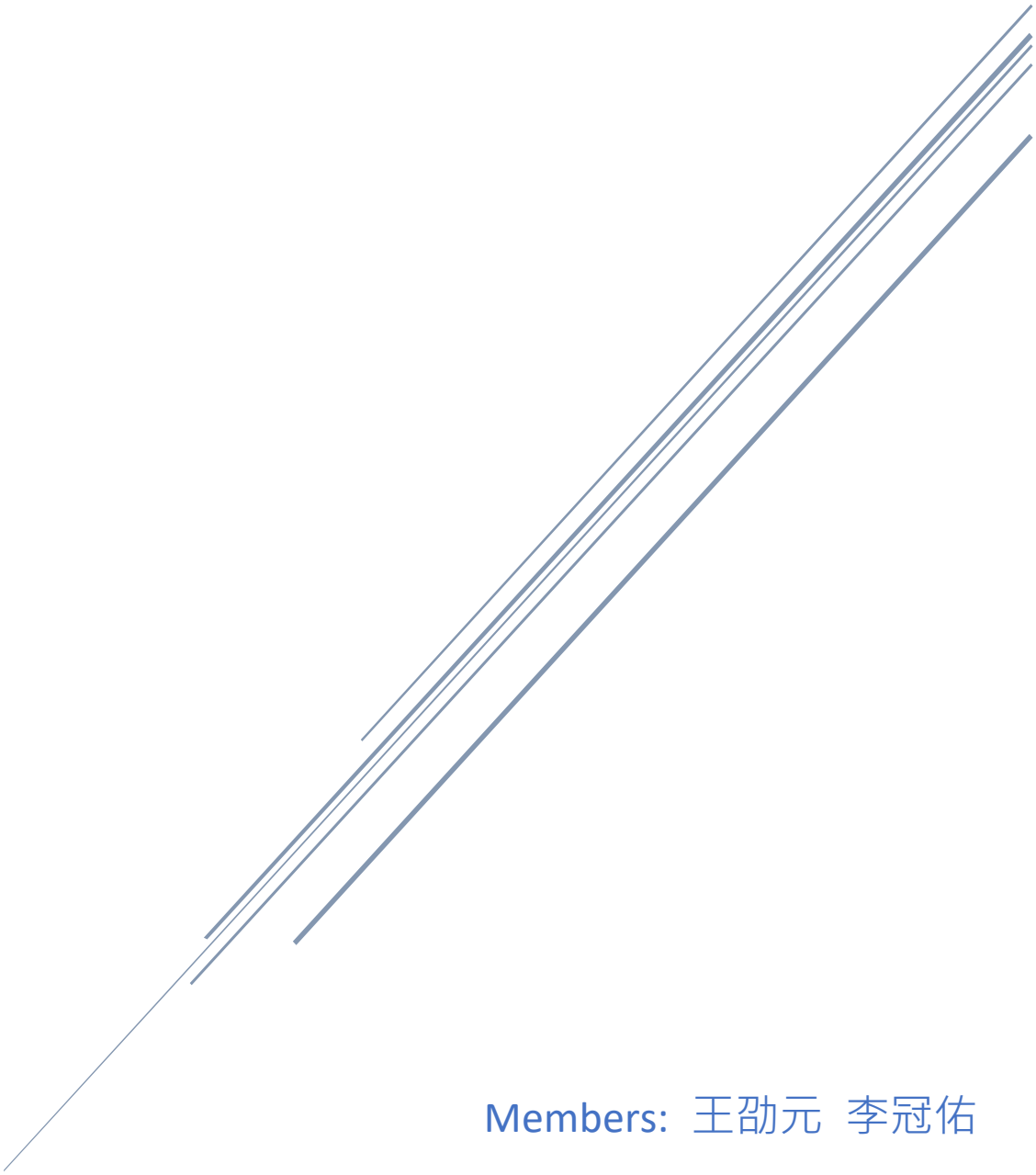
MP2:MUTI-PROGRAMMING

Report



Members: 王劭元 李冠佑

Contribution: 50 % 50 %

Part 1: Trace code

Trace code 的部分我們是使用 implement 前的 code 進行解釋

1. New->Ready

Kernel::ExecAll():

在這個 function 是將前面讀取 command 時，讀到的所有 program，將 program name 往下傳遞，讓下層的 function 可以繼續對他操作。

Kernel::Exec():

在這個 function，就是將每隻程式用一個 thread 去處理，並且會 new 出 addrspace，最後會去呼叫 Thread 的 Fork 來去 initialize stack，並且將 ForkExecute 這個 function pointer 傳下去。

Thread::Fork():

在這個 function 主要就是做兩件事，第一就是去呼叫 StackAllocate 去 Allocate 一個 process 自己的 stack，並 initialize 一些東西，並將 Kernel 傳下來的 ForkExecute 也再向下傳。最後會將此 Thread call scheduler 放進 Readyqueue 裡面。

Thread::StackAllocate:

在這個 function 的一開始，他就先去 Allocate 出了一個 stack，並且設定 stacktop 的位置，最後會將 ThreadRoot、ThreadBegin、ForkExecute 等等，都放在設定好的位置。等到 context switch 的時候，就可以輕易的獲得這些位置，得到一個新的 process 會需要的 argument 以及初始化（將 program load 進 memory 等等），之後一個新的 process 就可以這樣開始運作。在這之後一個 process 就已經 New 完了

Sceduler::ReadytoRun():

在這裡面就是將這個要 ReadytoRun 的 Thread 的 Status 改為 Ready，並將這個 Thread 丟進 Readyqueue 裡，意即可以開始被 Scheduler 從 Readyqueue 裡選中開始執行。在這邊開始，Thread 也就從 New state 變為 Ready State 了。

2. Running->Ready

Machine::Run():

在這個 function 會先去呼叫 oneinstruction()，去 fetch 現在要執行的指令，之後去呼叫 Onetick 去更新系統的 Tick，並且去檢查 Interrupt 等等。

Interrupt::OneTick():

當 Onetick 裡，讀到 yieldOnReturn 這個變數是 True 的時候，代表這時候要去 preempt 出現在的 Thread，讓 Scheduler 重新選擇要執行的 Thread，故這時會切換到 system mode，並且去讓 currentThread 去執行 Yield。

Thread::Yield():

在這裡面要做的事情就是放棄、讓出 CPU 的資源，讓 Scheduler 可以重新選擇 Thread，故裡面就會去呼叫 Scheduler 的 FindNextToRun，找到下一個要執行的 Thread，只要 Readyqueue 不是空的，就一定可以找到下一個等待執行的 Thread。若有找到，就會去要 scheduler 再把自己放回 Readyqueue，因為這個 Thread 是執行到一半，還需要再被執行的。並且讓 scheduler 去 call Run。但在這邊，我們在實作的部分有把被 preempt 出的 Thread 先放回 Readyqueue 裡，再去選擇下一個要被執行的 Thread，表示有可能 A thread 被 preempt 了，但又被選到要執行。

Scheduler::FindNextToRun():

在這個 Function，要做的事情只有先檢查 Readyqueue 是不是空的，如果是空的就 return Null，不然就把 Readyqueue 的 Front() pop out 並 return 回去。(直接取 Front 是因為現在是 FIFO)

Scheduler::ReadyToRun():

這邊的 ReadyToRun 跟上面有說過的也是做相同的事情，將自己 status 設成 Ready 並放到 Readyqueue。

Scheduler::Run():

在這裡面就會要去做 Context switch，因此會先去將一些需要儲存的資訊儲存起來，並且檢查一下有沒有 overflow，並把 current Thread 設成被 scheduler 挑中的 thread。接著就去做 Context switch。在這邊 old thread 的狀態就正式從 Running state 轉為 Ready state 了。

3. Running->Waiting

SynchConsoleOutput::PutChar():

在一開始，會先用 lock 把這段 code 鎖起來，不會有多個 Thread 一起執行這邊的 code。接著 Call 完 consoleOutput 的 putchar 後，就會去呼叫 semaphore 的 P()。最後確認 output 完後，就會把 lock 交出去，讓其他人也可以進來做。

Semaphore::P():

在這邊會去檢查 value 是不是為 0，如果是 0 代表現在沒有資源可以使用，而這邊的意思是，當還沒 output 完的話，console output 的 callback 和 synchConsoleOutput 的 callback 就也不會被 call 到，這樣這個 semaphore 的 V()就不會被 call 到。就會一直卡在這邊，所以這個 Thread 就會去 call Sleep()，將 CPU 讓出去。

Thread::Sleep():

當 call 到這個 sleep 的時候，代表這次 CPU burst 結束了，會把自己的 status 設成 BLOCKED，於是就要去找下一個要執行的 Thread，若沒有下一個要執行的 Thread 的話，就代表整個 system 暫時沒有人要用 CPU 做事，於是就會去 call Idle，這樣 system 就可以快進時間。若有找到下一個 thread 的話，就會去 call scheduler 的 run。

Scheduler::FindNextToRun():

這個 function 上面也介紹過了，要做的事大致就是從 Readyqueue 裡找出下一個要執行的 Thread。

Scheduler::Run():

在這邊也跟上面介紹的一樣，但由於在這邊 Thread 沒有重新放進 readyqueue，status 也是 BLOCKED，故在這邊 Thread 的狀態就正式從 Running state to Waiting state 了。

4. Waiting->Ready

Semaphore::V():

在 call 了 V()之後，若剛剛被卡在此 semaphore 的 queue 不是空的，那代表有人在等到此 resource。但這邊的意思是，output char 完成了。因此此 thread 就

可以被從這個 queue 裡拿出來，並 call ReadyToRun，回到 Readyqueue 裡，重新開始等待 scheduler 挑到就可以去使用 CPU 資源。最後就可以把 value++，代表有 resource 可以使用了。

Scheduler::ReadyToRun():

這個 function 做的事情也跟上面介紹的一樣，將 thread 的 status 設為 Ready，並放進 readyqueue。

5. Running->Terminated

ExceptionHandler(Exception type) case SC_Exit

在這個 case 中，會先去讀去這個 process 最後的 return value 並 print 出來，接著就呼叫 Thread Finish。

Thread::Finish():

在 Finish 中，由於他要做的事情與 Sleep 相似，於是在裡面就去 call Sleep 並傳入 True，代表這個 Thread 上的程式已經結束了。

Thread::Sleep():

在 Sleep 一開始做的事情也一樣，將 status 設成 BLOCKED，並找到下一個要執行的 Thread，接著會去 call Scheduler 的 Run()，並繼續把 True 傳下去，以告知這個 Thread 已經是 Finish 了。

Scheduler::FIndNextToRun():

在這邊也跟上述一樣，主要即為找出下一個要執行的 Thread。

Scheduler::Run():

在這邊由於傳進來的 finish 的 bool 為 true，因此會將 toBeDestroyed 設為 oldThread。接著做的事情就一樣，然後 context switch 完，下一個 Thread 往 SWITCH()下面開始執行後，就會去 CheckToBeDestroyed()，這時就會發現 toBeDestroyed 不為 NULL，就會將 delete 剛剛 finish 的 thread。在這之後，Thread 就正式從 Running state 變成 Terminated 了。

6. Ready->Running

Scheduler::FindNextToRun():

在這邊跟上面介紹的一樣，在這次的情況，假設 scheduler 選到的 nextThread 為我們接下來要看的 Thread。

Scheduler::Run():

要做的事情跟上面有介紹的的一樣，儲存完資訊後，就會去執行 context switch。

SWITCH(Thread*, Thread*):

在 switch 的一開始，會先把 old Thread 的 stack pointer 記錄下來，接著再把 register s0-s7($16-$23)，也都存起來，最後也會把 frame pointer 以及 programe counter 存下來。再來到 next Thread，會跟剛剛做類似的事情，但是是把之前存下來的 stack pointer, register s0-s7, frame pointer, program counter，都 load 回去，之後在讀指令時，就會從 next Thread 的 PC 去讀取，並且 register 的狀態也回復到 next Thread 以前被 switch 的狀態，這樣即可做到了 context switch。

在 switch 的最後，會 jump 回 return address，若此 thread 是新的 thread，那這時候就會跳到 ThreadRoot，這時候他就會去 jump 到 startUpPC，這樣就會去 call 到 ThreadBegin，再往下 call 到 Thread::Begin()的話，就會在裡面 CheckToBeDestroyed，並且 enable interrupt。

再來 jump 到 InitialPC 的話就會去執行 ForkExecute，若有需要 argument 的話，也在上面的 instruction move 到 a0 了。

最後如果這個 Thread 有 finish 的話，才會 jump 到 WhenDonePC，去做 clean up procedure。

(depends on the previous process state)

New->Ready:

若是從 New state 到 Ready state 的話，第一次就會去執行 ForkExecute，讓 thread 把 program load 進 memory 並建立 pagetable 等等。接著就會去執行 space->Execute，在這裡面就會去 call 到 Machine::Run()，就可以開始執行 instruction。

Running->Ready:

若為 Running to Ready state 的話，由於是從 oneTick 呼叫的，就會一個個 return 回去，回到 oneTick 後，由於 oneTick 又是 Machine::Run()呼叫的，故最後也會 return 回 Machine::Run()，又可以開始執行 instruction。

Waiting->Ready:

若為 Waiting 到 Ready state 的話，由於這邊的 V()，會讓 value++告知已經完成 output 了，故當原本卡在 V()等待的人，就可以跳出迴圈，並且一層層先 return 回 SysConsoleOutput::putChar()，再 return 回 ksyscall（但這邊我們還沒有直接定義，故也有可能是 putInt 呼叫的，但終究還是會 return 回 ksyscall），最後 return 回 exception handler，最後 return 回 OneInstruction，繼續往下執行。

For loop in Machine::Run():

在這邊開始，就可以去 Fetch instruction 後開始執行 user 要的 program。

Part 2 Implementation

Kernel.cc/kernel.h

首先，在 kernel.h 裡我們在 class kernel 的 private 裡新增一個名為 ThreadPriority 的陣列，是用來儲存 input thread 初始的 priority，接著在 kernel.cc 裡面，新增了一個新的 command line argument -ep，目的是將 thread 的 priority 初始化，我們一樣將 thread name 從 argv 讀取後存在 execfile[execfileNum]裡，execfileNum 是用來表示現在讀取到第幾個程式，然後將初始的 priority 先用 atoi 轉成 integer 後，再存到 threadPriority[execfileNum]裡，就完成了 kernel.cc/h 的 implement。

thread.cc/thread.h

在 thread.h 的 thread control block 裡，我們一共新增了 10 個 function 和 7 個 private 變數，以下一一介紹：

AddTicksInQueue()

AddTicksInQueue()是用來更新 thread 在 queue 裡的時間，只要拿 kernel 的 totalTicks 減掉 AgeBaseline 並用 TicksInQueue 累加上去，獲得的就是 thread 到現在在 queue 裡總共待了多久時間，主要在兩個地方會 call 到，一個是在做 aging 前，會去更新 thread 在 queue 裡的時間，另一個是在 FindNextToRun 的時候，下一個要執行的 thread 再從 queue remove 前，要更新一次時間，因為他接下來就不會在 queue 裡，因此要結算一下時間。

UpdateAgeBaseline()

private 變數 AgeBaseline 是要用來計算 thread 在 queue 裡待的時間，它會紀錄 thread 在 queue 裡一開始的時間點，之後只要拿 kernel 的 totalTicks 減掉 AgeBaseline 就可以得到這個 thread 在 queue 裡待的時間，而 UpdateAgeBaseline()就是用來更新 AgeBaseline，它會在兩個地方做更新，第一個地方是在新產生的 thread 在 ReadyToRun 被放到 queue 裡後就要 call UpdateAgeBaseline()做一次 AgeBaseline 的更新，第二個地方是在 Alarm 的 call back 後（一個 time slice 時間到了後），要去 aging 前，會先去 call AddTicksInQueue()去更新 thread 在 queue 的時間，之後就可以判斷是否需要做 aging(>=1500)，因為 call 了 AddTicksInQueue，因此需要將 AgeBaseline 做更新，把時間點設為現在 kernel 的 totalTicks，之後才可以獲得 thread 真正在 queue 裡

待的時間。

Getlevel()

Getlevel()的用途很單純，就是要獲得先在這個 thread 是在哪個 level 的 queue 的資訊，若是 priority 介於 100-149 代表在 L1 queue，因此 return 1，若介於 50-99，代表在 L2 queue，return 2，若是介於 0-49，代表是在 L3 queue，因此 return 3。

SetPriority(int num)

setPriority(int num)是用於更新 thread 的 priority，會拿本來的 priority 加上 num 以得到新的 priority，主要用在兩個地方，一個是在 thread 剛產生要初始 priority 時，另一個是若 thread 在 queue 裡待超過 1500 秒，要做 aging 增加 priority 時，會 call SetPriority(10)。

GetPriority()

GetPriority()單純就是要獲得現在 thread 的 priority，會直接 return priority 回去。

HandleAgingOld()

HandleAgingOld()的用途是判斷 thread 是否已經待在 Readyqueue 太久，要做 priority 的提升，若 TicksInQueue >= 1500 就 return true，並將 TicksInQueue-1500。否則就會 return false。

CalPredictBurst()

CalPredictBurst()會在 thread 由 running 進到 waiting state 的時候去預測下一次的

burst time，預測方法是拿這一次的 burst time(NowBurst)和這一次的 predict burst time(Predict)去取平均來預測下一次的 burst time(newPredict)，再來會將 nowBurst 的值給 AccuExecTime（為了印出 DEBUG 訊息，不然不需要這個變數），並將 NowBurst 歸零，Predict = newPredict，為下一次的計算做準備。

GetPredict()

GetPredict()單純就是要獲得現在 thread 的 predict burst time，會直接 return Predict 回去，會在 L1 queue 裡用到，因為 L1 queue 是 preemptive approximate SJF，會由 approximate burst time 小的先執行，因此需要 call GetPredict()以獲得 predict burst time。

setBurstStart()

setBurstStart()會去設定一個 thread 剛開始 CPU burst 的時間點(BurstStart)，設定的地方是在 Scheduler::Run()的時候，要 SWITCH 之前以及 SWITCH 之後，SWITCH 之前的是幫 next thread 設，SWITCH 之後是幫 old thread 設。

GetExexTime()

GetExecTime()會 return 執行時間(AccuExecTime)，用於要印 debug 訊息時。

scheduler.cc/scheduler.h

在 scheduler.h 中，我們在 class scheduler 裡加了四個 functions 以及 private 中的

三個 list，分別代表著 L1，L2，L3 queue，以實作 multilevel feedback queue，以下介紹這四個 functions:

Removethread(List<Thread *> *Readyqueue, int level, Thread* nextThread)

這個 function 會將即將執行的 nextThread 從 queue 中 remove 掉，並將它 return 回去。主要產生這個 function 是想連動著 Debug 訊息印出，較不會在很多地方都要自己去在寫 code 印出 Debug 訊息。

InsertToQueue(List<Thread *> *Readyqueue, int level, Thread* inThread)

這個 function 會將 thread append 到 Readyqueue 裡，主要用在兩個地方，第一個是在 ReadyToRun 的時候會依據 priority 將 thread insert 到對應的 readyqueue 裡，第二個是在做完 aging 後，若再增加 priority 後 priority 超過了原先 queue 的範圍，就會從原先 queue 中 remove 掉並 insert 到更上一層的 queue。主要產生這個 function 是想連動著 Debug 訊息印出，較不會在很多地方都要自己去在寫 code 印出 Debug 訊息。

DoAgeThreeQueue()

會藉由這個函式分別 call AgeQueue(L1, 1)，AgeQueue(L2, 2)，AgeQueue(L3, 3)去做 aging。

AgeQueue(List<Thread *> *Readyqueue, int level)

在 AgeQueue 裡會對 queue 做 iteration，iteration 的方法在 List.h 裡有範例可以參考，目標是掃過 queue 裡的每個 thread，判斷該 thread 是否需要做 aging(call

HandleAgingOld())，若需要的話就會去將該 thread 的 priority 加 10，同時須判斷該 thread 是否需要晉升到更高一層的 queue(L3->L2 or L2->L1)。

Alarm.cc

在 alarm.cc 中，當 time expire 後會去呼叫到 callback，在 callback 中我們會去 call DoAgeThreeQueue()去檢查是否要做 aging，接著，若 Nachos 不是在 IdleMode，就代表還有 thread 可以執行，因此首先我們要去 get 現在 thread 的 level，看他是屬於哪一個 level 的 queue，接著去看最高層的 L1 queue 是否是空的(因為 L1 順位最高，可以 preempt 其他 level 的 Thread)，若不是空的，我們就要去呼叫 YieldOnReturn 去 Yield，去看下一個要執行的 thread 是哪個 thread(還是有可能是自己)，若 L1 是空的，再來就是要看我們現在的 thread 是否屬於 L3，若是屬於 L3，由於 L3 是 preemptive 的 round-robin，因此也需要去呼叫 YieldOnReturn 去讓出 CPU 資源，給 Scheduler 重新挑下一個執行的 Thread，若現在的 thread 不是 L3 queue 的 thread，L1 queue 也沒有其他 thread 可執行，這會有兩種可能，第一種是現在的 thread 是最後一個 L1 thread，那由於他的順位最高，那它就會繼續執行，第二種可能是現在的 thread 是 L2 thread，那它也該繼續執行，因為 L2 queue 是 non-preemptive，所以 L2 不會去打斷 L2 的 thread，因此也會繼續執行。

流程

和 MP2 類似，一開始會由 kernel::ExecAll 和 kernel::Exec 去 new 新的 thread， 設定他初始的 priority， 一切準備就緒的 thread 就會經由 Scheduler::ReadyToRun 去放進

ReadyQueue 裡，與之前不同的是，現在我們有三種層級的 queue，因此要先去 get 現在 thread 的 priority，依照它的 priority 去放進對應的 queue 裡，同時，在這裡要去設定初始時間點，之後才可以計算這個 thread 待在 queue 的時間有多久，因此我們會呼叫 UpdateAgeBaseline()去設定 AgeBaseline 的時間

```
void Thread::UpdateAgeBaseline(){
    AgeBaseline = kernel->stats->totalTicks;
}
```

之後就是等需要做 context switch 時機到，這有兩種可能，第一種是由於 L3 queue 是 round-robin，time quantum 是 100 ticks，所以當 100 ticks 時間到了後，去會去呼叫 Alarm::callback，再去呼叫 YieldOnReturn()，由於 yieldOnReturn 是 true，之後在 Interrupt::OneTick 就會去呼叫到 Yield()，在 Yield 裡，先將 interrupt 關掉，再來，將 this 傳入 ReadyToRun 去把它放進 queue 裡，原因是因為就算 time expire 要找尋下一個 thread，還是有可能選到自己，例如在 L1 queue 中，因為 L1 是 shortest job first，因此是有可能自己本身還是 shortest job，需要選到自己繼續做，因此自己還是要放進 ready queue 裡，接著我們就要去 FindNextToRun，找尋下個執行的 thread，若我們找的 nextThread 不是 null，代表有其他 thread 可以執行，就結算一下執行時間，更新 nowBurst 和 AccuExecTime，接著就可以去 Scheduler::Run context switch

```
if (nextThread != NULL)
{
    NowBurst += kernel->stats->totalTicks - BurstStart;
    AccuExecTime = NowBurst;
    kernel->scheduler->Run(nextThread, FALSE);
}
```

在 Scheduler::Run 裡，我們新增的地方是在 SWITCH(oldThread, nextThread)的上一行與下一行去設定 thread 的 BurstStart，在 switch 的上一行會去為下一個 thread 設定

BurstStart，在 switch 的下一行會去為 switch 回來的 thread 重新設定 BurstStart，給他一個初始時間，以計算 thread 的執行時間

```
nextThread->setBurstStart();
SWITCH(oldThread, nextThread);
oldThread->setBurstStart();
```

```
void Thread::setBurstStart(){
    BurstStart = kernel->stats->totalTicks;
}
```

回過頭來講 Scheduler::FindNextToRun，首先也是先把 interrupt 關掉，接著，就去 L1，L2，L3 queue 裡去找下一個要執行的 thread，我們的 if 判斷是會先從 L1 開始判斷，因為 L1 是順位最高的 queue，它可以去 preempt 其他的 queue 的 thread，因此我們用 list.h 裡的 iteration 方法去找 shortest job，我們的做法是先將 L1->Front()的 approximate burst time 用一個變數(min_appro_burst)記起來，接下來用 iterate 的方式跑過 L1 每個 thread，若有 thread burst time 小於 min_appro_burst，就將變數的值改為這個 thread，如此一來跑完 L1 後就可得到 approximate burst time 最小的 thread，若 L1 是空的話，就改去從順位第二高的 L2 queue 裡找要執行的 thread，在 L2 queue 我們要找的是 priority 最高的 thread，因此也是用類似 L1 的方法，用一個變數(high_p)去記 L2->Front()的 priority，只是這次在 iterate 的時候是要找到更大的才會更新變數的值，這樣就可以找到 priority 最高的 thread 去執行，倘若現在連 L2 queue 都是空的話，那我們就去最底層的 L3 queue 找下一個要執行的 thread，由於 L3 queue 是 round-robin，因此就直接拿 L3->Front()當下一個執行的 thread 就好了，在選完 thread 後，不管是在 L1，L2，L3 queue，都是直接 return 選到的 thread 回去，並將它從 queue 中移除，並且要結算該 thread 在 queue 中待的時間，因為之後他就要去執行了不會待在 queue 裡，如果沒有 thread 在 queue 裡的話，就 return NULL 回去

```
lowest->AddTicksInQueue();
return Removethread(L1, 1, lowest);
```

```
highest->AddTicksInQueue();
return Removethread(L2, 2, highest);
```

```
L3->Front()->AddTicksInQueue();
return Removethread(L3, 3, L3->Front());
```

第二種做 context switch 的時機是當一個 thread 要做 I/O 時，它會進到 wait state，也就是會進到 Sleep，當一個 thread sleep 時，要先結算他的執行時間，拿之前的執行時間加上(kernel ticks – 執行的初始時間)即可更新執行時間，除此之外，還需要去計算下一次的 approximate burst time，以供之後選擇 shortest job 時用到，因此我們會去呼叫上面介紹過的 CalPredictBurst()來幫我們計算下一次的 approximate burst time

```
NowBurst += kernel->stats->totalTicks - BurstStart;
CalPredictBurst();
```

接著和之前一樣，它就會去 Scheduler::FindNextToRun 去找下一個可執行的 thread，若沒有可執行的 thread 就會進入 IdleMode，若有的話就會去找到下一個該被執行的 thread，再去 Scheduler::Run 裡 context switch，在 Scheduler::Run 裡 context switch 的方法也與剛剛上面提到的一樣，以上就是我們的 implement。