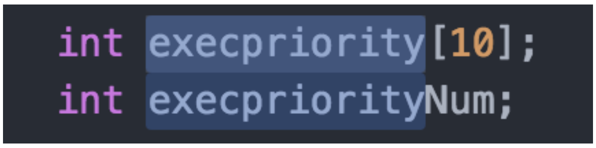
MP3 Report

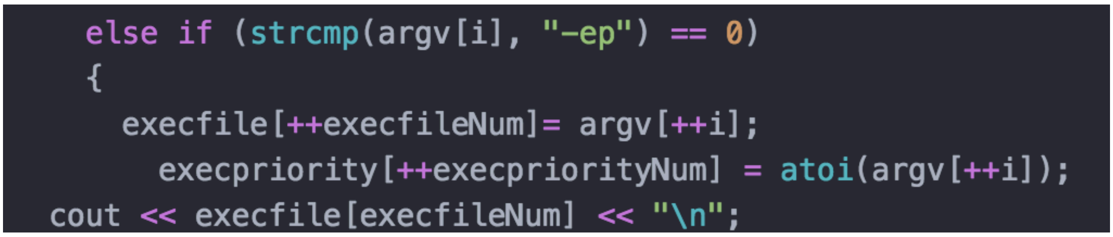
一. 接收 “-ep” 指令

首先我們在 kernel.h 中新增儲存 priority 的陣列以及 index ，如下：



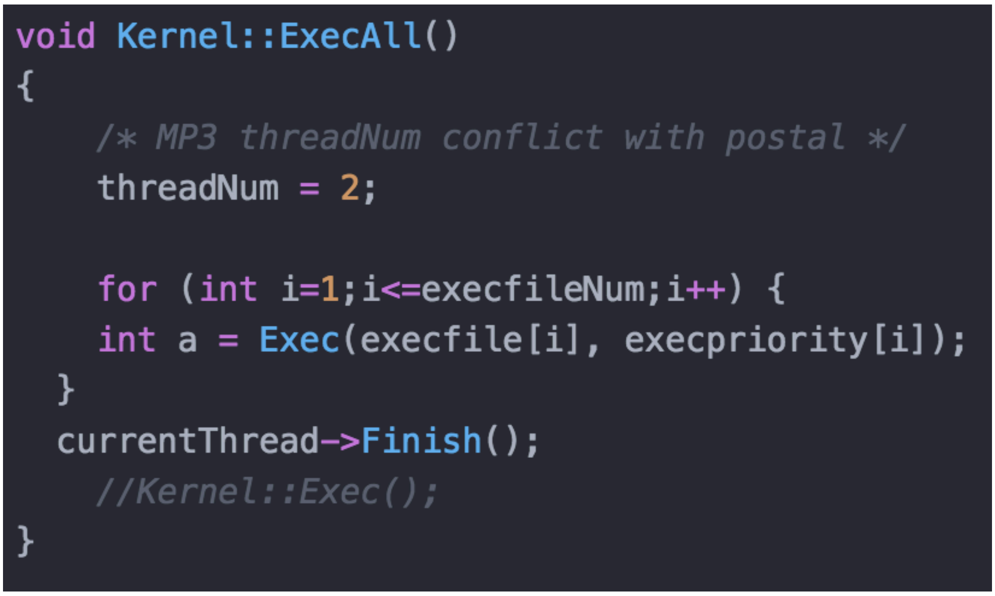
```
int execpriority[10];
int execpriorityNum;
```

並在 kernel.cc的建構式中，新增判斷 “-ep” 的 code，並且有將 argv[] 中的priority轉型成 int，如下：



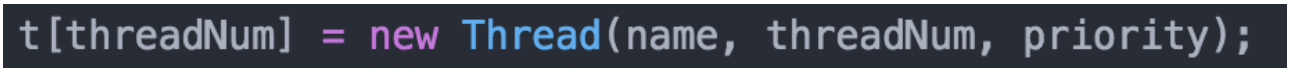
```
 else if (strcmp(argv[i], "-ep") == 0)
 {
  execfile[++execfileNum]= argv[++i];
   execpriority[++execpriorityNum] = atoi(argv[++i]);
cout << execfile[execfileNum] << "\n";
```

接下來，在 Kernel::Initialize() 裡頭除了有一剛開始產生的 main thread，還有在下頭也是在 Initialize() 裡的 postOfficeIn，他的建構式是在 network/post.cc裡可以看到他還有一個 postal worker thread ，因此我們在 kernel 的 ExecAll() 裡將 threadNum = 2 以避免和開頭的 main 和 postal 衝突，並將 Exec() 多傳入一個參數也就是剛存的 priority，如下：



```
void Kernel::ExecAll()
{
    /* MP3 threadNum conflict with postal */
    threadNum = 2;

    for (int i=1;i<=execfileNum;i++) {
    int a = Exec(execfile[i], execpriority[i]);
  }
  currentThread->Finish();
    //Kernel::Exec();
}
```

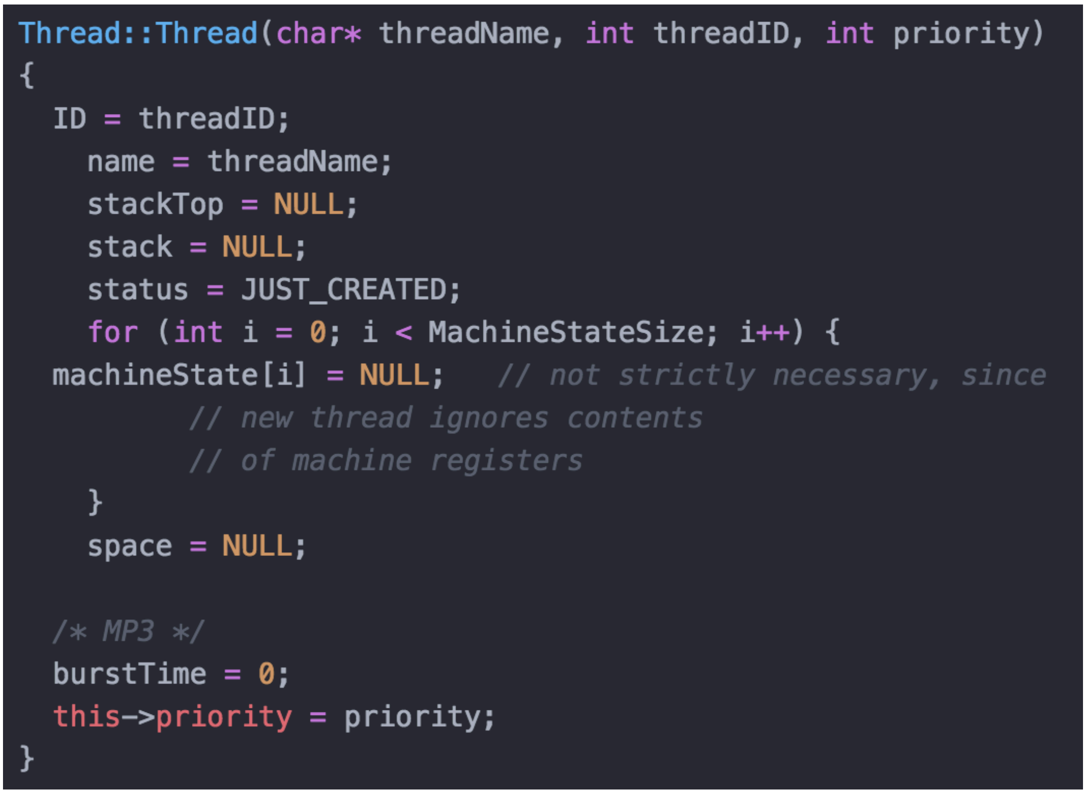
而 Exec() 裡的 new Thread 改成如下，將傳進 Exec() 的 priority 傳進 Thread 的建構式裡：



```
t[threadNum] = new Thread(name, threadNum, priority);
```

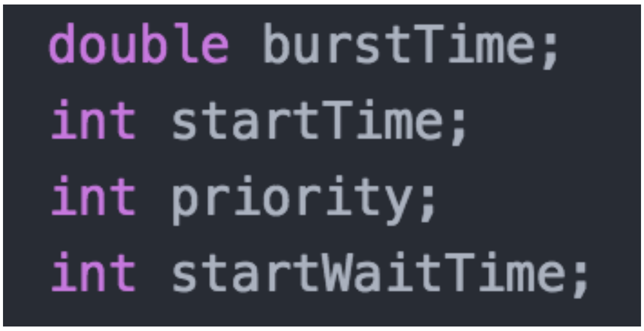
二.　Thread 的設計

這裡開始說明我們對 Thread.h 的更改，首先是建構式，我們新增了另一種多了priority 參數的建構式，如下：



```
Thread::Thread(char* threadName, int threadID, int priority)
{
  ID = threadID;
    name = threadName;
    stackTop = NULL;
    stack = NULL;
    status = JUST_CREATED;
    for (int i = 0; i < MachineStateSize; i++) {
  machineState[i] = NULL;    // not strictly necessary, since
          // new thread ignores contents
          // of machine registers
    }
    space = NULL;

  /* MP3 */
  burstTime = 0;
  this->priority = priority;
}
```

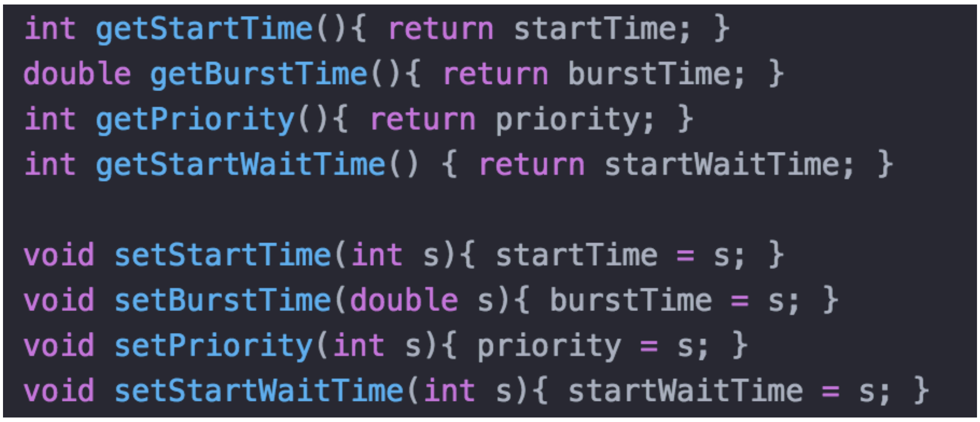
而我們將傳入的 priority 存在新增的 private 屬性資料 priority 裡，並且將 burstTime 設為 0，至於 burstTime，便是接下來要說明的，我們在 private 還新增了以下這些：



```
double burstTime;
int startTime;
int priority;
int startWaitTime;
```

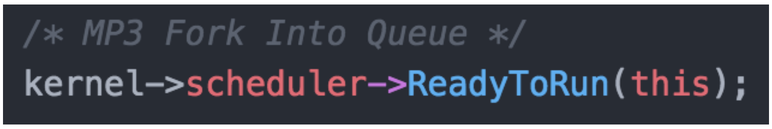
1. burstTime： 是用來儲存在實行 SJF 時，對 thread 所預估的 cpu 使用時間，而我們將所有新增的 thread 其 burstTime 初值都為 0，就如同剛上面建構式裡所設那般。
2. startTime：則是儲存 thread 在開始拿到 cpu 時的 kernel->stats->userTicks，在後面的 scheduler.cc 裡會說明使用的切確位置，而這是用來和原本預估的舊 burstTime 一起計算並更新 thread 的 burstTime。
3. startWaitTime： 則是記 thread 在被加入 ready queue 時的 kernel->stats->totalTicks，這是用來計算在 ready queue 裡等待的時間，並利用這個來計算隨著時間每過1500 ticks 就會增加 10 priority 的值。

並最後多上前面提及新增的 private 資料的 setter 和 getter，如下：



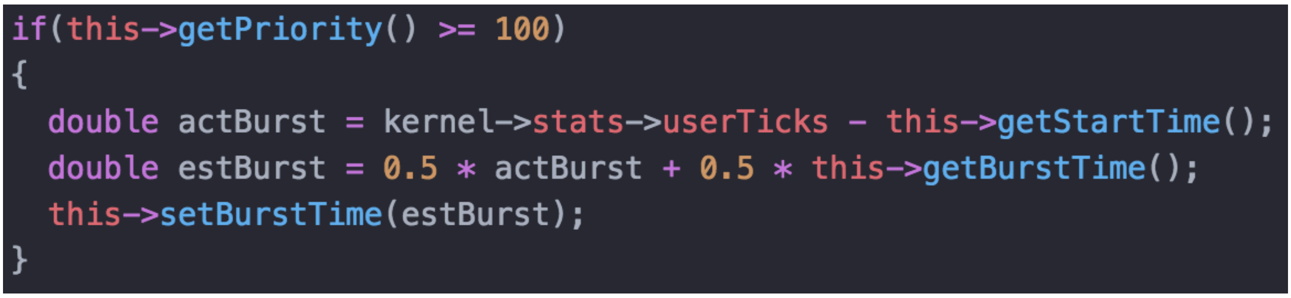
```
int getStartTime(){ return startTime; }
double getBurstTime(){ return burstTime; }
int getPriority(){ return priority; }
int getStartWaitTime() { return startWaitTime; }

void setStartTime(int s){ startTime = s; }
void setBurstTime(double s){ burstTime = s; }
void setPriority(int s){ priority = s; }
void setStartWaitTime(int s){ startWaitTime = s; }
```

在 kernel.cc 的 Exec() 裡 new 完 addrspace 後就會 Fork() thread 來讓它進 ready queue，因此在 thread.cc 中的 Fork() 裡便會呼叫 scheduler 的 ReadyToRun()，如下：



```
/* MP3 Fork Into Queue */
kernel->scheduler->ReadyToRun(this);
```

而我們也在 thread.cc 裡頭的 Yield() 和 Sleep() 裡頭加入更新 burstTime的 code，因為在 NachOS中只能透過這個兩個function，使正在執行的thread釋出CPU，換其他Thread執行。此時從CPU出來的thread需要更新burst time，計算部分如下：

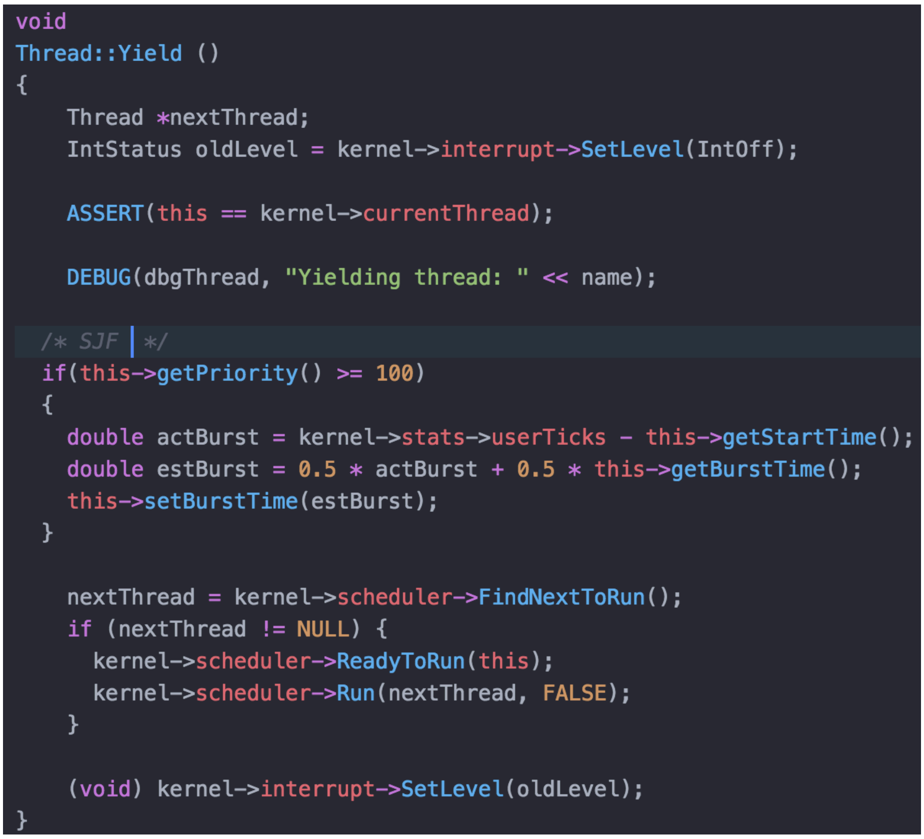


```
if(this->getPriority() >= 100)
{
  double actBurst = kernel->stats->userTicks - this->getStartTime();
  double estBurst = 0.5 * actBurst + 0.5 * this->getBurstTime();
  this->setBurstTime(estBurst);
}
```

actBurst 是剛才放掉 cpu 的 thread 實際上用了多久的 cpu，我們能從 userTicks 的差來得出，最後和原本預估的 burstTime 代入公式來計算新的 burstTime。

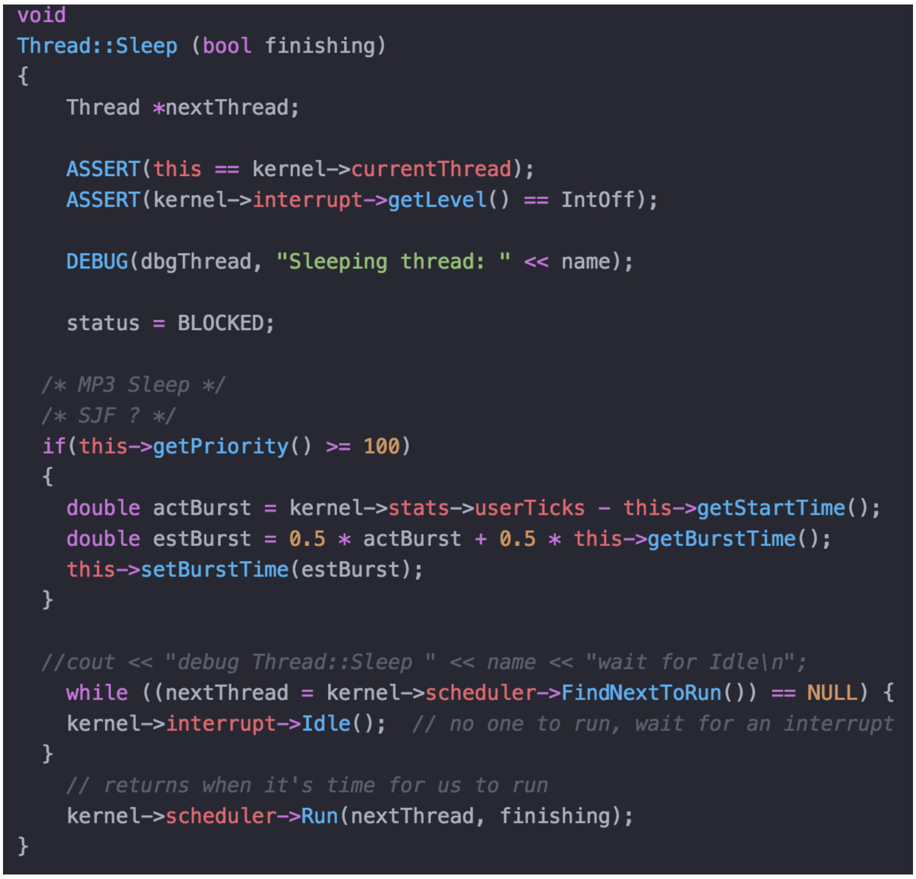
Yield()：

在 Yield() 裡會執行 setter() 更改 currentThread 的 burstTime，並且呼叫 FindNextToRun() 來尋找下一個 Thread 來執行，也就是執行 Yield() 一定會把 currentThread 換掉，至於下一個找的是哪個 Thread 來接替都有可能，如下：



```
void
Thread::Yield ()
{
    Thread *nextThread;
    IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);

    ASSERT(this == kernel->currentThread);

    DEBUG(dbgThread, "Yielding thread: " << name);

  /* SJF */
  if(this->getPriority() >= 100)
  {
    double actBurst = kernel->stats->userTicks - this->getStartTime();
    double estBurst = 0.5 * actBurst + 0.5 * this->getBurstTime();
    this->setBurstTime(estBurst);
  }

    nextThread = kernel->scheduler->FindNextToRun();
    if (nextThread != NULL) {
      kernel->scheduler->ReadyToRun(this);
      kernel->scheduler->Run(nextThread, FALSE);
    }

    (void) kernel->interrupt->SetLevel(oldLevel);
}
```

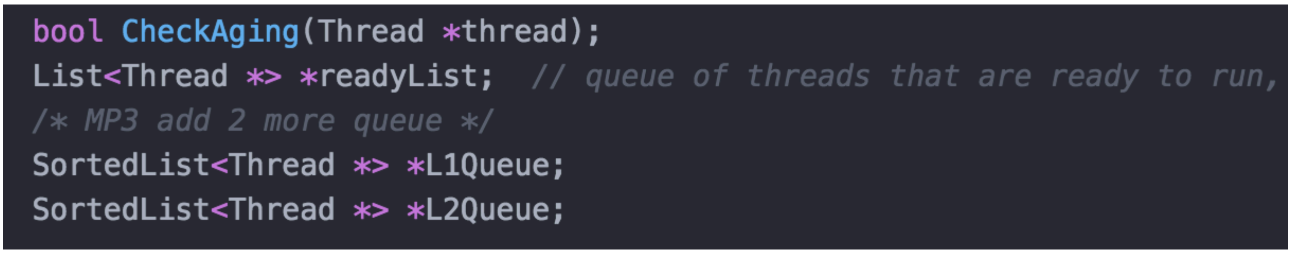
Sleep()：
這個是 Thread 做完或需要等待時呼叫的，而同樣的他放掉 cpu 我們會更改他的 burstTime，並用 scheduler->FindNextToRun() 找下一個能執行的，沒有的話就讓CPU Idle，程式如下：



```
void
Thread::Sleep (bool finishing)
{
    Thread *nextThread;

    ASSERT(this == kernel->currentThread);
    ASSERT(kernel->interrupt->getLevel() == IntOff);

    DEBUG(dbgThread, "Sleeping thread: " << name);

    status = BLOCKED;

  /* MP3 Sleep */
  /* SJF ? */
  if(this->getPriority() >= 100)
  {
    double actBurst = kernel->stats->userTicks - this->getStartTime();
    double estBurst = 0.5 * actBurst + 0.5 * this->getBurstTime();
    this->setBurstTime(estBurst);
  }

  //cout << "debug Thread::Sleep " << name << "wait for Idle\n";
    while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
    kernel->interrupt->Idle();  // no one to run, wait for an interrupt
  }
    // returns when it's time for us to run
    kernel->scheduler->Run(nextThread, finishing);
}
```

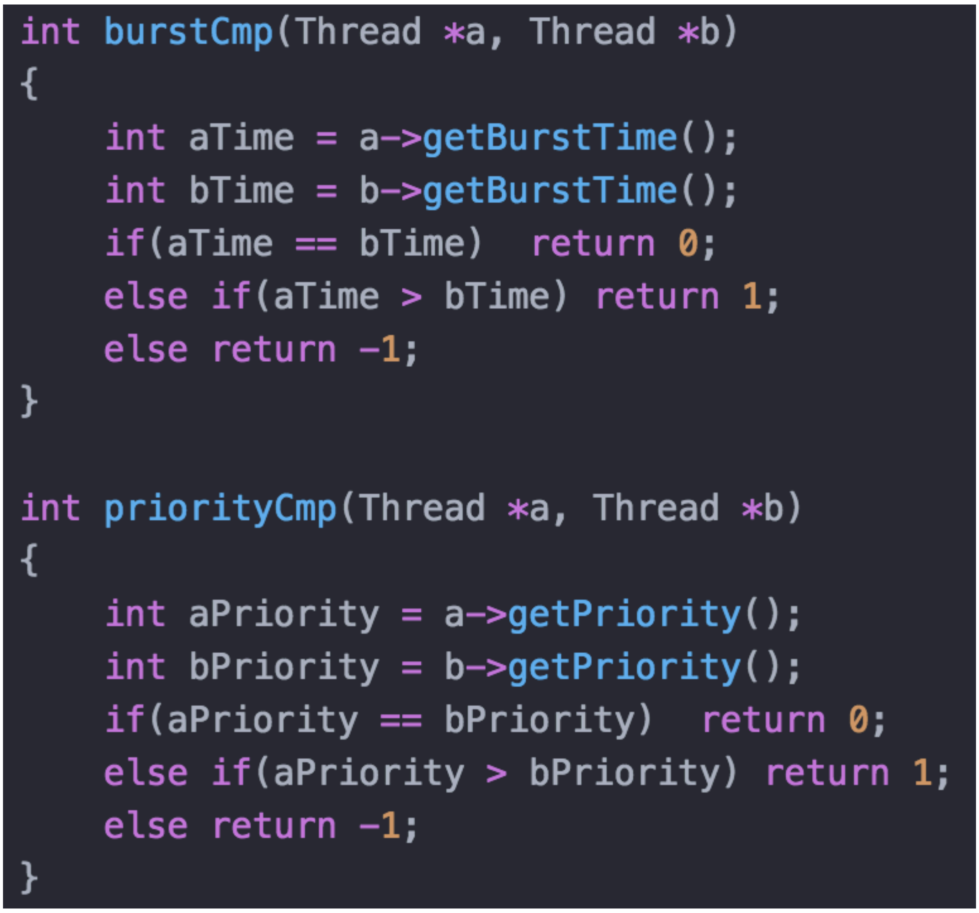
三. Scheduler 的設計

我們在 Scheduler.h 裡新增的函式及資料如下：



```
bool CheckAging(Thread *thread);
List<Thread *> *readyList;  // queue of threads that are ready to run,
/* MP3 add 2 more queue */
SortedList<Thread *> *L1Queue;
SortedList<Thread *> *L2Queue;
```

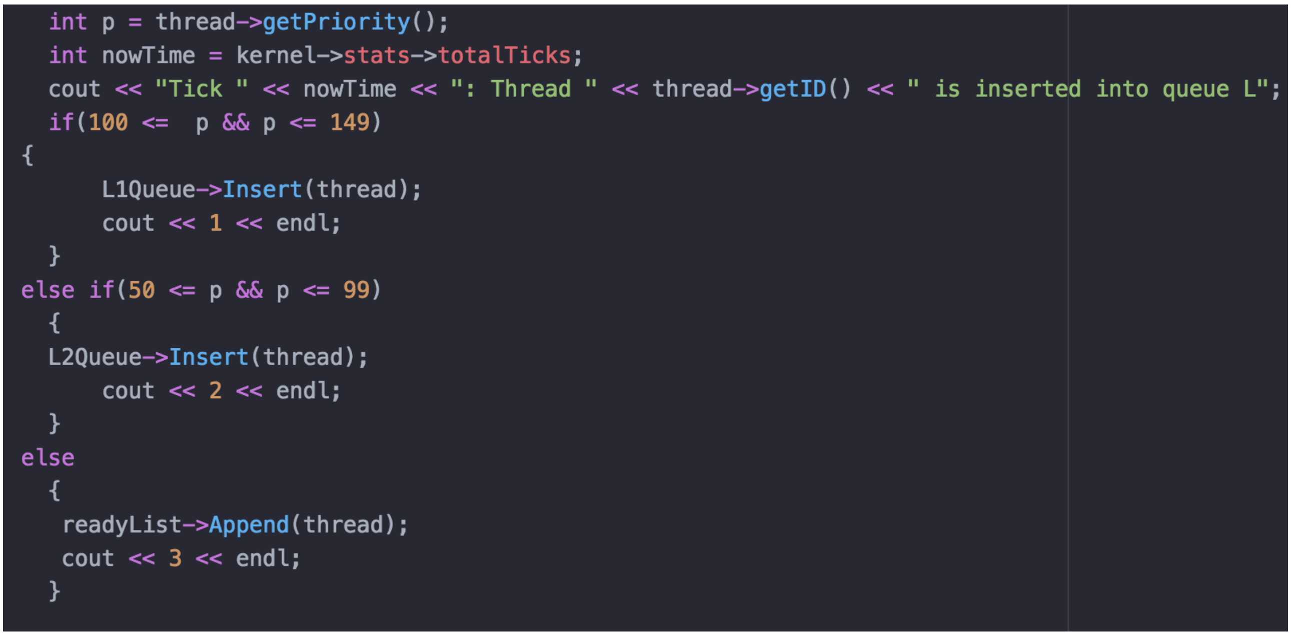
CheckAging() 是提供 Interrupt 的 OneTick() 那裡不斷檢查目前有沒有 ready queue 裡的 thread 需要增加 priority，等一下函式內容會一起在說明 Aging 的地方解釋。
而我們除了原本執行 RR 的 readyList 把它當 L3，我們又新增兩個 SortedList L1Queue 和 L2Queue， 使用NachOS提供的library，但由於在建立SortedList時需要自己提供compare function，所以我們在 scheduler 的建構式前新增這兩個 SortedList 的 compare 函式，如下：



```
int burstCmp(Thread *a, Thread *b)
{
    int aTime = a->getBurstTime();
    int bTime = b->getBurstTime();
    if(aTime == bTime)  return 0;
    else if(aTime > bTime) return 1;
    else return -1;
}

int priorityCmp(Thread *a, Thread *b)
{
    int aPriority = a->getPriority();
    int bPriority = b->getPriority();
    if(aPriority == bPriority)  return 0;
    else if(aPriority > bPriority) return 1;
    else return -1;
}
```

L1 就用 thread 的 burstTime 來比較，L2 則用 priority來比較。

1. ReadyToRun()：

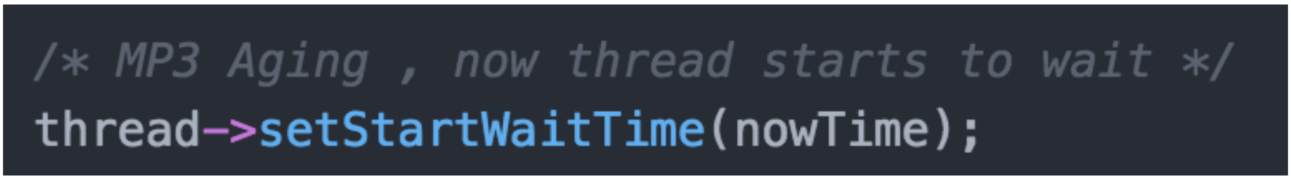


```
  int p = thread->getPriority();
  int nowTime = kernel->stats->totalTicks;
  cout << "Tick " << nowTime << ": Thread " << thread->getID() << " is inserted into queue L";
  if(100 <=  p && p <= 149)
{
      L1Queue->Insert(thread);
      cout << 1 << endl;
  }
else if(50 <= p && p <= 99)
  {
  L2Queue->Insert(thread);
      cout << 2 << endl;
  }
else
  {
   readyList->Append(thread);
   cout << 3 << endl;
  }
```

在 kernel.cc 裡 Exec() 裡會呼叫 Thread 的 Fork()，而 Fork() 裡則會呼叫 ReadyToRun()，這個函式就是把 thread 加到對應的 ready queue 裡，內容如下：

依照 Thread 的 priority 值來放到對應的 queue 裡，而這裡也會印出被加到哪個 ready queue 的訊息。而分完還會做以下的兩件事情：

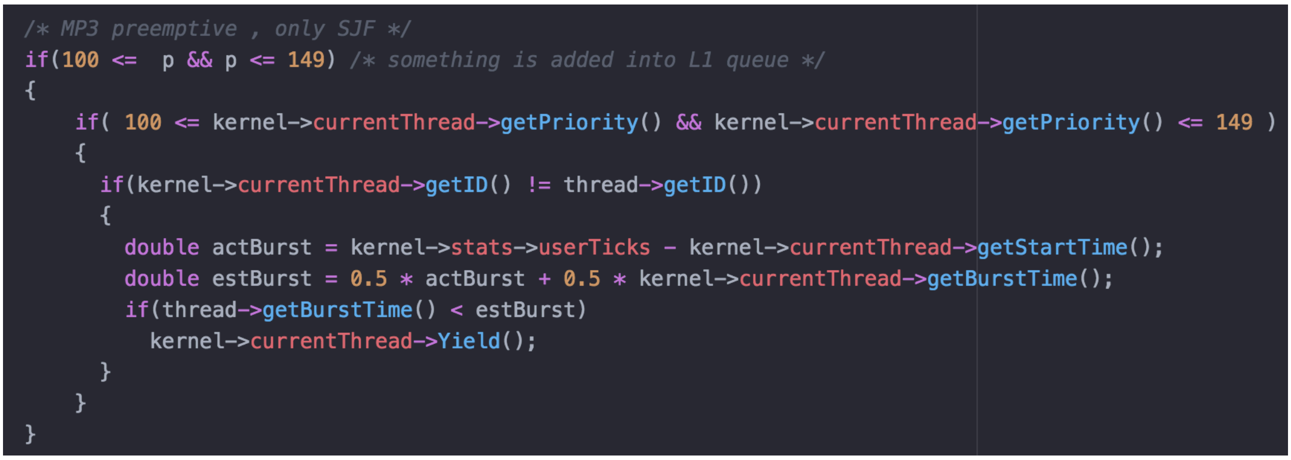
a.



```
/* MP3 Aging , now thread starts to wait */
thread->setStartWaitTime(nowTime);
```

在被加到 ready queue 時要開始計算他的等待時間，將現在的 TotalTick set 給 Thread 的 StartWaitTime。

b.

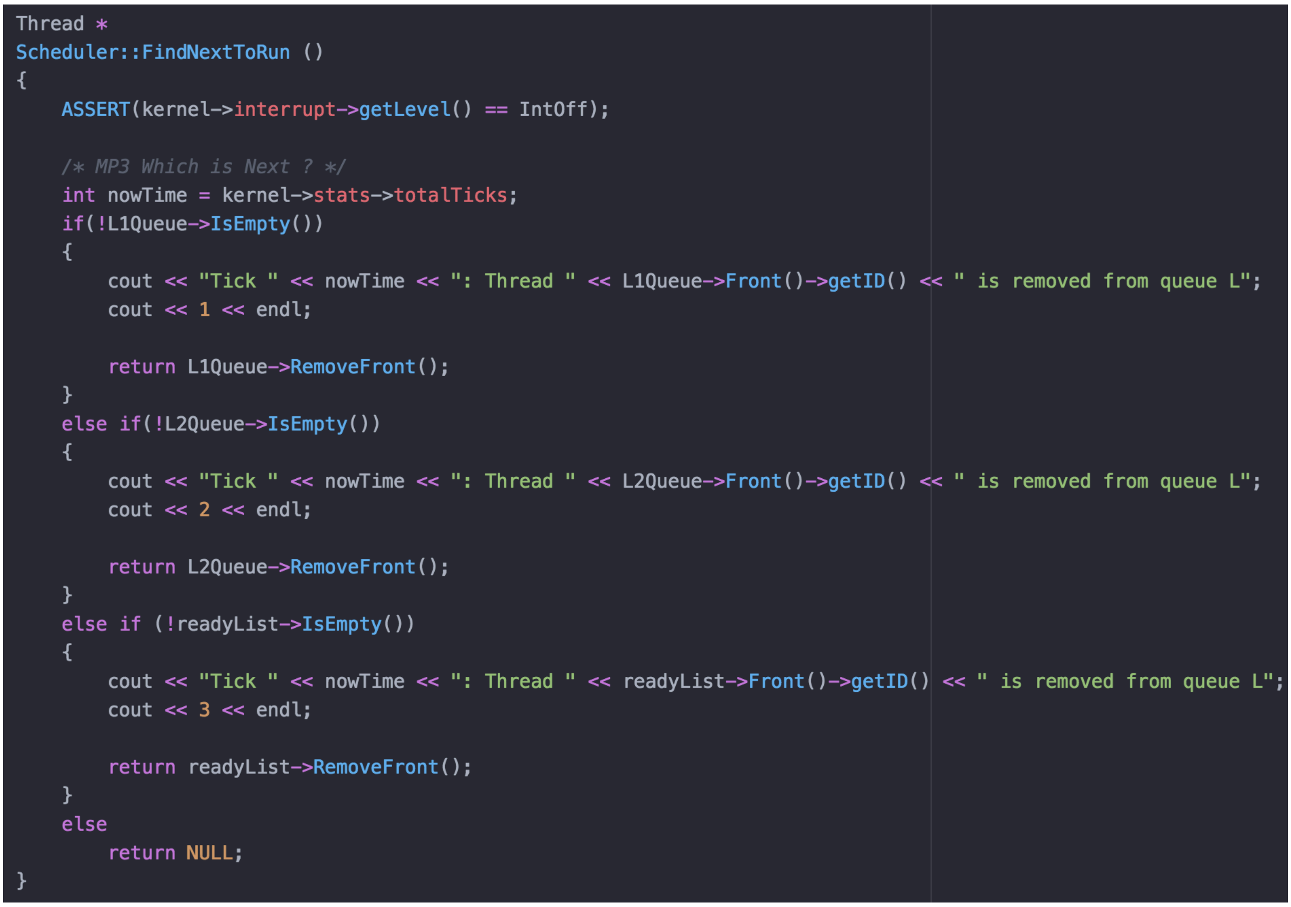


```
/* MP3 preemptive , only SJF */
if(100 <=  p && p <= 149) /* something is added into L1 queue */
{
    if( 100 <= kernel->currentThread->getPriority() && kernel->currentThread->getPriority() <= 149 )
    {
      if(kernel->currentThread->getID() != thread->getID())
      {
        double actBurst = kernel->stats->userTicks - kernel->currentThread->getStartTime();
        double estBurst = 0.5 * actBurst + 0.5 * kernel->currentThread->getBurstTime();
        if(thread->getBurstTime() < estBurst)
          kernel->currentThread->Yield();
      }
    }
}
```

L1 的 SJF preemptive發生的其中一種狀況：我們是設計在 currentThread 和新執行 readyToRun() 的 Thread 都屬於 L1 時，會判斷目前 currentThread 如果放掉 cpu 時所更新的 burstTime 跟現在新進 L1 的 Thread 誰的 burstTime 比較小，如果新進的比較小，那我們就讓 currentThread 呼叫 Yield() 以更換 Thread 來接替取得 cpu，而這邊要注意的是有一層判斷是 if(kernel->currentThread->getID() != thread->getID())，這是為了避免前面 currentThread 發生 Sleep() 時，沒有其他 Thread 去接替，而之後currentThread被喚醒並被加入 L1，加入L1時會檢查preemptive條件(加入L1的thread和kernel->currentThread比較)，但此時currentThread仍然是他自己，所以等於是跟自己比較，結果一定會變成他搶到了他自己(burst time算完以後一定會搶到)

因此kernel->currentThread就會讓給他自己，而 Yield()裡面又有readyToRun()，他又會把自己放入L1 queue中，然後又產生preemptive的檢查，造成無謂的遞迴呼叫，並反覆 insert into queue 和 remove from queue。而另外一種 preemptive 發生的狀況是在經由 Aging 從 L2 升上 L1 時，所進行的檢查有無插隊可能，會在 scheduler 的 checkAging() 裡說明。

2. FindNextToRun()：

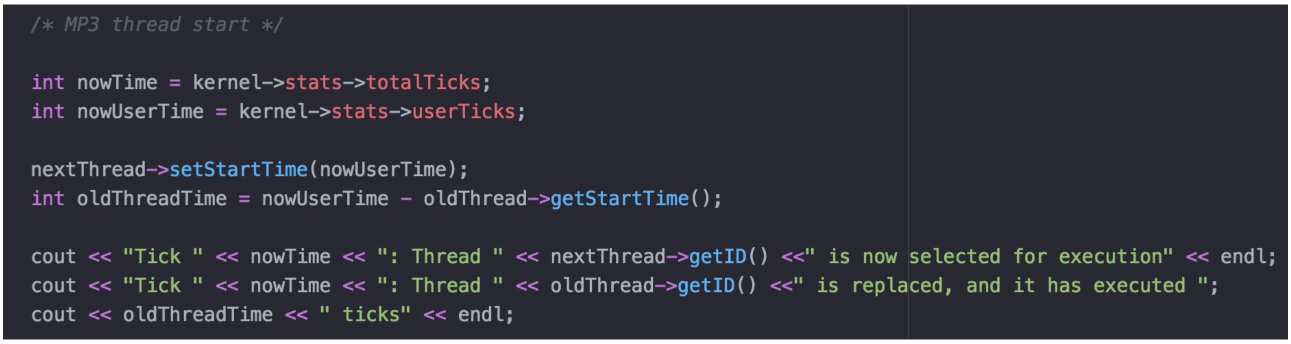


```
Thread *
Scheduler::FindNextToRun ()
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);

    /* MP3 Which is Next ? */
    int nowTime = kernel->stats->totalTicks;
    if(!L1Queue->IsEmpty())
    {
        cout << "Tick " << nowTime << ": Thread " << L1Queue->Front()->getID() << " is removed from queue L";
        cout << 1 << endl;

        return L1Queue->RemoveFront();
    }
    else if(!L2Queue->IsEmpty())
    {
        cout << "Tick " << nowTime << ": Thread " << L2Queue->Front()->getID() << " is removed from queue L";
        cout << 2 << endl;

        return L2Queue->RemoveFront();
    }
    else if (!readyList->IsEmpty())
    {
        cout << "Tick " << nowTime << ": Thread " << readyList->Front()->getID() << " is removed from queue L";
        cout << 3 << endl;

        return readyList->RemoveFront();
    }
    else
        return NULL;
}
```

在執行這個函式時，會從 L1 先檢查有沒有，沒有再一路找下去，如果都沒有 Thread 就回傳 NULL。而也是在這邊挑到的 Thread 會從 ready queue 裡拿出來，因此也在這印出離開 ready queue 的訊息。

3. Run()：

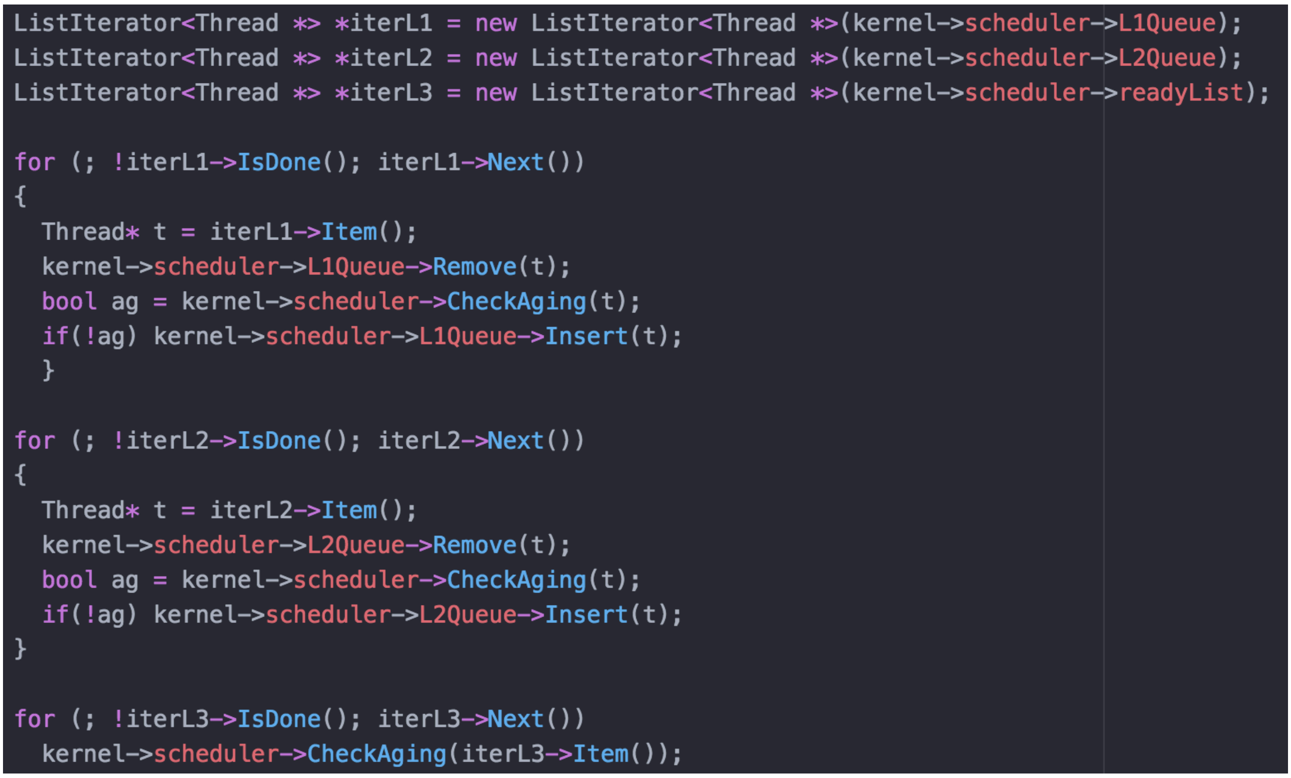
在 scheduler 的 Run() 中會執行 context switch，因此我們在這邊加了印出log 訊息的 code：



```
/* MP3 thread start */

int nowTime = kernel->stats->totalTicks;
int nowUserTime = kernel->stats->userTicks;

nextThread->setStartTime(nowUserTime);
int oldThreadTime = nowUserTime - oldThread->getStartTime();

cout << "Tick " << nowTime << ": Thread " << nextThread->getID() <<" is now selected for execution" << endl;
cout << "Tick " << nowTime << ": Thread " << oldThread->getID() <<" is replaced, and it has executed ";
cout << oldThreadTime << " ticks" << endl;
```

四. Aging 機制

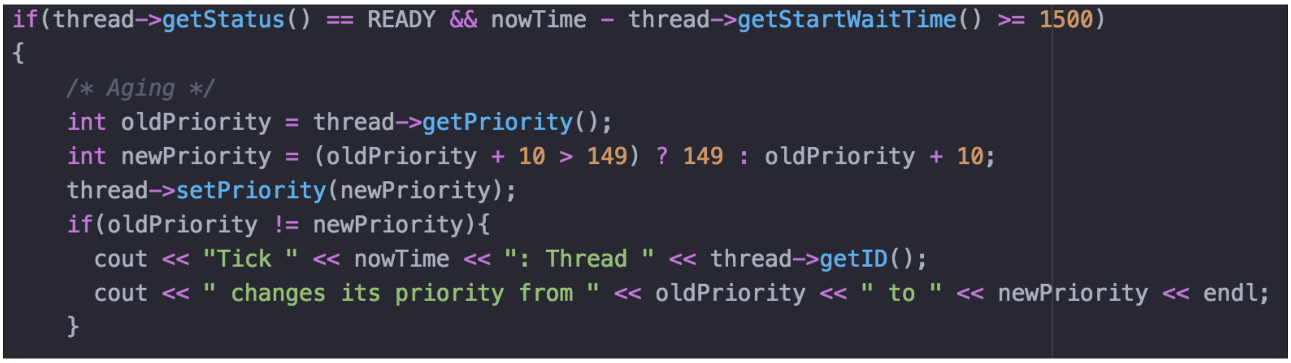
我們在 interrupt.cc 裡的 OneTick() 裡在每個 Tick 都檢查現在有沒有任何 queue 裡頭的 Thread 等待時間已經等於或超過 1500，我們會呼叫 scheduler->checkAging() 來執行這個檢查的動作，由於aging可能會改變這個thread在queue中的位置，因此必須先從這個queue中移除掉，做完更新後再插入，以確保這個queue是sorted的狀態，但L3 queue不需要sort因此不用移除再插入，OneTick() 裡頭新增的 code 如下：



```
ListIterator<Thread *> *iterL1 = new ListIterator<Thread *>(kernel->scheduler->L1Queue);
ListIterator<Thread *> *iterL2 = new ListIterator<Thread *>(kernel->scheduler->L2Queue);
ListIterator<Thread *> *iterL3 = new ListIterator<Thread *>(kernel->scheduler->readyList);

for (; !iterL1->IsDone(); iterL1->Next())
{
  Thread* t = iterL1->Item();
  kernel->scheduler->L1Queue->Remove(t);
  bool ag = kernel->scheduler->CheckAging(t);
  if(!ag) kernel->scheduler->L1Queue->Insert(t);
  }

for (; !iterL2->IsDone(); iterL2->Next())
{
  Thread* t = iterL2->Item();
  kernel->scheduler->L2Queue->Remove(t);
  bool ag = kernel->scheduler->CheckAging(t);
  if(!ag) kernel->scheduler->L2Queue->Insert(t);
}

for (; !iterL3->IsDone(); iterL3->Next())
  kernel->scheduler->CheckAging(iterL3->Item());
```

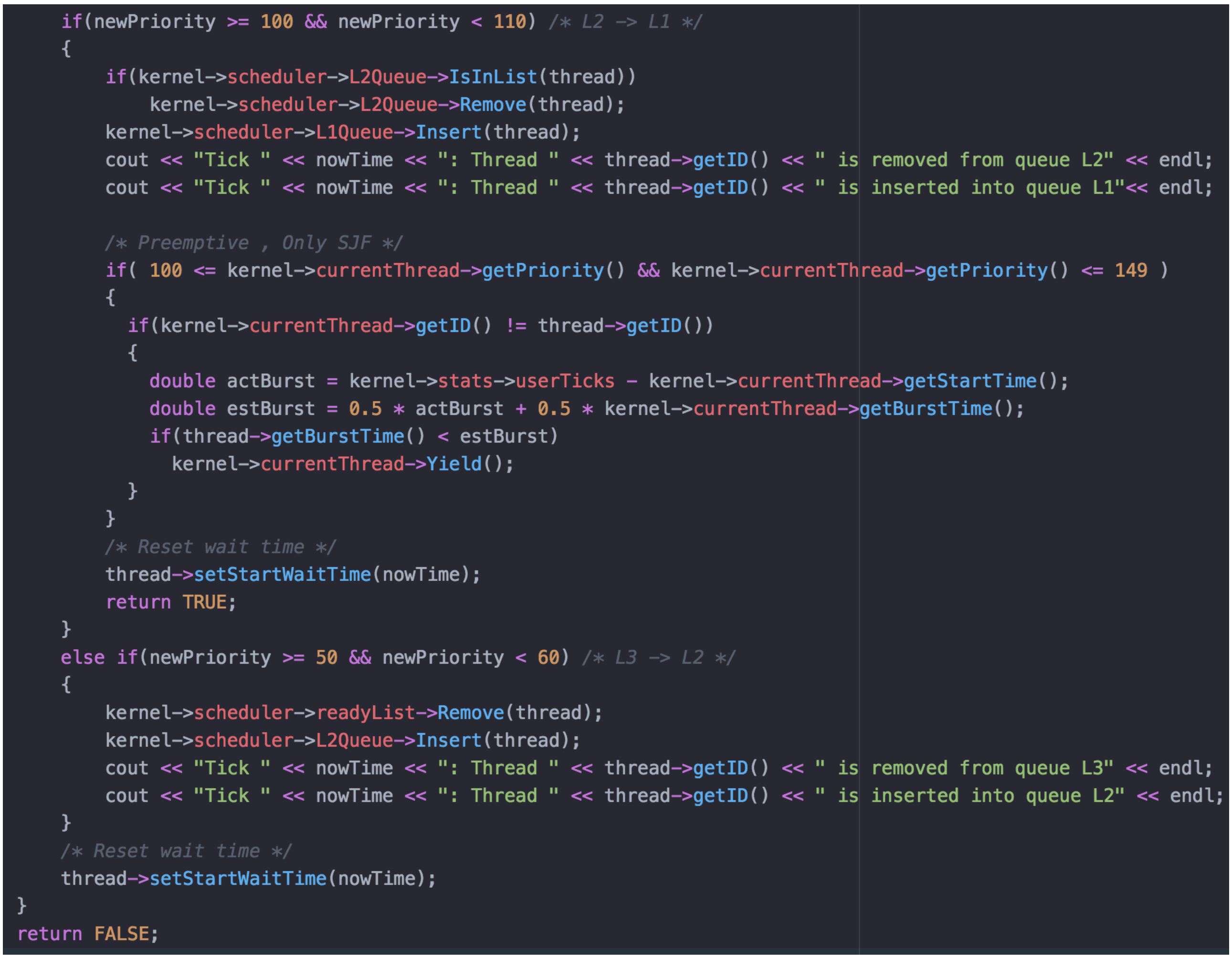
- scheduler->CheckAging：

首先先檢查這個thread確實還在ready queue中(status == READY)，並且等超過 1500 ticks 然後在 priority 不超過 149 的狀況下 增加 10 並且印出 priority 變更的訊息，如下：



```
if(thread->getStatus() == READY && nowTime - thread->getStartWaitTime() >= 1500)
{
    /* Aging */
    int oldPriority = thread->getPriority();
    int newPriority = (oldPriority + 10 > 149) ? 149 : oldPriority + 10;
    thread->setPriority(newPriority);
    if(oldPriority != newPriority){
      cout << "Tick " << nowTime << ": Thread " << thread->getID();
      cout << " changes its priority from " << oldPriority << " to " << newPriority << endl;
    }
```

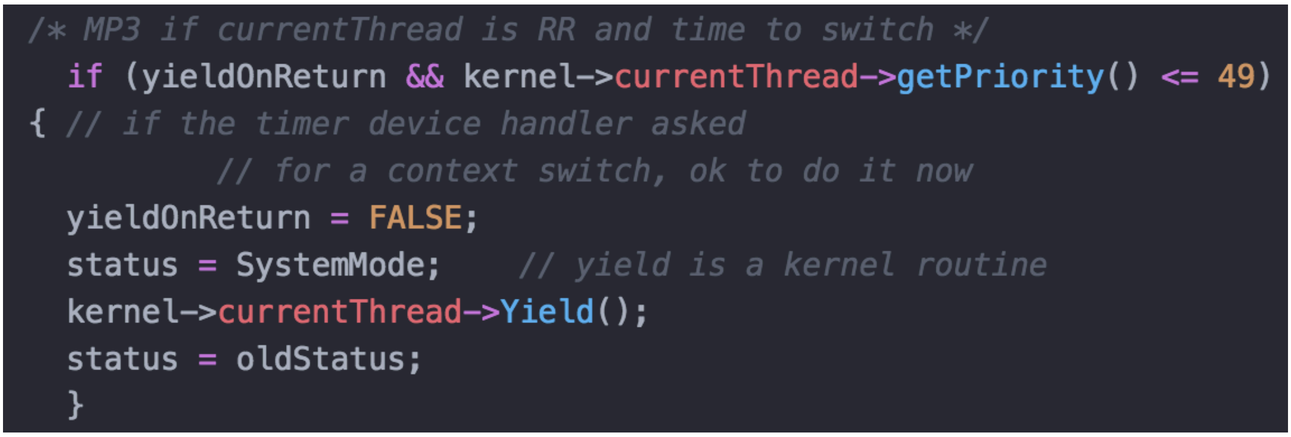
接下來檢查變更 priority 之後會不會有所處 ready queue 須改變的情況，會有兩種情況，L2->L1和L3->L2。同樣如上面readyToRun()，當有thread要進入L1時，發生preemptive檢查，我們必須要檢查他是不是就是currentThread，不然會產生自己搶自己的結果，並造成readyToRun()和Yield()不斷互相呼叫。
Aging更新完這個Thread以後，重新設定這個Thread的StartWaitTime，之後過了1500 ticks才會再次aging。程式如下：



```
    if(newPriority >= 100 && newPriority < 110) /* L2 -> L1 */
    {
        if(kernel->scheduler->L2Queue->IsInList(thread))
            kernel->scheduler->L2Queue->Remove(thread);
        kernel->scheduler->L1Queue->Insert(thread);
        cout << "Tick " << nowTime << ": Thread " << thread->getID() << " is removed from queue L2" << endl;
        cout << "Tick " << nowTime << ": Thread " << thread->getID() << " is inserted into queue L1"<< endl;

        /* Preemptive , Only SJF */
        if( 100 <= kernel->currentThread->getPriority() && kernel->currentThread->getPriority() <= 149 )
        {
          if(kernel->currentThread->getID() != thread->getID())
          {
            double actBurst = kernel->stats->userTicks - kernel->currentThread->getStartTime();
            double estBurst = 0.5 * actBurst + 0.5 * kernel->currentThread->getBurstTime();
            if(thread->getBurstTime() < estBurst)
              kernel->currentThread->Yield();
          }
        }
        /* Reset wait time */
        thread->setStartWaitTime(nowTime);
        return TRUE;
    }
    else if(newPriority >= 50 && newPriority < 60) /* L3 -> L2 */
    {
        kernel->scheduler->readyList->Remove(thread);
        kernel->scheduler->L2Queue->Insert(thread);
        cout << "Tick " << nowTime << ": Thread " << thread->getID() << " is removed from queue L3" << endl;
        cout << "Tick " << nowTime << ": Thread " << thread->getID() << " is inserted into queue L2" << endl;
    }
    /* Reset wait time */
    thread->setStartWaitTime(nowTime);
}
return FALSE;
```

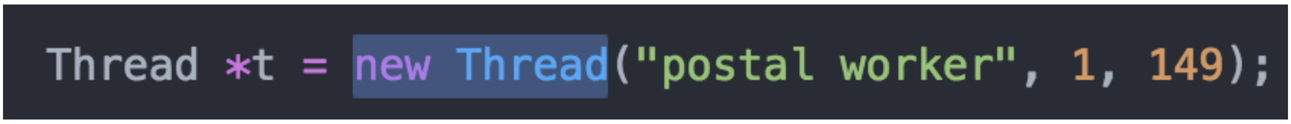
五. 補充

在 interrupt.cc 裡的 OneTick() 裡有原本執行 RR 的地方，我們將他更改一下判斷條件，讓如果目前執行的是 L3 的 Thread 才啟動 RR 的機制，更改條件如下：



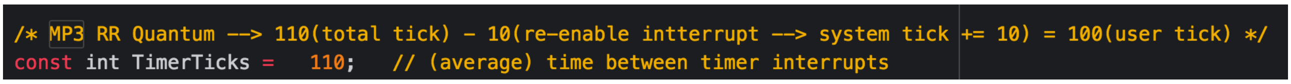
```
/* MP3 if currentThread is RR and time to switch */
   if (yieldOnReturn && kernel->currentThread->getPriority() <= 49)
{ // if the timer device handler asked
           // for a context switch, ok to do it now
   yieldOnReturn = FALSE;
   status = SystemMode;     // yield is a kernel routine
   kernel->currentThread->Yield();
   status = oldStatus;
   }
```

另外我們也將 postal 的 priority 用成最高，讓他在一剛開始 mainThread 結束接著執行並結束，如下：



```
Thread *t = new Thread("postal worker", 1, 149);
```

最後則是改變原本RR的時間：



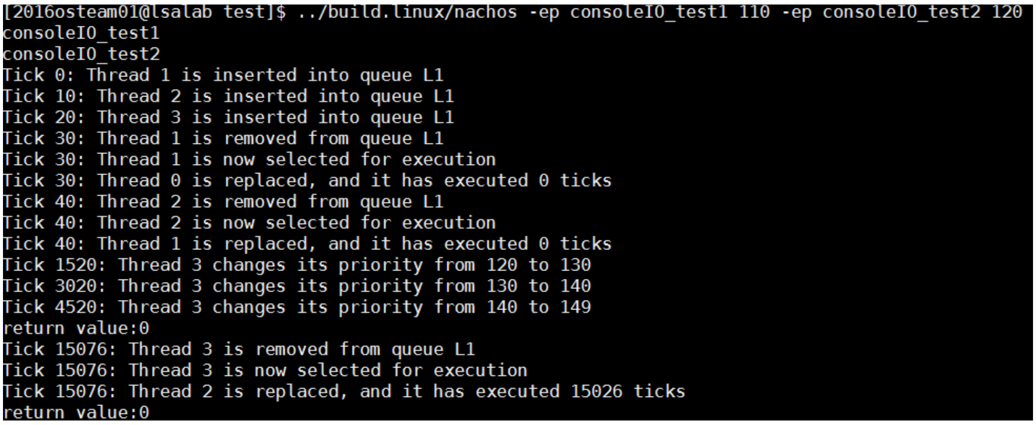
```
/* MP3 RR Quantum --> 110(total tick) - 10(re-enable intterrupt --> system tick += 10) = 100(user tick) */
const int TimerTicks =    110;    // (average) time between timer interrupts
```

註解說明一切

六. 測資結果

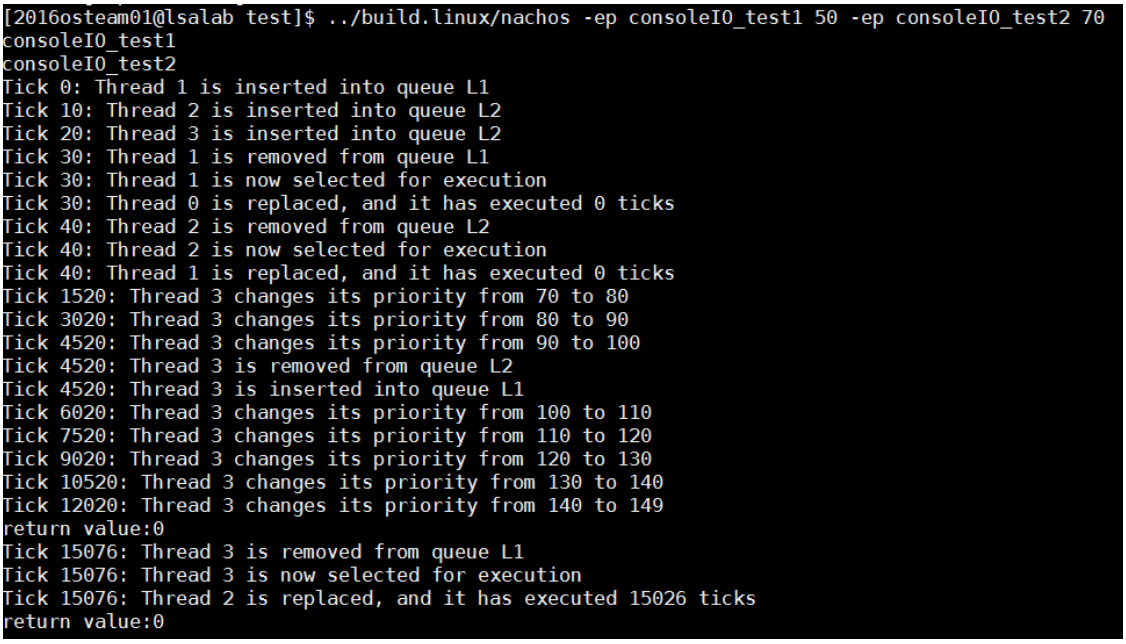
1. 首先基本的兩個程式都沒有用到 system call，也就是沒有 Interrupt：

a.Priority 110 and 120



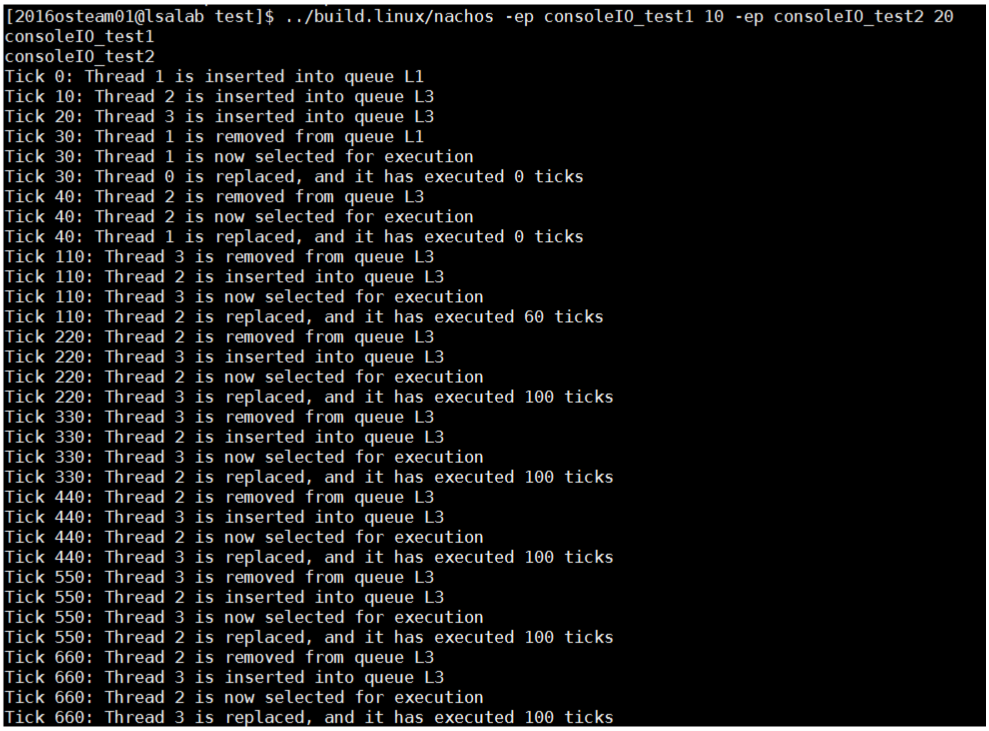
```
[2016osteam01@lsalab test]$ ../build.linux/nachos -ep consoleIO_test1 110 -ep consoleIO_test2 120
consoleIO_test1
consoleIO_test2
Tick 0: Thread 1 is inserted into queue L1
Tick 10: Thread 2 is inserted into queue L1
Tick 20: Thread 3 is inserted into queue L1
Tick 30: Thread 1 is removed from queue L1
Tick 30: Thread 1 is now selected for execution
Tick 30: Thread 0 is replaced, and it has executed 0 ticks
Tick 40: Thread 2 is removed from queue L1
Tick 40: Thread 2 is now selected for execution
Tick 40: Thread 1 is replaced, and it has executed 0 ticks
Tick 1520: Thread 3 changes its priority from 120 to 130
Tick 3020: Thread 3 changes its priority from 130 to 140
Tick 4520: Thread 3 changes its priority from 140 to 149
return value:0
Tick 15076: Thread 3 is removed from queue L1
Tick 15076: Thread 3 is now selected for execution
Tick 15076: Thread 2 is replaced, and it has executed 15026 ticks
return value:0
```

b. Priority 50 and 70



```
[2016osteam01@lsalab test]$ ../build.linux/nachos -ep consoleIO_test1 50 -ep consoleIO_test2 70
consoleIO_test1
consoleIO_test2
Tick 0: Thread 1 is inserted into queue L1
Tick 10: Thread 2 is inserted into queue L2
Tick 20: Thread 3 is inserted into queue L2
Tick 30: Thread 1 is removed from queue L1
Tick 30: Thread 1 is now selected for execution
Tick 30: Thread 0 is replaced, and it has executed 0 ticks
Tick 40: Thread 2 is removed from queue L2
Tick 40: Thread 2 is now selected for execution
Tick 40: Thread 1 is replaced, and it has executed 0 ticks
Tick 1520: Thread 3 changes its priority from 70 to 80
Tick 3020: Thread 3 changes its priority from 80 to 90
Tick 4520: Thread 3 changes its priority from 90 to 100
Tick 4520: Thread 3 is removed from queue L2
Tick 4520: Thread 3 is inserted into queue L1
Tick 6020: Thread 3 changes its priority from 100 to 110
Tick 7520: Thread 3 changes its priority from 110 to 120
Tick 9020: Thread 3 changes its priority from 120 to 130
Tick 10520: Thread 3 changes its priority from 130 to 140
Tick 12020: Thread 3 changes its priority from 140 to 149
return value:0
Tick 15076: Thread 3 is removed from queue L1
Tick 15076: Thread 3 is now selected for execution
Tick 15076: Thread 2 is replaced, and it has executed 15026 ticks
return value:0
```

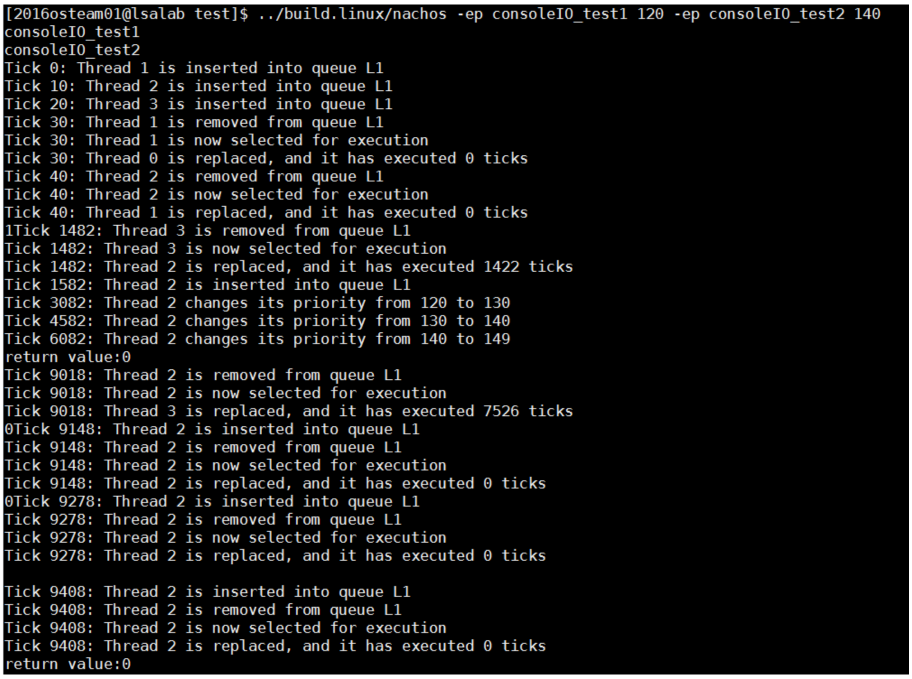
c. Priority 10 and 20



```
[2016osteam01@lsalab test]$ ../build.linux/nachos -ep consoleIO_test1 10 -ep consoleIO_test2 20
consoleIO_test1
consoleIO_test2
Tick 0: Thread 1 is inserted into queue L1
Tick 10: Thread 2 is inserted into queue L3
Tick 20: Thread 3 is inserted into queue L3
Tick 30: Thread 1 is removed from queue L1
Tick 30: Thread 1 is now selected for execution
Tick 30: Thread 0 is replaced, and it has executed 0 ticks
Tick 40: Thread 2 is removed from queue L3
Tick 40: Thread 2 is now selected for execution
Tick 40: Thread 1 is replaced, and it has executed 0 ticks
Tick 110: Thread 3 is removed from queue L3
Tick 110: Thread 2 is inserted into queue L3
Tick 110: Thread 3 is now selected for execution
Tick 110: Thread 2 is replaced, and it has executed 60 ticks
Tick 220: Thread 2 is removed from queue L3
Tick 220: Thread 3 is inserted into queue L3
Tick 220: Thread 2 is now selected for execution
Tick 220: Thread 3 is replaced, and it has executed 100 ticks
Tick 330: Thread 3 is removed from queue L3
Tick 330: Thread 2 is inserted into queue L3
Tick 330: Thread 3 is now selected for execution
Tick 330: Thread 2 is replaced, and it has executed 100 ticks
Tick 440: Thread 2 is removed from queue L3
Tick 440: Thread 3 is inserted into queue L3
Tick 440: Thread 2 is now selected for execution
Tick 440: Thread 3 is replaced, and it has executed 100 ticks
Tick 550: Thread 3 is removed from queue L3
Tick 550: Thread 2 is inserted into queue L3
Tick 550: Thread 3 is now selected for execution
Tick 550: Thread 2 is replaced, and it has executed 100 ticks
Tick 660: Thread 2 is removed from queue L3
Tick 660: Thread 3 is inserted into queue L3
Tick 660: Thread 2 is now selected for execution
Tick 660: Thread 3 is replaced, and it has executed 100 ticks
```

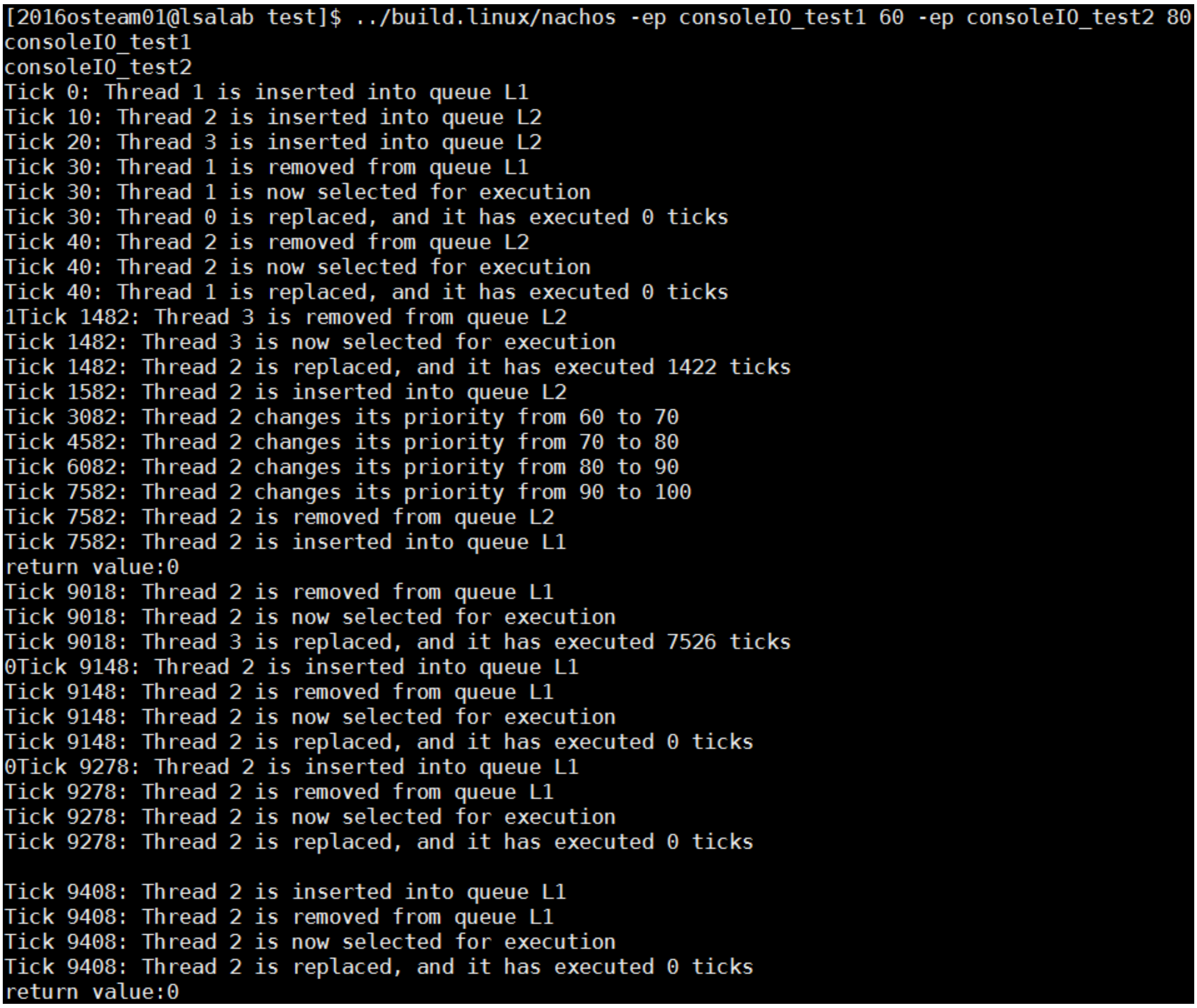
2. 再來是有一個程式有 PrintInt，也就是那個程式有 I/O 會sleep

a. Priority 120 and140



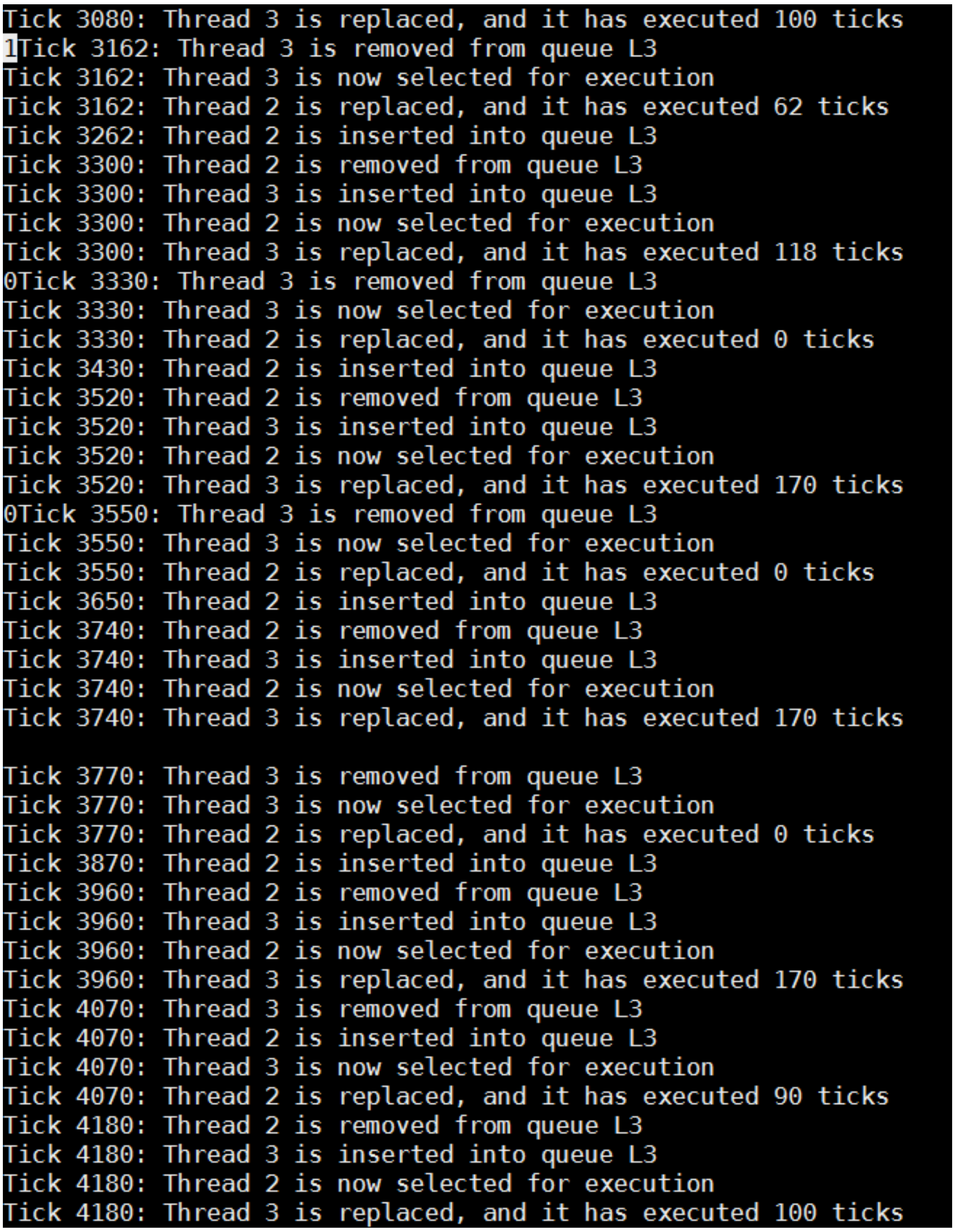
```
[2016osteam01@lsalab test]$ ../build.linux/nachos -ep consoleIO_test1 120 -ep consoleIO_test2 140
consoleIO_test1
consoleIO_test2
Tick 0: Thread 1 is inserted into queue L1
Tick 10: Thread 2 is inserted into queue L1
Tick 20: Thread 3 is inserted into queue L1
Tick 30: Thread 1 is removed from queue L1
Tick 30: Thread 1 is now selected for execution
Tick 30: Thread 0 is replaced, and it has executed 0 ticks
Tick 40: Thread 2 is removed from queue L1
Tick 40: Thread 2 is now selected for execution
Tick 40: Thread 1 is replaced, and it has executed 0 ticks
1Tick 1482: Thread 3 is removed from queue L1
Tick 1482: Thread 3 is now selected for execution
Tick 1482: Thread 2 is replaced, and it has executed 1422 ticks
Tick 1582: Thread 2 is inserted into queue L1
Tick 3082: Thread 2 changes its priority from 120 to 130
Tick 4582: Thread 2 changes its priority from 130 to 140
Tick 6082: Thread 2 changes its priority from 140 to 149
return value:0
Tick 9018: Thread 2 is removed from queue L1
Tick 9018: Thread 2 is now selected for execution
Tick 9018: Thread 3 is replaced, and it has executed 7526 ticks
0Tick 9148: Thread 2 is inserted into queue L1
Tick 9148: Thread 2 is removed from queue L1
Tick 9148: Thread 2 is now selected for execution
Tick 9148: Thread 2 is replaced, and it has executed 0 ticks
0Tick 9278: Thread 2 is inserted into queue L1
Tick 9278: Thread 2 is removed from queue L1
Tick 9278: Thread 2 is now selected for execution
Tick 9278: Thread 2 is replaced, and it has executed 0 ticks

Tick 9408: Thread 2 is inserted into queue L1
Tick 9408: Thread 2 is removed from queue L1
Tick 9408: Thread 2 is now selected for execution
Tick 9408: Thread 2 is replaced, and it has executed 0 ticks
return value:0
```

b. Priority 60 and 80



```
[2016osteam01@lsalab test]$ ../build.linux/nachos -ep consoleIO_test1 60 -ep consoleIO_test2 80
consoleIO_test1
consoleIO_test2
Tick 0: Thread 1 is inserted into queue L1
Tick 10: Thread 2 is inserted into queue L2
Tick 20: Thread 3 is inserted into queue L2
Tick 30: Thread 1 is removed from queue L1
Tick 30: Thread 1 is now selected for execution
Tick 30: Thread 0 is replaced, and it has executed 0 ticks
Tick 40: Thread 2 is removed from queue L2
Tick 40: Thread 2 is now selected for execution
Tick 40: Thread 1 is replaced, and it has executed 0 ticks
1Tick 1482: Thread 3 is removed from queue L2
Tick 1482: Thread 3 is now selected for execution
Tick 1482: Thread 2 is replaced, and it has executed 1422 ticks
Tick 1582: Thread 2 is inserted into queue L2
Tick 3082: Thread 2 changes its priority from 60 to 70
Tick 4582: Thread 2 changes its priority from 70 to 80
Tick 6082: Thread 2 changes its priority from 80 to 90
Tick 7582: Thread 2 changes its priority from 90 to 100
Tick 7582: Thread 2 is removed from queue L2
Tick 7582: Thread 2 is inserted into queue L1
return value:0
Tick 9018: Thread 2 is removed from queue L1
Tick 9018: Thread 2 is now selected for execution
Tick 9018: Thread 3 is replaced, and it has executed 7526 ticks
0Tick 9148: Thread 2 is inserted into queue L1
Tick 9148: Thread 2 is removed from queue L1
Tick 9148: Thread 2 is now selected for execution
Tick 9148: Thread 2 is replaced, and it has executed 0 ticks
0Tick 9278: Thread 2 is inserted into queue L1
Tick 9278: Thread 2 is removed from queue L1
Tick 9278: Thread 2 is now selected for execution
Tick 9278: Thread 2 is replaced, and it has executed 0 ticks

Tick 9408: Thread 2 is inserted into queue L1
Tick 9408: Thread 2 is removed from queue L1
Tick 9408: Thread 2 is now selected for execution
Tick 9408: Thread 2 is replaced, and it has executed 0 ticks
return value:0
```

c. RR



```
Tick 3080: Thread 3 is replaced, and it has executed 100 ticks
1Tick 3162: Thread 3 is removed from queue L3
Tick 3162: Thread 3 is now selected for execution
Tick 3162: Thread 2 is replaced, and it has executed 62 ticks
Tick 3262: Thread 2 is inserted into queue L3
Tick 3300: Thread 2 is removed from queue L3
Tick 3300: Thread 3 is inserted into queue L3
Tick 3300: Thread 2 is now selected for execution
Tick 3300: Thread 3 is replaced, and it has executed 118 ticks
0Tick 3330: Thread 3 is removed from queue L3
Tick 3330: Thread 3 is now selected for execution
Tick 3330: Thread 2 is replaced, and it has executed 0 ticks
Tick 3430: Thread 2 is inserted into queue L3
Tick 3520: Thread 2 is removed from queue L3
Tick 3520: Thread 3 is inserted into queue L3
Tick 3520: Thread 2 is now selected for execution
Tick 3520: Thread 3 is replaced, and it has executed 170 ticks
0Tick 3550: Thread 3 is removed from queue L3
Tick 3550: Thread 3 is now selected for execution
Tick 3550: Thread 2 is replaced, and it has executed 0 ticks
Tick 3650: Thread 2 is inserted into queue L3
Tick 3740: Thread 2 is removed from queue L3
Tick 3740: Thread 3 is inserted into queue L3
Tick 3740: Thread 2 is now selected for execution
Tick 3740: Thread 3 is replaced, and it has executed 170 ticks

Tick 3770: Thread 3 is removed from queue L3
Tick 3770: Thread 3 is now selected for execution
Tick 3770: Thread 2 is replaced, and it has executed 0 ticks
Tick 3870: Thread 2 is inserted into queue L3
Tick 3960: Thread 2 is removed from queue L3
Tick 3960: Thread 3 is inserted into queue L3
Tick 3960: Thread 2 is now selected for execution
Tick 3960: Thread 3 is replaced, and it has executed 170 ticks
Tick 4070: Thread 3 is removed from queue L3
Tick 4070: Thread 2 is inserted into queue L3
Tick 4070: Thread 3 is now selected for execution
Tick 4070: Thread 2 is replaced, and it has executed 90 ticks
Tick 4180: Thread 2 is removed from queue L3
Tick 4180: Thread 3 is inserted into queue L3
Tick 4180: Thread 2 is now selected for execution
Tick 4180: Thread 3 is replaced, and it has executed 100 ticks
```

組員貢獻：

103062121 劉亮廷：50% ， trace code and report
103062238 林子淵：50%， implement code and report