OS Homework 3 Report

Team21

葉宥忻 109062301

陳禹勳 109062134

contribution: 一起

1-1. New -> Ready

Kernel::ExecAll()

```
263  void Kernel::ExecAll()
264  {
265      for (int i=1;i<=execfileNum;i++) {
266          int a = Exec(execfile[i]);
267      }
268      currentThread->Finish();
269      //Kernel::Exec();
270  }
```

呼叫Kernel::Exec()去個別生成thread來執行kernel中所有的execfile，執行完後呼叫currentThread的Finish()，也就是結束掉main thread

Kernel::Exec(char*)

```
273  int Kernel::Exec(char* name)
274  {
275      t[threadNum] = new Thread(name, threadNum);
276      t[threadNum]->space = new AddrSpace();
277      t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);
278      threadNum++;
279
280      return threadNum-1;
```

新增一個thread給要執行的user program，再make一個新的pageTable給這個thread，接著呼叫Thread::Fork來做allocate stack，把thread放進ready queue等操作，最後回傳這個thread的thread number

Thread::Fork(VoidFunctionPtr, void*)

```
91   void
92   Thread::Fork(VoidFunctionPtr func, void *arg)
93   {
94       Interrupt *interrupt = kernel->interrupt;
95       Scheduler *scheduler = kernel->scheduler;
96       IntStatus oldLevel;
97
98       DEBUG(dbgThread, "Forking thread: " << name << " f(a): " << (int) func << " " << arg);
99       StackAllocate(func, arg);
100
101      oldLevel = interrupt->SetLevel(IntOff);
102      scheduler->ReadyToRun(this);    // ReadyToRun assumes that interrupts
103                                      // are disabled!
104      (void) interrupt->SetLevel(oldLevel);
105  }
```

先做StackAllocate()，allocate空間給要執行的thread，接下來disable interrupt，把thread放進ready queue，回復interrupt原本的狀態

Thread::StackAllocate(VoidFunctionPtr, void*)

```
306  void
307  Thread::StackAllocate (VoidFunctionPtr func, void *arg)
308  {
309      stack = (int *) AllocBoundedArray(StackSize * sizeof(int));
```

create一塊pagesize*2+size的空間，保護前後兩個page，來偵測有沒有overflow，回傳第二個 page，也就是第一個可以用的page

```
341  #ifdef x86
342      // the x86 passes the return address on the stack.  In order for SWITCH()
343      // to go to ThreadRoot when we switch to this thread, the return addres
344      // used in SWITCH() must be the starting address of ThreadRoot.
345      stackTop = stack + StackSize - 4;   // -4 to be on the safe side!
346      *(--stackTop) = (int) ThreadRoot;
347      *stack = STACK_FENCEPOST;
348  #endif
```

把stackTop移動到stack+StackSize-4的位置，然後--stackTop後指向threadRoot，當我們switch到這個process的時候，就會從Threadroot開始run

```
356  #else
357      machineState[PCState] = (void*)ThreadRoot;
358      machineState[StartupPCState] = (void*)ThreadBegin;
359      machineState[InitialPCState] = (void*)func;
360      machineState[InitialArgState] = (void*)arg;
361      machineState[WhenDonePCState] = (void*)ThreadFinish;
362  #endif
363  }
```

將ThreadRoot、ThreadBegin、func procedure、arg等pointer，存進machineState這個register，之後 要呼叫這些procedure或使用argument就從這些register找

Scheduler::ReadyToRun(Thread*)

```
56  void
57  Scheduler::ReadyToRun (Thread *thread)
58  {
59      ASSERT(kernel->interrupt->getLevel() == IntOff);
60      DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
61      //cout << "Putting thread on ready list: " << thread->getName() << endl ;
62      thread->setStatus(READY);
63      readyList->Append(thread);
64  }
```

thread的status設為READY，把thread丟進ready queue，表示他準備好執行

1-2. Running -> Ready

Machine::Run()

```
55  void
56  Machine::Run()
57  {
58      Instruction *instr = new Instruction;  // storage for decoded instruction
59      if (debug->IsEnabled('m')) {
60          cout << "Starting program in thread: " << kernel->currentThread->getName();
61      cout << ", at time: " << kernel->stats->totalTicks << "\n";
62      }
63      kernel->interrupt->setStatus(UserMode);
64      for (;;) {
65      DEBUG(dbgTraCode, "In Machine::Run(), into OneInstruction " << "== Tick " << kernel->stats->totalTicks << " ==");
66          OneInstruction(instr);
67      DEBUG(dbgTraCode, "In Machine::Run(), return from OneInstruction  " << "== Tick " << kernel->stats->totalTicks << " ==");
68
69      DEBUG(dbgTraCode, "In Machine::Run(), into OneTick " << "== Tick " << kernel->stats->totalTicks << " ==");
70      kernel->interrupt->OneTick();
71      DEBUG(dbgTraCode, "In Machine::Run(), return from OneTick " << "== Tick " << kernel->stats->totalTicks << " ==");
72      if (singleStep && (runUntilTime <= kernel->stats->totalTicks))
73          Debugger();
74      }
75  }
```

當user program開始執行的時候，準備好一個Instruction的data structure，並把status設成user mode，在for迴圈裡 OneInstruction一個一個讀並執行指令，每個指令執行完後呼叫OneTick()增加simulated time，同時檢查有沒有interrupt需要處理

Interrupt::OneTick()

```
147 void
148 Interrupt::OneTick()
149 {
150     MachineStatus oldStatus = status;
151     Statistics *stats = kernel->stats;
152
153 // advance simulated time
154     if (status == SystemMode) {
155         stats->totalTicks += SystemTick;
156     stats->systemTicks += SystemTick;
157     } else {
158     stats->totalTicks += UserTick;
159     stats->userTicks += UserTick;
160     }
161     DEBUG(dbgInt, "== Tick " << stats->totalTicks << " ==");
162
163 // check any pending interrupts are now ready to fire
164     ChangeLevel(IntOn, IntOff); // first, turn off interrupts
165                 // (interrupt handlers run with
166                 // interrupts disabled)
167     CheckIfDue(FALSE);      // check for pending interrupts
168     ChangeLevel(IntOff, IntOn); // re-enable interrupts
169     if (yieldOnReturn) {    // if the timer device handler asked
170                     // for a context switch, ok to do it now
171     yieldOnReturn = FALSE;
172     status = SystemMode;        // yield is a kernel routine
173     kernel->currentThread->Yield();
174     status = oldStatus;
175     }
176 }
```

看現在是user mode還是kernel mode增加simulated time，先disable interrupt，用 CheckIfDue()去檢查並處理interrupt，再enable interrupt回來，最後檢查yieldOnReturn是否為TRUE，若為TRUE就呼叫Yield()，做完再把yieldOnReturn設回FALSE，yieldOnReturn會根據以下的步驟被設為TRUE。

```
106     alarm = new Alarm(randomSlice); // start up time slicing
```

在kernel::Initialize會create一個alarm object

```
23  Alarm::Alarm(bool doRandom)
24  {
25      timer = new Timer(doRandom, this);
26  }
```

alarm會create一個time，並把doRandom和自己當作參數

```
38  Timer::Timer(bool doRandom, CallBackObj *toCall)
39  {
40      randomize = doRandom;
41      callPeriodically = toCall;
42      disable = FALSE;
43      SetInterrupt();
44  }
```

Time把傳進來的alarm object設為callBack object callPeriodically，doRandom決定是否random arrange interrupt，把disable設為FALSE，最後呼叫SetInterrupt()

```
69  void      You, 2 weeks ago • mp2
70  Timer::SetInterrupt()
71  {
72      if (!disable) {
73          int delay = TimerTicks;
74
75          if (randomize) {
76              delay = 1 + (RandomNumber() % (TimerTicks * 2));
77           }
78          // schedule the next timer device interrupt
79          kernel->interrupt->Schedule(this, delay, TimerInt);
80      }
81  }
82
```

接下來根據random決定delay interval，然後schedule這個timer device的interrupt，傳入timer自己和delay的時間

```
297  void
298  Interrupt::Schedule(CallBackObj *toCall, int fromNow, IntType type)
299  {
300      int when = kernel->stats->totalTicks + fromNow;
301      PendingInterrupt *toOccur = new PendingInterrupt(toCall, when, type);
302
303      DEBUG(dbgInt, "Scheduling interrupt handler the " << intTypeNames[type] << " at time = " << when);
304      ASSERT(fromNow > 0);
305
306      pending->Insert(toOccur);
307  }
```

create一個toOccur interrupt，會傳入我們前面傳的timer object(toCall)和interrupt發生的時間(when)，然後insert到PendingInterrupt中

```
321  bool
322  Interrupt::CheckIfDue(bool advanceClock)
323  {
324      PendingInterrupt *next;
325      Statistics *stats = kernel->stats;
326
327      ASSERT(level == IntOff);        // interrupts need to be disabled,
328                                      // to invoke an interrupt handler
329      if (debug->IsEnabled(dbgInt)) {
330      DumpState();
331      }
332      if (pending->IsEmpty()) {       // no pending interrupts
333      return FALSE;
334      }
335      next = pending->Front();
336
337      if (next->when > stats->totalTicks) {
338          if (!advanceClock) {        // not time yet
339              return FALSE;
340          }
341          else {              // advance the clock to next interrupt
342          stats->idleTicks += (next->when - stats->totalTicks);     You, 2 weeks ago • mp2
343          stats->totalTicks = next->when;
344          // UDelay(1000L); // rcgood - to stop nachos from spinning.
345      }
346      }
347
348      DEBUG(dbgInt, "Invoking interrupt handler for the ");
349      DEBUG(dbgInt, intTypeNames[next->type] << " at time " << next->when);
350
351      if (kernel->machine != NULL) {
352          kernel->machine->DelayedLoad(0, 0);
353      }
354
355      inHandler = TRUE;
356      do {
357          next = pending->RemoveFront();     // pull interrupt off list
358          DEBUG(dbgTraCode, "In Interrupt::CheckIfDue, into callOnInterrupt->CallBack, " << stats->totalTicks);
359          next->callOnInterrupt->CallBack();// call the interrupt handler
360          DEBUG(dbgTraCode, "In Interrupt::CheckIfDue, return from callOnInterrupt->CallBack, " << stats->totalTicks);
361      delete next;
362      } while (!pending->IsEmpty()
363              && (pending->Front()->when <= stats->totalTicks));
364      inHandler = FALSE;
365      return TRUE;
366  }
```

當OneTick或Idle呼叫CheckIfDue，而且時間到了interrupt該執行的時間時，會在359行呼叫interrupt的CallBack()，也就是timer的callback

```
52  void
53  Timer::CallBack()
54  {
55      // invoke the Nachos interrupt handler for this device
56      callPeriodically->CallBack();
57
58      SetInterrupt(); // do last, to let software interrupt handler
59                      // decide if it wants to disable future interrupts
60  }
```

呼叫callPeriodically(Alarm)的CallBack，同時再set一個新的interrupt

```
46  void
47  Alarm::CallBack()
48  {
49      Interrupt *interrupt = kernel->interrupt;
50      MachineStatus status = interrupt->getStatus();
51
52      if (status != IdleMode) {
53      interrupt->YieldOnReturn();
54      }
55  }
56
```

如果Machine的status不是IdleMode，呼叫interrupt的YieldOnReturn

```
190 Interrupt::YieldOnReturn()
191 {
192     ASSERT(inHandler == TRUE);
193     yieldOnReturn = TRUE;
194 }
```

把yieldOnReturn設為TRUE

Thread::Yield()

```
200  void
201  Thread::Yield ()
202  {
203      Thread *nextThread;
204      IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);
205
206      ASSERT(this == kernel->currentThread);
207
208      DEBUG(dbgThread, "Yielding thread: " << name);
209
210      nextThread = kernel->scheduler->FindNextToRun();
211      if (nextThread != NULL) {
212      kernel->scheduler->ReadyToRun(this);
213      kernel->scheduler->Run(nextThread, FALSE);
214      }
215      (void) kernel->interrupt->SetLevel(oldLevel);
216  }
```

當currentThread要把CPU的使用權交給其他thread，並且currentThread還沒finish，之後還要執行，所以用FindNextToRun找有沒有其他thread要run，有的話呼叫 ReadyToRun()把currentThread放回ready queue，然後呼叫scheduler run去做context switch

Scheduler::FindNextToRun()

```
74   Thread *
75   Scheduler::FindNextToRun ()
76   {
77       ASSERT(kernel->interrupt->getLevel() == IntOff);
78
79       if (readyList->IsEmpty()) {
80           return NULL;
81       } else {
82           return readyList->RemoveFront();
83       }
84   }
```

回傳下一個要做的thread，若無則回傳NULL

Scheduler::ReadyToRun(Thread*)

```
56  void
57  Scheduler::ReadyToRun (Thread *thread)
58  {
59      ASSERT(kernel->interrupt->getLevel() == IntOff);
60      DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
61      //cout << "Putting thread on ready list: " << thread->getName() << endl ;
62      thread->setStatus(READY);
63      readyList->Append(thread);
64  }
```

thread的status設為READY，把thread丟進ready queue，表示他準備好執行

Scheduler::Run(Thread*, bool)

```
103 void
104 Scheduler::Run (Thread *nextThread, bool finishing)
105 {
106     Thread *oldThread = kernel->currentThread;
107
108     ASSERT(kernel->interrupt->getLevel() == IntOff);
109
110     if (finishing) {    // mark that we need to delete current thread
111         ASSERT(toBeDestroyed == NULL);
112      toBeDestroyed = oldThread;
113     }
114
115     if (oldThread->space != NULL) { // if this thread is a user program,
116         oldThread->SaveUserState();     // save the user's CPU registers
117     oldThread->space->SaveState();
118     }
119
120     oldThread->CheckOverflow();         // check if the old thread
121                                         // had an undetected stack overflow
122
123     kernel->currentThread = nextThread;  // switch to the next thread
124     nextThread->setStatus(RUNNING);      // nextThread is now running
125
126     DEBUG(dbgThread, "Switching from: " << oldThread->getName() << " to: " << nextThread->getName());
```

如果old Thread是個user program，save他目前的state，然後check 他有沒有stack overflow，將 currentThread設為nextThread並把nextThread的status設為RUNNING

```
133     SWITCH(oldThread, nextThread);
```

呼叫SWITCH停止oldThread開始nextThread，完成switch

```
135     // we're back, running oldThread
136
137     // interrupts are off when we return from switch!
138     ASSERT(kernel->interrupt->getLevel() == IntOff);
139
140     DEBUG(dbgThread, "Now in thread: " << oldThread->getName());
141
142     CheckToBeDestroyed();       // check if thread we were running
143                                 // before this one has finished
144                                 // and needs to be cleaned up
145
146     if (oldThread->space != NULL) {     // if there is an address space
147         oldThread->RestoreUserState();     // to restore, do it.
148     oldThread->space->RestoreState();
149     }
150 }
```

當又回到了oldThread之後，CheckToBeDestroyed判斷是否刪除前一個執行的Thread，然後確認有沒有state要restore，有的話在把之前oldThread存下來的state還原回去

1-3. Running -> Waiting

SynchConsoleOutput::PutChar(char)

```
100   void
101   SynchConsoleOutput::PutChar(char ch)
102   {
103       lock->Acquire();
104       consoleOutput->PutChar(ch);
105       waitFor->P();
106       lock->Release();
107   }
```

1. lock -> Acquire(): 避免和其他thread同時output，透過lock::Acquire和semaphore::P去檢查和搶資源

2. consoleOutput -> PutChar(): 把char寫進display

3. waitFor->P(): 等待後面呼叫了CallBack()中的waitFor->V()表示output結束

4. lock->Release(): thread釋出lock

Semaphore::P()

```
75  void
76  Semaphore::P()
77  {
78      DEBUG(dbgTraCode, "In Semaphore::P(), " << kernel->stats->totalTicks);
79      Interrupt *interrupt = kernel->interrupt;
80      Thread *currentThread = kernel->currentThread;
81
82      // disable interrupts
83      IntStatus oldLevel = interrupt->SetLevel(IntOff);
84
85      while (value == 0) {        // semaphore not available
86      queue->Append(currentThread);   // so go to sleep
87      currentThread->Sleep(FALSE);
88      }
89      value--;            // semaphore available, consume its value
90
91      // re-enable interrupts
92      (void) interrupt->SetLevel(oldLevel);
93  }
```

先disable interrupt，因為85到89行確認value跟減value的動作不能被中斷，若value=0，代表有其他thread在用，所以這個thread被block住了，所以把它放進waiting queue，然後讓他sleep，若value>0，表示resource空下來了，所以用value--去搶這個resource，然後再回復interrupt的狀態。

List<T>::Append(T)

```
73  template <class T>
74  void
75  List<T>::Append(T item)
76  {
77      ListElement<T> *element = new ListElement<T>(item);
78
79      ASSERT(!IsInList(item));
80      if (IsEmpty()) {        // list is empty
81      first = element;
82      last = element;
83      } else {            // else put it after last
84      last->next = element;
85      last = element;
86      }
87      numInList++;
88      ASSERT(IsInList(item));
89  }
```

List是一個link list的結構，append會把新的item接在last的尾端，numInList++

Thread::Sleep(bool)

```
238  void
239  Thread::Sleep (bool finishing)
240  {
241      Thread *nextThread;
242
243      ASSERT(this == kernel->currentThread);
244      ASSERT(kernel->interrupt->getLevel() == IntOff);
245
246      DEBUG(dbgThread, "Sleeping thread: " << name);
247      DEBUG(dbgTraCode, "In Thread::Sleep, Sleeping thread: " << name << ", " << kernel->stats->totalTicks);
248
249      status = BLOCKED;
250      //cout << "debug Thread::Sleep " << name << "wait for Idle\n";
251      while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
252          kernel->interrupt->Idle();  // no one to run, wait for an interrupt
253      }
254      // returns when it's time for us to run
255      kernel->scheduler->Run(nextThread, finishing);
256  }
```

把thread status設為BLOCKED，如果ready queue是空的，呼叫Interrupt::Idle去檢查有沒有interrupt要處理，如果ready queue有thread在等待，把CPU從A thread交給ready queue的第一個 thread，A thread可能是因為已經做完或在等synchronization variable，若是前者，A thread之後會被刪掉；若是後者之後會把A thread重新放回ready queue，未來會叫醒A thread

Scheduler::FindNextToRun()

```
74   Thread *
75   Scheduler::FindNextToRun ()
76   {
77       ASSERT(kernel->interrupt->getLevel() == IntOff);
78
79       if (readyList->IsEmpty()) {
80           return NULL;
81       } else {
82           return readyList->RemoveFront();
83       }
84   }
```

回傳下一個要做的thread，若無則回傳NULL

Scheduler::Run(Thread*, bool)

```
103  void
104  Scheduler::Run (Thread *nextThread, bool finishing)
105  {
106      Thread *oldThread = kernel->currentThread;
107
108      ASSERT(kernel->interrupt->getLevel() == IntOff);
109
110      if (finishing) {    // mark that we need to delete current thread
111          ASSERT(toBeDestroyed == NULL);
112       toBeDestroyed = oldThread;
113      }
114
115      if (oldThread->space != NULL) { // if this thread is a user program,
116          oldThread->SaveUserState();     // save the user's CPU registers
117      oldThread->space->SaveState();
118      }
119
120      oldThread->CheckOverflow();         // check if the old thread
121                                          // had an undetected stack overflow
122
123      kernel->currentThread = nextThread;  // switch to the next thread
124      nextThread->setStatus(RUNNING);      // nextThread is now running
125
126      DEBUG(dbgThread, "Switching from: " << oldThread->getName() << " to: " << nextThread->getName());
```

如果old Thread是個user program，save他目前的state，然後check 他有沒有stack overflow，將 currentThread設為nextThread並把nextThread的status設為RUNNING

```
133      SWITCH(oldThread, nextThread);
```

呼叫SWITCH停止oldThread開始nextThread，完成switch

```
135      // we're back, running oldThread
136
137      // interrupts are off when we return from switch!
138      ASSERT(kernel->interrupt->getLevel() == IntOff);
139
140      DEBUG(dbgThread, "Now in thread: " << oldThread->getName());
141
142      CheckToBeDestroyed();       // check if thread we were running
143                                  // before this one has finished
144                                  // and needs to be cleaned up
145
146      if (oldThread->space != NULL) {     // if there is an address space
147          oldThread->RestoreUserState();     // to restore, do it.
148      oldThread->space->RestoreState();
149      }
150  }
```

當又回到了oldThread之後，CheckToBeDestroyed判斷是否刪除前一個執行的Thread，然後確認有沒有state要restore，有的話在把之前oldThread存下來的state還原回去

1-4. Waiting -> Ready

Semaphore::V()

```
103  void
104  Semaphore::V()
105  {
106      DEBUG(dbgTraCode, "In Semaphore::V(), " << kernel->stats->totalTicks);
107      Interrupt *interrupt = kernel->interrupt;
108
109      // disable interrupts
110      IntStatus oldLevel = interrupt->SetLevel(IntOff);
111
112      if (!queue->IsEmpty()) {  // make thread ready.
113      kernel->scheduler->ReadyToRun(queue->RemoveFront());
114      }
115      value++;
116
117      // re-enable interrupts
118      (void) interrupt->SetLevel(oldLevel);
119  }
```

先disable interrupt，因為85到89行確認value跟加value的動作不能被中斷，檢查waiting queue有沒有waiter，有的話把waiter放進ready queue，增加value來釋出資源，然後回復interrupt狀態

Scheduler::ReadyToRun(Thread*)

```
56   void
57   Scheduler::ReadyToRun (Thread *thread)
58   {
59       ASSERT(kernel->interrupt->getLevel() == IntOff);
60       DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
61       //cout << "Putting thread on ready list: " << thread->getName() << endl ;
62       thread->setStatus(READY);
63       readyList->Append(thread);
64   }
```

thread的status設為READY，把thread丟進ready queue，表示他準備好執行

1-5. Running -> Terminated

ExceptionHandler(ExceptionType) case SC_Exit

```
188                 case SC_Exit:
189                     DEBUG(dbgAddr, "Program exit\n");
190                             val=kernel→machine→ReadRegister(4);
191                             cout << "return value:" << val << endl;
192                     kernel→currentThread→Finish();
193                     break;
```

CPU收到SC_exit的exception type，故結束currentThread

Thread::Finish()

```
170 void
171 Thread::Finish ()
172 {
173     (void) kernel→interrupt→SetLevel(IntOff);
174     ASSERT(this == kernel→currentThread);
175
176     DEBUG(dbgThread, "Finishing thread: " << name);
177     Sleep(TRUE);                // invokes SWITCH
178     // not reached
179 }
```

當一個thread做完就會呼叫，先Disable interrupt，實作方式是呼叫Sleep裡面傳TRUE，讓currentThread Sleep，因為現在還沒完成switch，所以還不能刪除這個thread，Sleep的參數TRUE代表這個Thread已經結束了，之後才會在Scheduler::CheckToBeDestroyed將這個Thread刪除

Thread::Sleep(bool)

```
238 void
239 Thread::Sleep (bool finishing)
240 {
241     Thread *nextThread;
242
243     ASSERT(this == kernel->currentThread);
244     ASSERT(kernel->interrupt->getLevel() == IntOff);
245
246     DEBUG(dbgThread, "Sleeping thread: " << name);
247     DEBUG(dbgTraCode, "In Thread::Sleep, Sleeping thread: " << name << ", " << kernel->stats->totalTicks);
248
249     status = BLOCKED;
250     //cout << "debug Thread::Sleep " << name << "wait for Idle\n";
251     while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
252         kernel->interrupt->Idle();  // no one to run, wait for an interrupt
253     }
254     // returns when it's time for us to run
255     kernel->scheduler->Run(nextThread, finishing);
256 }
```

把thread status設為BLOCKED，如果ready queue是空的，呼叫Interrupt::Idle去檢查有沒有interrupt要處理，如果ready queue有thread在等待，把CPU從A thread交給ready queue的第一個 thread，A thread可能是因為已經做完或在等synchronization variable，若是前者，A thread之後會被刪掉；若是後者之後會把A thread重新放回ready queue，未來會叫醒A thread

Scheduler::FindNextToRun()

```
74  Thread *
75  Scheduler::FindNextToRun ()
76  {
77      ASSERT(kernel->interrupt->getLevel() == IntOff);
78
79      if (readyList->IsEmpty()) {
80          return NULL;
81      } else {
82          return readyList->RemoveFront();
83      }
84  }
```

回傳下一個要做的thread，若無則回傳NULL

Scheduler::Run(Thread*, bool)

```
103 void
104 Scheduler::Run (Thread *nextThread, bool finishing)
105 {
106     Thread *oldThread = kernel->currentThread;
107 
108     ASSERT(kernel->interrupt->getLevel() == IntOff);
109 
110     if (finishing) {    // mark that we need to delete current thread
111         ASSERT(toBeDestroyed == NULL);
112      toBeDestroyed = oldThread;
113     }
114 
115     if (oldThread->space != NULL) { // if this thread is a user program,
116         oldThread->SaveUserState();     // save the user's CPU registers
117     oldThread->space->SaveState();
118     }
119 
120     oldThread->CheckOverflow();         // check if the old thread
121                                         // had an undetected stack overflow
122 
123     kernel->currentThread = nextThread;  // switch to the next thread
124     nextThread->setStatus(RUNNING);      // nextThread is now running
125 
126     DEBUG(dbgThread, "Switching from: " << oldThread->getName() << " to: " << nextThread->getName());
```

如果old Thread是個user program，save他目前的state，然後check 他有沒有stack overflow，將 currentThread設為nextThread並把nextThread的status設為RUNNING

```
133     SWITCH(oldThread, nextThread);
```

呼叫SWITCH停止oldThread開始nextThread，完成switch

```
135     // we're back, running oldThread
136 
137     // interrupts are off when we return from switch!
138     ASSERT(kernel->interrupt->getLevel() == IntOff);
139 
140     DEBUG(dbgThread, "Now in thread: " << oldThread->getName());
141 
142     CheckToBeDestroyed();       // check if thread we were running
143                                 // before this one has finished
144                                 // and needs to be cleaned up
145 
146     if (oldThread->space != NULL) {     // if there is an address space
147         oldThread->RestoreUserState();     // to restore, do it.
148     oldThread->space->RestoreState();
149     }
150 }
```

當又回到了oldThread之後，CheckToBeDestroyed判斷是否刪除前一個執行的Thread，然後確認有沒有state要restore，有的話在把之前oldThread存下來的state還原回去

1-6. Ready -> Running

Scheduler::FindNextToRun()

```
74   Thread *
75   Scheduler::FindNextToRun ()
76   {
77       ASSERT(kernel->interrupt->getLevel() == IntOff);
78
79       if (readyList->IsEmpty()) {
80           return NULL;
81       } else {
82           return readyList->RemoveFront();
83       }
84   }
```

回傳下一個要做的thread，若無則回傳NULL

Scheduler::Run(Thread*, bool)

```
103  void
104  Scheduler::Run (Thread *nextThread, bool finishing)
105  {
106      Thread *oldThread = kernel->currentThread;
107
108      ASSERT(kernel->interrupt->getLevel() == IntOff);
109
110      if (finishing) {    // mark that we need to delete current thread
111          ASSERT(toBeDestroyed == NULL);
112       toBeDestroyed = oldThread;
113      }
114
115      if (oldThread->space != NULL) { // if this thread is a user program,
116          oldThread->SaveUserState();     // save the user's CPU registers
117      oldThread->space->SaveState();
118      }
119
120      oldThread->CheckOverflow();         // check if the old thread
121                                          // had an undetected stack overflow
122
123      kernel->currentThread = nextThread;  // switch to the next thread
124      nextThread->setStatus(RUNNING);      // nextThread is now running
125
126      DEBUG(dbgThread, "Switching from: " << oldThread->getName() << " to: " << nextThread->getName());
```

如果old Thread是個user program，save他目前的state，然後check 他有沒有stack overflow，將 currentThread設為nextThread並把nextThread的status設為RUNNING

```
133         SWITCH(oldThread, nextThread);
```

呼叫SWITCH停止oldThread開始nextThread，完成switch

```
135     // we're back, running oldThread
136
137     // interrupts are off when we return from switch!
138     ASSERT(kernel->interrupt->getLevel() == IntOff);
139
140     DEBUG(dbgThread, "Now in thread: " << oldThread->getName());
141
142     CheckToBeDestroyed();       // check if thread we were running
143                                 // before this one has finished
144                                 // and needs to be cleaned up
145
146     if (oldThread->space != NULL) {     // if there is an address space
147         oldThread->RestoreUserState();     // to restore, do it.
148     oldThread->space->RestoreState();
149     }
150 }
```

當又回到了oldThread之後，CheckToBeDestroyed判斷是否刪除前一個執行的Thread，然後確認有沒有state要restore，有的話在把之前oldThread存下來的state還原回去

SWITCH(Thread*, Thread*)

```
329   ** on entry, stack looks like this:
330   **      8(esp)  ->              thread *t2
331   **      4(esp)  ->              thread *t1
332   **       (esp)  ->              return address
```

```
342  _SWITCH:
343  SWITCH:
344          movl    %eax,_eax_save          # save the value of eax
345          movl    4(%esp),%eax            # move pointer to t1 into eax
346          movl    %ebx,_EBX(%eax)         # save registers
347          movl    %ecx,_ECX(%eax)
348          movl    %edx,_EDX(%eax)
349          movl    %esi,_ESI(%eax)
350          movl    %edi,_EDI(%eax)
351          movl    %ebp,_EBP(%eax)
352          movl    %esp,_ESP(%eax)         # save stack pointer
353          movl    _eax_save,%ebx          # get the saved value of eax
354          movl    %ebx,_EAX(%eax)         # store it
355          movl    0(%esp),%ebx            # get return address from stack into ebx
356          movl    %ebx,_PC(%eax)          # save it into the pc storage
357
358          movl    8(%esp),%eax            # move pointer to t2 into eax
359
360          movl    _EAX(%eax),%ebx         # get new value for eax into ebx
361          movl    %ebx,_eax_save          # save it
362          movl    _EBX(%eax),%ebx         # retore old registers
363          movl    _ECX(%eax),%ecx
364          movl    _EDX(%eax),%edx
365          movl    _ESI(%eax),%esi
366          movl    _EDI(%eax),%edi
367          movl    _EBP(%eax),%ebp
368          movl    _ESP(%eax),%esp         # restore stack pointer
369          movl    _PC(%eax),%eax          # restore return address into eax
370          movl    %eax,4(%esp)            # copy over the ret address on the stack
371          movl    _eax_save,%eax
372
373          ret
374
375  #endif // x86
```

先把%eax暫存到_eax_save，然後把t1 pointer存到%eax，346-352行把register存到t1的stack中，353-354行把%eax的restore後也存進t1的stack，355-356行把return address存進pc storage中。

358行把t2 pointer存到%eax。

360-361行把t2的%eax放到%ebx後暫存到_eax_save，362-369行從t2的stack restore register的值。

370把t2 return address存到%esp+4

371 restore t2 %eax的值。

373 ret 會回到%esp，下一個執行的指令是esp+4

depends on the previous process state，[New,Running,Waiting]→Ready

```
/* void ThreadRoot( void )
**
** expects the following registers to be initialized:
**       eax     points to startup function (interrupt enable)
**       edx     contains inital argument to thread function
**       esi     points to thread function
**       edi     point to Thread::Finish()
*/
```

New:會從Thread root開始做

```
for (;;) {
DEBUG(dbgTraCode, "In Machine::Run(), into OneInstruction " << "== Tick " << kernel->stats->totalTicks << " ==");
    OneInstruction(instr);
DEBUG(dbgTraCode, "In Machine::Run(), return from OneInstruction  " << "== Tick " << kernel->stats->totalTicks << " ==");

DEBUG(dbgTraCode, "In Machine::Run(), into OneTick " << "== Tick " << kernel->stats->totalTicks << " ==");
kernel->interrupt->OneTick();
DEBUG(dbgTraCode, "In Machine::Run(), return from OneTick " << "== Tick " << kernel->stats->totalTicks << " ==");
if (singleStep && (runUntilTime <= kernel->stats->totalTicks))
    Debugger();
}
```

Running:代表t2原本是在跑Machine::Run的OneTick，因為yield被放進了ready queue，context switch回來後會從之前的program counter的instruction繼續run

```
while (value == 0) {        // semaphore not available
queue->Append(currentThread);    // so go to sleep
currentThread->Sleep(FALSE);
}
value--;                    // semaphore available, consume its value
```

Waiting:t2原本是在Semaphore::P等待資源而被放進了waiting queue，接下來又因為資源可以用了而被放進ready queue，所以context switch回來後，會從Semaphore::P的while繼續run

Implementation

debug.h

```
33  // MP3
34  const char z = 'z';
35
```

新增一個debug flag ‘z’

stats.h

```
57  // MP3
58  // const int ConsoleTime =   100; // time to read or write one character
59  const int ConsoleTime = 1;
60
```

kernel.cc

```
122  #endif // FILESYS_STUB
123      // postOfficeIn = new PostOfficeInput(10);
124      // postOfficeOut = new PostOfficeOutput(reliability);
```

```
145      // delete postOfficeIn;
146      // delete postOfficeOut;
```

```
80
81          // MP3
82          else if (strcmp(argv[i], "-ep") == 0){
83              execfile[++execfileNum]= argv[++i];
84              priorities[execfileNum] = atoi(argv[++i]);
85          }
```

把要執行的檔案放進execfile中，並給它priority

```
269 void Kernel::ExecAll()
270 {
271     for (int i=1;i<=execfileNum;i++) {
272         // int a = Exec(execfile[i]);
273         You, 21 minutes ago • Uncommitted changes
274         // MP3
275         int a = Exec(execfile[i], priorities[i]);
276     }
277     currentThread->Finish();
278     //Kernel::Exec();
279 }
```

```
282 // MP3
283 // int Kernel::Exec(char* name)
284 int Kernel::Exec(char* name, int priority)
285 {
286     // t[threadNum] = new Thread(name, threadNum);
287     // MP3
288     t[threadNum] = new Thread(name, threadNum);
289     t[threadNum]->setPriority(priority);
290
291     t[threadNum]->space = new AddrSpace();
292     t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);
293     threadNum++;
294
295     return threadNum-1;
```

create好thread之後，設定他的initital priority

Kernel.h

```
40     // MP3
41     int Exec(char* name, int priority);
```

```
83   // MP3      You, 1 s
84   int priorities[10];
```

Scheduler.h

```
35      // MP3
36      int Aging();
37      // Thread* CheckPreemptive();
38      bool CheckPreemptive();
39
40      // SelfTest for scheduler is implemented in class Thread
41
42    private:
43
44      List<Thread *> *readyList;  // queue of threads that are ready to run,
45          // but not running
46      // MP3
47      SortedList<Thread *> *L1;
48      SortedList<Thread *> *L2;
49      List<Thread *> *L3;
```

Scheduler.cc

```
69    Scheduler::Scheduler()
70    {
71        readyList = new List<Thread *>;
72        // MP3
73        L1 = new SortedList<Thread *>(L1Compare);
74        L2 = new SortedList<Thread *>(L2Compare);
75        L3 = new List<Thread *>;
76
77        toBeDestroyed = NULL;
78    }
```

在scheduler新增L1, L2, L3三個queue，L1和L2是sortedList，因為要根據remain time和priority來排序，在create sortedlist時，要把compare的function傳進去(L1Compare, L2Compare)。

```
85    Scheduler::~Scheduler()
86    {
87        delete readyList;
88        // MP3
89        delete L1;
90        delete L2;
91        delete L3;
92    }
```

```
// MP3
int
L1Compare(Thread *t1, Thread *t2) {
    // smaller burst time first
    double t1ApproxRemainTime = t1->getApproxRemainTime();
    double t2ApproxRemainTime = t2->getApproxRemainTime();

    if (t1ApproxRemainTime > t2ApproxRemainTime){
        return 1;
    }
    else if (t1ApproxRemainTime < t2ApproxRemainTime){
        return -1;
    }
    else {
        return 0;
    }
}
```

L1Compare比較的是approxRemainTime，approxRemainTime較小的優先

```
// MP3
int
L2Compare(Thread *t1, Thread *t2) {
    // higher priority first
    int t1Priority = t1->getPriority();
    int t2Priority = t2->getPriority();

    if (t1Priority < t2Priority) {
        return 1;
    }
    else if (t1Priority > t2Priority) {
        return -1;
    }
    else {
        return 0;
    }
}
```

L2Compare比較的是priority，priority較大的優先

```
103  Scheduler::ReadyToRun (Thread *thread)
104  {
105      ASSERT(kernel->interrupt->getLevel() == IntOff);
106      DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
107      //cout << "Putting thread on ready list: " << thread->getName() << endl ;
108      thread->setStatus(READY);
109
110      // readyList->Append(thread);
111      // MP3
112      int threadPriority = thread->getPriority();
113
114      thread->setStartWaitingTime(kernel->stats->totalTicks);
115
116      if (threadPriority > 149) {
117          // out of range
118          DEBUG(z, "Thread priority is out of range");
119      }
120      else if (threadPriority >= 100) {
121          // L1 (100 - 149)
122          L1->Insert(thread);
123          thread->InsertedIntoQueue(1);
124      }
125      else if (threadPriority >= 50) {
126          // L2 (50 - 99)
127          L2->Insert(thread);
128          thread->InsertedIntoQueue(2);
129      }
130      else if (threadPriority >= 0) {
131          // L3 (0 - 49)
132          L3->Append(thread);
133          thread->InsertedIntoQueue(3);
134      }
135      else {
136          // out of range
137          DEBUG(z, "Thread priority is out of range");
138      }
139
```

當要將thread放進ready queue時，我們先將start waiting time設為現在，然後根據thread的priority用Insert(thread)把thread放進queue，InsertedIntoQueue()則設定thread的queuelevel並print出debug訊息

```
150  Thread *
151  Scheduler::FindNextToRun ()
152  {
153      ASSERT(kernel->interrupt->getLevel() == IntOff);
154
155      // if (readyList->IsEmpty()) {
156      //  return NULL;
157      // } else {
158      //  return readyList->RemoveFront();
159      // }
160      // MP3
161  >   if (!L1->IsEmpty()) {···
173  >   else if (!L2->IsEmpty()) {···
185  >   else if (!L3->IsEmpty()) {···
197      else {
198          return NULL;
199      }
200  }
```

FindNextToRun依序從L1, L2, L3尋找ready thread

```
161      if (!L1->IsEmpty()) {
162          Thread *currThread = kernel->currentThread;
163          Thread *nextThread = L1->RemoveFront();
164
165          nextThread->setCpuStartTime(kernel->stats->totalTicks);
166          nextThread->setTotalWaitingTime(0);
167
168          nextThread->RemovedFromQueue();
169          currThread->ContextSwitch(nextThread->getID());
170
171          return nextThread;
172      }
```

假設L1 queue中有next thread，則把它從L1 queue中remove，然後將cpuStartTime設為現在，totalWaitingTime重置為0，RemovedFromQueue()和ContextSwitch()會print出debug訊息，最後回傳next thread

```
276  // MP3
277  int
278  Scheduler::Aging()
279  {
280      Thread *thread;
281      ListIterator<Thread *> *iterator;
282
283  >   if (!L1->IsEmpty()) {···
299
300  >   if (!L2->IsEmpty()) {···
323
324  >   if (!L3->IsEmpty()) {···
347  }
```

```
283      if (!L1->IsEmpty()) {
284          iterator = new ListIterator<Thread *>(L1);
285          for (; !iterator->IsDone(); iterator->Next()) {
286              thread = iterator->Item();
287
288              // Increase total waiting time
289              bool doAging = thread->IncreaseTotalWaitingTime();
290              thread->setStartWaitingTime(kernel->stats->totalTicks);
291
292              // Check whether over 1500
293              if (doAging) {
294                  thread->ChangePriority();
295              }
296
297          }
298      }
```

```
300      if (!L2->IsEmpty()) {
301          iterator = new ListIterator<Thread *>(L2);
302          for (; !iterator->IsDone(); iterator->Next()) {
303              thread = iterator->Item();
304
305              // Increase total waiting time
306              bool doAging = thread->IncreaseTotalWaitingTime();
307              thread->setStartWaitingTime(kernel->stats->totalTicks);
308
309              // Check whether over 1500
310              if (doAging) {
311                  thread->ChangePriority();
312              }
313
314              if (thread->getPriority() > 99) {
315                  L2->Remove(thread);
316                  L1->Insert(thread);
317
318                  thread->RemovedFromQueue();
319                  thread->InsertedIntoQueue(1);
320              }
321          }
322      }
```

```
324     if (!L3->IsEmpty()) {
325         iterator = new ListIterator<Thread *>(L3);
326         for (; !iterator->IsDone(); iterator->Next()) {
327             thread = iterator->Item();
328 
329             // Increase total waiting time
330             bool doAging = thread->IncreaseTotalWaitingTime();
331             thread->setStartWaitingTime(kernel->stats->totalTicks);
332 
333             // Check whether over 1500
334             if (doAging) {
335                 thread->ChangePriority();
336             }
337 
338             if (thread->getPriority() > 49) {
339                 L3->Remove(thread);
340                 L2->Insert(thread);
341 
342                 thread->RemovedFromQueue();
343                 thread->InsertedIntoQueue(2);
344             }
345         }
346     }
347 }
```

Aging會對所有queue中的thread去做檢查看是否要等待超過1500 ticks，有的話就增加10 priority，L2和L3中的thread需要再確認queue level是否可以升級了

```
353  bool
354  Scheduler::CheckPreemptive()
355  {
356      Thread *currThread = kernel->currentThread;
357      int currThreadLevel = currThread->getQueueLevel();
358
359      if (currThreadLevel == 1) {
360          if (!L1->IsEmpty()) {
361              if (L1Compare(currThread, L1->Front()) == 1) {
362                  return TRUE;
363              }
364          }
365      }
366      else if (currThreadLevel == 2) {
367          if (!L1->IsEmpty()) {
368              return TRUE;
369          }
370      }
371      else {
372          if (!L1->IsEmpty()){
373              return TRUE;
374          }
375          if (!L2->IsEmpty()) {
376              return TRUE;
377          }
378          if (!L3->IsEmpty() && currThread->getAccuTicks() >= 100) {
379              return TRUE;
380          }
381      }
382
383      return FALSE;
384  }
```

1. 如果現在的thread是L1，只有L1的thread可以preemptive，所以拿他的remain time去和現在L1的第一個thread比，決定是否可以preemptive
2. 如果現在的thread是L2，只有L1的thread可以preemptive
3. 如果現在的thread是L3，L1和L2所有thread都可以preempt，L3則要看現在的time quantum用完沒

alarm.cc

```
46  void
47  Alarm::CallBack() 
48  {
49      Interrupt *interrupt = kernel->interrupt;
50      MachineStatus status = interrupt->getStatus();
51
52      // MP3
53      // calculate burst and remain time
54      kernel->currentThread->IncreaseAccuTicks();
55      kernel->currentThread->IncreaseCpuBurstTime();
56      kernel->currentThread->UpdateApproxRemainTime();
57      kernel->currentThread->setCpuStartTime(kernel->stats->totalTicks);   // Because we have to keep accumulate burst time
58
59      // do aging and then check preemptive
60      kernel->scheduler->Aging();
61      // nextThread = kernel->scheduler->CheckPreemptive();
62
63      if (status != IdleMode) {
64          if (kernel->scheduler->CheckPreemptive()) {
65              interrupt->YieldOnReturn();
66          }
67      }
68  }
```

在time slice到時，會呼叫CallBack()，在裡面計算accuTicks、cpuBurstTime、approxRemainTime，接著呼叫aging。最後確認thread是否可被其他thread搶，若可以則會呼叫YieldOnReturn()會把yieldOnReturn設為true，然後在OneTick()就會呼叫Yield()

thread.h

```
156     // MP3
157     int priority;
158     int queueLevel;
159
160     int startWaitingTime;
161     int totalWaitingTime;
162
163     double cpuStartTime;
164     double cpuBurstTime;     // T
165     double approxBurstTime;
166     double approxRemainTime;
167
168     double accuTicks;
```

在class Thread中新增幾項data，如下:

int priority:該thread的priority

int queueLevel:該thread在哪個ready queue

int startWaitingTime: 進入Waiting state的時間

int totalWaitingTime: 在waiting等了多少時間

double cpuStartTime: 進入Ready state的時間

double cpuBurstTime: 進入RuningState的時間

double approxBurstTime: 估計會進入RuningState的時間

double approxRemainTime: 估計所需要的剩餘時間

double accuTicks: 最後一次的連續cpuBurstTime

```
111     int getPriority() { return priority; }
112     int getQueueLevel() { return queueLevel; }
113     int getStartWaitingTime() { return startWaitingTime; }
114     int getTotalWaitingTime() { return totalWaitingTime; }
115
116     double getCpuStartTime() { return cpuStartTime; }
117     double getCpuBurstTime() { return cpuBurstTime; }
118     double getApproxBurstTime() { return approxBurstTime; }
119     double getApproxRemainTime() { return approxRemainTime; }
120
121     void setPriority(int initPriority) { priority = initPriority; }
122     void setStartWaitingTime(int newTime) { startWaitingTime = newTime; }
123     void setTotalWaitingTime(int newTime) { totalWaitingTime = newTime; }
124
125     void setCpuStartTime(double newTime) { cpuStartTime = newTime; }
126     void setCpuBurstTime(double newTime) { cpuBurstTime = newTime; }
127     void setAccuTicks(double newTick) { accuTicks = newTick; }
```

定義一些get和set function

Thread.cc

```
36  Thread::Thread(char* threadName, int threadID)
37  {
38      ID = threadID;
39      name = threadName;
40      // MP3
41      priority = 0;
42      queueLevel = 0;
43
44      totalWaitingTime = 0;
45      startWaitingTime = 0;
46
47      cpuStartTime = 0;
48      cpuBurstTime = 0;
49      approxBurstTime = 0;
50      approxRemainTime = 0;
51
52      accuTicks = 0;
```

Thread::Thread把剛才新加的data初始化為0

```
214  void
215  Thread::Yield ()
216  {
217      Thread *nextThread;
218      IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);
219
220      ASSERT(this == kernel->currentThread);
221
222      DEBUG(dbgThread, "Yielding thread: " << name);
223
224      // // MP3
225      // // nextThread = kernel->scheduler->FindNextToRun();
226      // // calculate burst and remain time
227      // this->IncreaseAccuTicks();
228      // this->IncreaseCpuBurstTime();
229      // this->UpdateApproxRemainTime();
230      // this->setCpuStartTime(kernel->stats->totalTicks);   // Because we have to keep accumulate burst time
231
232      // // do aging and then check preemptive
233      // kernel->scheduler->Aging();
234      // nextThread = kernel->scheduler->CheckPreemptive();
235      nextThread = kernel->scheduler->FindNextToRun();
236
237      if (nextThread != NULL) {
238          kernel->scheduler->ReadyToRun(this);
239          kernel->scheduler->Run(nextThread, FALSE);
240      }
241
242      (void) kernel->interrupt->SetLevel(oldLevel);
243  }
```

Thread::Yield()

current thread要將讓出cpu給其他thread，呼叫FindNextToRun()去找下一個thread

```
254  void
255  Thread::Sleep (bool finishing)
256  {
257      Thread *nextThread;
258
259      ASSERT(this == kernel->currentThread);
260      ASSERT(kernel->interrupt->getLevel() == IntOff);
261
262      DEBUG(dbgThread, "Sleeping thread: " << name);
263      DEBUG(dbgTraCode, "In Thread::Sleep, Sleeping thread: " << name << ", " << kernel->stats->totalTicks);
264
265      status = BLOCKED;
266
267      // MP3
268      this->IncreaseAccuTicks();
269      this->IncreaseCpuBurstTime();
270      // Only update approximate burst time from running to waiting
271      if (!finishing) {
272          this->UpdateApproxBurstTime();
273      }
274
275      //cout << "debug Thread::Sleep " << name << "wait for Idle\n";
276      while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
277          kernel->interrupt->Idle();  // no one to run, wait for an interrupt
278      }
279
280      // clear cpu burst time from running to waiting
281      this->setCpuBurstTime(0);
282
283
284      // returns when it's time for us to run
285      kernel->scheduler->Run(nextThread, finishing);
286  }
```

Thread::Sleep()

先計算accuTick、cpuBurstTime、approxBurstTime，然後因為thread從runnning移到了waiting，故要重設cpuBurstTime為0

```
293  // MP3
294  void
295  Thread::InsertedIntoQueue(int newQueueLevel)
296  {
297      queueLevel = newQueueLevel;
298
299      DEBUG(z, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" << ID << "] is inserted into queue L[" << queueLevel << "]");
300  }
```

Thread::InsertedIntoQueue()當有thread priority改變，以致從一個queue跑到另一個 queue時呼叫(如L2->L1，就會insert到L1)

```
302  // MP3
303  void
304  Thread::RemovedFromQueue()
305  {
306      DEBUG(z, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" << ID << "] is removed from queue L[" << queueLevel << "]");
307  }
```

Thread::RemovedFromQueue()當有thread priority改變，以致從一個queue跑到另一個 queue時呼叫(如L2->L1，就會從L2移出)

```
310  void
311  Thread::ChangePriority()
312  {
313      int oldPriority = this->priority;
314      int newPriority = min(oldPriority + 10, 149);
315
316      priority = newPriority;
317
318      DEBUG(z, "[C] Tick [" << kernel->stats->totalTicks << "]: Thread [" << ID << "] changes its priority from [" << oldPriority << "] to [" << newP
319  }
```

將priority增加10，上限是149

```
341  // MP3
342  void
343  Thread::IncreaseCpuBurstTime()
344  {
345      cpuBurstTime = cpuBurstTime + (kernel->stats->totalTicks - cpuStartTime);
346  }
```

累積cpuBurstTime

```
348  // MP3
349  bool
350  Thread::IncreaseTotalWaitingTime()
351  {
352      totalWaitingTime = totalWaitingTime + (kernel->stats->totalTicks - startWaitingTime);
353
354      if (totalWaitingTime >= 1500) {
355          totalWaitingTime = totalWaitingTime - 1500;
356          return TRUE;
357      }
358      else {
359          return FALSE;
360      }
361  }
```

累積totalWaitingTime，如果等了超過1500 Ticks，則減去1500並回傳true，代表要增加priority

```
323  Thread::UpdateApproxBurstTime()
324  {
325      double newApproxBurstTime = 0.5 * cpuBurstTime + 0.5 * approxBurstTime;
326
327      DEBUG(z, "[D] Tick [" << kernel->stats->totalTicks << "]: Thread [" << ID << "
328
329      // update approxBurstTime and reset approxRemainTime
330      approxBurstTime = newApproxBurstTime;
331      approxRemainTime = newApproxBurstTime;
332  }
```

當thread從running to waiting，用公式計算新的approxBurstTime和approxRemainTime

```
334  // MP3
335  void
336  Thread::ContextSwitch(int newThreadId)
337  {
338      DEBUG(z, "[E] Tick [" << kernel->stats->totalTicks << "]: Thread [" << newThreadId << "] is nc
339  }
```

```
363  // MP3
364  void
365  Thread::UpdateApproxRemainTime()
366  {
367      approxRemainTime = approxBurstTime - cpuBurstTime;
368  }
```

更新approxRemainTime

```
370  // MP3
371  void
372  Thread::IncreaseAccuTicks()
373  {
374      accuTicks = accuTicks + (kernel->stats->totalTicks - cpuStartTime);
375  }
```

累積AccuTicks

心得

part 1 因為跟前兩次有蠻多地方重疊，所以code trace比較快，也比較快理解thread是如何在各個state切換。

part 2 則讓我們學會multy queueLevel scheduling是如何在OS上實作的，像是non-preemtive跟preemptive在實作上的差異，以及SJF如何實現的等等。