MP3_report_56

- ## Team members and contribution

資工大三 108020022 周昱宏

資工大三 108071003 李彥璋

工作內容	負責人
Trace code	周昱宏、李彥璋
Implementation、測試、Debug	周昱宏
Report II-1	李彥璋
Report II-2	周昱宏

- ## II-1、Trace code

1-1. New -> Ready

threads/kernel.cc – Kernel:: ExecAll()

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

[265] 以 Kernel::Exec()，執行 execfile[]中所有檔案(user program).

[267] 當所有 execfile 執行結束，currentThread 以 Finish()結束 NachOS.

threads/kernel.cc – Kernel::Exec()

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

[274] 為每個檔案初始化一個 thread control block, 之後我們才能 call Thread::Fork().

[275] 賦予剛創建的 thread 一個定址空間(address space), 之後我們才能 run user program.

[276] 以 Fork(), turns a thread into one that the CPU can schedule and execute.

以下為 Thread::Thread()的 code.

```
36 Thread::Thread(char* threadName, int threadID)
37 {
38     ID = threadID;
39     name = threadName;
40     stackTop = NULL;
41     stack = NULL;
42     status = JUST_CREATED;
43     for (int i = 0; i < MachineStateSize; i++) {
44     machineState[i] = NULL;     // not strictly necessary, since
45                                 // new thread ignores contents
46                                 // of machine registers
47     }
48     space = NULL;
49 }
```

[42] 可以看到這個 thread 的 status 目前為 JUST_CREATED.
(The thread exists, but has no stack yet.
This state is a temporary state used during thread creation).

threads/thread.cc – Thread::Fork()

```
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 }
```

[92] Argument func is the address of a procedure where execution is to begin when the thread starts executing.
[92] Argument arg is an argument that should be passed to the procedure.

[94-95] 因為要使用 NachOS 的 interrupt & scheduler, 所以宣告這兩個指標.

[99] 以 StackAllocate(), does the work of allocating the stack and initializing the stack so that a call to SWITCH will cause it to run the procedure(將於下一小節說明).
[101] disable interrupt.
[102] scheduler puts the thread on the ready queue(將於下下一小節說明).

threads/thread.cc – Thread::StackAllocate()

```
1   void
2   Thread::StackAllocate (VoidFunctionPtr func, void *arg)
3   {
4       stack = (int *) AllocBoundedArray(StackSize * sizeof(int));
5       .
6       .
7       .
8       .
9       .
10      .
11  #ifdef x86
12      stackTop = stack + StackSize - 4;
13      *(--stackTop) = (int) ThreadRoot;
14      *stack = STACK_FENCEPOST;
15  #endif
16
17      machineState[PCState] = (void*)ThreadRoot;
18      machineState[StartupPCState] = (void*)ThreadBegin;
19      machineState[InitialPCState] = (void*)func;
20      machineState[InitialArgState] = (void*)arg;
21      machineState[WhenDonePCState] = (void*)ThreadFinish;
22  }
```

[4] Allocate memory for the stack, stack 指標指向頂部(Low address).

[12] StackTop 指向 stack 底部(High address), -4 確保安全.

[13] 根據以下註解,

The x86 passes the return address on the stack.
In order for SWITCH() to go to ThreadRoot when we switch to this thread, the return addres used in SWITCH() must be the starting address of ThreadRoot.

也就是說, 讓 stack 的第一個元素為 ThreadRoot 的 address, SWITCH() 將來 switch 到這個 thread 的時候, 就可以直接將 ThreadRoot 取出並執行.

[17-21] 設置 machine state, 之後 ThreadRoot 會做以下事項：

1. Calls an initialization routine that simply enables interrupts, 也就是 Thread::Begin().
2. Calls the user-supplied function, 也就是&ForkExecute, passing it the supplied argument, 也就是 t[ThreadNum].
3. Calls thread::Finish(), to terminate the thread.

threads/scheduler.cc – Scheduler::ReadyToRun()

```
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 }
```

[59] 在 thread::Finish()—(Thread::StackAllocate()執行結束的地方), 會先 disable interrupt, 這邊要確保是否正確 disable 了.
[62] Mark a thread as ready, but not running.
[63] Put it on the ready list, for later scheduling onto the CPU.

1-2. Running -> Ready

machine/mipssim.cc – Machine::Run()

Run() turns on the MIPS machine, simulating the execution of a user-level program on Nachos.

This routine should only be called after machine registers and memory have been properly initialized (指的是 Thread::StackAllocate()當中設置的 machine state).

```
1  void
2  Machine::Run()
3  {
4      Instruction *instr = new Instruction;
5  
6      kernel->interrupt->setStatus(UserMode);
7      for (;;) {
8          OneInstruction(instr);
9          kernel->interrupt->OneTick();
10     }
11 }
```

[7] infinite fetch-execute loop.

[8] 以 OneInstruction()，執行 actually 一個 instruction.

[9] 在每個 instruction 後，呼叫 OneTick()(將於下一小節說明).

machine/interrupt.cc – 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 }
```

[154-160] advances the time.

[164] turns off interrupt, 確保以下為 atomic.

[165] CheckIfDue() checks if any pending(待辦的) interrupts due.

以下將先解釋 CheckIfDue()的 code.(待會再回來 OneTick()).

```
1  bool
2  Interrupt::CheckIfDue(bool advanceClock)
3  {
4      PendingInterrupt *next;
5      Statistics *stats = kernel->stats;
6      ASSERT(level == IntOff);         // interrupts need to be disabled,
7                                       // to invoke an interrupt handler
8      if (pending->IsEmpty()) {        // no pending interrupts
9          return FALSE;
10     }
11     next = pending->Front();
12
13     if (next->when > stats->totalTicks) {
14         if (!advanceClock) {         // not time yet
15             return FALSE;
16         }
17         else {                       // advance the clock to next interrupt
18             stats->idleTicks += (next->when - stats->totalTicks);
19             stats->totalTicks = next->when;
20             // UDelay(1000L); // rcgood - to stop nachos from spinning.
21         }
22     }
23
24     inHandler = TRUE;
25     do {
26         next = pending->RemoveFront();    // pull interrupt off list
27         next->callOnInterrupt->CallBack();// call the interrupt handler
28     delete next;
29     } while (!pending->IsEmpty() && (pending->Front()->when <= stats->totalTicks));
30     inHandler = FALSE;
31     return TRUE;
32 }
```

[6] 確保 interrupted is disabled(因為這裡必須是 atomic).

[11] 取出 pending queue 的第一個元素.

首先必須先了解 CheckIfDue()的運作機制:

CheckIfDue() return False 有以下幾種狀況:

1. no pending interrupts (line [8-10]).
2. no interrupt is due and <u>the ready queue is not empty(從傳入 CheckIfDue()的參數 advanceClock=FALSE 可以得知)</u> (line [13-16]).
3. no interrupt is due and <u>the ready queue is empty (從傳入 CheckIfDue()的參數 advanceClock=TRUE 可以得知)</u>, then fast forward the clock to the next interrupt (line [17-21]).

CheckIfDue() return True: (some interrupt is due), if interrupts have been fired off (line [24 以下]).

之所以會 Running -> Ready, 是因為 Timer 定期發出一個 Interrupt.

而當 a timer interrupt occurs, the Timer::CallBack() gets run.

Timer::CallBack()會 calls Alarm::CallBack(),

Alarm::CallBack()接著又 calls YieldOnReturn().

以下為 YieldOnReturn()的 code:

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

[192] inHandler == TRUE, 是在 CheckIfDue() line[24]設定好的.

[193] yieldOnReturn = TRUE 在回到 OneTick()的時候會用到.

接著回到 Interrupt::OneTick() line[168]:

```
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 }
```

[168] enable interrupts.

[169-173] yieldOnReturn == TRUE 已經在上面解釋過, 首先恢復 yieldOnReturn FASLE(因為 timer 的目的已經達成), 接著轉換成 kernel mode(因為 Yield() is a kernel routine), 有關 Yield()的部分會在下一小節說明.

[174] 切換為 oldStatus (這裡應指 UserMode)，因為要回到 Machine::Run()的迴圈內繼續執行 user program.

threads/thread.cc – Thread::Yield()

Thread::Yield() suspends the calling thread and selects a new one for execution.

```
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 }
```

[204] disabled interrupts(disable interrupts, so that looking at the thread on the front of the ready list, and switching to it, can be done atomically).

[212] FindNextToRun() finds the next thread to run(將於下一小節說明).

[213] Run() runs the next thread(將於下下一小節說明).

threads/schedule.cc – 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 }
```

[82] Return the next thread to be scheduled onto the CPU.
[80] If there are no ready threads, return NULL.

threads/schedule.cc – Scheduler::ReadyToRun()

```
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  }
```

[62] Mark a thread as ready, but not running.
[63] Put it on the ready list, for later scheduling onto the CPU.

threads/schedule.cc – Scheduler::Run()

```
1  void
2  Scheduler::Run (Thread *nextThread, bool finishing)
3  {
4      Thread *oldThread = kernel->currentThread;
5      ASSERT(kernel->interrupt->getLevel() == IntOff);
6
7      if (oldThread->space != NULL) {     // if this thread is a user program,
8          oldThread->SaveUserState();     // save the user's CPU registers
9          oldThread->space->SaveState();
10     }
11
12     oldThread->CheckOverflow();         // check if the old thread
13                                         // had an undetected stack overflow
14
15     kernel->currentThread = nextThread;  // switch to the next thread
16     nextThread->setStatus(RUNNING);      // nextThread is now running
17
18     SWITCH(oldThread, nextThread);
19
20     // we're back, running oldThread
21
22     // interrupts are off when we return from switch!
23     ASSERT(kernel->interrupt->getLevel() == IntOff);
24
25     CheckToBeDestroyed();        // check if thread we were running
26                                  // before this one has finished
27                                  // and needs to be cleaned up
28
29     if (oldThread->space != NULL) {        // if there is an address space
30         oldThread->RestoreUserState();     // to restore, do it.
31         oldThread->space->RestoreState();
32     }
33 }
```

[7-10] 保存 oldThread 的資訊.

[16] nextThread is set to RUNNING state.

[18] SWITCH(), written in assembly code, actually switches the CPU from the old Thread (oldThread) with the new one (nextThread).這個部分會在 1-6 說明.

[20-25] back to oldThread, see whether it should be destroyed(terminated).

[29-32] 恢復 oldThread 的資訊.

1-3. Running -> Waiting

userprog/synchconsole.cc – SynchConsoleOutput::PutChar()
Write a character to the console display, waiting if necessary.

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

[103]以下為有關 Lock 的 code:

Lock 的實現,

```
160  Lock::Lock(char* debugName)
161  {
162      name = debugName;
163      semaphore = new Semaphore("lock", 1);
164      lockHolder = NULL;
165  }
```

[163] Initialize a semaphore, so that it can be used for synchronization(一開始為 unlocked).

lock->Acquire(),

```
183  void Lock::Acquire()
184  {
185      semaphore->P();
186      lockHolder = kernel->currentThread;
187  }
```

[185] Atomically wait until the lock is free, then set it to busy(P()將在下一小節說明).

[186] 讓 CurrentThread 持有這個 Lock.

回到 SynchConsoleOutput::PutChar(),

[104] 以下將介紹 ConsoleOutput::PutChar():

```
167 void
168 ConsoleOutput::PutChar(char ch)
169 {
170     ASSERT(putBusy == FALSE);
171     WriteFile(writeFileNo, &ch, sizeof(char));
172     putBusy = TRUE;
173     kernel->interrupt->Schedule(this, ConsoleTime, ConsoleWriteInt);
174 }
```

[170] 必須先確認 putBusy 是否==FALSE，代表目前沒有其他 thread 在輸出.

[171] Write characters to an open file.

[172] 將 putBusy 設定為 TRUE，代表正在 putChar.

[173] 以下介紹 Interrupt::Schedule():

```
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
304     ASSERT(fromNow > 0);
305
306     pending->Insert(toOccur);
307 }
```

[298]
toCall is the object to call when the interrupt occurs(也就是 ConsoleOutput::CallBack())
fromNow is how far in the future (in simulated time) the interrupt is to occur,這邊傳入的是 ConsoleTime，在這次作業當中 ConsoleTime 被設定為 1，所以下一個 tick 就會發生 ConsoleWrite Interrupt.

[301、306]
just put it(tocall) on a sorted list(interrupt pending list).

putChar 完成後，上述的 tocall 將被執行，也就是 ConsoleOutput::CallBack()，而它又會 call SynchConsoleOutput::CallBack()，

```
void
SynchConsoleInput::CallBack()
{
    waitFor->V();
}
```

SynchConsoleOutput::CallBack()呼叫 Semaphore::V()，以下介紹 Semaphore::V()：

```
103  void
104  Semaphore::V()
105  {
106      DEBUG(dbgTraCode, "In Semaphore::V(), " << kernel->s
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  }
```

Semaphore::V()做的事情：
Increment semaphore value, waking up a waiter if necessary.

[110] this operation must be atomic, so we need to disable interrupts.
[113] make thread ready.
[115] increases semaphore value，也因此之後 Semaphore::P()能夠執行(P()將在下一小節說明).

再次回到 SynchConsoleOutput::PutChar()，

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

[105] Wait until semaphore value > 0, then decrement，剛剛 Semaphore::V()已經 value++了，所以現在是 > 0 沒錯.(P()將在下一小節說明).
[106] Atomically set lock to be free, waking up a thread waiting for the lock, if any.

threads/synch.cc – Semaphore::P()

```
75  void
76  Semaphore::P()
77  {
78      DEBUG(dbgTraCode, "In Semaphore::P(), " << kernel-
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 avail
86      queue->Append(currentThread);   // so go to sleep
87      currentThread->Sleep(FALSE);
88      }
89      value--;            // semaphore available, consum
90
91      // re-enable interrupts
92      (void) interrupt->SetLevel(oldLevel);
93  }
```

[83] Checking the value and decrementing must be done atomically, so we need to disable interrupts before checking the value.

[85] 如果沒有 available 的 semaphore(value == 0), 將 currentThread(在等待 semaphore) Append 進 queue 裡面.

該 queue 定義於 semaphore 的 class 當中,

```
    List<Thread *> *queue;
        // threads waiting in P() for the value to be > 0
```

[87] Relinquish the CPU, because the current thread is blocked waiting on a synchronization variable (Semaphore).

Thread::Sleep()會在下下一小節說明.

[89] semaphore available, value--.

lib/list.cc – List<T>::Append

Append an "item" to the end of the list.

它其實就很像是一個 linked list...

```
template <class T>
void
List<T>::Append(T item)
{
    ListElement<T> *element = new ListElement<T>(item);

    ASSERT(!IsInList(item));
    if (IsEmpty()) {        // list is empty
    first = element;
    last = element;
    } else {            // else put it after last
    last->next = element;
    last = element;
    }
    numInList++;
    ASSERT(IsInList(item));
}
```

threads/thread.cc – Thread::Sleep()

```
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 << "
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 interr
253     }
254     // returns when it's time for us to run
255     kernel->scheduler->Run(nextThread, finishing);
256 }
```

[249] 在等待 semaphore 的 currentThread's status 設定為 BLOCKED.
[251] FindNextToRun() check whether there are threads on the ready queue or not, if not, call Idle(), 以下為 Idle()的 code:

```
207 void
208 Interrupt::Idle()
209 {
210     DEBUG(dbgInt, "Machine i
211     status = IdleMode;
212     DEBUG(dbgTraCode, "In In
213     if (CheckIfDue(TRUE)) {
214     DEBUG(dbgTraCode, "In In
215     status = SystemMode;
216     return;         // retur
217                 // a runnabl
218     }
219     DEBUG(dbgTraCode, "In In
220 
221     // if there are no pendi
222     // queue, it is time to
223     // operating, there are
224     // is not reached.  Inst
225 
226     DEBUG(dbgInt, "Machine i
227     cout << "No threads read
228     cout << "Assuming the pr
229     Halt();
230 }
```

[213] CheckIfDue(TRUE)如果 == true, 代表有 pending 的 interrupt, return in case there's now a runnable thread.
[229] if there are no pending interrupts, and nothing is on the ready queue, it is time to stop.

[255] 若有 nextThread, 就讓它執行.

threads/schedule.cc – Scheduler::FindNextToRun()
threads/schedule.cc – Scheduler::Run()

1-2. Running -> Ready 已經說明過, 不贅述.

1-4. Waiting -> Ready

threads/synch.cc – Semaphore::V()

1-3. Running -> Waiting (page.16)已經說明過.

threads/schedule.cc – Scheduler::ReadyToRun()

1-1. New -> Ready (page.6)已經說明過.

簡單來說，上面兩個，以 console output as an example，就是當 output 完成後，CallBack function 呼叫 Semaphore::V().

而 Semaphore::V() increases semaphore value, waking up a waiter if necessary(ReadyToRun()).

1-5. Running -> Terminated

userprog/exception.cc – ExceptionHandler() case SC_Exit

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

執行完所有的 Code, RaiseException -> ExceptionHandler (Exit system call is called).
[194] call Finish().

threads/thread.cc – 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 }
```

[173] Sleep() assumes interrupts are disabled.
[177] Call Sleep(), 注意這邊傳進去的參數是 TRUE.

threads/thread.cc – Thread::Sleep()

和 1-3. Running -> Waiting 提及過的幾乎一樣, 只是這次呼叫 Sleep() 的理由是 Relinquish the CPU, because the current thread has finished.

threads/schedule.cc – Scheduler::FindNextToRun()

1-2. Running -> Ready 已經說明過, 不贅述.

threads/schedule.cc – Scheduler::Run()

和 1-2. Running -> Ready 說明得不一樣的地方是這次傳進去的參數 finishing 是 TRUE.

```
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     }
```

```
20     // we're back, running oldThread
21 
22     // interrupts are off when we return from switch!
23     ASSERT(kernel->interrupt->getLevel() == IntOff);
24 
25     CheckToBeDestroyed();       // check if thread we were running
26                                 // before this one has finished
27                                 // and needs to be cleaned up
```

[110] indicate that we need to delete current thread.
[25] call CheckToBeDestroyed() check if thread we were running before has finished and needs to be cleaned up.

以下為 CheckToBeDestroyed()的 code:

```
160 void
161 Scheduler::CheckToBeDestroyed()
162 {
163     if (toBeDestroyed != NULL) {
164         delete toBeDestroyed;
165     toBeDestroyed = NULL;
166     }
167 }
```

剛剛 line [112]已經把 toBeDestroyed 設定成 oldThread,現在可以把它 delete 了.

1-6. Ready -> Running

threads/schedule.cc – Scheduler::FindNextToRun()
threads/schedule.cc – Scheduler::Run()
machine/mipssim.cc – Machine::Run()

1-2. Running -> Ready 已經說明過, 不贅述.

threads/switch.S – SWITCH(Thread*, Thread*)

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

其中 esp 是 stack pointer.

簡單來說, switch 做的事情如下:

1. 將 t1(oldThread)的相關 registers 保存至 memory.

2. 將 t2(newThread)的相關 registers 從 memory load 到 CPU registers.

3. ret, set CPU program counter to the memory address pointed by the value of register esp, 也就是 CPU 會執行 esp 位置的 threadRoot(threadBegin -> ForkExecute -> threadFinish).

```
        movl    %eax,_eax_save          # save the value of eax
        movl    4(%esp),%eax            # move pointer to t1 into eax
        movl    %ebx,_EBX(%eax)         # save registers
        movl    %ecx,_ECX(%eax)
        movl    %edx,_EDX(%eax)
        movl    %esi,_ESI(%eax)
        movl    %edi,_EDI(%eax)
        movl    %ebp,_EBP(%eax)
        movl    %esp,_ESP(%eax)         # save stack pointer
        movl    _eax_save,%ebx          # get the saved value of eax
        movl    %ebx,_EAX(%eax)         # store it
        movl    0(%esp),%ebx            # get return address from stack into ebx
        movl    %ebx,_PC(%eax)          # save it into the pc storage

        movl    8(%esp),%eax            # move pointer to t2 into eax

        movl    _EAX(%eax),%ebx         # get new value for eax into ebx
        movl    %ebx,_eax_save          # save it
        movl    _EBX(%eax),%ebx         # retore old registers
        movl    _ECX(%eax),%ecx
        movl    _EDX(%eax),%edx
        movl    _ESI(%eax),%esi
        movl    _EDI(%eax),%edi
        movl    _EBP(%eax),%ebp
        movl    _ESP(%eax),%esp         # restore stack pointer
        movl    _PC(%eax),%eax          # restore return address into eax
        movl    %eax,4(%esp)            # copy over the ret address on the stack
        movl    _eax_save,%eax

        ret
```

從後方註解 && Appendix C(放在下方),可以大致了解每個 instruction 在做甚麼.

x86 instructions (32bit, AT&T)

Instruction	Description
movl %eax, %ebx	move 32bit value from register eax to register ebx
movl 4(%eax), %ebx	move 32bit value at memory address pointed by (the value of register eax plus 4), to register ebx
ret	set CPU program counter to the memory address pointed by the value of register esp
pushl %eax	subtract register esp by 4 (%esp = %esp - 4), then move 32bit value of register eax to the memory address pointed by register esp
popl %eax	move 32bit value at the memory address pointed by register esp to register eax, then add register esp by 4 (%esp = %esp + 4)
call *%eax	push CPU program counter + 4 (return address) to stack, then set CPU program counter to memory address pointed by the value of register eax

IMPLENENTATION

2-1.

thread.h：Thread 的 Class 定義中額外新增了以下 member 與 method，如下圖。

```
140 public:
141     void SetPriority(int newPriority) { priority = newPriority; }
142     int GetPriority(void) { return priority; }
143
144     void SetApproximateBurstTime(int newApproximateBurstTime) { approximateBurstTime = newApproximateBurstTime; }
145     int GetApproximateBurstTime(void) { return approximateBurstTime; }
146
147     void SetExecutionTime(int newExecutionTime) { executionTime = newExecutionTime; }
148     int GetExecutionTime(void) { return executionTime; }
149
150     void SetWaitingTime(int newWaitingTime) { waitingTime = newWaitingTime; }
151     int GetWaitingTime(void) { return waitingTime; }
152
153     void SetL3waitingTime(int newL3waitingTime) { L3waitingTime = newL3waitingTime; }
154     int GetL3waitingTime(void) { return L3waitingTime; }
155
156     bool CanInL1ReadyList(void) { return this->GetPriority() >= 100 && this->GetPriority() <= 149; }
157     bool CanInL2ReadyList(void) { return this->GetPriority() >= 50 && this->GetPriority() <= 99; }
158     bool CanInL3ReadyList(void) { return this->GetPriority() >= 0 && this->GetPriority() <= 49; }
159
160 private:
161     int priority;
162
163     int approximateBurstTime;
164     int executionTime;
165     int waitingTime;
166     int L3waitingTime;
```

member：

[161]Thread 會根據 priority 進入不同 level 的 ready list。

[163]Thread 的 approximateBurstTime 用來決定 L1 ready list(preemptive SJF)的排程優先度。

[164] Thread 的 executionTime 為 Thread 在 CPU 中執行的時間。

[165] Thread 的 waitingTime 為 Thread 進入到 ready queue 等待的時間，若超過 1500 則更新為零並做 aging。

[166] Thread 的 L3waitingTime 為 Thread 在 L3 ready list 的等待時間，用以實作 RR。

method：

[141-154]為上述五個 member 的 set 及 get function。

[156-158]三個函式會根據 Thread 的 priority 決定是否可以進到相對應的 ready list，若一個 Thread 的 priority 為 55，則 CanInL2ReadyList()會回傳 true。

thread.cc：在 Yeild()函式中，將 this(thread)放入 ReadyToRun 前，新增下圖程式。

```
    int ExecutionTime = this->GetExecutionTime();
    int ApproximateBurstTime = this->GetApproximateBurstTime();

    this->SetApproximateBurstTime(max(0, ApproximateBurstTime - ExecutionTime));
```

因為 Yield()函式會終止當前在 CPU 執行的 Thread，並重新放回 ready list。因此根據 spec 中 running -> ready 的 approximateBurstTime 計算方式重新賦值。

thread.cc：在 Sleep()函式中，while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL)前，新增下圖程式。

```
    int originalApproximateBurstTime = this->GetApproximateBurstTime();
    int executiontime = this->GetExecutionTime();
    int newApproximateBurstTime = originalApproximateBurstTime * 0.5 + executiontime * 0.5;
    this->SetApproximateBurstTime(newApproximateBurstTime);
    DEBUG(dbgList, "[D] Tick [" << kernel->stats->totalTicks << "]: Thread [" << this->getID() << "] update approximate burst tim
```

因為 Sleep()函式會使 Thread 從 running -> waiting，因此根據 spec 中 approximateBurstTime 計算方式重新賦值。

scheduler.h：Sheduler 的 Class 定義中額外新增了以下 member 與 method，如下圖。

```
44  private:
45      SortedList<Thread *> *L1ReadyList;
46      SortedList<Thread *> *L2ReadyList;
47      List<Thread *> *L3ReadyList;
48
49  public:
50      void AdjustPriority(void);
```

member：

[45]利用 list.h 實作的 SortedList(初始化時會根據給定的 compare function 來排序 list 元素，如同 priority queue)來實現 L1 ready list(根據 approximateBurstTime 來排序)。

[46]利用 list.h 實作的 SortedList 來實現 L2 ready list(根據 priority 來排序)。

[47]利用 list.h 實作的 List 來實現 L3 ready list(RR 不需要排序，如同 queue)。

method：

[50]隨著 timer 做 aging 以及根據 Thread 的 prioity 做 ready list 的切換，詳細實作參閱下方。

scheduler.cc：額外宣告了兩個 compare function 分別為 L1CompareFun 以及 L2CompareFun，在 ready list 建構時作為排序依據。

```
int L1CompareFun(Thread *x, Thread *y)
{
    if (x->GetApproximateBurstTime() < y->GetApproximateBurstTime())
        return 1;

    else if (x->GetApproximateBurstTime() > y->GetApproximateBurstTime())
        return -1;
    else
        return 1;

    return 0;
}

int L2CompareFun(Thread *x, Thread *y)
{
    if (x->GetPriority() > y->GetPriority())
        return 1;

    else if (x->GetPriority() < y->GetPriority())
        return -1;
    else
        return 1;

    return 0;
}
```

L1CompareFun：專屬 L1 ready list 的排序依據，因為是 SJF，所以若 x 的 approximateBurstTime 小於 y 的 approximateBurstTime 則 x 要放置於 y 前面，反之。

L2CompareFun：專屬 L2 ready list 的排序依據，因為不是 SJF，所以若 x 的 priority 大於 y 的 priority 則 x 要放置於 y 前面，反之。

scheduler.cc：Scheduler 建構子，利用 list.h 宣告方式配合定義好的 compare function 進行 L1ReadyList、L2ReadyList、L3ReadyList 的初始化。

```
31  /*周改的*/
32  int L1CompareFun(Thread *, Thread *);
33  int L2CompareFun(Thread *, Thread *);
34  /*周改的*/
35  Scheduler::Scheduler()
36  {
37      readyList = new List<Thread *>;
38      toBeDestroyed = NULL;
39  
40      /*周改的*/
41      L1ReadyList = new SortedList<Thread *>(L1CompareFun);
42      L2ReadyList = new SortedList<Thread *>(L2CompareFun);
43      L3ReadyList = new List<Thread *>;
44      /*周改的*/
45  }
```

[32-33]因為 compare function 我是實作於下半部，因此事先宣告。

scheduler.cc：Scheduler 解構子，刪除宣告時的 ready list。

```
Scheduler::~Scheduler()
{
    delete readyList;

    /*周改的*/
    delete L1ReadyList;
    delete L2ReadyList;
    delete L3ReadyList;
    /*周改的*/
}
```

scheduler.cc：ReadyToRun()函式會根據參數 Thread 的 priority 放置入相對應的 ready list。

```
79      if (thread->CanInL1ReadyList())
80      {
81          if (!L1ReadyList->IsInList(thread))
82              L1ReadyList->Insert(thread);
83          DEBUG(dbgList, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" << thread->getID() << "] is inserted into queue L[" << 1 << "]");
84      }
85      else if (thread->CanInL2ReadyList())
86      {
87          if (!L2ReadyList->IsInList(thread))
88              L2ReadyList->Insert(thread);
89          DEBUG(dbgList, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" << thread->getID() << "] is inserted into queue L[" << 2 << "]");
90      }
91      else if (thread->CanInL3ReadyList())
92      {
93          DEBUG(dbgList, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" << thread->getID() << "] is inserted into queue L[" << 3 << "]");
94          if (!L3ReadyList->IsInList(thread))
95          {
96              L3ReadyList->Append(thread);
97          }
98      }
```

[79-84]根據前述 Thread Class method 決定此 thread 是否符合 L1 ready list 的 priority 範圍，再加上若此 thread 不在 L1 ready list 即可插入。

[85-90] L2 ready list 同 L1 ready list。

[91-98] L3 ready list 同 L1 ready list。

scheduler.cc：FindNextToRun()函式會決定下一個要進 CPU 執行的 Thread，ready list 順位優先度依序為 L1 ready list -> L2 ready list -> L3 ready list。

```
126     if (!L1ReadyList->IsEmpty())
127     {
128         DEBUG(dbgList, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" << L1ReadyList->Front()->getID() << "] is removed from queue L[" << 1 << "]");
129         return L1ReadyList->RemoveFront();
130     }
131
132     else if (!L2ReadyList->IsEmpty())
133     {
134         DEBUG(dbgList, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" << L2ReadyList->Front()->getID() << "] is removed from queue L[" << 2 << "]");
135         return L2ReadyList->RemoveFront();
136     }
137
138     else if (!L3ReadyList->IsEmpty())
139     {
140         DEBUG(dbgList, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" << L3ReadyList->Front()->getID() << "] is removed from queue L[" << 3 << "]");
141         Thread *removeThread = L3ReadyList->RemoveFront();
142         //Print();
143         return removeThread;
144     }
145
146     else
147         return NULL;
```

[126-130]最先判別 L1 ready list，若其不為空，則利用 list 的 RemoveFront(移除 approximateBurstTime 最短的)移除該元素並回傳。

[132-136]其次判別 L2 ready list，若其不為空，則利用 list 的 RemoveFront(移除 priority 最大的)移除該元素並回傳。

[138-144]最後判別 L3 ready list，若其不為空，則利用 list 的 RemoveFront 移除元素並回傳。

[146-147]若三個 ready list 皆為空，則 return NULL。

scheduler.cc：實作 AdjustPriority()函式，隨著 timer 做 aging 以及根據 Thread 的 prioity 做 ready list 的切換。

```
252 void Scheduler::AdjustPriority(void)
253 {
254 
255     ListIterator<Thread *> *it1 = new ListIterator<Thread *>(L1ReadyList);
256     ListIterator<Thread *> *it2 = new ListIterator<Thread *>(L2ReadyList);
257     ListIterator<Thread *> *it3 = new ListIterator<Thread *>(L3ReadyList);
258 
259     while (!it1->IsDone())
260     {
261         ASSERT(it1->Item()->getStatus() == READY);
262         it1->Item()->SetWaitingTime(it1->Item()->GetWaitingTime() + 100);
263         if (it1->Item()->GetWaitingTime() >= 1500)
264         {
265             int newPriority = it1->Item()->GetPriority() + 10;
266             newPriority = newPriority > 149 ? 149 : newPriority;
267             DEBUG(dbgList, "[C] Tick [" << kernel->stats->totalTicks << "]:
268             it1->Item()->SetPriority(newPriority);
269             it1->Item()->SetWaitingTime(0);
270         }
271         it1->Next();
272     }
```

[255-257]利用 list.h 宣告的 iterator 來遍歷三個 ready list。

[259-272]L1 ready list 的遍歷。

[262]根據每次 timer interrupt 更新每個 ready list 元素的 waiting time 增加 100。

[263-270]若元素的 waiting time 到達 1500，則要做 aging，將其 priority 增加 10(若超過 149 則回歸為 149)，並將 waiting time reset 為 0。

[271]指向下一個元素。

```
274     while (!it2->IsDone())
275     {
276         ASSERT(it2->Item()->getStatus() == READY);
277         it2->Item()->SetWaitingTime(it2->Item()->GetWaitingTime() + 100);
278         if (it2->Item()->GetWaitingTime() >= 1500)
279         {
280             int newPriority = it2->Item()->GetPriority() + 10;
281             newPriority = newPriority > 149 ? 149 : newPriority;
282             DEBUG(dbgList, "[C] Tick [" << kernel->stats->totalTicks << "]: 
283             it2->Item()->SetPriority(newPriority);
284             Thread *removeThread = it2->Item();
285             it2->Next();
286             L2ReadyList->Remove(removeThread);
287             DEBUG(dbgList, "[B] Tick [" << kernel->stats->totalTicks << "]: 
288             ReadyToRun(removeThread);
289             continue;
290         }
291         it2->Next();
292     }
```

[274-292]L2 ready list 的遍歷。

[277]根據每次 timer interrupt 更新每個 ready list 元素的 waiting time 增加 100。

[278-290]若元素的 waiting time 到達 1500，則要做 aging，將其 priority 增加 10(若超過 149 則回歸為 149)，並將 waiting time reset 為 0，這邊要注意的是有可能 aging 完會到達更上層的 ready list，因此我將其從該 ready list 移除並丟入 ReadyToRun 函式中讓此 Thread 放置於適當的 ready list。值得注意一點的事，為了讓 itreator 能繼續遍歷，我先將此 thread 記錄起來，指向下一個元素後，最後才刪除該元素，避免 iterator 指向 NULL。

[291]指向下一個元素。

```
294     while (!it3->IsDone())
295     {
296         ASSERT(it3->Item()->getStatus() == READY);
297         it3->Item()->SetWaitingTime(it3->Item()->GetWaitingTime() + 100);
298         if (it3->Item()->GetWaitingTime() >= 1500)
299         {
300             int newPriority = it3->Item()->GetPriority() + 10;
301             newPriority = newPriority > 149 ? 149 : newPriority;
302             DEBUG(dbgList, "[C] Tick [" << kernel->stats->totalTicks << "]
303             it3->Item()->SetPriority(newPriority);
304             Thread *removeThread = it3->Item();
305             it3->Next();
306             L3ReadyList->Remove(removeThread);
307             DEBUG(dbgList, "[B] Tick [" << kernel->stats->totalTicks << "]
308             ReadyToRun(removeThread);
309             continue;
310         }
311         it3->Next();
312     }
```

[294-312]L3 ready list 的遍歷。

[297]根據每次 timer interrupt 更新每個 ready list 元素的 waiting time 增加 100。

[298-310]若元素的 waiting time 到達 1500，則要做 aging，將其 priority 增加 10(若超過 149 則回歸為 149)，並將 waiting time reset 為 0，這邊要注意的是有可能 aging 完會到達更上層的 ready list，因此我將其從該 ready list 移除並丟入 ReadyToRun 函式中讓此 Thread 放置於適當的 ready list。值得注意一點的事，為了讓 itreator 能繼續遍歷，我先將此 thread 記錄起來，指向下一個元素後，最後才刪除該元素，避免 iterator 指向 NULL。

[311]指向下一個元素。

2-2.

kernel.cc：實作命令”-ep”，同其他命令的參數設置。這邊我額外宣告了 execfilePriority[](kernel.h)用來存放相同 index 的 execfile[]的初始 priority，方便其他函數使用。

```
else if (strcmp(argv[i], "-ep") == 0)
{
    execfile[++execfileNum] = argv[++i];
    execfilePriority[execfileNum] = atoi(argv[++i]);
    execfilePriority[execfileNum] = execfilePriority[execfileNum] > 149 ? 149 : execfilePriority[execfileNum];
    execfilePriority[execfileNum] = execfilePriority[execfileNum] < 0 ? 0 : execfilePriority[execfileNum];
    //cout << "File name is: " << execfile[execfileNum] << " and its priority number is " << execfilePriority[ex
}
```

kernel.cc：調整 Exec()函式額外增加一個參數(該 execfile 的 priority)，並在 Thread 初始化時將 approximateBurstTime、waitingTime、executionTime 初始值設成 0。

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

```
int Kernel::Exec(char *name, int priorty)
{
    t[threadNum] = new Thread(name, threadNum);
    t[threadNum]->space = new AddrSpace(usedPhyMem); //有改 有改 有改
    /*周改的*/
    t[threadNum]->SetPriority(priorty);
    t[threadNum]->SetApproximateBurstTime(0);
    t[threadNum]->SetWaitingTime(0);
    t[threadNum]->SetExecutionTime(0);
    /*周改的*/
    t[threadNum]->Fork((VoidFunctionPtr)&ForkExecute, (void *)t[threadNum]);
    threadNum++;

    return threadNum - 1;
```

alarm.cc：最後是修改 alarm.cc 中的 CallBack()函式(固定時間會呼叫)，因此函式中可呼叫宣告好的 AdjustPriority()做 aging，並隨時間遞增此 Thread 的 ExecutionTime 以及 L3 ready list 的等待時間(RR 方式若等待時間超過 100 秒就可以 Yield)，最後做 L1 及 L3 的 Preemptive 判斷。

```
void Alarm::CallBack()
{
    Interrupt *interrupt = kernel->interrupt;
    MachineStatus status = interrupt->getStatus();

    /*周改的*/
    kernel->scheduler->AdjustPriority();

    Thread *currentThread = kernel->currentThread;
    currentThread->SetExecutionTime(currentThread->GetExecutionTime() + 1000);
    currentThread->SetL3waitingTime(currentThread->GetL3waitingTime() + 1000);

    if (status != IdleMode && kernel->currentThread->GetPriority() >= 100)
    {
        interrupt->YieldOnReturn();
    }

    if (status != IdleMode && kernel->currentThread->GetPriority() <= 49)
    {
        if (currentThread->GetL3waitingTime() >= 99)
            interrupt->YieldOnReturn();
    }
    /*周改的*/
}
```

2-3.

在 debug.h 中新增專屬的 debug char，並沿用 debug 資訊於(a)~(e)。

```
const char dbgList = 'z'; //周改的
```

(a) Whenever a process is inserted into a queue：

新增在 scheduler.cc 的 ReadyToRun()函式裡。

```
if (thread->CanInL1ReadyList())
{
    if (!L1ReadyList->IsInList(thread))
        L1ReadyList->Insert(thread);
    DEBUG(dbgList, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" << thread->getID() << "] is inserted into queue L[" << 1 << "]");
}
else if (thread->CanInL2ReadyList())
{
    if (!L2ReadyList->IsInList(thread))
        L2ReadyList->Insert(thread);
    DEBUG(dbgList, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" << thread->getID() << "] is inserted into queue L[" << 2 << "]");
}
else if (thread->CanInL3ReadyList())
{
    DEBUG(dbgList, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" << thread->getID() << "] is inserted into queue L[" << 3 << "]");
    if (!L3ReadyList->IsInList(thread))
    {
        L3ReadyList->Append(thread);
    }
```

(b) Whenever a process is removed from a queue：

新增在 scheduler.cc 的 FindNextToRun()函式裡。

```
if (!L1ReadyList->IsEmpty())
{
    DEBUG(dbgList, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" << L1ReadyList->Front()->getID() << "] is removed from queue L[" << 1 << "]");
    return L1ReadyList->RemoveFront();
}

else if (!L2ReadyList->IsEmpty())
{
    DEBUG(dbgList, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" << L2ReadyList->Front()->getID() << "] is removed from queue L[" << 2 << "]");
    return L2ReadyList->RemoveFront();
}

else if (!L3ReadyList->IsEmpty())
{
    DEBUG(dbgList, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" << L3ReadyList->Front()->getID() << "] is removed from queue L[" << 3 << "]");
    Thread *removeThread = L3ReadyList->RemoveFront();
    //Print();
    return removeThread;
}
```

(c) Whenever a process changes its scheduling priority：

新增在 scheduler.cc 的 AdjustPriority()函式裡，L2 及 L3 ready list 的遍歷也是。

```
while (!it1->IsDone())
{
    ASSERT(it1->Item()->getStatus() == READY);
    it1->Item()->SetWaitingTime(it1->Item()->GetWaitingTime() + 100);
    if (it1->Item()->GetWaitingTime() >= 1500)
    {
        int newPriority = it1->Item()->GetPriority() + 10;
        newPriority = newPriority > 149 ? 149 : newPriority;
        DEBUG(dbgList, "[C] Tick [" << kernel->stats->totalTicks << "]: Thread [" << it1->Item()->getID() <<
        it1->Item()->SetPriority(newPriority);
        it1->Item()->SetWaitingTime(0);
    }
    it1->Next();
}
```

(d) Whenever a process updates its approximate burst time：

新增在 thread.cc 的 Sleep()函式裡。

```
int originalApproximateBurstTime = this->GetApproximateBurstTime();
int executiontime = this->GetExecutionTime();
int newApproximateBurstTime = originalApproximateBurstTime * 0.5 + executiontime * 0.5;
this->SetApproximateBurstTime(newApproximateBurstTime);
DEBUG(dbgList, "[D] Tick [" << kernel->stats->totalTicks << "]: Thread [" << this->getID() << "] update approximate burst time, fro
```

(e) Whenever a context switch occurs：

新增在 scheduler.cc 的 Run()函式裡。

```
// of view of the thread and from the perspective of the "outside world".

DEBUG(dbgList, "[E] Tick [" << kernel->stats->totalTicks << "]: Thread [" << nextThread->getID() << "] is now selected for ex
SWITCH(oldThread, nextThread);

```