Operating System
MP3 Report

組別：25

陳凱揚：Trace code、Implement、Report

簡佩如：Trace code、Implement、Report

1. Trace code

1-1 New → Ready

(a) Kernel::ExecAll()、Kernel::Exec(char*)

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

(b) Thread::Fork(VoidFunctionPtr, void*)

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

(c) Thread::StackAllocate(VoidFunctionPtr, void*)

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

```
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;
```

▲ 圖 1-1

(d) Scheduler::ReadyToRun(Thread*)

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

1-2 Running →Ready

(a) Machine::Run()、Interrupt::OneTick()

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

(b) Thread::Yield()

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

(c) Scheduler::FindNextToRun()、Scheduler::ReadyToRun(Thread*)

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

(d) Scheduler::Run(Thread*, bool)

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

1-3 Running →Waiting

(a) SynchConsoleOutput::PutChar(char)

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

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

▲ 圖 1-2

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

▲ 圖 1-3

(b) Semaphore::P()

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

```
75 void
76 Semaphore::P()
77 {
78     DEBUG(dbgTraCode, "In Semaphore::P(), " << kernel->stats->tota
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 }
```

▲ 圖 1-4

(c) List<T>::Append(T)

在 Append()裡會將傳進的 thread*存到 list 的最後面。

(d) Thread::Sleep(bool)

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

(e) Scheduler::FindNextToRun()、Scheduler::Run(Thread*, bool)

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

1-4 Waiting →Ready

(a) Semaphore::V()

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

```
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 }
```

▲ 圖 1-5

(b) Scheduler::ReadyToRun(Thread*)

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

1-5 Running →Terminated

(a) ExceptionHandler(ExceptionType)

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

(b) Thread::Finish()

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

(c) Thread::Sleep(bool)

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

(d) Scheduler::FindNextToRun()、Scheduler::Run(Thread*, bool)

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

1-6 Ready →Running

(a) Scheduler::FindNextToRun()、Scheduler::Run(Thread*, bool)

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

(b) SWITCH(Thread*, Thread*)

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

```
327 /* void SWITCH( thread *t1, thread *t2 )
328 **
329 ** on entry, stack looks like this:
330 **      8(esp)  ->              thread *t2
331 **      4(esp)  ->              thread *t1
332 **       (esp)  ->              return address
```

▲ 圖 1-6

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

```
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
```

▲ 圖 1-7

(c) Depends on the previous process state

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

2. Implement

2-1 Multilevel feedback queue

(a) /threads/thread.h

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

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

```
133     // ********** MP3 ********** //
134     int priority;
135     double approximateBurstTime;
136     double totalBurstTime;
137     double remainingBurstTime;
138     double lastExecTime;
139     int startTick;
140     int startAgeTick;
141     int totalAgeTick;
142     void UpdateBurstTime(bool toReady);
143     // ********** MP3 ********** //
```

▲ 圖 2-1

```
110     // ********** MP3 ********** //
111     void setPriority(int prior) { priority = prior; }
112     int getPriority() { return priority; }
113     void setStartTick(int tick) { startTick = tick; }
114     void setStartAgeTick(int tick) { startAgeTick = tick; }
115     void setTotalAgeTick(int tick) { totalAgeTick = tick; }
116     double getRemainingBurstTime() { return remainingBurstTime; }
117     double getLastExecTime() { return lastExecTime; }
118     int getLevel();
119     void UpdateAgeTick();
120     void UpdatePriority();
121     // ********** MP3 ********** //
```

▲ 圖 2-2

(b) /threads/thread.cc

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

```
274 // ********** MP3 ********** //
275 int
276 Thread::getLevel()
277 {
278     ASSERT(priority>=0 && priority<=149);
279     if(priority >= 100){
280         return 1;
281     }
282     else if(priority >= 50){
283         return 2;
284     }
285     else{
286         return 3;
287     }
288 }
289
290 void
291 Thread::UpdateAgeTick()
292 {
293     totalAgeTick += (kernel->stats->totalTicks-startAgeTick);
294 }
295
```

▲ 圖 2-3

```
296 void
297 Thread::UpdatePriority()
298 {
299     if(priority!=149 && totalAgeTick>=1500){
300         int oldPriority = priority;
301         priority = min(149, priority+10);
302         DEBUG(dbgSche, "[C] Tick[" << kernel->stats->totalTicks << "]: Threa
303             << oldPriority << "] to [" << priority << "]");
304         totalAgeTick -= 1500;
305     }
306 }
307
308 void
309 Thread::UpdateBurstTime(bool toReady)
310 {
311     lastExecTime = (double)(kernel->stats->totalTicks-startTick);
312     // runnung -> ready
313     if(toReady){
314         totalBurstTime += lastExecTime;
315         remainingBurstTime = max(0.0, approximateBurstTime-totalBurstTime);
316     }
317     // runnung -> waiting
318     else{
319         double oldApproximateBurstTime = approximateBurstTime;
320         totalBurstTime += lastExecTime;
321         approximateBurstTime = 0.5*totalBurstTime+0.5*approximateBurstTime;
322         DEBUG(dbgSche, "[D] Tick[" << kernel->stats->totalTicks << "]: Threa
323             << oldApproximateBurstTime << "], add [" << approximateBurstTime
324         totalBurstTime = 0;
325         remainingBurstTime = approximateBurstTime;
326     }
327 }
328 // ********** MP3 ********** //
```

▲ 圖 2-4

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

```
49      // ********** MP3 ********** //
50      priority = 0;
51      approximateBurstTime = 0.0;
52      totalBurstTime = 0.0;
53      remainingBurstTime = 0.0;
54      lastExecTime = 0.0;
55      startTick = 0;
56      startAgeTick = 0;
57      totalAgeTick = 0;
58      // ********** MP3 ********** //
```

▲ 圖 2-5

```
220     // ********** MP3 ********** //
221     UpdateBurstTime(TRUE);
222     kernel->scheduler->ReadyToRun(this);
223     nextThread = kernel->scheduler->FindNextToRun();
224     if (nextThread != this) {
225         kernel->scheduler->Run(nextThread, FALSE);
226     }
227     // ********** MP3 ********** //
```

▲ 圖 2-6

```
262     status = BLOCKED;
263     // ********** MP3 ********** //
264     UpdateBurstTime(FALSE);
265     // ********** MP3 ********** //
266     //cout << "debug Thread::Sleep " << name << "wait for Idle\n";
267     while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
268         kernel->interrupt->Idle();  // no one to run, wait for an interrupt
269     }
270     // returns when it's time for us to run
271     kernel->scheduler->Run(nextThread, finishing);
```

▲ 圖 2-7

(c) /threads/scheduler.h

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

```
37      // ********** MP3 ********** //
38      bool IsL1Empty() { return L1->IsEmpty(); }
39      void UpdateAging();
40      // ********** MP3 ********** //
41    private:
42      // queue of threads that are ready to run, but not running
43      // ********** MP3 ********** //
44      List<Thread*> *L1;
45      List<Thread*> *L2;
46      List<Thread*> *L3;
47      void UpdateAging(List<Thread*> *list, int level);
48      void AppendToQueue(List<Thread*> *list, Thread *thread, int level);
49      Thread* RemoveFromQueue(List<Thread*> *list, Thread *thread, int level);
50      // ********** MP3 ********** //
```

▲ 圖 2-8

(d) /threads/scheduler.cc

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

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

```
206 // ********** MP3 ********** //
207 void
208 Scheduler::UpdateAging()
209 {
210     UpdateAging(L1, 1);
211     UpdateAging(L2, 2);
212     UpdateAging(L3, 3);
213 }
214
215 void
216 Scheduler::UpdateAging(List<Thread*> *list, int level)
217 {
218     ListIterator<Thread*> iter(list);
219     while(!iter.IsDone()){
220         Thread *thread = iter.Item();
221         thread->UpdateAgeTick();
222         thread->setStartAgeTick(kernel->stats->totalTicks);
223         thread->UpdatePriority();
224         iter.Next();
225         int priority = thread->getPriority();
226         if(level==2 && priority>=100){
227             RemoveFromQueue(L2, thread, 2);
228             AppendToQueue(L1, thread, 1);
229         }
230         else if(level==3 && priority>=50){
231             RemoveFromQueue(L3, thread, 3);
232             AppendToQueue(L2, thread, 2);
233         }
234     }
235 }
```

▲ 圖 2-9

```
237 void
238 Scheduler::AppendToQueue(List<Thread*> *list, Thread *thread, int level)
239 {
240     DEBUG(dbgSche, "[A] Tick[" << kernel->stats->totalTicks << "]: Thread
241     list->Append(thread);
242 }
243
244 Thread*
245 Scheduler::RemoveFromQueue(List<Thread*> *list, Thread *thread, int level)
246 {
247     DEBUG(dbgSche, "[B] Tick[" << kernel->stats->totalTicks << "]: Thread
248     list->Remove(thread);
249     return thread;
250 }
251 // ********** MP3 ********** //
```

▲ 圖 2-10

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

```
34      // ********** MP3 ********** //
35      L1 = new List<Thread*>;
36      L2 = new List<Thread*>;
37      L3 = new List<Thread*>;
38      // ********** MP3 ********** //
```

▲ 圖 2-11

```
49      // ********** MP3 ********** //
50      delete L1;
51      delete L2;
52      delete L3;
53      // ********** MP3 ********** //
```

▲ 圖 2-12

```
275 void
276 Scheduler::Print()
277 {
278     // ********** MP3 ********** //
279     cout << "L1 contents:\n";
280     L1->Apply(ThreadPrint);
281     cout << "L2 contents:\n";
282     L2->Apply(ThreadPrint);
283     cout << "L3 contents:\n";
284     L3->Apply(ThreadPrint);
285     // ********** MP3 ********** //
286 }
```

▲ 圖 2-13

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

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

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

```
64 void
65 Scheduler::ReadyToRun (Thread *thread)
66 {
67     ASSERT(kernel->interrupt->getLevel() == IntOff);
68     DEBUG(dbgThread, "Putting thread on ready list: " <<
69     //cout << "Putting thread on ready list: " << thread
70     thread->setStatus(READY);
71     // ********** MP3 ********** //
72     int priority = thread->getPriority();
73     if(priority >= 100){
74         AppendToQueue(L1, thread, 1);
75     }
76     else if(priority >= 50){
77         AppendToQueue(L2, thread, 2);
78     }
79     else{
80         AppendToQueue(L3, thread, 3);
81     }
82     thread->setStartAgeTick(kernel->stats->totalTicks);
83     thread->setTotalAgeTick(0);
84     // ********** MP3 ********** //
85 }
```

▲ 圖 2-14

```
96 Scheduler::FindNextToRun ()
97 {
98     ASSERT(kernel->interrupt->getLevel() == IntOff);
99
100    // ********** MP3 ********** //
101    if(!L1->IsEmpty()){
102        ListIterator<Thread*> iter(L1);
103        Thread *resThread = NULL;
104        for(resThread = iter.Item(); !iter.IsDone(); iter.Next()){
105            Thread *nowThread = iter.Item();
106            if(nowThread->getRemainingBurstTime() < resThread->getRemainingBurstTime()){
107                resThread = nowThread;
108            }
109        }
110        return RemoveFromQueue(L1, resThread, 1);
111    }
112    else if(!L2->IsEmpty()){
113        ListIterator<Thread*> iter(L2);
114        Thread *resThread = NULL;
115        for(resThread = iter.Item(); !iter.IsDone(); iter.Next()){
116            Thread *nowThread = iter.Item();
117            if(nowThread->getPriority() > resThread->getPriority()){
118                resThread = nowThread;
119            }
120        }
121        return RemoveFromQueue(L2, resThread, 2);
122    }
123    else if(!L3->IsEmpty()){
124        return RemoveFromQueue(L3, L3->Front(), 3);
125    }
126    else{
127        return NULL;
128    }
129    // ********** MP3 ********** //
```

▲ 圖 2-15

```
179     // ********** MP3 ********** //
180     nextThread->setStartTick(kernel->stats->totalTicks);
181     DEBUG(dbgSche, "[E] Tick[" << kernel->stats->totalTic
182             << oldThread->getID() << "] is replaced, and
183
184     SWITCH(oldThread, nextThread);
185
186     oldThread->setStartTick(kernel->stats->totalTicks);
187     // ********** MP3 ********** //
```

▲ 圖 2-16

(e) /threads/alarm.cc

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

```
47 Alarm::CallBack()
48 {
49     Interrupt *interrupt = kernel->interrupt;
50     MachineStatus status = interrupt->getStatus();
51     // ********** MP3 ********** //
52     kernel->scheduler->UpdateAging();
53     if (status != IdleMode) {
54         int level = kernel->currentThread->getLevel();
55         bool isL1Empty = kernel->scheduler->IsL1Empty();
56         if(level==1 || (level==2 && !isL1Empty) || level==3){
57             interrupt->YieldOnReturn();
58         }
59     }
60     // ********** MP3 ********** //
```

▲ 圖 2-17

2-2 Command line argument “-ep”

(a) /threads/kernel.h

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

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

▲ 圖 2-18

```
85      // ********** MP3 **
86      int priorities[10];
87      // ********** MP3 **
```

▲ 圖 2-19

(b) /threads/kernel.cc

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

```
55      // ********** MP3 ********** //
56      } else if (strcmp(argv[i], "-e") == 0) {
57          ASSERT(i+1 < argc);
58          execfile[++execfileNum] = argv[++i];
59          priorities[execfileNum] = 0;
60          cout << execfile[execfileNum] << "\n";
61      } else if (strcmp(argv[i], "-ep") == 0) {
62          ASSERT(i+2 < argc);
63          execfile[++execfileNum] = argv[++i];
64          int priority = atoi(argv[++i]);
65          ASSERT(priority>=0 && priority<=149);
66          priorities[execfileNum] = priority;
67          cout << execfile[execfileNum] << "\n";
68      // ********** MP3 ********** //
```

▲ 圖 2-20

```
298     t[threadNum] = new Thread(name, threadNum);
299     t[threadNum]->space = new AddrSpace();
300     t[threadNum]->setPriority(priority);
301 // ********** MP3 ********** //
```

▲ 圖 2-21

2-3 Dubugging flag z

(a) /lib/debug.h

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

[A]：Scheduler::AppendToQueue()

[B]：Scheduler::RemoverFromQueue()

[C]：Thread::UpdatePriority()

[D]：Thread::UpdateBurstTime()

[E]：Scheduler::Run()

```
33 // ********** MP3 ********** //
34 const char dbgSche = 'z';  // process schedule
35 // ********** MP3 ********** //
```

▲ 圖 2-22

3. Difficulties and Feedback

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