Operating System

MP3: CPU scheduling

Team Member(Team 10): 林子謙 (112065517)、陳奕潔 (110000212)

Contributions: Both contributed to CPU scheduler implementation and report writing.

Contents

1 Code Tracing

本次作業中，需要 trace 的函式大多在 MP1、MP2 解釋過。因此，除了前二次作業未提及的函式會較為詳細說明，其他函式將只概述，並嘗試以 thread scheduling 的角度解釋。

1.1 New → Ready

1.1.1 Kernel::ExecAll()

`Kernel::ExecAll()` 是 main thread 用來將所有 user program 建立各 threads 的函式。當所有 user program 都初始化後，main thread 的工作結束，因此呼叫 `Thread::Finish()`。

1.1.2 Kernel::Exec(char*)

`Kernel::Exec()` 透過每一 user program 的檔案名稱建立其 thread，包含初始化其 thread control block、allocate a memory space（但在 MP2 支援 mutil-programming 後，這時候還沒有 allocate，而是在 `AddrSpace::Load()` 時才 allocate）、allocate 和初始化其 stack。

1.1.3 Thread::Fork(VoidFunctionPtr, void*)

`Kernel::Fork()` 負責初始化該 thread 的 stack 以及將此 thread 變為 `READY`。

此函式 function address 及其參數傳給 Thread::StackAllocate() 做 stack 的初始化；將在 disable interrupts 後，該函式呼叫 `Scheduler::ReadyToRun()`，將此 thread 放入 ready queue。最後，重新 enable interrupts 後便 return。

1.1.4 Thread::StackAllocate(VoidFunctionPtr, void*)

`Thread::StackAllocate()` 是真正初始化 stack 的函式。此函式定義好 stack pointer（`stackTop`）後，將 `ThreadRoot`、`ThreadBegin`、傳入的 function address 及其參數，與 `ThreadBegin` 放入 stack。

1.1.5 Scheduler::ReadtToRun(Thread*)

`Scheduler::ReadtToRun()` 是將傳入的 thread 之 state 從 `JUST_CREATED` 改成 `READY` 的地方。更新完 thread 的 state 後，此函式把該 thread 從 job queue 丟入 ready queue。

1.1.6 Brief summary

透過 `Kernel::ExecAll()`，每一個 user program 可以建立相對應的 thread。當一 thread 的 state 從 `JUST_CREATED` 轉為 `READY`，代表有足夠的資源讓它能被放入 memory 並隨時等待著被 CPU 執行。這個過程中，需要 `Kernel::Exec()` 做各類的初始化，`Scheduler::ReadtToRun()` 更新其 state 及所屬的 queue。

1.2 Running → Ready

1.2.1 Machine::Run()

`Machine::Run()` 是模擬 NachOS 執行 user thread 的函式，會無限 fetch instructions。此函式每透過 `OneInstruction()` 執行一次 instruction，就會透過 `Interrupt::OneTick()` 更新一次 machine 的 ticks。

1.2.2 Interrupt::OneTick()

`Interrupt::OneTick()` 是用來推進 machine 的時間。它會根據 machine 目前所處的 mode 決定推進多少時間。而且，因為時間的遞移，它會檢查是否有 interrupts（呼叫 `CheckIfDue()`）或 timer（利用 flag `yieldOnReturn == TRUE`）到期。

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

(a) Alarm::CallBack()

```
185 void Interrupt::YieldOnReturn() {
186     ASSERT(inHandler == TRUE);
187     yieldOnReturn = TRUE;
188 }
```

(b) Interrupt::YieldOnReturn()

Figure 1: Context Switch 發生的條件

在 running → ready 的情況下，`Alarm::CallBack()` 這個 timer 到期的 interrupt 會被 `CheckIfDue()` 觸發（如 Figure 29 所示）。`CheckIfDue()` 會呼叫 `Interrupt::YieldOnReturn()`，`Interrupt::YieldOnReturn()` 將 `yieldOnReturn` 設為 `TRUE`（如 Figure 1b 所示）。這一系列流程，使 `Interrupt::OneTick()` 能進入 `if (yieldOnReturn)` 的條件式，觸發 `Thread::Yield()`。

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

Figure 2: Alarm::Alarm()

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

(a) Timer::Timer()

```
67  void Timer::SetInterrupt() {
68      if (!disable) {
69          int delay = TimerTicks;
70
71          if (randomize) {
72              delay = 1 + (RandomNumber() % (TimerTicks * 2));
73          }
74          // schedule the next timer device interrupt
75          kernel->interrupt->Schedule(this, delay, TimerInt);
76      }
77  }
```

(b) Timer::SetInterrupt()

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

(c) Timer::CallBack()

Figure 3: Timer() 及其 method

而 Alarm 是一個 Timer object，如 Figure 2 所示。

Timer 的建構子 (如 Figure 3a 所示) 中有觸發 SetInterrupt() method。在目前的設定下，SetInterrupt() 負責設定在 100 ticks（目前 TimerTicks 為 100）之後到期的 timer interrupt，如 Figure 3b 所示。

Timer 另一 method Timer::CallBack() 負責執行 CallBack() 裡的程式碼，並再次觸發 SetInterrupt() method，如 Figure 3c 所示。

因此，當 Alarm 被建構時，就會有一個 100 ticks 後到期的 timer interrupt；當此 interrupt 到期，會執行 Alarm::CallBack() 裡的程式碼，並再次設定一個新的 100 ticks 後到期的 timer interrupt。如此循環往復，直到 machine 停止。

1.2.3 Thread::Yield()

Thread::Yield() 是 timer 到期時，負責 context switch 的函式，如 Figure 35 所示。

在禁止 interrupts 後，它會呼叫 Scheduler::FindNextToRun()，拿到應該接著被執行的 thread，nextThread。若 ready queue 為空，它會重新 enable interrupts 並返回；否則將目前正在執行的 thread 用 Scheduler::ReadyToRun() 丟回 ready queue，呼叫 Scheduler::Run() 做 context switch，以執行 nextThread。

1.2.4 Scheduler::FindNextToRun()

Scheduler::FindNextToRun() 負責找到 ready queue 裡的第一個 thread，如 Figure 24 所示。

若 ready queue 為空，它會回傳 NULL；否則透過 wating queue 的 method，RemoveFront()，取得第一個 thread。

```
202  void Thread::Yield() {
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  }
```

Figure 4: Thread::Yield()

```
69  Thread *Scheduler::FindNextToRun() {
70      ASSERT(kernel->interrupt->getLevel() == IntOff);
71
72      if (readyList->IsEmpty()) {
73          return NULL;
74      } else {
75          return readyList->RemoveFront();
76      }
77  }
```

Figure 5: Scheduler::FindNextToRun()

1.2.5 Scheduler::ReadyToRun(Thread*)

在此 Scheduler::ReadyToRun() 則將傳入的 thread 之 state 從 RUNNING 改為 READY，其餘如上所述。

1.2.6 Scheduler::Run(Thread*, bool)

Scheduler::Run() 是專責 context switch 的函式。

它首先檢查傳入的 thread 是否已完成，由於此時傳入的 finishing 為 FALSE，因此不會更新 toBeDestroyed。

接著，它會儲存要被切掉的 thread 的資料，並做 context switch：更新 kernel->currentThread、把 nextThread 的 state 設為 RUNNING，及呼叫 SWITCH() 做真正的 context switch。

若 context switch 完的 thread 不是第一次執行，即此 thread 有被執行過 context switch，則函式會透過 CheckToBeDestroyed() 刪除 toBeDestroyed（如果是第一次執行，會在 Thread::Begin() 去呼叫 CheckToBeDestroyed()），並會重新抓取該 thread 的資料以繼續執行。

1.2.7 Brief summary

在 Machine::Run() 裡 for 會無限迴圈的狀態下，for 中呼叫到的 Interrupt::OneTick()，使 machine 的時間得以持續推進。以目前的設定下，每 100 ticks，timer 到期，強制目前正在執行的 thread 釋放 CPU 給下一個應該被執行的 thread——CheckIfDue() 檢查到 Alarm::CallBack() 到期而執行此 interrupt，間接觸發 Thread::Yield()，與此同時 Alarm::CallBack() 又會新增一個 timer interrupt。Thread::Yield() 會呼叫一系列 context switch 前的準備工作及執行 context switch 之相關函式，完成一次 context switch。

1.3 Running → Waiting

1.3.1 SynchConsoleOutput::PutChar(char)

```
91  void SynchConsoleOutput::PutChar(char ch) {
92      lock->Acquire();
93      consoleOutput->PutChar(ch);
94      waitFor->P();
95      lock->Release();
96  }
```

Figure 6: SynchConsoleOutput::PutChar()

SynchConsoleOutput::PutChar() 是負責將被傳入的字元印到 console 的函式，如 Figure 6 所示。

```
173  void Lock::Acquire() {
174      semaphore->P();
175      lockHolder = kernel->currentThread;
176  }
```

(a) Lock::Acquire()

```
189  void Lock::Release() {
190      ASSERT(IsHeldByCurrentThread());
191      lockHolder = NULL;
192      semaphore->V();
193  }
```

(b) Lock::Release()

Figure 7: Lock() 的取用與歸還

為了確保印的過程不會被打斷，它會透過 Lock::Acquire() 向系統要求 lock，等到印完才透過 Lock::Release() 還回去。這兩個函式分別透過 Semaphore::P() 及 Semaphore::V() 取要或釋放 lock，如 Figure 7a 及 7b。另外，因為印字元需要用到 console 這一資源，此函式也會透過 Semaphore::P() 向系統拿取資源。

1.3.2 Semophore::P()

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

Figure 8: Semophore::P()

Semophore::P() 是管理是否給予資源的函式，如 Figure 8 所示。

由於這個函式是 atomic，在禁止 interrupts 後，它才會確認資源的剩餘量。若可用資源數為零，代表所需的資源都被佔用，它會將目前正在執

行的 thread 丟到 device queue 裡，並呼叫 Thread::Sleep() 跟下一個可以被執行的 thread 做 context switch；否則將資源減去一個，並重新 enable interrupts。

在 running → waiting 的情況下，代表可用資源數為零，因此此函式會執行前者的情形。

1.3.3 List<T>::Append(T)

```
68  template <class T>
69  void List<T>::Append(T item) {
70      ListElement<T> *element = new ListElement<T>(item);
71
72      ASSERT(!IsInList(item));
73      if (IsEmpty()) {  // list is empty
74          first = element;
75          last = element;
76      } else {  // else put it after last
77          last->next = element;
78          last = element;
79      }
80      numInList++;
81      ASSERT(IsInList(item));
82  }
```

Figure 9: List<T>::Append()

List<T>::Append() 是將傳入的 item 放到該 list object（/lib 裡實作好的資料結構，有 List、SortedList 等類型）的最尾端，如 Figure 9 所示。

此函式首先為傳入的 itme 建立相對應的資料型態。確認此元素不在 list 裡後，接著檢查要放入的 list 是否為空。若為空，則將 List 中 first 及 last attribute 都指向這個新元素；否則，更新 last->next 及 last。最後將 List 的元素數加一，並確保此元素確實放入 list 後就返回。

1.3.4 Thread::Sleep(bool)

Thread::Sleep() 是將目前正在執行的 thread 的 state 從 RUNNING 改為 BLOCKED，並完成一次 context switch。若 ready queue 裡沒有一個等待被執行的 thread（nextThread 是 NULL），則 machine 會進入 idle，等待下一個

interrupt 產生；否則，會將下一個可以被執行的 thread 及接收到 finishing（目前正在執行的 thread 的執行完成與否）接續傳給 Scheduler::Run()。

在 Running → Waiting 的情形下，此函數被傳入的 finishing 為 FALSE。若 nextThread 是 NULL，代表 machine 會等待目前正在執行的 thread 所等待的 I/O or event 到達再繼續執行此 thread；若 nextThread 不是 NULL，則 nextThread 與 finishing 被傳給 Scheduler::Run()。

1.3.5 Scheduler::FindNextToRun()

Scheduler::FindNextToRun() 如上所述。

1.3.6 Scheduler::Run(Thread*, bool)

Scheduler::Run() 的行為也如上所述。

1.3.7 Brief summary

由於 SynchConsoleOutput::PutChar() 需要確保印字元的過程中不被打斷而要求 lock 及拿到 console 這個資源，會間接或直接呼叫到 Semophore::P()（歸還 lock 則間接呼叫 Semophore::V()）。在 Running → Waiting 的情形下，由於要求的資源不能馬上獲得，Semophore::P() 使 thread 被丟入 device queue 並強制睡眠，等到資源被歸還。

1.4 Waiting → Ready

1.4.1 Semaphore::V()

Semaphore::V() 與 Semophore::P() 相反，是管理資源歸還的函式，如 Figure 30 所示。

這個函式同樣是 atomic。在禁止 interrupts 後，它才確認 device queue 是否有 thread（有無正在等資源的 thread）。若有，則將此 queue 的第一個 thread 重新放入 ready queue。最後將資源數加一，並重新 enable interrupts。

1.4.2 Scheduler::ReadyToRun(Thread*)

在此 Scheduler::ReadyToRun() 則將傳入的 thread 之 state 從 BLOCKED 改為 READY，其餘如上所述。

1.4.3 Brief summary

當目前正在執行的 thread 歸還資源時（如上述的 SynchConsoleOutput::PutChar() 或一些 callback 函式），會間接或直接呼叫到 Semaphore::V()，使最先等待此資源的 thread 能從 waiting state 重新回到 ready state。

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

Figure 10: Semaphore::V()

1.5 Running → Terminated

1.5.1 ExceptionHandler(ExceptionType) case SC_Exit

```
276                 case SC_Exit:
277                     DEBUG(dbgAddr, "Program exit\n");
278                     val = kernel->machine->ReadRegister(4);
279                     cout << "return value:" << val << endl;
280                     kernel->currentThread->Finish();
281                     break;
```

Figure 11: ExceptionHandler(ExceptionType) case SC_Exit

ExceptionHandler(ExceptionType) case SC_Exit 是 raise 正在執行的 thread 執行完畢的 exception，如 Figure 11 所示。

此函式藉由傳入的 ExceptionType 去判斷是哪一 exception 發生，並透過 r2 拿到這個 exception 是哪一子類型。在這個情況中，exception type 是 SyscallException，且屬於 SC_Exit case。在這個 case，函式會去讀取 r4 的值，在 terminal 印出相應的資訊之後就呼叫 Thread::Finish()。

1.5.2 Thread::Finish()

Thread::Finish() 是一個 thread 執行完畢會呼叫的函式。如果所有 thread 都執行完畢（kernel->execExit 是 TRUE），則 machine 會停止並關閉；否則，它會傳入 finishing 為 TRUE 的值給 Thread::Sleep()。

1.5.3 Thread::Sleep(bool)

在 Running → Terminated 的情形下，Thread::Sleep() 將目前正在執行的 thread 的 state 從 RUNNING 改為 BLOCKED，且被傳入的值為 TRUE。其餘行為如上所述。

1.5.4 Scheduler::FindNextToRun()

Scheduler::FindNextToRun() 如上所述。

1.5.5 Scheduler::Run(Thread*, bool)

在 Running → Terminated 的情形下，finishing 的值為 TRUE，因此 toBeDestroyed 會被更新為目前正在執行的 thread。其餘行為如上所述。

1.5.6 Brief summary

當一個 user program 被讀取到 return 0 時，代表該 thread 要結束執行。return 0 會呼叫停止執行 thread 的 system call，也就是觸發 ExceptionHandler(ExceptionType case SC_Exit。

ExceptionHandler(ExceptionType) case SC_Exit 所呼叫的 Thread::Finish()，會傳入 finishing 為 TRUE 的值給 Thread::Sleep()，讓目前正在執行的 thread 被 toBeDestroyed 指向，藉此刪除這個執行完成的 thread。

由此可知，Thread::Sleep() 的傳入值為 FALSE 時，會實現 Running → Ready 或 Running → Waiting 的 context switch；若傳入值為 TRUE 時，則會實現 Running → Terminated 的 context switch。

1.6 Ready → Running

1.6.1 Scheduler::FindNextToRun()

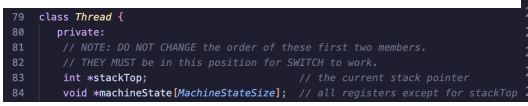
Scheduler::FindNextToRun() 如上所述。

1.6.2 Scheduler::Run(Thread*, bool)

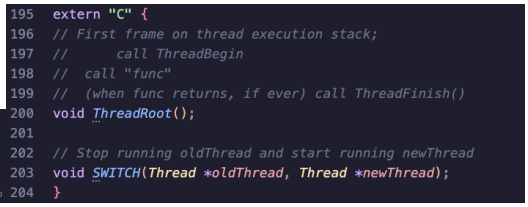
依據被傳入的 finishing 會有所不同，若 finishing 的值為 FALSE，可參照 1.2.6，若值為 TRUE，可參照 1.5.5。

1.6.3 SWITCH(Thread*, Thread*)

SWITCH() 是真正實作 context switch 的地方，以下說明相關的程式碼。



(a) 在 thread.h 宣告 thread 的 stack



(b) 在 thread.h 用 extern 宣告 SWITCH()

Figure 12: thread.h 中 SWITCH() 相關變數宣告

在 1.1.4 提到 stack 的 allocation 與初始化所用到的變數會在這邊宣告，`stackTop` 會指向 `ThreadRoot`。由此可知，`stackTop` 跟 `machineState` 都屬於 thread 的 stack 範圍。

此外，`SWITCH()` 在 thread.h 做外部宣告，讓其位於 switch.S 中的定義能在 thread.h/thread.cc 被調用，如 Figure 12b 所示。

在 switch.h 中，有先定義好 switch.S 會需要用到的 offsets，如 Figure 13 所示。

在 switch.h 中，註解說明 `thread *t1`（即 `oldThread`）存在 `4(esp)`，`thread *t2`（即 `newThread`）存在 `8(esp)`，`return address` 則存在 `esp`。

由此可知，`4(esp)` 會存取到 `stackTop`，每 +4 就會依序存取到 `machineState` 裡的一個元素。此外，它有定義 x86 所使用的變數，如 Figure 14 所示。其中，之所以要定義 `SWITCH` 及 `_SWITCH`，是因為不同的系統規範會對變數的符號有特定規範，不符合就無法識別到此變數。

`SWITCH` 在把 `oldThread` 存到 general purpose register `eax` 之前，會先把 `eax` 所指到的 address space 內容存到 `_eax_save` 所指到的 memory，如 Figure 15 所示。

接著，此函式把 `oldThread` 的 address、使用到的 registers、stack pointer，與 return address 存到相對應的 `machineState` 位置。其中有使用到 `ebx` 的原因是，x86 沒有直接複製 memory space 裡的資料至另一 memory space 的指令，因此需要 register 做中介來搬移資料。

儲存好 `oldThread` 的資訊後，`eax` 會指到 `newThread` 的位置，以同樣的邏輯把 `newThread` 的資料及資訊搬回到相應的 registers 裡。最後返回 caller。

1.6.4 Previous Process State

- **New → Ready → Running**

  如 Figuer 16 所示，如果 `newThread` 還沒被執行，會從第 313 行開始執行，將其 function 用到的 argument 放到 thread，依序呼叫 `Thread::Begin()`、其 function（會呼叫 `ForkExecute()`、`Thread::Finish()`（期間可能會被 context swich）。

```
#ifdef x86

/* the offsets of the registers from the beginning of the thread object */
#define _ESP 0
#define _EAX 4
#define _EBX 8
#define _ECX 12
#define _EDX 16
#define _EBP 20
#define _ESI 24
#define _EDI 28
#define _PC 32

/* These definitions are used in Thread::AllocateStack(). */
#define PCState (_PC / 4 - 1)
#define FPState (_EBP / 4 - 1)
#define InitialPCState (_ESI / 4 - 1)
#define InitialArgState (_EDX / 4 - 1)
#define WhenDonePCState (_EDI / 4 - 1)
#define StartupPCState (_ECX / 4 - 1)

#define InitialPC %esi
#define InitialArg %edx
#define WhenDonePC %edi
#define StartupPC %ecx

#endif  // x86
```

Figure 13: 在 switch.h 中 #defind 的 offsets

```
/* void SWITCH( thread *t1, thread *t2 )
**
** on entry, stack looks like this:
**      8(esp)  ->              thread *t2
**      4(esp)  ->              thread *t1
**       (esp)  ->              return address
**
** we push the current eax on the stack so that we can use it as
** a pointer to t1, this decrements esp by 4, so when we use it
** to reference stuff on the stack, we add 4 to the offset.
*/
        .comm   _eax_save,4             /* assign the global variable, _eax_save, a memory with size 4 byte */# datatype?

        .globl  SWITCH
    .globl  _SWITCH
```

Figure 14: SWITCH() 所用到的變數

```
342  _SWITCH:
343  SWITCH:                                 /* the plain numbers are the offsets. */
344          movl    %eax,_eax_save          /* in order to use %eax as ptr to stack (save %eax value into memory)*/# save the value of eax (general purpose
345          movl    4(%esp),%eax            /* get t1 stack ptr (from memory to register)*/# move pointer to t1 into eax
346          movl    %ebx,_EBX(%eax)         /* save t1 regs' value onto the stack (into memory, according to offset) */# 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          /* Why don't we just save _eax_save to _EAX(%eax) (stack)? x86 doesn't support memory to memory movement. (The m
354          movl    %ebx,_EAX(%eax)         # store it
355          movl    0(%esp),%ebx            /* Why don't we just save 0(%esp) to _PC(%eax) (stack)? */# get return address from stack into ebx
356          movl    %ebx,_PC(%eax)          # save it into the pc storage
357
358          movl    8(%esp),%eax            /* get t2 stack ptr */# move pointer to t2 into eax
359
360          movl    _EAX(%eax),%ebx         /* move t2 value to current regs (from memory to reg)*/# 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
```

Figure 15: SWITCH() 的內容

```
295  #ifdef x86
296
297          .text
298          .align  2
299
300          .globl  ThreadRoot
301          .globl  _ThreadRoot
302
303  /* void ThreadRoot( void )
304  **
305  ** expects the following registers to be initialized:
306  **      eax     points to startup function (interrupt enable)  [general purpose register]
307  **      edx     contains inital argument to thread function    [general purpose register]
308  **      esi     points to thread function                      [general purpose register]
309  **      edi     point to Thread::Finish()                      [general purpose register]
310  */
311  _ThreadRoot:
312  ThreadRoot:                     /* When did these regs be initialized? In Thread::StackAllocate(). (use register offset defined
313          pushl   %ebp            /* push original base pointer onto the stack */* esp (stack pointer) - 4 並將 ebp 的 32bit valu
314          movl    %esp,%ebp       /* move the value stored in the stack pointer into the base pointer */* move 32bit value from e
315          pushl   InitialArg      /* push initial argument to the thread, %edx, onto the stack */* esp (stack pointer) - 4 並將 T
316          call    *StartupPC      /* mark return address (PC + 4) and go to StartupPC, %ecx (defined in switch.h, Thread::Begin())
317          call    *InitialPC      /* mark return address and go to InitialPC, %esi (defined in switch.h, thread function) */* PC
318          call    *WhenDonePC     /* mark return address and go to WhenDonePC, %edi (defined in switch.h, Thread::Finish()) */* P
319
320          # NOT REACHED
321          movl    %ebp,%esp       /* recover of line 314 */* 還原 */
322          popl    %ebp            /* recover of line 313 */* 原本的 PC, 並 + 4 */
323          ret                     /* return to %esp */* PC 指到 stack */
```

Figure 16: ThreadRoot in switch.S

其中，`Thread::Begin()` 會 load memory space，把 user program 檔案內的內容 load 進 memory 後，再跳到 `ForkExecute()`，`ForkExecute()` 裡的 `ForkExecute()` 會呼叫 `Machine::Run()()` 開始依次 fetch instructions。

- **Running → Ready → Running & Waiting → Ready → Running**

  如果 `newThread` 已經不是第一次被執行，則 `stackTop` 不再會指向 `ThreadRoot`，而是從上次被打斷的地方開始執行。

  在 Running → Ready 的情況下，`newThread` 是原本正在執行，卻因為 time sharing 機制被強迫釋出 CPU，在 `Thread::Yield()` 被觸發 context switch，因此 `newThread` 會直接被放入 ready queue，等待隨時被執行，詳細情形可參照 1.2.7。

  在 Waiting → Ready 的情況下，`newThread` 是原本是因為要不到所需資源而進到 device queue，在拿到資源後，才被重新放入 ready queue，等待被執行，詳細情形可參照 1.3.7。

1.6.5 Machine:Run()

`Machine:Run()` 的行為如上所述。

1.6.6 Brief summary

當一 thread 被放到 ready queue 後，若輪到它時（即 ready queue 排第一個的 thread），會被 `Scheduler::FindNextToRun()` 回傳，而被放入 `Scheduler::Run()` 的其中一個傳入值。在 `Scheduler::Run()` 裡，透過組合語言將目前正在執行的 thread 與下一個要被執行的 thread 來迅速的儲存與 restore 資料與資訊，完成 context switch。而目前正在執行的 thread 完成與否會如何的行為，可參照 1.6.2。

2 CPU Scheduler Implementation

2.1 分析：可能觸發 context switch 的條件

首先我們將會觸發 scheduler 檢查需不需要 context switch 的情形區分為以下 4 種：

1. **`Scheduler::ReadyToRun()`** → 有新的 process 被加進 ready queue，priority 比目前執行的 process 還高 (包含 aging rearrange 3 個 queue 中的 processes)。

2. **`Thread::Yield()`** → L3 Round Robin 執行時間到。

3. `Thread::Sleep()` → process 執行完畢主動讓出 CPU。(`Thread::Finish()`)

4. `Thread::Sleep()` → process 需要使用 I/O 資源，進入 device queue。

而這 4 種 scheduler 可能需要安排 context switch 的情形又分成 2 類：

1. Preemptive (scheduler 主動 context switch)：第 1、2 個情形屬此類。為 scheduler 依照各個 queue 中以及 queue 和 queue 之間的優先度主動去安排 context switch。

2. Non-Preemptive (scheduler 被動 context switch)：第 3、4 個情形屬此類。為 process 主動讓出 CPU 的情形。

第 1 類，preemptive，的情況下，又能夠分為兩種格局的 preemption 檢查：

1. queue 與 queue 之間的 preemption：priority 較高的 queue 有新的 process 被加入。

2. queue 內部的 preemption：現在執行的 queue 中有 priority 更高的 process 被加入。

因此我們將實作 non-preemptive 以及 preemptive 的情形下 scheduler 如何選擇下一個要執行的 process。而由於我們發現在 nachOS 中，只有兩個安全的函式能夠處理合法的 context switch，其中一個為 `Thread::Sleep()`，負責處理 non-preemptive 的 context switch，另一個為 `Thread::Yield()`，負責處理 preemptive 的 context switch，因此我們會從這兩個函式著手去修改成 multi-level feedback queue 的 scheduling。

2.2 分析：需要的時間變數 & 更新時間變數的時機

以下為一個 process 在 muti-level feedback queue 中需要的時間變數。

1. **curr_approximatedBurstTime：** t_i

2. **last_approximatedBurstTime：** t_{i-1}

3. **rem_approximatedBurstTime：** $t_i - T$

4. **timeStamp_startRunning：**process 進入 running state 開始使用 CPU 的時間。

5. **timeStamp_startReady：**process 進入 ready state 等待被執行的時間。

6. **totalRunningTime：**process 在 running state 中所花費的總時間。

更新變數的時機會在 process 確定需要轉換 state 時，包含：

1. **New → Ready**
2. **Running → Ready**
3. **Running → Waiting**
4. **Waiting → Ready**
5. **Running → Terminated**
6. **Ready → Running**

2.3 實作：Command line －參數設定

為了能夠在 command line 輸入欲執行程式的 priority，我們需要先更改 `kernel.cc` 和 `kernel.h` 的部分程式碼。

首先，我們需要一個能夠紀錄程式相對應 priority 的陣列，因此我們在 `kernel.h` 中為 Kernel class 宣告了 `int threadPriority[10]`，以 index 為 process ID 紀錄 process priority，如 Figure 17a所示。

```
30  class Kernel {
77    private:
78      Thread *t[10];
79      char *execfile[10];
80      int threadPriority[10];
81      int execfileNum;
82      int threadNum;
83      bool randomSlice;    // enable pseudo-random time slicing
84      bool debugUserProg;  // single step user program
85      double reliability;  // likelihood messages are dropped
86      char *consoleIn;     // file to read console input from
87      char *consoleOut;    // file to send console output to
88  #ifndef FILESYS_STUB
89      bool formatFlag;  // format the disk if this is true
90  #endif
91  };
```

(a) Kernel Class - `threadPriority` array

```
28  Kernel::Kernel(int argc, char **argv) {
29      randomSlice = FALSE;
30      debugUserProg = FALSE;
31      execExit = FALSE;
32      consoleIn = NULL;   // default is stdin
33      consoleOut = NULL;  // default is stdout
34      frameTable[NumPhysPages] = {0};
35      NumFreeFrame = NumPhysPages;
36      threadPriority[10] = {0};
37  #ifndef FILESYS_STUB
38      formatFlag = FALSE;
39  #endif
40      reliability = 1;  // network reliability, default is 1.0
41      hostName = 0;     // machine id, also UNIX socket name
42                        // 0 is the default machine id
43      for (int i = 1; i < argc; i++) {
44          if (strcmp(argv[i], "-ep") == 0) {
45              execfile[++execfileNum] = argv[++i];
46              threadPriority[execfileNum] = atoi(argv[++i]);
47          } else if (strcmp(argv[i], "-rs") == 0) {
48              ASSERT(i + 1 < argc);
49              RandomInit(atoi(argv[i + 1]));  // initialize pseudo-random
50                                              // number generator
51              randomSlice = TRUE;
52              i++;
53          } else if (strcmp(argv[i], "-s") == 0) {
54              debugUserProg = TRUE;
```

(b) `Kernel::Kernel()` - `-ep` Command

Figure 17: Command line 參數設定

接著，在 `Kernel::Kernel()` 中，加上了能夠偵測`-ep` 指令，紀錄程式 priority 的程式碼，將 priority 儲存至 Kernel class 的 `threadPriority[10]` 陣列中，請參考 Figure 17b 中程式碼第 44～46 行。

2.4 實作：multilevel feedback queue －需要的基本宣告與前置作業

在將 nachOS 改造成 multilevel feedback queue 的形式上，有幾項需要的事前準備工作。

1. **宣告 L1、L2、L3 ready queue 指標在 Scheduler class 中。**請參考 Figure 18。

```
20  class Scheduler {
45    private:
46    //  List<Thread*>* readyList;  // queue of threads that are ready to run,
47    //                             // but not running
48
49      SortedList<Thread*>* L1;
50      SortedList<Thread*>* L2;
51      List<Thread*>* L3;
52    Thread* toBeDestroyed;  // finishing thread to be destroyed
53                            // by the next thread that runs
54  };
```

Figure 18: Scheduler Class - Queues' Pointer Declaration

L1、L2 是需要依照條件排序的，因此使用 `SortedList<Thread*>*`，而 L3 是單純的 round robin，因此使用 `List<Thread*>*` 即可。由於 `SortedList<Thread*>*` 需要一個用於排序依據的 compare 函式，所以我們分別為 L1 和 L2，寫了 `L1Cmp()` 和 `L2Cmp()`。請參考 Figure 19。

L1 是基於 `rem_approximatedBurstTime` 為優先度，也就是 $t_i - T$ 去排序，而 L2 則是以 process 本身的 priority 為標準。在這裡我們運用 `list.cc` 中 `Insert()` 的規則，若新加入的 process 優先度較大，則回傳 -1，若相同則比較 process ID (process ID 較小者為優先)，若優先度較小則回傳 1。

2. **建立和釋放 L1、L2、L3 的空間。**請參考 Figure 20a、Figure 20b。

   我們讓 L1、L2、L3 queue 隨著 Scheduler object 一起建立和刪除。

3. **宣告變數，紀錄 priority 和所屬 queue 資訊在 Thread class 裡。**請參考 Figure 21。

   Scheduler 要幫忙依照 process 的 priority 安排適當的 queue，並且由於改造成 multiple queue 的形式後，每一個 queue 有自己的 scheduling 規則。因此我們宣告了 `priority` 和 `inWhichQueue`，讓 process 能夠記錄自己的 priority 以及被安排在哪個 queue，方便讓 scheduler 依照相對應的規則進行 scheduling。

```
446  static int L1Cmp(Thread *newThread, Thread *cmpThread) {
447      if (newThread->rem_approximatedBurstTime <
448          cmpThread->rem_approximatedBurstTime) {
449          return -1;
450      } else if (newThread->rem_approximatedBurstTime ==
451                 cmpThread->rem_approximatedBurstTime) {
452          if (newThread->getID() < cmpThread->getID()) {
453              return -1;
454          } else {
455              return 1;
456          }
457      } else {
458          return 1;
459      }
460  }
461
462  static int L2Cmp(Thread *newThread, Thread *cmpThread) {
463      if (newThread->priority > cmpThread->priority) {
464          return -1;
465      } else if (newThread->priority == cmpThread->priority) {
466          if (newThread->getID() < cmpThread->getID()) {
467              return -1;
468          } else {
469              return 1;
470          }
471      } else {
472          return 1;
473      }
474  }
```

Figure 19: Scheduler - Compare Function for L1 & L2

```
36  Scheduler::Scheduler() {
37      // readyList = new List<Thread *>;  // old version
38      L1 = new SortedList<Thread *>(L1Cmp);
39      L2 = new SortedList<Thread *>(L2Cmp);
40      L3 = new List<Thread *>;
41      toBeDestroyed = NULL;
42  }
```

(a) `Scheduler::Scheduler()` - Queues Initialization

```
49  Scheduler::~Scheduler() {
50      // delete readyList;  // old version
51      delete L1;
52      delete L2;
53      delete L3;
54  }
```

(b) `Scheduler::~Scheduler()` - Queues Deletion

Figure 20: Queues' Memory Space Assignment and Deletion

```
79  class Thread {
174
175     AddrSpace *space;  // User code this thread is running.
176
177 ····int·priority;·····//·priority·for·ready·queue
178 ····int·inWhichQueue;··//·L1:·1,·L2:·2,·L3:·3
179
180     float weight;
181     float curr_approximatedBurstTime;
182     float last_approximatedBurstTime;
183     float rem_approximatedBurstTime;
184     int timeStamp_startRunning;  // enter running state time
185     int totalRunningTime;
186     int timeStamp_startReady;  // enter ready state time
187     int totalReadyTime;
188 };
```

Figure 21: Thread Class - Priority and Queue Number

4. **初始化 `priority` 和 `inWhichQueue`**。請參考 Figure 22a、Figure 22b。

   最一開始的 process 是 main process，隨著 kernel 的建立而出現，它並沒有被設定 priority。我們擔心若沒有 default priority 可以確立所屬的 queue，在運行上會出現問題。因此統一在一個 process 被建立的時候就設定 default 的 priority 為最高。

   接著由於之後的 process 都是被 fork 出來的，並且在 command line 會輸入 process 的 priority，所以我們讓 process 在要被 fork 建立時就 assign 相對應被輸入的 priority 給它，作為初始化。

```
37  Thread::Thread(char *threadName, int threadID) {
38      ID = threadID;
39      name = threadName;
40  ····priority·=·149;
41  ····inWhichQueue·=·1;··//·Default·in·L1
42      weight = 0.5;
43      isExec = false;
44      stackTop = NULL;
45      stack = NULL;
46      status = JUST_CREATED;
47      for (int i = 0; i < MachineStateSize; i++) {
48          machineState[i] = NULL;  // not strictly necessary, since
49                                   // new thread ignores contents
50                                   // of machine registers
51      }
52      space = NULL;
53      TimeUpdate_NewToReady();
54  }
```

(a) Thread::Thread() - Default Priority and Queue Number Initialization

```
268 int Kernel::Exec(char *name) {
269     t[threadNum] = new Thread(name, threadNum);
270     t[threadNum]->setIsExec();
271     t[threadNum]->space = new AddrSpace();
272     t[threadNum]->priority = threadPriority[threadNum];
273     t[threadNum]->Fork((VoidFunctionPtr)&ForkExecute, (void *)t[threadNum]);
274     DEBUG(dbgThread, "new thread num: " << threadNum);
275     threadNum++;
276
277     return threadNum - 1;
```

(b) Kernel::Exec() - Priority Initialization

Figure 22: Processes' Priority and Queue Number Initialization

5. **將 process 放入適當的 queue 中**。請參考 Figure 23。

   最後，一個 process 被建立好後，需要依照 priority 被 scheduler 放入相對應的 ready queue，Scheduler::ReadyToRun() 就負責這樣的工作。因此我們修改 Scheduler::ReadyToRun()，並且使用 list.cc 提供的 Append() 和有排序功能的 Insert() 函式來實現。另外，這裡也必須更新 process 內部的 inWhichQueue 變數，紀錄現在被放進去哪個 ready queue。

```
64  void Scheduler::ReadyToRun(Thread *thread) {
65      ASSERT(kernel->interrupt->getLevel() == IntOff);
66      DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
67      // cout << "Putting thread on ready list: " << thread->getName() << endl;
68      thread->setStatus(READY);
69      thread->TimeUpdate_ToReady(kernel->stats->totalTicks);
70
71      // readyList->Append(thread);
72      DEBUG(dbgThread, ":D, " << thread->getID() << ": "
73                               << thread->rem_approximatedBurstTime);
74      // Multilevel feedback queue
75      if (thread->priority <= 49) {
76          L3->Append(thread);
77          DEBUG(dbgZ, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread ["
78                                   << thread->getID()
79                                   << "] is inserted into queue L[" << 3 << "]");
80          thread->inWhichQueue = 3;
81      } else if (thread->priority <= 99) {
82          L2->Insert(thread);
83          DEBUG(dbgZ, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread ["
84                                   << thread->getID()
85                                   << "] is inserted into queue L[" << 2 << "]");
86          thread->inWhichQueue = 2;
87      } else {
88          L1->Insert(thread);
89          DEBUG(dbgZ, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread ["
90                                   << thread->getID()
91                                   << "] is inserted into queue L[" << 1 << "]");
92          thread->inWhichQueue = 1;
93      }
94      // CheckPreempt_btMultipleReadyQueue();
95      // CheckPreempt_inSingleReadyQueue();
96      DEBUG(dbgThread, ":OO");
97  }
```

Figure 23: Scheduler::ReadyToRun()

2.5 實作：multilevel feedback queue － Non-Preemptive Scheduling

Non-Preemptive 的實作分為兩種，第一種為 process 工作結束，第二種為 process 進入 waiting state 等待資源。兩種皆只要 scheduler 單純從

最高 priority 的 queue 中，拿取最高 priority 的 process 繼續執行即可。而 Scheduler::FindNextToRun() 就是 scheduler 負責選擇下一個需要執行的 process 的地方。所以我們修改 Scheduler::FindNextToRun() 的內容，讓其從最高 priority 的 queue 開始挑選起。詳細如 Figure 24 所示。

```
107 Thread *Scheduler::FindNextToRun() {
108     ASSERT(kernel->interrupt->getLevel() == IntOff);
109 
110     // if (readyList->IsEmpty()) {
111     //     return NULL;
112     // } else {
113     //     return readyList->RemoveFront();
114     // }
115 
116     // Multilevel feedback queue
117     if (!L1->IsEmpty()) {
118         DEBUG(dbgZ, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread ["
119                                  << L1->Front()->getID()
120                                  << "] is removed from queue L[" << 1 << "]");
121         return L1->RemoveFront();
122     } else if (!L2->IsEmpty()) {
123         DEBUG(dbgZ, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread ["
124                                  << L2->Front()->getID()
125                                  << "] is removed from queue L[" << 2 << "]");
126         return L2->RemoveFront();
127     } else if (!L3->IsEmpty()) {
128         DEBUG(dbgZ, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread ["
129                                  << L3->Front()->getID()
130                                  << "] is removed from queue L[" << 3 << "]");
131         return L3->RemoveFront();
132     } else {
133         return NULL;
134     }
135 }
```

Figure 24: Scheduler::FindNextToRun()

這兩種 Non-Preemptive 的實作方式不同之處在於，若 process 工作結束，則之後會被刪除，不再被使用；但若 process 只是因為在排隊等待使用資源而被 context switch，則還需要重新進入 ready queue，因此需要維護 process 內紀錄時間的變數。所以，在 Thread::Sleep() 中，我們在一個 process 決定要被 context switch 進入 device queue 之前，先呼叫 TimeUpdate_RunningToWaiting() 來維護時間變數 (TimeUpdate_RunningToWaiting() 的詳細內容會在之後 2.8 節說明)，請參考 Figure 25。

2.6 實作：multilevel feedback queue － Preemptive Scheduling

Preemptive 的實作包含在 2.1 節提到的－新 priority 較高的 process 被加入 ready queue 中、round robin 時間到。

```
249 void Thread::Sleep(bool finishing) {  // finish v I/O etc.
250     Thread *nextThread;
251
252     ASSERT(this == kernel->currentThread);
253     ASSERT(kernel->interrupt->getLevel() == IntOff);
254
255     DEBUG(dbgThread,
256         "Sleeping thread: " << name << ", finishing? " << finishing);
257     DEBUG(dbgTraCode, "In Thread::Sleep, Sleeping thread: "
258                 << name << ", " << kernel->stats->totalTicks);
259
260     status = BLOCKED;
261     // cout << "debug Thread::Sleep " << name << "wait for Idle\n";
262     while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
263         kernel->interrupt->Idle();  // no one to run, wait for an interrupt
264     }
265     // returns when it's time for us to run
266     DEBUG(dbgZ, "[E] Tick ["
267                 << kernel->stats->totalTicks << "]: Thread ["
268                 << nextThread->getID()
269                 << "] is now selected for execution, thread ["
270                 << this->getID() << "] is replaced, and it has executed ["
271                 << kernel->stats->totalTicks - timeStamp_startRunning
272                 << "] ticks");
273     if (!finishing) {
274         TimeUpdate_RunningToWaiting(kernel->stats->totalTicks);
275         DEBUG(dbgZ, "[D] Tick ["
276                     << kernel->stats->totalTicks << "]: Thread ["
277                     << this->getID()
278                     << "] update approximate burst time, from: ["
279                     << last_approximatedBurstTime << "], add ["
280                     << kernel->stats->totalTicks - timeStamp_startRunning
281                     << "], to [" << curr_approximatedBurstTime << "]");
282     }
283     kernel->scheduler->Run(nextThread, finishing);
284 }
```

Figure 25: Thread::Sleep()

在有新 priority 較高的 process 被加入 ready queue 中的情況下，要先經過 queue 和 queue 之間的檢查後，再檢查單一個 queue 中的 priority。我們為 Scheduler class 實作了兩個 method，`Scheduler::CheckPreempt_btMultipleReadyQueue()` 和 `Scheduler::CheckPreempt_inSingleReadyQueue()`，來分別檢查這兩種情形。這兩個 method 都會回傳一個布林值來決定要不要 preempt。

```
333 bool Scheduler::CheckPreempt_btMultipleReadyQueue() {
334     bool contextSwitch = false;
335     if ((!L1->IsEmpty()) && (kernel->currentThread->inWhichQueue > 1)) {
336         contextSwitch = true;
337     }
338     if ((!L2->IsEmpty()) && (kernel->currentThread->inWhichQueue > 2)) {
339         contextSwitch = true;
340     }
341     return contextSwitch;
342 }
```

Figure 26: Scheduler::CheckPreempt_btMultipleReadyQueue()

`Scheduler::CheckPreempt_btMultipleReadyQueue()` 的實作如 Figure 26 所示。`Scheduler::CheckPreempt_btMultipleReadyQueue()` 會依序由最高 priority 的 queue 開始檢查起，若有 queue 中有東西，並且這個有東西的 queue priority 比現在執行的 process 所屬的 queue priority 還高，則代表需要 context switch，回傳 true，反之則回傳 false。

`Scheduler::CheckPreempt_inSingleReadyQueue()` 的實作如 Figure 27 所示。這個函式主要是針對 L1 queue 所設計，因為 L2 為內部不能 preemption 的 ready queue，L3 是採用 round robin，因此 L2 和 L3 都用不到。在 `Scheduler::CheckPreempt_inSingleReadyQueue()` 中，我們先檢查現在執行的 process 是否隸屬於 L1 且不為 L1 中最後一個 process，若是，則檢查在 L1 中 priority 最高的 process 的 remaining burst time 是否比現在執行的 process 還少；若比現在執行的 process 還少，則需要 context switch，回傳 true，反之則檢查 process ID；若比現在執行的 process ID 還小，則回傳 true，反之回傳 false。

另一種情形是針對 L3 round robin 的 preemption，我們設計了 `Scheduler::CheckYield()` 作為 scheduler 的另一個 method。它會檢查現在執行的 process 是否隸屬於 L3 且執行時間超過 100 ticks，若是則要被 context switch，回傳 true，反之回傳 false。詳細請見 figure 28。

以上這兩種需要檢查 preemption 的情況，皆需要安排在 `Alarm::CallBack()` 中。`Alarm::CallBack()` 是一個 callback function，會在 `Machine::Run()` 中的 `OneTick()` 中的 `CheckIfDue()` 被觸發。它是一個在原本的 nachOS 中，負責通知作業系統需不需要進行 preemptive context switch 的合法管道，能確保在進行 `Thread::Yield()` 之前，能夠完成現在作業系統所有應該完成的工作 (因為 nachOS 中原本唯一可以 preemption 的方式只有 round robin)，例如讓占用資源的 process 用 callback 還資源（`CheckIfDue()`）、前進系統

```
344 bool Scheduler::CheckPreempt_inSingleReadyQueue() {
345     bool contextSwitch = false;
346     DEBUG(dbgThread, "totalRunning: "
347                      << kernel->currentThread->totalRunningTime
348                      << ", rem_appr: "
349                      << kernel->currentThread->curr_approximatedBurstTime -
350                         (kernel->stats->totalTicks -
351                          kernel->currentThread->timeStamp_startRunning +
352                          kernel->currentThread->totalRunningTime));
353     if (kernel->currentThread->inWhichQueue == 1 && !L1->IsEmpty()) {
354         if (kernel->currentThread->curr_approximatedBurstTime -
355             (kernel->stats->totalTicks -
356              kernel->currentThread->timeStamp_startRunning +
357              kernel->currentThread->totalRunningTime) >
358             L1->Front()->rem_approximatedBurstTime) {
359             contextSwitch = true;
360         } else if (kernel->currentThread->curr_approximatedBurstTime -
361                    (kernel->stats->totalTicks -
362                     kernel->currentThread->timeStamp_startRunning +
363                     kernel->currentThread->totalRunningTime) ==
364                    L1->Front()->rem_approximatedBurstTime &&
365                    kernel->currentThread->getID() > L1->Front()->getID()) {
366             contextSwitch = true;
367         }
368     }
369     return contextSwitch;
370 }
```

Figure 27: Scheduler::CheckPreempt_inSingleReadyQueue()

```
317 int Scheduler::CheckYield(Thread *currThread) {
318     int doYield = 1;
319     if (currThread->inWhichQueue == 1) {
320         return !doYield;
321     } else if (currThread->inWhichQueue == 2) {
322         return !doYield;
323     } else {
324         if (kernel->stats->totalTicks - currThread->timeStamp_startRunning >=
325             100) {
326             return doYield;
327         } else {
328             return !doYield;
329         }
330     }
331 }
```

Figure 28: Scheduler::CheckYield()

時間 (totalTicks) 等等。若確定需要 context switch，Alarm::CallBack() 會呼叫 Interrupt::YieldOnReturn() 將 yieldOnReturn 設為 true，以觸發 OneTick() 呼叫 Yield()。為了不打亂原本 nachOS 的安排，並且 spec 也規定不可以更動 machine folder 中的內容，所以我們在 Alarm::CallBack() 中放入這兩種檢查 preemption 的三個函式：Scheduler::CheckPreempt_btMultipleReadyQueue()、Scheduler::CheckPreempt_inSingleReadyQueue()、Scheduler::CheckYield()，如 Figure 29 第 50～51 行所示。其中，我們將 Scheduler::CheckPreempt_btMultipleReadyQueue() 和 Scheduler::CheckPreempt_inSingleReadyQueue() 包含在 51 行的 Scheduler::Aging() 中，我們會在 2.7 節中說明為何這麼實作以及關於 aging 的詳細實作。

```
44  void Alarm::CallBack() {
45      Interrupt *interrupt = kernel->interrupt;
46      MachineStatus status = interrupt->getStatus();
47      /* Aging and check preemption */
48
49      if (status != IdleMode &&
50          (kernel->scheduler->CheckYield(kernel->currentThread) ||
51           kernel->scheduler->Aging())) {
52          interrupt->YieldOnReturn();
53      }
54  }
```

Figure 29: Alarm::CallBack()

若談到為何不是將 Scheduler::CheckPreempt_btMultipleReadyQueue()、Scheduler::CheckPreempt_inSingleReadyQueue() 放在 Scheduler::ReadyToRun() 裡，是因為我們發現在這次作業的第 4 筆測資中會有其中一個 process 永遠還不了資源、作業系統沒有正確執行 OneTick() 前進時間的情形 (例如在 Figure 30，Semaphore::V() 中會在第 112 行就 context switch 走，114 行的 value 沒有辦法加回去)。這個發現進而引發我們去更了解原本 nachOS 中的層次設計，而決定放在 Alarm::CallBack() 裡作為定時的例行性檢查，接著交由 Thread::Yield() 去切換 process。

2.7 實作：multilevel feedback queue － Aging

Aging 的機制我們依照 spec 規定實作，在 Alarm::CallBack() 被觸發時呼叫 Scheduler::Aging()。作為一個 scheduler 的 method，它的細節如 Figure 31 所示。這個 method 會依序 scan through 3 個 ready queue 中所有的 process，檢查是否有 process 已經在 ready queue 裡等待超過 1500 ticks。若一個 process 等待超過 1500 ticks，我們會幫它的 priority 增加 10 (上限為 149)，並且更新它在 ready queue 中開始等待的時間。最後 Scheduler::Aging() 會呼叫 Scheduler::ReArrange() 重新安排 3 個 queue 裡的 processes，將需要換 queue 的 process 移動後更新 inWhichQueue。Scheduler::ReArrange() 的實作請參考 Figure 32。

```
101  void Semaphore::V() {
102      DEBUG(dbgTraCode, "In Semaphore::V(), " << kernel->stats->totalTicks);
103      Interrupt *interrupt = kernel->interrupt;
104      DEBUG(dbgThread, "thread " << kernel->currentThread->getName()
105                               << " In Semaphore::V(), "
106                               << kernel->stats->totalTicks);
107
108      // disable interrupts
109      IntStatus oldLevel = interrupt->SetLevel(IntOff);
110
111      if (!queue->IsEmpty()) {  // make thread ready.
112          kernel->scheduler->ReadyToRun(queue->RemoveFront());
113      }
114      value++;
115
116      // re-enable interrupts
117      (void)interrupt->SetLevel(oldLevel);
118      // kernel->scheduler->CheckPreempt_btMultipleReadyQueue();
119      // kernel->scheduler->CheckPreempt_inSingleReadyQueue();
120  }
```

Figure 30: Semaphore::V()

```
235  int Scheduler::Aging() {
236      ListIterator<Thread *> *iterator;
237      Thread *thread;
238      for (int i = 1; i <= 3; i++) {
239          if (i == 1) {
240              iterator = new ListIterator<Thread *>(L1);
241          } else if (i == 2) {
242              iterator = new ListIterator<Thread *>(L2);
243          } else {
244              iterator = new ListIterator<Thread *>(L3);
245          }
246          for (; !iterator->IsDone(); iterator->Next()) {
247              thread = iterator->Item();
248              if (kernel->stats->totalTicks - thread->timeStamp_startReady >=
249                  1500) {
250                  int old_priority = thread->priority;
251                  thread->priority =
252                      (thread->priority + 10 > 149) ? 149 : thread->priority + 10;
253                  thread->timeStamp_startReady = kernel->stats->totalTicks;
254                  DEBUG(dbgZ, "[C] Tick [" << kernel->stats->totalTicks
255                                           << "]: Thread [" << thread->getID()
256                                           << "] changes its priority from ["
257                                           << old_priority << "] to ["
258                                           << thread->priority << "]");
259              }
260          }
261          delete iterator;
262      }
263      return ReArrange();
264      // DEBUG(dbgThread, "U_U " << kernel->stats->totalTicks);
265  }
```

Figure 31: Scheduler::Aging()

```
int Scheduler::ReArrange() {
    ListIterator<Thread *> *iterator;
    Thread *thread;
    SortedList<Thread *> *new_L1;
    SortedList<Thread *> *new_L2;
    List<Thread *> *new_L3;
    new_L1 = new SortedList<Thread *>(L1Cmp);
    new_L2 = new SortedList<Thread *>(L2Cmp);
    new_L3 = new List<Thread *>;

    for (int i = 1; i <= 3; i++) {
        if (i == 1) {
            iterator = new ListIterator<Thread *>(L1);
        } else if (i == 2) {
            iterator = new ListIterator<Thread *>(L2);
        } else {
            iterator = new ListIterator<Thread *>(L3);
        }
        for (; !iterator->IsDone(); iterator->Next()) {
            thread = iterator->Item();
            if (thread->priority > 99) {
                new_L1->Insert(thread);
                if (thread->inWhichQueue != 1) {
                    thread->inWhichQueue = 1;
                }
            } else if (thread->priority > 49) {
                new_L2->Insert(thread);
                if (thread->inWhichQueue != 2) {
                    thread->inWhichQueue = 2;
                }
            } else {
                new_L3->Append(thread);
                if (thread->inWhichQueue != 3) {
                    thread->inWhichQueue = 3;
                }
            }
        }
        delete iterator;
    }
    delete L1;
    delete L2;
    delete L3;
    L1 = new_L1;
    L2 = new_L2;
    L3 = new_L3;
    int doYeild1 = CheckPreempt_btMultipleReadyQueue();
    int doYeild2 = CheckPreempt_inSingleReadyQueue();
    return (doYeild1 || doYeild2);
}
```

Figure 32: Scheduler::ReArrange()

我們能在 Scheduler::ReArrange() 中看到，最後在完成所有 processes 的移動後，會呼叫 Scheduler::CheckPreempt_btMultipleReadyQueue()、Scheduler::CheckPreempt_inSingleReadyQueue()，因為若有 process 由於 aging 的緣故移出了原本的 queue 更換到了其它 queue，就等同於有新的 process 被加入了 queue 中，需要去檢查有沒有要被 preempt 的情形。

而又因為 Scheduler::Aging() 需要在 Alarm::CallBack() 中被呼叫，如 Figure 29 的 51 行，且當呼叫 Scheduler::Aging() 時，也需要呼叫到 Scheduler::CheckPreempt_btMultipleReadyQueue()、Scheduler::CheckPreempt_inSingle 因此，我們將這兩個函式包在 Scheduler::ReArrange() 裡，讓 Scheduler::Aging() 呼叫 Scheduler::ReArrange() 就能一併檢查是否 preempt。這也就是為何我們在 2.6 節中提到，檢查是否有新 priority 較高的 process 被加入 ready queue 中的 preemption，Scheduler::CheckPreempt_btMultipleReadyQueue()、Scheduler::CheckPreempt_inSingleReadyQueue()，被包含在了 Scheduler::Aging() 一起做的原因。

2.8 實作：multilevel feedback queue －時間資訊的更新

時間資訊更新方面，我們在 2.2 節中提到，當 process 切換 state 時，就必須要去維護時間相關的變數，所以我們在 Thread Class 中加上了如 Figure 33 所示的幾個 method，以下將一一說明。

1. New → Ready：在一個 process 剛被 fork 出來加進 ready queue 時，變數所需要的更動如下。
   (a) curr_approximatedBurstTime：依據 spec 指示初始化為 0。
   (b) last_approximatedBurstTime：不在意。
   (c) rem_approximatedBurstTime：利用 curr_approximatedBurstTime 和 totalRunningTime 計算。
   (d) timeStamp_startRunning：還沒有進去過 running state 因此不在意。
   (e) timeStamp_startReady：要更新為現在進入 ready queue 的時間。
   (f) totalRunningTime：還沒有進去過 running state 因此設為 0。
2. Running → Ready：在一個 process 從 running state 被 preempt 後進到 ready queue 時，變數所需要的更動如下。
   (a) curr_approximatedBurstTime：維持，因為只有進入 waiting state 時才能更新。
   (b) last_approximatedBurstTime：維持，因為只有進入 waiting state 時才能更新 curr_approximatedBurstTime。

```
79  class Thread {
115     void TimeUpdate_ToReady(int system_totalTicks) {
116         timeStamp_startReady = system_totalTicks;
117     }
118     void TimeUpdate_ToRun(int system_totalTicks) {
119         timeStamp_startRunning = system_totalTicks;
120     }
121     void TimeUpdate_NewToReady() {
122         curr_approximatedBurstTime = 0;
123         totalRunningTime = 0;
124         rem_approximatedBurstTime =
125             curr_approximatedBurstTime - totalRunningTime;
126     }
127     void TimeUpdate_RunningToReady(int system_totalTicks) {
128         totalRunningTime += system_totalTicks - timeStamp_startRunning;
129         rem_approximatedBurstTime =
130             curr_approximatedBurstTime - totalRunningTime;
131     }
132     void TimeUpdate_RunningToWaiting(int system_totalTicks) {
133         totalRunningTime += system_totalTicks - timeStamp_startRunning;
134         last_approximatedBurstTime =
135             curr_approximatedBurstTime;
136         curr_approximatedBurstTime = weight * totalRunningTime +
137                                      (1 - weight) * curr_approximatedBurstTime;
138         rem_approximatedBurstTime = curr_approximatedBurstTime;
139         totalRunningTime = 0;
```

Figure 33: Thread Class - Time Update

(c) rem_approximatedBurstTime：因為 totalRunningTime 有更新，因此需要利用 curr_approximatedBurstTime 和 totalRunningTime 重新計算。

(d) timeStamp_startRunning：不在意。

(e) timeStamp_startReady：要更新為現在進入 ready queue 的時間。

(f) totalRunningTime：需要累加被 preempt 前在 running state 的時間 (+= totalTicks - timeStamp_startRunning)。

3. Running → Waiting：在一個 process 從 running state 進到 device queue 時，變數所需要的更動如下。

(a) curr_approximatedBurstTime：利用 weight(在這次作業中設為 0.5) 和上一次的 curr_approximatedBurstTime 更新。

(b) last_approximatedBurstTime：更新為上一次的 curr_approximatedBurstTime。

(c) rem_approximatedBurstTime：更新為這次新算好的 curr_approximatedBurstTime。

(d) timeStamp_startRunning：不在意。

(e) timeStamp_startReady：不在意。

(f) totalRunningTime：依據 spec 指示更新為 0。

4. Waiting → Ready：在一個 process 從 device queue 進到 ready queue 時，變數所需要的更動如下。

(a) curr_approximatedBurstTime：維持。

(b) last_approximatedBurstTime：維持。

(c) rem_approximatedBurstTime：維持。

(d) timeStamp_startRunning：不在意。

(e) timeStamp_startReady：更新為現在時間。

(f) totalRunningTime：不在意。

5. Running → Terminated：在一個 process 要 terminate 時，所有的值都不在意，因為不會再使用到這支 process。

(a) curr_approximatedBurstTime：不在意。

(b) last_approximatedBurstTime：不在意。

(c) rem_approximatedBurstTime：不在意。

(d) timeStamp_startRunning：不在意。

(e) timeStamp_startReady：不在意。

(f) totalRunningTime：不在意。

6. Ready → Running：最後 process 要進到 running state 時，只需要更新進到 running state 的時間就好。

   (a) curr_approximatedBurstTime：維持。

   (b) last_approximatedBurstTime：維持。

   (c) rem_approximatedBurstTime：維持。

   (d) timeStamp_startRunning：更新為現在時間。

   (e) timeStamp_startReady：維持。

   (f) totalRunningTime：不在意。

綜合以上我們發現，只要進到 ready queue 就要更新 timeStamp_startReady (開始等待被執行的時間)、只要進到 running state 就要更新 timeStamp_startRunning (開始執行使用 CPU 的時間)。綜合起來我們將這些變化整理出了 Figure 33 中的 5 個 method。

接著是更新時間的 method 要被呼叫的時機。`TimeUpdate_ToReady()` 和 `TimeUpdate_ToRun()` 比較直觀，分別在，進入 ready queue，`Scheduler::ReadyToRun()` 中呼叫，以及進入 running state，`Scheduler::Run()` 中呼叫。如 Figure 23 第 69 行和 Figure 34 第 174 行所示。而 `TimeUpdate_NewToReady()` 由於是管理 process 在一開始被建立的時候的初始值，因此我們選擇在 `Thread::Thread()` 中呼叫，因為只要一被 `Kernel::Exec()` 建立好，就會在 `Thread::Fork()` 函式中呼叫 `Thread::ReadyToRun()` 將其放進 ready queue 中，請參考 Figure 22a 第 53 行。`TimeUpdate_RunningToReady()` 的部分則是在 `Thread::Yield()` 中呼叫，請參考 Figure 35。因為 `Thread::Yield()` 是處理所有 preemption 切換的函式，而就是因為被 preempt 才會從 running state 轉換到 ready state。最後是 `TimeUpdate_RunningToWaiting()`，這個更新我們選擇在 `Thread::Sleep` 處理，讓確定只是需要等待資源而不是要被 terminate 的 process 維護這些時間變數，請參考 Figure 25 第 274 行。

```
154  void Scheduler::Run(Thread *nextThread, bool finishing) {
173      nextThread->setStatus(RUNNING);      // nextThread is now running
174      nextThread->TimeUpdate_ToRun(kernel->stats->totalTicks);
175      DEBUG(dbgThread, "Updated startRunning time: "
176                       << kernel->currentThread->timeStamp_startRunning);
177
178      DEBUG(dbgThread, "Switching from: " << oldThread->getName()
179                       << " to: " << nextThread->getName());
180      // This is a machine-dependent assembly language routine defined
181      // in switch.s.  You may have to think
182      // a bit to figure out what happens after this, both from the point
183      // of view of the thread and from the perspective of the "outside world".
184
185      DEBUG(dbgThread, "Before SWITCH(), oldThread "
186                       << oldThread->getName()
187                       << " status: " << oldThread->getStatus());
188      SWITCH(oldThread, nextThread);
189      DEBUG(dbgThread, "After SWITCH(), oldThread "
190                       << oldThread->getName()
191                       << " status: " << oldThread->getStatus());
192
193      // we're back, running oldThread
194
195      // interrupts are off when we return from switch!
196      ASSERT(kernel->interrupt->getLevel() == IntOff);
197
198      DEBUG(dbgThread, "Now in thread: " << oldThread->getName());
199
200      CheckToBeDestroyed();  // check if thread we were running
201                             // before this one has finished
202                             // and needs to be cleaned up
203
204      if (oldThread->space != NULL) {     // if there is an address space
205          oldThread->RestoreUserState();  // to restore, do it.
206          oldThread->space->RestoreState();
207      }
208  }
```

Figure 34: Scheduler::Run()

```
206  void Thread::Yield() {  // time sharing
207      Thread *nextThread;
208      IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);
209
210      ASSERT(this == kernel->currentThread);
211
212      DEBUG(dbgThread, "Yielding thread: " << name);
213      TimeUpdate_RunningToReady(kernel->stats->totalTicks);
214      nextThread = kernel->scheduler->FindNextToRun();
215      if (nextThread != NULL) {
216          DEBUG(dbgZ, "[E] Tick [" << kernel->stats->totalTicks << "]: Thread ["
217                                   << nextThread->getID()
218                                   << "] is now selected for execution, thread ["
219                                   << this->getID()
220                                   << "] is replaced, and it has executed ["
221                                   << this->totalRunningTime << "] ticks");
222          kernel->scheduler->ReadyToRun(this);
223          kernel->scheduler->Run(nextThread, FALSE);
224      }
225
226      (void)kernel->interrupt->SetLevel(oldLevel);
227  }
```

Figure 35: Thread::Yield()