CS342301: Operating System
MP3: CPU scheduling

Team member & contribution:

I. **江佩霖 111062118**
   i. Trace code
   ii. Implement function
   iii. 測試, Debug

II. **陳庭竣 111020025**
   i. Trace code
   ii. Implement function
   iii. 測試, Debug

I. Trace Code

由於很多 trace code 的 function 在 MP1、2 均有出現過，會簡要帶過(省略詳細介紹)。

1. New ~ Ready

i. Kernel::ExecAll() MP2

連續呼叫 Exec()，所有 user program 都執行後呼叫 Finish()結束當前執行緒。

ii. Kernel::Exec(char*) MP2

建立新的 thread，設定基本資料(setIsExec()、AddrSpace()…)後呼叫 Fork()。。

iii. Thread::Fork(VoidFunctionPtr , void*) MP2

呼叫 StackAllocate()分配 thread 的 stack 空間，呼叫 ReadyToRun()將 thread 加入排程。

iv. Thread::StackAllocate(VoidFunctionPtr , void*) MP2

準備固定大小的 stack memory，將 Routine 存入 Kernel Register 中。

v. Scheduler::ReadyToRun(Thread*) MP2

將 thread 的狀態設為 READY，並放入 ready queue 中等待 scheduling。

流程

- 傳入 user program 後，透過 **Kernel::ExecAll()**、**Kernel::Exec(char*)**，生成新的 thread
- thread state 從 JUST_CREATE → READY(**ReadyToRun()**修改)

2. Running ~ Ready

i. **Machine::Run()** MP1

模擬系統運作。呼叫 OneInstruction()處理 user program instruction(讀取指令、decode、執行)。呼叫 OneTick()。

ii. **Interrupt::OneTick()** MP1

模擬時間演進。

更新系統時間，檢查是否有 interrupt 需要處理。

Time device handler 若要求 context switch (yieldOnReturn == True)，則執行 Yield()。

補充: yieldOnReturn 參數設定

```
231  void Interrupt::YieldOnReturn()
232  {
233      // check if there is in a pending
234      ASSERT(inHandler == TRUE);
235      yieldOnReturn = TRUE;
236  }
```

補充: 呼叫 YieldOnReturn()函數 : Alarm::Callback()

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

Kernel::initialize()就先創建一個 new Alarm()，Alarm()函數會創建一個 Timer()，並呼叫 SetInterrupt()

補充:SetInterrupt()

```
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  }
```

Timer 被創建後就呼叫，在 100ticks 後 schedule 一個 interrupt，interrupt 所呼叫的 Timer::Callback()會再次呼叫 SetInterrupt()重新計數 100ticks，實現系統固定時間中斷一次的功能。另外，Timer::Callback()中會呼叫 Alarm::Callback()，設定 yieldOnReturn 參數使系統做 context switch。

iii. **Thread::Yield()** MP1

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

呼叫 FindNextToRun()找到下一個執行緒。有，則透過 ReadyToRun()將原本的 thread 放回 ready queue 中(state: RUNNING → READY)，呼叫 Run()讓 cpu 執行新的 thread。

iv. **Scheduler::FindNextToRun()** MP2
確認 ready queue 中是否有待執行的 thread，有則返回。

v. **Scheduler::ReadyToRun(Thread*)** MP2 (同 trace code part 1)

vi. **Scheduler::Run(Thread* , bool)** MP2
傳入的 nextThread 狀態設置 READY → RUNNING(從 ready queue 中調出的 thread 狀態均為 READY)，調用 SWITCH 完成硬體 context switch (switch 到 new thread)。

流程

- Machine::Run() 會不斷執行 instruction 並呼叫 OneTick()更新時間。
  若需要 context switch，呼叫 Scheduler::ReadyToRun(Thread*) 將當前執行 thread 狀態 RUNNING → READY；呼叫 Yield()，找到下一個要執行的 thread 傳入 Scheduler::Run()中，將其狀態 READY → RUNNING 並執行。

3. Running ~ Waiting

i. SynchConsoleOutput::PutChar(char) MP1

Lock → Acquire() / Release() 用來實現同一時間僅有一個 writer，拿到 lock 的 thread 才能執行(定義在 SynchConsoleOutput()中)。

呼叫 **ConsoleOutput::PutChar()**，用 waitFor → P()使還未進到 PutChar()的字元先等待。

補充: Lock → Acquire() / Release() 兩者會呼叫 semaphore 函數，實現資源管理。

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

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

ii. Semaphore::P()

```
74  void Semaphore::P() {
75      DEBUG(dbgTraCode, "In Semaphore::P(), " << kernel->stats->totalTicks);
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 available
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  }
```

和 Semaphore::V() 一起實現多個執行緒的同步控制。判斷 value (信號量) 是否可用。

- 若 value 為 0，代表有其他 thread 正在使用資源，則用 Append()將當前 thread 放到 waiting queue 中，呼叫 Sleep() (傳入參數為 False 因為還未執行，不須刪除 thread) 使其進入休眠狀態，直到其他執行緒透過 V() 釋放資源。
- 若 value 大於 0，代表資源可用，當前執行緒成功取得資源並將 value -1。

iii. List<T>::Append(T)

```
69  template <class T>
70  void List<T>::Append(T item) {
71      ListElement<T> *element = new ListElement<T>(item);
72
73      ASSERT(!IsInList(item));
74      if (IsEmpty()) {  // list is empty
75          first = element;
76          last = element;
77      } else {  // else put it after last
78          last->next = element;
79          last = element;
80      }
81      numInList++;
82      ASSERT(IsInList(item));
83  }
```

List<T>是 NachOS 定義的資料結構，實作一個 waiting queue。

Semaphore::P()透過 Append()將 thread 放到 queue 的尾端。

iv. **Thread::Sleep(bool)** MP2

將當前 thread 狀態 RUNNING → BLOCK，呼叫 FindNextToRun()從 ready queue 中找到新的 thread (若沒有則將 CPU mode 設置為 idle()，處理尚未完成的 interrupt 或直接關機)，呼叫 Scheduler::Run()執行。

v. **Scheduler::FindNextToRun()** MP2 (同 trace code part 2)

vi. **Scheduler::Run(Thread* , bool)** MP2 (同 trace code part 2)

流程

- 主要實現系統中的資源管理：透過 Lock → Acquire() / Release()機制實現在輸出資源的限制(只有一個 thread 可以拿到 console 硬體的資源)、Semaphore::V() / P()負責驗證、管理資源的取用、釋放。
- 若 thread 取不到資源，則進入 waiting queue 中等待資源(state: BLOCK)；若有 thread 釋放資源(呼叫 Semaphore::V())，則將 waiting queue 中的第一個 thread 取出放到 ready queue 中(state: READY)

4. Waiting ~ Ready

i. Semaphore::V()

```
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  }
```

通常在 Callback function 呼叫。

從 waiting queue 中喚醒一個 thread (放到 waiting queue 的 thread 都在等待資源，state: BLOCKING)，透過 ReadyToRun()將其 state: BLOCKING → READY，並放入 ready queue 中等待執行，將可用資源+1。

ii. Scheduler::ReadyToRun(Thread*) MP2 (同 trace code part 1)

流程

- 歸還 lock 或是 callback function 會呼叫 Semaphore::V()，代表資源釋放。
- **呼叫 Semaphore::P() :** lock->require()、SynchDisk::ReadSector()、SynchDisk::WriteSector()
- **呼叫 Semaphore::V() :** lock->release()、SynchDisk::CallBack()

5. Running ~ Terminated

i. ExceptionHandler(ExceptionType) case SC_Exit MP1

```
51  void ExceptionHandler(ExceptionType which) {
62      switch (which) {
64              switch (type) {
275                 // exit
276                 case SC_Exit:
277                     DEBUG(dbgAddr, "Program exit\n");
278                     // read the addr in reg
279                     val = kernel->machine->ReadRegister(4);
280                     cout << "return value:" << val << endl;
281                     // stop the process
282                     kernel->currentThread->Finish();
283                     break;
```

流程同 MP1(參數 which 得知哪種 exception，再透過 Register 2 讀取 exception 的 type)，這邊對應到 SC_Exit。

讀取 Register4 中的資料後呼叫 Thread::Finish()。

ii. Thread::Finish() MP2

首先確保僅 kernel→currentThread 可呼叫，確認所有工作均執行完畢後呼叫 Thread::Sleep()完成執行緒的中止(傳入 True 表示執行緒以結束，可刪除)。

iii. Thread::Sleep(bool) MP2 (同 trace code part 2)

基本流程同 trace code part 2，僅呼叫 Scheduler::Run()時傳入 bool 為 True。

iv. Scheduler::FindNextToRun() MP2 (同 trace code part 2)

v. Scheduler::Run(Thread* , bool) MP2 (同 trace code part 2)

由於 Sleep()呼叫時傳入 bool 為 True，context switch 前會將 oldthread 標記為應刪除(toBeDestroyed = oldthread)。Context switch 後會透過呼叫 CheckToBeDestroyed()將 thread 刪除。

補充: CheckToBeDestroyed()

```
152 void Scheduler::CheckToBeDestroyed() {
153     if (toBeDestroyed != NULL) {
154         delete toBeDestroyed;
155         toBeDestroyed = NULL;
156     }
157 }
```

流程

- 當 Machine::Run()讀取到 user program 的最後時，會讀取到 Exit 的 syscall。透過 ExceptionHandler 呼叫 Finish() → Sleep(**True**) → Run(nextThread , **True**)。
- 在 Scheduler::Run()中透過 toBeDestroyed 參數刪除 thread (state: RUNNING →刪除)

6. Ready ~ Running

i. **Scheduler::FindNextToRun()** MP2 (同 trace code part 2)

ii. **Scheduler::Run(Thread* , bool)** MP2 (同 trace code part 3)

流程同 trace code part 3，詳細說明 Context Switch back 的部分。

若切換到的 thread 之前曾經 switch 過，則往下執行(繼續之前執行到的部分)，若是全新的 thread，會從 threadRoot 開始執行。

iii. **SWITCH(Thread* , Thread*)**

```
295  #ifdef x86
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)
307  **       edx     contains inital argument to thread function
308  **       esi     points to thread function
309  **       edi     point to Thread::Finish()
310  */
```

```
311  _ThreadRoot:
312  ThreadRoot:
313          pushl   %ebp
314          movl    %esp,%ebp
315          pushl   InitialArg
316          call    *StartupPC
317          call    *InitialPC
318          call    *WhenDonePC
319
320          # NOT REACHED
321          movl    %ebp,%esp
322          popl    %ebp
323          ret
```

```
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
333  **
334  ** we push the current eax on the stack so that we can use it as
335  ** a pointer to t1, this decrements esp by 4, so when we use it
336  ** to reference stuff on the stack, we add 4 to the offset.
337  */
338          .comm   _eax_save,4
339
340          .globl  SWITCH
341   .globl  _SWITCH
```

```
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          movl    8(%esp),%eax            # move pointer to t2 into eax
358          movl    _EAX(%eax),%ebx         # get new value for eax into ebx
359          movl    %ebx,_eax_save          # save it
360          movl    _EBX(%eax),%ebx         # retore old registers
361          movl    _ECX(%eax),%ecx
362          movl    _EDX(%eax),%edx
363          movl    _ESI(%eax),%esi
364          movl    _EDI(%eax),%edi
365          movl    _EBP(%eax),%ebp
366          movl    _ESP(%eax),%esp         # restore stack pointer
367          movl    _PC(%eax),%eax          # restore return address into eax
368          movl    %eax,4(%esp)            # copy over the ret address on the stack
369          movl    _eax_save,%eax
370          ret
371  #endif // x86
```

當創建新的 thread 時，會呼叫 StackAllocate()，allocate memory 給 thread 並將基本資料存在 machineState[]這個結構中。

```
302  void Thread::StackAllocate(VoidFunctionPtr func, void *arg) {
343  #ifdef PARISC
349  #else
350      machineState[PCState] = (void *)ThreadRoot;
351      machineState[StartupPCState] = (void *)ThreadBegin;
352      machineState[InitialPCState] = (void *)func;
353      machineState[InitialArgState] = (void *)arg;
354      machineState[WhenDonePCState] = (void *)ThreadFinish;
355  #endif
356  }
```

根據 StackAllocate()，可以知道: (PCState, StartupPCState, InitialPCState, InitialArgState, WhenDonePCState)分別是(7, 2, 5, 3, 6)，且儲存的資料如上圖。

在呼叫 SWITCH(oldthread, nextThread)時，會將 register 資料存到 oldthread (對應 code 行數 344 - 356)，並取出 nextThread 中的資料(對應 code 行數 357 - 369)。

執行新的 thread 時，若曾經 SWITCH 過，會因為 SWITCH code 中

movl %esp, _ESP(%eax) // 352 行, 將 stack pointer 的值覆寫在 stackTop

而使 stackTop 被覆寫為 ret，跳轉到前次儲存的返回地址繼續還未執行完的部分。若為全新的 thread，會執行 stackTop 中的 ThreadRoot (執行: ThreadBegin(), (void*)func, ThreadFinish())，實現一個 thread 從創建到執行結束的流程。

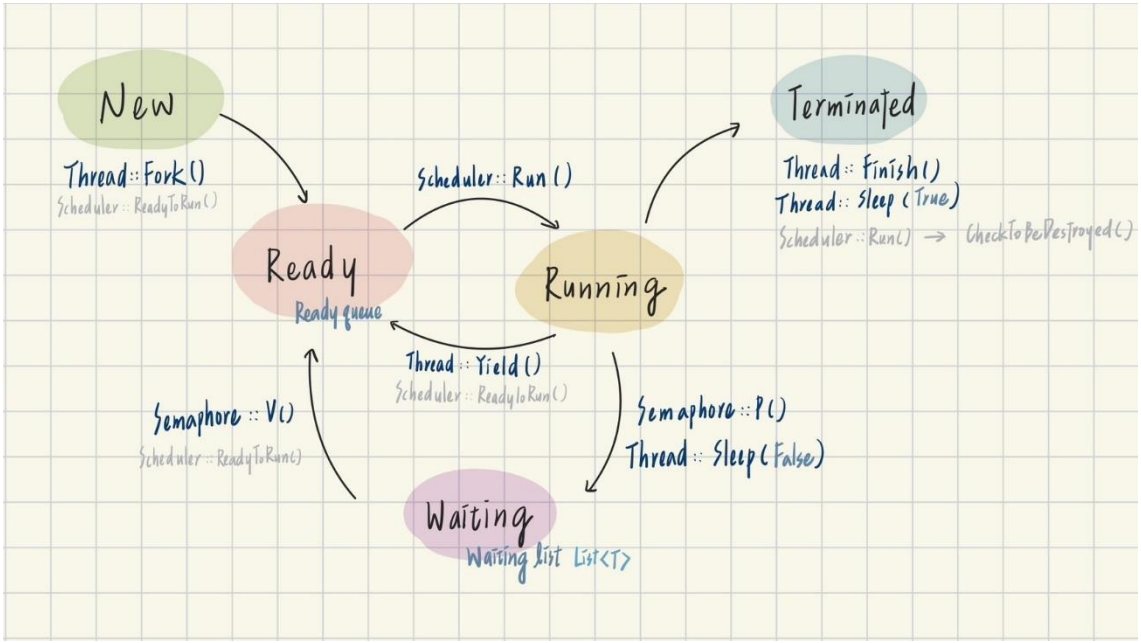
iv. Machine::Run() MP2 (同 trace code part 2)

在 loop 中執行 next thread

流程

- 透過 FindNextToRun()從 ready queue 中找到等待執行的 thread，進入 Scheduler::Run()。
- 在 Scheduler::Run()中將待執行的 thread 狀態 READY → RUNNING，接著進行 SWITCH()。
- 依是否有被 SWITCH()，分成兩種狀況：
  - 有：繼續執行上次 SWITCH()後的程式碼
  - 無：為全新的 thread，呼叫 ThreadRoot，依序呼叫 ThreadBegin()、(void*)func、ThreadFinish()
    - **ThreadBegin()**: 刪除待刪除的 thread (呼叫 CheckToBeDestoryed())，enable interrupts
    - **(void*)func(ForkExecute(Thread *t))**: 尋找檔案資料(AddrSpace::Load())，初始化(AddrSpace::Execute())並進入 Machine::Run()的 loop 中執行
    - **ThreadFinish()**: 當執行完(void*)func，代表此 thread 完成了，呼叫 ThreadFinish() → Finish()，結束並刪除 thread。

詳細的 thread status 對應切換函數的圖表如下



II. Implement the I/O system calls in NachOS

更動的檔案:

Scheduler.h / .cc、 stats.h 、thread.h / .cc、alarm.cc 、debug.h 、kernel.h / .cc

實作大綱:

- **Scheduler**: 創建排序的 queue list 資料結構並維護 Update Priority 的函數；
- **stats.h**: 調整 spec 上說明的 ConsoleTime；
- **debug.h**: 新增 dbgScheduler 來維護新的 debug 資訊；
- **alarm.cc**: 修改 Callback 函數，呼叫 Update Priority 並根據新的 Priority 分配 queue list
- **kernel**: 增加 Exec()參數以對應-ep 新增的傳入參數，增加 priorityFlag 和陣列記錄執行 user program 時的 priority。
- **thread**: 增加 spec 要求須記錄的參數(waiting time, use time, burst time, …)，並實作取值函數。維護並實作 queue list 的比較函數。

1. multilevel feedback queue

➢ **Scheduler.h** Scheduler class

Public:

```
void UpdatePriority();
bool L1_Empty() {
      return readyList_L1->IsEmpty();
}
bool L2_Empty() {
      return readyList_L2->IsEmpty();
}
bool L3_Empty() {
      return readyList_L3->IsEmpty();
}
int L1_Front_Remain() {
      if (readyList_L1->IsEmpty() == 0)
         return readyList_L1->Front()->get_Burst_Time()-readyList_L1->Front()-
>get_Use_Time();
      else
         return -1;
}
```

Private:

```
    SortedList<Thread *> *readyList_L1, *readyList_L2;
    List<Thread *> *readyList_L3;
```

- **Scheduler..cc** Scheduler::Scheduler()

  建立三種 level 各自的 queue 為 L1、L2、L3。

```
Scheduler::Scheduler()
{
    readyList_L1 = new SortedList<Thread *>(cmp_BurstTime);
    readyList_L2 = new SortedList<Thread *>(cmp_Priority);
    readyList_L3 = new List<Thread *>;

    toBeDestroyed = NULL;
}
```

- **Scheduler..cc** Scheduler::ReadyToRun(Thread *thread)

  依照 Thread 的 priority 來分配該 thread 要進入哪一個 ready queue。若 priority 介於 0-49，則進入 L3；若 priority 介於 50-99，則進入 L2；若 priority 介於 100-149，則進入 L1。設定在 ready queue 的起始等待時間。

```
void Scheduler::ReadyToRun (Thread *thread) {

    ASSERT(kernel->interrupt->getLevel() == IntOff);
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
    //cout << "Putting thread on ready list: " << thread->getName() << endl ;
    thread->setStatus(READY);

    thread->wait_Start_Time = kernel->stats->totalTicks;

    Thread *old_Thread = kernel->currentThread;

    if(thread->get_Priority() >= 100) {
        DEBUG(dbgScheduler, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" << thread->getID() << "] is inserted into queue L[1]");
        readyList_L1->Insert(thread);
    }
    else if(thread->get_Priority() >= 50) {
        DEBUG(dbgScheduler, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" << thread->getID() << "] is inserted into queue L[2]");
        readyList_L2->Insert(thread);
    }
    else {
        DEBUG(dbgScheduler, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" << thread->getID() << "] is inserted into queue L[3]");
        readyList_L3->Append(thread);
    }

}
```

- **Scheduler..cc** cmp_BurstTime(Thread *a, Thread *b)

  實作 L1 queue 的比較函數，L1 queue 的排序方法是按照預估的剩餘執行時間，剩餘越少時間，則順位越高。

```
int cmp_BurstTime(Thread *a, Thread *b){
    double at = a->get_Burst_Time() - a->get_Use_Time();
    double bt = b->get_Burst_Time() - b->get_Use_Time();

    if(at < 0) at = 0;
    if(bt < 0) bt = 0;

    if(at > bt) return 1;
    else if(at < bt) return -1;
    else if(a->getID() > b->getID()) return 1;
    else return -1;
}
```

- **Scheduler..cc** cmp_Priority(Thread *a, Thread *b)

  實作 L2 的比較函數，L2 queue 的排序方法是依照 priority 的值，priority 值越大，則順位越高。

```
int cmp_Priority(Thread *a, Thread *b){
    int ap = a->get_Priority();
    int bp = b->get_Priority();

    if(ap > bp) return -1;
    else if(ap < bp) return 1;
    else if(a->getID() > b->getID()) return 1;
    else return -1;
}
```

- **Scheduler..cc** Scheduler::FindNextToRun()

  尋找下一個進入 CPU 執行的 thread，從 L1 開始找，若 L1 為空，則找 L2；若 L2 為空，則找 L3。

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

    if (readyList_L1->IsEmpty() == 0) {
        Thread *thread = readyList_L1->Front();
        DEBUG(dbgScheduler, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" << thread->getID() << "] is removed from queue L[1]");
        return readyList_L1->RemoveFront();   // ?
    }
    else if (readyList_L2->IsEmpty() == 0) {
        Thread *thread = readyList_L2->Front();
        DEBUG(dbgScheduler, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" << thread->getID() << "] is removed from queue L[2]");
        return readyList_L2->RemoveFront();
    }
    else if (readyList_L3->IsEmpty() == 0) {
        Thread *thread = readyList_L3->Front();
        DEBUG(dbgScheduler, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" << thread->getID() << "] is removed from queue L[3]");
        return readyList_L3->RemoveFront();
    }
    else {
        return NULL;
    }
}
```

- **Scheduler..cc** Scheduler::Run (Thread *nextThread, bool finishing)

  儲存並更新舊的 thread 資訊(累積執行時間)，為新的 thread 更新好狀態(開始執行時間)。

```
    if (oldThread->space != NULL) { // if this thread is a user program,
        oldThread->SaveUserState();     // save the user's CPU registers
    oldThread->space->SaveState();
    }

    oldThread->CheckOverflow();         // check if the old thread
                                        // had an undetected stack overflow

    int old_ID = oldThread->getID();
    int new_ID = nextThread->getID();

    double old_Use_Time = oldThread->get_Use_Time();

    DEBUG(dbgThread, "Switching from: " << oldThread->getName() << " to: " << nextThread->getName());

    DEBUG(dbgScheduler, "[E] Tick [" << kernel->stats->totalTicks << "]: Thread [" << new_ID
                                     << "] is now selected for execution, thread [" << old_ID
                                     << "] is replaced, and it has executed [" << old_Use_Time << "] ticks");

    kernel->currentThread = nextThread;  // switch to the next thread
    nextThread->setStatus(RUNNING);      // nextThread is now running

    nextThread->burst_Start_Time = kernel->stats->totalTicks;
    nextThread->set_Wait_Time(0);
```

- **Scheduler..cc** Scheduler::UpdatePriority()

  呼叫 aging 更新 thread 的 priority，在 L2 或 L3 有 thread 升級時，將 thread 移出原先的層級，呼叫 ReadyToRun 放入新對應的層級重新排序。

```
void
Scheduler::UpdatePriority()
{
    if(readyList_L1->IsEmpty() == 0){
        ListIterator<Thread *> *iter1 = new ListIterator<Thread *>(readyList_L1);
        for (; !iter1->IsDone(); iter1->Next()) {
            iter1->Item()->aging(kernel->stats->totalTicks - iter1->Item()->wait_Start_Time);
        }
        delete iter1;
    }

    if(readyList_L2->IsEmpty() == 0){
        ListIterator<Thread *> *iter2 = new ListIterator<Thread *>(readyList_L2);
        for (; !iter2->IsDone(); iter2->Next()) {
            bool upgrade = iter2->Item()->aging(kernel->stats->totalTicks - iter2->Item()->wait_Start_Time);
            if(upgrade) {
                if(iter2->Item()->getID() > 0) {
                    DEBUG(dbgScheduler, "[C] Tick [" << kernel->stats->totalTicks << "]: Thread [" << iter2->Item()->getID() << "] is removed from queue L[1]");
                    readyList_L2->Remove(iter2->Item());
                    kernel->scheduler->ReadyToRun(iter2->Item());
                }
            }
            else {
                readyList_L2->Remove(iter2->Item());
                kernel->scheduler->ReadyToRun(iter2->Item());
            }
        }
        delete iter2;
    }
```

```
    if(readyList_L3->IsEmpty() == 0){
        ListIterator<Thread *> *iter3 = new ListIterator<Thread *>(readyList_L3);
        for (; !iter3->IsDone(); iter3->Next()) {
            bool upgrade = iter3->Item()->aging(kernel->stats->totalTicks - iter3->Item()->wait_Start_Time);
            if(upgrade) {
                if(iter3->Item()->getID() > 0) {
                    DEBUG(dbgScheduler, "[C] Tick [" << kernel->stats->totalTicks << "]: Thread [" << iter3->Item()->getID() << "] is removed from queue L[2]");
                    readyList_L3->Remove(iter3->Item());
                    kernel->scheduler->ReadyToRun(iter3->Item());
                }
            }
        }
        delete iter3;
    }
}
```

- **Alarm.cc** Alarm::CallBack()

  呼叫 UpdatePriority 更新每個 thread 的 priority，並檢查是否有 preempt 的狀況產生，若有，則呼叫 YieldOnReturn。

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

    // update priority
    scheduler->UpdatePriority();

    int label = 0;

    if(kernel->currentThread->get_Priority() < 50){
        if(scheduler->L1_Empty() == 0) label = 1;
        if(scheduler->L2_Empty() == 0) label = 1;
        if(kernel->stats->totalTicks - kernel->currentThread->burst_Start_Time >= 100) label = 1;
    }

    else if(kernel->currentThread->get_Priority() >= 50 && kernel->currentThread->get_Priority() < 100){
        if(scheduler->L1_Empty() == 0) label = 1;
    }

    else {
        int current_remain = kernel->currentThread->get_Burst_Time() - kernel->currentThread->get_Use_Time() - (kernel->stats->totalTicks - kernel->currentThread->burst_Start_Time);
        if(scheduler->L1_Empty() == 0 && current_remain > scheduler->L1_Front_Remain()) label = 1;
    }

    if(label == 1) {
        if (status != IdleMode) {
            interrupt->YieldOnReturn();
        }
    }
}
```

- **Kernel.cc** Kernel::Exec(int idx, char *name)
  新增 idx 參數傳入，將讀進來的 thread 按照 idx 做區分，到 Thread constructor 初始化 name、id、priorty。

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

int Kernel::Exec(int idx, char *name) {
    if(exec_Priority[idx] > 0) {
        t[threadNum] = new Thread(name, threadNum, exec_Priority[idx]);
    } else {
        t[threadNum] = new Thread(name, threadNum);
    }

    t[threadNum]->space = new AddrSpace();
    t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);
    threadNum++;

    return threadNum - 1;
```

- **Thread.cc** Thread::Thread(char* threadName, int threadID, int P)
  初始化 Thread 資訊(新增初始化在 thread.h 中定義的實作參數)

```
Thread::Thread(char* threadName, int threadID, int P)
{
    ID = threadID;
    name = threadName;

    priority = P;
    burst_Time = 0;
    burst_Start_Time = 0;
    use_Time = 0;
    accumulate_Use_Time = 0;
    wait_Time = 0;
    wait_Start_Time = 0;

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

- **Thread.cc** Thread::Yield ()

  更新 thread 實際執行的時間，呼叫 FindNextToRun()，找到下個 running 的 thread，呼叫 ReadyToRun()讓現在的 thread 進入 ready state，呼叫 Run 實際執行。

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

    ASSERT(this == kernel->currentThread);

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

    // running -> ready, update used use time

    double total_Use_Time = this->get_Use_Time() + (kernel->stats->totalTicks - this->burst_Start_Time);
    this->set_Use_Time(total_Use_Time);

    this->wait_Start_Time = kernel->stats->totalTicks;

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

- **Thread.cc** Thread::Sleep (bool finishing)

  將原本的 thread 放入 waiting state 中，更新 burst time，並設定累積執行時間為 0。

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

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

    DEBUG(dbgThread, "Sleeping thread: " << name);
    DEBUG(dbgTraCode, "In Thread::Sleep, Sleeping thread: " << name << ", " << kernel->stats->totalTicks);

    // running -> waiting, update burst time

    double total_Use_Time = this->use_Time + (kernel->stats->totalTicks - this->burst_Start_Time);
    this->set_Use_Time(total_Use_Time);

    double New_Burst_Time = 0.5 * this->burst_Time + 0.5 * this->use_Time;
    this->set_Burst_Time(New_Burst_Time);

    if(finishing == 0 && this->use_Time != 0) {
        DEBUG(dbgScheduler, "[D] Tick [" << kernel->stats->totalTicks << "]: Thread [" << this->getID()
                                         << "] update approximate burst time, from: [" << this->burst_Time
                                         << "], add [" << this->use_Time  << "], to [" << New_Burst_Time << "]");
    }
```

- **Thread.cc** Thread::aging(int T)

  在 ready queue 中的 thread 連續等待超過 1500 ticks，將 priority 增加 10。

```
bool Thread::aging(int T) {
    this->set_Wait_Time(T);

    if(this->wait_Time < 1500) return FALSE;

    this->wait_Start_Time = kernel->stats->totalTicks;

    int old_Priority = this->priority;
    int new_Priority = old_Priority + 10;
    if(new_Priority > 149) new_Priority = 149;

    if(new_Priority != old_Priority) {

        DEBUG(dbgScheduler, "[C] Tick [" << kernel->stats->totalTicks << "]: Thread [" << this->getID()
                                         << "] changes its priority from [" << old_Priority << "] to ["
                                         << new_Priority << "]");
        //
        //this->set_Wait_Time(this->wait_Time % 1500);
        this->set_Wait_Time(0);
        this->priority = new_Priority;
    }

    if((old_Priority < 100 && new_Priority >= 100) || (old_Priority < 50 && new_Priority >= 50)) {
        return TRUE;
    }
    return FALSE;
}
```

2. Add a command line argument

- Kernel.cc Kernel::kernel()

  增加判斷-ep 的指令

```
83   #endif
84                cout << "Partial usage: nachos [-n #] [-m #]\n";
85            } else if (strcmp(argv[i], "-ep") == 0) {   // TODO
86                priorityFlag = TRUE;
87                execfile[++execfileNum] = argv[++i];
88                exec_Priority[execfileNum] = atoi(argv[++i]);
89            }
```

3. Add a debugging flag

- Debug.h 新增一行判斷指令

```
const char dbgScheduler = 'z';
```

- Scheduler.cc ReadyToRun()

  [A]debug info : 判斷該放入哪個 ready queue 時印出 debug 資訊

- Scheduler.cc FindNextToRun()

  [B]debug info : 決定哪個 thread 是下一個執行時印出 debug 資訊

- Scheduler.cc Run(Thread *nextThread, bool finishing)

  [E]debug info : 印出當前 thread 的執行時間

- Scheduler.cc Scheduler::UpdatePriority()

  [X]debug info : 若判斷更新後的 priority 不適用當前 queue，則印出 remove 資訊。

- Thread.cc Thread::Sleep(bool finishing)
  [D]debug info : context switch 且當前 thread 未做完(finishing == false)，更新 burst time 並印出 debug 訊息。
- Thread.cc Thread::aging(int T)
  [C]debug info : 若 waiting time>1500 更新 priority，印出 debug 資訊。

4. 成果