NACHOS-4.0

MP3

第 21 組

106062173 吳東弦(一起 TRACE CODE,負責整理 REPORT)

106062271 郭子豪(一起 TRACE CODE,負責實作 CODING)

一、 trace code

1. New→Ready

（1） Kernel::ExecAll()

在 ExecAll()中，先跑一個回圈（Exec()）將要執行之程式的執行緒建立好，接著準備將 main thread 結束掉（Finish()）。

（2） Kernel::Exec(char*)

在 Exec()中，模擬利用 main thread 複製（Fork()）出將要執行之程式的 thread，並分配各自所需的分頁表（AddrSpace()）。

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

在 Fork()中，先把 thread 的 PCB 資訊初始化（StackAllocate()），再將其加入 ready queue 中（ReadyToRun()）。

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

在 StackAllocate()中，先分配記憶體空間（AllocBoundedArray()），在將 PCB 的暫存器狀態（machineState[]）初始化為對應 stack 位置。PCstate 對應到 ThreadRoot（stack top）；StartupPCState 對應到 ThreadBegin……。

（5） Scheduler::ReadyToRun(Thread*)

在 ReadyToRun()中，把 thread 的狀態設定為 ready（setStatus()），並將其加入 ready queue 中（readyList）。

2. Running→Ready

（1） Machine::Run()

在 Run()中，會不斷去讀取 thread 的指令（OneInstruction()），每讀取一次指令，就會遞進一次 tick（OneTick()）。

（2） Interrupt::OneTick()

在 OneTick()中，會先依照 kernel mode 或 user mode 遞進 totalTicks，再去檢查是否有 I/O 做完，如果有的話要去對硬體做中斷排程（CheckIfDue()），最後去檢查 Round-Robin 的 time quantum 是否用完（yieldOnReturn），如果用完的話要將 context switch 加入排程（Yield()）。

（3） Thread::Yield()

在 Yield()中，會去檢查是否有其他的 thread 要執行（FindNextToRun()），如果有的話就會將它加入排程（Run）。

（4） Scheduler::FindNextToRun()

在 FindNextToRun()中，會去檢查 ready queue 是否有其他 thread 要去執行（IsEmpty()），如果有的話就將它取出來（RemoveFront()）。

（5） Scheduler::ReadyToRun(Thread*)

在 ReadyToRun()中，把要 switch 出去的 thread 狀態設定為 ready（setStatus()），並將其放回 ready queue 中（readyList）。

（6） Scheduler::Run(Thread*, bool)

在 Run()中，先將 thread 狀態存回記憶體中，並檢查是否發生堆疊溢位（CheckOverflow()），再 switch 出去；當 switch 回來後，則將 thread 狀態讀回暫存器中。而在實體機器進行 context switch 後（SWITCH），虛擬機中的下一個 thread 也會開始執行。

3. Running→Waiting

（1） SynchConsoleOutput::PutChar(char)

在 PutChar()中，先取得設備的 mutex（lock）後，執行 I/O（PutChar()），等到 I/O 結束發出中斷指令（waitFor）再釋放 mutex。

（2） Semaphore::P()

在 P()中，就是 semaphore 的 wait 機制，當 value 大於零表示有資源可以使用；當 value 等於零表示目前沒有資源，會把 thread 加入此 semaphore 的等待隊列（Append()），並且讓他進 waiting queue 睡眠（Sleep()）。

（3） SynchList<T>::Append(T)

在 Append()中，實作加入 semaphore 等待隊列（List）的過程，如果 queue 是空的，加入的 thread 就是第一個元素；如果 queue 不是空的，就會排到隊列的最後面。

（4） Thread::Sleep(bool)

在 sleep()中，先把當前 thread 設為 blocked，並加入 waiting queue，再去檢查是否有其他的 thread 要執行（FindNextToRun()），如果有的話就會將它加入排程（Run）。

（5） Scheduler::FindNextToRun()

在 FindNextToRun()中，會去檢查 ready queue 是否有其他 thread 要去執行（IsEmpty()），如果有的話就將它取出來（RemoveFront()），準備進行 context switch。

（6） Scheduler::Run(Thread*, bool)

在 Run()中，先將 thread 狀態存回記憶體中，並檢查是否發生堆疊溢位（CheckOverflow()），再 switch 出去；當 switch 回來後，則將 thread 狀態讀回暫存器中。而在實體機器進行 context switch 後（SWITCH），虛擬機中的下一個 thread 也會開始執行。

4. Waiting→Ready

（1） Semaphore::V()

在 V()中，當 thread 做完 I/O，發出中斷訊號後，OS 就會將他放回 ready queue（ReadyToRun()），並釋出自己所佔用的資源（value）。

（2） Scheduler::ReadyToRun(Thread*)

在 ReadyToRun()中，把 thread 的狀態設定為 ready（setStatus()），並將其放回 ready queue 中（readyList）。

5. Running→Terminated

（1） ExceptionHandler(ExceptionType) case SC_Exit

在 SC_Exit 中，當 CPU 暫存器指向（ReadRegister()）結束程式的 system call 時，就會去將 thread 給結束掉（Finish()）。

（2） Thread::Finish()

在 Finish()中將結束 thread 的訊號(TRUE)傳給了 Sleep()。

（3） Thread::Sleep(bool)

在 sleep()中，先把當前 thread 設為 blocked，再去檢查是否有其他的 thread 要執行（FindNextToRun()），如果有的話就會將它加入排程（Run），並將結束訊號傳下去（finishing）。

（4） Scheduler::FindNextToRun()

在 FindNextToRun()中，會去檢查 ready queue 是否有其他 thread 要去執行（IsEmpty()），如果有的話就將它取出來（RemoveFront()），準備進行 context switch。

（5） Scheduler::Run(Thread*, bool)

在 Run()中，先標記當前要結束的 thread（toBeDestroyed），並將 thread 狀態存回記憶體中，檢查是否發生堆疊溢位（CheckOverflow()），再 switch 出去；當 switch 回來將 thread 狀態讀回暫存器後，就將剛剛的 thread 給結束掉（CheckToBeDestroyed()）。而在實體機器進行 context switch 後（SWITCH），虛擬機中的下一個 thread 也會開始執行。

6. Ready→Running

（1） Scheduler::FindNextToRun()

在 FindNextToRun()中，會去檢查 ready queue 是否有其他 thread 要去執行（IsEmpty()），如果有的話就將它取出來（RemoveFront()），準備進行 context switch。

（2） Scheduler::Run(Thread*, bool)

在 Run()中，要執行的 thread 在實體機器 context switch 後（SWITCH），虛擬機中的下一個 thread 也會開始執行。

（3） SWITCH(Thread*, Thread*)

- 先將要 switch 出去的 thread 暫存器資訊存回 PCB

```
movl    %eax,_eax_save   // 先把 eax 的 data 存起來
movl    4(%esp),%eax   // 借 eax，讓 eax 取得 4(%esp)的值
movl    %ebx,_EBX(%eax)   // 依照 stack top 對應偏移值將 data 存回 PCB
        (ecx、edx、esi、edi、ebp（base pointer）、esp（stack pointer）)
movl    _eax_save,%ebx    // 借 ebx，讓 ebx 讀取 eax 原本的 data
movl    %ebx,_EAX(%eax)   // eax 原本的 data 也要記得存回 PCB
movl    0(%esp),%ebx   // 借 ebx，讓 ebx 指向 stack top
movl    %ebx,_PC(%eax)   // 讓 PCB 的_PC 取得 stack top
```

- 再將要 switch 進來的 thread 之 PCB 資訊讀進暫存器

```
movl    8(%esp),%eax   // 借 eax，讓 eax 指向 8(%esp)
movl    _EAX(%eax),%ebx   // 借 ebx，讓 ebx 取得 PCB 的 EAX 資訊
movl    %ebx,_eax_save   // 先把 PCB 的 EAX 資訊存起來
movl    _EBX(%eax),%ebx   // 依照 PCB 對應偏移資訊讀進暫存器中
        (_ECX、_EDX、_ESI、_EDI、_EBP（base）、_ESP（stack））
movl    _PC(%eax),%eax   // 借 eax，讓 eax 取得_PC 資訊
movl    %eax,4(%esp)   // 讓 4(%esp)取得 eax 的值
movl    _eax_save,%eax   // PCB 的 EAX 資訊也要記得讀進 eax
ret   // return 同時，PC 正指向新 thread 的 ThreadRoot()
```

- ThreadRoot()呼叫對應的函數

```
call    *StartupPC   // 對應 ThreadBegin()
call    *InitialPC   // 對應 ForkExcute()
call    *WhenDonePC   // 對應 ThreadFinish()
```

- ForkExecute(Thread *t)

```
t->space->Execute(t->getName());   // 準備執行新 thread
```

- AddrSpace::Execute(char* fileName)

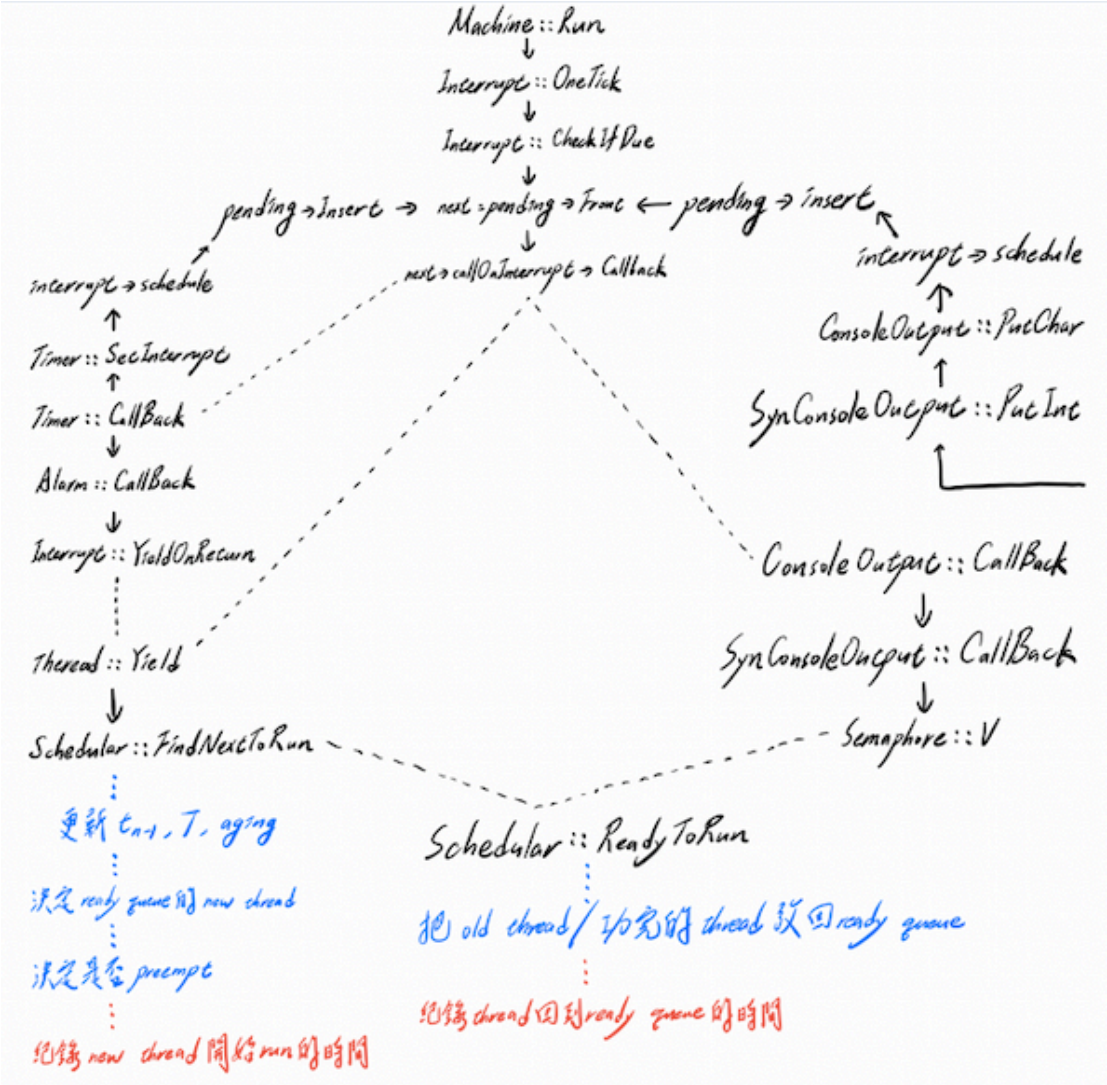
```
kernel->machine->Run();   // 開始執行新 thread
```

（4） Machine::Run()

thread 回到 running 狀態，形成一個 context switch 循環。

二、 Implementation

1. 流程圖



2. 黑色部分

黑色部分流程為 nachos 原本的 code，從 Machine::Run()之後分別有兩個部分會影響 thread 在 queue 間的變動。

左邊表示 Round-Robin 機制觸發 timer 發出中斷，FindNextToRun 會從 ready queue 挑一個 thread 準備切換至 running state，ReadyToRun 會把原本在 running state 的 thread 放回 ready queue。

右邊表示有 thread 想做 I/O，當 thread 要做 I/O 時，FindNextToRun 也會從 ready queue 挑一個 thread 準備切換至 running state（不在圖中），做完 I/O 後觸發中斷，ReadyToRun 會把原本在 waiting queue 的 thread 放回 ready queue。

3. 藍色部分（實作 thread 要切換 state / queue）

（1） FindNextToRun

```
Scheduler::FindNextToRun () {
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    int lv;
    Thread* ret = NULL;
    Thread* cur_thread = kernel->currentThread;
    int cur_thread_priority = cur_thread->get_priority();
    scheduling()

    if(!L[1]->IsEmpty()) {
        ret = select_L1();
        lv = 1;
    } else if(!L[2]->IsEmpty()) {
        ret = select_L2();
        lv = 2;
    } else if(!L[3]->IsEmpty()) {
        ret = L[3]->Front();
        lv = 3;
    } else {
        return NULL;
    }
    if(cur_thread->getStatus() != BLOCKED) { //BLOCKED mean that  thread's status is wait.If a thread is BLOCKED ,it should choose a threa
queue.
        if(cur_thread_priority > 99) {
            if(ret->get_priority() < 100 || ret->get_t() > cur_thread->get_t()) {
                ret = NULL;
            }
        } else if(cur_thread_priority > 49) {
            if(ret->get_priority() < 100 && ret != cur_thread) {
                ret = NULL;
            }
        }
    }
    if(ret != NULL) {
        L[lv]->Remove(ret);
        DEBUG(z, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" << ret->getID() << "] is removed from queue L[" << lv << "]");
        ret->set_just_run(kernel->stats->totalTicks);
    }
    return ret;
```

- old thread 準備要離開 running state，可能是因為 time quantum 用完，或是準備要做 I/O。
- 接著從 ready queue 中挑出 new thread，L[1]優先於 L[2]優先於 L[3]。
- 如果 new thread 存在，且 old thread 不是要進行 I/O，new thread 就與 old thread 比較優先序位決定是否要 preemptive。
- 如果要 preemptive，紀錄 new thread 開始 running 的時間

（2） ReadyToRun

```
void
Scheduler::ReadyToRun (Thread *thread) {
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
    thread->setStatus(READY);
    thread->set_just_ready(kernel->stats->totalTicks); //set the time that thread add into ready queue
    if(thread->get_priority() > 99) {
        DEBUG(z, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" << thread->getID() << "] is inserted into queue L[1]");
        L[1]->Append(thread);
    } else if(thread->get_priority() > 49) {
        DEBUG(z, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" << thread->getID() << "] is inserted into queue L[2]");
        L[2]->Append(thread);
    } else {
        DEBUG(z, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" << thread->getID() << "] is inserted into queue L[3]");
        L[3]->Append(thread);
    }
    //cout << "Putting thread on ready list: " << thread->getName() << endl ;
//    thread->setStatus(READY);
//    readyList->Append(thread);
//
}
```

- old thread 準備要回 ready queue，可能是因為 time quantum 用完（來自 running state），或是 I/O 做完（來自 waiting queue）。
- 先將 old thread 狀態設為 ready，並記錄回到 ready queue 的時間。
- 再將 old thread 放回對應的 ready queue（L[1]、L[2]、L[3]）。

4. 紅色部分（Update T / t / Aging）

```
void Scheduler::scheduling() {
    Thread* cur_thread = kernel->currentThread;
    update_T();

    if(cur_thread->getStatus() == BLOCKED)
        update_t();

    update_aging();
}
```

- old thread 準備要離開 running state，紀錄目前累計 cpu burst time（T），同時紀錄 aging。
- 如果要做 I/O，先用剛剛紀錄的累計 cpu burst time（T）計算新的 approximate burst time（t），再將累計 cpu burst time（T）歸零。

三、 Summary & FeedBack

本次 trace code 的部分與 MP1、MP2 多有重疊，並沒有很難。在實作的部分，遇到主要的瓶頸是排程的詳細規則，看完 Ch5 後即使知道大致的理論，然而在實務上，可以有不同的理解，從 iLMS 討論區的熱絡程度可見一斑。其一是對於 T、t、aging 理解錯誤，導致 code 多次修改，其二是對於 preemptive 時機不確定，導致不知道要把 code 寫在哪裡，後來知道實作允許 I/O 做完的 interrupt 在 alarm 後才觸發，那麼基本上都把 code 寫在 schedular.cc 就可以了。