Operating System HW－CPU Scheduling

Group 18

106061132 黃友廷 106061146 陳兆廷

1. Code Tracing and Implementation

(1). New→Ready

這個步驟在模擬有個新的 Thread 要進入 Ready 的階段。因此要創一個新的 thread 並將這個 thread 放到 ready queue 等待 scheduler 去選擇。

```
int main(int argc, char **argv)
{
    int i;
    char *debugArg = "";

    // before anything else, initialize the debugging system
    for (i = 1; i < argc; i++) {···
    debug = new Debug(debugArg);

    DEBUG(dbgThread, "Entering main");

    kernel = new KernelType(argc, argv);
    kernel->Initialize();

    CallOnUserAbort(Cleanup);       // if user hits ctl-C


    // kernel->SelfTest();

    //<REPORT>
    // kernel->Run();
    kernel->InitializeAllThreads();
    //<REPORT>


    return 0;       You, 3 weeks ago • TODO finish
}
```

這份作業的 main()是在 threads/main.cc 裡面，而這個 main()在第一個 for loop 會把 debug argument 記錄到 debugArg 這個 list 裡面，然後就會使用 KernelType()這個建構子 new 一個 kernel 出來，最後面在使用 kernel->InitializeAllThreads()對各個 thread 進行初始化。

```
UserProgKernel::UserProgKernel(int argc, char **argv)        You, 3 weeks ago • T
        : ThreadedKernel(argc, argv)
{
    debugUserProg = FALSE;
    consoleIn = NULL;
    consoleOut = NULL;
    for (int i = 0; i < NumPhysPages; i++)
        PhysicalPageUsed[NumPhysPages] = FALSE;
    execfileNum = 0;
    for (int i = 1; i < argc; i++) {
        if (strcmp(argv[i], "-s") == 0) {···
        else if (strcmp(argv[i], "-e") == 0) {···
        //<TODO>
        // Get execfile & its priority & burst time from argv, then save them.
        else if (strcmp(argv[i], "-epb") == 0) {
            execfile[++execfileNum] = argv[++i];
            threadPriority[execfileNum] = atoi(argv[++i]);
            threadRemainingBurstTime[execfileNum] = atoi(argv[++i]);
        }
        //<TODO>
```

在這次作業我們所使用的 KernelType 是 UserProgKernel，而這個建構子會把 argv[]的輸入存到 thread 的對應資訊裡面，exefile[]是負責存各個 thread 所需要執行的 file，threadPriority[]則是　存 thread 的初始優先度，threadRemainingBurstTime[]則是存一開始預測所需的 CPU burst

time，execfileNum 是存總共有幾個 threadfile 需要被執行。

- UserProgKernel::InitializeAllThreads()

```
void UserProgKernel::InitializeAllThreads()      You, seconds ago • Uncommitted changes
{
    for (int i = 1; i <= execfileNum; i++){
        // cout << "execfile[" << i << "]: " << execfile[i] << " start " << endl;
        int a = InitializeOneThread(execfile[i], threadPriority[i], threadRemainingBurstTime[i]);
        // cout << "execfile[" << i << "]: " << execfile[i] << " end "<< endl;
    }
    // After InitializeAllThreads(), let main thread be terminated that we can start to run our thread.
    currentThread->Finish();
    // kernel->machine->Run();
}
```

第一個 for loop 會去使用 UserProgKernel::InitializeOneThread()創造各一個 thread 給每一個要被執行的 file。在這次作業中就是 hw2_test1.c, hw2_test2.c。currentThread->Finish()會將現在正在執行的 main thread 停止並且刪除，去切換成在 Ready queue 中應該被執行的 thread。

- UserProgKernel:: InitializeOneThread()

```
int UserProgKernel::InitializeOneThread(char* name, int priority, int burst_time)      You,
{
    //<TODO>
    // When each execfile comes to Exec function, Kernel helps to create a thread for it.
    // While creating a new thread, thread should be initialized, and then forked.
    t[threadNum] = new Thread(name, threadNum);
    t[threadNum]->set_Priority(priority);
    t[threadNum]->set_RunTime(0);
    t[threadNum]->set_RRTime(0);
    t[threadNum]->set_WaitTime(0);
    t[threadNum]->set_RemainingBurstTime(burst_time);
    t[threadNum]->space = new AddrSpace();
    t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);
    //<TODO>

    threadNum++;
    return threadNum - 1;
}
```

為了實作出一個 Multilevel Feedback Queue Scheduler 並有不同的 scheduling algorithm，一個 thread 會有 ID、Priority、RunTime、RRTime、WaitTime、RemainBurstTime 等資訊，並在此函式中定義。

ID	根據 thread 被加入的先後順序去設定 ID
Priority	決定 thread 要進入哪一個 level 的 ready queue
RunTime	thread 在 CPU 執行了多少時間，初始化為 0
RRTime	thread 在 Run Robin 下在 CPU 執行了多少時間，初始化為 0
WaitTime	thread 在其 ready queue 中等待了多少時間，初始化為 0
RemainBurstTime	thread 距離預測的 CPU burst time 還剩下多少時間，初始化為預測值

Thread::Fork()會去進行 Fork 產生一個子程序，這個子程序會去呼叫執行 ForkExecute 這個 function，t[threadNum]則是 ForkExecute 會用到的 argument。

◆ Thread::Fork()

```
void Thread::Fork(VoidFunctionPtr func, void *arg)      You, seconds ago • Uncommitted chan
{
    Interrupt *interrupt = kernel->interrupt;
    Scheduler *scheduler = kernel->scheduler;
    IntStatus oldLevel;

    DEBUG(dbgThread, "Forking thread: " << name << " f(a): " << (int) func << " " << arg);

    StackAllocate(func, arg);

    oldLevel = interrupt->SetLevel(IntOff);
    scheduler->ReadyToRun(this);    // ReadyToRun assumes that interrupts
                                    // are disabled!
    (void) interrupt->SetLevel(oldLevel);
}
```

呼叫 StackAllocate()去 allocate 一個 execute stack 去儲存 func 和 arg，然後使用 SetLevel()先將 interrupt 設為 disable 的狀態，接著呼叫 ReadyToRun 把 thread 放到適當的 ready queue，最後再將 interrupt 改回原本的狀態。

```
void ForkExecute(Thread *t)      You, seconds ago • Uncommitted changes
{
    // cout << "Thread: " << (void *) t << endl;
    DEBUG(dbgMLFQ, "ForkExecute => fork thread id: " << t->getID() << ", currentTick:
    //<TODO>
    // When Thread t goes to Running state in the first time, its file should be load
    // Hint: This function would not be called until Thread t is on running state.
    if (!t->space->Load(t->getName())) {
        cout << "inside !Load(FileName)" << endl;
        return;             // executable not found
    }
    t->space->Execute(t->getName());
    //<TODO>
}
```

ForkExecute 就是先使用 space->Load()把對應檔案打開並且讀入，再使用 space->Execute()去執行這個檔案。

- Thread::StackAllocate()

```
void Thread::StackAllocate (VoidFunctionPtr func, void *arg)
{
    stack = (int *) AllocBoundedArray(StackSize * sizeof(int));
```

```
#else
    machineState[PCState] =(void *)ThreadRoot;
    machineState[StartupPCState] = (void *)ThreadBegin;
    machineState[InitialPCState] = (void *)func;
    machineState[InitialArgState] = (void *)arg;
    machineState[WhenDonePCState] = (void *)ThreadFinish;
#endif
}
```

StackAllocate 主要是先用 AllocBoundedArray 去 allocate 一個 bounded array 給 thread 的 stack 使用，然後再將各個 registers (machineState[])去賦予所需要的數值。

- Scheduler::ReadyToRun()

```
void Scheduler::ReadyToRun (Thread *thread)     You, seconds ago • Uncommitted changes
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    // DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());

    Statistics* stats = kernel->stats;
    //<TODO>
    int queuelevel = 0;
    // According to priority of Thread, put them into corresponding ReadyQueue.
    // After inserting Thread into ReadyQueue, don't forget to reset some values.
    // Hint: L2 ReadyQueue is preemptive priority.
    // When putting a new thread into L1 ReadyQueue, you need to check whether preemptior
    thread->setStatus(READY);
    if (thread->get_Priority() >= 0 && thread->get_Priority() <= 49) {
        L3ReadyQueue->Append(thread);
        queuelevel = 3;
    } else if (thread->get_Priority() >= 50 && thread->get_Priority() <= 99) {
        L2ReadyQueue->Insert(thread);
        queuelevel = 2;
    } else if (thread->get_Priority() >= 100 && thread->get_Priority() <= 149) {
        L1ReadyQueue->Insert(thread);
        queuelevel = 1;
    } else {
        cout << "NOT VALID PRIORITY" << endl;
    }
    DEBUG(dbgMLFQ, "[InsertToQueue] Tick ["<< stats->totalTicks << "]: Thread [" << threa
    //<TODO>
    // readyList->Append(thread);
}
```

ReadyToRun 就是將傳進來的 thread 的 Status 設定為 READY，並且依據他的 priority 去放到對應的 ready queue 裡等待 scheduler 去選擇並執行。

(2). Running→Ready

- Machine::Run()

```
void Machine::Run()
{
    Instruction *instr = new Instruction;  // storage for decoded instruction

    if (debug->IsEnabled('m')) {
        cout << "Starting program in thread: " << kernel->currentThread->getName();
        cout << ", at time: " << kernel->stats->totalTicks << "\n";
    }
    kernel->interrupt->setStatus(UserMode);
    for (;;) {
        OneInstruction(instr);
        kernel->interrupt->OneTick();
        if (singleStep && (runUntilTime <= kernel->stats->totalTicks))
            Debugger();
    }
}
```

功能是去模擬執行 user 的 program 在 NachOS 上，setStatus(UserMode)會先將 OS 的設定成 user mode，接著進入 for loop 去呼叫 OneInstruction()去從已經被 compile 成 binary file 的 User program 中讀取 instruction 並執行。Interrupt->OneTick()則是會去檢查是否有 interrupt 被呼叫。

  - Interrupt::OneTick()

```
void Interrupt::OneTick()
{
    MachineStatus oldStatus = status;
    Statistics *stats = kernel->stats;

// advance simulated time
    if (status == SystemMode) {
        stats->totalTicks += SystemTick;
        stats->systemTicks += SystemTick;
    } else {                    // USER_PROGRAM
        stats->totalTicks += UserTick;
        stats->userTicks += UserTick;
    }
    DEBUG(dbgInt, "== Tick " << stats->totalTicks << " ==");

// check any pending interrupts are now ready to fire
    ChangeLevel(IntOn, IntOff); // first, turn off interrupts
                // (interrupt handlers run with
                // interrupts disabled)
    CheckIfDue(FALSE);      // check for pending interrupts
    ChangeLevel(IntOff, IntOn); // re-enable interrupts
    if (yieldOnReturn) {    // if the timer device handler asked
                    // for a context switch, ok to do it now
        yieldOnReturn = FALSE;
        status = SystemMode;        // yield is a kernel routine
        kernel->currentThread->Yield();
        status = oldStatus;
    }
}       You, 2 weeks ago • TODO finish
```

第一個 if else 主要是去計算目前的 tick 數量，幫助我們去了解各個事件的發生時間。再來會先將 interrupt 的狀態設定為 disable，接著 CheckIfDue()檢查是否有 interrupt 需要被執行，最後再將 interrupt 的狀態改回 enable。最後面的 if 則是去判斷目前這個 thread 需不需要被換成其他在 ready queue 等待的 thread，如果需要的話，就會使用 currentThread->Yield()去進行 thread 的交換。而這裡的 yeildOneReturn 是由 Alarm::CallBack()去決定的，如果 thread 需要進行交換就會把 yeildOneReturn 設為 true，否則 yeildOneReturn 就是 false。

- Thread::Yield()

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

    ASSERT(this == kernel->currentThread);

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

    //<TODO>
    // 1. Put current_thread in running state to ready state
    // 2. Then, find next thread from ready state to push on running state
    // 3. After resetting some value of current_thread, then context switch
    Statistics* stats = kernel->stats;
    nextThread = kernel->scheduler->FindNextToRun();
    if (nextThread != NULL) {
        DEBUG(dbgMLFQ, "[ContexSwitch] Tick ["<< stats->totalTicks << "]: Thread [" << n
        kernel->currentThread->set_RRTime(0);
        kernel->scheduler->ReadyToRun(this);
        nextThread->set_WaitTime(0);
        kernel->scheduler->Run(nextThread, FALSE);
    }
        You, seconds ago • Uncommitted changes
    //<TODO>

    (void) kernel->interrupt->SetLevel(oldLevel);
}
```

先用 Scheduler::FindNextToRun()去從 Ready queue 找出下一個要被執行的 thread，如果下一個 thread 存在就會把目前 thread 的 RRTime 歸 0，接著使用 Scheduler::ReadyToRun()去把目前的 thread 放進適當的 ready queue 裡。然後把下一個要被執行的 thread 的 WaitTime 歸 0，並且使用 scheduler->Run()進行 context switch，把 CPU 中執行的 thread 換成下一個 thread。

- Scheduler::FindNextToRun()

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

    /*if (readyList->IsEmpty()) {
    return NULL;
    } else {
        return readyList->RemoveFront();
    }*/

    //<TODO>
    Statistics* stats = kernel->stats;
    Thread* thread;
    // a.k.a. Find Next (Thread in ReadyQueue) to Run
    if (! L1ReadyQueue->IsEmpty()) {
        thread = L1ReadyQueue->RemoveFront();
    DEBUG(dbgMLFQ, "[RemoveFromQueue] Tick ["<< stats->totalTicks << "]: Thread [" << thread->getID() <<"] is removed from queue L1");
        return thread;
    } else if (! L2ReadyQueue->IsEmpty()) {
        thread = L2ReadyQueue->RemoveFront();
    DEBUG(dbgMLFQ, "[RemoveFromQueue] Tick ["<< stats->totalTicks << "]: Thread [" << thread->getID() <<"] is removed from queue L2");
        return thread;
    } else if (! L3ReadyQueue->IsEmpty()) {
    thread = L3ReadyQueue->RemoveFront();
        DEBUG(dbgMLFQ, "[RemoveFromQueue] Tick ["<< stats->totalTicks << "]: Thread [" << thread->getID() <<"] is removed from queue L3");
        return thread;
    } else {
        return NULL;
    }
    //<TODO>
}
```

  Scheduler::FindNextToRun()的實作就是依照 L1、L2、L3 的順序，取出 queue 裡最高順位的 Thread 執行回傳。
- Scheduler::ReadyToRun()
  已於(1)解釋。
- Scheduler::Run()
  將於(6)連同 Switch 解釋。

(3). Running→Waiting

當目前的 thread 需要 I/O 時就會進入 waiting state，等待 I/O 完成後才會回到 ready queue 等待 scheduler 去選擇。而這次作業的 testfile 裡面都有用到 PrintInt()這個 function 去印數字在 terminal 上，因此會需要用到 I/O。而 PrintInt()實際上是用 system call 的方式去實做，所以所觸發的途徑會與 HW1 類似。

- ExceptionHandler()

```
void ExceptionHandler(ExceptionType which)
{
    int type = kernel->machine->ReadRegister(2);
    int val, status;

    switch (which) {
    case SyscallException:
        switch(type) {
            case SC_Halt:
                DEBUG(dbgAddr, "Shutdown, initiated by user program.\n");
                kernel->interrupt->Halt();
                break;
            case SC_PrintInt:
                // DEBUG(dbgMLFQ, "Print Int");
                val=kernel->machine->ReadRegister(4);
                kernel->synchConsoleOut->PutInt(val);
                DEBUG(dbgMLFQ, "\n");
                return;
```

因為 argument 是從 register 4 開始往後存，所以第一個 argument 就是存在 register 4，因此會先使用 ReadRegister(4)去把 register 的數值讀出來。接著使用 PutInt()這個 function 去完成這個 system call。

- SynchConsoleOutput::PutInt()

```
void SynchConsoleOutput::PutInt(int value){
    char str[10];
    int index = 0;
    sprintf(str, "%d\r\n\0", value);

    lock->Acquire();
    do{
        consoleOutput->PutChar(str[index]);
        index++;
        waitFor->P();
    } while (str[index] != '\0');
    lock->Release();
}
```

PutInt()一開始會先用 sprintf 將要印出的數字轉換成完整要在 terminal 上顯示的 string，然後使用 lock->Acquire 和 lock->Release() (這兩個 function 是 Atomical)去確保同時只會有一個 thread 在 terminal 上印訊息。

- SynchConsoleOutput::PutChar()
  PutInt()使用 while loop 則是使用 PutChar()去把 char 一個一個印出來，waitFor->P()則是為了確保輸出在 terminal 時 I/O 有完整執行完才會繼續印下一個 char。
  - Semaphore::P()

```
void Semaphore::P()
{
    Interrupt *interrupt = kernel->interrupt;
    Thread *currentThread = kernel->currentThread;      You, 2 weeks ago

    // disable interrupts
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    while (value == 0) {        // semaphore not available
        queue->Append(currentThread);   // so go to sleep
        currentThread->Sleep(FALSE);
        // cout << "Run to Waiting over" << endl;
    }
    value--;            // semaphore available, consume its value

    // re-enable interrupts
    (void) interrupt->SetLevel(oldLevel);
}
```

當 value = 0 時就是代表現在 semaphore 是 not availabe，所以會將 currentThread 放到 I/O waiting queue 裡面等待，並且使用 thread->sleep()去找出下一個要執行的 thread，進行交換。而當 value>0 時就會減掉一個 value，繼續進行。

```
1[InsertToQueue] Tick [8683]: Thread [1] is inserted into queue L1
[RemoveFromQueue] Tick [8683]: Thread [1] is removed from queue L1
[InsertToQueue] Tick [8694]: Thread [1] is inserted into queue L1
[RemoveFromQueue] Tick [8694]: Thread [1] is removed from queue L1

[InsertToQueue] Tick [8705]: Thread [1] is inserted into queue L1
[RemoveFromQueue] Tick [8705]: Thread [1] is removed from queue L1
```

這邊可以注意到印一次數字會需要印三個 char(%d, \r, \n)，這也是為甚麼印一次完整的數字會讓 thread 進入 Ready queue 三次。

- SynchList<T>::Append()
  在這裡將 currentThread 用 Append()加入 ready queue 代表即將被 wait 的 Thread 要被加入回到 ready queue 了。
- Thread::Sleep()

```
void Thread::Sleep (bool finishing)      You, seconds ago • Uncommitted changes
{
    Thread *nextThread;

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

    DEBUG(dbgThread, "Sleeping thread: " << name << ", ID: " << ID);

    status = BLOCKED;
    while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL)
        kernel->interrupt->Idle();  // no one to run, wait for an interruptd

    //<TODO>
    // In Thread::Sleep(finishing), we put the current_thread to waiting or terminated state (depend o
    // , and determine finishing on Scheduler::Run(nextThread, finishing), not here.
    // 1. Update RemainingBurstTime
    // 2. Reset some value of current_thread, then context switch
    if (nextThread != this) {
        Statistics* stats = kernel->stats;
        DEBUG(dbgMLFQ, "[ContexSwitch] Tick ["<< stats->totalTicks << "]: Thread [" << nextThread->get
        if (this->get_RunTime() != 0) {
            DEBUG(dbgMLFQ, "[UpdateRemainingBurstTime] Tick ["<< stats->totalTicks << "]: Thread [" <<
            this->set_RemainingBurstTime(this->get_RemainingBurstTime() - this->get_RunTime());
            this->set_RunTime(0);
        }
        nextThread->set_RRTime(0);
        nextThread->set_WaitTime(0);
        kernel->scheduler->Run(nextThread, finishing);
    }
    //<TODO>
}
```

整體與 thread->Yeild()很像，依樣需要先用 FindNextToRun()去找出下一個要被執行的 thread，並且去更新 RemainingBurstTime、RunTime、RRTime、WaitTIme，最後在使用 scheduler->Run()去進行 context switch 以及執行 nextThread。唯一不同的是不需要將 currentThread 放到 ready queue 中，因為我們在 Semaphore->P()時就已經將他放在 I/O waiting queue 裡了。

- Scheduler::ReadyToRun()
  已於(1)解釋。
- Scheduler::Run()
  將於(6)連同 Switch 解釋。

(4). Waiting→Ready

這個步驟在模擬 Thread 從 waiting 進入 ready queue 的過程。

- Semaphore::V()

```
void
Semaphore::V()
{
    Interrupt *interrupt = kernel->interrupt;

    // disable interrupts
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    if (!queue->IsEmpty()) {  // make thread ready.
        kernel->scheduler->ReadyToRun(queue->RemoveFront());
        // cout << "Ready to Run over" << endl;
    }
    value++;

    // re-enable interrupts
    (void) interrupt->SetLevel(oldLevel);
}
```

Semaphore::V()與 P()做相反的事情。當 ready queue 還不是空的，會利用 ReadyToRun 呼叫 ready queue 的下一位執行，若是空的就將 value 加上一單位。相同於 P()，也會將 Interrupt 關閉再開啟。

- Scheduler::ReadyToRun()
  已於(1)解釋。

(5). Running→Terminated

這個步驟在模擬一個外來指令終止了一個 Thread，將它從 Running state 直接 Terminated。

- ExceptionHandler()

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

啟動這個終止程序的是一個 System Call。藉由 SC_Exit，將 Thread 直接變為 Finish()的狀態。

- Thread::Finish()

```
void
Thread::Finish ()
{
    (void) kernel->interrupt->SetLevel(IntOff);
    ASSERT(this == kernel->currentThread);

    DEBUG(dbgThread, "Finishing thread: " << name << ", ID: " << ID);

    Sleep(TRUE);               // invokes SWITCH
    // not reached
}
```

Thread::Finish()會將這個 Thread 進入 Sleep。

- Thread::Sleep()

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

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

    DEBUG(dbgThread, "Sleeping thread: " << name << ", ID: " << ID);

    status = BLOCKED;
    while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL)
        kernel->interrupt->Idle();  // no one to run, wait for an interruptd

    //<TODO>
    // In Thread::Sleep(finishing), we put the current_thread to waiting or terminated state (depend on finishing)
    // , and determine finishing on Scheduler::Run(nextThread, finishing), not here.
    // 1. Update RemainingBurstTime
    // 2. Reset some value of current_thread, then context switch
    if (nextThread != this) {
        Statistics* stats = kernel->stats;
        DEBUG(dbgMLFQ, "[ContexSwitch] Tick ["<< stats->totalTicks << "]: Thread [" << nextThread->getID() <<"] is now selected for execut
        if (this->get_RunTime() != 0) {
            DEBUG(dbgMLFQ, "[UpdateRemainingBurstTime] Tick ["<< stats->totalTicks << "]: Thread [" << this->getID() <<"] update remaining
            this->set_RemainingBurstTime(this->get_RemainingBurstTime() - this->get_RunTime());
            this->set_RunTime(0);
        }
        nextThread->set_RRTime(0);
        nextThread->set_WaitTime(0);
        kernel->scheduler->Run(nextThread, finishing);
    }
    //<TODO>
}
```

首先要先把 Interrupt 關掉，接著利用前面提到的 FindNextToRun()去儲存下一個要執行的 Thread。如果下個 Thread 是其他的 Thread，就要有一系列的時間交接跟設定。先將這個 Thread 總結一下，把剩餘的時間計算好，然後歸零；接著初始下個 Thread 的 RoundRobin 時間值跟等待的時間，最後執行下個 Thread。

- Scheduler::FindNextToRun()

已在(2)解釋。

- Scheduler::Run()
  在(6)隨著 SWITCH 解釋。

(6). Ready→Running

這個步驟在模擬 Thread 在 Ready queue 中被選出後執行。首先如同上個步驟中的 Finish()/Sleep，當這個 Thread 因為事情而結束，輪到下一個 Thread 執行時，會經過 Scheduler::FindNextToRun()挑選出下一個要執行的 Thread，挑選完後用 Scheduler::Run()執行。

- Scheduler::FindNextToRun()
  已於(2)解釋。
  - Scheduler::Run()

```
void
Scheduler::Run (Thread *nextThread, bool finishing)
{
    Thread *oldThread = kernel->currentThread;

//  cout << "Current Thread" <<oldThread->getName() << "    Next Thread"<<nextThread->getName()<<endl;

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

    if (finishing) {    // mark that we need to delete current thread
        ASSERT(toBeDestroyed == NULL);
        toBeDestroyed = oldThread;
    }
```

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

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

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

    // This is a machine-dependent assembly language routine defined
    // in switch.s.  You may have to think
    // a bit to figure out what happens after this, both from the point
    // of view of the thread and from the perspective of the "outside world".

    cout << "Switching from: " << oldThread->getID() << " to: " << nextThread->getID() << endl;
    SWITCH(oldThread, nextThread);

    // we're back, running oldThread

    // interrupts are off when we return from switch!
    ASSERT(kernel->interrupt->getLevel() == IntOff);

    DEBUG(dbgThread, "Now in thread: " << kernel->currentThread->getID());

    CheckToBeDestroyed();       // check if thread we were running
                                // before this one has finished
                                // and needs to be cleaned up
```

假設這個 Thread 已經該結束了，就把它刪除掉(toBeDestroyed)。接著將新的 Thread 變為現在的 Thread(kernel->currentThread)並更新狀態成 Running。進入 SWITCH 以組合語言執行 Switch 的工作。

```
/* 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.
*/
```

在 SWITCH 一開始，t2 是存在 8(esp)，t1 存在 4(esp)，而(esp)為 return 的位址。ESP 為 Stack Pointer。

```
336         .comm   _eax_save,4
337
338         .globl  SWITCH
339     SWITCH:
340         movl    %eax,_eax_save          # save the value of eax
341         movl    4(%esp),%eax            # move pointer to t1 into eax
342         movl    %ebx,_EBX(%eax)         # save registers
343         movl    %ecx,_ECX(%eax)
344         movl    %edx,_EDX(%eax)
345         movl    %esi,_ESI(%eax)
346         movl    %edi,_EDI(%eax)
347         movl    %ebp,_EBP(%eax)
348         movl    %esp,_ESP(%eax)         # save stack pointer
349         movl    _eax_save,%ebx          # get the saved value of eax
350         movl    %ebx,_EAX(%eax)         # store it
351         movl    0(%esp),%ebx            # get return address from stack into ebx
352         movl    %ebx,_PC(%eax)          # save it into the pc storage
353
354         movl    8(%esp),%eax            # move pointer to t2 into eax
355
356         movl    _EAX(%eax),%ebx         # get new value for eax into ebx
357         movl    %ebx,_eax_save          # save it
358         movl    _EBX(%eax),%ebx         # retore old registers
359         movl    _ECX(%eax),%ecx
360         movl    _EDX(%eax),%edx
361         movl    _ESI(%eax),%esi
362         movl    _EDI(%eax),%edi
363         movl    _EBP(%eax),%ebp
364         movl    _ESP(%eax),%esp         # restore stack pointer
365         movl    _PC(%eax),%eax          # restore return address into eax
366         movl    %eax,4(%esp)            # copy over the ret address on the stack
367         movl    _eax_save,%eax
368
369         ret
```

340、341 行在把原本放在 reg eax 的值取出存在_eax_save 後，把 t1 放入 eax。

342～348、350 是將所有 reg 依照 eax 的位置存好，也就是 t1 的位置。

349 將原本 eax 的值，也就是_eax_save 存入 ebx。

351、352 把 return address 存入 ebx 中，並存在 PC storage 裡。

354 將 t2 放入 eax。

356、357 把新 eax 的值放入 ebx，並存在_eax_save。

358～364 把 t2 的 register value 存回原本的 register。

365 把原本存在 PC 的 return address 放回 eax。

366 把 eax 指向 4(%esp)，也就是把原本 return address 存在 stack 的位置(%esp)往上移了 4 bytes。

367 把原本 eax 的值放回 eax。

(7). Update priority

在每個 Thread 執行後，需要有個計時器將正在執行的 Thread 的 priority、timer 等做更新，也就是整個 scheduler 最重要的一個部分，若沒有這個更新，每一步都都會是一模一樣的。

- Alarm::CallBack()

```
void Alarm::CallBack()
{
    Interrupt *interrupt = kernel->interrupt;
    MachineStatus status = interrupt->getStatus();
    //<TODO>
    // In each 100 ticks,
    // 1. Update Priority
    // kernel->currentThread->set_RemainingBurstTime(kernel->currentThread->get_RemainingBurstTime() - 1
    Statistics* stats = kernel->stats;

    // 2. Update RunTime & RRTime
    // 3. Check Round Robin
    if (kernel->currentThread != NULL) {
        kernel->currentThread->set_RunTime(kernel->currentThread->get_RunTime()+100);
        kernel->currentThread->set_RRTime(kernel->currentThread->get_RRTime()+100);

        if (kernel->currentThread->get_Priority() >= 100 && kernel->scheduler->TotalFront() != NULL && k
        (kernel->currentThread->get_RemainingBurstTime() > kernel->scheduler->TotalFront()->get_Remainin
            kernel->interrupt->YieldOnReturn();
        }

        else if (kernel->currentThread->get_Priority() >= 50 && kernel->currentThread->get_Priority() < 1
        kernel->scheduler->TotalFront()->get_Priority() >= 100) {
            kernel->interrupt->YieldOnReturn();
        }

        else if (kernel->currentThread->get_Priority() >= 0 && kernel->currentThread->get_Priority() <= 4
        (kernel->currentThread->get_RRTime() >= 200 || kernel->scheduler->TotalFront()->get_Priority() >=
            kernel->interrupt->YieldOnReturn();
        }

        kernel->scheduler->UpdatePriority();

    }
    //<TODO>
```

CallBack()是每 100 ticks 就會執行一次，所以當程式每次進入這個 function 時都要把 currentThread 的 RunTime 和 RRTime 都加 100，以此來計算 currentThread 在 CPU 執行了多久。接著就是去檢查各個 ready queue 有沒有存在需要與 currentThread 進行 context switch 的 thread，需要進行 context switch 的條件有以下這幾個。

- 在較高 priority 的 ready queue 中存在 thread (Level 1 > Level 2 > Level 3)
- 在 Level 1 ready queue 時則是比較 thread 的 RemainingBurstTime，越短的 thread 有越高的順位被 CPU 執行
- 在 Level 3 ready queue 時則是採取 RoundRobin，所以只要 Level 3 的 thread 執行滿 200 ticks 就要進行 context switch，換其他在 Level 3 ready queue 等待的 thread 執行。

這裡 implement 的方式是只要符合特定的條件，就會去呼叫 interrupt->YeildOneReturn()，去讓 yeildOneReturn 這個 flag 改為 true，讓 currentThread 可以由 Running state 變為 Ready state。

最後面呼叫 UpdatePriority() 對各個 level 的 ready queue 裡 thread 的 priority 進行 aging。

- Scheduler::TotalFront ()

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

    if (! L1ReadyQueue->IsEmpty()) {
        thread = L1ReadyQueue->Front();
        return thread;
    } else if (! L2ReadyQueue->IsEmpty()) {
        thread = L2ReadyQueue->Front();
        return thread;
    } else if (! L3ReadyQueue->IsEmpty()) {
    thread = L3ReadyQueue->Front();
        return thread;
    } else {
        return NULL;
    }

    //TODO>
}
```

TotalFront()就會從 ready queue 裡面選出優先度最高的 thread 進行回傳，優先度排列是 L1 > L2 > L3，所以較高優先度的 ready queue 有 thread 時，就會優先回傳。

- Scheduler::UpdatePriority()

```
void Scheduler::UpdatePriority()      You, seconds ago • Uncommitted changes
{
    //ListIterator<Thread*> *iter2(L2ReadyQueue);
    // cout << "UpdatePriority\n";
    Thread* ReadyThread;
    ListIterator<Thread *> *iter1 = new ListIterator<Thread *>(L1ReadyQueue);
    ListIterator<Thread *> *iter2 = new ListIterator<Thread *>(L2ReadyQueue);
    Statistics* stats = kernel->stats;

    for (; !iter1->IsDone(); iter1->Next()) {
        ReadyThread = iter1->Item();
        ReadyThread->set_WaitTime(ReadyThread->get_WaitTime() + 100);
        if (ReadyThread->get_WaitTime() > 400) {
            if (ReadyThread->get_Priority() < 140) {

                DEBUG(dbgMLFQ, "[UpdatePriority] Tick ["<< stats->totalTicks << "]: Thread [" << ReadyThread
                ReadyThread->set_Priority(ReadyThread->get_Priority() + 10);
                L1ReadyQueue->Remove(ReadyThread);
                DEBUG(dbgMLFQ, "[RemoveFromQueue] Tick ["<< stats->totalTicks << "]: Thread [" << ReadyThrea
                L1ReadyQueue->Insert(ReadyThread);
                DEBUG(dbgMLFQ, "[InsertToQueue] Tick ["<< stats->totalTicks << "]: Thread [" << ReadyThread-
            }
        }
    }
```

這個部分就是依序去檢查各個 level 的 ready queue 裡 thread 的 WaitTime 有沒有超過 400 ticks，如果有就需要把 priority 提高 10，並且重新 sorting 讓 ready queue 裡面是依照特定順序排列。做法就是使用 SortedList 的 Insert 去達成，因為我們有去定義不同 level ready queue 所需要用到的 compare，而 insert 就會使用這個 compare 去讓 ready queue 能夠按照特定順序去排列。

2. instruction execution

(1). 這個作業是將一個要被執行的檔案(ex: hw2_test1)當作一個 process 去讓 NachOS 執行，而每個 process 並不會再分成多個 thread，因此可以把一個要被執行的檔案直接當作一個 thread。

```
userprog/nachos -epb <execute file> <p1> <bt1> -epb <execute file> <p2> <bt2> -d z
```

execute file： 需要 NachOS 去執行的 file

p1, p2： 對應 thread 的初始 priority

bt1, bt2： 對應 thread 的預測 CPU burst time

-d z： 選擇所需要顯示的 DEBUG message

ex:

```
userprog/nachos -epb test/hw2_test1 40 5000 -epb test/hw2_test2 80 4000 -d z
```

(2). debug.h:

```
// The pre-defined debugging flags are:

const char dbgAll = '+';        // turn on all debug messages
const char dbgThread = 't';     // threads
const char dbgSynch = 's';      // locks, semaphores, condition vars
const char dbgInt = 'i';        // interrupt emulation
const char dbgMach = 'm';       // machine emulation (USER_PROGRAM)
const char dbgDisk = 'd';       // disk emulation (FILESYS)
const char dbgFile = 'f';       // file system (FILESYS)
const char dbgAddr = 'a';       // address spaces (USER_PROGRAM)
const char dbgNet = 'n';        // network emulation (NETWORK)


//<REPORT>
const char dbgMLFQ = 'z';
//<REPORT>
```

定義 debug flag，讓我們能夠去選擇在執行 NachOS 時要顯示那些 debug message。

```
//----------------------------------------------------------------------
// DEBUG
//      If flag is enabled, print a message.
//----------------------------------------------------------------------
#define DEBUG(flag,expr)
    if (!debug->IsEnabled(flag)) {} else {              \
        cerr << expr << "\n";                           \
    }
```

定義 DEBUG 這個 macro function，我們會使用這個 macro function 在 NachOS code 裡面適當位置印出助教所要求的 message。這個作業所使用的 debug flag 是 dbgMLFQ，所以使用的方式就會是下圖這樣子。

```
Statistics* stats = kernel->stats;
nextThread = kernel->scheduler->FindNextToRun();
if (nextThread != NULL) {
    DEBUG(dbgMLFQ, "[ContexSwitch] Tick ["<< stats->totalTicks << "]: Thread [" << nextThread->getID() <<"] is now s
    kernel->currentThread->set_RRTime(0);
    kernel->scheduler->ReadyToRun(this);
    nextThread->set_WaitTime(0);
    kernel->scheduler->Run(nextThread, FALSE);
}
```

3. 小組分工

106061132 黃友廷：Coding、Report

106061146 陳兆廷：Report

106000147 沈永聖（已退選）：Coding、Report