CS3423 OS Project: SJF Scheduling on NachOS

107061218 謝霖泳

107061114 吳騏佑

Part 1. Trace Code

本大題僅針對一開始解壓縮完的 code 討論，<u>不包含</u>我們implement的部分。 為方便閱讀，我將對應的code一起放上來，考量版面，code中重複、雷同或非重點部分將以...帶過。

1. New→Ready

   (1) userprog/userkernel.cc UserProgKernel::`InitializeAllThreads()`

```
void UserProgKernel::InitializeAllThreads() {
    for (int i = 1; i <= execfileNum; i++) {
        int a = InitializeOneThread(execfile[i]);
    }
    currentThread->Finish();
}
```

> 使用for迴圈讀取整個execfile陣列，對每個execfile做處理。在每次迴圈中，`InitializeOneThread()`會幫該execfile建立一個thread。當所有的execfile都被initialize後，會結束掉當前的thread。

   (2) userprog/userkernel.cc UserProgKernel:: `InitializeOneThread(char*)`

```
int UserProgKernel::InitializeOneThread(char* name) {
    threadNum++;
    return threadNum - 1;
}
```

> kernel會幫execfile建立一個新的thread，因為initialize了一個新的thread，所以`threadNum++`，並return `threadNum` increment前的值。

(3) threads/thread.cc Thread::`Fork(VoidFunctionPtr, void*)`

```
void Thread::Fork(VoidFunctionPtr func, void *arg) {
    Interrupt *interrupt = kernel->interrupt;
    Scheduler *scheduler = kernel->scheduler;
    IntStatus oldLevel;
    ...
    StackAllocate(func, arg);
    oldLevel = interrupt->SetLevel(IntOff);
    scheduler->ReadyToRun(this);
    (void) interrupt->SetLevel(oldLevel);
}
```

> 先建立thread的`interrupt`和`scheduler`兩個指標，接著透過`StackAllocate()`為thread安排stack上的空間，最後呼叫`ReadyToRun()`。

(4) threads/thread.cc Thread::`StackAllocate(VoidFunctionPtr, void*)`

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

#ifdef PARISC
    stackTop = stack + 16;
    stack[StackSize - 1] = STACK_FENCEPOST;
#endif
...
}
```

> 設定stack的位置，安排一塊記憶體空間，並依照不同的系統架構去計算`stackTop`等參數。還有值得注意到的一點是，每個thread的底部都放著一個`STACK_FENCEPOST`的token, fence即籬笆，用來控管這個stack的邊界，以偵測是否有stack overflow產生。

(5) threads/scheduler.cc Scheduler::`ReadyToRun(Thread*)`

```
void Scheduler::ReadyToRun (Thread *thread) {
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    DEBUG(dbgThread, "Putting thread on ready list: " << thread-
>getName());
    thread->setStatus(READY);
    readyList->Append(thread);
}
```

> 將thread的status改為`READY`，並令該thread進入ready queue。

2. Ready→Running (1) threads/scheduler.cc Scheduler::`FindNextToRun()`

```
Thread *Scheduler::FindNextToRun() {
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    if (readyList->IsEmpty())
        return NULL;
    else
        return readyList->RemoveFront();
}
```

> 從ready queue中尋找下一個要執行的thread，並將其從中移除，回傳找到的thread，找不到則回傳`NULL`。

(2) threads/scheduler.cc Scheduler::`Run(Thread*, bool)`

```
void Scheduler::Run (Thread *nextThread, bool finishing) {
    Thread *oldThread = kernel->currentThread;
    ...
    if (finishing) {
        ASSERT(toBeDestroyed == NULL);
        toBeDestroyed = oldThread;
    }
#ifdef USER_PROGRAM
    if (oldThread->space != NULL) {
        oldThread->SaveUserState();
        oldThread->space->SaveState();
    }
#endif
    oldThread->CheckOverflow();
    kernel->currentThread = nextThread;
    nextThread->setStatus(RUNNING);
    ...
    SWITCH(oldThread, nextThread);
    ...
    CheckToBeDestroyed();
#ifdef USER_PROGRAM
    if (oldThread->space != NULL) {
        oldThread->RestoreUserState();
        oldThread->space->RestoreState();
    }
#endif
}
```

> 將`currentThread`設為`oldThread`，呼叫`SaveState()`將`oldThread`的state存下來，並檢查`finishing` flag，如果`finishing == 1`，代表已經完成，就會在之後被destroy。之後將`nextThread`設為`currentThread`，並將其狀態設定為`RUNNING`，呼叫`SWITCH()`對這兩個thread進行context switch。

(3) threads/switch.s `SWITCH(Thread*, Thread*)`

```
SWITCH:
    movl    %eax,_eax_save
    movl    4(%esp),%eax
    movl    %ebx,_EBX(%eax)
    ...
    movl    8(%esp),%eax

    movl    _EAX(%eax),%ebx
    movl    %ebx,_eax_save
    movl    _EBX(%eax),%ebx
    ...
    ret
```

> 這段assembly code的目的就是達成context switch的效果。將原本old thread `t1`的資訊存到一些gereral purpose register中，並將pointer移到`t2`的位置，去讀取原先存在gereral purpose register中有關`t2`的資訊。

(4) machine/mipssim.cc Machine::`Run()`

```
void Machine::Run() {
    Instruction *instr = new Instruction;
    ...
    kernel->interrupt->setStatus(UserMode);
    for (;;) {
        OneInstruction(instr);
        kernel->interrupt->OneTick();
        if (singleStep && (runUntilTime <= kernel->stats->totalTicks))
            Debugger();
    }
}
```

> 在context switch後，用迴圈`for(;;)`將目前thread `executeFile`中的instruction一行一行執行。

3. Running→Ready

(1) machine/mipssim.cc Machine::`Run()`

同上。

(2) machine/interrupt.cc Interrupt::`OneTick()`

```
void Interrupt::OneTick() {
    MachineStatus oldStatus = status;
    Statistics *stats = kernel->stats;
    if (status == SystemMode) {
        stats->totalTicks += SystemTick;
        stats->systemTicks += SystemTick;
    } else { ... }
    ...
    CheckIfDue(FALSE);
    ...
    if (yieldOnReturn) {
        ...
        kernel->currentThread->Yield();
        ...
    }
}
```

> 用來模擬時間，有點類似Verilog裡面的clk訊號。根據現在的mode(即user mode或kernel mode)去更新`UserTick`或`SystemTick`。先將interrupt關掉，避免在檢查途中被其他interrupt打斷，用`CheckIfDue()`檢查是否有該處理的interrupt，檢查完後重新把interrupt打開，並檢查是否該執行`Yield()`。

(3) threads/thread.cc Thread::`Yield()`

```
void Thread::Yield () {
    Thread *nextThread;
    IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);
    ...
    nextThread = kernel->scheduler->FindNextToRun();
    if (nextThread != NULL) {
        kernel->scheduler->ReadyToRun(this);
        kernel->scheduler->Run(nextThread, FALSE);
    }
    (void)kernel->interrupt->SetLevel(oldLevel);
}
```

> `Yield()`用來釋放CPU的資源，將目前執行中的thread停下來，放到ready queue中，並呼叫`FindNextToRun()`找到下一個可執行的thread，使其開始執行。

(4) threads/scheduler.cc Scheduler::`FindNextToRun()`

同上。

(5) threads/scheduler.cc Scheduler::`ReadyToRun(Thread*)`

同上。

(6) threads/scheduler.cc Scheduler::`Run(Thread*, bool)`

同上。

4. Running→Waiting

(1) userprog/exception.cc ExceptionHandler(ExceptionType) `case SC_PrintInt`

```
case SC_PrintInt:
    val=kernel->machine->ReadRegister(4);
    kernel->synchConsoleOut->PutInt(val);
    ...
    return;
```

> 在執行`executeFile`內的instruction時，若出現exception, `ExceptionHandler`會先去判斷是哪一種exception。在這個case中，會從register 4中讀取integer，並呼叫`PutInt()`system call來把它印出來。

(2) userprog/synchconsole.cc SynchConsoleOutput::`PutInt()`

```
void SynchConsoleOutput::PutInt(int value) {
    char str[10];
    int index = 0;
    sprintf(str, "%d\0", value);
    lock->Acquire();
    do{
        consoleOutput->PutChar(str[index]);
        index++;
        waitFor->P();
    } while (str[index] != '\0');
    lock->Release();
}
```

> 因為output只有支援`PutChar()`，所以想印出`value`，就要將`value`裡面每個digit分開來印。而我們知道一個32 bit的integer最多可以有32*log(2) = 10位，故宣告一個長度為10的char array `str`。接著，因為同一個int各個bit需要依序輸出，中間不能斷掉，因此要acquire `lock`，來確保`PutInt`這件事是atomic的。

(3) machine/console.cc ConsoleOutput::`PutChar(char)`

```
void SynchConsoleOutput::PutChar(char ch) {
    lock->Acquire();
    consoleOutput->PutChar(ch);
    waitFor->P();
    lock->Release();
}
```

> 觸發一個interrupt，用以在output印出一個字元。

(4) threads/synch.cc Semaphore::`P()`

```
void Semaphore::P() {
    Interrupt *interrupt = kernel->interrupt;
    Thread *currentThread = kernel->currentThread;
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    while (value == 0) {
        queue->Append(currentThread);
        currentThread->Sleep(FALSE);
    }
    value--;
    (void) interrupt->SetLevel(oldLevel);
}
```

> 此處和第六章講義中描述的semaphore概念基本上差不多，用來確保特定數量的resource正在被使用。如果這時候semaphore的`value == 0`，代表沒有資源可以使用。這時候就要把這個thread放到semaphore的queue中，並叫他去睡覺，卡在這個while loop中；反之，如果此時有資源可以用，那就`value--`，讓他去使用資源。

(5) threads/synchlist.cc SynchList<T>::`Append(T)`

```
void SynchList<T>::Append(T item) {
    lock->Acquire();
    list->Append(item);
    listEmpty->Signal(lock);
    lock->Release();
}
```

> 當thread要append到list中時，必須先acquire lock，以確保mutual exclusion。append完成後就用`Signal()`通知下一個wait的thread，然後將lock release掉。

(6) threads/thread.cc Thread::`Sleep(bool)`

```
void Thread::Sleep (bool finishing) {
    Thread *nextThread;
    ...
    status = BLOCKED;
    while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL)
        kernel->interrupt->Idle();
    kernel->scheduler->Run(nextThread, finishing);
}
```

> 會觸發`Sleep()`的原因是因為thread在等待IO，因此需要將thread的`status`設為`BLOCKED`，並呼叫`FindNextToRun()`找下一個thread來執行。如果`FindNextToRun()`回傳值為`NULL`，代表ready queue已經空掉了，這時就將kernel的狀態設為idle。

(7) threads/scheduler.cc Scheduler::`FindNextToRun()`

同上。

(8) threads/scheduler.cc Scheduler::`Run(Thread*, bool)`

同上。

5. Waiting→Ready

(1) threads/synch.cc Semaphore::`V()`

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

    if (!queue->IsEmpty())
        kernel->scheduler->ReadyToRun(queue->RemoveFront());

    value++;
    (void) interrupt->SetLevel(oldLevel);
}
```

> 和`Semaphore::P()`類似，當這個thread使用完資源後，就要將`value++`讓其他正在等待的thread可以跳出`Semaphore::P()`的while loop。

(2) threads/scheduler.cc Scheduler::`ReadyToRun(Thread*)`

同上。

6. Running→Terminated

(1) userprog/exception.cc ExceptionHandler(ExceptionType) `case SC_Exit`

```
case SC_Exit:
    ...
    val=kernel->machine->ReadRegister(4);
    cout << "return value:" << val << endl;
    kernel->currentThread->Finish();
    break;
```

> `SC_Exit`這個case會呼叫`Finish()`，作為一個thread的結束，並將register 4的值印出。

(2) threads/thread.cc Thread::`Finish()`

```
void Thread::Finish () {
    (void) kernel->interrupt->SetLevel(IntOff);
    ...
    Sleep(TRUE);
}
```

> 呼叫`Sleep()`，將`finishing`設為`TRUE`傳入。

(3) threads/thread.cc Thread::`Sleep(bool)`

同上。

(4) threads/scheduler.cc Scheduler::`FindNextToRun()`

同上。

(5) threads/scheduler.cc Scheduler::`Run(Thread*, bool)`

同上。

Part 2. Implementation

我將需要我們implement的地方列出來如下：

- userprog/
  - `userkernel.cc`: 2 TODO
- lib/
  - `debug.h`: 1 TODO
- threads/
  - `scheduler.h`: 1 TODO
  - `scheduler.cc`: 7 TODO
  - `thread.h`: 1 TODO
  - `thread.cc`: 2 TODO
  - `alarm.cc`: 1 TODO

接著開始逐一說明如何implement的。

- `userkernel.cc`: TODO 1

```
//<TODO>
void ForkExecute(Thread *t) {
    DEBUG(dbgSJF, "ForkExecute => fork thread id: " << t->getID() << ",
currentTick: " << kernel->stats->totalTicks);

    t->space->Load(t->getName());
    t->space->Execute(t->getName());
}
//<TODO>
```

> 因為當每個thread第一次進到runnung state時，都要先load到memory中才能被執行，而原始的code中只有execute的部分，因此運用`addrspace.h`中的`Load()` function，新增`t->space->Load(t->getName());`一行，以便將thread load進memory，也才能讓terminal顯示得跟助教一樣，有`[AddrSpace:: Load over]`字樣。

- `userkernel.cc`: TODO 2

```
int UserProgKernel::InitializeOneThread(char* name) {
    //<TODO>
    t[threadNum] = new Thread(name, threadNum);
    t[threadNum]->space = new AddrSpace();
    t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);
    // <TODO>
    threadNum++;
    return threadNum - 1;
}
```

> 當每個`execfile`要被執行時，kernel要先幫他創立一個thread負責執行，因此加入`t[threadNum] = new Thread(name, threadNum);`來`new`一個thread給他，並用`t[threadNum]->space = new AddrSpace();`來分配space給這個thread，最後呼叫`Fork()`做stack allocation。

- `debug.h`: TODO 1

```
//<TODO>
const char dbgSJF = 'j';
//<TODO>
```

> 加上一行`dbgSJF`的定義，並將他assign為`'j'`。

- `scheduler.h`: TODO 1

```
private:
    SchedulerType schedulerType;
    //<TODO>
    SortedList<Thread* > *readyQueue;
    //<TODO>
    Thread *toBeDestroyed;
```

> 定義`readyQueue`，型態為`SortedList`，因為這樣才能在我insert thread進來的時候自動依照我定義的rule排序，以達到shortest job first的目的。

- `scheduler.cc`: TODO 1

```
//<TODO>
static int SortingRule(Thread *t1, Thread *t2);
//<TODO>
```

> 定義`SortedList`的sorting rule，其型別為`static int`, input為兩個`Thread*`。

- `scheduler.cc`: TODO 2

```
//<TODO>
Scheduler::Scheduler() {
    readyQueue = new SortedList<Thread *>(SortingRule);
    toBeDestroyed = NULL;
}
//<TODO>
```

> 宣告`readyQueue`變數，其形態為`SortedList`，這樣等等才會自動幫我們依照`SortingRule`這個compare function進行排序。

- `scheduler.cc`: TODO 3

```
//<TODO>
Scheduler::~Scheduler() {
    delete readyQueue;
}
//<TODO>
```

> 在`Scheduler`的destructor中，用`delete()`指令將`readyQueue`刪掉。

- `scheduler.cc`: TODO 4

```
//<TODO>
void Scheduler::ReadyToRun (Thread *thread) {
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
    if (!readyQueue->IsEmpty()) {
        DEBUG(dbgSJF, "***Thread [" << readyQueue->Front()->getID()
              << "]'s and thread [" << thread->getID() << "]'s burst time are "
              << readyQueue->Front()->getPredictedBurstTime() << " and "
              <<  thread->getPredictedBurstTime()<< "***");
    }

    thread->setStatus(READY);
    DEBUG(dbgSJF, "[I] Tick [" << kernel->stats->totalTicks << "]: Thread ["
          << thread->getID() << "] is inserted into the readyQueue");
    readyQueue->Insert(thread);

    if (thread->getPredictedBurstTime()
        < kernel->currentThread->getPredictedBurstTime()) {
        kernel->interrupt->YieldOnReturn();
    }
}
//<TODO>
```

> 在前面新增一個`if`，來印出debugging message，如果`readyQueue`不是空的，那就印出傳進來的thread的burst time以及`readyQueue`中第一個thread的burst time。(可以直接取`readyQueue`的第一個是因為`readyQueue`是`SortedList`，本來就已經按照burst time排好了)

> 再來就是將`thread`給insert到`readyQueue`中，用的方法是呼叫`Insert()`，而不是原本code寫的`Append()`，因為`Append()`只會將這個`thread`插到最後面，不會進行排序，所以要呼叫`Insert()`才能依照我給的sorting rule去insert到正確的位置。此外，在`thread`被insert到`readyQueue`之前，先依照要求印出另一個debugging message，顯示誰要被insert到`readyQueue`。

> 最後幾行是preemptive SJF的精華。在這個function的末端加入一個`if`條件，判斷是否要preempt。如果`currentThread`的burst time比傳進來這個`thread`的burst time還要長的話，就要preempt，而此處我實作preemption的方法是call `interrupt->YieldOnReturn()`。

- `scheduler.cc`: TODO 5

```
//<TODO>
Thread *Scheduler::FindNextToRun () {
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    if (readyQueue->IsEmpty()) {
        return NULL;
    }
    else {
        Thread *temp = readyQueue->RemoveFront();
        DEBUG(dbgSJF, "[R] Tick [" << kernel->stats->totalTicks
                       << "]: Thread [" << temp->getID()
                       << "] is removed from the readyQueue");
        return temp;
    }
}
//<TODO>
```

> 宣告一個thread pointer變數`*temp`，指到`readyQueue`的頭並把它從`readyQueue`中remove掉，它就是`FindNextToRun()`想要找的thread，也就是`readyQueue`中下一個要run的thread，找到之後印出相應的debugging message，並把它return回來。

- `scheduler.cc`: TODO 6

```
//<TODO>
void Scheduler::Print() {
    cout << "Ready list contents:\n";
    readyQueue->Apply(ThreadPrint);
}
//<TODO>
```

> 將原本code中的`readyList`改為`readyQueue`。

- `scheduler.cc`: TODO 7

```
//<TODO>
static int SortingRule(Thread *t1, Thread *t2) {
    int t1BurstTime = t1->getRunTime();
    int t2BurstTime = t2->getRunTime();
    int t1id = t1->getID();
    int t2id = t2->getID();

    if (t1BurstTime == t2BurstTime) {
        if (t1id == t2id)
            return 0;
        else if (t1id > t2id)
            return -1;
        else
            return 1;
    }
    else if (t1BurstTime > t2BurstTime)
        return -1;
    else
        return 1;
}
// <TODO>
```

> 寫一個sorting rule給`SortedList`進行排序。因為本次project在實現sjf的scheduling algorithm，所以裡所當然地先依照兩個thread的burst time進行排序，如果兩個thread的burst time相同，按照作業的描述應該優先執行id較大者，故比完burst time接著再比id。

- `thread.h`: TODO 1

```
int getID() {
    return ID;
}
void setRunTime(int t) {
    RunTime = t;
}
int getRunTime() {
    return RunTime;
}
void setPredictedBurstTime (int t) {
    PredictedBurstTime = t;
}
int getPredictedBurstTime() {
    return PredictedBurstTime;
}
```

> 在`Class Thread`中define `ID`的getter function、`RunTime`的getter/setter function及`PredictedBurstTime`的getter/setter function。

- `thread.cc`: TODO 1

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

    ASSERT(this == kernel->currentThread);
    DEBUG(dbgThread, "Yielding thread: " << name);

    nextThread = kernel->scheduler->FindNextToRun();

    DEBUG(dbgSJF, "[YS] Tick [" << kernel->stats->totalTicks << "] : Thread [" << nextThread->getID() << "] is now selected for execution, thread [" << kernel->currentThread->getID() << "] is replaced, and it has executed [" << kernel->currentThread->getRunTime() << "] ticks");

    this->setStatus(READY);

    if (nextThread != NULL) {

        kernel->scheduler->ReadyToRun(this);
        kernel->scheduler->Run(nextThread, FALSE);
    }
    (void)kernel->interrupt->SetLevel(oldLevel);
}
//<TODO>
```

> 這裡要印出第五個debugging message，因為這個debugging message要依照有沒有preemption發生印出`<YS>`或`<S>`，而我的設計是有preemption發生就會觸發`Yield()`，沒有就會觸發`Sleep()`，所以在這裡要印出`<YS>`加上第五個debugging message的內容。最後，將`currentThread`的`status`設為`READY`。

- `thread.cc`: TODO 2

```
//<TODO>
void Thread::Sleep (bool finishing) {
    Thread *nextThread;
    ASSERT(this == kernel->currentThread);
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    DEBUG(dbgThread, "Sleeping thread: " << name);

    status = BLOCKED;

    int PrevBurstTime = kernel->currentThread->getPredictedBurstTime();
    int ApproximateBurstTime = 0.5 * kernel->currentThread->getRunTime()
                             + 0.5 * kernel->currentThread->getPredictedBurstTime();

    kernel->currentThread->setPredictedBurstTime(ApproximateBurstTime);
    while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL)
        kernel->interrupt->Idle();

    DEBUG(dbgSJF, "[S] Tick [" << kernel->stats->totalTicks << "] : Thread [" << nextThread->getID() << "] is now selected for execution, thread [" << kernel->currentThread->getID() << "] is replaced, and it has executed for [" << kernel->currentThread->getRunTime() << "] ticks");

    if (kernel->currentThread->getID() > 0) {
        DEBUG(dbgSJF, "[U] Tick [" << kernel->stats->totalTicks << "]: Thread [" << kernel->currentThread->getID() << "] updates approximate burst time, from [" << PrevBurstTime << "], add [" << kernel->currentThread->getPredictedBurstTime() - PrevBurstTime << "] to [" << ApproximateBurstTime << "]");
    }
    kernel->scheduler->Run(nextThread, finishing);
}
//<TODO>
```

> 先設一個變數`PrevBurstTime`把thread現在的burst time先計下來，因為等一下update approximate burst time的debugging message會用到。再利用公式$t_i = 0.5 * T + 0.5 * t_{i-1}$計算`ApproximateBurstTime`並呼叫`setPredictedBurstTime()`更新之。值得注意的是，這裡的T就是current thread的runtime，而t_{i-1}就是current thread原本的`PredictedBurstTime`。

> 當找到`nextThread`之後，中間的`while`迴圈便會跳出，這時候印出相應的debugging message。

- `alarm.cc`: TODO 1

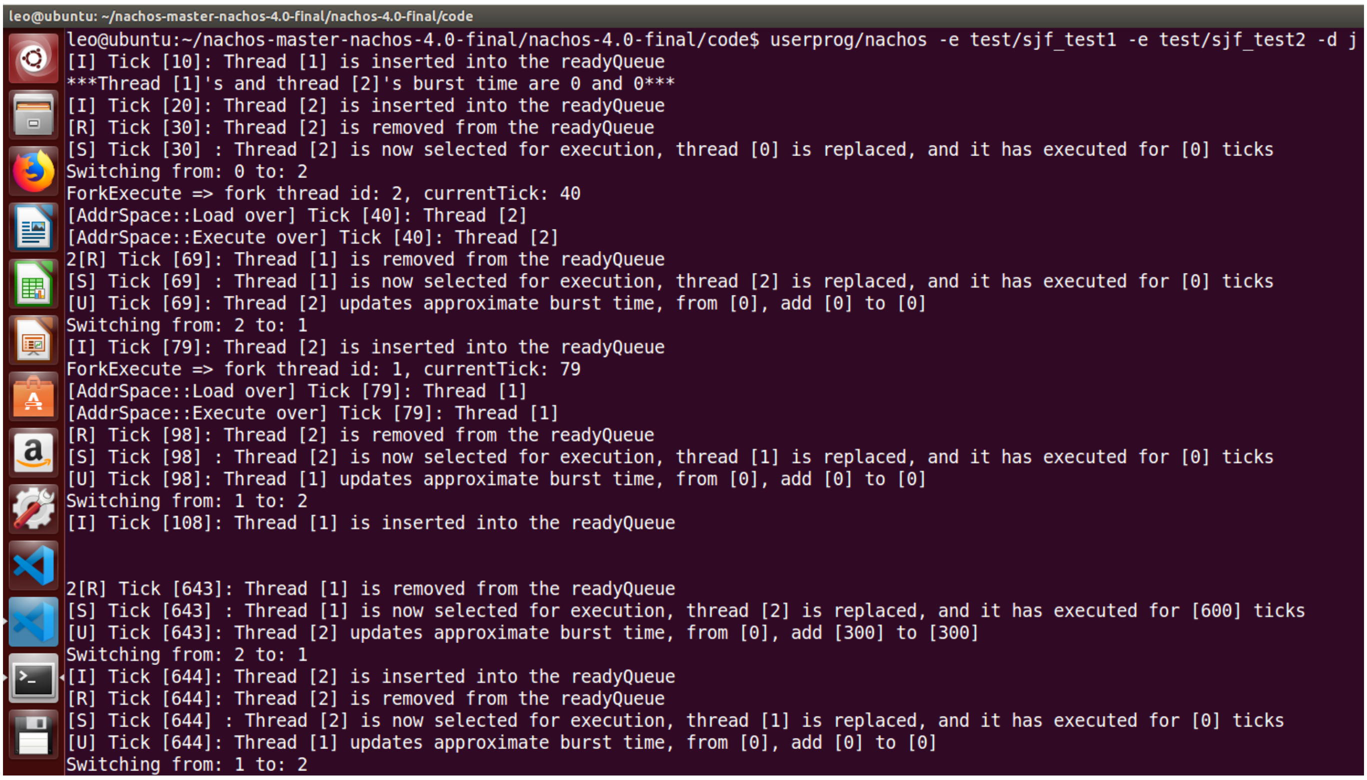
```
//<TODO>
void Alarm::CallBack() {
    Interrupt *interrupt = kernel->interrupt;
    MachineStatus status = interrupt->getStatus();

    Thread *t = kernel->currentThread;
    t->setRunTime(t->getRunTime() + 100);

    if (status == IdleMode) {
        if (!interrupt->AnyFutureInterrupts()) {
            timer->Disable();
        }
    }
    else {
        // interrupt->YieldOnReturn();
    }
}
//<TODO>
```

> 加入`Thread *t = kernel->currentThread;`及`t->setRunTime(t->getRunTime() + 100);`兩行，用來達成每100個ticks更新一次`RunTime`的效果。最後依照助教要求將`else`中的`interrupt->YieldOnReturn();`這行comment掉。

Part 3. Execution Result



```
leo@ubuntu: ~/nachos-master-nachos-4.0-final/nachos-4.0-final/code
leo@ubuntu:~/nachos-master-nachos-4.0-final/nachos-4.0-final/code$ userprog/nachos -e test/sjf_test1 -e test/sjf_test2 -d j
[I] Tick [10]: Thread [1] is inserted into the readyQueue
***Thread [1]'s and thread [2]'s burst time are 0 and 0***
[I] Tick [20]: Thread [2] is inserted into the readyQueue
[R] Tick [30]: Thread [2] is removed from the readyQueue
[S] Tick [30] : Thread [2] is now selected for execution, thread [0] is replaced, and it has executed for [0] ticks
Switching from: 0 to: 2
ForkExecute => fork thread id: 2, currentTick: 40
[AddrSpace::Load over] Tick [40]: Thread [2]
[AddrSpace::Execute over] Tick [40]: Thread [2]
2[R] Tick [69]: Thread [1] is removed from the readyQueue
[S] Tick [69] : Thread [1] is now selected for execution, thread [2] is replaced, and it has executed for [0] ticks
[U] Tick [69]: Thread [2] updates approximate burst time, from [0], add [0] to [0]
Switching from: 2 to: 1
[I] Tick [79]: Thread [2] is inserted into the readyQueue
ForkExecute => fork thread id: 1, currentTick: 79
[AddrSpace::Load over] Tick [79]: Thread [1]
[AddrSpace::Execute over] Tick [79]: Thread [1]
[R] Tick [98]: Thread [2] is removed from the readyQueue
[S] Tick [98] : Thread [2] is now selected for execution, thread [1] is replaced, and it has executed for [0] ticks
[U] Tick [98]: Thread [1] updates approximate burst time, from [0], add [0] to [0]
Switching from: 1 to: 2
[I] Tick [108]: Thread [1] is inserted into the readyQueue


2[R] Tick [643]: Thread [1] is removed from the readyQueue
[S] Tick [643] : Thread [1] is now selected for execution, thread [2] is replaced, and it has executed for [600] ticks
[U] Tick [643]: Thread [2] updates approximate burst time, from [0], add [300] to [300]
Switching from: 2 to: 1
[I] Tick [644]: Thread [2] is inserted into the readyQueue
[R] Tick [644]: Thread [2] is removed from the readyQueue
[S] Tick [644] : Thread [2] is now selected for execution, thread [1] is replaced, and it has executed for [0] ticks
[U] Tick [644]: Thread [1] updates approximate burst time, from [0], add [0] to [0]
Switching from: 1 to: 2
```

這只是output的一部分，所有內容和助教完全相同，太開心了！

Part 4. Team Member Contribution

(unit: %)	謝霖泳	吳騏佑
trace code	80	20
implementation	100	0
report	60	40