OS Project-2 Report

R11921008 羅恩至

Part 1. System Call – Sleep()

1. Motivation and Problem Analysis

此題的目標為設置 Sleep function，來讓 Thread 能進入休眠模式。而為了實現此一目標，根據投影片的說明，需要在 userprog/syscall.h 中宣告一個 Sleep() 函式，並透過 WaitUntil()及 CallBack()這兩個函式，來分別讓 Thread 進入休眠或停止休眠。

2. Implementation

首先在 userprog/syscall.h 中宣告 Sleep 函式

```
#define SC_ThreadYield  10
#define SC_PrintInt 11
#define SC_Sleep    12
```

```
void PrintInt(int number);  //my System
void Sleep(int number);
#endif /* IN_ASM */
```

並在 test/start.s 中加上 Sleep 函式所對應的組合語言

```
    .globl  PrintInt
    .ent    PrintInt
PrintInt:
    addiu   $2,$0,SC_PrintInt
    syscall
    j       $31
    .end    PrintInt

    .globl  Sleep
    .ent    Sleep
Sleep:
    addiu   $2,$0,SC_Sleep
    syscall
    j       $31
    .end    Sleep

/* dummy function to keep gcc happy */
        .globl  __main
```

為了完成 WaitUntil()及 CallBack()，讓 Thread 能順利進入或停止休眠，因此需要修改 threads 資料夾中的 alarm.h 以及 alarm.cc 的程式碼。

threads/alarm.h

```
#include "copyright.h"
#include "utility.h"
#include "callback.h"
#include "timer.h"
#include <list>
#include "thread.h"

class Rest {
    public:
        Rest():count_current(0) {};
        void PutToBed(Thread *t, int x);
    bool PutToReady();
    bool IsEmpty();
    private:
        class Sleeping {
            public:
                Sleeping(Thread* t, int x):
                    sleeper(t), when(x) {};
                Thread* sleeper;
                int when;
        };

    int count_current;
    std::list<Sleeping> threadList;
};
```

```
// The following class defines a software alarm clock.
class Alarm : public CallBackObj {
  public:
    Alarm(bool doRandomYield);  // Initialize the timer, and callback
                // to "toCall" every time slice.
    ~Alarm() { delete timer; }

    void WaitUntil(int x);  // suspend execution until time > now + x

  private:
    Timer *timer;       // the hardware timer device
    Rest restList;
    void CallBack();        // called when the hardware
                // timer generates an interrupt
};
```

threads/alarm.cc

```
void
Alarm::CallBack()
{
    Interrupt *interrupt = kernel->interrupt;
    MachineStatus status = interrupt->getStatus();
    bool woken = restList.PutToReady();

    if (status == IdleMode && !woken && restList.IsEmpty()) {   // is it time to quit?
        if (!interrupt->AnyFutureInterrupts()) {
        timer->Disable();   // turn off the timer
    }
    } else {            // there's someone to preempt
    interrupt->YieldOnReturn();
    }
}

void Alarm::WaitUntil(int x) {
    IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);
    Thread* t = kernel->currentThread;
    cout << "Alarm::WaitUntil go to sleep" << endl;
    restList.PutToBed(t, x);
    kernel->interrupt->SetLevel(oldLevel);
}
```

```
bool Rest::IsEmpty() {
    return threadList.size() == 0;
}

void Rest::PutToBed(Thread*t, int x) {
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    threadList.push_back(Sleeping(t, count_current + x));
    t->Sleep(false);
}

bool Rest::PutToReady() {
    bool woken = false;
    count_current++;
    for(std::list<Sleeping>::iterator it = threadList.begin();
        it != threadList.end(); ) {
        if(count_current >= it->when) {
            woken = true;
            cout << "Bedroom::PutToReady Thread woken" << endl;
            kernel->scheduler->ReadyToRun(it->sleeper);
            it = threadList.erase(it);
        } else {
            it++;
        }
    }
    return woken;
}
```

根據上方幾張圖所示，在 threads/alarm.h 中多宣告了幾個變數，並宣告一個名為 Rest 的 Class 來執行，其中 count_current 用來當作計數器(每毫秒會計一次)，在 alarm.cc 中則是多撰寫了幾個相關的函式。這樣當有程式去呼叫 Sleep()時，也會去呼叫 WaitUntil()，以將程式丟入 Rest 中進行休眠；而同樣地，每隔一段時間 kernel 也會去呼叫一次 CallBack()，當 CallBack()被呼叫時，就會到 Rest 中確認哪個 thread 需要被喚醒，並把需要被喚醒地 thread 從 restList 裡移除，並放到 ReadyToRun()裡面。如此便完成 Sleep()的操作。

之後設定兩個測資 sleep 和 sleep2 確認是否能正確執行。根據以下兩測資，執行時右圖的測資(sleep2)會先執行十次左圖的測資(sleep)才會執行一次。

```
#include "syscall.h"
main() {
    int i;
    for(i = 0; i < 5; i++) {
    Sleep(1000000);
    PrintInt(2222);
    }
    return 0;
}
```

```
#include "syscall.h"
main() {
    int i;
    for(i = 0; i < 20; i++) {
    Sleep(100000);
    PrintInt(10);
    }
    return 0;
}
```

3. Result

根據執行結果，可看到 sleep2 先執行了十次，才第一次執行了 sleep，代表有正確的執行出來。

```
os@OS:~/nachos-4.0/code/userprog$ ./nachos -e ../test/sleep -e ../test/sleep2
Total threads number is 2
Thread ../test/sleep is executing.
Thread ../test/sleep2 is executing.
Sleeping time:1000000 ms
Alarm::WaitUntil go sleep
Sleeping time:100000 ms
Alarm::WaitUntil go sleep
Rest::Ready Thread Woken!
Print integer:10
Sleeping time:100000 ms
Alarm::WaitUntil go sleep
Rest::Ready Thread Woken!
Print integer:10
Sleeping time:100000 ms
Alarm::WaitUntil go sleep
Rest::Ready Thread Woken!
Print integer:10
Sleeping time:100000 ms
Alarm::WaitUntil go sleep
Rest::Ready Thread Woken!
Print integer:10
Sleeping time:100000 ms
```

```
Print integer:10
Sleeping time:100000 ms
Alarm::WaitUntil go sleep
Rest::Ready Thread Woken!
Print integer:10
Sleeping time:100000 ms
Alarm::WaitUntil go sleep
Rest::Ready Thread Woken!
Print integer:10
Sleeping time:100000 ms
Alarm::WaitUntil go sleep
Rest::Ready Thread Woken!
Print integer:10
Sleeping time:100000 ms
Alarm::WaitUntil go sleep
Rest::Ready Thread Woken!
Rest::Ready Thread Woken!
Print integer:10
Sleeping time:100000 ms
Alarm::WaitUntil go sleep
Print integer:2222
Sleeping time:1000000 ms
Alarm::WaitUntil go sleep
Rest::Ready Thread Woken!
```

```
Print integer:10
Sleeping time:100000 ms
Alarm::WaitUntil go sleep
Rest::Ready Thread Woken!
Print integer:10
Sleeping time:100000 ms
Alarm::WaitUntil go sleep
Rest::Ready Thread Woken!
Print integer:10
Sleeping time:100000 ms
Alarm::WaitUntil go sleep
Rest::Ready Thread Woken!
Print integer:10
Sleeping time:100000 ms
Alarm::WaitUntil go sleep
Rest::Ready Thread Woken!
Print integer:10
Sleeping time:100000 ms
Alarm::WaitUntil go sleep
Rest::Ready Thread Woken!
Print integer:10
Sleeping time:100000 ms
Alarm::WaitUntil go sleep
Rest::Ready Thread Woken!
```

```
Print integer:10
Sleeping time:100000 ms
Alarm::WaitUntil go sleep
Rest::Ready Thread Woken!
Print integer:10
Sleeping time:100000 ms
Alarm::WaitUntil go sleep
Rest::Ready Thread Woken!
Print integer:10
Sleeping time:100000 ms
Alarm::WaitUntil go sleep
Rest::Ready Thread Woken!
Rest::Ready Thread Woken!
Print integer:10
return value:0
Print integer:2222
Sleeping time:1000000 ms
Alarm::WaitUntil go sleep
Rest::Ready Thread Woken!
Print integer:2222
Sleeping time:1000000 ms
Alarm::WaitUntil go sleep
Rest::Ready Thread Woken!
Print integer:2222
```

```
Sleeping time:1000000 ms
Alarm::WaitUntil go sleep
Rest::Ready Thread Woken!
Print integer:2222
return value:0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 500000100, idle 499998941, system 530, user 629
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
os@OS:~/nachos-4.0/code/userprog$
```

討論：

(1) 發現在前幾行輸出中，在 sleep2 執行第一次 woken 前，程式 print 了兩次 WaitUntil go sleep(紅框部分)，這是因為一開始在執行時，先讓 sleep 和 sleep2 都進入休眠後，再緊接著喚醒 sleep2 所導致的結果。

(2) 在輸出結果的第四張圖片中，程式連續 print 了兩次 Ready Thread Woken(紅框部分)，這是因為 sleep2 已經喚醒了十次之後，sleep 也跟著被喚醒而出現的結果。

Part 2. CPU Scheduling

1. Motivation and Problem Analysis

這次的目標是要實作出不同種的 Scheduling 方式。在課程中所學到的 Scheduling 方式有許多種，包括 FIFO、RR、SJF、Priority 等等。而根據 Project1 的實作經驗，發現 Nachos 所預設的 Scheduling 方式為 RR，因此後來在實作這題時，除了原本的 RR 之外，也另外再選了 SJF 及 Priority 兩種方式來實作。

2. Implementation

(1) 首先於 threads/thread.cc 檔中，在 Thread::SelfTest()中建立幾個 array 來撰寫測資，包括 name(thread 名稱)、burst(各 thread 所需執行時間)、priority(各 thread 執行的優先順序)以及 arrive(thread 的抵達時間)，來給定各 thread 的參數，並建構 setBurstTime()、getBurstTime()、setPriority()、getPriority()等函式，以進行之後的排程處理。

```
static void
SimpleThread()
{
    Thread *thread = kernel->currentThread;
    while (thread->getBurstTime() > 0) {
        thread->setBurstTime(thread->getBurstTime() - 1);
        printf("Thread %s is running, remaining time =  %d\n", kernel->currentThread->getName(),
                kernel->currentThread->getBurstTime());
        kernel->interrupt->OneTick();
    }
}
```

```
void
Thread::SelfTest()
{
    DEBUG(dbgThread, "Entering Thread::SelfTest");

    //Add
    // Test Case 1
    const int number     = 3;
    char *name[number]   = {"A", "B", "C"};
    int burst[number]    = {3, 10, 4};
    int priority[number] = {4, 5, 3};
    int arrive[number] = {3, 0, 5};

    // Test Case 2
    //const int number     = 4;
    //char *name[number]   = {"A", "B", "C","D"};
    //int burst[number]    = {3, 9, 7, 4};
    //int priority[number] = {5, 1, 3, 2};
    //int arrive[number] = {3, 0, 5, 1};

    //Thread *t = new Thread("forked thread");

    //Add
    Thread *t;
    for (int i = 0; i < number; i ++) {
        t = new Thread(name[i]);
        t->setPriority(priority[i]);
        t->setBurstTime(burst[i]);
        t->Fork((VoidFunctionPtr) SimpleThread, (void *)NULL);
    }

    //t->Fork((VoidFunctionPtr) SimpleThread, (void *) 1);
    //SimpleThread(0);
    kernel->currentThread->Yield();
}
```

(2) 在 threads/thread.h 檔中，於 Thread 的 class 的 public 中增加四個函式 setBurstTime()、getBurstTime()、setPriority()、getPriority()，並在 private 中宣告兩變數 burstTime 和 priority。

```
void Begin();   // Startup code for the thread
void Finish();      // The thread is done executing

void CheckOverflow();     // Check if thread stack has overflowed
void setStatus(ThreadStatus st) { status = st; }
char* getName() { return (name); }
void Print() { cout << name; }
void SelfTest();     // test whether thread impl is working

//Add
void setBurstTime(int t)  {burstTime = t;}
int getBurstTime()     {return burstTime;}
void setPriority(int t) {priority = t;}
int getPriority()    {return priority;}
```

```
private:
  // some of the private data for this class is listed above

  int *stack;      // Bottom of the stack
      // NULL if this is the main thread
      // (If NULL, don't deallocate stack)
  ThreadStatus status;  // ready, running or blocked
  char* name;

  //Add
  int burstTime;
  int priority;

  void StackAllocate(VoidFunctionPtr func, void *arg);
        // Allocate a stack for thread.
      // Used internally by Fork()
```

(3) 接著在 threads/kernel.cc 檔中，加上針對不同 Scheduling 的判斷式。在此設定預設排程的 type 為 RR，並根據使用者傳入的參數指令，來決定 CPU 的 Scheduling 方式為何。(當使用者輸入 RR 將執行 RR Scheduling，輸入 PRIORITY 則執行 Priority Scheduling，輸入 SJF 則執行 Shortest Job First Scheduling)

```
ThreadedKernel::ThreadedKernel(int argc, char **argv)
{
    randomSlice = FALSE;
    //Add
    type = RR;

    for (int i = 1; i < argc; i++) {
        if (strcmp(argv[i], "-rs") == 0) {
        ASSERT(i + 1 < argc);
        RandomInit(atoi(argv[i + 1]));// initialize pseudo-random
                        // number generator
        randomSlice = TRUE;
        i++;
        } else if (strcmp(argv[i], "-u") == 0) {
            cout << "Partial usage: nachos [-rs randomSeed]\n";
    }

    //Add
        else if(strcmp(argv[i], "RR") == 0) {
            type = RR;
        }
    else if (strcmp(argv[i], "PRIORITY") == 0) {
            type = Priority;
        }
    else if (strcmp(argv[i], "SJF") == 0) {
            type = SJF;
        }
    //
    }
}
```

(4) 在 threads/kernel.h 檔中，在 class ThreadedKernel 的 private 中新增宣告變數 SchedulerType type，讓使用者能多輸入一個參數來指定要用的 Scheduling 方式(如：$ ./nachos RR、$ ./nachos SJF 等)

```
    Thread *currentThread;  // the thread holding the CPU
    Scheduler *scheduler; // the ready list
    Interrupt *interrupt; // interrupt status
    Statistics *stats;    // performance metrics
    Alarm *alarm;   // the software alarm clock

  private:
    bool randomSlice;   // enable pseudo-random time slicing
    SchedulerType type;
};
```

(5) 在 threads/scheduler.h 檔中，新增宣告 SchedulerType 相關的程式碼。

```
enum SchedulerType {
        RR,     // Round Robin
        SJF,
        Priority
};

class Scheduler {
  public:
    Scheduler();        // Initialize list of ready threads
    Scheduler(SchedulerType type);
    ~Scheduler();               // De-allocate ready list

    void ReadyToRun(Thread* thread);
                    // Thread can be dispatched.
    Thread* FindNextToRun();    // Dequeue first thread on the ready
                    // list, if any, and return thread.
    void Run(Thread* nextThread, bool finishing);
                    // Cause nextThread to start running
    void CheckToBeDestroyed();  // Check if thread that had been
                    // running needs to be deleted
    void Print();           // Print contents of ready list

    // SelfTest for scheduler is implemented in class Thread

    //Add
    void setSchedulerType(SchedulerType t) {schedulerType = t;}
    SchedulerType getSchedulerType() {return schedulerType;}
  private:
    SchedulerType schedulerType;
    List<Thread *> *readyList;  // queue of threads that are ready to run,
                    // but not running
    Thread *toBeDestroyed;      // finishing thread to be destroyed
                    // by the next thread that runs
};

#endif // SCHEDULER_H
```

(6) 最後在 threads/scheduler.cc 檔中，根據不同的 Scheduling 方法，來建立出不同的 ReadyList。首先宣告 SJFCompare 和 PriorityCompare 兩個函式，分別對先前所建立的 burstTime 和 priority 做處理。而在這兩變數中，擁有比較小數值的 thread 會先被執行，便能完成執行 SJF Scheduling 和 Priority Scheduling。

```
int SJFCompare(Thread *a, Thread *b) {
    if(a->getBurstTime() == b->getBurstTime())
        return 0;
    return a->getBurstTime() > b->getBurstTime() ? 1 : -1;
    }

int PriorityCompare(Thread *a, Thread *b) {
    if(a->getPriority() == b->getPriority())
        return 0;
    return a->getPriority() > b->getPriority() ? 1 : -1;
    }
```

接著在 Scheduler::Scheduler 的部分，修改成可以輸入 type 的方式，並針對不同的 type 來建立出不同的 List，以完成不同 type 的 Scheduling。一樣若使用者未指定 type 則預設執行 RR Scheduling。

```
Scheduler::Scheduler()
{
    Scheduler(RR);
}

Scheduler::Scheduler(SchedulerType type)
{
    schedulerType = type;
    switch(schedulerType) {
        case RR:
        readyList = new List<Thread *>;
        break;
        case SJF:
        readyList = new SortedList<Thread *>(SJFCompare);
        break;
        case Priority:
        readyList = new SortedList<Thread *>(PriorityCompare);
        break;
    }
    //readyList = new List<Thread *>;
    toBeDestroyed = NULL;
}
```

3. Result

首先在 threads/thread.cc 中建立兩個測資，分別如下兩張圖所示：

```
// Test Case 1
const int number      = 3;
char *name[number]    = {"A", "B", "C"};
int burst[number]     = {3, 10, 4};
int priority[number] = {4, 5, 3};
int arrive[number] = {3, 0, 5};
```

```
// Test Case 2
const int number      = 4;
char *name[number]    = {"A", "B", "C","D"};
int burst[number]     = {3, 9, 7, 4};
int priority[number] = {5, 1, 3, 2};
int arrive[number] = {3, 0, 5, 1};
```

根據 Test 1，若使用 SJF Scheduling，thread 的執行順序為 A->C->B(burst 陣列數值小的先執行)；而若使用 Priority Scheduling，thread 的執行順序為 C->A->B(priority 陣列數值小的先執行)。

根據 Test 2，若使用 SJF Scheduling，thread 的執行順序為 A->D->C->B；而若使用 Priority Scheduling，thread 的執行順序則為 B->D->C->A。

Test 1 執行結果如下：

RR

```
ejlo@OS2:~/nachos-4.0/code/threads$ ./nachos RR
Thread A is running, remaining time =  2
Thread A is running, remaining time =  1
Thread A is running, remaining time =  0
Thread B is running, remaining time =  9
Thread C is running, remaining time =  3
Thread C is running, remaining time =  2
Thread C is running, remaining time =  1
Thread C is running, remaining time =  0
Thread B is running, remaining time =  8
Thread B is running, remaining time =  7
Thread B is running, remaining time =  6
Thread B is running, remaining time =  5
Thread B is running, remaining time =  4
Thread B is running, remaining time =  3
Thread B is running, remaining time =  2
Thread B is running, remaining time =  1
Thread B is running, remaining time =  0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 2600, idle 130, system 2470, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
```

SJF

```
ejlo@OS2:~/nachos-4.0/code/threads$ ./nachos SJF
Thread A is running, remaining time =  2
Thread A is running, remaining time =  1
Thread A is running, remaining time =  0
Thread C is running, remaining time =  3
Thread C is running, remaining time =  2
Thread C is running, remaining time =  1
Thread C is running, remaining time =  0
Thread B is running, remaining time =  9
Thread B is running, remaining time =  8
Thread B is running, remaining time =  7
Thread B is running, remaining time =  6
Thread B is running, remaining time =  5
Thread B is running, remaining time =  4
Thread B is running, remaining time =  3
Thread B is running, remaining time =  2
Thread B is running, remaining time =  1
Thread B is running, remaining time =  0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 2600, idle 130, system 2470, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
```

可看到 thread 的執行順序為 A->C->B，與預期的結果相符。

Prioroty

```
ejlo@OS2:~/nachos-4.0/code/threads$ ./nachos PRIORITY
Thread C is running, remaining time =  3
Thread C is running, remaining time =  2
Thread C is running, remaining time =  1
Thread C is running, remaining time =  0
Thread A is running, remaining time =  2
Thread A is running, remaining time =  1
Thread A is running, remaining time =  0
Thread B is running, remaining time =  9
Thread B is running, remaining time =  8
Thread B is running, remaining time =  7
Thread B is running, remaining time =  6
Thread B is running, remaining time =  5
Thread B is running, remaining time =  4
Thread B is running, remaining time =  3
Thread B is running, remaining time =  2
Thread B is running, remaining time =  1
Thread B is running, remaining time =  0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 2600, idle 130, system 2470, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
```

可看到 thread 的執行順序為 C->A->B，與預期的結果相符。

Test 2 執行結果如下：

RR

```
ejlo@OS2:~/nachos-4.0/code/threads$ ./nachos RR
Thread A is running, remaining time =  2
Thread A is running, remaining time =  1
Thread A is running, remaining time =  0
Thread C is running, remaining time =  6
Thread C is running, remaining time =  5
Thread C is running, remaining time =  4
Thread C is running, remaining time =  3
Thread C is running, remaining time =  2
Thread C is running, remaining time =  1
Thread C is running, remaining time =  0
Thread D is running, remaining time =  3
Thread B is running, remaining time =  8
Thread B is running, remaining time =  7
Thread B is running, remaining time =  6
Thread B is running, remaining time =  5
Thread B is running, remaining time =  4
Thread B is running, remaining time =  3
Thread D is running, remaining time =  2
Thread D is running, remaining time =  1
Thread D is running, remaining time =  0
Thread B is running, remaining time =  2
Thread B is running, remaining time =  1
Thread B is running, remaining time =  0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 2700, idle 140, system 2560, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
```

SJF

```
ejlo@OS2:~/nachos-4.0/code/threads$ ./nachos SJF
Thread A is running, remaining time =  2
Thread A is running, remaining time =  1
Thread A is running, remaining time =  0
Thread D is running, remaining time =  3
Thread D is running, remaining time =  2
Thread D is running, remaining time =  1
Thread D is running, remaining time =  0
Thread C is running, remaining time =  6
Thread C is running, remaining time =  5
Thread C is running, remaining time =  4
Thread C is running, remaining time =  3
Thread C is running, remaining time =  2
Thread C is running, remaining time =  1
Thread C is running, remaining time =  0
Thread B is running, remaining time =  8
Thread B is running, remaining time =  7
Thread B is running, remaining time =  6
Thread B is running, remaining time =  5
Thread B is running, remaining time =  4
Thread B is running, remaining time =  3
Thread B is running, remaining time =  2
Thread B is running, remaining time =  1
Thread B is running, remaining time =  0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 2700, idle 140, system 2560, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
```

可看到 thread 的執行順序為 A->D->C->B，與預期的結果相符。

Priority

```
ejlo@OS2:~/nachos-4.0/code/threads$ ./nachos PRIORITY
Thread B is running, remaining time =  8
Thread B is running, remaining time =  7
Thread B is running, remaining time =  6
Thread B is running, remaining time =  5
Thread B is running, remaining time =  4
Thread B is running, remaining time =  3
Thread B is running, remaining time =  2
Thread B is running, remaining time =  1
Thread B is running, remaining time =  0
Thread D is running, remaining time =  3
Thread D is running, remaining time =  2
Thread D is running, remaining time =  1
Thread D is running, remaining time =  0
Thread C is running, remaining time =  6
Thread C is running, remaining time =  5
Thread C is running, remaining time =  4
Thread C is running, remaining time =  3
Thread C is running, remaining time =  2
Thread C is running, remaining time =  1
Thread C is running, remaining time =  0
Thread A is running, remaining time =  2
Thread A is running, remaining time =  1
Thread A is running, remaining time =  0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 2700, idle 130, system 2570, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
```

可看到 thread 的執行順序為 B->D->C->A，與預期的結果相符。

總結以上，發現在兩個測資中不論哪種測資，三種 Scheduling type 均有正確的執行結果。

討論：

(1) 除了以上三種 Scheduling type 之外，課程上還有講述 FCFS(先進先做)的 Scheduling type，自己一開始也有嘗試實作此一方法，實作方式與其他 type 類似，均是透過建立陣列與判斷用的函式來完成，然而輸出結果卻不知為何，與 RR 的輸出相同，找了許久也未找出原因，因此最後決定不使用此種方法。

(2) 在實作此項作業時，花了不少時間在 trace code 和 debug 上，因為只要一個小地方出錯(例如 Scheduler 的 type 變數忘記補等)，就會導致整個 nachos 無法執行，而自己在實作時，也曾因為未處理好而導致 segmentation fault 發生。後來再從頭重新撰寫一次才成功執行。

Reference

1. https://morris821028.github.io/2014/05/30/lesson/hw-nachos4-2/
2. https://morris821028.github.io/2014/05/24/lesson/hw-nachos4/
3. https://wiiwu959.github.io/2019/10/10/2019-10-10-OS_HW1-2019/
4. https://wiiwu959.github.io/2019/10/29/2019-10-29-OS_HW2-2019/
5. http://blog.terrynini.tw/tw/OS-NachOS-HW1/