OS Nachos Report 2

Motivation

1. Motivation & Problem Analysis

- 前置作業：根據助教的提示，進行相當多的的Trace code及搜尋去了解整個Code運行的模式
- 發現：
  - 時間運行模擬：在整個系統中，我發現時間的運行很容易讓人暈頭轉向。所以為了透徹理解，我從一開始的Thread/Main.cc開始一路往下追蹤，最終整理出了以下的模式。
    - Userprogram Kernel: 繼承Threded Kernel，存有各個Thread(如Project1)
    - Threaded Kernel: 初始化shceduler(處理規劃process排程), interrupt(與整個時間運行相關密切), stats(時間統計), alarm(其中包含Timer作為時間基準)
    - 時間規則
      1. 模擬時間單位是Ticks，其中UserTicks=1, SystemTicks=10, TimerTicks=100，這些累積的資訊，以及目前運行到哪個Tick，存放在Kernel->stats
      2. Interrupt裡面有OneTick()這個方法，這是時間運行的關鍵：每次Machine執行一次Userprogram便會往前跳一次；在Interrupt從關閉到開啟(3.續述)也將跳一次。
      3. 承第2點，scheduler要ReadyToRun()或是FindNextToRUn()要將Interrupt關掉!(影響後面實作Sleep)
      4. Interrupt OneTick()執行到最後，會根據yieldOnReturn的值，進行Context Switch(與排程相關)
      5. 每次Interrupt Idle()時(且沒有process在運行時)， Nachos模擬會進行時間快進，直接跳過空閒時間進到下一個未來Interrupt開始的時間並繼續模擬
      6. interrupt找到時間到期並要開始的中斷時，會讓中斷生效並觸發生效者Alarm->CallBack()(這與後面sleep的關係密切)
      7. 每TimerTicks(100個Ticks)會有一次Timer Interrupt，由Timer向Interrupt申請
      8. Timer關掉(Timer->Disable())等同於模擬結束，只會發生在Idle()且完全沒有未來的interrupt(若有就會按照5.進行快進)
    - 時間順序
      1. Timer向interrupt申請100Ticks後要一次的Timer Interrupt
      2. Interrupt根據到期的Timer Interrupt，執行Alarm->Callback()
      3. 因為Callback()包含 Interrupt->YieldOnReturn()，在這邊會進行Context Switch
      4. 重複以上動作直到全部結束

2. My plan to deal with the problem

在經過以上了解後，我預定的做法如下：

- Sleep
  1. 根據助教投影片的提示，定義新的Syscall、用Machine Code分配暫存器、在Exception中引流到alarm->WaitUntil()
  2. 在WaitUntil()中，由於Thread本身的Sleep()函數原本已有實作，缺乏的只有管控者。我認為Thread應該歸sheduler統一管理運行狀況，故引流到sheduler中。
  3. 由於每100 Ticks會觸發一次Timer Interrupt，所以當Callback()接收到訊號，請管理者scheduler查看誰應該要甦醒。

- CPU scheduling
    - 藉由上面的討論，已經知道Context Switch在哪發生，並且應是由sheduler進行排程管理，所以：
    1. 對於FIFO：因為先來先到，所以Timer Interrupt時不要Context Switch便可以達到。
    2. 對於Priority：設計額外吃Flag的Code，表明每一個的重要性高低，並且改動scheduler FindNextToRun()的行為，讓高優先級的會先被運行。

3. Inplementation

- Sleep

1. 根據提示 Define a system call number of Sleep()：模仿上面的號碼延伸

```
(userprog/syscall.h)
...
#define SC_ThreadYield  10
#define SC_PrintInt 11
#define SC_Sleep 12 // N2
...
```

2. 根據提示：Prepare register for Sleep()：模仿上面的Syscall延伸

```
(test/start.s)
...
    .globl  PrintInt
    .ent    PrintInt
PrintInt:
    addiu   $2,$0,SC_PrintInt
    syscall
    j       $31
    .end    PrintInt

/* N2 */
    .globl Sleep
    .ent Sleep
Sleep:
    addiu   $2,$0,SC_Sleep
    syscall
    j       $31
    .end    Sleep
...
```

3. 根據提示：Add a new case for Sleep() in Exception Handler：也是模仿上面其他Syscall的寫法；根據上面Nachos的註解，可以知道使用者呼叫Sleep(x)裡面的變數x會存在4號暫存器中，故把它取出。根據提示呼叫kernel->alarm->WaitUntil()。

```
(userprog/exception.cc)
...
```

```
case SC_PrintInt:
    val=kernel->machine->ReadRegister(4);
    cout << "Print integer:" <<val << endl;
    return;

// N2
case SC_Sleep:
    cout << "SYSCALL SLEEP" << endl;
    val=kernel->machine->ReadRegister(4);
    kernel->alarm->WaitUntil(val);
    return;
...
```

4. alarm接收到要暫緩執行的指令後，傳送給scheduler請他進行睡覺的管理。

```
(threads/alarm.cc)
...
//N2
void Alarm::WaitUntil(int x){
    DEBUG(dbgThread, "WAITUNTIL "<<x);
    kernel->scheduler->PutCurrentThreadToSleep(x);
}
```

5. scheduler處理PutCurrentThreadToSleep()的要求：
    - 觀察目前scheduler類中的內容，可以發現readyList中存有各個準備運行的Thread，並且其他方法如ReadyToRun()及FindNextToRun()都是直接與他接觸。所以需要另外一個地方存放正在Sleep的Thread
    - 根據提示：Don't forget the useful data structure **list** in the lib folder：在觀看並了解List(以及其iterator)的內容與用法後，建立一個新的**sleepList** 來存放正在睡覺的Thread(模仿readyList的宣告方式，並在建構時初始化之)。

```
(threads/scheduler.h)
...
// N2: Sleep related
List<Thread*>* sleepList;
void PutCurrentThreadToSleep(int duration);
void CheckWakeups();
...
```

```
(threads/scheduler.cc)
Scheduler::Scheduler(SchedulerType schType)
{
    //N2: Specify Scheduler Type
    schedulerType = schType;
    readyList = new List<Thread *>;
    toBeDestroyed = NULL;

    //N2
    sleepList = new List<Thread*>;
}
```

- 為了讓Thread可被控制的去睡覺，需要一個變數來追蹤還要睡多少時間：在此我讓Thread類多了sleepCountdown這個變數，紀錄還要睡多久的時間。

```
(threads/thread.h)
class Thread {
    private:
        ...

    public:
        Thread(char* debugName);        // initialize a Thread
        ~Thread();                // deallocate a Thread
        ...

        // N2: Aspects for sleep and scheduling
        int priorityLevel;
        int burstTime;
        int sleepCountdown;
```

- alarm呼叫scheduler->PutCurrentThreadToSleep()時，設定他要睡多久(duration)、將其加到sleepList確保還能追蹤到，並且呼叫Thread本身的Sleep()方法。而由於Sleep方法會觸發到FindNextToRun()，根據作者的註解，在進行ReadyToRun()或FindToRun()時要關掉Interrupt(若不關掉在呼叫Thread::Sleep()時會跳錯，故參考Thread::Fork()裡面的寫法進行暫時關閉，直到Sleep()執行完畢。

```
(threads/alarm.cc)
    // N2: Sleep
    void Scheduler::PutCurrentThreadToSleep(int duration){
        // FindNextToRun / ReadyToRun must ensure interrupt disabled
        IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);
        Thread* currThr = kernel->currentThread;
        currThr->sleepCountdown = duration;
        sleepList->Append(currThr);
        currThr->Sleep(false); // not finishing just resting

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

- 在進行Trace code後，可以發現原先的Code已經把Sleep()所需要的步驟執行完畢。值得觀察的點有兩項：首先，若找不到下一個運行的Thread，會進行interrupt的Idle：經由第一部分的討論，可以發現這邊Nachos模擬時會快進到下一個interrupt觸發的時間點。其二，根據註解在FindNextToRun()時要關掉interrupt，也呼應了前面採取的作法。

```
(threads/thread.cc)
void
Thread::Sleep (bool finishing)
{
    Thread *nextThread;

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

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

    status = BLOCKED;

    // Find until we get availalbe nextThread
    while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL){
        // no one to run, wait for an interrupt
        kernel->interrupt->Idle();
    }

    // returns when it's time for us to run
    kernel->scheduler->Run(nextThread, finishing);
}
```

以上完成使其睡著的做法，接下來討論使其甦醒的方法。
6. 根據先前的討論，每100 Ticks觸發一次Timer Interrupt，並且由interrupt觸發Alarm的CallBack()方法。所以100個Ticks，要回到sheduler尋找sleepList中有哪些人需要醒來。同時，本來Timer停止的條件為沒有更多的Interrupt(僅有readyList沒東西時才會進行interrupt->Idle()測試，並且試圖快進到下一個interrupt。所以這個敘述等同readyList已經是空的)，但這邊要 **多加一個睡覺的List也要是空** 的才可以一切結束(前面討論過timer->Disable就是全部中止的意思)：

```
(threads/alarm.cc)
void
Alarm::CallBack()
{
    // Provoked by interrupt::oneTick, every 100 Ticks, do a RR context
switch
    Interrupt *interrupt = kernel->interrupt;
    MachineStatus status = interrupt->getStatus();

    if (status == IdleMode  && kernel->scheduler->sleepList->IsEmpty()) {
// is it time to quit?
        if (!interrupt->AnyFutureInterrupts()) {
            timer->Disable();   // turn off the timer
            // This will stop future interrupts!!!
        }
    } else {            // there's someone to preempt
        // N2 I assume this is for RR
        if(kernel->scheduler->schedulerType == RR) interrupt-
>YieldOnReturn();
```

```
        else if(kernel->scheduler->schedulerType == Priority){
            if(kernel->scheduler->ExistHigherPriority()) interrupt-
>YieldOnReturn();
        }
    }

    // N2: Check Wakeups
    kernel->scheduler->CheckWakeups();
}
```

7. 接上一步，由於是每TimerTicks(=100 Ticks)觸發一次，所以遍歷整個sleepList，把每個成員還要睡的時間扣除TimerTicks進行更新。若扣完後時間小於0，代表是時候該起床，紀錄後從sleepList刪除，並讓scheduler以ReadyToRun的方法放到ReadyList排隊準備運行。註：記得要關掉interrupt!!!

```
(threads/scheduler.cc)
...
void Scheduler::CheckWakeups(){
    DEBUG(dbgThread, "SLEEPLIST SIZE "<<sleepList->NumInList());
    ListIterator<Thread*> iter(sleepList);
    while(!iter.IsDone()){
        Thread* currentSleeper = iter.Item();
        iter.Next();
        currentSleeper->sleepCountdown -= TimerTicks;
        DEBUG(dbgThread, "SLEEPER "<<currentSleeper->getName()<<" REMAINING
"<<currentSleeper->sleepCountdown);

        if(currentSleeper->sleepCountdown <= 0){
            DEBUG(dbgThread, "TIME TO WAKE UP: "<<currentSleeper-
>getName());
            sleepList->Remove(currentSleeper);

            // FindNextToRun / ReadyToRun must ensure interrupt disabled
            IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);
            ReadyToRun(currentSleeper);
            kernel->interrupt->SetLevel(oldLevel);
        }
    }
}
```

8. 以上便是主要的實作。另外我設定了一些Debug的字串，讓待會測試的時候可以看得更清楚。

- CPU Sheduling
    - 指定排程方法：由於要切換不同的排程方法，所以在一開始cmd執行放flag時，就要規定目前用的是哪一個排程模式。從助教提示的main.cc往下Trace code，可以發現與thread相關的parse arguments在userkernel.cc及kernel.cc當中。由於兩者互為繼承關係，我決定把指定排程的參數設在kernel.cc中：
        - 新變數：存放排程的對應enum號碼

```
(threads/kernel.h)
class ThreadedKernel {
    public:
        ThreadedKernel(int argc, char **argv);
                        // Interpret command line arguments
        ~ThreadedKernel();      // deallocate the kernel
        ...
        SchedulerType schType; //N2

    private:
        ...
    };
```

- 初始化：模仿其他flag parse的方法，在建構子時初始化為想要的排程方法。以後要呼叫時，便可以用 -sch -[r/s/p/f]，分別對應RR/SJF/Priority/FIFO。

```
(threads/kernel.cc)
ThreadedKernel::ThreadedKernel(int argc, char **argv)
{
    schType = RR;
    randomSlice = FALSE;
    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";
        }
        //N2 Parse here
        else if (strcmp(argv[i], "-sch") == 0){
            ASSERT(i + 1 < argc);    // next argument define scheduler
type
            char* schArg = argv[i + 1];
            if(strcmp(schArg, "r") == 0) schType = RR;
            else if(strcmp(schArg, "s") == 0) schType = SJF;
            else if(strcmp(schArg, "p") == 0) schType = Priority;
            else if(strcmp(schArg, "f") == 0) schType = FIFO;
            i++;
        }
    }
}
```

1. Round Robin (RR)：此為原來Code預設的排程方法。就前面的討論，可知道每100個Ticks便會由alarm->Callback()觸發一次的context switch，也就是最多每100 Ticks將會把CPU的使用權限交給

下一個在等待的Thread。從第一次作業的Thread Management中，也可以觀察到這樣的現象(先印出test1、再進到test2、如此反覆到執行結束)。

2. FIFO：對於先進先出的排程方法，應該是一個Thread完全執行完畢(除非該Thread主動讓給其他人)才會換到下一個來的Thread當中。所以只要把每次Timer Interrupt的Context Switch關掉，便可以達到所要的方法：只有在schedulerType是RR時，才要呼叫YieldOnReturn()，讓下次interrupt->OneTick()時觸發Thread->Yield。而在這邊FIFO會直接跳過這行，繼續執行現在的Thread。

```
(threads/alarm.cc)
void
Alarm::CallBack()
{
    // Provoked by interrupt::oneTick, every 100 Ticks, do a RR context
switch
    Interrupt *interrupt = kernel->interrupt;
    MachineStatus status = interrupt->getStatus();

    if (status == IdleMode  && kernel->scheduler->sleepList->IsEmpty())
{   // is it time to quit?
        if (!interrupt->AnyFutureInterrupts()) {
            timer->Disable();   // turn off the timer
            // This will stop future interrupts!!!
        }
    } else {            // there's someone to preempt
        // N2 I assume this is for RR
        if(kernel->scheduler->schedulerType == RR) interrupt-
>YieldOnReturn();
        else if(kernel->scheduler->schedulerType == Priority){
            if(kernel->scheduler->ExistHigherPriority()) interrupt-
>YieldOnReturn();
        }
    }

    // N2: Check Wakeups
kernel->scheduler->CheckWakeups();
}
```

3. Priority：這邊還要決定每個Thread的優先級：

   - 在Thread類中新增Priority相關的參數：prioirtyLevel，數字越小代表優先級越高。

```
(threads/thread.h)
class Thread {
    private:
        ...

    public:
        Thread(char* debugName);        // initialize a Thread
        ~Thread();                // deallocate a Thread
        ...
        // N2: Aspects for sleep and scheduling
        int priorityLevel;
        int burstTime;
        int sleepCountdown;

    private:
        ...
    };
```

- 在呼叫 -e [Some Filename]時，後面多吃一個數字來表示為該Thread的優先級。觀察-e這個Flag的Parsing Argument放在userkernel.cc中：當選定的排程模式是Priority時，把下面一個參數當成優先級傳入(例如：./userprog/nachos -e ./test/test1 12 -e ./test/test2 1)

```
(userprog/userkernel.cc)
UserProgKernel::UserProgKernel(int argc, char **argv)
: ThreadedKernel(argc, argv)
{
    ...
    else if (strcmp(argv[i], "-e") == 0) {
        //execfile[++execfileNum]= argv[++i];
        //N2
        ++execfileNum;
        execfile[execfileNum]= argv[++i];
        if(schType==Priority){
            prioirtyList[execfileNum] = atoi(argv[++i]);
        }
    }
       ...
    }
}
```

- 並且在各Thread建立時，若排程是Priorty則把優先級設立。

```
(userprog/userkernel.cc)
void
UserProgKernel::Run()
{

    cout << "Total threads number is " << execfileNum << endl;
    for (int n=1;n<=execfileNum;n++)
        {
        t[n] = new Thread(execfile[n]);
        t[n]->space = new AddrSpace();

        //N2
        if(schType==Priority){
            t[n]->priorityLevel = prioirtyList[n];
            cout << "Thread " << execfile[n] << " is executing with
priority "<<prioirtyList[n] << endl;
        }

        t[n]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[n]);
        cout << "Thread " << execfile[n] << " is executing." << endl;
        }
    ThreadedKernel::Run();
}
```

- Priority改變的是Scheduler::FindNextToRun()的運行模式。一開始對於RR或FIFO而言，都是拿readyList的第一個Thread運行(同時從List中刪除該Thread)。在Priority中，則是要找到優先級最高(也就是priorityLevel數值最低者)運行，並且從readyList中刪除。在這邊我使用iterator遍歷整個readyList，鎖定最小的之後，將其從readyList刪除後回傳：

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

    if (readyList->IsEmpty()) {
        return NULL;
    } else {
        if(schedulerType==RR || schedulerType==FIFO){
            return readyList->RemoveFront();
        }
        else if (schedulerType==Priority){
            ListIterator<Thread*> iter(readyList);
            Thread* targetThread = iter.Item();
            iter.Next();

            while(!iter.IsDone()){
                if(iter.Item()->priorityLevel < targetThread-
>priorityLevel){
                    // Priority Level value Lower -> is more important
                    targetThread = iter.Item();
                }
                iter.Next();
            }
            readyList->Remove(targetThread);
            return targetThread;
        }
        else{
            return readyList->RemoveFront();
        }
    }
}
(threads/scheduler.cc)
```

- 最後，對於新回到readyList的Thread，若是其優先級比目前運行的Thread更高，應要觸發Context Switch。由於Context Switch發生在每100 Ticks的alarm::CallBack()當中，我在這邊設

計一個新的方法scheduler::ExistHigherPriority()，讓scheduler觀察是否有優先級更高的Thread存在於readyList當中：

```
bool Scheduler::ExistHigherPriority(){
    int priorityNow = kernel->currentThread->priorityLevel;
    DEBUG(dbgThread, "READYLIST SIZE "<<readyList->NumInList());
    ListIterator<Thread*> iter(readyList);

    while(!iter.IsDone()){
        Thread* currentThread = iter.Item();
        iter.Next();
        if(currentThread->priorityLevel < priorityNow) return true;
    }
    return false;
}
```

- 承上面，如果有更優先的存在，在Callback()的時候讓cuurentThread讓出CPU，這時候就會自動找到高優先級的並繼續執行：

```
void
Alarm::CallBack()
{
    // Provoked by interrupt::oneTick, every 100 Ticks, do a RR context
switch
    Interrupt *interrupt = kernel->interrupt;
    MachineStatus status = interrupt->getStatus();

    if (status == IdleMode  && kernel->scheduler->sleepList->IsEmpty())
{   // is it time to quit?
        if (!interrupt->AnyFutureInterrupts()) {
            timer->Disable();   // turn off the timer
            // This will stop future interrupts!!!
        }
    } else {            // there's someone to preempt
        // N2 I assume this is for RR
        if(kernel->scheduler->schedulerType == RR) interrupt-
>YieldOnReturn();
        else if(kernel->scheduler->schedulerType == Priority){
            if(kernel->scheduler->ExistHigherPriority()) interrupt-
>YieldOnReturn();
        }
    }

    // N2: Check Wakeups
    kernel->scheduler->CheckWakeups();
}
```

4. Result

- Sleep 測試
  - Write your own test code test.c like test1 and test2: 為了進行測試並看Sleep的結果，我在test.c中，讓其呼叫五次Sleep的Syscall，並且睡眠時間隨次數遞增，從一開始的200 Ticks，每次增加100Ticks，最後一次會睡600Ticks。每次睡完後印出-9999(僅是方便辨識)。

```
(test/test.c)
#include "syscall.h"
main(){
    int i=0;
    for (i=0; i<5;i++){
        Sleep(200+i*100);
        PrintInt(-9999);
    }
}
```

  - 執行結果(1): 開啟Thread的Debug、印出所有動作並觀察。可以發現這跟前面討論的相符：呼叫Syscall call後，觸發waituntil，進到sleepList中；之後每100Ticks觸發並減少剩餘時間，醒來後印出前面刻意的-9999，同時下一次的睡眠時間增長。

```
jch@JCH:~/nachos-4.0/code$ ./userprog/nachos -d t -e ./test/test
Entering main
Total threads number is 1
Forking thread: ./test/test f(a): 134583029 0x95e0500
Putting thread on ready list: ./test/test
Thread ./test/test is executing.
Finishing thread: main
Sleeping thread: main
Switching from: main to: ./test/test
Beginning thread: ./test/test
Deleting thread: main
SYSCALL SLEEP
WAITUNTIL 200
Sleeping thread: ./test/test
SLEEPLIST SIZE 1
SLEEPER ./test/test REMAINING 100
SLEEPLIST SIZE 1
SLEEPER ./test/test REMAINING 0
TIME TO WAKE UP: ./test/test
Putting thread on ready list: ./test/test
Switching from: ./test/test to: ./test/test
Now in thread: ./test/test
Yielding thread: ./test/test
Print integer:-9999
SYSCALL SLEEP
WAITUNTIL 300
Sleeping thread: ./test/test
SLEEPLIST SIZE 1
SLEEPER ./test/test REMAINING 200
SLEEPLIST SIZE 1
```

```
SLEEPER ./test/test REMAINING 100
SLEEPLIST SIZE 1
SLEEPER ./test/test REMAINING 0
TIME TO WAKE UP: ./test/test
Putting thread on ready list: ./test/test
Switching from: ./test/test to: ./test/test
Now in thread: ./test/test
Yielding thread: ./test/test
Print integer:-9999
SYSCALL SLEEP
WAITUNTIL 400
Sleeping thread: ./test/test
SLEEPLIST SIZE 1
SLEEPER ./test/test REMAINING 300
SLEEPLIST SIZE 1
SLEEPER ./test/test REMAINING 200
SLEEPLIST SIZE 1
SLEEPER ./test/test REMAINING 100
SLEEPLIST SIZE 1
SLEEPER ./test/test REMAINING 0
TIME TO WAKE UP: ./test/test
Putting thread on ready list: ./test/test
Switching from: ./test/test to: ./test/test
Now in thread: ./test/test
Yielding thread: ./test/test
Print integer:-9999
SYSCALL SLEEP
WAITUNTIL 500
Sleeping thread: ./test/test
SLEEPLIST SIZE 1
SLEEPER ./test/test REMAINING 400
SLEEPLIST SIZE 1
SLEEPER ./test/test REMAINING 300
SLEEPLIST SIZE 1
SLEEPER ./test/test REMAINING 200
SLEEPLIST SIZE 1
SLEEPER ./test/test REMAINING 100
SLEEPLIST SIZE 1
SLEEPER ./test/test REMAINING 0
TIME TO WAKE UP: ./test/test
Putting thread on ready list: ./test/test
Switching from: ./test/test to: ./test/test
Now in thread: ./test/test
Yielding thread: ./test/test
Print integer:-9999
SYSCALL SLEEP
WAITUNTIL 600
Sleeping thread: ./test/test
SLEEPLIST SIZE 1
SLEEPER ./test/test REMAINING 500
SLEEPLIST SIZE 1
SLEEPER ./test/test REMAINING 400
SLEEPLIST SIZE 1
SLEEPER ./test/test REMAINING 300
```

```
SLEEPLIST SIZE 1
SLEEPER ./test/test REMAINING 200
SLEEPLIST SIZE 1
SLEEPER ./test/test REMAINING 100
SLEEPLIST SIZE 1
SLEEPER ./test/test REMAINING 0
TIME TO WAKE UP: ./test/test
Putting thread on ready list: ./test/test
Switching from: ./test/test to: ./test/test
Now in thread: ./test/test
Yielding thread: ./test/test
Print integer:-9999
return value:0
Finishing thread: ./test/test
Sleeping thread: ./test/test
SLEEPLIST SIZE 0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 2100, idle 1789, system 130, user 181
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
```

- 執行結果(2)：把睡眠的test.c跟另外一個test4.c一起執行，其中test4.c相當單純，只是從0印到99。可以發現符合一開始test的設計：每兩次Syscall sleep中間因為會隨著次數睡眠時間增長(200 Ticks ->300 Ticks ->400 Ticks ->500 Ticks ->600 Ticks)，由另外一個Thread印出的整數數量上升(從6個->13個->19個->24個->30個)。

```
(test/test4.c)
#include "syscall.h"
main()
    {
        int n;
        for (n=0;n<100;n++)
            PrintInt(n);
    }
```

```
jch@JCH:~/nachos-4.0/code$ ./userprog/nachos -e ./test/test -e ./test/test4
Total threads number is 2
Thread ./test/test is executing.
Thread ./test/test4 is executing.
SYSCALL SLEEP
Print integer:0
Print integer:1
Print integer:2
Print integer:3
Print integer:4
Print integer:5
Print integer:-9999
SYSCALL SLEEP
Print integer:6
Print integer:7
Print integer:8
Print integer:9
Print integer:10
Print integer:11
Print integer:12
Print integer:13
Print integer:14
Print integer:15
Print integer:16
Print integer:17
Print integer:18
Print integer:-9999
SYSCALL SLEEP
Print integer:19
Print integer:20
Print integer:21
Print integer:22
Print integer:23
Print integer:24
Print integer:25
Print integer:26
Print integer:27
Print integer:28
Print integer:29
Print integer:30
Print integer:31
Print integer:32
Print integer:33
Print integer:34
Print integer:35
Print integer:36
Print integer:37
Print integer:-9999
SYSCALL SLEEP
Print integer:38
Print integer:39
Print integer:40
Print integer:41
Print integer:42
Print integer:43
Print integer:44
Print integer:45
Print integer:46
Print integer:47
Print integer:48
Print integer:49
Print integer:50
Print integer:51
Print integer:52
Print integer:53
Print integer:54
Print integer:55
Print integer:56
Print integer:57
Print integer:58
Print integer:59
Print integer:60
Print integer:61
Print integer:-9999
SYSCALL SLEEP
Print integer:62
Print integer:63
Print integer:64
Print integer:65
Print integer:66
Print integer:67
Print integer:68
Print integer:69
Print integer:70
Print integer:71
Print integer:72
Print integer:73
Print integer:74
Print integer:75
Print integer:76
Print integer:77
Print integer:78
Print integer:79
Print integer:80
Print integer:81
Print integer:82
Print integer:83
Print integer:84
Print integer:85
Print integer:86
Print integer:87
Print integer:88
Print integer:89
Print integer:90
Print integer:91
Print integer:-9999
return value:0
Print integer:92
Print integer:93
Print integer:94
Print integer:95
Print integer:96
Print integer:97
Print integer:98
Print integer:99
return value:0
```

```
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 2300, idle 74, system 320, user 1906
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
```

- CPU Scheduling 測試(>=2 test case)
    - Test case 設計：很簡單的三個檔案test1.c, test2.c, test3.c，其中test1.c印出99~50(兩位數的正整數)、test2.c印出100~150(三位數的正整數)、test3.c印出-50~-100(負數)。藉由位數跟正負號可以方便觀察到底是哪個Thread正在執行。

```
(test/test1.c)
#include "syscall.h"
main()
    {
        int n;
        for (n=99;n>=50;n--)
            PrintInt(n);
    }
```

```
#include "syscall.h"
(test/test2.c)
main()
        {
            int     n;
            for (n=100;n<=150;n++)
                    PrintInt(n);
        }
```

```
(test/test3.c)
#include "syscall.h"
main()
    {
        int n;
        for (n=-50;n>=-100;n--)
            PrintInt(n);
    }
```

1. RR：一開始預設的排程方法，所以輸出模式會跟project1相同，每次Timer Interrupt執行Context Switch。所以可以看到結果，兩位數(test1)與三位數(test2)與負數(test3)每隔一段時間(100 Ticks)便

會讓給下個人，形成Round Robin。

```
jch@JCH:~/nachos-4.0/code$ ./userprog/nachos -sch r -e ./test/test1 -e
./test/test2 -e ./test/test3
Total threads number is 3
Thread ./test/test1 is executing.
Thread ./test/test2 is executing.
Thread ./test/test3 is executing.
Print integer:99
Print integer:98
Print integer:100
Print integer:101
Print integer:102
Print integer:103
Print integer:104
Print integer:-50
Print integer:-51
Print integer:-52
Print integer:-53
Print integer:-54
Print integer:97
Print integer:96
Print integer:95
Print integer:94
Print integer:93
Print integer:92
Print integer:105
Print integer:106
Print integer:107
Print integer:108
Print integer:109
Print integer:-55
Print integer:-56
Print integer:-57
Print integer:-58
Print integer:-59
Print integer:91
Print integer:90
Print integer:89
Print integer:88
Print integer:87
Print integer:110
Print integer:111
Print integer:112
Print integer:113
Print integer:114
Print integer:-60
Print integer:-61
Print integer:-62
Print integer:-63
Print integer:-64
Print integer:86
Print integer:85
Print integer:84
Print integer:83
Print integer:82
Print integer:115
Print integer:116
Print integer:117
Print integer:118
Print integer:119
Print integer:-65
Print integer:-66
Print integer:-67
Print integer:-68
Print integer:-69
Print integer:81
Print integer:80
Print integer:79
Print integer:78
Print integer:77
Print integer:120
Print integer:121
Print integer:122
Print integer:123
Print integer:124
Print integer:125
Print integer:-70
Print integer:-71
Print integer:-72
Print integer:-73
Print integer:-74
Print integer:-75
Print integer:76
Print integer:75
Print integer:74
Print integer:73
Print integer:72
Print integer:71
Print integer:126
Print integer:127
Print integer:128
Print integer:129
Print integer:130
Print integer:-76
Print integer:-77
Print integer:-78
Print integer:-79
Print integer:-80
Print integer:70
Print integer:69
Print integer:68
Print integer:67
Print integer:66
Print integer:131
Print integer:132
Print integer:133
Print integer:134
Print integer:135
Print integer:-81
Print integer:-82
Print integer:-83
Print integer:-84
Print integer:-85
Print integer:65
Print integer:64
Print integer:63
Print integer:62
Print integer:61
Print integer:136
Print integer:137
Print integer:138
Print integer:139
Print integer:140
Print integer:141
Print integer:-86
Print integer:-87
Print integer:-88
Print integer:-89
Print integer:-90
Print integer:-91
Print integer:60
Print integer:59
Print integer:58
Print integer:57
Print integer:56
Print integer:55
Print integer:142
Print integer:143
Print integer:144
Print integer:145
Print integer:146
Print integer:-92
Print integer:-93
Print integer:-94
Print integer:-95
Print integer:-96
Print integer:54
Print integer:53
Print integer:52
Print integer:51
Print integer:50
Print integer:147
Print integer:148
Print integer:149
Print integer:150
return value:0
return value:0
Print integer:-97
Print integer:-98
Print integer:-99
Print integer:-100
return value:0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 3103, idle 71, system 370, user 2662
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
```

2. FIFO
   - 執行結果一：執行順序不變，排程方法從RR轉換成FIFO(藉由-sch f這個Flag)，可以發現結果便是先進先出：先處理test1(兩位數)、結束後處理test2(三位數)、結束後才進到test3(負數)：

```
jch@JCH:~/nachos-4.0/code$ ./userprog/nachos -sch f -e ./test/test1 -e
./test/test2 -e ./test/test3
Total threads number is 3
Thread ./test/test1 is executing.
Thread ./test/test2 is executing.
Thread ./test/test3 is executing.
Print integer:99
Print integer:98
Print integer:97
Print integer:96
Print integer:95
Print integer:94
Print integer:93
Print integer:92
Print integer:91
Print integer:90
Print integer:89
Print integer:88
Print integer:87
Print integer:86
Print integer:85
Print integer:84
Print integer:83
Print integer:82
Print integer:81
Print integer:80
Print integer:79
Print integer:78
Print integer:77
Print integer:76
Print integer:75
Print integer:74
Print integer:73
Print integer:72
Print integer:71
Print integer:70
Print integer:69
Print integer:68
Print integer:67
Print integer:66
Print integer:65
Print integer:64
Print integer:63
Print integer:62
Print integer:61
Print integer:60
Print integer:59
Print integer:58
Print integer:57
Print integer:56
Print integer:55
Print integer:54
Print integer:53
Print integer:52
Print integer:51
Print integer:50
return value:0
Print integer:100
Print integer:101
Print integer:102
Print integer:103
Print integer:104
Print integer:105
Print integer:106
Print integer:107
Print integer:108
Print integer:109
Print integer:110
Print integer:111
Print integer:112
Print integer:113
Print integer:114
Print integer:115
Print integer:116
Print integer:117
Print integer:118
Print integer:119
Print integer:120
Print integer:121
Print integer:122
Print integer:123
Print integer:124
Print integer:125
Print integer:126
Print integer:127
Print integer:128
Print integer:129
Print integer:130
Print integer:131
Print integer:132
Print integer:133
Print integer:134
Print integer:135
Print integer:136
Print integer:137
Print integer:138
Print integer:139
Print integer:140
Print integer:141
Print integer:142
Print integer:143
Print integer:144
Print integer:145
Print integer:146
Print integer:147
Print integer:148
Print integer:149
Print integer:150
return value:0
Print integer:-50
Print integer:-51
Print integer:-52
Print integer:-53
Print integer:-54
Print integer:-55
Print integer:-56
Print integer:-57
Print integer:-58
Print integer:-59
Print integer:-60
Print integer:-61
Print integer:-62
Print integer:-63
Print integer:-64
Print integer:-65
Print integer:-66
Print integer:-67
Print integer:-68
Print integer:-69
Print integer:-70
Print integer:-71
Print integer:-72
Print integer:-73
Print integer:-74
Print integer:-75
Print integer:-76
Print integer:-77
Print integer:-78
Print integer:-79
Print integer:-80
Print integer:-81
Print integer:-82
Print integer:-83
Print integer:-84
Print integer:-85
Print integer:-86
Print integer:-87
Print integer:-88
Print integer:-89
Print integer:-90
Print integer:-91
Print integer:-92
Print integer:-93
Print integer:-94
Print integer:-95
Print integer:-96
Print integer:-97
Print integer:-98
Print integer:-99
Print integer:-100
return value:0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 2800, idle 68, system 70, user 2662
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
```

- 執行結果二：把前面sleep所設計的test.c拿來用：傳入順序為test.c(包含Sleep的Syscall)、test2.c(三位數)、test3.c(負數)。可見一開始test.c因為sleep放掉使用權後，由test2接管並把自己的部分執行完畢，之後test3接手也把自己的執行完。醒來後的test.c由於排在隊列最後面，在FIFO機制下等到兩位執行完後才執行完成。

```
jch@JCH:~/nachos-4.0/code$ ./userprog/nachos -sch f -e ./test/test -e
./test/test2 -e ./test/test3
Total threads number is 3
Thread ./test/test is executing.
Thread ./test/test2 is executing.
Thread ./test/test3 is executing.
SYSCALL SLEEP
Print integer:100
Print integer:101
Print integer:102
Print integer:103
Print integer:104
Print integer:105
Print integer:106
Print integer:107
Print integer:108
Print integer:109
Print integer:110
Print integer:111
Print integer:112
Print integer:113
Print integer:114
Print integer:115
Print integer:116
Print integer:117
Print integer:118
Print integer:119
Print integer:120
Print integer:121
Print integer:122
Print integer:123
Print integer:124
Print integer:125
Print integer:126
Print integer:127
Print integer:128
Print integer:129
Print integer:130
Print integer:131
Print integer:132
Print integer:133
Print integer:134
Print integer:135
Print integer:136
Print integer:137
Print integer:138
Print integer:139
Print integer:140
Print integer:141
Print integer:142
Print integer:143
Print integer:144
Print integer:145
Print integer:146
Print integer:147
Print integer:148
Print integer:149
Print integer:150
return value:0
Print integer:-50
Print integer:-51
Print integer:-52
Print integer:-53
Print integer:-54
Print integer:-55
Print integer:-56
Print integer:-57
Print integer:-58
Print integer:-59
Print integer:-60
Print integer:-61
Print integer:-62
Print integer:-63
Print integer:-64
Print integer:-65
Print integer:-66
Print integer:-67
Print integer:-68
Print integer:-69
Print integer:-70
Print integer:-71
Print integer:-72
Print integer:-73
Print integer:-74
Print integer:-75
Print integer:-76
Print integer:-77
Print integer:-78
Print integer:-79
Print integer:-80
Print integer:-81
Print integer:-82
Print integer:-83
Print integer:-84
Print integer:-85
Print integer:-86
Print integer:-87
Print integer:-88
Print integer:-89
Print integer:-90
Print integer:-91
Print integer:-92
Print integer:-93
Print integer:-94
Print integer:-95
Print integer:-96
Print integer:-97
Print integer:-98
Print integer:-99
Print integer:-100
return value:0
Print integer:-9999
SYSCALL SLEEP
Print integer:-9999
SYSCALL SLEEP
Print integer:-9999
SYSCALL SLEEP
Print integer:-9999
SYSCALL SLEEP
Print integer:-9999
return value:0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

```
Ticks: total 3800, idle 1713, system 120, user 1967
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
```

3. Priority
   - 執行結果(1)：分別給予test1優先及99、test2優先級88、test3優先級12。由於優先級定義為越小越優先，故執行順序test3(負數)->test2(三位數)->test1(兩位數)

```
jch@JCH:~/nachos-4.0/code$ ./userprog/nachos -sch p -e ./test/test1 99
-e ./test/test2 88 -e ./test/test3 12
Total threads number is 3
Thread ./test/test1 is executing with priority 99
Thread ./test/test1 is executing.
Thread ./test/test2 is executing with priority 88
Thread ./test/test2 is executing.
Thread ./test/test3 is executing with priority 12
Thread ./test/test3 is executing.
Print integer:-50
Print integer:-51
Print integer:-52
Print integer:-53
Print integer:-54
Print integer:-55
Print integer:-56
Print integer:-57
Print integer:-58
Print integer:-59
Print integer:-60
Print integer:-61
Print integer:-62
Print integer:-63
Print integer:-64
Print integer:-65
Print integer:-66
Print integer:-67
Print integer:-68
Print integer:-69
Print integer:-70
Print integer:-71
Print integer:-72
Print integer:-73
Print integer:-74
Print integer:-75
Print integer:-76
Print integer:-77
Print integer:-78
Print integer:-79
Print integer:-80
Print integer:-81
Print integer:-82
Print integer:-83
Print integer:-84
Print integer:-85
Print integer:-86
Print integer:-87
Print integer:-88
Print integer:-89
Print integer:-90
Print integer:-91
Print integer:-92
Print integer:-93
Print integer:-94
Print integer:-95
Print integer:-96
Print integer:-97
Print integer:-98
Print integer:-99
Print integer:-100
return value:0
Print integer:100
Print integer:101
Print integer:102
Print integer:103
Print integer:104
Print integer:105
Print integer:106
Print integer:107
Print integer:108
Print integer:109
Print integer:110
Print integer:111
Print integer:112
Print integer:113
Print integer:114
Print integer:115
Print integer:116
Print integer:117
Print integer:118
Print integer:119
Print integer:120
Print integer:121
Print integer:122
Print integer:123
Print integer:124
Print integer:125
Print integer:126
Print integer:127
Print integer:128
Print integer:129
Print integer:130
Print integer:131
Print integer:132
Print integer:133
Print integer:134
Print integer:135
Print integer:136
Print integer:137
Print integer:138
Print integer:139
Print integer:140
Print integer:141
Print integer:142
Print integer:143
Print integer:144
Print integer:145
Print integer:146
Print integer:147
Print integer:148
Print integer:149
Print integer:150
return value:0
Print integer:99
Print integer:98
Print integer:97
Print integer:96
Print integer:95
Print integer:94
Print integer:93
Print integer:92
Print integer:91
Print integer:90
Print integer:89
Print integer:88
Print integer:87
Print integer:86
Print integer:85
Print integer:84
Print integer:83
Print integer:82
Print integer:81
Print integer:80
Print integer:79
Print integer:78
Print integer:77
Print integer:76
Print integer:75
Print integer:74
Print integer:73
Print integer:72
Print integer:71
Print integer:70
Print integer:69
Print integer:68
Print integer:67
Print integer:66
Print integer:65
Print integer:64
Print integer:63
Print integer:62
Print integer:61
Print integer:60
Print integer:59
Print integer:58
Print integer:57
Print integer:56
Print integer:55
Print integer:54
Print integer:53
Print integer:52
Print integer:51
Print integer:50
return value:0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 2800, idle 68, system 70, user 2662
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
```

- 執行結果(2)：分別給予test.c(包含Sleep的Syscall)優先及0、test2.c(印出三位數)優先級2、test3.c(印出負數)優先級1。在優先級越小越優先下，包含Sleep的test.c會有最高優先級，每次醒來後回到readyList後，會把優先級較低的test2.c與test3.c擠出CPU，並且印出-9999後繼續睡下去。而test3優先級高於test2，執行時負數(test3)永遠在三位數(test2)之前。

```
jch@JCH:~/nachos-4.0/code$ ./userprog/nachos -sch p -e ./test/test 0 -e
./test/test2 2 -e ./test/test3 1
Total threads number is 3
Thread ./test/test is executing with priority 0
Thread ./test/test is executing.
Thread ./test/test2 is executing with priority 2
Thread ./test/test2 is executing.
Thread ./test/test3 is executing with priority 1
Thread ./test/test3 is executing.
SYSCALL SLEEP
Print integer:-50
Print integer:-51
Print integer:-52
Print integer:-53
Print integer:-54
Print integer:-55
Print integer:-56
Print integer:-57
Print integer:-58
Print integer:-59
Print integer:-60
Print integer:-61
Print integer:-9999
SYSCALL SLEEP
Print integer:-62
Print integer:-63
Print integer:-64
Print integer:-65
Print integer:-66
Print integer:-67
Print integer:-68
Print integer:-69
Print integer:-70
Print integer:-71
Print integer:-72
Print integer:-73
Print integer:-74
Print integer:-75
Print integer:-76
Print integer:-77
Print integer:-78
Print integer:-79
Print integer:-80
Print integer:-81
Print integer:-9999
SYSCALL SLEEP
Print integer:-82
Print integer:-83
Print integer:-84
Print integer:-85
Print integer:-86
Print integer:-87
Print integer:-88
Print integer:-89
Print integer:-90
Print integer:-91
Print integer:-92
Print integer:-93
Print integer:-94
Print integer:-95
Print integer:-96
Print integer:-97
Print integer:-98
Print integer:-99
Print integer:-100
return value:0
Print integer:100
Print integer:101
Print integer:102
Print integer:103
Print integer:104
Print integer:105
Print integer:-9999
SYSCALL SLEEP
Print integer:106
Print integer:107
Print integer:108
Print integer:109
Print integer:110
Print integer:111
Print integer:112
Print integer:113
Print integer:114
Print integer:115
Print integer:116
Print integer:117
Print integer:118
Print integer:119
Print integer:120
Print integer:121
Print integer:122
Print integer:123
Print integer:124
Print integer:125
Print integer:126
Print integer:127
Print integer:128
Print integer:129
Print integer:130
Print integer:131
Print integer:132
Print integer:133
Print integer:134
Print integer:135
Print integer:136
Print integer:137
Print integer:-9999
SYSCALL SLEEP
Print integer:138
Print integer:139
Print integer:140
Print integer:141
Print integer:142
Print integer:143
Print integer:144
Print integer:145
Print integer:146
Print integer:147
Print integer:148
Print integer:149
Print integer:150
return value:0
Print integer:-9999
return value:0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 2500, idle 373, system 160, user 1967
```

```
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
```

Extra effort / observation / reflection

這次的Project感覺花了很多時間了解Nachos模擬的做法。相對於上一次的Thread Management時處理每個Thread對應到的Address Space，這次把kernel的各個部分，包含scheduler、interrupt、alarm、stats四者都走過了一遍，在觀察彼此合作模式的同時，又由於這四者間互相呼叫而顯得更加錯綜複雜。從一開始不大懂Hint背後所含的意思，到一步步了解，歸納後，終於了解Code運作的方式，並到最後可以把一些想法加進去，這整個過程著實花了不少時間；然而也讓我的確相比於上次的Project，對Nachos的作業系統又更深了一層。而在設計Test Case時，為了讓我更了解到自己的做法會產生怎麼樣的影響，我試圖將Sleep與CPU Scheduling的功能合併測試(如以上的實驗)，並且從錯誤中反思應該怎麼樣改為我想要的模式，直到產生我要的成果。總歸而言，這次Project實在讓我觀察到了很多Nachos的設計，無論是藉由註解、投影片、其他大學的文檔、又或是我測試下得出的結果。