作業系統 - 作業二 (System Call & CPU Scheduling)

系級：電機所碩一　姓名：楊冠彥　學號：R11921091

tags: `Operating system`, `Ubuntu 14.04`

Motivation

Part 1. System call - Sleep()

作業的目標是要實現Sleep()的功能，根據作業的說明，可以輕易確認我們需要完成WaitUntil()這個function，並且當我們呼叫CallBack() 時要幫我們檢查應該喚醒哪個thread，在alarm.c應該要實現每 X 次tick產生一個interrupt，並在userprog/syscall.h定義Sleep system call編號，在test/start.s準備暫存器，在execption.cc的 ExceptionHandler 中為 Sleep() 增加一個新case，並要注意 kernel->alarm->WaitUntil() 的使用，這樣應該可以完成Part1的部分。

Part 2. CPU scheduling

基本上看到題目，我就先去看scheduler(scheduler.cc&scheduler.h)和thread(thread.cc&thread.h)相關的程式碼，在scheduler.cc有看到如下程式碼：

`Code`

code/threads/scheduler.cc

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

因此可推斷應該只要把scheduler功能寫在這個檔案，再在其他程式補上對應需要的部分，就能實現。另外在看thread.cc時有發現scheduler原本應該是直接按readyList裡的順序執行thread，所以前述功能部份應該可以在`scheduler.cc`中將thread按照規則放進readyList中，來達成需要的功能，接著應該只要逐步補上因為這邊更動，導致須更動的其他部分程式碼，並在kernel.cc撰寫測試部分即可。最後作業說明有要求當製作多個sceduler方法時，需要製作sceduler type切換的功能可以用switch語法實現。

Implementatoin

Part 1. System call - Sleep()

根據作業簡報描述，我先做定義Sleep system call編號的動作。

`Code`

code/userprog/syscall.h

```
...
#define SC_PrintInt 11
#define SC_Sleep    12\
...
void PrintInt(int number);  //my System Call
void Sleep(int number);
...
```

接著準備用於暫存sleep的register，這邊是組合語言很天書，跟我在寫編譯器作業的時候一樣，但這裡只需照著其他部分做增加就好。

`Code`

code/test/start.s

```
...
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
...
```

加入Sleep的exception case，定義當接收到SC_Sleep這個system call編號時所要做的行為。

`Code`

code/userprog/exception.cc

```
...
    case SC_PrintInt:
        val=kernel->machine->ReadRegister(4);
        cout << "Print integer:" <<val << endl;
        return;
    case SC_Sleep:
        val=kernel->machine->ReadRegister(4);
        cout << "Sleep Time " << val << "(ms) " << endl;
        kernel->alarm->WaitUntil(val);
        return;
/*  case SC_Exec:
        DEBUG(dbgAddr, "Exec\n");
        val = kernel->machine->ReadRegister(4);
        kernel->StringCopy(tmpStr, retVal, 1024);
        cout << "Exec: " << val << endl;
        val = kernel->Exec(val);
        kernel->machine->WriteRegister(2, val);
        return;
```

```
*/
...
```

接著按照作業投影片的提示，中斷常式在 NachOS 裡面應該是 `Alarm`，`Alarm`會每 X 次ticks產生一次interrupt，且`alarm.h`裡應該會有WaitUntil()這個function，根據提示可判斷我們應該要利用這個function 來實作 sleep，也就是計算每次時脈中斷觸發中斷常式的次數，當計數到達指定的 sleep 時間後，就把該 `Thread` 再次放回 Read Queue 等待執行。

`Code`

code/threads/alarm.h

```
...
class sleep_list {
    public:
        sleep_list():_current_interrupt(0) {};
        void put_to_sleep(Thread *t, int x);
    bool put_to_ready();
    bool IsEmpty();
    private:
        class sleep_thread {
            public:
                sleep_thread(Thread* t, int x):
                    sleeper(t), when(x) {};
                Thread* sleeper;
                int when;
        };

    int _current_interrupt;
    std::list<sleep_thread> _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
    sleep_list _sleeplist;
    void CallBack();         // called when the hardware
                // timer generates an interrupt
};
...
```

要做到前述的事情，可以先看一下`alarm.cc`，會很清出看到原本只有實作constructer及`CallBack()`。所以要做到的就是當很多的thread sleep時，應該要有一個 List 儲存這些sleep中的thread。而不同的thread喚醒時間也應該會不同，所以會要能儲存它們各自的喚醒時間。即必須寫一個class把thread跟喚醒時間合在一起，接著再放進List存起來。

Code

code/threads/alarm.cc

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

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

void
Alarm::WaitUntil(int x) {
    //Close interrupt
    IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);
    Thread* t = kernel->currentThread;
    // burst time
    int worktime = kernel->stats->userTicks - t->getStartTime();
    t->setBurstTime(t->getBurstTime() + worktime);
    t->setStartTime(kernel->stats->userTicks);
    cout << "Alarm::WaitUntil go sleep" << endl;
    _sleeplist.put_to_sleep(t, x);

    //Open interrupt
    kernel->interrupt->SetLevel(oldLevel);
}

bool sleep_list::IsEmpty() {
    return _threadlist.size() == 0;
}

void sleep_list::put_to_sleep(Thread*t, int x) {
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    _threadlist.push_back(sleep_thread(t, _current_interrupt + x));
    t->Sleep(false);
}

bool sleep_list::put_to_ready() {
    bool woken = false;

    _current_interrupt ++;

    for(std::list<sleep_thread>::iterator it = _threadlist.begin();
        it != _threadlist.end(); ) {
        if(_current_interrupt >= it->when) {
            woken = true;
```

```
//              cout << "sleep_list::put_to_ready Thread woken" << endl;
                kernel->scheduler->ReadyToRun(it->sleeper);
                it = _threadlist.erase(it);
            } else {
                it++;
            }
        }
        return woken;
}
```

Result:

我寫了三支簡單程式測試sleep，分別為：

code/test/sleep.c

```
#include "syscall.h"

main()
{
    PrintInt(11921);
    Sleep(1000000);
    return 0;
}
```

code/test/sleep1.c

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

    }
    return 0;
}
```

code/test/sleep2.c

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

    }
    return 0;
}
```

1. 基本測試

```
u14@ubuntu:~/r11921091_nachos2/nachos-4.0/code$ ./userprog/nachos -e ./test/sleep
Total threads number is 1
Thread ./test/sleep is executing.
Print integer:0
Sleep Time 1000000(ms)
Alarm::WaitUntil go sleep
Print integer:1
Sleep Time 1000000(ms)
Alarm::WaitUntil go sleep
Print integer:2
Sleep Time 1000000(ms)
Alarm::WaitUntil go sleep
return value:0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 300000100, idle 299999911, system 90, user 99
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. Call任意兩支程式

```
u14@ubuntu:~/r11921091_nachos2/nachos-4.0/code$ ./userprog/nachos -e ./test/sleep -e ./test/sleep1
Total threads number is 2
Thread ./test/sleep is executing.
Thread ./test/sleep1 is executing.
Print integer:0
Sleep Time 1000000(ms)
Alarm::WaitUntil go sleep
Print integer:0
Sleep Time 1000000(ms)
Alarm::WaitUntil go sleep
Print integer:1
Sleep Time 1000000(ms)
Alarm::WaitUntil go sleep
Print integer:1
Sleep Time 1000000(ms)
Alarm::WaitUntil go sleep
Print integer:2
Sleep Time 1000000(ms)
Alarm::WaitUntil go sleep
Print integer:2
Sleep Time 1000000(ms)
Alarm::WaitUntil go sleep
return value:0
Print integer:3
Sleep Time 1000000(ms)
Alarm::WaitUntil go sleep
Print integer:4
Sleep Time 1000000(ms)
Alarm::WaitUntil go sleep
return value:0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 500000200, idle 499999734, system 220, user 246
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
```

```
u14@ubuntu:~/r11921091_nachos2/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.
Print integer:0
Sleep Time 1000000(ms)
Alarm::WaitUntil go sleep
Sleep Time 500000(ms)
Alarm::WaitUntil go sleep
Print integer:618
Sleep Time 500000(ms)
Alarm::WaitUntil go sleep
Print integer:618
Sleep Time 500000(ms)
Alarm::WaitUntil go sleep
Print integer:1
Sleep Time 1000000(ms)
Alarm::WaitUntil go sleep
Print integer:618
Sleep Time 500000(ms)
Alarm::WaitUntil go sleep
Print integer:618
Sleep Time 500000(ms)
Alarm::WaitUntil go sleep
Print integer:2
Sleep Time 1000000(ms)
Alarm::WaitUntil go sleep
Print integer:618
Sleep Time 500000(ms)
Alarm::WaitUntil go sleep
Print integer:618
return value:0
return value:0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 300000100, idle 299999636, system 200, user 264
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
```

Part 2. CPU scheduling

這個部分我實做了First-Come-First-Service(FCFS)、Shortest-Job-First(SJF) 、 Priority，並實現這三個與原本Nachos的Round-Robin(RR)的切換。

第一步先去threads/kernel.h增加Initialize(...)這個method，去threads/kernel.cc新增scheduler因為我待會要新增有scheduler type的模式。

`Code`

code/threads/kernel.h

```
class ThreadedKernel {
  public:
    ...
    void Initialize(SchedulerType type);  //initialize the kernel--separated
    ...
};
```

code/threads/kernel.cc

```
...
void
ThreadedKernel::Initialize(SchedulerType type)
{
    stats = new Statistics();           // collect statistics
    interrupt = new Interrupt;          // start up interrupt handling
```

```
    scheduler = new Scheduler(type);    // initialize the ready queue
    alarm = new Alarm(randomSlice);     // start up time slicing

    // We didn't explicitly allocate the current thread we are running in.
    // But if it ever tries to give up the CPU, we better have a Thread
    // object to save its state.
    currentThread = new Thread("main");
    currentThread->setStatus(RUNNING);

    interrupt->Enable();
}
...
```

我在thread.cc增加SelfTest Code，定義執行程式名稱、thread優先權、burst time、thread body按照排程優先序執行程式，並將執行過的程式burst time-1並顯示。接下來的步驟就會從SelfTestCode慢慢倒回去，直到將要做的功能全部做出來。

`Code`

code/threads/thread.cc

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

void
Thread::SchedulingTest()
{
    const int thread_num = 5;
    char *name[thread_num] = {"A", "B", "C", "D", "E"};
    int thread_priority[thread_num] = {5, 4, 2, 3, 1};
    int thread_burst[thread_num] = {2, 7, 1, 6, 5};

    Thread *t;
    for (int i = 0; i < thread_num; i ++) {
        t = new Thread(name[i]);
        t->setPriority(thread_priority[i]);
        t->setBurstTime(thread_burst[i]);
        t->Fork((VoidFunctionPtr) threadBody, (void *)NULL);
    }
    kernel->currentThread->Yield();
}
```

在這裡我使用到了setBurstTime(int t), getBurstTime(), setStartTime(int t)等function，所以要記得在thread.h寫一下這些function，完成它們的功能。

`Code`

code/threads/thread.h

```
class Thread {
 private:
   ...
 public:
   ...
   void setBurstTime(int t)    {burstTime = t;}
   int getBurstTime()      {return burstTime;}
   void setStartTime(int t)    {startTime = t;}
   int getStartTime()      {return startTime;}
   void setPriority(int t) {execPriority = t;}
   int getPriority()       {return execPriority;}
   static void SchedulingTest();
 private:
   // some of the private data for this class is listed above

   // my add
   int burstTime;  // predicted burst time
   int startTime;  // the start time of the thread
   int execPriority;   // the execute priority of the thread
   ...
```

接著要在kernel.cc加入呼叫test code的程式碼。

`Code`

code/threads/kernel.cc

```
void
ThreadedKernel::SelfTest() {
   ...
   currentThread->SelfTest();   // test thread switching
   Thread::SchedulingTest();
                // test semaphore operation
   semaphore = new Semaphore("test", 0);
   ...
}
```

接著在main.cc增加下面程式碼，讓我們能在執行nachos時呼叫我們所需使用的排程方法。

`Code`

code/threads/main.cc

```
int
main(int argc, char **argv)
{
    ...
    DEBUG(dbgThread, "Entering main");

    SchedulerType type = RR;
    if(strcmp(argv[1], "FCFS") == 0) {
    type = FIFO;
    } else if (strcmp(argv[1], "SJF") == 0) {
    type = SJF;
    } else if (strcmp(argv[1], "PRIORITY") == 0) {
```

```
        type = Priority;
        } else if (strcmp(argv[1], "RR") == 0) {
        type = RR;
        }

        kernel = new KernelType(argc, argv);
        ...
}
```

接著我們來撰寫主要功能的部分，開始動`scheduler.h`和`scheduler.cc`。

先在scheduler.h增加Scheduler Type讓待會scheduler.cc可以呼叫到。

`Code`

code/threads/scheduler.h

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

class Scheduler {
  public:
    Scheduler();
    Scheduler(SchedulerType type);      // Initialize list of ready threads
    ...
    SchedulerType getSchedulerType() {return schedulerType;}
    void setSchedulerType(SchedulerType t) {schedulerType = t;}
  private:
    ...
```

接著就可以去scheduler.cc撰寫我們主要的排程方法，以及對應的方法。前面在看thread.cc時有發現scheduler原本應該是直接按readyList裡的順序執行thread，所以我的做法就是將thread按照規則放進`readyList`中，並在constructor決定要使用哪種排程，且宣告相應的 compare function。

`Code`

code/threads/scheduler.cc

```
...
#include "main.h"

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;
}
```

```
int FIFOCompare(Thread *a, Thread *b) {
    return 1;
}
//----------------------------------------------------------------------
// Scheduler::Scheduler
//  Initialize the list of ready but not running threads.
//  Initially, no ready threads.
//----------------------------------------------------------------------
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;
    case FIFO:
        readyList = new SortedList<Thread *>(FIFOCompare);
    }
    toBeDestroyed = NULL;
}

//----------------------------------------------------------------------
// Scheduler::~Scheduler
//  De-allocate the list of ready threads.
//----------------------------------------------------------------------

Scheduler::~Scheduler()
...
```

接著針對要執行RR 或 PRIORITY 這類Preemptive的排程，我們必須在 alarm.cc的Alarm::CallBack() 判斷要使用的是否為這兩種排程方法，如果是則要呼叫 interrupt->YieldOnReturn() 去查看是否有更需要優先的 process 要執行。

`Code`

code/threads/alarm.cc

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

    kernel->currentThread->setPriority(kernel->currentThread->getPriority() - 1);

```

```
    if (status == IdleMode && !woken && _sleeplist.IsEmpty()) {// is it time
to quit?
        if (!interrupt->AnyFutureInterrupts()) {
        timer->Disable();   // turn off the timer
    }
    } else {            // there's someone to preempt
    if(kernel->scheduler->getSchedulerType() == RR ||
        kernel->scheduler->getSchedulerType() == Priority ) {
//      interrupt->YieldOnReturn();
//      cout << "=== interrupt->YieldOnReturn ===" << endl;
        interrupt->YieldOnReturn();
    }
    }
}
```

接下來多數人可能都會以為這樣就完成了，但殊不知userkernel.h, userkernel.cc, netkernel.cc和netkernel.h要增加Initialize(SchedulerType type)的funtion到.h後綴的file，就是header檔，並在.cc的file，寫一下對應輸入SchedulerType的funtion，不然他們會call不到有schedulertype的狀況，導致make報error，我本來也沒特別注意到，因為自己寫得時候看code好像缺少這部分不太合理就直接加了，但幫同學debug的時候遇到有不只一位同學有這樣的問題。

`Code`

code/userprog/userkernel.h

```
...
class UserProgKernel : public ThreadedKernel {
  public:
    ...
    void Initialize();      // initialize the kernel
    void Initialize(SchedulerType type);
    ...
```

code/userprog/userkernel.cc

```
...
void
UserProgKernel::Initialize()
{
    Initialize(RR);
}
void
UserProgKernel::Initialize(SchedulerType type)
{
    ThreadedKernel::Initialize(type);   // init multithreading

    machine = new Machine(debugUserProg);
    fileSystem = new FileSystem();
#ifdef FILESYS
    synchDisk = new SynchDisk("New SynchDisk");
#endif // FILESYS
}
...
```

code/network/netkernel.h

```
...
class NetKernel : public UserProgKernel {
  public:
    ...
    void Initialize();    // initialize the kernel
    void Initialize(SchedulerType);
    ...
```

code/network/netkernel.cc

```
...
void
NetKernel::Initialize() {
    Initialize(RR);
}
void
NetKernel::Initialize(SchedulerType type)
{
    UserProgKernel::Initialize(type);   // init other kernel data structs

    postOfficeIn = new PostOfficeInput(10);
    postOfficeOut = new PostOfficeOutput(reliability, 10);
}
...
```

Result:

這裡會執行兩種不同的 test case，並將作業一test1.c和test2.c也拿來測試。

test case1

第一個 test case 的 SchedulingTest Code 也就是我自己的SelfTest method如下：

`Code`

code/threads/thread.cc

```
void
Thread::SchedulingTest()
{
    const int thread_num = 5;
    char *name[thread_num] = {"A", "B", "C", "D", "E"};
    int thread_priority[thread_num] = {5, 4, 2, 3, 1};
    int thread_burst[thread_num] = {2, 7, 1, 6, 5};

    Thread *t;
    for (int i = 0; i < thread_num; i ++) {
        t = new Thread(name[i]);
        t->setPriority(thread_priority[i]);
        t->setBurstTime(thread_burst[i]);
        t->Fork((VoidFunctionPtr) threadBody, (void *)NULL);
    }
    kernel->currentThread->Yield();
}
```

以下為執行FCFS、RR、SJF、PRIORITY四種CPU排程的輸出結果：

FCFS

按照助教的意思，只需要說明其中一個算法，所以我說明一下FCFS的輸出結果，FCFS算法只是根據作業的到達時間來排程，ready queue中最先出現的作業將第一個獲得 CPU使用權。task到達的時間越短，越早獲得 CPU使用權。如果第一個task的運作時間是所有task中最長的，則可能會產生starvation的問題。故如下圖所示，因為上面宣告`char *name[thread_num] = {"A", "B", "C", "D", "E"};`，然後我們是照順序擺進Ready queue，所以結果是按 A->B->C->D->E 的順序執行。

```
u14@ubuntu:~/r11921091_nachos2/nachos-4.0/code$ ./threads/nachos FCFS
*** thread 0 looped 0 times
*** thread 1 looped 0 times
*** thread 0 looped 1 times
*** thread 1 looped 1 times
*** thread 0 looped 2 times
*** thread 1 looped 2 times
*** thread 0 looped 3 times
*** thread 1 looped 3 times
*** thread 0 looped 4 times
*** thread 1 looped 4 times
A: remaining 1
A: remaining 0
B: remaining 6
B: remaining 5
B: remaining 4
B: remaining 3
B: remaining 2
B: remaining 1
B: remaining 0
C: remaining 0
D: remaining 5
D: remaining 4
D: remaining 3
D: remaining 2
D: remaining 1
D: remaining 0
E: remaining 4
E: remaining 3
E: remaining 2
E: remaining 1
E: remaining 0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

RR

```
u14@ubuntu:~/r11921091_nachos2/nachos-4.0/code$ ./threads/nachos RR
*** thread 0 looped 0 times
*** thread 1 looped 0 times
*** thread 0 looped 1 times
*** thread 1 looped 1 times
*** thread 0 looped 2 times
*** thread 1 looped 2 times
*** thread 0 looped 3 times
*** thread 1 looped 3 times
*** thread 1 looped 4 times
*** thread 0 looped 4 times
B: remaining 6
B: remaining 5
B: remaining 4
B: remaining 3
B: remaining 2
B: remaining 1
B: remaining 0
D: remaining 5
D: remaining 4
D: remaining 3
D: remaining 2
D: remaining 1
D: remaining 0
E: remaining 4
A: remaining 1
A: remaining 0
C: remaining 0
E: remaining 3
E: remaining 2
E: remaining 1
E: remaining 0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

SJF

```
u14@ubuntu:~/r11921091_nachos2/nachos-4.0/code$ ./threads/nachos SJF
*** thread 0 looped 0 times
*** thread 1 looped 0 times
*** thread 0 looped 1 times
*** thread 1 looped 1 times
*** thread 0 looped 2 times
*** thread 1 looped 2 times
*** thread 0 looped 3 times
*** thread 1 looped 3 times
*** thread 0 looped 4 times
*** thread 1 looped 4 times
C: remaining 0
A: remaining 1
A: remaining 0
E: remaining 4
E: remaining 3
E: remaining 2
E: remaining 1
E: remaining 0
D: remaining 5
D: remaining 4
D: remaining 3
D: remaining 2
D: remaining 1
D: remaining 0
B: remaining 6
B: remaining 5
B: remaining 4
B: remaining 3
B: remaining 2
B: remaining 1
B: remaining 0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

PRIORITY

```
u14@ubuntu:~/r11921091_nachos2/nachos-4.0/code$ ./threads/nachos PRIORITY
*** thread 0 looped 0 times
*** thread 1 looped 0 times
*** thread 0 looped 1 times
*** thread 1 looped 1 times
*** thread 0 looped 2 times
*** thread 1 looped 2 times
*** thread 0 looped 3 times
*** thread 1 looped 3 times
*** thread 1 looped 4 times
*** thread 0 looped 4 times
E: remaining 4
E: remaining 3
E: remaining 2
E: remaining 1
E: remaining 0
C: remaining 0
D: remaining 5
D: remaining 4
D: remaining 3
D: remaining 2
D: remaining 1
D: remaining 0
B: remaining 6
B: remaining 5
B: remaining 4
B: remaining 3
B: remaining 2
B: remaining 1
B: remaining 0
A: remaining 1
A: remaining 0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

test case2

第二個 test case 的 SchedulingTest Code 如下：

`Code`

code/threads/thread.cc

```
void
Thread::SchedulingTest()
{
    const int thread_num = 4;
    char *name[thread_num] = {"A", "B", "C", "D"};
    int thread_priority[thread_num] = {4, 3, 2, 1};
    int thread_burst[thread_num] = {10, 1, 1, 2};

    Thread *t;
    for (int i = 0; i < thread_num; i ++) {
        t = new Thread(name[i]);
        t->setPriority(thread_priority[i]);
        t->setBurstTime(thread_burst[i]);
        t->Fork((VoidFunctionPtr) threadBody, (void *)NULL);
    }
    kernel->currentThread->Yield();
}
```

以下為執行FCFS、RR、SJF、PRIORITY四種CPU排程的輸出結果：

FCFS

也有按照FCFS的算法，先到先得的排程執行。

```
u14@ubuntu:~/r11921091_nachos2/nachos-4.0/code$ ./threads/nachos FCFS
*** thread 0 looped 0 times
*** thread 1 looped 0 times
*** thread 0 looped 1 times
*** thread 1 looped 1 times
*** thread 0 looped 2 times
*** thread 1 looped 2 times
*** thread 0 looped 3 times
*** thread 1 looped 3 times
*** thread 0 looped 4 times
*** thread 1 looped 4 times
A: remaining 9
A: remaining 8
A: remaining 7
A: remaining 6
A: remaining 5
A: remaining 4
A: remaining 3
A: remaining 2
A: remaining 1
A: remaining 0
B: remaining 0
C: remaining 0
D: remaining 1
D: remaining 0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

RR

```
u14@ubuntu:~/r11921091_nachos2/nachos-4.0/code$ ./threads/nachos RR
*** thread 0 looped 0 times
*** thread 1 looped 0 times
*** thread 0 looped 1 times
*** thread 1 looped 1 times
*** thread 0 looped 2 times
*** thread 1 looped 2 times
*** thread 0 looped 3 times
*** thread 1 looped 3 times
*** thread 1 looped 4 times
*** thread 0 looped 4 times
B: remaining 0
C: remaining 0
D: remaining 1
D: remaining 0
A: remaining 9
A: remaining 8
A: remaining 7
A: remaining 6
A: remaining 5
A: remaining 4
A: remaining 3
A: remaining 2
A: remaining 1
A: remaining 0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

SJF

```
u14@ubuntu:~/r11921091_nachos2/nachos-4.0/code$ ./threads/nachos SJF
*** thread 0 looped 0 times
*** thread 1 looped 0 times
*** thread 0 looped 1 times
*** thread 1 looped 1 times
*** thread 0 looped 2 times
*** thread 1 looped 2 times
*** thread 0 looped 3 times
*** thread 1 looped 3 times
*** thread 0 looped 4 times
*** thread 1 looped 4 times
B: remaining 0
C: remaining 0
D: remaining 1
D: remaining 0
A: remaining 9
A: remaining 8
A: remaining 7
A: remaining 6
A: remaining 5
A: remaining 4
A: remaining 3
A: remaining 2
A: remaining 1
A: remaining 0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

PRIORITY

```
u14@ubuntu:~/r11921091_nachos2/nachos-4.0/code$ ./threads/nachos PRIORITY
*** thread 0 looped 0 times
*** thread 1 looped 0 times
*** thread 0 looped 1 times
*** thread 1 looped 1 times
*** thread 0 looped 2 times
*** thread 1 looped 2 times
*** thread 0 looped 3 times
*** thread 1 looped 3 times
*** thread 1 looped 4 times
*** thread 0 looped 4 times
D: remaining 1
D: remaining 0
C: remaining 0
B: remaining 0
A: remaining 9
A: remaining 8
A: remaining 7
A: remaining 6
A: remaining 5
A: remaining 4
A: remaining 3
A: remaining 2
A: remaining 1
A: remaining 0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

同時執行HW1 test1.c和test2.c

先貼一下test1.c和test2.c的code

`Code`

test1.c

```c
#include "syscall.h"
main()
    {
        int n;
        for (n=9;n>5;n--)
            PrintInt(n);
    }
```

test2.c

```c
#include "syscall.h"

main()
        {
                int     n;
                for (n=20;n<=25;n++)
                        PrintInt(n);
        }
```

以下為執行FCFS、RR、SJF、PRIORITY四種CPU排程的輸出結果：

FCFS

完全符合FCFS的排程算法，先來的test1先執行完，再執行test2。

```
u14@ubuntu:~/r11921091_nachos2/nachos-4.0/code$ ./userprog/nachos FCFS -e ./test/test1 -e ./test/
test2
Total threads number is 2
Thread ./test/test1 is executing.
Thread ./test/test2 is executing.
Print integer:9
Print integer:8
Print integer:7
Print integer:6
return value:0
Print integer:20
Print integer:21
Print integer:22
Print integer:23
Print integer:24
Print integer:25
return value:0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

RR

就跟HW1時一樣的輸出結果。

```
u14@ubuntu:~/r11921091_nachos2/nachos-4.0/code$ ./userprog/nachos RR -e ./test/test1 -e ./test/te
st2
Total threads number is 2
Thread ./test/test1 is executing.
Thread ./test/test2 is executing.
Print integer:9
Print integer:8
Print integer:7
Print integer:20
Print integer:21
Print integer:22
Print integer:23
Print integer:24
Print integer:6
return value:0
Print integer:25
return value:0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

SJF

test1較短，所以執行完再執行test2。

```
u14@ubuntu:~/r11921091_nachos2/nachos-4.0/code$ ./userprog/nachos SJF -e ./test/test1 -e ./test/t
est2
Total threads number is 2
Thread ./test/test1 is executing.
Thread ./test/test2 is executing.
Print integer:9
Print integer:8
Print integer:7
Print integer:6
return value:0
Print integer:20
Print integer:21
Print integer:22
Print integer:23
Print integer:24
Print integer:25
return value:0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

PRIORITY

沒給定的priority，所以它就沒有priority可以採用只能像Round-Robin一樣跑。

```
u14@ubuntu:~/r11921091_nachos2/nachos-4.0/code$ ./userprog/nachos PRIORITY -e ./test/test1 -e ./test/test2
Total threads number is 2
Thread ./test/test1 is executing.
Thread ./test/test2 is executing.
Print integer:9
Print integer:8
Print integer:7
Print integer:20
Print integer:21
Print integer:22
Print integer:23
Print integer:24
Print integer:6
return value:0
Print integer:25
return value:0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

很明顯程式是有效的，能夠成功且換四種排程算法。

The difficulties I encountered

幫三位同學debug遇到一些問題，底下為我還記得的四項問題：

1. 同學寫sleep 測試程式sleep.c時，寫成如下格式：

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

       }
       return 0;
   }
   ```

   研判應該是vm使用ubuntu14，而ubuntu14預設gcc std應該為c90，而上面這種在for迴圈宣告int i=0的用法是c99才支援，所以就error囉！

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

       }
       return 0;
   }
   ```

   改成這樣就沒有error了！

2. userkernel.cc的 UserProgKernel::Initialize(SchedulerType type)內的 ThreadedKernel::Initialize(type);沒有帶入type
3. netkernel.cc 的 NetKernel::Initialize(SchedulerType type)內的UserProgKernel::Initialize(type)沒有帶入type
4. 寫code時誤刪thread.cc中的SelfTest()所以就error了！

Reference

[Nachos hw2 assignments] https://cool.ntu.edu.tw/courses/21578/files/2923205?wrap=1
[Nachos Documentation] https://homes.cs.washington.edu/~tom/nachos/