電機所碩二
張光耀
R04921005

Operating System

Project 2 - System Call and CPU Scheduling

First Part. System Call

Observation:

在實作CPU排程前必須先實作sleep函數，首先在trace code的過程中，發現其實system call的實作大同小異，例如之前在test檔案有看過 PrintInt 的函數，將數字印出的system call，故在前半部分可以模仿 PrintInt( ) 去宣告一個新的system call。在此我將我實作的Sleep函數命名為 mySleep( )。

Implementation:

```
#define SC_Halt       0
#define SC_Exit       1
#define SC_Exec       2
#define SC_Join       3
#define SC_Create     4
#define SC_Open       5
#define SC_Read       6
#define SC_Write      7
#define SC_Close      8
#define SC_ThreadFork     9
#define SC_ThreadYield   10
#define SC_PrintInt 11
#define SC_mySleep     12
```

```
void PrintInt(int number);   //my System Call
void mySleep(int number);
```

上圖為 syscall.h 這個標頭檔，這裡為宣告system call 的地方，可以模仿 printInt 這個函數實作 mysleep( ) 的宣告。

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

        .globl  mySleep
        .ent    mySleep
mySleep:
        addiu   $2,$0,SC_mySleep
        syscall
        j   $31
        .end    mySleep
```

上圖為 start.s，接下來是去此 assembly code 增加 mySleep( ) 的宣告， assembly language 是用來協助對nachos 的 kernel 呼叫system call ，每個 system call 皆有對應的數字 (定義在 syscall.h)，而system call 的實作code置於 register 2 內，input 的 argument 放置於 register 4 內。

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

    switch (which) {
    case SyscallException:
        switch(type) {
        case SC_Halt:
            DEBUG(dbgAddr, "Shutdown, initiated by user program.\n");
            kernel->interrupt->Halt();
            break;
        case SC_PrintInt:
            val=kernel->machine->ReadRegister(4);
            cout << "Print integer:" << val << endl;
            return;
        case SC_mySleep:
            val=kernel->machine->ReadRegister(4);
            cout << "Go to sleep ! Sleep time: " << val << "(ms) " << endl;
            kernel->alarm->WaitUntil(val);
            return;
```

上圖為 userprog/exception.cc ， 此處為定義呼叫到對應的 system call 的 exception handle 方式，先去 register 4 讀取 input argument，再去對kernel 呼叫 WaitUntil ( )，alarm 為作業系統的鬧鐘，每過一小段時間就會呼叫 CallBack( ) ，可以把他當作一個計數器，每當alarm響一次大約是一毫秒 (ms)，所以可以把他當作一個累計器，當使用者呼叫 mySleep ( ) 這個函數，就開始把 thread 丟進一個地方 (waiting list) 睡眠等待，並把時間記錄下來， 每次 alarm 響就去累計已經睡眠的時間，並去檢查是否有 thread 的等待時間已經超過原本輸入的argument，若有就將其喚醒。故我們必須去修改 WaitUntil 函數。

```cpp
class PlaceForSleep {
public:
    PlaceForSleep():now_interrupt(0) {};
    void PutToPlace(int input_time, Thread *t);
    bool Calling();
    bool IsEmpty();
private:
    class Place {
    public:
        Place(int input_time, Thread* t):
        sleeper(t), WhenForWake(input_time) {};
        Thread* sleeper;
        int WhenForWake;
    };

    int now_interrupt;
    std::list<Place> _Places;
};

// 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. (periodic calling)
    ~Alarm() { delete timer; }

    void WaitUntil(int input_time); // stop execution until time > now + input time

private:
    Timer *timer;         // timer device
    PlaceForSleep _PlaceForSleep;
    void CallBack();          // called When the hardware
    // timer generates an interrupt
};
```

上圖為 alarm.h 標頭檔，根據上述，若要實作 sleep 函數的話，必須創造一個地方把要設定為睡眠的thread 丟進去，並記錄它何時要被喚醒，所以在這裡創造一個 PlaceForSleep 的 class ，並將其定義在 class Alarm 的 private ，PlaceForSleep 的 class 內開一個 list 去記錄有哪些 thread 正在睡眠，同時記錄何時該喚醒 (when)，並定義一系列的函數去support sleep的動作： PutToPlace(int, Thread*) —> 將thread 放進waiting list 、Calling() —>檢查waiting list 有沒有thread 該喚醒、IsEmpty() —>檢查waiting list 是不是空的，宣告好之後就要進入實作的部分。

```
void
Alarm::WaitUntil(int input_time) {
    IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);
    Thread* t = kernel->currentThread;

    // burst time
    int worktime = kernel->stats->userTicks - t->getStime();
    t->setPreBtime(t->getPreBtime() + worktime);
    t->setStime(kernel->stats->userTicks);
    cout << "Alarm::WaitUntil go sleep" << endl;
    _PlaceForSleep.PutToPlace(input_time, t);
    kernel->interrupt->SetLevel(oldLevel);
}
```

上圖是alarm.cc的WaitUntil實作部分，下部分的burst time 部分為 CPU scheduling 在後面會在介紹。這裡主要是將 thread state 記錄起來，並將這個 thread 放置到 sleep waiting list 。

```
void PlaceForSleep::PutToPlace(int input_time, Thread*t) {
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    _Places.push_back(Place(now_interrupt + input_time, t));
    t->Sleep(false); // set the thread to sleep
}
```

上圖是alarm.cc的PutToPlace實作部分，將current thread 串在 sleep waiting list (_Place) 後面，並將current thread 設置為睡眠狀態，並記錄何時該甦醒(now_interrupt+input_time)

```
void
Alarm::CallBack()
{
    Interrupt *interrupt = kernel->interrupt;
    MachineStatus status = interrupt->getStatus();
    bool IsWoken = _PlaceForSleep.Calling();

    kernel->currentThread->setPri(kernel->currentThread->getPri() - 1);  //change the Pri

    if (status == IdleMode && !IsWoken && _PlaceForSleep.IsEmpty()) {// is it time to quit?
        if (!interrupt->AnyFutureInterrupts()) {
        timer->Disable();   // turn off the timer
    }
    } else {             // for preemptive design
    if(kernel->scheduler->getSchedulerType() == RR ||
        kernel->scheduler->getSchedulerType() == Priority ) {
        interrupt->YieldOnReturn();
        }
    }
}
```

上圖是alarm.cc的CallBack( )實作部分，有部分 priority 與 preemptive design 為 CPU scheduling 在後面會在介紹。 可以看到每次 CallBack 都會利用 Calling 這個函數去檢查有沒有thread 要被喚醒。

```
bool PlaceForSleep::Calling() {
    bool IsWoken = false;

    now_interrupt = now_interrupt + 1;

    for(std::list<Place>::iterator it = _Places.begin(); it != _Places.end(); ) {
        if(now_interrupt >= it->WhenForWake) {
            IsWoken = true;
            cout << "PlaceForSleep::Calling Thread Woken" << endl;
            kernel->scheduler->ReadyToRun(it->sleeper);
            it = _Places.erase(it);
        } else {
            it++;
        }
    }
    return IsWoken;
}
```

```
bool PlaceForSleep::IsEmpty() {
    return _Places.size() == 0;
}
```

上圖是alarm.cc的Calling( ) 與 IsEmpty( ) 實作部分，在 Calling( ) 中，因為alarm 每過一小段時間都會執行，所以剛好能來當累計器，利用 now_interrupt 每次都加一來記錄已經響過幾次了，並把這個紀錄數字拿來當時鐘的標準，每次都去檢查 now_interrupt >=it->WhenForWake ，若有人該甦醒就把那個thread 設ReadyToRun。在 IsEmpty( ) 中，就是單純是看list 是不是空的，利用size of list 去檢查。

```
#include "syscall.h"
main()
    {
        int n;
        for (n=9;n>5;n--){
            mySleep(10000000);
            PrintInt(n);
        }
    }
```

上述已完成 mySleep 的實作部分，上圖為模仿 test1.cc 的測試code，我將其命名為sleeptest1，注意要將其新增於Makefile 上才能產生它的 object file。測試結果圖呈現於下一頁。

下圖為 mySleep( ) system call 的測試結果，每印一個數字都會讓這個thread sleep 一段時間，時間到了就會把它叫起來繼續執行，並在正確的地方開始執行，所以這個結果符合預期。

Second Part. CPU scheduling

在 CPU scheduling 這個部分，我先實作了 Priority ，而 Bonus 部分我實作了FCFS (First Come First Serve)、 SJF (Shortest Job First)。

Priority scheduling:

每一個thread都有它的優先順序 (利用一個整數來記錄優先權)。CPU的使用權分配給具有最高優先權的thread，若具有相同優先順序的行程就按照FCFS來分配CPU。

Bonus:

FCFS scheduling:

目前的thread結束時，選擇在等待 list (Ready list) 中等待最久的thread，也就是最先來的thread先處理。

SJF scheduling:

選擇執行時間最短的process先執行，但此種方法必須事先知道或是估算process所需的執行時間。

```
class Thread {
  private:
    // NOTE: DO NOT CHANGE the order of these first two members.
    // THEY MUST be in this position for SWITCH to work.
    int *stackTop;             // the current stack pointer
    void *machineState[MachineStateSize];  // all registers except for stackTop

  public:
    Thread(char* debugName);          // initialize a Thread
    ~Thread();                // deallocate a Thread
                      // NOTE -- thread being deleted
                      // must not be running when delete
                      // is called
    // basic thread operations
    void Fork(VoidFunctionPtr func, void *arg);
                      // Make thread run (*func)(arg)
    void Yield();         // Relinquish the CPU if any
                  // other thread is runnable
    void Sleep(bool finishing); // Put the thread to sleep and
                  // relinquish the processor
    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

    void setPri(int t)  {execPri = t;}
    int getPri()          {return execPri;}
    void setPreBtime(int t) {PreBtime = t;}
    int getPreBtime()         {return PreBtime;}
    void setStime(int t)      {Stime = t;}
    int getStime()        {return Stime;}
    static void TestForScheduling();
 private:
   // some of the private data for this class is listed above

   int PreBtime;
   int Stime;
   int execPri;

   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;

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

上圖為 thread.h 的class thread部分，為了要implement 上述三個方法，我們必須在 class 的 private member 有 predicted burst time (PreBtime) 、 Start time (Stime)、Priority (exePri)，才能進行thread間的比較進而完成排程。在函式方面則是需要有設定、取得 start time、burst time、priority 等等的function。

```
enum SchedulerType {
        FIFO,
        RR,
        Priority,
        SJF
};
```

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

上圖為 schedule.h 標頭檔，為了在 testing part 能夠方便切換各個排程演算法，所以必須額外定義各個不同的演算法代號（schedulerType），並且設計一設定scheduler的function達到切換的目的 (setSchedulerType)。

```
int PriCmp(Thread *a, Thread *b) {
    if(a->getPri() == b->getPri())
        return 0;
    else if(a->getPri() > b->getPri())
        return 1;
    else
        return -1;
}
int SJFCmp(Thread *a, Thread *b) {
    if(a->getPreBtime() == b->getPreBtime())
        return 0;
    else if(a->getPreBtime() > b->getPreBtime())
        return 1;
    else
        return -1;
}
int FIFOCmp(Thread *a, Thread *b) {
    return 1;
}
```

上圖為 schedule.cc 的實作部分，因為scheduler的class 中是用一個STL sorted list 來貯存正在執行的 thread，這裡採用STL的特性，直接去更改 compare 的規則，就可以達到排程的效果。針對 Priority 就利用每個thread中所記錄的 priority member 進行比較(這裡

採數字較小為較優先)。針對 SJF 就利用每個thread中所記錄的預測burst time，讓較小的優先。 FCFS 則不需特別做排序，先進入的就在前面。

```
Scheduler::Scheduler(SchedulerType type)
{
    schedulerType = type;
    switch(schedulerType) {
    case Priority:
        readyList = new SortedList<Thread *>(PriCmp);
        break;
    case RR:
        readyList = new List<Thread *>;
        break;
    case FIFO:
        readyList = new SortedList<Thread *>(FIFOCmp);
        break;
    case SJF:
        readyList = new SortedList<Thread *>(SJFCmp);

    }
    toBeDestroyed = NULL;
}
```

上圖為 schedule.cc 的實作部分，根據所選擇的scheduling algorithm 去排序 waiting list，利用上述的各種comparison 法則套用到 STL sorted list 當中。

```
void
Alarm::CallBack()
{
    Interrupt *interrupt = kernel->interrupt;
    MachineStatus status = interrupt->getStatus();
    bool IsWoken = _PlaceForSleep.Calling();

    kernel->currentThread->setPri(kernel->currentThread->getPri() - 1);  //change the Pri

    if (status == IdleMode && !IsWoken && _PlaceForSleep.IsEmpty()) {// is it time to quit?
        if (!interrupt->AnyFutureInterrupts()) {
        timer->Disable();   // turn off the timer
    }
    } else {            // for preemptive design
    if(kernel->scheduler->getSchedulerType() == RR ||
        kernel->scheduler->getSchedulerType() == Priority ) {
        interrupt->YieldOnReturn();
        }
    }
}
```

上圖為alarm.cc 的實作部分，針對Priority排程，若要有 preemptive 的設計，故每次在callback時，就必須呼叫 YieldOnReturn 去看是否有更優先的thread 需要去執行。

Test code:

```
//  Choose the sheduler type
    SchedulerType type = RR;
    if(strcmp(argv[1], "FirstComeFirstServe") == 0)      type = FIFO;
    else if (strcmp(argv[1], "ShortestJobFirst") == 0)  type = SJF;
    else if (strcmp(argv[1], "RoundRobin") == 0)        type = RR;
    else if (strcmp(argv[1], "Priority") == 0)          type = Priority;

//
```

上圖為thread/main.cc，在 testing 的部分，先在main.cc 加入一個scheduling type 判別，讓使用者可以簡單地利用 input argument 切換想 testing 的 algorithm。

Switching interface 使用：

若想要測試 SJF algorithm 只要在 code/threads/下執行

./nachos ShortestJobFirst 即可開始測試

若想要測試 FCFS algorithm 只要在 code/threads/下執行

./nachos FirstComeFirstServe 即可開始測試

若想要測試 Priority algorithm 只要在 code/threads/下執行

./nachos Priority 即可開始測試

```
void
threadTestBody() {
    Thread *thread = kernel->currentThread;
    while (thread->getPreBtime() > 0) {
        thread->setPreBtime(thread->getPreBtime() - 1);
        kernel->interrupt->OneTick();
        printf("%s: task remaining %d\n", kernel->currentThread->getName(), kernel->currentThread->
            getPreBtime());
    }
    printf("%s: Finish", kernel->currentThread->getName());
}
void
Thread::TestForScheduling()
{
    int thread_num = 5;
    char *name[thread_num] = {"Process 1", "Process 2", "Procsee 3", "Process 4", "Process 5"};
    int thread_Pri[thread_num] = {1, 2, 5, 3, 6};
    int thread_burst[thread_num] = {2, 8, 6, 7, 4};

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

Test code 的設計如上圖，定義一個 TestForScheduling 的 function ，設定好 thread的數目、各個thread的名稱、各個thread的 priority (越小越優先)、各個thread的 CPU burst time ，再經由一個For loop 將上述的數值assign 到每個thread，利用fork產生新的thread

並傳向threadTestBody這個function，在threadTestBody即是模擬一個 thread 在使用CPU的情形，使用結束會印出Finish。

```
void
ThreadedKernel::SelfTest() {
   Semaphore *semaphore;
   SynchList<int> *synchList;

   LibSelfTest();        // test library routines

   currentThread->SelfTest();   // test thread switching
   Thread::TestForScheduling();
                 // test semaphore operation
   semaphore = new Semaphore("test", 0);
   semaphore->SelfTest();
   delete semaphore;

                 // test locks, condition variables
                 // using synchronized lists
   synchList = new SynchList<int>;
   synchList->SelfTest(9);
   delete synchList;

   ElevatorSelfTest();
}
```

上圖為threadkernel.cc，完成TestForScheduling後將其加入至SelfTest( )中，即可測試。

Result — Case1

Setting:

int thread_num = 5;

*char *name[thread_num] = {"Process 1", "Process 2", "Procsee 3", "Process 4", "Process 5"};*

int thread_Pri[thread_num] = {1, 2, 5, 3, 6};

int thread_burst[thread_num] = {2, 8, 6, 7, 4};

Result — Case2

Setting:

int thread_num = 3;

*char *name[thread_num] = {"Process 1", "Process 2", "Process 3"};*

int thread_Pri[thread_num] = {1, 5, 8};

int thread_burst[thread_num] = {2, 7, 4};

Case1:

Priority algorithm:

```
morris@ubuntu:~/Downloads/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
Process 1: task remaining 1
Process 1: task remaining 0
Process 1: Finish
Process 2: task remaining 7
Process 2: task remaining 6
Process 2: task remaining 5
Process 2: task remaining 4
Process 2: task remaining 3
Process 2: task remaining 2
Process 2: task remaining 1
Process 2: task remaining 0
Process 2: Finish
Process 4: task remaining 6
Process 4: task remaining 5
Process 4: task remaining 4
Process 4: task remaining 3
Process 4: task remaining 2
Process 4: task remaining 1
Process 4: task remaining 0
Process 4: Finish
Process 3: task remaining 5
Process 3: task remaining 4
Process 3: task remaining 3
Process 3: task remaining 2
Process 3: task remaining 1
Process 3: task remaining 0
Process 3: Finish
Process 5: task remaining 3
Process 5: task remaining 2
Process 5: task remaining 1
Process 5: task remaining 0
Process 5: Finish
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 2900, idle 130, system 2770, 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
```

結果符合預期：執行順序為 Process 1 (Priority = 1) —> Process 2 (Priority = 2) —> Process 4 (Priority = 3) —> Process 3 (Priority = 5) —> Process 5 (Priority = 6)

FirstComeFirstServe algorithm:

```
morris@ubuntu:~/Downloads/nachos-4.0/code/threads$ ./nachos FirstComeFirstServe
*** 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
Process 1: task remaining 1
Process 1: task remaining 0
Process 1: Finish
Process 2: task remaining 7
Process 2: task remaining 6
Process 2: task remaining 5
Process 2: task remaining 4
Process 2: task remaining 3
Process 2: task remaining 2
Process 2: task remaining 1
Process 2: task remaining 0
Process 2: Finish
Process 3: task remaining 5
Process 3: task remaining 4
Process 3: task remaining 3
Process 3: task remaining 2
Process 3: task remaining 1
Process 3: task remaining 0
Process 3: Finish
Process 4: task remaining 6
Process 4: task remaining 5
Process 4: task remaining 4
Process 4: task remaining 3
Process 4: task remaining 2
Process 4: task remaining 1
Process 4: task remaining 0
Process 4: Finish
Process 5: task remaining 3
Process 5: task remaining 2
Process 5: task remaining 1
Process 5: task remaining 0
Process 5: Finish
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 2800, idle 190, system 2610, 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
```

結果符合預期：執行順序為 Process 1 —> Process 2 —> Process 3 —> Process 4—> Process 5

ShortestJobFirst algorithm:

```
morris@ubuntu:~/Downloads/nachos-4.0/code/threads$ ./nachos ShortestJobFirst
*** 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
Process 1: task remaining 1
Process 1: task remaining 0
Process 1: Finish
Process 5: task remaining 3
Process 5: task remaining 2
Process 5: task remaining 1
Process 5: task remaining 0
Process 5: Finish
Process 3: task remaining 5
Process 3: task remaining 4
Process 3: task remaining 3
Process 3: task remaining 2
Process 3: task remaining 1
Process 3: task remaining 0
Process 3: Finish
Process 4: task remaining 6
Process 4: task remaining 5
Process 4: task remaining 4
Process 4: task remaining 3
Process 4: task remaining 2
Process 4: task remaining 1
Process 4: task remaining 0
Process 4: Finish
Process 2: task remaining 7
Process 2: task remaining 6
Process 2: task remaining 5
Process 2: task remaining 4
Process 2: task remaining 3
Process 2: task remaining 2
Process 2: task remaining 1
Process 2: task remaining 0
Process 2: Finish
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 2800, idle 190, system 2610, 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
```

結果符合預期：執行順序為 Process 1 (Burst time = 2) —> Process 5 (Burst time = 4) —> Process 3 (Burst time = 6) —> Process 4 (Burst time = 7) —> Process 2 (Burst time = 8)

Case2: Priority algorithm

```
morris@ubuntu:~/Downloads/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
Process 1: task remaining 1
Process 1: task remaining 0
Process 1: Finish
Process 2: task remaining 6
Process 2: task remaining 5
Process 2: task remaining 4
Process 2: task remaining 3
Process 2: task remaining 2
Process 2: task remaining 1
Process 2: task remaining 0
Process 2: Finish
Process 3: task remaining 3
Process 3: task remaining 2
Process 3: task remaining 1
Process 3: task remaining 0
Process 3: Finish
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
```

結果符合預期：執行順序為 Process 1 (Priority = 1) —> Process 2 (Priority = 5) —> Process 3 (Priority = 8)

FirstComeFirstServe algorithm:

```
morris@ubuntu:~/Downloads/nachos-4.0/code/threads$ ./nachos FirstComeFirstServe
*** thread 0 looped 3 times
*** thread 1 looped 3 times
*** thread 0 looped 4 times
*** thread 1 looped 4 times
Process 1: task remaining 1
Process 1: task remaining 0
Process 1: Finish
Process 2: task remaining 6
Process 2: task remaining 5
Process 2: task remaining 4
Process 2: task remaining 3
Process 2: task remaining 2
Process 2: task remaining 1
Process 2: task remaining 0
Process 2: Finish
Process 3: task remaining 3
Process 3: task remaining 2
Process 3: task remaining 1
Process 3: task remaining 0
Process 3: Finish
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 2600, idle 170, system 2430, 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
```

結果符合預期：執行順序為 Process 1—> Process 2 —> Process 3

ShortestJobFirst algorithm:

```
morris@ubuntu:~/Downloads/nachos-4.0/code/threads$ ./nachos ShortestJobFirst
*** thread 0 looped 3 times
*** thread 1 looped 3 times
*** thread 0 looped 4 times
*** thread 1 looped 4 times
Process 1: task remaining 1
Process 1: task remaining 0
Process 1: Finish
Process 3: task remaining 3
Process 3: task remaining 2
Process 3: task remaining 1
Process 3: task remaining 0
Process 3: Finish
Process 2: task remaining 6
Process 2: task remaining 5
Process 2: task remaining 4
Process 2: task remaining 3
Process 2: task remaining 2
Process 2: task remaining 1
Process 2: task remaining 0
Process 2: Finish
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 2600, idle 170, system 2430, 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
```

結果符合預期：執行順序為 Process 1 (Burst time = 2) —> Process 3 (Burst time = 4) —> Process 2 (Burst time = 7)

Discussion:

System call: 從實作mySleep 這個system call，了解nachos實際是怎麼去呼叫一個system call，需通過 assembly code 等等的協助才能有效率地呼叫，其中值得注意的是，利用每一次alarm當累計器，其時間精準度應該不會相當精確。

CPU scheduling: 實作的三個演算法皆正確，在實作的過程中，FCFS算是最容易實現的，幾乎不用做什麼改變，但CPU的使用效率可能就沒有這麼好，SJF 是最佳化的演算法，但是其實在真正運行的時候，很難去預測burst time 到底是多少，無法像test code內直接去定義數值，Priority 則是可能會發生 starvation 的情況，且真正在定義其優先權時亦須考慮很多其他的因素。