HW2 System Call & CPU Scheduling

R08942087 吳彬睿

1. Motivation

(1) System call: Sleep

現在我們的 nachos 裡面並沒有 sleep 的 system call，因此我們必須實作一個 sleep 的 system call 來讓我們的 threads 可以休息。

(2) CPU Scheduling

原本 nachos 裡面的 scheduling 方式是 RR 的，讓不同 process 輪流執行，現在我們要加入其他的 scheduling 方法: FCFS, SJS, Priority。

同時我們要 implement 自我測試的 code，這樣可以方便且快速測是我們的 scheduling 是否是正確的。

2. Implementation

(1) System Call: Sleep

code/userprog/syscall.h

```
#define SC_PrintInt 11
#define SC_Sleep 12

#ifndef IN_ASM
```

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

void Sleep(int number);

```

幫 sleep systemcall 編號(12)，並加入 sleep 的程式碼定義。

code/test/start.s

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

```

把 sleep 變成系統的一個 API，讓 c++可以以 library 的方式 call 他。

user/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;
```

在 trap 到 os 之中後，會觸發 exception，接著這邊就要觸發 wait 指令。

code/threads/alarm.h

```
#include <list>
#include "thread.h"
class sleepList {
    public:
        sleepList():_current_interrupt(0) {};
        void PutToSleep(Thread *t, int x);
    bool PutToReady();
    bool IsEmpty();
    private:
        class sleepThread {
            public:
                sleepThread(Thread *t, int x):
                    sleeper(t), when(x) {};
                Thread* sleeper;
                int when;
        };

    int _current_interrupt;
    std::list<sleepThread> _threadlist;
};

```

code/threads/alarm.cc

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

    bool woken = _sleepList.PutToReady();
    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
        interrupt->YieldOnReturn();
    }
}

void
Alarm::WaitUntil(int x){
    //關中斷
    IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);
    Thread* t = kernel->currentThread;
    //cout << "Alarm::WaitUntil go sleep" << endl;
    _sleepList.PutToSleep(t, x);
    //開中斷
    kernel->interrupt->SetLevel(oldLevel);
}

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

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

bool
sleepList::PutToReady() {
    bool woken=false;
    _current_interrupt ++;
    for(std::list<sleepThread>::iterator it = _threadlist.begin(); it!=_threadlist.end(); ){
        if(_current_interrupt >= it->when){
            woken=true;
            //cout << "sleepList::PutToReady Thread woken" << endl;
            kernel->scheduler->ReadyToRun(it->sleeper);
            it = _threadlist.erase(it);
        } else {
            it++;
        }
    }
    return woken;
}
```

在 os 中會有個 clock，每 tick 一次就會觸發一次中斷，並去執行該中斷該執行的 code，在 nachos 裡面執行的是 alarm，而 alarm 裡面原本就有定義好 WaitUntil，因此我們就要利用這個 function 來實作 sleep，我們實作一個 sleep list 來儲存每個 thread sleep 的時間。然後當觸發 time 的中段的時候，我們就把計數器往上加 1，然後判斷有沒有一些 thread 可以回去 ready queue 的。

code/test/sleep1.cc

```
#include "syscall.h"

main() {
    int i;
    for(i=0; i<10; i++){
        Sleep(500000);
        PrintInt(1);
    }
    return 0;
}
```

code/test/sleep2.cc

```
#include "syscall.h"

main() {
    int i;
    for(i=0; i<5; i++){
        Sleep(1000000);
        PrintInt(2);
    }
    return 0;
}
```

Sleep1 的 delay 是 500000(ms)，然後會顯示 1；而 Sleep2 的 delay 是 1000000(ms)，然後會顯示 2。

code/test/Makefile

```

all: halt shell matmult sort test1 test2 sleep1 sleep2

sleep1: sleep1.o start.o
	$(LD) $(LDFLAGS) start.o sleep1.o -o sleep1.coff
	../bin/coff2noff sleep1.coff sleep1

sleep2: sleep2.o start.o
	$(LD) $(DLFLAGS) start.o sleep2.o -o sleep2.coff
	../bin/coff2noff sleep2.coff sleep2

```

(2) CPU Scheduling

- **Self testing:**

code/thread/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=4;
    char * name[thread_num] = {"A", "B", "C", "D"};
    int thread_priority[thread_num] = {1, 9, 8, 7};
    int thread_burst[thread_num] = {8, 7, 4, 5};

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

code/thread/thread.h

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

```
    int burstTime;
    int startTime;
    int execPriority;
```

code/thread/kernel.cc

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

   LibSelfTest();          // test library routines

   currentThread->SelfTest();    // test thread switching
   Thread::SchedulingTest();

                  // 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();
```

code/thread/main.cc

```cpp

    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);
    kernel->Initialize(type);

```

code/thread/scheduler.h

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

```

code/machine/machine.h

```
const unsigned int PageSize = 128;       // set the page size equal to
                        // the disk sector size, for simplicity

const unsigned int NumPhysPages = 64;
const int MemorySize = (NumPhysPages * PageSize);
const int TLBSize = 4;           // if there is a TLB, make it small
```

- **Scheduling:**

code/thread/scheduler.h

```

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

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

```

code/thread/scheduler.cc

```
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()
{
    Scheduler(RR);
    //readyList = new List<Thread *>;
    //toBeDestroyed = NULL;
}

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;
}
```

code/thread/alarm.cc

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

    bool woken = _sleepList.PutToReady();
    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){
            cout << "=== interrupt->YieldOnReturn ===" << endl;
            interrupt->YieldOnReturn();
        }
    }
}
void
Alarm::WaitUntil(int x){
    //關中斷
    IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);
    Thread* t = kernel->currentThread;

    int worktime = kernel->stats->userTicks - t->getStartTime();
    t->setBurstTime(t->getBurstTime()+worktime);
    t->setStartTime(kernel->stats->userTicks);

    //cout << "Alarm::WaitUntil go sleep" << endl;
    _sleepList.PutToSleep(t, x);
    //開中斷
    kernel->interrupt->SetLevel(oldLevel);
}
```

在運行的時候計算 burstTime。

3. Result

(1) System Call: Sleep

```
aa@aa:~/r08942087_Nachos1/nachos-4.0/code/userprog$ ./nachos -e ../test/sleep1 -e ../test
/sleep2
Total threads number is 2
Thread ../test/sleep1 is executing.
Thread ../test/sleep2 is executing.
Sleep Time:500000(ms)
Sleep Time:1000000(ms)
Print integer:1
Sleep Time:500000(ms)
Print integer:1
Sleep Time:500000(ms)
Print integer:2
Sleep Time:1000000(ms)
Print integer:1
Sleep Time:500000(ms)
Print integer:1
Sleep Time:500000(ms)
Print integer:2
Sleep Time:1000000(ms)
Print integer:1
Sleep Time:500000(ms)
Print integer:1
Sleep Time:500000(ms)
Print integer:2
Sleep Time:1000000(ms)
Print integer:1
Sleep Time:500000(ms)
Print integer:1
Sleep Time:500000(ms)
Print integer:2
Sleep Time:1000000(ms)
Print integer:1
Sleep Time:500000(ms)
Print integer:1
return value:0
Print integer:2
return value:0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

Sleep1 的 delay 是 500000(ms)，然後會顯示 1；而 Sleep2 的 delay 是 1000000(ms)，然後會顯示 2。從實驗結果可以看到 sleep 真的有起作用，我們畫面中會看到兩個 1 後才看到一個 2，代表 systemcall sleep 有起到作用。

(2) CPU Scheduling

- FCFS

```
aa@aa:~/r08942087_Nachos1/nachos-4.0/code/userprog$ ./nachos FCFS -e ../
test/test1 -e ../test/test2
cpu sceduleing type:3
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!
```

(case 1)

```
A: remaining 5
A: remaining 4
A: remaining 3
A: remaining 2
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 3
C: remaining 2
C: remaining 1
C: remaining 0
D: remaining 4
D: remaining 3
D: remaining 2
D: remaining 1
D: remaining 0
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

(case 2)

再 First Come First Serve 裡面，先到的會先完成，所以會有個類似 fifo 的感覺，因此順序是 A->B->C->D。

- SJF

```
aa@aa:~/r08942087_Nachos1/nachos-4.0/code/userprog$ ./nachos SJF -e ../t
est/test1 -e ../test/test2
cpu sceduleing type:1
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!
```

(case 1)

```
C: remaining 3
C: remaining 2
C: remaining 1
C: remaining 0
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 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!
```

(case 2)

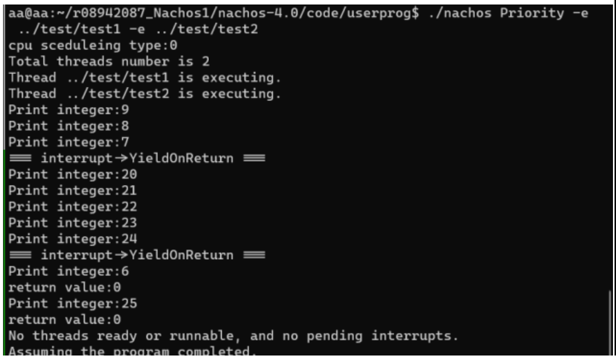
T_Name	A	B	C	D
T_burst	8	7	4	5

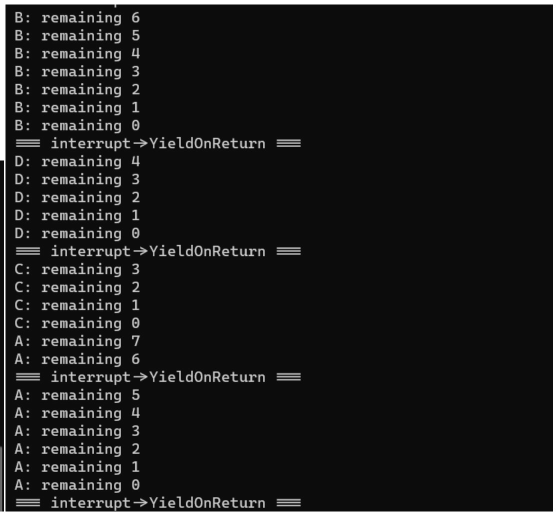
Case2:

Burst time: C < D < B < A

因此如果用 Short Job First 的話，會照著 Burst time 小的先執行，因此完成順序會是 C->D->B->A，結果也跟預期的依樣。

- Priority





(case 1) (case 2)

T_Name	A	B	C	D
T_Priority	1	9	7	8

Case 1:

因為兩個 thread 他們的 priority 相同，所以他們是輪流在執行，這便變成是 Round Robin 的感覺。

Case 2:

Priority: B>D>C>A

如果是 Priority 的話，會照著她們重要的優先順序跑，因此會是照著 B->D->C->A 的順序完成。