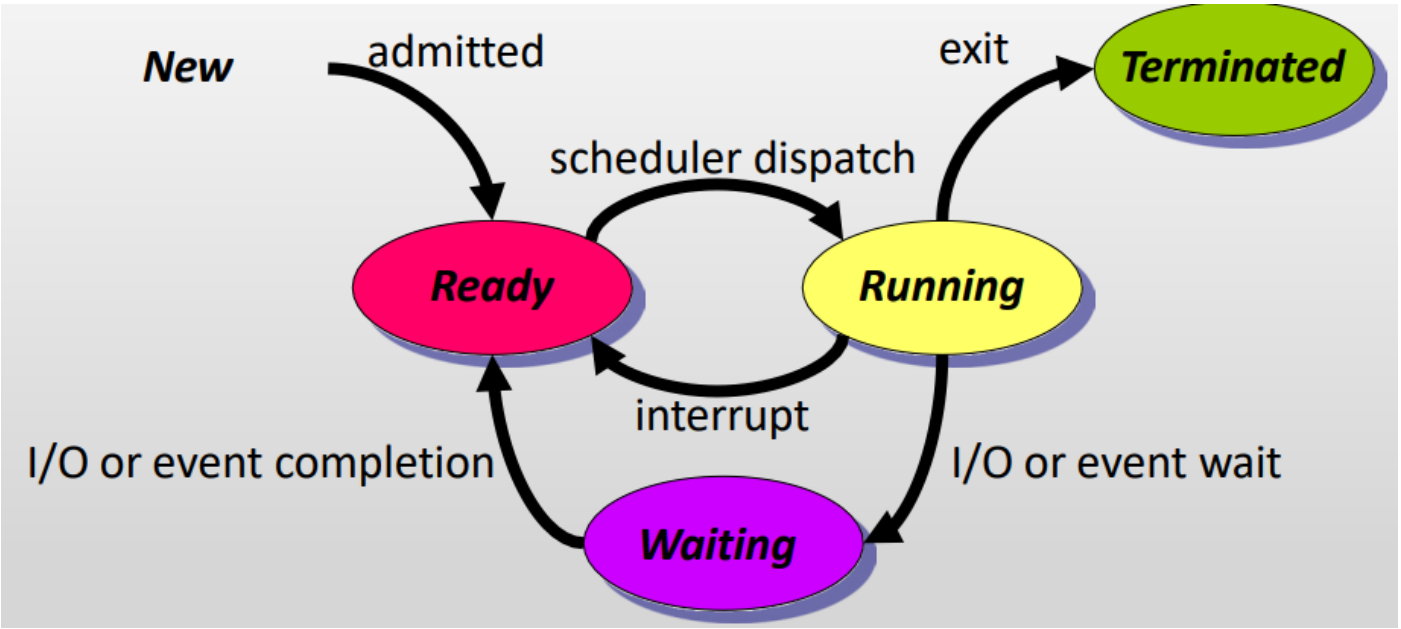
OS HW03 GROUP 01

Part 1:Trace Code

1. Explain following path



i. New – Ready

當 process 從 New state 到 Ready state:準備將 CPU 資源和記憶體分配給 process,通常發生在 process 剛建立時,OS 會分配 process 需要得 CPU 資源和記憶體並將 process 轉移到 ready queue 當中。

```
void Kernel::ExecAll()
{
    for (int i=1;i<=execfileNum;i++) {
        int a = Exec(execfile[i]);
    }
    currentThread->Finish();
    //Kernel::Exec();
}
```

首先進入 Kernel::ExecAll()依序呼叫 Kernel::Exec()執行所有執行檔案，當執行結束時結束 nachOS。

```
int Kernel::Exec(char* name)
{
    t[threadNum] = new Thread(name, threadNum);
    t[threadNum]->space = new AddrSpace();
    t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);
    threadNum++;

    return threadNum-1;
}
```

進入 Kernel::Exec()，首先為這個執行檔新增並初始化一個 thread 和記憶體空間 AddrSpace，透過 Fork()將程式 load 進 memory。

```
void Thread::Fork(VoidFunctionPtr func, void *arg)
{
    Interrupt *interrupt = kernel->interrupt;
    Scheduler *scheduler = kernel->scheduler;
    IntStatus oldLevel;

    StackAllocate(func, arg);
    oldLevel = interrupt->SetLevel(IntOff);

    scheduler->ReadyToRun(this);    // ReadyToRun assumes that interrupts are disabled!
    (void) interrupt->SetLevel(oldLevel);
}
```

在 Thread::Fork()分別新增 interrupt pointer 和 scheduler pointer，接下來分配並初始化 process 的 stack，並將 process 丟進 ready list。

```

void
Thread::StackAllocate (VoidFunctionPtr func, void *arg)
{
    stack = (int *) AllocBoundedArray(StackSize * sizeof(int));

#ifdef PARISC
    // HP stack works from low addresses to high addresses
    // everyone else works the other way: from high addresses to low addresses
    stackTop = stack + 16;  // HP requires 64-byte frame marker
    stack[StackSize - 1] = STACK_FENCEPOST;
#endif
#ifdef SPARC
    stackTop = stack + StackSize - 96;  // SPARC stack must contains at
                // least 1 activation record
                // to start with.
    *stack = STACK_FENCEPOST;
#endif
#ifdef PowerPC // RS6000
    stackTop = stack + StackSize - 16;  // RS6000 requires 64-byte frame marker
    *stack = STACK_FENCEPOST;
#endif
#ifdef DECMIPS
    stackTop = stack + StackSize - 4;   // -4 to be on the safe side!
    *stack = STACK_FENCEPOST;
#endif
#ifdef ALPHA
    stackTop = stack + StackSize - 8;   // -8 to be on the safe side!
    *stack = STACK_FENCEPOST;
#endif
#ifdef x86
    // the x86 passes the return address on the stack.  In order for SWITCH()
    // to go to ThreadRoot when we switch to this thread, the return addres
    // used in SWITCH() must be the starting address of ThreadRoot.
    stackTop = stack + StackSize - 4;   // -4 to be on the safe side!
    *(--stackTop) = (int) ThreadRoot;
    *stack = STACK_FENCEPOST;
#endif
#ifdef PARISC
    machineState[PCState] = PLabelToAddr(ThreadRoot);
    machineState[StartupPCState] = PLabelToAddr(ThreadBegin);
    machineState[InitialPCState] = PLabelToAddr(func);
    machineState[InitialArgState] = arg;
    machineState[WhenDonePCState] = PLabelToAddr(ThreadFinish);
```

```
#else
    machineState[PCState] = (void*)ThreadRoot;
    machineState[StartupPCState] = (void*)ThreadBegin;
    machineState[InitialPCState] = (void*)func;
    machineState[InitialArgState] = (void*)arg;
    machineState[WhenDonePCState] = (void*)ThreadFinish;
#endif
}
```

Thread::StackAllocate()首先分配一段 array 給 stack，並針對不同的架構進行修正，並在記憶體上限都設置一個 flag STACK_FENCEPOST 防止溢位。

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

    thread->setStatus(READY);
    readyList->Append(thread);
}
```

在 Scheduler::ReadyToRun()中將 process 的狀態設為 ready 並將 process 移入 readyList 當中等待執行。

ii. Running – Ready

Running state 到 Ready state 的過程是指一個正在執行的 process 由於某些原因被 scheduler 從 CPU 上移出，並放入到 ready queue 中，等待下一次被執行。

process state 的轉換是由作業系統的調度器負責的。scheduler 根據一定的策略，決定哪個 process 應該被調度執行。當一個 process 被 scheduler 從 CPU 上移除時，scheduler 會根據以下幾種情況進行處理：

- Time slice 結束：如果一個 process 的 time slice 已經結束，則 scheduler 會將它從 CPU 上移除，並放入到 ready queue 中，等待下一次被調度執行。
- I/O 操作：如果一個 process 正在等待 I/O 操作的結果，則調度器會將它從 CPU 上移除，並放入到 ready queue 中，等待 I/O 操作完成。
- 被中斷：如果一個 process 被其他 process 中斷，則 scheduler 會將它從 CPU 上移除，並放入到 ready queue 中。

```
void Machine::Run()
{
    Instruction *instr = new Instruction;  // storage for decoded instruction
    kernel->interrupt->setStatus(UserMode);
    for (;;) {
        OneInstruction(instr);
        kernel->interrupt->OneTick();
        if (singleStep && (runUntilTime <= kernel->stats->totalTicks))
            Debugger();
    }
}
```

這邊 nachOS 依照 MIPS 架構去實現執行程式的過程，透過無窮迴圈每一個 tick 下，模擬 CPU 去 fetch 指令。

```
void Interrupt::OneTick()
{
    MachineStatus oldStatus = status;
    Statistics *stats = kernel->stats;
```

```
// advance simulated time
    if (status == SystemMode) {
          stats->totalTicks += SystemTick;
          stats->systemTicks += SystemTick;
    } else {
          stats->totalTicks += UserTick;
          stats->userTicks += UserTick;
    }

// check any pending interrupts are now ready to fire
    ChangeLevel(IntOn, IntOff); // first, turn off interrupts
                // (interrupt handlers run with
                // interrupts disabled)
    CheckIfDue(FALSE);      // check for pending interrupts
    ChangeLevel(IntOff, IntOn); // re-enable interrupts
    if (yieldOnReturn) {    // if the timer device handler asked
                            // for a context switch, ok to do it now
          yieldOnReturn = FALSE;
          status = SystemMode;        // yield is a kernel routine
          kernel->currentThread->Yield();
          status = oldStatus;
    }
}
```

進到 Interrupt::OneTick()，首先確認當前執行的 mode 是 user mode 或是 system mode 去對個別 tick 進行累加，並確認是否有 pending Interrupt，確保沒有之後進到 currentThread->Yield 中。

```
void Thread::Yield ()
{
    Thread *nextThread;
    IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);

    ASSERT(this == kernel->currentThread);

    nextThread = kernel->scheduler->FindNextToRun();
    if (nextThread != NULL) {
          kernel->scheduler->ReadyToRun(this);
          kernel->scheduler->Run(nextThread, FALSE);
    }
    (void) kernel->interrupt->SetLevel(oldLevel);
}
```

進到 Thread::Yield()，進行 thread 的切換執行，首先 disable interrupt，接下來從 scheduler 那邊拿下一個要執行的 process，並執行 Run()。

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

    if (readyList->IsEmpty()) {
        return NULL;
    } else {
        return readyList->RemoveFront();
    }
}
```

Scheduler::FindNextToRun()會從 readyList 當中把最前面的 process 抓出來並回傳，readyList 結構如 stack。

```
void Scheduler::ReadyToRun (Thread *thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    thread->setStatus(READY);
    readyList->Append(thread);
}
```

進到 Scheduler::ReadyToRun()中，將當前得 process state 設成 ready，並將 process 丟進 readyList 當中等待被執行。

```
void Scheduler::Run (Thread *nextThread, bool finishing)
{
    Thread *oldThread = kernel->currentThread;

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

    if (finishing) {    // mark that we need to delete current thread
         ASSERT(toBeDestroyed == NULL);
         toBeDestroyed = oldThread;
    }

    if (oldThread->space != NULL) { // if this thread is a user program,
        oldThread->SaveUserState();     // save the user's CPU registers
        oldThread->space->SaveState();
    }

    oldThread->CheckOverflow();         // check if the old thread
                        // had an undetected stack overflow

    kernel->currentThread = nextThread;  // switch to the next thread
    nextThread->setStatus(RUNNING);      // nextThread is now running

    SWITCH(oldThread, nextThread);

    // interrupts are off when we return from switch!
    ASSERT(kernel->interrupt->getLevel() == IntOff);

    CheckToBeDestroyed();       // check if thread we were running
                    // before this one has finished
                    // and needs to be cleaned up

    if (oldThread->space != NULL) {     // if there is an address space
        oldThread->RestoreUserState();     // to restore, do it.
        oldThread->space->RestoreState();
    }
}
```

最後進到 Scheduler::Run()，進行 Context switch 並執行下一個 process，首先判斷 process 是否結束，若結束則將前一個 process 設為 toBeDestroyed，接下來進行溢位判斷。將 current thread 指向下一個要執行的 process，並將狀態設為 RUNNING，接續透過 SWITCH()進行 thread 的切換，最後將 oldThread 恢復原狀。

iii. Running – Waiting

Running state 到 Waiting state 的過程是指一個正在執行的 process 由於等待某些資源而無法繼續執行，因此被調度器從 CPU 上移除，並放入到 waiting queue 中，等待資源可用。

process state 的轉換是由 OS 的 scheduler 負責的。scheduler 根據一定的策略，決定哪個 process 應該被執行。當一個 process 被 scheduler 從 CPU 上移除時，scheduler 會根據以下幾種情況進行處理：

- I/O 操作：如果一個 process 正在等待 I/O 操作的結果，則 scheduler 會將它從 CPU 上移除，並放入到 waiting queue 中，等待 I/O 操作完成。
- 資源競爭：如果一個 process 正在等待某些資源，而這些資源被其他 process 佔用，則 scheduler 會將它從 CPU 上移除，並放入到 waiting queue 中，等待資源可用。
- 系統事件：如果發生了某些系統事件，例如中斷或是同步問題，semaphore, lock…，則 scheduler 會將正在執行的 process 從 CPU 上移除(block)，並放入到 waiting queue 中，等待系統事件處理完畢。

```
void
Semaphore::P()
{
    Interrupt *interrupt = kernel->interrupt;
    Thread *currentThread = kernel->currentThread;

    // disable interrupts
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    while (value == 0) {        // semaphore not available
    queue->Append(currentThread);   // so go to sleep
    currentThread->Sleep(FALSE);
    }
    value--;            // semaphore available, consume its value

    // re-enable interrupts
    (void) interrupt->SetLevel(oldLevel);
}
```

當發生 synchronize problem 時系統會呼叫 Sleep()，將當前的 process 進到 sleep 狀態(blocking)，此時當前的 process 會被丟入 queue 當中 waiting。

```
void Thread::Sleep (bool finishing)
{
    Thread *nextThread;

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

    status = BLOCKED;

    while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
        kernel->interrupt->Idle();  // no one to run, wait for an interrupt
    }
    // returns when it's time for us to run
    kernel->scheduler->Run(nextThread, finishing);
}
```

在 Thread::Sleep()當中，若進到特殊狀況，process 被迫進入 waiting 狀態，首先當前得 status

會被設為 BLOCKED 狀態，此時 scheduler 會去取得下一個 process 執行，若 scheduler 當中是空得則進入 idle 狀態，否則執行下一個 process。

iv. Waiting – Ready

Waiting state 到 Ready state 的過程是指一個處於 Waiting state 的 process 由於等待的資源可用，因此被 scheduler 從 waiting queue 中移除，並放入到 ready queue 中，等待下一次被執行。

process state 的轉換是由 nachOS 的 scheduler 負責的。scheduler 根據一定的策略，決定哪個 process 應該被執行。當一個 process 被 scheduler 從 waiting queue 中移除時，scheduler 會根據以下幾種情況進行處理：

- 資源可用：如果一個 process 正在等待某些資源，而這些資源已經可用，則 scheduler 會將它從 waiting queue 中移除，並放入到 ready queue 中，等待下一次被執行。
- 優先級：如果一個 process 的優先級高於其他 process，則 scheduler 會將它從 waiting queue 中移除，並放入到 ready queue 中，等待下一次被調度執行。
- poling：如果 scheduler 正在使用 poling 策略，則它會按照一定的順序從 waiting queue 中移除 process，並放入到 ready queue 中。

```
void
Semaphore::V()
{
    Interrupt *interrupt = kernel->interrupt;

    // disable interrupts
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    if (!queue->IsEmpty()) {  // make thread ready.
    kernel->scheduler->ReadyToRun(queue->RemoveFront());
    }
    value++;

    // re-enable interrupts
    (void) interrupt->SetLevel(oldLevel);
}
```

當 semaphore 被歸還時，表示某個等待中得 process 得到他要的資源，因此可以回復執行，這邊一樣過程中 interrupt 先被 disable 掉，這時候進到 waiting queue 當中看有沒有 process 正在 waiting，若有 process 正在 block 狀態，這時候執行 ReadyToRun()並將 waiting queue 中的 process 吐回去進到 readyList 中。

v. Running – Terminated

當 process 執行結束，這時候 process 將從 running state 進到 terminated state

```
void Kernel::ExecAll()
{
    for (int i=1;i<=execfileNum;i++) {
        int a = Exec(execfile[i]);
    }
    currentThread->Finish();
}
```

當執行完所有執行檔之後會呼叫 Thread::Finish()

```
void
Thread::Finish ()
{
    (void) kernel->interrupt->SetLevel(IntOff);
    ASSERT(this == kernel->currentThread);
    Sleep(TRUE);               // invokes SWITCH not reached
}
```

通常由 ThreadRoot 進行呼叫，並且不會直接將 process 的資源直接刪除，而是呼叫 scheduler 的 destructor 進行，並且整個過程是由另一個 thread 去執行。Finish()最後會呼叫 Sleep()並傳送參數 finishing=True 表示結束執行 thread。接下來 CPU 會去執行下一個 thread(FindNextToRun())。

vi. Ready – Running

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

    if (readyList->IsEmpty()) {
        return NULL;
    } else {
        return readyList->RemoveFront();
    }
}
```

當前一 process 執行結束會是因為特殊原因被 BLOCK，次時 scheduler 會進到 FindNextToRun() 從 readyList 中尋找下一個要被執行的 process，若 readyList 不為空則回傳 readyList 中隊頂端的 process 並執行 Run()。

```
/* void SWITCH( thread *t1, thread *t2 )
**
** on entry, stack looks like this:
**      8(esp)  ->              thread *t2
**      4(esp)  ->              thread *t1
**       (esp)  ->              return address
**
** we push the current eax on the stack so that we can use it as
** a pointer to t1, this decrements esp by 4, so when we use it
** to reference stuff on the stack, we add 4 to the offset.
*/
        .comm   _eax_save,4

        .globl  SWITCH
      .globl  _SWITCH
_SWITCH:
SWITCH:
        movl    %eax,_eax_save          # save the value of eax
        movl    4(%esp),%eax            # move pointer to t1 into eax
        movl    %ebx,_EBX(%eax)         # save registers
        movl    %ecx,_ECX(%eax)
        movl    %edx,_EDX(%eax)
        movl    %esi,_ESI(%eax)
        movl    %edi,_EDI(%eax)
        movl    %ebp,_EBP(%eax)
        movl    %esp,_ESP(%eax)         # save stack pointer
        movl    _eax_save,%ebx          # get the saved value of eax
```

```
        movl    %ebx,_EAX(%eax)         # store it
        movl    0(%esp),%ebx            # get return address from stack into ebx
        movl    %ebx,_PC(%eax)          # save it into the pc storage

        movl    8(%esp),%eax            # move pointer to t2 into eax

        movl    _EAX(%eax),%ebx         # get new value for eax into ebx
        movl    %ebx,_eax_save          # save it
        movl    _EBX(%eax),%ebx         # retore old registers
        movl    _ECX(%eax),%ecx
        movl    _EDX(%eax),%edx
        movl    _ESI(%eax),%esi
        movl    _EDI(%eax),%edi
        movl    _EBP(%eax),%ebp
        movl    _ESP(%eax),%esp         # restore stack pointer
        movl    _PC(%eax),%eax          # restore return address into eax
        movl    %eax,4(%esp)            # copy over the ret address on the stack
        movl    _eax_save,%eax
        ret
    #endif // x86
```

參數:

- t1: 指向目前正在執行的 process 的 pointer
- t2: 指向要切換到的 process 的 pointer

功能描述:

1. 保存 eax register:
    - 將 eax register 中的值存入 _eax_save 變數中。
    - 這一步非常重要，因為接下來的步驟會修改 eax register，需要先保存其原始值。
2. 保存目前 process 狀態:

- 從堆疊中取得目前 process t1 的地址。
- 將目前 process 的各個 register (ebx, ecx, edx, esi, edi, ebp, esp) 的值分別存入 t1 process 的相應記憶體位置。

3. 保存 return address:
    - 從堆疊中取得目前函數的返回地址。
    - 將返回地址存入 t1 process 的 PC 儲存位置。
4. 恢復新 process 狀態:
    - 從堆疊中取得新的 process t2 的地址。
    - 從 t2 process 的記憶體位置中讀取 eax register 值並存入 ebx register。
    - 恢復 t2 process 的各個 register (ebx, ecx, edx, esi, edi, ebp, esp) 的值。
    - 恢復 t2 process 的 PC 值，也就是新函數的執行地址。
5. 準備 return:
    - 將新的返回地址 (t2 process 的 PC 值) 複製到堆疊中。
    - 將 eax register 值恢復成之前保存的值。

6. return:
    - 執行新的函數 (t2 process 的函數)。

函數中使用的變數:

- _eax_save: 用於保存 eax register 原始值的變數。
- _EBX(%eax), ...: 用於保存各個 register 值的變數，它們被定義在 t1 process 的記憶體空間中。
- _PC(%eax): 用於保存 PC 值 (返回地址) 的變數，它被定義在 t1 process 的記憶體空間中。

Part 2:Implementation

1. Detail of your implementation

Kernel.h 新增 threadPriority 存取每個預計執行的 process 個別的 priority

```
class Kernel {
    …
    private:
      int threadPriority[10];
    …
}
```

Kernel.cc constructor 新增"-ep" argument 判斷是否進行 CPU schduling
因應 INPUT ARGUMENT 格式: $ ../build.linux/nachos –ep test1 40 –ep test2 80

```
else if(strcmp(argv[i], "-ep")==0){
    ASSERT(i+2<argc);
    execfile[++execfileNum] = argv[++i];
    threadPriority[execfileNum] = atoi(argv[++i]);

    if(threadPriority[execfileNum] > 149){
        threadPriority[execfileNum] = 149;
    }else if(threadPriority[execfileNum] < 0){
        threadPriority[execfileNum] = 0;
    }
    cout << execfile[execfileNum]<<"\n";
    cout << "Priority = " << threadPriority[execfileNum]<<"\n";
}
```

KERNEL::ExecAll()在 funciton Exec()新增一個參數傳遞個別 thread 的 priority

```
void Kernel::ExecAll()
{
    for (int i=1;i<=execfileNum;i++) {
        int a = Exec(execfile[i], threadPriority[i]);
    }
    currentThread->Finish();
    //Kernel::Exec();
}
```

Kernel::Exec(char * name, int priority) 新增參數並初始化 thread priority

```
int Kernel::Exec(char* name, int priority)
{
    t[threadNum] = new Thread(name, threadNum);
    t[threadNum]->space = new AddrSpace();
    t[threadNum]->setPriority(priority);
    t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);
    threadNum++;

    return threadNum-1;
}
```

Thread.h 新增 set/get priority

```
class Thread {
    …
    private:
        // Add priority to thread
        int priority;
    public:
        int getPriority() { return (priority);}
        void setPriority(int p) { priority = p; }
    …
}
```

Scheduler.h 新增 srorted list priority queue 存取要執行的 processes

```
class Scheduler {
    …
    private:
        SortedList<Thread *> *priorityList; // queue of threads that
    …
};
```

Scheduler.cc 新增 comparison function 比較 thread 之間的 priority

```
int compare_priority(Thread *t1, Thread *t2){
    if(t1->getPriority() < t2->getPriority()){
        return -1;
    }else if(t1->getPriority() > t2->getPriority()){
        return 1;
    }else{
        return (t1->getID() < t2->getID())?-1:1;
    }
    return 0;
}
```

Initialize priorityList in scheduler constructor

```
Scheduler::Scheduler()
{
    readyList = new List<Thread *>;
    priorityList = new SortedList<Thread *>(compare_priority);
    toBeDestroyed = NULL;

}
```

Delete priorityList at destructor

```
Scheduler::~Scheduler()
{
    delete readyList;
    delete priorityList;
}
```

Thread * Scheduler::FindNextToRun() 將要執行得 process 從 priorityList 取出回傳

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

    DEBUG(dbgExpr, "[B] Tick [" << kernel->stats->totalTicks<<"]: Thread ["
            <<priorityList->Front()->getID() << "] is removed from priority List");
    if(!priorityList->IsEmpty()){
        return priorityList->RemoveFront();
    }else{
        return NULL;
    }
}
```

新增 debugger flag attribute ‘z’
debug.h 新增 macro

```
    const char dbgExpr = 'z';       // debugger for cpu schduling
```

[A] Whenever a process is inserted into a priority queue

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

    DEBUG(dbgExpr, "[A] Tick ["<<kernel->stats->totalTicks<<"]: Thread ["
                    << thread->getID() << "] is inserted into queue priorityList");
    thread->setStatus(READY);

    if(!priorityList->IsInList(thread)){
        priorityList->Insert(thread);
    }
}
```

[B] Whenever a process is removed from a queue

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

    DEBUG(dbgExpr, "[B] Tick [" << kernel->stats->totalTicks<<"]: Thread ["
            <<priorityList->Front()->getID() << "] is removed from priority List");
    if(!priorityList->IsEmpty()){
        return priorityList->RemoveFront();
    }else{
        return NULL;
    }
}
```

[C] Whenever a process changes its scheduling priority
並不會發生
[D] Whenever a context switch occurs

```
void Scheduler::Run (Thread *nextThread, bool finishing)
{
    …
    nextThread->setStatus(RUNNING);      // nextThread is now running
    DEBUG(dbgExpr, "[D] Tick [" <<  kernel->stats->totalTicks << "]: Thread [" <<
nextThread->getID() << "] is now selected for execution, thread [" << oldThread->getID()
```

```
<< "] is replaced, and it has executed ["<< kernel->stats->totalTicks -
kernel->stats->prevTicks << "] ticks");
    kernel->stats->prevTicks = kernel->stats->totalTicks;
    …
}
```

Result:
ConsolIO_test1 priority: 70
ConsolIO_test2 priority: 50
ConsolIO_test3 priority: 80
執行順序: ConsolIO_test3→ ConsolIO_test1→ ConsolIO_test2

```
        ".bss", filepos 0x0, mempos 0x1c0, size 0x0
3
3
3
3
3
3
3
3
3
3
3
3
3
3
return value:0
1
1
1
1
return value:0
2
2
2
2
2
return value:0
done
OK (1.019 sec real, 1.019 sec wall)
Triggering a new build of os_group11_ta
Finished: SUCCESS
```

Part 3:Contribution

1. **Describe details and percentage of each member's contribution.**
   100%by 楊士賢