NachOS – MP3

CPU Scheduling

宋體淮, R09921135, Electrical Engineering

I. Goal

The default CPU scheduling algorithm of Nachos is a simple round-robin scheduler with 100 ticks time quantum. The goal of this project is to replace it with **other scheduling methods (FIFO, SJF, Priority), and understand the implementation of process lifecycle management and context switch mechanism**.

II. Assignment

1. Trace code

Explain the purposes and details of the following 6 code paths to understand how nachos manages the lifecycle of a process (or thread).

1.1 New→Ready

- 這個階段其實就是一條 thread 剛被建立、程式被 load 進記憶體，最後被放進 ready queue 裡等待 CPU 執行的過程。

main()：

- 首先主程式會 bootstrap NachOS kernel。
  - 主程式接收 command line 參數，並利用 strcmp 做剖析。
  - 會把要執行的程式名稱加進 `execfile`，等到之後執行。
  - 接著創建一個 kernel，對其初始化，執行 `kernel->ExecAll()` 來跑使用者程式。

```
else if (strcmp(argv[i], "-e") == 0) {
execfile[++execfileNum]= argv[i];
        cout << execfile[execfileNum] << "\n";
        i++;
}
```

Kernel::ExecAll()：

- 在這裡會從 execfileNum 判斷有多少要執行的程式(或是稱作 thread)，並以 Exec()來一一執行。
- 跑完之後因為 main thread 的功能完成了，因此讓 main thread -> Finish()，結束 NachOS。

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

Kernel::Exec ()：

- 首先為這個程式創造一條 thread，並給予其定址空間(AddrSpace)。
- 透過 Fork()載入真正要執行的程式，傳入一個 function pointer (&ForkExecute)，以及這個 thread 自己(t[threadNum])。

```
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;
}
```

ForkExecute()：

- ForkExecute()何時才會被呼叫呢？
  - 當這個 thread 被 scheduler 選到之後會做 context switch (在 switch.S 中進行)，換成這個 thread 的 context 之後會呼叫一個 ThreadRoot 函式(也定義在 switch.S)，ThreadRoot 函式會執行以下步驟：
    - Thread->Begin()。
    - ForkExecute() (user program 就是在這邊執行)。
    - Thread->Finish()。
- 這個函式會呼叫 AddrSpace:: Load()來將 thread 載入到記憶體中，並呼叫 AddrSpace::Execute()來執行它。

```
void ForkExecute(Thread *t)
{
        if ( !t->space->Load(t->getName()) ) {
        return;
    }
    t->space->Execute(t->getName());
}
```

AddrSpace::Execute()：

- 在此會將 currentThread 與目前 thread 的定址空間(this)拉在一起。
- 初始化一些 user program register。
- 把該 thread 的 page table 載入進來。
- 呼叫 kernel->machine->Run()來開始執行程式，fetch instruction。

```
void AddrSpace::Execute(char* fileName)
{

    kernel->currentThread->space = this;

    this->InitRegisters();
    this->RestoreState();

    kernel->machine->Run();

    ASSERTNOTREACHED();
}
```

Thread::Fork()：

- 回到 Fork()，它會呼叫 StackAllocate()來為此 thread 安排其 stack 空間(在真實機器上的)，其中 func 就是&ForkExecute，arg 就是 thread 自己。
- StackAllocate()執行結束，將 interrupt disable，因為這個過程不能被打斷，並將此 thread 加進 ReadyToRun 裡面(就是在等待 CPU 的 ready queue)。

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

Thread::StackAllocate()：

- `AllocBoundedArray()` 會回傳一個 array，代表一段合法可以 access 的記憶體(此 thread 的 stack frame)，這個記憶體是在真實機器上的記憶體，並讓 `stack` 指標指向其頂部 (low Address)。

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

- 讓 `stackTop` 指向 stack 的底部 (high Address)，為了確保安全，多減一格 (`stackSize - 4`)。
- 讓 `stackTop - 1` 的內容是 `ThreadRoot` 函式(因函式名稱代表位址，所以這是將 `ThreadRoot` 函式的 address push 進 stack)，以便將來 x86 組語做 context switch 之後可以直接從 stack 的這個地方 access 到 `ThreadRoot` 函式，去開始執行程式，因此可發現它是所有 thread 的執行入口。

```
#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
```

- 最後讓 `machineState` (host machine register) 的各個特定 element 去存特定的 function pointer。這些步驟都是為了之後 context switch 的時候，能讓 kernel 正確執行特定的 function，以及正確找到這個 thread 的 stack。

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

Note：在 `thread.cc/thread.h` 中定義了 `machineState`，它共有 `MachineStateSize`(75)個 items，代表的是 host machine register(真實機器的)，不是 nachos register (user program register)，完全是為了 context switch 所需要。

AllocBoundedArray ()：

- 此函式定義在 `sysdep.cc`，用來回傳一個 stack 空間，其中可以發現它會

在 stack 前後各多 allocate 一個 page 的大小(pgSize * 2 + size)，當產生了指向這兩個 page 範圍中的 memory access 時，就能以此來判定是否 access 到「不合法位址」來 catch thread overflow(因有用 mprotect()函式包起來)。

- mprotect()在 sysdep.cc 中因有#include<sys/mman.h>而可以使用。

```
AllocBoundedArray(int size)
{
#ifdef NO_MPROT
    return new char[size];
#else
    int pgSize = getpagesize();
    char *ptr = new char[pgSize * 2 + size];

    mprotect(ptr, pgSize, 0);
    mprotect(ptr + pgSize + size, pgSize, 0);
    return ptr + pgSize;
#endif
}
```

Scheduler::ReadyToRun()：

- 回到 ReadyToRun()，當 Fork 函式進行完 StackAllocate()後接著呼叫此函式，將 thread 的狀態設為 ready。
- 把剛配置好的 thread 放進 readyList，等待 scheduler 選到該 thread 去執行。

```
Scheduler::ReadyToRun (Thread *thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());

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

1.2 Running→Ready

- 通常會發生這個階段可能是有一些 interrupt 發生(time slice 到了、或者被更高優先權的 process preempt 等等)。
- 簡單來說，當 thread 1 要被 thread 2 給 preempt 時，必須 Yield(讓出控制權)，而 Yield 裡面會做：
  - disable interrupt (確保整個 thread 切換的過程是 atomic 的)。

— FindNextThreadToRun

— Run (context switch 在此執行)。

Machine::Run()：

- 此函式定義於 Machine.h，在 Mipssim.c 裡面實作，用於模擬 MIPS 架構的執行過程。
- 它就是用一個無窮迴圈反覆抓取 user program 的程式碼並 decode，然後用 OneTick 來模擬每個 clock 的執行。
- 再來回顧一下此函式是被誰呼叫的：

  — 當一個 user program 的 thread 被 scheduler 選到做完 context switch 之後，執行到 ForkExecute() 時，其中會呼叫 AddrSpace::Execute()。

```
void ForkExecute(Thread *t)
{
        if ( !t->space->Load(t->getName()) ) {
        return;
    }
    t->space->Execute(t->getName());
}
```

  — 在 AddrSpace::Execute() 裡就會呼叫 kernel->machine->Run() 來開始執行程式，fetch instruction。

```
void AddrSpace::Execute(char* fileName)
{

    kernel->currentThread->space = this;

    this->InitRegisters();
    this->RestoreState();

    kernel->machine->Run();

    ASSERTNOTREACHED();
}
```

Interrupt::OneTick()：

- 遞增 stats 裡面所記錄的 Ticks，來模擬的系統時間的前進(根據現在是 user mode 還是 system mode)。
- 在 CheckIfDue 中檢查是否有要執行的 interrupt，並執行它。
- 根據 yieldOnReturn (此 flag 來自 timer interrupt) 決定是否要做 context

switch，若要的話則：

— `yieldOnReturn` 必須先恢復為 false (否則若 timer 還沒到下次要 interrupt 時，卻因 flag 忘記 reset，會不停的 context switch)。

— 讓 kernel 現在執行的 thread (`currentThread`) 跑 `Yield()`，把 CPU 資源讓出來給另一個 thread。

```
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;
    }
    DEBUG(dbgInt, "== Tick " << stats->totalTicks << " ==");

// 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;
    }
}
```

- `yieldOnReturn` 何時會被設為 true 呢？首先在 kernel 初始化時會創建一個 `Alarm` 物件。

```
  stats = new Statistics();           // collect statistics
  interrupt = new Interrupt;          // start up interrupt handling
  scheduler = new Scheduler();        // initialize the ready queue
  alarm = new Alarm(randomSlice);     // start up time slicing
  machine = new Machine(debugUserProg);
  synchConsoleIn = new SynchConsoleInput(consoleIn); // input from stdin
  synchConsoleOut = new SynchConsoleOutput(consoleOut); // output to stdout
  synchDisk = new SynchDisk();    //
```

Alarm::Alarm()：

- `Alarm` 的 constructor 會新建一個 `timer` 物件，其中 `doRandom` 參數指的是「是否要隨機觸發 timer interrupt」，預設是 false。

```
// 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
                                // this method is not yet implemented

  private:
    Timer *timer;               // the hardware timer device

    void CallBack();            // called when the hardware
                                // timer generates an interrupt
};
```

- 而 this 指的就是 Alarm 物件本身，目的是等等 timer interrupt 時觸發的 callback 要回來觸發 Alarm 自己。

```
Alarm::Alarm(bool doRandom)
{
    timer = new Timer(doRandom, this);
}
```

```
Timer::Timer(bool doRandom, CallBackObj *toCall)
{
    randomize = doRandom;
    callPeriodically = toCall;
    disable = FALSE;
    SetInterrupt();
}
```

Timer::SetInterrupt()：

- 跳到這個地方，它會做 kernel->interrrupt->Schedule(this, delay, TimerInt)，代表向 kernel 的 interrupt 物件排程一個離現在 delay 時間以後的中斷事件(要 preempt 其他 thread 就是一種中斷事件)，並且是由 TimerInt 硬體觸發的。

```
Timer::SetInterrupt()
{
    if (!disable) {
       int delay = TimerTicks;

       if (randomize) {
            delay = 1 + (RandomNumber() % (TimerTicks * 2));
        }
       // schedule the next timer device interrupt
       kernel->interrupt->Schedule(this, delay, TimerInt);
    }
}
```

Interrupt::Schedule()：

- 而 Schedule 會把剛剛傳進來的參數(Timer 本身)包成一個 PendingInterrupt 物件，並放入 interrupt 的 pendinglist 中，

等待 OneTicks() 中的 CheckIfDue() 去觸發這個 interrupt。

```
Interrupt::Schedule(CallBackObj *toCall, int fromNow, IntType type)
{
    int when = kernel->stats->totalTicks + fromNow;
    PendingInterrupt *toOccur = new PendingInterrupt(toCall, when, type);

    DEBUG(dbgInt, "Scheduling interrupt handler the " << intTypeNames[type] << " at time = " << when)
;
    ASSERT(fromNow > 0);

    pending->Insert(toOccur);
}
```

Interrupt::CheckIfDue()：

- next 是一個 PendingInterrupt 物件，而 next->callOnInterrupt 是剛剛被包起來的 Timer，所以 next->callOnInterrupt->CallBack() 就是呼叫 Timer.CallBack()，跳回到 Timer 去執行中斷的內容。

```
inHandler = TRUE;
do {
    next = pending->RemoveFront();     // pull interrupt off list
    next->callOnInterrupt->CallBack();// call the interrupt handler
    delete next;
} while (!pending->IsEmpty()
           && (pending->Front()->when <= stats->totalTicks));
inHandler = FALSE;
return TRUE;
```

Timer::CallBack()：

- 在這邊會呼叫 callPeriodically->CallBack()，這其實就是 Alarm.CallBack()。
- 除此之外，又再次呼叫 SetInterrupt()，再 schedule 一次同樣的中斷事件，如此反覆的中斷，就可以做到只要 time slice 一到就中斷這個 thread。

```
Timer::CallBack()
{
    // invoke the Nachos interrupt handler for this device
    callPeriodically->CallBack();

    SetInterrupt();      // do last, to let software interrupt handler
                         // decide if it wants to disable future interrupts
}
```

Alarm::CallBack()：

- 就是在這邊真正執行中斷的內容，會呼叫 interrupt->YieldOnReturn()。

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

    if (status != IdleMode) {
        interrupt->YieldOnReturn();
    }
}
```

Interrupt:: YieldOnReturn()：

- 就是在這個地方把 `yieldOnReturn` 這個 flag 設為 true。所以上面繞了一大圈只是要表示說可以 context switch 了，還沒有真的 switch。

```
Interrupt::YieldOnReturn()
{
    ASSERT(inHandler == TRUE);
    yieldOnReturn = TRUE;
}
```

Interrupt::OneTick()：

- 解釋完了 `yieldOnReturn` 何時被設為 true 之後，再回到 `OneTick()`。這時會讓 kernel 現在執行的 thread (`currentThread`)跑 `Yield()`。

```
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;
}
```

Thread::Yield()：

- `Yield()`的目的就是要切換 thread 來執行 (最後會間接透過 Run( )來做 context switch)。
- scheduler 會藉由 `FindNextToRun()`，把 `nextThread`，從 ready queue(`readyList`)裡面拿出來。
- 藉由 `ReadToRun()`把目前的 thread 放回 ready queue。
- 運行 scheduler。

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

    ASSERT(this == kernel->currentThread);

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

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

Scheduler::FindNextToRun()：

- 檢查 `readList` 是否為空，否的話就 de-queue 並 return 下一條(front) thread。

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

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

Scheduler::ReadyToRun()：

- 將準備要執行的 thread(執行 `Yield` 的那一條 thread)的 status 設置為 ready，並放入 `readyList`。

```
Scheduler::ReadyToRun (Thread *thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());

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

Scheduler::Run()：

- `Run()` 基本上就是執行下一條 thread，而 context switch 在此進行，步驟大致如下：
  - 如果 `finishing` 此一參數為 true，代表原本的 thread 已經執行完成了要刪除掉，此時讓 `toBeDestroyed` 指向 `oldThread`(原本的

thread)。

— 保存 oldThread 的 UserState (基本上就是 user program 對應到的 register set)，存入 thread class(類似 PCB)。

— 檢查 oldthread 的 stack 有沒有 overflow(用到不該用的地方)。

— 將 kernel 所執行的 currentThread 改為準備要執行的 thread，並設置 status 為 Running，接著呼叫 SWITCH() 組語正式進行線程切換。

- SWITCH() 分別是在 thread.h、switch.h 定義相關巨集和參數，而在 switch.s 實作和進行。
- SWITCH() 裡面在做的事是把 CPU register 換成新的 thread 的 register state (詳情參見 1.6 節)。

— CheckToBeDestroyed() 會真正去刪除上面步驟所說的 toBeDestroyed。

```
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

    DEBUG(dbgThread, "Switching from: " << oldThread->getName() << " to: " << nextThread->getName());

    // This is a machine-dependent assembly language routine defined
    // in switch.s.  You may have to think
    // a bit to figure out what happens after this, both from the point
    // of view of the thread and from the perspective of the "outside world".

    SWITCH(oldThread, nextThread);

    // we're back, running oldThread

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

    DEBUG(dbgThread, "Now in thread: " << oldThread->getName());

    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();
    }
}
```

1-3. Running→Waiting (only need to consider console output as an example)

- 會發生 Running→Waiting 通常是因為和做 I/O 有關，因此發生 interrupt，並透過 `Sleep()` 函式來 block 掉這個 thread。

- 回顧一下，在 kernel 的初始化階段會創建 `SynchConsoleInput`、`SynchConsoleOutput` 物件，其中 `consoleIn`、`consoleOut` 的預設值為 NULL (代表 `stdin` 跟 `stdout`)。

```
stats = new Statistics();           // collect statistics
interrupt = new Interrupt;          // start up interrupt handling
scheduler = new Scheduler();        // initialize the ready queue
alarm = new Alarm(randomSlice);     // start up time slicing
machine = new Machine(debugUserProg);
synchConsoleIn = new SynchConsoleInput(consoleIn); // input from stdin
synchConsoleOut = new SynchConsoleOutput(consoleOut); // output to stdout
synchDisk = new SynchDisk();    //
```

- 而 `synchConsoleOut` 裡面其實又包含了 `ConsoleOutput`(定義於 `console.h`)，以及 `Lock` 跟 `Semaphore` 的宣告。

```
SynchConsoleOutput::SynchConsoleOutput(char *outputFile)
{
    consoleOutput = new ConsoleOutput(outputFile, this);
    lock = new Lock("console out");
    waitFor = new Semaphore("console out", 0);
}
```

- 再來看到 `ConsoleOutput` 的建構子，可發現 `toCall` 其實就是指 `synchConsoleOut` 物件本身，而 `callWhenDone = toCall` 此行將 `SynchConsoleOutput` 和 `ConsoleOutput` 之間緊密的牽連在一起了。

```
ConsoleOutput::ConsoleOutput(char *writeFile, CallBackObj *toCall)
{
    if (writeFile == NULL)
        writeFileNo = 1;                            // display = stdout
    else
        writeFileNo = OpenForWrite(writeFile);

    callWhenDone = toCall;
    putBusy = FALSE;
}
```

SynchConsoleOutput::PutChar()：

- 從這個函式開始看起。由於不能同時有兩個 thread 在做 console 輸出，故先搶鎖(`lock->Acquire()`)。
- 接著執行 `consoleOutput->PutChar`：

— writeFileNo 在初始化的時候已經被設成 1 (就是 stdout) 了，故 WriteFile() 會將 1 個字元寫上 stdout。

— PutChar 的最後會將 ConsoleOutput 本身放進去 interrupt pending list，排程一個 interrupt 事件，這個 interrupt 的目的是為了要釋放 semaphor 資源並喚醒睡在 semaphor waiting queue 的人 。

  - ConsoleTime 在本次作業被設定為 1，即下一個 tick 就會發生 console write interrupt。

```
ConsoleOutput::PutChar(char ch)
{
    ASSERT(putBusy == FALSE);
    WriteFile(writeFileNo, &ch, sizeof(char));
    putBusy = TRUE;
    kernel->interrupt->Schedule(this, ConsoleTime, ConsoleWriteInt);
}
```

— 當 ConsoleTime 過去，console write interrupt 發生，就會執行 ConsoleOutput->Callback( )。

— 這個 CallBack 裡面又呼叫了 callWhenDone->CallBack()，其實就是 SynchConsoleOutput->CallBack()。

```
ConsoleOutput::CallBack()
{
    putBusy = FALSE;
    kernel->stats->numConsoleCharsWritten++;
    callWhenDone->CallBack();
}
```

— waitFor->V() 就是 signal function，因為已完成輸出了一個字元，所以釋放一個 semaphor 資源(代表可以再印下一個了)。

```
SynchConsoleInput::CallBack()
{
    waitFor->V();
}
```

- 執行 waitFor->P()，等同 wait function，占用一個 semaphor 資源，因為他可能要再印第二個字元，但因為自己才剛用掉了 semaphor 資源，所以這時候就會被放進 semaphor 的 waiting queue 裡，並做 Sleep()。
- 做到此代表 putChar() 程序全部完成，呼叫 lock->Release() 來釋放 lock，可以讓其他 thread 來使用。

```
SynchConsoleOutput::PutChar(char ch)
{
    lock->Acquire();
    consoleOutput->PutChar(ch);
    waitFor->P();
    lock->Release();
}
```

Semaphore::P()：

- P()等同於 wait function，要佔用資源，因此若沒有資源(value==0)要把自己 suspend 掉(避免 CPU 的 busy-waiting)，所以把自己這個 thread 加到 semaphore 的 waiting queue。
- 並且呼叫 Sleep()，其中參數 false 的意思是「thread 還沒結束等等還要回來執行我」。
- 而 semaphor 的資源就是 value，那 value 是何時指定的呢？
  - 在初始化 semaphore 物件時，就要指定初始資源有多少。
  - 來看到 SynchConsoleOutput 的 constructor，新建了一個初始值為 0 的 semaphore。

```
SynchConsoleOutput::SynchConsoleOutput(char *outputFile)
{
    consoleOutput = new ConsoleOutput(outputFile, this);
    lock = new Lock("console out");
    waitFor = new Semaphore("console out", 0);
}
```

  - 那是誰第一次釋放資源的呢？回憶上面所講到的，在做 consoleOutput->PutChar()之後會排程一個 interrupt，當此 interrupt 被執行時會來到 SynchConsoleOutput 的 CallBack()，在這裡會做 waitFor->V()釋放資源，代表說「做完了，可以再印下一個字元(有資源可以用)」。
  - 由於要讓第一個人一定可以使用資源，所以才會安排第一個 process 不用做 waitFor->P()就可以先印字元。

```
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);
}
```

SynchList<T>::Append(T)：

- 來看到 waiting queue 的實作，可以發現它其實就是一個 singly linked list。

```
List<T>::Prepend(T item)
{
    ListElement<T> *element = new ListElement<T>(item);

    ASSERT(!IsInList(item));
    if (IsEmpty()) {            // list is empty
        first = element;
        last = element;
    } else {                    // else put it before first
        element->next = first;
        first = element;
    }
    numInList++;
    ASSERT(IsInList(item));
}
```

Thread::Sleep()：

- 當把這個等待 semaphor 的 thread 加進 waiting queue 之後，就會呼叫 `Sleep()`。首先會把 thread 的狀態設為 blocked。
- 藉由 `FindNextToRun()` 讓 scheduler 從 ready queue 找出下一條要執行的 thread。
  - 若無(NULL)，`Idle()`裡面會判斷若有 interrupt 的話則 advance clock 到該 interrupt 要發生的時間，若無 interrupt 則直接 halt 程式。
  - 若有，則呼叫 `Run()` 來執行 context switch，換下一個 thread 擁有 CPU 使用權。

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

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

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

    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);
}
```

Scheduler::FindNextToRun()：

- (參見 1.2 節)

Scheduler::Run()：

- (參見 1.2 節)
- 要注意的是，這裡面收到的 `finishing` 參數為 false，因為上一個 thread 只是被 block 掉而已，還沒 finish，之後還要執行它。

1-4. Waiting→Ready (only need to consider console output as an example)

- 回顧上一節，`SynchConsoleOutput::PutChar` 的例子，當做完，`consoleOutput` 的 `PutChar` 之後，會再繞一大圈去執行 `CallBack` 來釋放 semaphor 資源，喚醒睡在 waiting queue 的人。

Semaphore::V()：

- 當要釋放資源之前，thread 會先看這個 semaphor 的 waiting queue 裡面有沒有其他 thread 在等待，並從裡面 de-queue，再執行 `ReadyToRun()` 把該 thread 重新放進 ready queue，狀態從 blocked 變為 ready。
- 把 semaphore value++。

```
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);
}
```

Scheduler::ReadyToRun()：

- (參見 1.2 節)

1-5. Running→Terminated (start from the Exit system call is called)

- 一個 process 從 running 到 terminated 代表他執行完了，這個程式可以結束

掉並且換下個程式來執行了。當我們要把這個程式 terminate 時，要呼叫 SC_Exit 這個 system call。

ExceptionHandler()：

- 以現在的例子而言，exception 類型是 SyscallException，且 system call 類型是 SC_Exit，這個 system call 會執行 kernel->currentThread->Finish()。

```
    case SC_Exit:
                {
DEBUG(dbgAddr, "Program exit\n");
                val=kernel->machine->ReadRegister(4);
                cout << "return value:" << val << endl;
                kernel->currentThread->Finish();
```

Thread::Finish()：

- 這裡面會呼叫 Sleep()。
- 這裡跟上面不一樣的是，Sleep 的參數為 true，最後會再被傳進 Run()裡面，代表 thread 要結束了。

```
Thread::Finish ()
{
    (void) kernel->interrupt->SetLevel(IntOff);
    ASSERT(this == kernel->currentThread);

    DEBUG(dbgThread, "Finishing thread: " << name);
    Sleep(TRUE);                                    // invokes SWITCH
    // not reached
}
```

Thread::Sleep()：

- (參見 1.3 節)
- 這裡面的 Run()的 finishing 參數是 true，代表 thread 要結束了。

Scheduler::FindNextToRun()：

- (參見 1.2 節)

Scheduler::Run()：

- (參見 1.2 節)
- 但這裡面收到的 finishing 參數為 true，因為上一個 thread 呼叫了

finish( )。

1-6. Ready→Running

Scheduler::FindNextToRun()：

- (參見 1.2 節)

Scheduler::Run()：

- (參見 1.2 節)
- 在這個階段，當 nextThread 的 status 被設置為 running 後，接下來馬上就會做 context switch。

SWITCH()：

- SWITCH 主要是透過 switch.h 來 define macro。
- 以及 thread.h 內的外部宣告(extern)。
- 最後在 switch.s 裡面使用組合語言實作(機器是屬於 x86 架構)。

switch.h：

- 在這邊 define macro，宣告一些 register 的位置，為了讓 switch.s 取用。

```
#ifdef x86

/* the offsets of the registers from the beginning of the thread object */
#define _ESP     0
#define _EAX     4
#define _EBX     8
#define _ECX     12
#define _EDX     16
#define _EBP     20
#define _ESI     24
#define _EDI     28
#define _PC      32

/* These definitions are used in Thread::AllocateStack(). */
#define PCState         (_PC/4-1)
#define FPState         (_EBP/4-1)
#define InitialPCState  (_ESI/4-1)
#define InitialArgState (_EDX/4-1)
#define WhenDonePCState (_EDI/4-1)
#define StartupPCState  (_ECX/4-1)

#define InitialPC       %esi
#define InitialArg      %edx
#define WhenDonePC      %edi
#define StartupPC       %ecx

#endif // x86
```

thread.h：

- 在 scheduler::Run 裡面的 SWITCH()函式是在這邊宣告的。
- 透過 extern 的宣告，使得 x86 組語能夠與 C 語言互相呼叫。

```
extern "C" {
// First frame on thread execution stack;
//      call ThreadBegin
//      call "func"
//      (when func returns, if ever) call ThreadFinish()
void ThreadRoot();

// Stop running oldThread and start running newThread
void SWITCH(Thread *oldThread, Thread *newThread);
}
```

switch.s：

- 這裡的 SWITCH:代表的是一個名為 SWITCH 的記憶體區段，所以當 C 語言程式執行到 SWITCH()函式時就會執行這邊的內容。
- 一個很重要的觀念是，nachos 本身也是一個程式，所以現在執行到 nachos 的哪裡也需要記錄下來，這些記錄就保存在真實機器的 register，所以在做 context switch 的時候就是在交換這些 register 狀態，讓機器可以真的在 nachos 的程式裡跳來跳去。
- 首先前半部分要做的事是將 t1 thread 的 host machine register 的狀態存到 t1 的 machineState 中：
  - 把 4(%esp)存到%eax。4(%esp)代表「esp 指到的地址再加 4」的內容，也就是 t1，為什麼呢？回想一下 procedure call 的過程：
    - 當要呼叫一個函式時(在此為 SWITCH())，caller 會把函式的 parameters(在此為 Thread *t1、Thread *t2)以相反順序 push 到 stack，再把 return address(也就是現在執行到的地方，因為等下要再回來)push 進去，所以狀態會如下圖所示：

```
on entry, stack looks like this:
     8(esp)  ->              thread *t2
     4(esp)  ->              thread *t1
      (esp)  ->              return address
```

  - 所以此時 eax 指到的地址就是 t1 這個指標所在的地方，而 t1 指到的地方就是 t1 thread 這個 object，所以我們就可以對 eax 做 offset 去 access 到 t1 thread 裡面的成員。

- 因為這樣，所以在宣告 Thread 這個 class 的時候，有兩個 member：`int *stackTop` 和 `void *machineState[]` 一定要是頭兩個宣告，如此一來就可以透過 `0(%eax)` 來 access 到 `stackTop`，或是 `4(%eax)` 來 access 到 `machineState [0]`。

— 把所有 register 狀態(`eax`、`ebx`、`ecx`…)存到 `machineState`，最後是把 `0(%eax)` 也就是 return address 存到 `PCState`，代表當 t1 thread 又有 CPU 使用權的時候就要接著執行 program counter 裡的內容。

- 接著後半部分要做的事是將 t2 thread 的 `machineState` 載入到 host machine register 中：

— 一樣，把 `8(%esp)` 存到`%eax`，所以此時 `eax` 指到的地方就是 t2 thread 所在的地方。

— 把所有 `machineState` 狀態存到 register (`eax`、`ebx`、`ecx`…)，最後是把 `PCState` 裡的 return address 放到 `0(%esp)`，也就是 `esp` 指到的地指的內容(原本的程式是寫 `4(%esp)` 疑似有寫錯)。

- 接著執行 `ret`，它會把 `0(%esp)` 也就是 return address 放到 `eip` (program counter)，代表 t2 thread 要跳回到他在做 context switch 之前在做的事，所以對 t2 thread 來說 context switch 就像是做了一個 procedure call 而已。
- 要注意的是雖然這個 `SWITCH()` 是 t1 thread 呼叫的，但 `SWITCH()` 做完之後回到的地方不是剛剛 t1 thread 做到一半的地方(也就是 `Scheduler::Run` 裡面)，因為此時 host machine 是在執行 t2 thread 的內容。

```
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
```

- 回到 t2 thread，它的 return address 可能有幾種狀況：
  - t2 thread 剛執行到一半被打斷：
    - 這時候 return 會回到 `Scheduler::Run` 裡面的 `SWITCH()` 之後。
    - 此時先呼叫 `CheckToBeDestroyed()`，檢查看看是否有 thread 需要被 delete 掉(Terminate)。
    - 將自己的相關 states 都恢復原狀 (user program register、page table…)。
    - 接著再回到更上層的 `OneTick()`，再回到 `Machine::Run` 裡面的無限迴圈。
  - t2 thread 是第一次被執行到：
    - t2 thread 剛被建立起來，在 `StackAllocate()` 時會為 `machineState` 存入對應的函式，其中 `PCState` 會存入 `ThreadRoot` 函式，所以這時候 t2 thread 的 return address 就是 `ThreadRoot` 的位址，來看到在 `switch.s` 裡為 `ThreadRoot` 的定義：

- 可以看到 ThreadRoot 主要會 call 三個函式：
- 第一個是 StartupPC。StartupPC 就是 ecx 裡存的位址(可參考在 switch.h 中的定義)，而 ecx 存的位址又是來自 machineState[StartupPCState]裡存的函式，所以是 ThreadBegin()，ThreadBegin()會執行 Thread::Begin，先檢查有沒有要 destroy 的 thread，有的話就 delete 掉，再把 interrupt 做 enable。
- 第二個是 InitialPC。InitialPC 對應到 machineState[InitialPCState]裡存的函式，也就是 ForkExecute()，在裡面會做 Machine::Run 真正執行 user program 的內容。
- 第三個是 WhenDonePC。WhenDonePC 對應到 machineState[WhenDonePCState]裡存的函式，也就是 ThreadFinish()，ThreadFinish()會執行 Thread::Finish，來把 thread 結束掉。

```
ThreadRoot:
        pushl   %ebp
        movl    %esp,%ebp
        pushl   InitialArg
        call    *StartupPC
        call    *InitialPC
        call    *WhenDonePC

        # NOT REACHED
        movl    %ebp,%esp
        popl    %ebp
        ret
```

---

2. Implement other scheduling methods (FIFO, SJF, Priority).

thread.cc：

- 為了要驗證 scheduling 的正確性，在這個檔案下新增 self-test code，如此一來就不需要 run 實體的 process，只是單純跑 SchedulingTest()就可測試。
- 在此先定義了 6 個 thread，以及各自的 priority 與 burst time。而 threadBody()是每個 thread 被 scheduler 選到時會執行的內容。

```
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 = 6;
        char *name[thread_num] = {"A", "B", "C", "D", "E", "F"};
        int thread_priority[thread_num] = {4, 2, 6, 5, 1, 3};
        int thread_burst[thread_num] = {5, 8, 7, 2, 9, 3};

        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();
}
```

thread.h：

- 首先定義不同 scheduling methods 所需要的參數，burst time (for SJF), priority (for Priority)。

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

scheduler.h：

- 同樣在此定義 scheduling 所需要的參數。

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

class Scheduler {
  public:
        Scheduler();            // Initialize list of ready threads
        Scheduler(SchedulerType type);
        ~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
```

kernel.cc：

- 記得把 `SchedulingTest` 加到 `ThreadedKernel::SelfTest`，這樣在一開始創立 kernel 之後所執行的一系列 self test 裡就會執行到這個 scheduling test。

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

   LibSelfTest();               // test library routines

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

main.cc：

- 為了要能在一開始選定我們要的 scheduling method，所以新增 command line 參數`-st`。

```
  } else if (strcmp(argv[i], "-st") == 0) {
      if(strcmp(argv[i + 1], "FCFS") == 0) {
          type = FIFO;
      } else if (strcmp(argv[i + 1], "SJF") == 0) {
          type = SJF;
      } else if (strcmp(argv[i + 1], "PRIORITY") == 0) {
          type = Priority;
      } else if (strcmp(argv[i + 1], "RR") == 0) {
          type = RR;
      } else {
          cout << "Invalid scheduler type! Automatically set to be RR\n";
      }
  }
```

scheduler.cc：

- 為不同的 scheduling methods 定義各自的 ready list，並有各自 sort 的方法 (compare function)。

```
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;
}
```

Result：

- 首先測試 Round-Robin。可看到它是按照順序執行下來，且到了一定時間會 yield，故符合預期。

```
henry@henry-VirtualBox:~/nachos/mp3/nachos-4.0_mp3/code/userprog$ ./nachos -st RR
=== interrupt->YieldOnReturn ===
=== interrupt->YieldOnReturn ===
A: remaining 4
A: remaining 3
A: remaining 2
A: remaining 1
A: remaining 0
B: remaining 7
B: remaining 6
=== interrupt->YieldOnReturn ===
C: remaining 5
C: remaining 4
C: remaining 3
C: remaining 2
C: remaining 1
C: remaining 0
D: remaining 1
=== interrupt->YieldOnReturn ===
E: remaining 8
E: remaining 7
E: remaining 6
E: remaining 5
E: remaining 4
E: remaining 3
E: remaining 2
E: remaining 1
=== interrupt->YieldOnReturn ===
F: remaining 2
F: remaining 1
F: remaining 0
B: remaining 5
B: remaining 4
B: remaining 3
B: remaining 2
=== interrupt->YieldOnReturn ===
D: remaining 0
E: remaining 0
B: remaining 1
B: remaining 0
```

- 測試 FIFO (FCFS)。可看到它是按照順序執行下來並且不發生 preempt，全部執行完了才換下一條 thread，故符合預期。

```
henry@henry-VirtualBox:~/nachos/mp3/nachos-4.0_mp3/code/userprog$ ./nachos -st FCFS
A: remaining 4
A: remaining 3
A: remaining 2
A: remaining 1
A: remaining 0
B: remaining 7
B: remaining 6
B: remaining 5
B: remaining 4
B: remaining 3
B: remaining 2
B: remaining 1
B: remaining 0
C: remaining 5
C: remaining 4
C: remaining 3
C: remaining 2
C: remaining 1
C: remaining 0
D: remaining 1
D: remaining 0
E: remaining 8
E: remaining 7
E: remaining 6
E: remaining 5
E: remaining 4
E: remaining 3
E: remaining 2
E: remaining 1
E: remaining 0
F: remaining 2
F: remaining 1
F: remaining 0
```

- 測試 SJF。可看到它是按照 burst time 由小到大執行，故符合預期。

```
henry@henry-VirtualBox:~/nachos/mp3/nachos-4.0_mp3/code/userprog$ ./nachos -st SJF
D: remaining 1
D: remaining 0
F: remaining 2
F: remaining 1
F: remaining 0
A: remaining 4
A: remaining 3
A: remaining 2
A: remaining 1
A: remaining 0
C: remaining 5
C: remaining 4
C: remaining 3
C: remaining 2
C: remaining 1
C: remaining 0
B: remaining 7
B: remaining 6
B: remaining 5
B: remaining 4
B: remaining 3
B: remaining 2
B: remaining 1
B: remaining 0
E: remaining 8
E: remaining 7
E: remaining 6
E: remaining 5
E: remaining 4
E: remaining 3
E: remaining 2
E: remaining 1
E: remaining 0
```

- 測試 Priority。可看到它是按照 priority 由小到大執行，故符合預期。

```
E: remaining 8
E: remaining 7
E: remaining 6
=== interrupt->YieldOnReturn ===
E: remaining 5
E: remaining 4
E: remaining 3
E: remaining 2
E: remaining 1
E: remaining 0
Total threads number is 0
=== interrupt->YieldOnReturn ===
B: remaining 7
B: remaining 6
B: remaining 5
B: remaining 4
B: remaining 3
B: remaining 2
B: remaining 1
B: remaining 0
=== interrupt->YieldOnReturn ===
F: remaining 2
F: remaining 1
F: remaining 0
A: remaining 4
A: remaining 3
A: remaining 2
A: remaining 1
A: remaining 0
=== interrupt->YieldOnReturn ===
D: remaining 1
D: remaining 0
C: remaining 5
C: remaining 4
C: remaining 3
C: remaining 2
C: remaining 1
=== interrupt->YieldOnReturn ===
C: remaining 0
```