MP3_report_37

tags: 筆記， 作業系統， OS

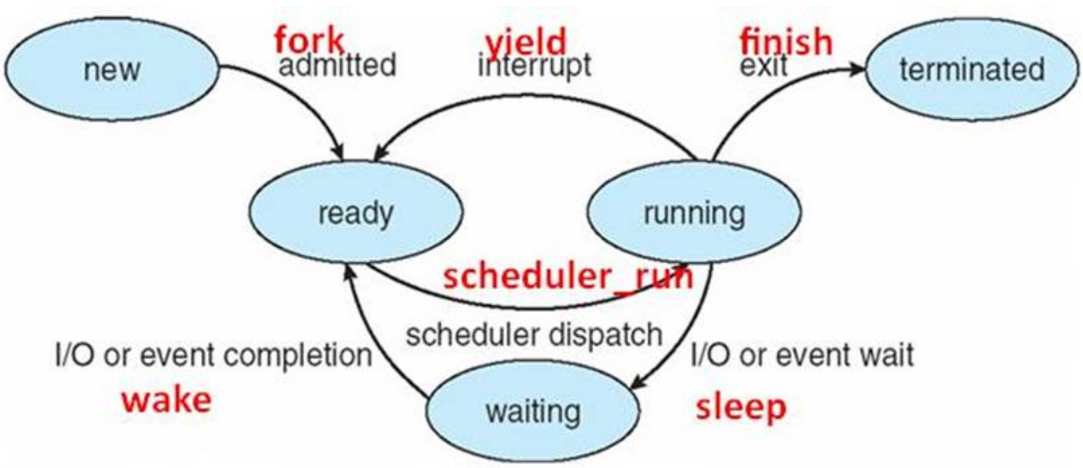
Team Member & Contributions

- # 資應碩二 **107065522** 陳子潔
- # 數學大四 **105021127** 徐迺茜

工作項目	分工
Trace Code	陳子潔 & 徐迺茜
報告撰寫	陳子潔 & 徐迺茜
功能實作	陳子潔
功能測試	徐迺茜

Trace code

先來張圖幫助理解:



1-1. New→Ready

前情提要:

- 主程式(main) **Bootstrap** the NachOS kernel
  - 主程式接收命令列參數 (int argc, **argv)，並利用strcmp做剖析
  - 做一些簡單的初始化 (DEBUG, XXXTest, XXXFlag...,etc)
  - 正式載入(宣告?)Kernel，並做許多初始化

```
.
.
kernel = new Kernel(argc, argv);
kernel->Initialize();
.
kernel->ExecAll();
.
```

Kernel::ExecAll()

```
void Kernel::ExecAll()
{
    // 1.
    for (int i=1; i<=execfileNum; i++) {
        int a = Exec(execfile[i]);
    }
    // 2.
    currentThread->Finish();
}
```

1. 此函數會依序執行 (**Exec**) 每一個檔案
    - 而"execfile"在kernel初始化的時候就會剖析終端機參數來決定:

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

2. 當所有的程式 (execfile) 順利執行 (Exec) 結束，currentThread (mainThread) 就能呼叫 Finish( ) 來結束NachOS了

Kernel::Exec(char*)

```
int Kernel::Exec(char* name)
{
   // 1.
   t[threadNum] = new Thread(name, threadNum);
   // 2.
   t[threadNum]->space = new AddrSpace();
   // 3.
   t[threadNum]->Fork((VoidFunctionPtr)&ForkExecute, (void *)t[threadNum]);
   threadNum++;
   return threadNum-1;
}
```

- 深入探討Exec，此函式大致上做三件事:

    1. new一個Thread Class(類似Thread Control Block)，給要執行的thread

        - 再往下追蹤的話，可看到Thread初始化行為如下:

        ```
        Thread::Thread(char* threadName, int threadID)
        {
            ID = threadID;
            name = threadName;
            stackTop = NULL;
            stack = NULL;
            status = JUST_CREATED;
            for (int i = 0; i < MachineStateSize; i++) {
                machineState[i] = NULL;
            }
            space = NULL;
        }
        ```

    2. 分配一個定址空間(Address Space)給剛創建的Thread，並做簡單初始化(配置PageTable、清空Memory)

        - 到這邊程式都還沒載入Memory，只有初始化而已

```
AddrSpace::AddrSpace()
{
    pageTable = new TranslationEntry[NumPhysPages];
    for (int i = 0; i < NumPhysPages; i++) {
        pageTable[i].virtualPage = i;
        pageTable[i].physicalPage = i;
        pageTable[i].valid = TRUE;
        pageTable[i].use = FALSE;
        pageTable[i].dirty = FALSE;
        pageTable[i].readOnly = FALSE;
    }

    // zero out the entire address space
    bzero(kernel->machine->mainMemory, MemorySize);
}
```

3. 透過t->Fork()函數的呼叫，完成Stack的配置與初始化，並將Program Load進Memory，注意**ForkExecute**這個function pointer，它會是將來Thread::Begin( )之後馬上執行的函式

    - **補充:** 新的Thread在拿到控制權後大致運作如下 (借助**SWITCH.s**的幫助來執行以下動作):
        1. Thread->Begin( )
        2. ForkExecute( )
            - Machine抓取Program Counter存放的指令address來Decode
            - ForkExecute又會大致做兩件事情:
                1. t->space->Load(t->getName())
                    - 將Program Load進Memory (順便做一些Virtual Memory相關的設定...)
                2. t->space->Execute(t->getName())
                    - 一些Registers跟Page Table相關設定...
                    - **Machine->Run( ) !!!**
                        - Infinite Loop
                            - OneInstruction(instr);
                            - OneTick();
                            - OneInstruction(instr);
                            - OneTick();
                            - ...
        3. Thread->Finish( )

Thread::Fork(VoidFunctionPtr, void*)

```
void
Thread::Fork(VoidFunctionPtr func, void *arg)
{
    // 1.
    Interrupt *interrupt = kernel->interrupt;
    Scheduler *scheduler = kernel->scheduler;
    IntStatus oldLevel;

    // 2.
    StackAllocate(func, arg);

    // 3.
    oldLevel = interrupt->SetLevel(IntOff);
    scheduler->ReadyToRun(this);
    (void) interrupt->SetLevel(oldLevel);
}
```

- Fork大致上的流程如下:

1. 為了要使用NachOS的interrupt與scheduler模組，先宣告2個指標

2. 呼叫StackAllocate來幫剛創立的thread配置Stack以及設置MachineState，值得注意的是，這邊接收的參數為 **(&ForkExecute, t[threadNum])**，運作細節於下一小節說明

3. StackAllocate執行結束，將Interrupt Disable，並呼叫scheduler將此Thread餵進Ready List，而ReadToRun的細節將於下下節說明

Thread::StackAllocate(VoidFunctionPtr, void*)

- 截錄StackAllocate的關鍵程式碼

```
void
Thread::StackAllocate(VoidFunctionPtr func, void *arg)
{
    // 1.
    stack = (int *) AllocBoundedArray(StackSize * sizeof(int));

    // 2.
    stackTop = stack + StackSize - 4;

    // 3.
    *(--stackTop) = (int) ThreadRoot;

    // 定義於thread.cc:21:const int STACK_FENCEPOST = 0xdedbeef;
    // 基本上就是Stack底部再往上一格的意思，防止不小心存取越界
    *stack = STACK_FENCEPOST;

    // 4.
    machineState[PCState] = (void*)ThreadRoot;
    machineState[StartupPCState] = (void*)ThreadBegin;
    machineState[InitialPCState] = (void*)func;
    machineState[InitialArgState] = (void*)arg;
    machineState[WhenDonePCState] = (void*)ThreadFinish;
}
```

- 這邊做的事情主要為

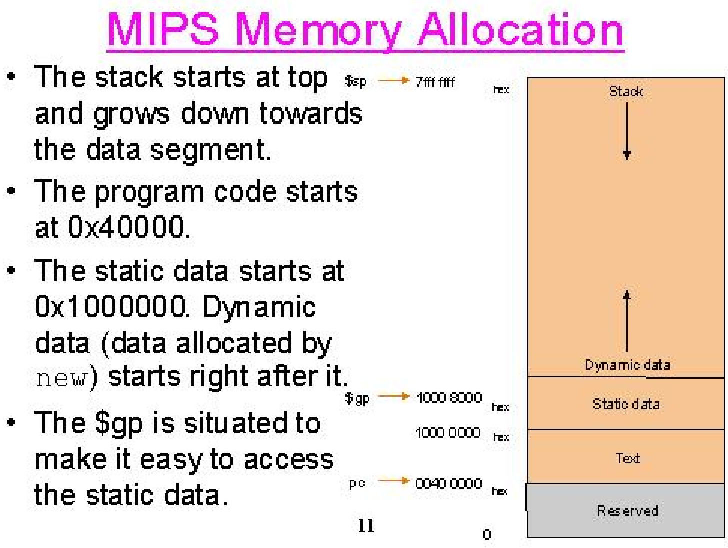
1. Alloc一個Array，並讓stack指標(有點像是Stack Frame)指向其頂部(Low Address)
2. 讓StackTop指向Stack的底部(High Address)，為了確保安全，多減一格(StackSize - 4)
3. 讓Stack裡面的第一個元素為ThreadRoot函式的Address (也許可以想成，將ThreadRoot函式Address Push進Stack)，以便將來x86組語做SWITCH的時候可以直接從Stack裡面取出ThreadRoot來執行(Call)
4. 設置MachineState，這邊是與**Switch.h & s**互相呼應 (由於NachOS是跑在Host上的虛擬機，這裡的MachineState應該有點類似於給Host用的Registers)
    - 註: 參考Thread.h內的註解

      > A thread running a user program actually has **two** sets of CPU registers
      >
      > one for its state while executing **user code**, one for its state while executing **kernel code**.

5. 值得注意的是，這裡的func以及arg其實就是之前傳進來的 **(&ForkExecute, t[threadNum])** 於將來程式Run的時候會再做執行

```
* machineState[InitialPCState] = (void*)func;
* machineState[InitialArgState] = (void*)arg;
```

- 補充: stackTop = stack + StackSize - 4，此行是因為在MIPS架構中，Stack是由High Address長到Low Address的



Scheduler::ReadyToRun(Thread*)

- 當Fork函數進行完StackAllocate後，會先Disable Interrupt，接著呼叫此函式，將剛配置好的Thread餵進readyList

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

1-2. Running→Ready

說明:

- 通常從Run -> Ready可能是有一些Interrupt發生(Time Slice到了、或者被更高優先權的Process搶奪CPU...等)
- 而NachOS利用Machine::Run以及Interrupt::OneTick來模擬User Program於MIPS架構中的每一個Clock的執行過程
- 簡單來說，Thread 1若要從Run -> Ready給Thread 2執行，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 (這函式的組成挺複雜的...，總之**Context Switch**在此進行)

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

```
void
Machine::Run()
{
    Instruction *instr = new Instruction;
    kernel->interrupt->setStatus(UserMode);
    for (;;) {
      OneInstruction(instr);
      kernel->interrupt->OneTick();
    }
}
```

- Test Program基本上都是在UserMode下執行的，有需要用到SysCall的話才會切換Mode

- 可看出其實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

- 以Run -> Ready來說，可能情況有二:

    1. 程式執行到一半，出於某些原因主動Yield(讓出)控制權:
        - OneInstruction(instr) Decode後發現是一個System Call "**SC_ThreadYield**" (定義於syscall.h)，要求Thread轉移控制權
        - 於是RaiseException()
            - 轉移到SystemMode
            - 呼叫ExceptionHandler
            - Exception Handler運作細節參見MP1，並在其中呼叫ThreadYield()此一Syscall來轉移控制權

- **然而我們發現NachOS並沒有實作ThreadYield的System Call...，所以可能之情況為另一種...**

    2. **Hardware Timer**定期發出一個**Interrupt**來呼叫**"YieldOnReturn()"**函式
        - 此函式會將"**y**ieldOnReturn"設為True
        - 將來OneTick執行看到yieldOnReturn Flag為True，就會去執行Yiled(最終目標是做Context Switch來讓Thread 2順利執行)
        - 回憶: Alarm 的 CallBack( )會呼叫interrupt->YieldOnReturn()

- 至於Hardware Timer是如何定期 (Every **TimerTicks**) Timer Interrupt，過程挺複雜的，詳細參見Alarm以及Timer兩份檔案...

Interrupt::OneTick()

```
void
Interrupt::OneTick()
{
    // 1.
    if (status == SystemMode) {
        stats->totalTicks += SystemTick;
        stats->systemTicks += SystemTick;
    }
    else {
        stats->totalTicks += UserTick;
        stats->userTicks += UserTick;
    }

    // 2.
    ChangeLevel(IntOn, IntOff);
    CheckIfDue(FALSE);
    ChangeLevel(IntOff, IntOn);

    // 3.
    if (yieldOnReturn) {
        yieldOnReturn = FALSE;
        status = SystemMode;
        kernel->currentThread->Yield();
        status = oldStatus;
    }
}
```

- 對應程式碼中的標記，OneTick這邊主要做三件事:

1. 遞增stats裡面所記錄的totalTicks，順便判斷是在SystemMode還是UserMode，增加對應的執行Ticks
2. Disable Interrupt (確保下一條指令執行是Atomic的)
    - **CheckIfDue**會檢查是否有下一條已經到期的pending Interrupt，並執行它
    - 關鍵程式碼:

```
inHandler = TRUE;

do {
    next = pending->RemoveFront();
    next->callOnInterrupt->CallBack();
    delete next;
}while(!pending->IsEmpty()&&(pending->Front()->when<=stats->totalTicks))
```

- 這裡面的**next** Interrupt其實就是YieldOnReturn( )，會將yieldOnReturn此Flag設置為True

3. 執行完CheckIfDue後，yield的Flag被設置為True，進入迴圈
    - yieldOnReturn必須先恢復False (不然下一個Thread如果看到Flag為True，可能會有BUG)
    - 這邊切換到SystemMode，並執行Yield()，細節見下一節
    - Thread2執行完畢後回來，切換為oldStatus (通常是切回UserMode)，因為之後又要回到Machine::Run()的迴圈內了

Thread::Yield()

- 承上，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)

```
void
Thread::Yield ()
{
    Thread *nextThread;
    // 1
    IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);

    ASSERT(this == kernel->currentThread);

    // 2
    nextThread = kernel->scheduler->FindNextToRun();
    if (nextThread != NULL) {
        kernel->scheduler->ReadyToRun(this);
        kernel->scheduler->Run(nextThread, FALSE);
    }
    // 3
    (void) kernel->interrupt->SetLevel(oldLevel);
}
```

1. 首先Disable Interrupt (Yield的過程不容許打斷)
2. 排班器(scheduler)從readyList找出nextThread
    - 將目前執行的Thread放回readyList (run -> ready)
    - 運行(Run) nextThread
3. 將Interrupt Level恢復原本 (通常是Enable Interrupt)

Scheduler::FindNextToRun()

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

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

- 檢查readList是否為空，否的話就DeQueue並Return下一條 (Front) Thread

Scheduler::ReadyToRun(Thread*)

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

- 將準備要執行的Thread的Status設置為Ready，並放入readyList

Scheduler::Run(Thread*, bool)

```
void
Scheduler::Run (Thread *nextThread, bool finishing)
{
    // 0.
    if (finishing) {
        ASSERT(toBeDestroyed == NULL);
        toBeDestroyed = oldThread;
    }

    // 1.
    if (oldThread->space != NULL) {
        oldThread->SaveUserState();
        oldThread->space->SaveState();
    }

    // 2.
    oldThread->CheckOverflow();

    // 3.
    kernel->currentThread = nextThread;
    nextThread->setStatus(RUNNING);
    SWITCH(oldThread, nextThread);

    // 4.
    CheckToBeDestroyed();

    // 5.
    if (oldThread->space != NULL) {
        oldThread->RestoreUserState();
        oldThread->space->RestoreState();
    }
}
```

- Run基本上就是執行下一條Thread，而Context Switch在此進行，步驟大致可分成

    0. 如果finishing此一參數為True，代表上一個Thread已經執行完成了，此時讓toBeDestroyed指向oldThread(上一個Thread)

    1. 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)

        ```
        void
        Thread::SaveUserState()
        {
            for (int i = 0; i < NumTotalRegs; i++)
                userRegisters[i] = kernel->machine->ReadRegister(i);
        }
        ```

        - 接著還會保存Address Space的State (但其實NachOS在這邊尚未實現此功能，目前也用不著)

        ```
        void AddrSpace::SaveState()
        {}
        ```

    2. Check a thread's stack to see if it has overrun the space that has been allocated for it.

```
void
Thread::CheckOverflow()
{
    if (stack != NULL) {
        ASSERT(*stack == STACK_FENCEPOST);
   }
}
```

3. 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()進行線程切換
    - 值得注意的是，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 (我的電腦是Intel x86架構，故組語部分也是執行x86組語)，詳細過程見1-6節
4. 當程式執行到此，表示又Switch回原本的Thread 1了
    - 此時先呼叫CheckToBeDestroyed()，檢查看看是否有Thread需要被Delete掉(Terminate)
    - 這是因為在NachOS中，Thread執行完畢後不能自己Delete自己(因為自己正在使用自己)，故需依靠下一個Thread來Delete自己
    - CheckToBeDestroyed的程式碼，就是delete而已
        - 有趣的是，delete完一個Thread之後，要將指標設置為NULL
        - 這點是出於資訊安全的考量 (Keywords: Dangling Pointer, Double Free)

    ```
    void
    Scheduler::CheckToBeDestroyed()
    {
        if (toBeDestroyed != NULL) {
            delete toBeDestroyed;
            toBeDestroyed = NULL;
        }
    }
    ```

5. 最後一步，將oldThread的相關states都恢復原狀 (userRegisters, AddressSpace的PageTable...)

    Code:

```
void
Thread::RestoreUserState()
{
    for (int i = 0; i < NumTotalRegs; i++)
        kernel->machine->WriteRegister(i, userRegisters[i]);
}
```

```
void AddrSpace::RestoreState()
{
    kernel->machine->pageTable = pageTable;
    kernel->machine->pageTableSize = numPages;
}
```

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

說明:

- 以NachOS來說，Running -> Waiting通常是因為I/O之類的Interrupt發生，透過Sleep()函式來Block掉某個Running Thread

補充

- **synchconsole.h**基本上就是用來處理I/O Ouput同步問題的介面

- Data structures for **synchronized access** to the keyboard and console display devices

- 在 **kernel->initialize()** 時
  - consoleIn / Out的預設值為NULL (代表stdin跟stdout)，作為參數傳入SynchConsoleInput / Output Class內

```
synchConsoleIn = new SynchConsoleInput(consoleIn);
synchConsoleOut = new SynchConsoleOutput(consoleOut);
```

- 而**synchConsoleOut**裡面其實又包含了**ConsoleOutput** (定義於console.h)
  - 這邊可以清楚的看到，Lock跟Semaphore (**waitFor**) 的宣告

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

- 承上，再更深入追蹤**ConsoleOutput**建構子
  - 可發現console(stdout)的"toCall"其實就是指**synchConsoleOut**的**CallBackObj**
  - 注意callWhenDone = toCall，此行將SynchConsoleOutput 和 ConsoleOutput之間緊密的牽連在一起了
  - 有點像是說ConsoleOutput每成功put一個char或int，就呼叫SynchConsoleOutput來進行stdout的同步顯示

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

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

  - SynchConsoleOutput的CallBack:

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

這邊使用 **.../build.linux/nachos -C** 指令進行Console測試可能會比較清楚

- NachOS在測試Console Input/Output的程式碼:

```
do {
    ch = synchConsoleIn->GetChar();
    if(ch != EOF) synchConsoleOut->PutChar(ch);   // echo it!
} while (ch != EOF);
```

SynchConsoleOutput::PutChar(char)

承上述補充，我們從Console GetChar，並呼叫SynchConsoleOutput::PutChar

```
void
SynchConsoleOutput::PutChar(char ch)
{
    // 1.
    lock->Acquire();

    // 2.
    consoleOutput->PutChar(ch);

    // 3.
    waitFor->P();
    lock->Release();
}
```

1. 由於console(stdout)為一個互斥存取的物件(不能同時有兩個Thread在做輸出)，故先lock->Acquire()
    - Lock定義於synch.h內
    - 可看出Lock的最底層其實是用semaphore來實現

    ```
    Lock::Lock(char* debugName)
    {
        name = debugName;
        // initially, unlocked
        semaphore = new Semaphore("lock", 1);
        lockHolder = NULL;
    }
    ```

    - 而Lock->Acquire()其實就是讓CurrentThread持有這個Lock

    ```
    void Lock::Acquire()
    {
        semaphore->P();
        lockHolder = kernel->currentThread;
    }
    ```

2. consoleOutput->PutChar

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

    - 注意，**writeFileNo**在初始化的時候已經被設成 **1 (stdout)** 了，故WriteFile會將 1 個char寫上stdout (或者也可以事先透過 -co 指令來設定要寫到哪啦...)
    - putBusy設為True代表正在putChar，如有其他Thread想同時輸出，ASSERT(putBusy == FALSE)就會報錯
    - PutChar的最後會將ConsoleOutput本身餵進去interrupt pending list (之後會執行CallBack)
        - **ConsoleTime**在本次作業被設定為 1 ， 即下一個tick就會發生 “console write” Interrupt
    - 當ConsoleTime (1 tick)過去，console write interrupt發生，執行ConsoleOutput->Callback( )

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

    - 此時可看到，這個CallBack裡面又呼叫了callWhenDone->CallBack( )，而先前提過，**callWhenDone就是SynchConsoleOutput**

- 再追蹤**callWhenDone->CallBack( )**，可以發現這邊其實只做了waitFor->V()這個動作 (waitFor是一個Semaphore Class)

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

3. 承上，由於putChar完成後，console write intterupt發生，最後讓semaphore++ (因為waitFor->V( )的關係)
   - 這邊的 waitFor->P( ); 將會成功執行，讓semaphore - - ，運作細節見下一小節
   - 做到此代表putChar( )程序全部完成，呼叫lock->Release()來釋放lock

Semaphore::P()

承上，putChar的最後呼叫了此函式

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

    // 1.
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    // 2
    while (value == 0) {
        queue->Append(currentThread);  // so go to sleep
        currentThread->Sleep(FALSE);
    }

    // semaphore available, consume its value
    value--;

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

1. 這邊先Disable Interrupt
2. 然後檢查semaphore value是否 > 0，若無，while(value==0)成立
   - 將等待semaphore的currentThread Append 進 queue裡
   - 這邊的queue定義於sync.h的Semaphore Class中

> // threads waiting in P( ) for the value to be > 0
>
> List<Thread *> *queue;

SynchList<T>::Append(T)

- List的Appdend定義以及實作於list.h & cc這兩個檔案內
- 由下列C++ template可以看出，其實就是寫的很厲害的Single Linked List

```
template <class T>
void
List<T>::Append(T item)
{
    ListElement<T> *element = new ListElement<T>(item);

    ASSERT(!IsInList(item));

    if (IsEmpty()) {
        first = element;
        last = element;
    }
    // else put it after last
    else {
        last->next = element;
        last = element;
    }

    numInList++;
    ASSERT(IsInList(item));
}
```

Thread::Sleep(bool)

- 回想PutChar的情境，若Thread 遇到 Semaphore Value == 0的情形，表示目前某個資源(stdout?)正有人要互斥存取

```
while (value == 0) {
    queue->Append(currentThread);     // so go to sleep
    currentThread->Sleep(FALSE);
}
```

1. 將currentThread Append進Semaphore的waiting queue裡面
2. 呼叫Sleep函式

- 深入追蹤Thread::Sleep

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

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

    // 1.
    status = BLOCKED;

    // 2.
    while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
        kernel->interrupt->Idle();
    }
    kernel->scheduler->Run(nextThread, finishing);
}
```

1. 簡單來說，就是讓等待Semaphore的currentThread變成Blocked的status
2. 接著scheduler從readyList找出下一條要執行的Thread
    - 若無(NULL)，則呼叫Idle( )，裡面會判斷是否該Advance Clock或者直接Halt程式
    - 若有(nextThread)，則呼叫scheduler->Run讓nextThread執行

Scheduler::FindNextToRun()

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

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

- 承上，while迴圈中的FindNextToRun運作很簡單，就是一個DeQueue的動作，會Return要執行的Thread的指標

Scheduler::Run(Thread*, bool)

- 這邊的Run運作基本上與 1 - 2節一模一樣，故不再重複
- 值得注意的是，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

```
void
Scheduler::Run (Thread *nextThread, bool finishing)
{
    // 0.
    if (finishing) {
        ASSERT(toBeDestroyed == NULL);
        toBeDestroyed = oldThread;
    }

    // 1.
    if (oldThread->space != NULL) {
        oldThread->SaveUserState();
        oldThread->space->SaveState();
    }

    // 2.
    oldThread->CheckOverflow();

    // 3.
    kernel->currentThread = nextThread;
    nextThread->setStatus(RUNNING);
    SWITCH(oldThread, nextThread);

    // 4.
    CheckToBeDestroyed();
    // 5.
    if (oldThread->space != NULL) {
        oldThread->RestoreUserState();
        oldThread->space->RestoreState();
    }
}
```

- Run基本上就是執行下一條指令，而Context Switch在此進行，步驟大致可分成

    0. 如果finishing此一參數為True，代表上一個Thread已經執行完成了，此時讓toBeDestroyed指向oldThread(上一個Thread)

    1. 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)

       Code:

       ```
       void
       Thread::SaveUserState()
       {
           for (int i = 0; i < NumTotalRegs; i++)
               userRegisters[i] = kernel->machine->ReadRegister(i);
       }
       ```

- 接著還會保存Address Space的State (但其實NachOS在這邊尚未實現此功能，目前也用不著)

```
void AddrSpace::SaveState()
{}
```

2. Check a thread’s stack to see if it has overrun the space that has been allocated for it.

```
void
Thread::CheckOverflow()
{
    if (stack != NULL) {
        ASSERT(*stack == STACK_FENCEPOST);
   }
}
```

3. 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()進行線程切換
   - 值得注意的是，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 (我的電腦是Intel x86架構，故組語部分也是執行x86組語)，詳細過程見1-6節
4. 當程式執行到此，表示又Switch回原本的Thread 1了
   - 此時先呼叫CheckToBeDestroyed()，檢查看看是否有Thread需要被Delete掉(Terminate)
   - 這是因為在NachOS中，Thread執行完畢後不能自己Delete自己(因為自己正在使用自己)，故需依靠下一個Thread來Delete自己
   - CheckToBeDestroyed的程式碼，就是delete而已，有趣的是，delete完一個Thread之後，要將指標設置為NULL，這點是出於資訊安全的考量 (Keywords: Dangling Pointer, Double Free)

```
void
Scheduler::CheckToBeDestroyed()
{
    if (toBeDestroyed != NULL) {
        delete toBeDestroyed;
        toBeDestroyed = NULL;
    }
}
```

5. 最後一步，將oldThread的相關states都恢復原狀 (userRegisters, AddressSpace的PageTable...)

   Code:

```
void
Thread::RestoreUserState()
{
    for (int i = 0; i < NumTotalRegs; i++)
        kernel->machine->WriteRegister(i, userRegisters[i]);
}
```

```
void AddrSpace::RestoreState()
{
    kernel->machine->pageTable = pageTable;
    kernel->machine->pageTableSize = numPages;
}
```

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

說明:

```
void SynchConsoleOutput::PutChar(char ch)
{
    ...
    // 2.
    consoleOutput->PutChar(ch);
    // 3.
    waitFor->P();
    ...
}
```

- 回想上一節，SynchConsoleOutput::PutChar的例子...
  - 步驟 2 時，consoleOutput作PutChar(ch)
  - 而PutChar最後完成的時候 (WriteFile to Stdout完成)，會再繞一大圈去執行這個CallBack

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

- 經過了waitFor->V( )的呼叫，semaphore value++，步驟 3 的waitFor->P( )才能順利進行下去而不被卡住

Semaphore::V()

- 這邊要注意的是，Interrupt已經在呼叫V( )之前就被Disable了，確保Semaphore的操作是Atomic的

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

    // 1.
    if (!queue->IsEmpty()) {
        kernel->scheduler->ReadyToRun(queue->RemoveFront());
    }

    // 2.
    value++;

    // 3.
    (void) interrupt->SetLevel(oldLevel);
}
```

1. 這邊會檢查 Semaphore的 " List<Thread *> *queue " ，並從裡面DeQueue，找出準備從BLOCKED狀態變到READY的Thread

2. 找出來後，簡單的把semaphore value++ (有點類似釋放這個LOCK讓別人用的意思)

3. 還原interrupt Level (通常是Enable)

Scheduler::ReadyToRun(Thread*)

- ReadyToRun的運作與 1 - 2節一樣，故直接貼上

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

1. 將準備要執行的Thread的Status設置為Ready
2. 放入readyList

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

說明:

- 通常Thread從Run -> Terminated，代表他已經執行完所有的Code了，資源可以被釋放了
  - NachOS裡面的Thread不能自己Delete自己 (見1 - 3節的Run)
  - 故需要透過下一條Thread的幫忙來Delete前一條Thread
    - **currentThread**會去呼叫Finish( )
    - Finish( )裡面再度呼叫Sleep(True)
      - currentThread在此被設置為**BLOCKED**狀態
      - 執行 scheduler->Run(nextThread, True);
      - Run函式會把舊的(已完成)Thread Delete掉，並讓下一條Thread能執行
  - 若沒有下一條Thread呢? (最後一條Thread執行完畢時)
    - 代表所有User Program運作完畢，**Machine Idel**中
    - 在Sleep( )函式中
      - 透過interrupt->Idle( )呼叫Halt( )來Terminate 整個NachOS

ExceptionHandler(ExceptionType) case SC_Exit

- 這邊可能的狀況之一為，執行到 " Return 0 ; "
- OneInstruction Decode完後，RaiseException(systemCall)
- 回想MP1: 經由start.s與syscall.h的幫助，控制權來到ExceptionHandler
- 執行**SC_Exit system call**
  - 其實就是作這一條指令:

    > kernel->currentThread->Finish();

Thread::Finish()

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

    Sleep(TRUE); // invokes SWITCH

    // not reached
}
```

- Finish裡面再度呼叫Sleep
- 值得注意的是，Sleep的參數為True，最後會再被接力傳進Run( )裡面，代表Thread Finishing

Thread::Sleep(bool)

- 這邊的敘述同 1 - 3節，故直接複製

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

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

    // 1.
    status = BLOCKED;

    // 2.
    while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
        kernel->interrupt->Idle();
    }
    kernel->scheduler->Run(nextThread, finishing);
}
```

1. 簡單來說，就是讓currentThread變成Blocked的status
2. 接著scheduler從readyList找出下一條要執行的Thread
    - 若無(NULL)，則呼叫**Idle( )**
        - Idle裡面會判斷是否該Advance Clock或者直接Halt程式
    - 若有(nextThread)，則呼叫**scheduler->Run**讓nextThread執行
        - 要注意的是，在SC_EXIT這個例子中，這邊的bool finishing參數為True

Scheduler::FindNextToRun()

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

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

- 承上，while迴圈中的FindNextToRun運作很簡單，就是一個DeQueue的動作，會Return要執行的Thread的指標

Scheduler::Run(Thread*, bool)

- 這邊的Run運作基本上與 1 - 2節類似
- 值得注意的是，這裡面收到的finishing參數為**True**，因為上一個Thread呼叫了finish( )

```
void
Scheduler::Run (Thread *nextThread, bool finishing)
{
    // 0.
    if (finishing) {
        ASSERT(toBeDestroyed == NULL);
        toBeDestroyed = oldThread;
    }

    // 1.
    if (oldThread->space != NULL) {
        oldThread->SaveUserState();
        oldThread->space->SaveState();
    }

    // 2.
    oldThread->CheckOverflow();

    // 3.
    kernel->currentThread = nextThread;
    nextThread->setStatus(RUNNING);
    SWITCH(oldThread, nextThread);

    // 4.
    CheckToBeDestroyed();
    // 5.
    if (oldThread->space != NULL) {
        oldThread->RestoreUserState();
        oldThread->space->RestoreState();
    }
}
```

- Run基本上就是執行下一條指令，而Context Switch在此進行，步驟大致可分成

    0. finishing參數為True，代表上一個Thread已經執行完成了，此時讓toBeDestroyed指向oldThread(上一個Thread)

    1. 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)

Code:

```
void
Thread::SaveUserState()
{
    for (int i = 0; i < NumTotalRegs; i++)
        userRegisters[i] = kernel->machine->ReadRegister(i);
}
```

- 接著還會保存Address Space的State (NachOS在這邊尚未實現此功能)

```
void AddrSpace::SaveState()
{}
```

2. Check a thread's stack to see if it has overrun the space that has been allocated for it.

```
void
Thread::CheckOverflow()
{
    if (stack != NULL) {
        ASSERT(*stack == STACK_FENCEPOST);
    }
}
```

3. 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()進行線程切換
   - 值得注意的是，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 (我的電腦是Intel x86架構，故組語部分也是執行x86組語)，詳細過程見1-6節
4. 當程式執行到此，表示又Switch回原本的Thread 1了
   - 此時先呼叫CheckToBeDestroyed()，檢查看看是否有Thread需要被Delete掉(Terminate)
   - 因為先前finishing參數被設置為True，故toBeDestroyed的值 != NULL
   - 接下來直接將toBeDestroyed delete掉

```
void
Scheduler::CheckToBeDestroyed()
{
    if (toBeDestroyed != NULL) {
        delete toBeDestroyed;
        toBeDestroyed = NULL;
    }
}
```

5. 因為oldThread已經在第 4 步被Delete了，故If判斷式不成立，接下來就沒事了

1-6. Ready→Running

Scheduler::FindNextToRun()

Scheduler::Run(Thread*, bool)

- 上述兩個函式完全與前面一樣，故不再重複
- 這邊比較重要的是，當nextThread的status被設置為Running後，接下來馬上就會作**Machine Dependent**的**Context Switch**

SWITCH(Thread*, Thread*)

- SWITCH主要是透過**switch.h**的輔助 (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，
  - 實驗室的Host Server應當屬於**x86**架構，以下針對其作詳細探討

1. **switch.h** (define macro)

```
#ifdef x86
#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

#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
```

- 這邊宣告一些register的位置，為了讓switch.s取用

2. **thread.h** 外部宣告(extern)

```
extern "C" {
    void ThreadRoot();
    void SWITCH(Thread *oldThread, Thread *newThread);
}
```

- scheduler::Run裡面會呼叫這邊定義的SWITCH
- 透過extern的宣告以及compiler的輔助，使得x86組語能夠與C語言互相呼叫

3. **switch.s** 實作細節

```
#include "switch.h"
#ifdef x86
        .text
        .align  2
        .globl  ThreadRoot
        .globl  _ThreadRoot
_ThreadRoot:
ThreadRoot:
        pushl   %ebp
        movl    %esp,%ebp
        pushl   InitialArg
        call    *StartupPC
        call    *InitialPC
        call    *WhenDonePC

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


        .comm   _eax_save,4
        .globl  SWITCH
        .globl  _SWITCH
_SWITCH:
SWITCH:
        movl    %eax,_eax_save
        movl    4(%esp),%eax
        movl    %ebx,_EBX(%eax)
        movl    %ecx,_ECX(%eax)
        movl    %edx,_EDX(%eax)
        movl    %esi,_ESI(%eax)
        movl    %edi,_EDI(%eax)
        movl    %ebp,_EBP(%eax)
        movl    %esp,_ESP(%eax)
        movl    _eax_save,%ebx
        movl    %ebx,_EAX(%eax)
        movl    0(%esp),%ebx
        movl    %ebx,_PC(%eax)

        movl    8(%esp),%eax

        movl    _EAX(%eax),%ebx
        movl    %ebx,_eax_save
        movl    _EBX(%eax),%ebx
        movl    _ECX(%eax),%ecx
        movl    _EDX(%eax),%edx
        movl    _ESI(%eax),%esi
        movl    _EDI(%eax),%edi
        movl    _EBP(%eax),%ebp
        movl    _ESP(%eax),%esp
        movl    _PC(%eax),%eax
        movl    %eax,4(%esp)
        movl    _eax_save,%eax

        ret
#endif // x86
```

解釋:

- 在C語言的StackAllocate中，我們已經將未來要執行的函式的address放進Host CPU所對應的registers裡面了
    - 注意，這邊的PCState裡面存的是ThreadRoot的function pointer

```
void Thread::StackAllocate (VoidFunctionPtr func, void *arg)
{
    ...
#ifdef x86
    machineState[PCState] = (void*)ThreadRoot;
    machineState[StartupPCState] = (void*)ThreadBegin;
    machineState[InitialPCState] = (void*)func;
    machineState[InitialArgState] = (void*)arg;
    machineState[WhenDonePCState] = (void*)ThreadFinish;
#endif
}
```

- 於是我們在組語中所看到的register value分別代表:
    - ecx: points to startup function
        - 對應到C的(void*)ThreadBegin (interrupt enable)
        - 裡面會做kernel->interrupt->Enable();
    - edx: contains inital argument to thread function
        - 對應到C的(void*)arg;
    - esi: points to thread function
        - 對應到C的(void*)func (其實就是ForkExecute))
    - edi: point to Thread::Finish()
        - 對應到C的(void*)ThreadFinish
    - **esp** (組語執行到最後，esp裡面會存放新Thread的PCState的值)
        - 對應到C的(void*)ThreadRoot;
- 我們在scheduler::Run( )中呼叫SWITCH:
    - 剛切換到組語時，stack內存放的值如下:

```
**      8(esp)  ->    thread *t2
**      4(esp)  ->    thread *t1
**      0(esp)  ->    return address
```

- 接著做的事情可簡單分為以下
    1. 將 t1 (舊thread) 的所有相關 registers 保存起來 (需配置一塊空間於Memory)
    2. 將 t2 (新thread) 的所有相關 registers從Memory裡面的對應位置 Load 進 CPU registers 裡面
        - 回憶: Switch.h所定義的address offset
    3. ret
        - set CPU program counter to the memory address pointed by the value of register **esp**
        - 將來程式會抓取 esp 裡面所存放的位置來執行 (ThreadRoot)
        - ThreadRoot主要作三件事情:
            1. call *StartupPC
                - Thread::ThreadBegin( )
            2. call *InitialPC
                - Kernel::ForkExecute( )
            3. call *WhenDonePC
                - Thread::ThreadFinish ( )

(depends on the previous process state)

- 當執行到這邊時，表示控制權又回到Thread 1這邊了
    - 可能是Thread 2那邊又作了一個Context Switch回來
    - 或者它成功finish了，
    - 或它等待I/O所以被BLOCKED了
- 注意，目前Program Counter記載的指令仍然在 **Run( )** 函式內
- 而此時Thread 1的Case可能有幾種情形:

1. Thread 1 (Old Thread)原先是**BLOCKED** (waiting) status，然後Run的**finishing**參數為**True**

        - 下一行的CheckToBeDestroyed( )會將Thread 1 Delete掉
        - Run執行完畢後Return
        - 回想 1 - 1節，Kernel::ExecAll( )繼續執行下一個execFile (User Programs)

    2. Thread 1 原先是 **BLOCKED**，但**finished**參數為**False**

        - 恢復舊有的UserRegisters Set與Address Space的States (PageTable之類的)
        - 因為程式原先為BLOCK，此時會繼續檢查當初BLCOK的event是否滿足 (比如說正在 waitFor->P( ))
            - 若不滿足 (e.g., Semaphore == 0)
                - 繼續在Blocked狀態
                - 此時CPU scheduler會再從readyList找出下一個Thread來執行
            - 若滿足 (e.g., Semaphore > 0)
                - 需要Wake up這個BLOCKED thread (將其狀態從BLOCKED改為READY並放入 readyList)
                - Wake up的細節有點複雜，牽扯到了各種Synchronization機制，可參考synch.h & cc以及synchconsole.h & cc，會比較清楚一點

    3. Thread 1 原先就是 **Ready**狀態 (比如說RR排班，被SWITCH回來)

        - 情境: Thread 1 接收到一個timer Interrupt，裡面的CallBack是"YieldOnReturn"，逼其 Yield (最後目標是Context Switch)
        - Thread 1變成Ready Status
        - Context Switch到Thread 2
        - Thread 2 執行完(finish)、或者執行到一半(yield 或 sleep)，轉讓CPU控制權出來
        - 回到了Thread 1，恢復原本的Registers set
        - Scheduler::Run執行完畢
        - 返回Interrupt::OneTick()
        - 返回到Machine::Run()的迴圈中
            - CPU繼續抓(**OneInstruction**) Thread 1的下一條指令 (在Program Counter裡)，並透過**OneTick**來模擬執行

    4. 至於Thread 1原先是**New**或**Zombie** status呢?

        - 應該是不太可能有這種情況發生啦...

for loop in Machine::Run()

```
void
Machine::Run()
{
    Instruction *instr = new Instruction;
    kernel->interrupt->setStatus(UserMode);
    for (;;) {
      OneInstruction(instr);
      kernel->interrupt->OneTick();
    }
}
```

- Test Program基本上都是在UserMode下執行的
- 可看出其實就是用一個無窮迴圈反覆抓取User Program的程式碼並Decode (透過 OneInstruction)，然後用OneTick來模擬每個Clock的執行

綜合上述例子總結:

Machine::Run()就是在模擬MIPS架構CPU的每一條指令每一個Tick的執行，而有兩種情形Thread會轉移CPU使用權

1. 被動式，OneTick裡面的CheckIfDue函式發現有Interrput

    - 而Interrupt裡面的CallBack就是YieldOnReturn
        - yieldOnReturn這個flag被設置為True
    - 回到OneTick，偵測到yieldOnReturn為True
        - 切換到SystemMode，並執行kernel->currentThread->Yield( );

        - Yield( )裡面會呼叫
            - scheduler->ReadyToRun(this) 及
            - scheduler->Run(nextThread, FALSE);
        - 而scheduler->Run裡面會與x86組語搭配，執行SWITCH
            - 控制權轉讓到Thread 2
            - Thread 2執行完畢後，回到scheduler->Run後面的指令 (SWITCH之後的程式碼)
            - 恢復Thread 1的states，RETURN

2. 主動式

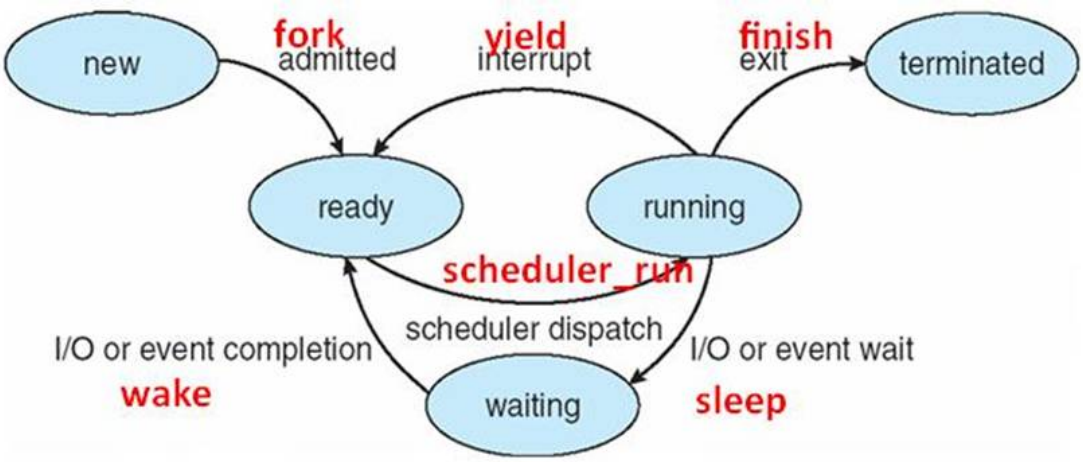
    1. Thread執行到一半後因為某些原因，需要等待I/O event之類的

        - 例如waitFor->P( )
        - event尚未發生，先去sleep
        - 此時thread被BLOCKED了，讓出控制權

    2. Thread的程式碼順利執行完畢，return

        - raiseException(SC_EXIT system call)
            - exceptionHandler 偵測到case SC_EXIT，呼叫kernel->currentThread->Finish();
        - finish裡面呼叫sleep(True)
            - sleep裡面將舊Thread設置為BLOCKED狀態，並呼叫scheduler::Run(nextThread, True)
        - Run執行
            - 偵測到finishing flag為True
            - toBeDestroyed指標指向舊Thread
            - scheduler->Run偵測到toBeDestroyed != NULL
            - delete前一個Thread

- 有趣的是，Thread執行結束後，是在**BLOCKED**的status下被delete掉的，而NachOS裡面似乎尚未用到ZOMBIE以及TERMINATE這兩種status
- 再複習一下:



Implementation

2-1 Implement a multilevel feedback queue

- 為了完成這次的MultiLevel Feedback Queue，於**Thread**結構、**Alarm**以及**Scheduler**皆需要做更動

Thread.h

- 宣告幾個排班用的變數，以及set & get Method

```
Public:
    void setBurstTime(int t) {burstTime = t;}
    void setWaitingTime(int t){waitingTime = t;}
    void setExecutionTime(int t){executionTime = t;}
    void setPriority(int p){priority = p;}
    void setL3Time(int t){L3Time = t;}
    int getBurstTime(){return (burstTime);}
    int getWaitingTime(){return (waitingTime);}
    int getExecutionTime(){return (executionTime);}
    int getPriority(){return (priority);}
    int getL3Time(){return (L3Time);}
Private:
    int burstTime;
    int waitingTime;
    int executionTime;
    int L3Time;
    int priority;
```

Thread.c

按照作業中的提示:

- Only update approximate burst time ti (include both user and kernel mode) when process change its state **from running to waiting.**
- 推測應該是更改Sleep這裡啦...
- 更新時間的公式應該是這樣吧... 照著SPEC亂刻的...

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

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

    status = BLOCKED;

    int prevBurstTime = this->getBurstTime();
    int newBurstTime = 0.5*prevBurstTime + 0.5*this->getExecutionTime();
    this->setBurstTime(newBurstTime);
    int diff = newBurstTime - prevBurstTime;

    while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
        kernel->interrupt->Idle();
    }

    kernel->scheduler->Run(nextThread, finishing);
}
```

scheduler.h

- 新增一個updatePriority函式 (為了做Aging)以及三個ReadyQueue

```
Public:
    void updatePriority();
private:
    SortedList<Thread *> *L1ReadyList;
    SortedList<Thread *> *L2ReadyList;
    List<Thread *> *L3ReadyList;
```

scheduler.c

- 新增兩個compare function (可參考 interrupt 裡面的 compare 寫法)
- L3因為是用RR排班，所以不用Compare，按照Time Quantum輪流就好了

```
static int
compareL1(Thread* t1, Thread* t2)
{
    if ( t1->getBurstTime() > t2->getBurstTime() ) return 1;
    else if ( t1->getBurstTime() < t2->getBurstTime() ) return -1;
    else return t1->getID() < t2->getID() ? -1 : 1;

    return 0;
}

static int
compareL2(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;
}
```

建構子 & 解構子

```
Scheduler::Scheduler()
{
    L1ReadyList = new SortedList<Thread *>(compareL1);
    L2ReadyList = new SortedList<Thread *>(compareL2);
    L3ReadyList = new List<Thread *>;

    toBeDestroyed = NULL;
}

Scheduler::~Scheduler()
{
    delete L1ReadyList;
    delete L2ReadyList;
    delete L3ReadyList;
}
```

實作updatePriority( )

- 這邊簡單來說就是利用 **ListIterator** (定義於list.h) 來遍歷ready queue裡面的全部Thread，並更新waiting time
    - 這邊我只在Timer Interrupt (每100ticks) 之間update waiting time
    - 判斷更新後的waiting time是否大於1500，並做Aging
    - 這邊要注意 Priority 需控制在 0 ~ 149之間
        - 故新增一個 if 條件式做判斷

```
void Scheduler::updatePriority()
{
ListIterator<Thread *> *iter1 = new ListIterator<Thread *>(L1ReadyList);
ListIterator<Thread *> *iter2 = new ListIterator<Thread *>(L2ReadyList);
ListIterator<Thread *> *iter3 = new ListIterator<Thread *>(L3ReadyList);

Statistics *stats = kernel->stats;
int oldPriority;
int newPriority;
// L1
for( ; !iter1->IsDone(); iter1->Next() ){
    ASSERT( iter1->Item()->getStatus() == READY);

iter1->Item()->setWaitingTime(iter1->Item()->getWaitingTime()+TimerTicks);
    if(iter1->Item()->getWaitingTime() >= 1500
    && iter1->Item()->getID() > 0 ){

        oldPriority = iter1->Item()->getPriority();
        newPriority = oldPriority + 10;
        if (newPriority > 149){
            newPriority = 149;
        }
        iter1->Item()->setPriority(newPriority);
        iter1->Item()->setWaitingTime(0);
    }
}
// L2
for( ; !iter2->IsDone(); iter2->Next() ){
    ASSERT( iter2->Item()->getStatus() == READY);

iter2->Item()->setWaitingTime(iter2->Item()->getWaitingTime()+TimerTicks);
    if(iter2->Item()->getWaitingTime() >= 1500
    && iter2->Item()->getID() > 0 ){
        oldPriority = iter2->Item()->getPriority();
        newPriority = oldPriority + 10;
        if (newPriority > 149){
            newPriority = 149;
        }
        iter2->Item()->setPriority(newPriority);
        L2ReadyList->Remove(iter2->Item());
        ReadyToRun(iter2->Item());
    }
}
// L3
for( ; !iter3->IsDone(); iter3->Next() ){
    ASSERT( iter3->Item()->getStatus() == READY);

iter3->Item()->setWaitingTime(iter3->Item()->getWaitingTime()+TimerTicks);
    if( iter3->Item()->getWaitingTime() >= 1500
    && iter3->Item()->getID() > 0 ){
        oldPriority = iter3->Item()->getPriority();
        newPriority = oldPriority + 10;
        if (newPriority > 149){
            newPriority = 149;
        }
        iter3->Item()->setPriority(newPriority);
        L3ReadyList->Remove(iter3->Item());
        ReadyToRun(iter3->Item());
    }
}
}
```

修改排班演算法 (從L1 依序判斷到 L3來做新Thread的插入)

- 可以注意的是，L1跟L2的Insert會順便呼叫之前寫的Compare函式來進行sorted insert

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

    thread->setStatus(READY);

    if(thread->getPriority() >= 100 && thread->getPriority() <= 149)
    {
        if( !kernel->scheduler->L1ReadyList->IsInList(thread) ){
            L1ReadyList->Insert(thread);
        }
    }
    else if ( (thread->getPriority() >= 50 && thread->getPriority() <= 99) )
    {
        if( !L2ReadyList->IsInList(thread) ){
            L2ReadyList->Insert(thread);
        }
    }
    else if ( (thread->getPriority() >= 0 && thread->getPriority() <= 49) )
    {
        if( !L3ReadyList->IsInList(thread) ){
            L3ReadyList->Append(thread);
        }
    }
}
```

未來要從ready queue挑選下一個要執行的Thread時，只要依序從L1挑到L3就好了

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

    if( !L1ReadyList->IsEmpty() ){
        return L1ReadyList->RemoveFront();
    }
    else if ( !L2ReadyList->IsEmpty() ){
        return L2ReadyList->RemoveFront();
    }
    else if ( !L3ReadyList->IsEmpty() ){
        return L3ReadyList->RemoveFront();
    }
    else {
        return NULL;
    }
}
```

thread.c

- 這邊記得把ReadyToRun從迴圈裡面拉出來，放到FindNextToRun之前
- 不然在某些情況下會發生Priority比較高的Process比Priority低的Process晚跑的情況

```
void
Thread::Yield ()
{
    ...
    kernel->scheduler->ReadyToRun(this);
    nextThread = kernel->scheduler->FindNextToRun();
    ...
}
```

Alarm.c

- 根據Hint:
    - The operations of preemption and priority updating can be delayed until the next timer alarm interval
- 於是我們在每個Timer Interrupt之間進行
    1. Aging判斷
    2. execution time累計

3. L1或L3的preemptive判斷

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

    kernel->scheduler->updatePriority();

    Thread *thread = kernel->currentThread;
    thread->setExecutionTime(thread->getExecutionTime() + TimerTicks);
    thread->setL3Time(thread->getL3Time() + TimerTicks);

    if ( kernel->currentThread->getID() > 0
        && status != IdleMode
        && kernel->currentThread->getPriority() >= 100 )
    {
        interrupt->YieldOnReturn();
    }

    if ( status != IdleMode && kernel->currentThread->getPriority() < 50 ) {
            if ( kernel->currentThread->getL3Time() >= 99 ){
              interrupt->YieldOnReturn();
      }
    }
}
```

2-2 Add a command line argument “-ep”

kernel.h

```
Private:
    Thread* t[51];
    int threadPriority[51];
    char*   execfile[51];
```

- 這邊我新增了 threadPriority 這個陣列，用來儲存Thread對應到的Priority
- 為了防止遇到極大量Thread的測資，我把陣列大小擴增到 51 (多1個為了存main thread)

Kernel.c

- 於Kernel的建構子新增一個 “-ep” 的指令來設定Thread初始Priority，很簡單不解釋

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

- 接著微調ExecAll函式，多接收一個threadPriority[i]參數

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

- Exec這邊多接收一個priority參數，並做一些初始化設定 (懶人作法)
    - 比較好的做法應該是修改thread的建構子，在裡面完成一切初始化，保持Exec這邊語法簡潔

```
int Kernel::Exec(char* name, int priority)
{
    t[threadNum] = new Thread(name, threadNum);
    t[threadNum]->setBurstTime(0);
    t[threadNum]->setWaitingTime(0);
    t[threadNum]->setExecutionTime(0);
    t[threadNum]->setPriority(priority);
    t[threadNum]->space = new AddrSpace();
    t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);
    threadNum++;
    return threadNum-1;
}
```

2-3 Add a debugging flag ‘z’

這邊不再貼程式碼上來，簡單敘述我在哪裡加上Debug訊息

A. Scheduler::ReadyToRun

B. Scheduler::FindNextToRun 以及 Aging的時候 (升級Ready Queue)

C. Scheduler::updatePriority()，每一次Aging完之後

D. Thread::Sleep，Status = BLOCKED之後

E. Scheduler::Run裡面的SWITCH( )之前

示意圖:

```
3[D] Tick[4154]: Thread[3] update approximate burst time, from:100, add [450], to [550].
[B] Tick[4154]: Thread[1] is removed from queue L[1].
[E] Tick[4154]: Thread[1] is now selected for execution, Thread[3] is replaced, and it has executed [1000] ticks.
[A] Tick[4164]: Thread[3] is inserted into queue L[2].
[C] Tick[4200]: Thread[3] changes its priority from [72] to [82].
```

Reference

1. 向**NachOS 4.0**作業進發（**2**）, https://morris821028.github.io/2014/05/30/lesson/hw-nachos4-2/ (https://morris821028.github.io/2014/05/30/lesson/hw-nachos4-2/)
2. **CSE120/Nachos**中文教程**.pdf**, https://github.com/zhanglizeyi/CSE120/blob/master/Nachos中文教程.pdf (https://github.com/zhanglizeyi/CSE120/blob/master/Nachos%E4%B8%AD%E6%96%87%E6%95%99%E7%A8%8B.pdf)
3. [Nachos 4.0] Nachos System Call Implementation Sample, http://puremonkey2010.blogspot.com/2013/05/nachos-40-nachos-system-call.html (http://puremonkey2010.blogspot.com/2013/05/nachos-40-nachos-system-call.html)
4. csie.ntust/homework/99/OS, http://neuron.csie.ntust.edu.tw/homework/99/OS/homework/homework2/B9715029-hw2-1/ (http://neuron.csie.ntust.edu.tw/homework/99/OS/homework/homework2/B9715029-hw2-1/)
5. C語言中EOF是什麼意思？, https://kknews.cc/zh-tw/tech/aee435n.html (https://kknews.cc/zh-tw/tech/aee435n.html)
6. Assembler Directives, https://docs.oracle.com/cd/E26502_01/html/E28388/eoiyg.html (https://docs.oracle.com/cd/E26502_01/html/E28388/eoiyg.html)
7. x86-64 Stack Frame Layout, https://sdasgup3.github.io/2017/10/12/x86_64_Stack_Frame_Layout.html (https://sdasgup3.github.io/2017/10/12/x86_64_Stack_Frame_Layout.html)
8. Linux Zombie進程狀態介紹 以及 如何清理, https://www.itread01.com/articles/1476250830.html (https://www.itread01.com/articles/1476250830.html)
9. 避免linux zombie process, https://ryan0988.pixnet.net/blog/post/38792928 (https://ryan0988.pixnet.net/blog/post/38792928)
10. strchr() - C語言庫函數, http://tw.gitbook.net/c_standard_library/c_function_strchr.html (http://tw.gitbook.net/c_standard_library/c_function_strchr.html)

11. C++中cout和cerr的区别？, [https://blog.csdn.net/Garfield2005/article/details/7639833 (https://blog.csdn.net/Garfield2005/article/details/7639833)](https://blog.csdn.net/Garfield2005/article/details/7639833)

隨便亂寫: NachOS Trace順序 (One By One Code)

main.c:

- Debug::Debug(char *flagList)
- Kernel::Kernel(int argc, char **argv)
- kernel->Initialize()
    - Thread::Thread(**main, 0**)
    - currentThread->setStatus(RUNNING);
    - Statistics::Statistics()
    - Interrupt::Interrupt()
        - SortedList(int (*comp)(T x, T y)) : List<T>() { compare = comp;};
    - Scheduler::Scheduler()
        - List<T>::List()
    - **Alarm::Alarm(bool doRandom)** (**Jay:** 這邊可以多注意，**Alarm**剛被宣告的時候就埋了一個 **(Alarm) Timer Interrupt**)
        - Timer(bool doRandom, CallBackObj *toCall)
            - Timer::SetInterrupt()
                - kernel->interrupt->Schedule(this, delay, TimerInt);
                    - PendingInterrupt(toCall, when, type)
                    - **pending->Insert(toOccur)**
                        - ListElement<T>(item)
                        - ASSERT(!IsInList(item));
                        - if: IsEmpty() * ...
                        - elif: compare(item, this->first->item) * ...
                        - else: * ...
                        - ASSERT(IsInList(item));
    - Machine::Machine(debugUserProg)
    - SynchConsoleInput::SynchConsoleInput(char *inputFile)
        - ConsoleInput::ConsoleInput(char *readFile, CallBackObj *toCall)
            - if: readFile == NULL
                - ...
            - else:
                - readFileNo = OpenForReadWrite(readFile, TRUE)
            - kernel->interrupt->Schedule(this, ConsoleTime, ConsoleReadInt);
        - Lock::Lock(char* debugName ("console in"))
            - Semaphore::Semaphore(char* debugName ("lock"), int initialValue (1))
                - List<T>::List()
        - Semaphore::Semaphore(char* debugName ("console in"), int initialValue (0))
            - List<T>::List()
    - SynchConsoleOutput::SynchConsoleOutput(char *outputFile)
        - ConsoleOutput::ConsoleOutput(char *writeFile, CallBackObj *toCall)
            - if writeFile == NULL
                - 略...
            - else
                - writeFileNo = OpenForWrite(writeFile)
        - Lock::Lock(char* debugName ("console out"))
            - Semaphore::Semaphore(char* debugName ("lock"), int initialValue (1))
                - List<T>::List()
        - Semaphore::Semaphore(char* debugName ("console out"), int initialValue (0))
            - List<T>::List()
    - SynchDisk::SynchDisk()
        - 略...

    - PostOfficeInput::PostOfficeInput(int nBoxes)
        - 略...
    - PostOfficeOutput::PostOfficeOutput(double reliability)
        - 略...
    - **interrupt->Enable() 重要!!!**
- CallOnUserAbort(Cleanup)
    - 略... (偵測使用者按下ctrl-C)
- kernel->ExecAll()
    - Exec(execfile[i])
        - Thread::Thread(char* threadName, int threadID)
        - AddrSpace::AddrSpace()
            - new TranslationEntry[NumPhysPages];
            - bzero(kernel->machine->mainMemory, MemorySize)
        - Thread::Fork(VoidFunctionPtr func (ForkExecute), void *arg (t[threadNum]))
            - StackAllocate(func, arg)
                - AllocBoundedArray(StackSize * sizeof(int))
                    - 略...
                - machineState[xxx] = *func
            - (void) interrupt->SetLevel(IntOff);
            - scheduler->ReadyToRun(this);
                - ASSERT(kernel->interrupt->getLevel() == IntOff)
                - thread->setStatus(READY);
                - readyList->Append(thread);
                    - 略...
            - **interrupt->SetLevel(oldLevel)** (**重要!!!** 這是一個很神秘的函式...)
                - ChangeLevel(old, now);
                - if ((now == IntOn) && (old == IntOff)): **OneTick() 重要!!!**
                    - ChangeLevel(IntOn, IntOff);
                    - **CheckIfDue(FALSE);**
                        - ASSERT(level == IntOff)
                        - if (pending->IsEmpty()) return FALSE;
                        - next = pending->Front();
                        - 略 (計算時間)
                        - if (kernel->machine != NULL) {kernel->machine->DelayedLoad(0, 0);}
                        - **inHandler = TRUE;**

```
do {
    next = pending->RemoveFront();
    next->callOnInterrupt->CallBack();
    delete next;
} while ( !pending->IsEmpty() && (pending->Front()->w
```

                        - **inHandler = FALSE;**
                        - return TRUE;
                    - ChangeLevel(IntOff, IntOn);
                    - if (**yieldOnReturn**): **kernel->currentThread->Yield()**; **(重要!)**
                        - kernel->interrupt->SetLevel(IntOff);
                            - 略...
                        - kernel->scheduler->FindNextToRun();
                            - 略...
                        - Code:

```
if (nextThread != NULL) {
  kernel->scheduler->ReadyToRun(this)
  kernel->scheduler->Run(nextThread, FALSE)
}
```

      - 略**... (**到**Run**這邊之後已經非常複雜了，總之最後會透過**SWITCH**呼叫**x86**組語，然後跑到**ForkExecute**，再到**Execute**再到**Machine::Run**，然後就是經典的 **for(;;){OneInstru, OneTick}**迴圈了**!!)**
    - kernel->interrupt->SetLevel(oldLevel);
- currentThread->Finish()