

OS MP3 Report

Team 27 : 112062232賴允中、111070070林標松

分工表

組員	Trace Code	Report編寫	Implementation
賴允中	Y	Y	Y
林標松	Y	Y	Y

Table of Contents

1. Trace Code ----- 2
2. Implementation ----- 32
3. Reflection ----- 47

1. Trace Code

1-1. New → Ready

- **threads/kernel.cc ⇒ Kernel::ExecAll()**

此函數會執行 kernel 初始化時，放在 execfile[] 裡所有要執行的 user program，它們來自於執行 NachOS 時所輸入的參數。

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

Main thread 會循環呼叫 **Kernel::Exec()**，執行 execfile[] 裡的 program。當所有program都執行完後，呼叫 **Thread::Finish()** 以結束 current thread，釋放thread的資源。

- **threads/kernel.cc ⇒ Kernel::Exec()**

此函數會為 user program 初始化它的 thread control block (TCB)，並分配一個 address space 紿此 thread，用來管理 thread 在記憶體中的狀態。最後將此 thread 放到 ready queue 中等待執行。

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

    return threadNum - 1;
}
```

首先，為此 user program 建構一個 thread 物件 (TCB)，並把這個 TCB 儲存在 kernel 用以管理 thread 的 t[] 中(threadNum為其 index)。將其 isExec 參數設為 true，代表此為用來執行 user program 的 thread。

接著，呼叫 [AddrSpace::AddrSpace\(\)](#) 來分配這個 thread 可用的 address space。最後呼叫 [Thread::Fork\(\)](#) 把要執行的程式碼載入，這邊可發現傳入了 [ForkExecute\(\)](#) 函數，並將現有 thread 數量+1。

- threads/thread.cc ⇒ Thread::Fork()

在 [Kernel::Exec\(\)](#) 裡，初始化完thread的address space後，會呼叫 [Fork\(\)](#) 函數。此函數會分配一塊 stack 記憶體空間給 thread 使用，放置好裡面的內容，最後把 thread 放到 ready queue 中等待被執行。

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

    DEBUG(dbgThread, "Forking thread: " << name << " f(a): " << (int)func << " " << arg);
    StackAllocate(func, arg);

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

首先，透過變數 interrupt 和 scheduler 獲取 kernel 的 interrupt 狀態以及 CPU scheduler。然後，呼叫 [StackAllocate\(\)](#) 來初始化此 thread 的 stack 空間。這裡把 [ForkExecute\(\)](#) 傳進去作為 thread 的執行起點，以開始執行user program。

分配完 stack 後，接著呼叫 [Scheduler::ReadyToRun\(\)](#) 來將 thread 放到 ready queue 裡，等待 scheduler 排程以使用CPU。

- threads/thread.cc ⇒ Thread::StackAllocate()

此函數會配置一塊記憶體空間給 thread 的 stack (可以存 thread 的 local variable、return address 等等)。

```
stack = (int *)AllocBoundedArray(StackSize * sizeof(int));
```

首先，呼叫 [AllocBoundedArray\(\)](#) 來配置一塊固定大小的 stack 空間。

StackSize (定義在thread.h裡) 為 8×1024 ，這裡配置了

$\text{StackSize} * \text{sizeof(int)} = 8192$ int 空間的 stack memory。

配置完 stack 的空間後，會根據不同的處理器架構，執行不同的區塊。

而我們是在 x86 架構下運行 NachOS，故執行以下區塊：

```
#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 address
    // 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
```

由於 stack 從高地址向低地址增長，所以首先將 stackTop 設置為 stack 的最高位置，然後將 ThreadRoot 放置在 stackTop 上。這樣當 thread 開始運行時，將會從 [ThreadRoot\(\)](#) 開始。

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

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

```
/* void ThreadRoot( void )
**
** expects the following registers to be initialized:
**     eax      points to startup function (interrupt enable)
**     edx      contains initial argument to thread function
**     esi      points to thread function
**     edi      point to Thread::Finish()
**/
```

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

▲ switch.h 中的定義

[ThreadRoot\(\)](#) 定義於 switch.S 中，負責調用被放在 ecx, esi, edi register 中的各函數。而將函數放入 register 這件事在 context switch 時會被 [SWITCH\(\)](#) 完成，稍後會詳細解釋這部分。

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

最後，把 ThreadRoot 和其他的 routine 存入 kernel 的 register，確保 thread 在正確的 routine 和參數下運行。注意此處的 (void *)func，即為剛剛傳入的 [ForkExecute\(\)](#)，其負責載入和執行 user program。

- threads/scheduler.cc ⇒ Scheduler::ReadyToRun()

在 [Thread::Fork\(\)](#) 裡，配置完 thread 的 stack memory 後，就會呼叫此函數把 thread 放入 ready queue 中。而此函數目的很簡單，就是把傳入的 thread 狀態設成 [READY](#)，並放到 scheduler 的 ready list 裡，等待排程。

```
void Scheduler::ReadyToRun(Thread *thread) {
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
    // cout << "Putting thread on ready list: " << thread->getName() << endl ;
    thread->setStatus(READY);
    readyList->Append(thread);
}
```

1-2. Running → Ready

- machine/mipssim.cc ⇒ Machine::Run()

此函數主要在模擬 CPU 不斷地在執行 instruction 的動作，直到最後遇到程式要結束時，要執行 Exit 的指令來結束 user program。

```
for (;;) {
    DEBUG(dbgTraCode, "In Machine::Run(), into OneInstruction "
          << "== Tick " << kernel->stats->totalTicks << " ==");
    OneInstruction(instr);
    DEBUG(dbgTraCode, "In Machine::Run(), return from OneInstruction "
          << "== Tick " << kernel->stats->totalTicks << " ==");
```

利用無限迴圈 for loop，不斷執行 instruction，透過呼叫 OneInstruction() 來讀取指令、解碼，並執行該指令。

```
DEBUG(dbgTraCode, "In Machine::Run(), into OneTick "
      << "== Tick " << kernel->stats->totalTicks << " ==");
kernel->interrupt->OneTick();
DEBUG(dbgTraCode, "In Machine::Run(), return from OneTick "
      << "== Tick " << kernel->stats->totalTicks << " ==");
```

執行完該指令，接著呼叫 OneTick() 來檢查是否有已經到期要等待處理的 interrupt，若沒有 interrupt 要處理，就回到無限迴圈，繼續執行 user program 指令。

- machine/interrupt.cc ⇒ Interrupt::OneTick()

此函數將模擬時間往前移，並檢查特定時間有無 interrupt 正等待處理。

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

將模擬時間往前進一個 tick。

```
// 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
```

呼叫 [CheckIfDue\(\)](#) 來檢查 pending interrupts，這裡為了確保 atomic 地執行，因此先關閉 interrupt，檢查完成才重啟 interrupt。

```
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;
}
```

如果 yieldOnReturn 為 True，代表有 context switch 的請求（timer 到期了），先將 yieldOnReturn 恢復回 False，接著切換到 system mode，才能去執行 yield routine。呼叫 [Thread::Yield\(\)](#) 切斷 currentThread 的執行狀態，將其放回 ready list 重新等待被 schedule。最後把 status 切回 old status（通常是 user mode）。

-----補充-----

- timer 的 interrupt 產生細節：

在 [Kernel::Initialize\(\)](#) 裡，會初始化 kernel 的 alarm 物件 (software alarm clock)，其負責 time slicing 的任務。

- **threads/alarm.cc ⇒ Alarm::Alarm()**

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

Alarm 的建構子會呼叫 [Timer\(\)](#) 建構子來初始化 software alarm clock，傳入的 call back object 是此 alarm，doRandom 為 false（在 [Kernel::Kernel\(\)](#) 函數裡 randomSlice 被設為 FALSE），因此可知於固定時間間隔會發起一次 time interrupt。

- **machine/timer.cc ⇒ Timer::Timer()**

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

此為 Timer 的建構子。可以發現在建構 timer 時，就會先呼叫一次 [Timer::SetInterrupt\(\)](#) 函數，把 timer interrupt 排程在未來的某時間點。

- **machine/timer.cc ⇒ Timer::SetInterrupt()**

```
void 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);
    }
}
```

此函數可排程一個於 delay 時間後發生的 timer interrupt，delay 的隨機與否取決於建構 timer 時傳入的 doRandom 參數。而因為 randomize 是 false，因此會固定在 TimerTicks（為 100 ticks，定義在 machine/stats.h 裡）時間後排程一個 time interrupt。

- **machine/timer.cc** ⇒ **Timer::CallBack()**

```
void 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
}
```

在處理 timer interrupt 時，會呼叫此函數。由於建構 timer 時傳入的 toCall 為 alarm，所以這裡呼叫的 callback 函數是 [Alarm::CallBack\(\)](#)。

可以發現在這裡又呼叫了一次 [Timer::SetInterrupt\(\)](#) 函數，透過再排程一個 timer interrupt 實現 time slicing，達到每 100 ticks 就中斷一次的功能。

- **threads/alarm.cc** ⇒ **Alarm::CallBack()**

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

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

在此函數裡，會先檢查 machine 是否在 IDLE 狀態。隨後呼叫 [Interrupt::YieldOnReturn\(\)](#) 函數，把 yieldOnReturn 設為 TRUE。

- **machine/interrupt.cc** ⇒ **Interrupt::YieldOnReturn()**

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

此函數目的只需要把 yieldOnReturn 的 flag 設為 TRUE，讓 [Interrupt::OneTick\(\)](#) 函數來檢查是否需要做 context switch 動作。注意此函數是被 interrupt handler 所呼叫，當 interrupt 都處理完畢才會回去 [Interrupt::OneTick\(\)](#) 處理 context switch。

- **threads/thread.cc ⇒ Thread::Yield()**

此函數讓握有 CPU 資源的 current thread 主動將自己放回 ready queue，切換到 next thread 來使用 CPU。

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

為了確保 context switch 過程可以 atomic 地執行，先 disable interrupt，並且確保 caller 是 current thread (CPU holder)。

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

呼叫 [Scheduler::FindNextToRun\(\)](#) 來找到 ready queue 裡下一個等待執行的 thread，並指派給 nextThread。如果 nextThread 為空，代表 ready list 為空，則不動作，返回 [Machine::Run\(\)](#) 的無限迴圈繼續執行 instructions。

如果 nextThread 非空，代表有在 ready list 找到可執行的 thread，就會呼叫 [Scheduler::ReadyToRun\(\)](#) 將 current thread 放回 ready queue，並呼叫 [Scheduler::Run\(\)](#) 進行 context switch，以讓 CPU 執行 nextThread。最後把 interrupt 恢復成原來的狀態 (oldLevel)。

- threads/scheduler.cc ⇒ Scheduler::FindNextToRun()

此函數會在 ready list 中依照設定好的演算法（目前是回傳最前面的元素）尋找下個最高順位的 thread，回傳給剛剛 Yield() 中的 nextThread 然後將其 pop 掉。若 ready list 是空的，則回傳 NULL。

```
Thread *  
Scheduler::FindNextToRun() {  
    ASSERT(kernel->interrupt->getLevel() == IntOff);  
  
    if (readyList->IsEmpty()) {  
        return NULL;  
    } else {  
        return readyList->RemoveFront();  
    }  
}
```

為確保 atomic 地執行，先檢查 interrupt 是否已被關閉，正常情形在 Thread::Yield() 裡已經有把 interrupt disable 了。接著，判斷 ready list 是否是空的，如果是空的回傳 NULL，非空則回傳 ready list 的 front element (thread)。而 RemoveFront() 函數除了回傳 front thread，還會把此 thread 從 ready list 裡移除。

- threads/scheduler.cc ⇒ Scheduler::ReadyToRun()

此函數目的很簡單，就是把傳入的 thread 狀態設成 READY，並放到 scheduler 的 ready list 裡，等待排程。

```
void Scheduler::ReadyToRun(Thread *thread) {  
    ASSERT(kernel->interrupt->getLevel() == IntOff);  
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());  
    // cout << "Putting thread on ready list: " << thread->getName() << endl ;  
    thread->setStatus(READY);  
    readyList->Append(thread);  
}
```

- threads/scheduler.cc ⇒ Scheduler::Run()

此函數主要進行 context switch 的工作，將 CPU 資源切換給 nextThread 使用。因為我們是從 **RUNNING→READY** 的角度切入（還沒執行完），故與 finishing 這個變數相關的區塊在此不提及。

```
1 if (oldThread->space != NULL) { // if this thread is a user program,  
2     oldThread->SaveUserState(); // save the user's CPU registers  
3     oldThread->space->SaveState();  
4 }
```

若 oldThread 是 user program thread （會被分配 address space） ，則會將其 CPU registers 和 memory 狀態保存下來。

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

將 kernel 的 currentThread 指派為準備接手 CPU 的 thread，狀態設為 **RUNNING** 。

```
1 // This is a machine-dependent assembly language routine defined  
2 // in switch.s. You may have to think  
3 // a bit to figure out what happens after this, both from the point  
4 // of view of the thread and from the perspective of the "outside world".  
5  
6 SWITCH(oldThread, nextThread);
```

呼叫 **SWITCH()** 以保存 oldThread 的 machine registers 並載入 nextThread 的 machine registers，後面會詳細解釋其內容。

```
// we're back, running oldThread
```

此處重要的是，註解裡寫 **SWITCH()** 返回後，會回到 oldThread，這就要用上面註解寫的 **perspective of the “outside world”** 去思考。

舉個例子，現在只有兩個 thread A 和 B。假設先從 A 呼叫了 `SWITCH()` 切換到 B，然後 B 遲早會切換回 A。此時，因為 A 切斷時保存的 register 都會被恢復（包括 PC），所以從外界看起來就像是從 `SWITCH()` 的下一行繼續執行。

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

```
1 void Scheduler::CheckToBeDestroyed() {  
2     if (toBeDestroyed != NULL) {  
3         delete toBeDestroyed;  
4         toBeDestroyed = NULL;  
5     }  
6 }
```

切換到 newThread 後，如果 oldThread 是已經完成工作的 thread，則會透過 `CheckToBeDestroyed()` 將其刪除，但目前的情境不是。

```
1 if (oldThread->space != NULL) { // if there is an address space  
2     oldThread->RestoreUserState(); // to restore, do it.  
3     oldThread->space->RestoreState();  
4 }
```

如果自己是還沒完工的 user program thread，那麼要將自己的 register, memory 資料寫回 kernel，才能正常繼續執行。

1-3. Running → Waiting

在 `Kernel::Kernel()` 裡，預設 `consoleOut` 為 `NULL` (代表使用 `stdout`)。

在 `Kernel::Initialize()` 裡，會初始化 `synchConsoleOut` 的資料結構，把 `consoleOut` 作為參數傳入 `synchConsoleOutput` 建構子。

-----補充-----

- **userprog/synchconsole.cc ⇒**
SynchConsoleOutput::SynchConsoleOutput()

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

首先，呼叫 `ConsoleOutput()` 建構子來初始化基本的 hardware display，並將 `this` 傳入作為稍後的回調對象。接著，呼叫 `Lock()` 建構子初始化 `lock`，這是用於確保每次只有一個 thread 可以進行 `PutChar`，實現註解中的 ”only one writer at a time” 。後面呼叫 `Semaphore()` 建構的 `waitFor`，則是用以等待 `PutChar` 動作完成，直到硬體輸出完成才觸發 `callBack`。

- **machine/console.cc ⇒ ConsoleOutput::ConsoleOutput()**

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

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

因傳入的 writeFile 是 NULL，writeFileNo 設為 1，代表 display 是 stdout。callWhenDone 是一個 CallBackObj 的指標，用於儲存在輸出操作完成時需要調用的回調物件。剛剛在 SynchConsoleOutput 中，傳入的是 this，這樣 ConsoleOutput 與 SynchConsoleOutput 形成緊密的聯繫，讓輸出完成後可以通知同步層進一步處理。

- **threads/synch.cc ⇒ Lock::Lock()**

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

可以發現 lock 底層是由 Semaphore 實作，且是 binary semaphore (mutex lock)，不是 0 就是 1，一次只有一個 thread 能拿到 lock，實現了同時只允許一個 writer 進行輸出。

- **threads/synch.cc ⇒ Semaphore::Semaphore()**

```
Semaphore::Semaphore(char *debugName, int initialValue) {
    name = debugName;
    value = initialValue;
    queue = new List<Thread *>;
}
```

根據傳入的 initialValue (此處為 0)，設定 semaphore 的初值。接著創建一個 queue，用來儲存因 semaphore 的值小於 0 而進入等待狀態的 threads。

- **userprog/synchconsole.cc ⇒ SynchConsoleOutput::PutChar()**

此函數將字元輸出到 console display 上，附帶了同步化的處理。

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

首先，呼叫 `Lock::Acquire()` 函數來取得 lock，確保同一時間只能有一個 writer 進行操作。

拿到 lock 的 thread 才能執行下面的 `ConsoleOutput::PutChar()` 函數，將字元寫到模擬的 display buffer 上。為了等待寫入完成（觸發 interrupt 後執行 callback），調用了 `waitFor->P()` 讓 thread 進入 BLOCKED status，直到 interrupt 到來。最後透過 `Lock::Release()` 釋放 lock。

- - - - - 補充 - - - - -

- `threads/synch.cc` ⇒ `Lock::Acquire()`

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

呼叫 `Semaphore::P()` 會嘗試將底層的 semaphore 資源減 1（稍後會詳細提到此函數），因前面有提到 lock 的底層機制是 binary semaphore。接著將 lockHolder 設為 currentThread，代表現在 lock 被當前的 thread 持有。

- machine/console.cc ⇒ `ConsoleOutput::PutChar()`

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

此函數模擬硬體層面的字元輸出操作。因為 `writeFileNo` 在初始化時已經被設為 1 (stdout)，所以 `WriteFile()` 會將 1 個 `char` 寫入 `stdout`。

將 `putBusy` 設為 `TRUE`，代表正在進行 `PutChar`，如果有其他 thread 也要輸出，會被 `ASSERT(putBusy == FALSE)` 擋下來並報錯。

最後，排程一個 `interrupt` 以通知輸出動作已經完成，將 `ConsoleOutput` 本體傳進 `Schedule()` 作為 `CallBackObject`，這樣處理 `interrupt` 時就會呼叫 `ConsoleOutput::CallBack()`，透過其解除 `putBusy` 和釋放。

- threads/synch.cc ⇒ `Lock::Release()`

```
void Lock::Release() {
    ASSERT(IsHeldByCurrentThread());
    lockHolder = NULL;
    semaphore->V();
}
```

Output 完成後，將 lock 釋放。這裡呼叫 `Semaphore::V()` 以將其底層 `semaphore` 的數量 +1 (1-4. Waiting → Ready 會詳細介紹此函數)，並更新 `lockHolder` 為 `NULL`，代表 lock 不再被任何 thread 佔用。

回到 `SynchConsoleOutput::PutChar()`。呼叫完 `ConsoleOutput::PutChar()` 函數後，會接著調用 `waitFor→P()`，也就是 `Semaphore::P()`，來等待 `PutChar` 動作結束後的 callback。

- threads/synch.cc ⇒ Semaphore::P()

此函數主要在檢查資源數量 (semaphore value) 是否大於 0。如果是，代表可以使用資源，將資源數量 -1；反之，代表還不能使用資源，要等到 semaphore value 大於 0，因此先將 thread 放到waiting queue 等待。

```
void Semaphore::P() {
    DEBUG(dbgTraCode, "In Semaphore::P(), " << kernel->stats->totalTicks);
    Interrupt *interrupt = kernel->interrupt;
    Thread *currentThread = kernel->currentThread;

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

因為檢查 semaphore 的 value 和對 value 進行減1的動作，必定要 atomic 地執行，故要先將 interrupt disable 掉。

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

接著，判斷 semaphore 的 value，如果為 0，就呼叫 [List::Append\(\)](#) 把 currentThread 加到 waiting queue 的末端，並呼叫 [Thread::Sleep\(\)](#) 函數 將 thread 狀態變更為 **BLOCKED**。若 semaphore value 大於0，代表可以 使用資源，直接將 value 減 1 以取用資源。

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

檢查和更正 semaphore value 的動作執行完後，把 interrupt 重新啟用。

- **lib/list.cc** ⇒ **List<T>::Append()**

此函數目的很簡單，將傳入的 item 加到 list 的末端。

```
template <class T>
void List<T>::Append(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 after last
        last->next = element;
        last = element;
    }
    numInList++;
    ASSERT(IsInList(item));
}
```

List 是 NachOS 定義好的資料結構，實現了一個以 linked list 實作的 queue。當加入一個 item 至 list 時，會先創造一個 ListElement 物件包裝它，接著會確認它目前沒有存在 List 裡，然後將其加到List的末端。

本例子中，**Semaphore::P()** 會調用 queue→Append(currentThread)，這會把 currentThread 加入先前於 SynchConsoleOutput 裡定義的 waitFor semaphore 的 waiting queue 的尾端。

- **threads/thread.cc** ⇒ **Thread::Sleep()**

此函數可釋放手中的 CPU 資源，因為 thread 已經完成工作，或是被 blocked 在等待同步。

此處例子是因為等待同步化而被 blocked。

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

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

首先，檢查 caller 是否為 current thread，且確保 interrupt 已經被 disable 了，這是為了保障從 ready queue 拿出 front thread，到 switch 完成前不會被其他 incoming interrupt 打斷。此處例子，Semaphore::P() 函數裡所呼叫的 currentThread→Sleep(FALSE)，傳入參數 finishing 是 FALSE，代表此 thread 還未完成，在等待同步。

```
status = BLOCKED;
// cout << "debug Thread::Sleep " << name << "wait for Idle\n";
while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
    kernel->interrupt->Idle(); // no one to run, wait for an interrupt
}
```

把當前的 thread (currentThread) 狀態設為 BLOCKED，並進入 while 迴圈，呼叫 Scheduler::FindNextToRun() 檢查 ready queue 裡是否有 thread 等待排程。

如果沒有，就呼叫 Interrupt::Idle()，檢查有沒有 pending interrupt，以產生 thread 放到 ready queue 裡（I/O interrupt 可以讓其他 waiting queue 中的 thread 回到 ready queue，1-4. Waiting → Ready 會提到）。如果也沒有 pending interrupt 要處理，就呼叫 Halt()，直接關機。

```
// returns when it's time for us to run
kernel->scheduler->Run(nextThread, finishing);
```

如果有，就呼叫 Scheduler::Run() 來執行下一個 thread，準備進行 context switch。注意 finishing 也會被繼續傳進去。

- threads/scheduler.cc ⇒ Scheduler::FindNextToRun()

此函數會在 ready list 中依照設定好的演算法（目前是回傳最前面的元素）尋找下個最高順位的 thread，回傳給剛剛的 nextThread 然後將其 pop 掉。若 ready list 是空的，則回傳 NULL。

```
Thread *  
Scheduler::FindNextToRun() {  
    ASSERT(kernel->interrupt->getLevel() == IntOff);  
  
    if (readyList->IsEmpty()) {  
        return NULL;  
    } else {  
        return readyList->RemoveFront();  
    }  
}
```

為確保 atomic 地執行，先檢查 interrupt 是否已被關閉。接著，判斷 ready list 是否是空的，如果是空的回傳 NULL，非空則回傳 ready list 的 front element (thread)。而 RemoveFront() 函數除了回傳 front thread，還會把此 thread 從 ready list 裡移除。

- threads/scheduler.cc ⇒ Scheduler::Run()

此函數主要進行 context switch 的工作，將 CPU 資源切換給 nextThread 使用。因為我們是從 **RUNNING→WAITING** 的角度切入（還沒執行完），故與 finishing 這個變數相關的區塊在此不提及。相同的解釋在 1-2. Running → Ready 的最後，故此處不重複敘述。

1-4. Waiting → Ready

回到 [1-3. Running → Waiting](#) 的 `SynchConsoleOutput::PutChar()`，我們討論到，當做完 `ConsoleOutput::PutChar()` 函數後，會排程一個 interrupt 來通知完成輸出。

- **machine/console.cc ⇒ ConsoleOutput::CallBack()**

當 console write interrupt 發生，interrupt handler 會呼叫 `ConsoleOutput` 的 `CallBack()` 函數。這是模擬硬體層的回調，並不做同步有關的處理。

此函數被呼叫時，代表下一個字元可以被輸出到 display 上。

```
void ConsoleOutput::CallBack() {
    DEBUG(dbgTraCode, "In ConsoleOutput::CallBack(), " << kernel->stats->totalTicks);
    putBusy = FALSE;
    kernel->stats->numConsoleCharsWritten++;
    callWhenDone->CallBack();
}
```

首先，將 `putBusy` 設為 `FALSE`，代表做完輸出動作了。

接著將 `numConsoleCharsWritten` 變數加 1。最後，呼叫 `callWhenDone` 物件（此處為 `SynchConsoleOutput`）的 `CallBack()` 函數，通知更上層的同步層進行同步化處理。

- **userprog/synchconsole.cc ⇒ SynchConsoleOutput::CallBack()**

```
void SynchConsoleOutput::CallBack() {
    DEBUG(dbgTraCode, "In SynchConsoleOutput::CallBack(), " << kernel->stats->totalTicks);
    waitFor->V();
}
```

此處調用 `waitFor→V()`，也就是 `Semaphore::V()`，來釋放 `waitFor` semaphore。雖然名字跟上面的很像，但它做的是更高層次的同步處理，與硬體輸出無關。

- threads/synch.cc ⇒ Semaphore::V()

此函數主要在釋放 semaphore 資源，並喚醒在等待此資源的 thread。此處例子是喚醒在上一小節在等待 waitFor 資源的 blocked thread。

```
void Semaphore::V() {
    DEBUG(dbgTraCode, "In Semaphore::V(), " << kernel->stats->totalTicks);
    Interrupt *interrupt = kernel->interrupt;

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

因為此函數也會更改到 semaphore 的 value 值，所以必定要 atomic 地執行，故會先將 interrupt 關掉。

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

若 waiting list 裡有 thread 存在，就呼叫 Scheduler::ReadyToRun()，將裡面的 front thread 哸醒，放回 ready queue 裡等待重新被 schedule。最後將 semaphore 的 value +1，釋放資源。

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

最後將 interrupt 恢復。

- threads/scheduler.cc ⇒ Scheduler::ReadyToRun()

此函數目的很簡單，前面也已經提過，就是把傳入的 thread 狀態設成 READY，並放到 scheduler 的 ready list 裡等待排程。

此處例子為將 waiting queue 的 front thread 傳入，將其設為 READY。

1-5. Running → Terminated

在 [Machine::Run\(\)](#) 函數中，當讀取到 user program 的最後一條指令時，該指令通常是用來執行 Exit 的 system call，表示程式即將結束。

此時，程式會呼叫 [Machine::OneInstruction\(\)](#) 進行指令解碼，發現這是一條 system call 指令後，會進一步觸發 ExceptionHandler 來處理該請求。

- **userprog/exception.cc ⇒ ExceptionHandler()**

此函數負責處理任何發生的 exception。

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

以此處為例，要處理的是 SC_Exit 的 system call。從傳入的參數 which 得知是哪種 exception，這裡是 SyscallException。透過讀取 register[2] 的資料，可得知是哪種 system call，此處是 SC_Exit。

這裡呼叫了 currentThread 自己的 [Thread::Finish\(\)](#) 函數，結束自己。

- **threads/thread.cc** ⇒ **Thread::Finish()**

`ExceptionHandler()` 裡，`SC_Exit` 呼叫此函數，以結束 current thread。

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

首先，先 disable interrupt，因為最後呼叫的 `Sleep()`，要避免發生 race condition。接著，確認調用者是當前正在使用 CPU 的 `currentThread`，因為只有目前執行中的 thread 才能呼叫 `Finish()` 來結束自身的執行。

```
if (kernel->execExit && this->getIsExec()) {
    kernel->execRunningNum--;
    if (kernel->execRunningNum == 0) {
        kernel->interrupt->Halt();
    }
}
```

如果 `execExit` 為真（表示 thread 執行完要終止 NachOS），且該 thread 的 `isExec` 為真（即該 thread 為 user program thread），則將 kernel 的 `execRunningNum`（目前正在執行的 user program thread 的數量）-1。若 `execRunningNum` 減至 0，表示已無任何執行中的 thread，則直接呼叫 `Halt()` 關掉 NachOS。

```
Sleep(TRUE); // invokes SWITCH
```

最後呼叫 `Thread::Sleep()`，傳入 TRUE 代表此 thread 已經完成工作，等待被釋放。接著會進行 context switch，把 CPU 讓給其他 thread。

- **threads/thread.cc** ⇒ **Thread::Sleep()**

此函數會釋放手中的 CPU 資源，因為 thread 已經完成，或是被 blocked 在等待同步。詳細已在 [1-3. Running → Waiting](#) 提及，此處僅簡述 finishing 參數造成的差異。

本小節例子中，傳入參數 finishing 為 **TRUE**，代表此 thread 已經完工。

```
// returns when it's time for us to run  
kernel->scheduler->Run(nextThread, finishing);
```

如果 ready queue 裡有下一個要等待使用CPU的 ready thread，就呼叫 [Scheduler::Run\(\)](#) 來執行下一個 thread，此時進行 context switch。這裡也會把 finishing 傳進去，此例子中是 TRUE。

- threads/scheduler.cc ⇒ Scheduler::FindNextToRun()

和 1-2. Running → Ready 小節的部分完全相同，故不重述。

- threads/scheduler.cc ⇒ Scheduler::Run()

和前面 1-2. Running → Ready 小節和 1-3. Running → Waiting 類似，重複的部分不再贅述。但要注意這邊傳入的 finishing 參數為 TRUE，和前面不同。

```
1 if (finishing) { // mark that we need to delete current thread
2     ASSERT(toBeDestroyed == NULL);
3     toBeDestroyed = oldThread;
4 }
```

在此例中，傳入的 finishing 為 TRUE，表示 currentThread 已執行完成，需要被釋放。因此，將 toBeDestroyed 設置為 oldThread，以便稍後在切換到新 thread 後，對其進行資源釋放。

```
1 void Scheduler::CheckToBeDestroyed() {
2     if (toBeDestroyed != NULL) {
3         delete toBeDestroyed;
4         toBeDestroyed = NULL;
5     }
6 }
```

```
1 void Thread::Begin() {
2     ASSERT(this == kernel->currentThread);
3     DEBUG(dbgThread, "Beginning thread: " << name);
4
5     kernel->scheduler->CheckToBeDestroyed();
6     kernel->interrupt->Enable();
7     if (kernel->execExit && this->getIsExec()) {
8         kernel->execRunningNum++;
9     }
10 }
```

這次 [CheckToBeDestroyed\(\)](#) 就會觸發了。此函數有兩種情況會觸發，一種是已被標記為要刪除的 oldThread switch 到一個執行到一半的 thread，執行到一半的 thread 會從 [SWITCH\(\)](#) 後的 [CheckToBeDestroyed\(\)](#) 開始執行，此時會刪除 oldThread，另一種是 switch 到完全沒被執行過的 new thread 時，其執行的起點函數 [ThreadRoot\(\)](#) 會呼叫 [Thread::Begin\(\)](#)，其中也會調用一次此函數做檢查。

1-6. Ready → Running

- **threads/scheduler.cc ⇒ Scheduler::FindNextToRun()**

此函數運作和前面提到的一樣，故不重述。

- **threads/scheduler.cc ⇒ Scheduler::Run()**

此函數運作和前面提到的一樣，故不重述。這一小節注重在，當 nextThread 的 status 被設為 **RUNNING** 後，接下來會呼叫 **SWITCH()**，進行 context switch。

- **threads/switch.s ⇒ SWITCH(Thread *t1, Thread *t2)**

SWITCH() 負責底層的 context switch 工作。其操作全建立於 register 和 memory 之間的資料搬移，因此在分析之前，我們要先了解 NachOS 是如何規劃它們的排序方式。

```
2 #define _ESP 0
3 #define _EAX 4
4 #define _EBX 8
5 #define _ECX 12
6 #define _EDX 16
7 #define _EBP 20
8 #define _ESI 24
9 #define _EDI 28
10 #define _PC 32
```

```
1 #define InitialPC % esi
2 #define InitialArg % edx
3 #define WhenDonePC % edi
4 #define StartupPC % ecx
```

上面是 switch.h 中的定義，我們可以從這裡知道，這些 register 和每個 thread 物件起始位置的距離（偏移量）。其中 esp 代表 stack pointer，指向目前 stack 的頂端；ebp 代表 base pointer，指向目前 stack 的基礎位置（底端）；pc 則指向目前的 instruction 的位置。而 esi, edx, edi, ecx 這幾個在前面有出現過，他們就是用來存放 **ThreadRoot()** 需要的函數與一些參數。剩下的 eax, ebx 則作為協助暫存中間值和搬移使用。

```
1  /* void SWITCH( thread *t1, thread *t2 )
2  **
3  ** on entry, stack looks like this:
4  **     8(%esp) ->      thread *t2
5  **     4(%esp) ->      thread *t1
6  **     (%esp) ->       return address
7  **
8  ** we push the current eax on the stack so that we can use it as
9  ** a pointer to t1, this decrements esp by 4, so when we use it
10 ** to reference stuff on the stack, we add 4 to the offset.
11 */
```

SWITCH() 的註解明白地展示了其 stack 的結構。其使用了 eax 以儲存 thread 1 的起始地址，我的理解是，因為後續會大量以此地址為基準做資料的搬移，用 eax 變數可以做到乾淨的符號表達和省去不必要的存取時間，不用每次都從 4(%esp) 加載一次。

了解這些後，我們可以來分析 SWITCH() 的細節了。

```
1  _SWITCH:
2  SWITCH:
3  movl %eax,_eax_save      # save the value of eax
4  movl 4(%esp),%eax        # move pointer to t1 into eax
5  movl %ebx,_EBX(%eax)    # save registers
6  movl %ecx,_ECX(%eax)
7  movl %edx,_EDX(%eax)
8  movl %esi,_ESI(%eax)
9  movl %edi,_EDI(%eax)
10 movl %ebp,_EBP(%eax)
11 movl %esp,_ESP(%eax)    # save stack pointer
12 movl _eax_save,%ebx     # get the saved value of eax
13 movl %ebx,_EAX(%eax)    # store it
14 movl 0(%esp),%ebx        # get return address from stack into ebx
15 movl %ebx,_PC(%eax)      # save it into the pc storage
```

先看上半部份，這裡的工作是將 thread 1 的資料保存起來，這樣下次如果 switch 回來了就可以完好如初的繼續執行。至於這些資料被保存到哪裡，我們回來看 Thread class 的結構就可以得到答案。

```

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

```

之前幾次作業我無法理解為何這裡的註解寫不能更動他們的順序，原來是因為 `SWITCH()` 是判斷與 thread 物件起點的相對位置來操作的。

以 `ESP` 為例子，在 `switch.h` 中定義其偏移量為 0，也就能解釋為何在 `Thread` class 中它被擺在最前面 (index 0)，且不能更動順序，否則 `sp` 就會指錯地方！

```

1 movl  8(%esp),%eax      # move pointer to t2 into eax
2
3 movl  _EAX(%eax),%ebx    # get new value for eax into ebx
4 movl  %ebx,_eax_save    # save it
5 movl  _EBX(%eax),%ebx    # restore old registers
6 movl  _ECX(%eax),%ecx
7 movl  _EDX(%eax),%edx
8 movl  _ESI(%eax),%esi
9 movl  _EDI(%eax),%edi
10 movl  _EBP(%eax),%ebp
11 movl  _ESP(%eax),%esp    # restore stack pointer
12 movl  _PC(%eax),%eax    # restore return address into eax
13 movl  %eax,4(%esp)       # copy over the ret address on the stack
14 movl  _eax_save,%eax
15
16 ret

```

再來看下半部分。和上面同理，為了方便，一開始我們讓 `eax` 儲存 `thread 2` 的起始地址。後面幾行可看出 `movl` 的方向是將資料從記憶體往硬體寄存器運送，畢竟我們準備要接著運行 `thread 2`。

根據 spec 的 Note，無視第 13 行。最後 `ret` 會返回到 `0(%esp)` 的地址 (11 行做的事)，也就是 `return address of thread B`。

- machine/mipssim.cc ⇒ Machine::Run()

(a) 當 context switch 到一個 NEW thread :

會從 ThreadRoot() 開始執行，先執行 Thread::Begin() 函數，接著執行 thread 要執行的function，此處是 ForkExecute()，此函數將 user program 載入memory，接著呼叫 AddrSpace::Execute() 來執行 user program，裡面會呼叫 Machine::Run()，進入 for loop 執行指令。

(b) 當 context switch 到一個 BLOCKED thread :

代表此 thread 還在等待同步化。switch 到此 thread 後，會執行 Scheduler::Run() 裡，SWITCH() 的下一行指令，也就是 CheckToBeDestroyed()，檢查是否有完工的 thread 要被刪除，如果是 user program thread，要將 user register 和其 address space 載回 kernel，接著會 return 回 Thread::Sleep()，再 return 回 Semaphore::P() 裡的 while 迴圈，去檢查 semaphore (此處為 waitFor) 資源的 value 是否大於0了，如果value大於0，就會繼續執行 SynchConsoleOutput::PutChar() 動作。

簡單來說，分成是否有 switch 過，如果有 switch 過就會從上次 SWITCH() 後的程式碼開始執行，沒有 switch 過就會從 ThreadRoot() 開始依序執行。

2. Implementation

2-1

(a) 將原本 Scheduler class 中的 readyList 換成 L1, L2, L3。

```
1 Scheduler::Scheduler() {
2     // readyList = new List<Thread *>;
3
4     // MP3
5     L1 = new List<Thread *>; // Preemptive SJF
6     L2 = new List<Thread *>; // Non-preemptive priority
7     L3 = new List<Thread *>; // Round Robin
8
9     toBeDestroyed = NULL;
10 }
```

```
1 Scheduler::~Scheduler() {
2     // delete readyList;
3     // MP3
4     delete L1;
5     delete L2;
6     delete L3;
7 }
```

(b) 為 Thread class 加入這些等等會用到的變數，並在建構 thread 物件時設定初值為 0。因為這些變數被放在 private 中，所以我們寫了幾個函數以取值和改值。

```
1 int priority;           // Priority of the thread (between 0 and 149)
2 int queueLevel;         // Which queue the thread is in
3 int initRunningTick;    // The tick when the thread starts running
4
5 double burstTime;       // Burst time (T) of the thread
6 double approxBurstTime; // Approximate burst time ( $t_i$ ) of the thread
7 double remainBurstTime; // Remaining burst time ( $T - t_i$ ) of the thread
8
9 int readyStartTick;     // The tick when the thread enters ready queue
```

```
1 void setPriority(int p) { priority = p; }
2 int getPriority() { return (priority); }
3 void UpdatePriority();
4
5 int getRemainBurstTime() { return (remainBurstTime); }
6 void UpdateRemainBurstTime();
7
8 double getBurstTime() { return (burstTime); }
9 void UpdateBurstTime();
10
11 double getApproxBurstTime() { return (approxBurstTime); }
12 void UpdateApproxBurstTime();
13
14 int getInitRunningTick() { return (initRunningTick); }
15 void UpdateInitRunningTick();
16
17 void setStartAgingTick(int tick) { readyStartTick = tick; } // Set the tick when the thread enters ready queue
18
19 void setQueueLevel(int level) { queueLevel = level; } // Set the queue level
20 int getQueueLevel() { return queueLevel; } // Get the queue level
```

- (c) 將原本 `Scheduler::ReadyToRun()` 中塞入 `readyList` 的邏輯改成根據其 `priority` 的值放入三種 queue。

```
1 //readyList->Append(thread);
2
3 // MP3 2-1(c)
4 if (thread->getPriority() >= 100) {
5     L1->Append(thread);
6 } else if (thread->getPriority() >= 50) {
7     L2->Append(thread);
8 } else {
9     L3->Append(thread);
10 }
```

- (d) 首先是 `FindNextToRun()`，我們要改成根據不同情況從三種 queue 中選擇下個執行的 `thread`。既然優先級為 $L1 > L2 > L3$ ，`if-else` 順序就是 $L1 \rightarrow L2 \rightarrow L3$ 。至於細節到 (f) (g) (h) 小節會詳細解釋。

接著要更改 `Alarm::CallBack()` 中觸發 `YieldOnReturn()` 的邏輯。優先級最高的是 $L1$ ，只要 $L1$ 還有元素，不管 `currentThread` 屬於第幾層，我們都要確認是否可以執行 `preemption`。如果 $L1$ 沒元素了，那除非 `currentThread` 屬於 $L2$ (non-preemptive)，否則也要確認 `preempt` 發生的可能性。

```
1 if (status != IdleMode) {
2     int currentQueueLevel = kernel->currentThread->getQueueLevel();
3     if(!scheduler->L1->IsEmpty() || currentQueueLevel == 3 || currentQueueLevel == 1)
4         interrupt->YieldOnReturn();
5 }
```

- (e) 實作 Aging 機制。thread 中需要有個變數來記錄，每當 thread 進入到 ready queue 的開始 tick 數：

```
int readyStartTick;           // The tick when the thread enters ready queue
```

而 readyStartTick 代表 thread 進入到 ready queue 時的 tick，因此要在 `Scheduler::ReadyToRun()` 裡去紀錄此變數，將 thread 在插入對應的 ready queue 後的現在 total tick 數，存入 readyStartTick 變數：

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

    thread->UpdateRemainBurstTime();
    thread->setStatus(READY);
    //cout << "Thread Priority: " << thread->getPriority() << endl;
    //readyList->Append(thread);

    // MP3
    // Stop accumulating T when the thread becomes ready state (2-1 (f))

    if (thread->getPriority() >= 100) {
        addToQueue(thread, L1, 1);
    } else if (thread->getPriority() >= 50 && thread->getPriority() <= 99) {
        addToQueue(thread, L2, 2);
    } else {
        addToQueue(thread, L3, 3);
    }
    thread->setStartAgingTick(kernel->stats->totalTicks);
}
```

在 thread class 裡，增加一個能夠依照 thread 的等待時間，更新 priority 的函數：

```
void UpdatePriority();
```

此函數實作如下：

```

void Thread::UpdatePriority() {
    int totalAgingTime = kernel->stats->totalTicks - readyStartTick;
    if(priority < 149 && totalAgingTime > 1500)
    {
        int oldPriority = priority;
        priority += 10;
        if(priority > 149) priority = 149;
        DEBUG('z', "[C] Tick [" << kernel->stats->totalTicks << "]: Thread [" << ID << "] changes its
priority from [" << oldPriority << "] to [" << priority << "]");
        readyStartTick = kernel->stats->totalTicks;
    }
}

```

首先，先利用現在的 total ticks 減去 thread 進入 ready queue 時的 tick 數 (readyStartTick)，得到此 thread 在 ready queue 裡等待的時間。接著，利用這個 total waiting tick 來判斷是否超過 1500 ticks，如果 thread 的 priority 此時不是最高(149)，且等待時間超過 1500 ticks，就要將此 thread 的 priority +10，並且確保加完後不會超過最高值 (149)，最後將 readyStartTick 設為現在的total ticks，如此便完成了 aging 的動作。因為這裡實作了更新 thread 的 scheduling priority，要加上 DEBUG [C] 的訊息。

因為我們有 3 個 level 的 ready queue，因此在 Scheduler 裡定義了兩個函數。一個是 public function，[UpdateThreadAging\(\)](#) 來給外面呼叫，此函數會將所有 ready queue 裡的 thread 做 aging；而另外一個是private function，只供 Scheduler 內部呼叫，[UpdateAgeInQueue\(\)](#) 會對每一層 queue 裡的 thread 做 aging：

```

class Scheduler {
public:
    void UpdateThreadAging(); // 更新所有thread的aging

private:
    // MP3
    void UpdateAgeInQueue(List<Thread*>* queue, int queueLevel); // 更新所有thread在queue中的aging
}

```

[UpdateThreadAging\(\)](#) 的實作：

```
void Scheduler::UpdateThreadAging() {
    UpdateAgeInQueue(L1, 1);
    UpdateAgeInQueue(L2, 2);
    UpdateAgeInQueue(L3, 3);
}
```

此函數會呼叫Scheduler的內部函數，[UpdateAgeInQueue\(\)](#)，將3個level的ready queue都分別呼叫，對每個ready queue做aging。

[UpdateAgeInQueue\(\)](#) 的實作：

```
void Scheduler::UpdateAgeInQueue(List<Thread*>* queue, int queueLevel){
    ListIterator<Thread*> iter(queue);
    while(!iter.IsDone()){
        Thread* currentThread = iter.Item();
        //currentThread->UpdateAgingTime();
        //currentThread->setStartAgingTick(kernel->stats->totalTicks);
        currentThread->UpdatePriority();
        iter.Next();

        int updatePriority = currentThread->getPriority();
        if(queueLevel == 2 && updatePriority >= 100){
            removeFromQueue(currentThread, L2, 2);
            addToQueue(currentThread, L1, 1);
        }
        else if(queueLevel == 3 && updatePriority >= 50){
            removeFromQueue(currentThread, L3, 3);
            addToQueue(currentThread, L2, 2);
        }
    }
}
```

此函數會用傳入的 ready queue，創建一個 ListIterator 物件，透過 while 週圈遍歷 ready queue 裡的 threads，對每個元素進行 aging。

首先，呼叫目前遍歷到的 thread (currentThread) 的 [UpdatePriority\(\)](#)，來將此 thread 做 aging。接著，利用更新後的 priority，判斷是否需要移動到更上層的 ready queue，如果現在是在 L2，但是更新後 priority 大於等於100，就要將此thread從 L2 從拔除，加入到 L1；如果現在是在

L3，但是更新後 priority 大於等於 50，就要將此 thread 從 L3 從拔除，加入到 L2。如果已經在 L1 則不需要做移動。

更新 thread 的 priority 動作要在 time alarm 被 call back 時才能執行，因此在 `Alarm::CallBack()` 裡呼叫 `Scheduler::UpdateThreadAging()`，來對所有 ready queue 中的 thread 做 aging：

```
void Alarm::CallBack() {
    Interrupt *interrupt = kernel->interrupt;
    MachineStatus status = interrupt->getStatus();
    Scheduler *scheduler = kernel->scheduler;
    // MP3
    scheduler->UpdateThreadAging(); // 更新所有thread的aging time

    if (status != IdleMode) {
        int currentQueueLevel = kernel->currentThread->getQueueLevel();
        if (!scheduler->L1->IsEmpty() || currentQueueLevel == 3 || currentQueueLevel == 1)
            interrupt->YieldOnReturn();
    }
}
```

(f) L1 要使用 SJF 的邏輯做 preemption。thread 中需要的變數為：

```
1 double burstTime;           // Burst time (T) of the thread
2 double approxBurstTime;     // Approximate burst time (t_i) of the thread
3 double remainBurstTime;    // Remaining burst time (T - t_i) of the thread
```

當 thread 狀態變成 **BLOCKED** (waiting) 時，也就是 **Sleep()** 被呼叫時，要更新 approxBurstTime，並且將 burstTime 歸零。我將這兩件事包裝在 **UpdateApproxBurstTime()** 函數中，在 **Sleep()** 中改變狀態後呼叫。

```
1 void Thread::Sleep(bool finishing) {
2     Thread *nextThread;
3
4     ASSERT(this == kernel->currentThread);
5     ASSERT(kernel->interrupt->getLevel() == IntOff);
6
7     DEBUG(dbgThread, "Sleeping thread: " << name);
8     DEBUG(dbgTraCode, "In Thread::Sleep, Sleeping thread: " << name << ", " << kernel->stats->totalTicks);
9
10    status = BLOCKED;
11
12    // Update the approximated burst time when the thread becomes waiting state, and
13    // return the accumulated T to zero. (MP3)
14    UpdateApproxBurstTime();
```

```
1 void Thread::UpdateApproxBurstTime() {
2     UpdateBurstTime();
3     ✓ double newApproxBurstTime = (double)(approxBurstTime * 0.5 + burstTime * 0.5); // t_i
4     DEBUG('z', "[D] Tick [" << kernel->stats->totalTicks << "]: Thread [" << ID << "] update ap-
5     proxBurstTime = newApproxBurstTime;
6     ✓ burstTime = 0.0;
7 }
```

```
1 void Thread::UpdateBurstTime() {
2     burstTime += (double)(kernel->stats->totalTicks - initRunningTick);
3 }
```

當 thread 狀態變成 **READY** 時，也就是 **Yield()** 被呼叫時，要暫停 burstTime 的累計（不是歸零），直到回到 **RUNNING** 才繼續。其實這就相當於，等到 **RUNNING** 時把 tick 基數更新，用新的系統時鐘當減數。因為維持在 **READY** 也不會呼叫 **UpdateBurstTime()**，結果跟暫停累計的效果相同。

但剛轉換狀態時還是要透過 `UpdateBurstTime()` 把目前累計的 `burstTime` 更新，否則就會少累計到一段時間。

```
1 void Thread::Yield() {
2     Thread *nextThread;
3     IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);
4
5     ASSERT(this == kernel->currentThread);
6
7     DEBUG(dbgThread, "Yielding thread: " << name);
8
9     kernel->scheduler->ReadyToRun(this); // Put currentThread to ready list (MP3)
10    nextThread = kernel->scheduler->FindNextToRun();
11    if (nextThread != NULL) {
12        kernel->scheduler->Run(nextThread, FALSE);
13    }
14    (void)kernel->interrupt->SetLevel(oldLevel);
15 }
```

```
1 void Scheduler::ReadyToRun(Thread *thread) {
2     ASSERT(kernel->interrupt->getLevel() == IntOff);
3     DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
4     // cout << "Putting thread on ready list: " << thread->getName() << endl ;
5
6     thread->UpdateRemainBurstTime();
7     thread->setStatus(READY);
```

```
1 void Thread::UpdateRemainBurstTime() {
2     UpdateBurstTime();
3     remainBurstTime = approxBurstTime - burstTime;
4     if(remainBurstTime < 0) { ←
5         remainBurstTime = 0;
6     }
7 }
```

這裡要處理一下可能會發生還沒
估值，但 `burstTime` 已經開始計
算導致的負數問題（會導致 L1
一開始就選錯 `thread`）。

我將這件事包裝在 `ReadyToRun()` 中。同時 `ReadyToRun()` 的位置我也做了小調整，原本是在決定完 `nextThread` 後才要讓 `currentThread` 進入 `READY` 的，但既然 `currentThread` 每次都必須跟 `queue` 中的元素進行比較一輪，所以乾脆先把它丟回 `queue` 中（`ReadyToRun()` 做的事），讓 `FindNextToRun()` 一起比較，就不用特地多寫一個比較函數了（我們原本是這麼做的）。

節錄自 `Scheduler::Run()`，狀態回到 `RUNNING` 後更新 tick 基數。

```
1 kernel->currentThread = nextThread; // switch to the next thread
2 nextThread->setStatus(RUNNING); // nextThread is now running
3
4 // MP3
5 // Resume accumulating T when the thread moves back to the running state. (2-1 (f))
6 nextThread->UpdateInitRunningTick();
```

```
1 void Thread::UpdateInitRunningTick() {
2     initRunningTick = kernel->stats->totalTicks;
3 }
```

在這些地方處理好 burst time 相關的計算後，就可以來看 L1 的運作邏輯了。

在 `Scheduler::FindNextToRun()` 中：

```
1 if (!L1->IsEmpty()) { // Preemptive SJF
2     pickedThread = L1->Front();
3     it = new ListIterator<Thread *>(L1);
4     while(!it->IsDone()) {
5         Thread *t = it->Item();
6         if (t->getRemainBurstTime() == pickedThread->getRemainBurstTime()) {
7             if (t->getId() < pickedThread->getId()) {
8                 pickedThread = t;
9             }
10        } else if (t->getRemainBurstTime() < pickedThread->getRemainBurstTime()) {
11            //cout << "Thread" << t->getId() << " : t_i - T = " << t->getRemainBurstTime() << ", ";
12            //cout << "Thread" << pickedThread->getId() << " : t_i - T = " << pickedThread->getRemainBurstTime() << endl;
13            pickedThread = t;
14        }
15        it->Next();
16    }
17    return removeFromQueue(pickedThread, L1, 1);
18 }
```

跟在一個序列中尋找最大最小值同樣概念。我先預設 L1 的首個元素就是最後選中的 thread，然後遍歷 L1，如果找到的 thread 有更低的 remain burst time，那就更新選中的 thread。如果 remain burst time 相同，要選擇 thread id 比較小的。`getRemainBurstTime()` 就只是個回傳 remain burst time 的取值函數。

(g) L2 相對簡單很多，我們只需要取得現在的 priority，然後做跟 L1 類似的判斷即可。`getPriority()` 一樣只是個回傳 priority 的取值函數。

在 `Scheduler::FindNextToRun()` 中：

```
1 else if (!L2->IsEmpty()) { // Non-preemptive Priority
2     pickedThread = L2->Front();
3     it = new ListIterator<Thread *>(L2);
4     while(!it->IsDone()) {
5         Thread *t = it->Item();
6         if (t->getPriority() == pickedThread->getPriority()) {
7             if (t->getID() < pickedThread->getID()) {
8                 pickedThread = t;
9             }
10        } else if (t->getPriority() > pickedThread->getPriority()) {
11            pickedThread = t;
12        }
13        it->Next();
14    }
15    return removeFromQueue(pickedThread, L2, 2);
16 }
```

(h) L3 更簡單，只要每次 preempt 都取 queue 最前面的元素就是 round-robin 了。

在 `Scheduler::FindNextToRun()` 中：

```
1 else if (!L3->IsEmpty()) { // Round Robin
2     pickedThread = L3->Front();
3     return removeFromQueue(pickedThread, L3, 3);
4 }
```

三個 queue 都沒找到東西就回傳 NULL。

```
1 else {
2     //cout << "No thread in ready queue" << endl;
3     return NULL;
4 }
```

- (i) 由於我們的 `UpdateThreadAging()` 和 `YieldOnReturn()` 都是透過 `Alarm::CallBack()` 呼叫，因此完全符合這裡的要求，將 preemption 和更新 priority 的動作延遲到下一次 time alarm。

```
1 void Alarm::CallBack() {
2     Interrupt *interrupt = kernel->interrupt;
3     MachineStatus status = interrupt->getStatus();
4     Scheduler *scheduler = kernel->scheduler;
5     // MP3
6     scheduler->UpdateThreadAging(); // 更新所有thread的aging time
7
8
9     if (status != IdleMode) {
10         int currentQueueLevel = kernel->currentThread->getQueueLevel();
11         if(!scheduler->L1->IsEmpty() || currentQueueLevel == 3 || currentQueueLevel == 1)
12             interrupt->YieldOnReturn();
13     }
14 }
15 }
```

2-2

(a) 對 kernel 做修改，使其可以解讀 “-ep” 指令。首先，在 Kernel class 裡 (kernel.h) 增加一個 private 陣列，來儲存每個執行檔的優先順序 (Priority)，並且更改原本 [Kernel::Exec\(\)](#) 函數的宣告，多傳入一個 priority 參數。

```
int execfilePriority[10]; // MP3
```

```
//int Exec(char *name);  
int Exec(char *name, int priority); // MP3
```

(b) 在 [Kernel::Kernel\(\)](#) 裡，增加解析 “-ep” 指令的程式碼。

```
} else if (strcmp(argv[i], "-e") == 0) {  
    execfile[++execfileNum] = argv[++i];  
    execfilePriority[execfileNum] = 0; // default priority ,MP3 add  
    cout << execfile[execfileNum] << "\n";  
} else if (strcmp(argv[i], "-ep") == 0) { // MP3  
    ASSERT(i + 2 < argc);  
    execfile[++execfileNum] = argv[++i];  
    int filePriority = atoi(argv[++i]);  
    ASSERT(filePriority >= 0 && filePriority <= 149);  
    execfilePriority[execfileNum] = filePriority;
```

在原本解析 “-e” 指令的程式碼裡，將執行檔的 priority 設為 0，讓其去 Level 3 的 queue，使用 NachOS 預設的 Round-Robin Scheduling。而在解析 “-ep” 指令的程式碼裡，將讀到的執行檔 priority 儲存進對應的 execfilePriority[] 裡，並要先檢查 priority 是否介於 0 到 149 之間。

(c) 更改 Kernel::ExecAll() 和 Kernel::Exec() 函數

```
void Kernel::ExecAll() {
    for (int i = 1; i <= execfileNum; i++) {
        int a = Exec(execfile[i], execfilePriority[i]); // MP3
    }
    currentThread->Finish();
    // Kernel::Exec();
}
```

```
1 int Kernel::Exec(char *name, int priority) {
2     t[threadNum] = new Thread(name, threadNum);
3     t[threadNum]->setIsExec();
4     t[threadNum]->space = new AddrSpace();
5     t[threadNum]->setPriority(priority);
6     t[threadNum]->Fork((VoidFunctionPtr)&ForkExecute, (void *)t[threadNum]);
7     threadNum++;
8
9     return threadNum - 1;
10 }
11
```

將原本的 `ExecAll()` 裡的 `Exec()` 函數更改為前面新宣告的形式。在 `Exec()` 裡，因為要將 `priority` 記錄在對應的 `thread` 裡，因此在創造 `thread` 物件後，利用新增的 `Thread::setPriority()` 函數將 `priority` 存在 `thread control block` 裡。

2-3

在 lib/debug.h 裡，多增加一個flag ‘z’，來 debug scheduler 的行為。

```
const char dbgScheduler = 'z'; // MP3 , scheduler behavior
```

(a) 加入 [A] 在我們定義的 `Scheduler::addToQueue()` 中。

```
1 void Scheduler::addToQueue(Thread *thread, List<Thread *> *queue, int queueLevel) {
2     queue->Append(thread);
3     //thread->setStartAgingTick(kernel->stats->totalTicks); // 設定thread進入ready queue的tick
4     thread->setQueueLevel(queueLevel);
5     DEBUG('z', "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" << thread->getID() << "] is inserted into queue L[" << queueLevel << "]");
6 }
7 }
```

這裡除了呼叫 `List::Append()` 將 thread 放入對應的 ready queue，還需要呼叫我定義的 `Thread::setQueueLevel()`，將 thread 所在的 ready queue level 儲存。

(b) 加入 [B] 在我們定義的 `Scheduler::removeFromQueue()` 中。

```
1 Thread *Scheduler::removeFromQueue(Thread* thread, List<Thread *> *queue, int queueLevel) {
2     queue->Remove(thread);
3     DEBUG('z', "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" << thread->getID() << "] is removed from queue L[" << queueLevel << "]");
4     return thread;
5 }
```

這裡會呼叫 `List::Remove()`，來將特定的 thread 從對應的 ready queue 中拔除。

(c) 加入 [C] 在 `UpdatePriority()` 中。

```
1 void Thread::UpdatePriority() {
2     int totalAgingTime = kernel->stats->totalTicks - readyStartTick;
3     if(priority < 149 && totalAgingTime > 1500)
4     {
5         int oldPriority = priority;
6         priority += 10;
7         if(priority > 149) priority = 149;
8         DEBUG('z', "[C] Tick [" << kernel->stats->totalTicks << "]: Thread [" << ID << "] changes its priority from [" << oldPriority << "] to [" << priority << "]");
9         cout << "Thread [" << ID << "] changes its priority from [" << oldPriority << "] to [" << priority << "]" << endl;
10        readyStartTick = kernel->stats->totalTicks;
11    }
12 }
```

(d) 加入 [D] 在 `UpdateApproxBurstTime()` 中。

```

1 void Thread::UpdateApproxBurstTime() {
2     UpdateBurstTime();
3     double newApproxBurstTime = (double)(approxBurstTime * 0.5 + burstTime * 0.5); // t_
4     DEBUG('z', "[D] Tick [" << kernel->stats->totalTicks << "]: Thread [" << ID << "] update approximate burst time, from: [" << approxBurstTime << "], add [" <
5     approxBurstTime = newApproxBurstTime;
6     burstTime = 0.0;
7 }

```

(e) 加入 [E] 在 Scheduler::Run() 裡面的 SWITCH() 之前，或是說狀態轉為 RUNNING 後。

```

1 void Scheduler::Run(Thread *nextThread, bool finishing) {
2     Thread *oldThread = kernel->currentThread;
3
4     ASSERT(kernel->interrupt->getLevel() == IntOff);
5
6     if (finishing) { // mark that we need to delete current thread
7         ASSERT(toBeDestroyed == NULL);
8         toBeDestroyed = oldThread;
9     }
10
11    if (oldThread->space != NULL) { // if this thread is a user program,
12        oldThread->SaveUserState(); // save the user's CPU registers
13        oldThread->space->SaveState();
14    }
15
16    oldThread->CheckOverflow(); // check if the old thread
17        // had an undetected stack overflow
18
19    kernel->currentThread = nextThread; // switch to the next thread
20    nextThread->setStatus(RUNNING); // nextThread is now running
21
22    // MP3
23    // Resume accumulating T when the thread moves back to the running state. (2-1 f())
24    nextThread->UpdateInitRunningTick();
25    DEBUG(dbgScheduler, "[E] Tick [" << kernel->stats->totalTicks << "]: Thread [" << nextThread->getId() << "] is now selected for execution, thread [" << oldThread->getId() << "] is replaced, and it has exe
26
27    DEBUG(dbgThread, "Switching from: " << oldThread->getName() << " to: " << nextThread->getName());
28
29    // This is a machine-dependent assembly language routine defined
30    // in switch.s. You may have to think
31    // a bit to figure out what happens after this, both from the point
32    // of view of the thread and from the perspective of the "outside world".
33
34    SWITCH(oldThread, nextThread);
35

```

3. Reflection

此次作業在實作的部分，難度提升非常多。在測試程式碼時，各個測資輪流錯誤，改這裡就錯那裡，真的很燒腦。我們原本一直以為是大方向的整體邏輯錯誤，但實際上只是有些小細節沒注意到，例如：忘記在 addToQueue() 時，設定 thread 的 queue level，導致後面判斷錯誤。還有 L2 priority 判斷的地方，大於小於竟然打反了，浪費了好幾個小時才找到…

看到測資都通過時，頓時放下心中大石，畢竟死線剩 12 小時的時候我們可是一個 bug 都沒找到，真的很怕最後會寫不出來交一坨屍體。也透過這次實作，對 CPU Scheduling 有非常多的瞭解！