Cover Page

Team61

Member List

> 108070039 羅稚涵
> 108070038 簡志宇

Member Contribution

> Trace code 羅稚涵, 簡志宇
> Part I 流程說明 羅稚涵
> Part I function說明 簡志宇
> Implementation 簡志宇
> Part II report 羅稚涵

1. Trace Code

1-1. New → Ready

- 目的:
  這個階段中 process 從 New state 進入到 Ready state，要由 kernel 來把 terminal 讀進來的執行檔放入 ready list，因此會牽涉到 thread 的創建以及 scheduler 的操作。
- Kernel::ExecAll()

  從 terminal 讀進來的 execfile 經過此函數依序被執行。當所有的 execfile 都被放入 ready list 後currentThread(main Thread) 就會執行 finish()，結束掉這個 thread。

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

- Kernel::Exec(char*)

  這個 function 為要執行的process 建立一個 Thread object(類似 PCB)提供 kernel 能夠辨識及操作這些 thread。創建以及初始化 Thread object 的過程於 MP2 有解析過了，因此這邊就只簡述一下：

  1. construct a Thread，設定 thread 的 ID, thread name, state等。
  2. 賦予這個新創的 Thread 自己的 address space(虛擬的，當他被放到 running queue 時用此來設定與連接 main memory)。

  Fork 讓此 thread 能夠順利被執行到，內容會在下一點詳細說明。

  前面的threadNum++讓現在的thread的數量，也就是下一個thread的編號，所以return現在的thread時需要 threadNum-1。

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

  return threadNum-1;
}
```

- Thread::Fork(VoidFunctionPtr, void*)

  這個 function 的功用在於調用 VoidFunctionPtr func 指到的 function(也就是 ForkExecute，要執行新的 thread)，讓 muiltiprogramming 能夠順利執行。有兩個步驟來實行：

  1. 呼叫 StackAllocate(func, arg) 來配置並且初始化 stack
  2. 把這個 new thread 放進 ready queue 裡等待使用 CPU，ReadytoRun()必須在 interrupt 被 disable 的狀態下執行。

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

    StackAllocate(func, arg);

    oldLevel = interrupt->SetLevel(IntOff);
    scheduler->ReadyToRun(this);

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

- Thread::StackAllocate(VoidFunctionPtr, void*)

  顧名思義就是 allocate 一個 execution stack 給這個新的 thread，並且初始化相關的參數(例如stackTop(目前 stack 最上方)、STACK_FENCEPOST(測試 stack overflow 的記號))。這個 function 實作了多種 machine 的 architecture，以下code說明以 x86 的 machine 為例子。

```
void
Thread::StackAllocate (VoidFunctionPtr func, void *arg)
{
    stack = (int *) AllocBoundedArray(StackSize * sizeof(int));
.
.
.
#ifdef x86
    // the x86 passes the return address on the stack.  In order for SWITCH()
    // to go to ThreadRoot when we switch to this thread, the return addres
    // used in SWITCH() must be the starting address of ThreadRoot.
    stackTop = stack + StackSize - 4; // -4 to be on the safe side!
    *(--stackTop) = (int) ThreadRoot;
    *stack = STACK_FENCEPOST;
#endif
.
```

```
.
.
}
```

另外，在 allocate stack 時為了要讓此 thread 在 context switch 後(進入 running state)能正確執行到他的 program，所以將初始的 stack frame 設為 ThreadRoot，其包含以下三個動作：

1. enable interrupts
2. 呼叫 (*func)(arg) (ForkExecute，會去執行此 thread 的 program)
3. 呼叫 Thread::Finish() (程式結束，thread 被 block 住)

```
#ifdef PARISC
    // On HPUX，function pointer需經過檢查並轉換才能正確指到該 function
    machineState[PCState] = PLabelToAddr(ThreadRoot);
    machineState[StartupPCState] = PLabelToAddr(ThreadBegin);
    machineState[InitialPCState] = PLabelToAddr(func);
    machineState[InitialArgState] = arg;
    machineState[WhenDonePCState] = PLabelToAddr(ThreadFinish);
#else
    machineState[PCState] = (void*)ThreadRoot;
    machineState[StartupPCState] = (void*)ThreadBegin;
    machineState[InitialPCState] = (void*)func;
    machineState[InitialArgState] = (void*)arg;
    machineState[WhenDonePCState] = (void*)ThreadFinish;
#endif
```

- Scheduler::ReadyToRun(Thread*)

  把 thread 的 state 改成 Ready 並放進 ready queue的尾端，表示已經準備好要使用 CPU 資源了，接下來就等 scheduler 安排使用 CPU。

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

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

1-2. Running → Ready

- 目的:
  這個階段的process由Running state進入到Ready state。這是在user mode情況下執

行，過程中會檢查是否有interrupt，有已經準備好的thread要先執行等，最後再由Scheduler::Run()來檢查context switch的過程是否正確。

- Machine::Run()

  這個 function 的細部說明也有在 MP1 裡面講解過了，這邊就大概講解一下其作用及流程。它模擬一個 user program 執行的過程(在 user mode 執行)，會在infinite loop裡進行下列步驟：

  1. OneInstruction(instr)：不斷地讀取並解析指令
  2. kernel->interrupt->OneTick()：每讀一行指令的同時，系統就會把時間往前調一個時段，並且檢查有無 interrupt (下一節詳細說明)
  3. Debugger()：若執行 user program 時是以 debug 模式則會呼叫此函式來將詳細資訊印出來。

```
void
Machine::Run()
{
    Instruction *instr = new Instruction;  // storage for decoded instruction

    kernel->interrupt->setStatus(UserMode);
    for (;;) {
      OneInstruction(instr);
      kernel->interrupt->OneTick();

      if (singleStep && (runUntilTime <= kernel->stats->totalTicks))
        Debugger();
    }
}
```

- Interrupt::OneTick()

  讓 machine 模擬的 time 往前進一段時間，並且：

  1. call CheckIfDue(FALSE)：檢查有沒有 interrupt 預定被排在經過的這段時間內，如果有就執行該 interrupt 的處理。(這部分 MP1 已解析過)
  2. 檢查 yieldOnReturn：檢查是否需要切換目前的 thread，讓其他 thread 使用 CPU 資源。如果需要就會切換到 kernel mode 並且呼叫 Thread::Yield() 來把目前的 thread put to sleep，並且進行context switch到下個 thread。

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

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

// check any pending interrupts are now ready to fire
    ChangeLevel(IntOn, IntOff); // first, turn off interrupts
        // (interrupt handlers run with
        // interrupts disabled)
    CheckIfDue(FALSE);    // check for pending interrupts
    ChangeLevel(IntOff, IntOn); // re-enable interrupts

    if (yieldOnReturn) {  // if the timer device handler asked
            // for a context switch, ok to do it now
  yieldOnReturn = FALSE;
  status = SystemMode;    // yield is a kernel routine
  kernel->currentThread->Yield();
  status = oldStatus;
    }
}
```

- Thread::Yield()

  呼叫此 Yield() 的 thread 會檢查 ready queue 裡面是否有其他準備好要執行的 thread 存在：

  1. 若有，目前的 thread (currentThread) 將會放棄 CPU 的使用權，並透過 scheduler 找到下一個 thread (newThread) 然後讓它執行(如何找以及執行，在下三節說明)，在此同時 currentThread 也會將自己放到 ready queue 的尾端。之後當 currentThread 又重新排到 CPU 使用權時會 return 回這個函式(因為當時放棄 CPU 時指令讀到這裡)，並且執行最後一行以將interrupt 的狀態設成原本狀態。
  2. 若沒有，目前的 thread (currentThread) 則不會放棄 CPU 的使用權，而是繼續執行下去。注意這裡的實作方式與真正的 OS 有點不同，通常實作上還是會把 currentThread 放到 ready queue，而因為沒有其他 ready 的 thread，所以 currentThread 又馬上排到 CPU 使用權而繼續執行。

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

      ASSERT(this == kernel->currentThread);
  ```

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

- Scheduler::FindNextToRun()

  這個 function 在找出 ready queue 最前面的 thread，把它 return 回 caller，同時將它從 ready queue 移除，若 ready queue 是空的則 return NULL。

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

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

- Scheduler::ReadyToRun(Thread*)
  - 在1-1.已經有講解過了，這邊說明就略過。
- Scheduler::Run(Thread*, bool)

  這個 function 的工作就是在處理 CPU 使用權交接時的各項檢查以及動作，分別有以下這些：

  1. 透過 finishing 這個變數來分辨 oldThread(currentThread) 是否應被標示為已完成(toBeDestroyed)。這取決於 currentThread 是在何時呼叫此 function 的：
     a. 若是因為 timer 時間到而被迫放棄 CPU 使用權(Thread::Yield())，則因為還沒執行完所以 finishing = FALSE。
     b. 若是因為已經完成而呼叫 Thread::Finish()，則 finishing 就會是 TRUE 並且預定會進入 terminate state(thread 會被 destroy)。
  2. 若 oldThread (currentThread) 是 user program，則它會有自己的 address space (cf.在 Nachos 裡，main thread 也是一個 thread，而它不具 address space)，因此要把此 user program 目前的 CPU register 狀態以及 address space 狀態保存起來，以供之後 context switch 回來這個 thread 時可以回復。

3. 確認 oldThread (currentThread) 有無 stack overflow 的情況，有的話會造成 NachOS crash 掉。
4. 實行 context switch，將 nextThread 的 state 設為 running，然後呼叫 SWITCH(oldThread, nextThread) 來把 registers 的值換成 nextThread 的，並且開始執行 nextThread 的 program (SWITCH 函式會在1-6.詳細說明)。
5. 現在 oldThread 已經取回 CPU 使用權，檢查之前有無已執行完成的 thread 需要被 destroy，有就把它 delete 掉。注意：會跑到這邊的 code，表示 oldThread 當時在 context switch 時 finishing 是 FALSE 的，否則在當時的 nextThread 執行時就會把它刪掉了(同樣會在1-6.詳細說明)。
6. 因為要繼續跑 oldThread，所以如果是 user program 就把它的 CPU registers 以及 address space狀態回復。

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

// 1.
    if (finishing) {
         ASSERT(toBeDestroyed == NULL);
         toBeDestroyed = oldThread;
    }
// 2.
    if (oldThread->space != NULL) {
        oldThread->SaveUserState();
        oldThread->space->SaveState();
    }
// 3.
    oldThread->CheckOverflow();
// 4.
    kernel->currentThread = nextThread;
    nextThread->setStatus(RUNNING);

    SWITCH(oldThread, nextThread);
// 5.
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    CheckToBeDestroyed();
// 6.
    if (oldThread->space != NULL) {
        oldThread->RestoreUserState();
        oldThread->space->RestoreState();
    }
}
```

1-3. Running → Waiting

- 目的:
  這個階段的process由Running state進入到Waiting state。這裡是要將thread put to sleep，所以需要Semaphore::P()，這過程需要設定lock與對SynchConsoleOutput::Putchar()呼叫。
- SynchConsoleOutput::PutChar(char)

```
void
SynchConsoleOutput::PutChar(char ch)
{
    lock->Acquire();   //lock -- only one writer at a time || Acquire -- wait unti
l the lock is FREE, then set it to BUSY
    consoleOutput->PutChar(ch);
    waitFor->P();      //wait for EOF or a char to be available?
    lock->Release();   //Release -- set lock to be FREE, waking up a thread waitin
g in Acquire if necessary
}
```

SynchConsoleOutput 這個 object 是用來同步 console display 的，每當 user program putChar時，它就需要做一些處理使得 console display 能夠同步顯示出文字。

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

下面說明處理的流程：

1. 取得 lock：

   說明一下 Lock 這個 object，這是用來實現 synchronization 的工具，而其又是以 Semaphore實作出來的。初始化時是 unlock 狀態所以 lockHolder = NULL。

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

   因為 console display 同時只能有一個 thread 使用，因此設定成取得這個 lock 的 thread 才有使用權。Lock::Acquire 這個 function 就是呼叫 Semaphore->P() (會

等到 lock 是 FREE 時才釋放，詳細說明見下一節) 來取得 lock，然後將 lockHolder 設為 currentThread。

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

2. 呼叫 console display 的 PutChar：

   先看看 ConsoleOutput 這個 object，constructor 的部分設定了輸出的方式以及 callback object (這邊的 toCall 就是 SynchConsoleOutput)

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

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

   再來回到 PutChar，在取得 lock 後就能開始進行輸出的動作。先確認是否有其他人也在做PutChar，若有就會報錯；再來輸出 char (WriteFile)；然後將 putBusy 設成 TRUE 以表示現在有人在做 PutChar，最後將這個 PutChar 的 interrupt 排程，使其能夠被正確執行 (把自己也傳進去是為了讓自己在 interrupt 處理完成後能被通知進而作後續動作)。

```
void
ConsoleOutput::PutChar(char ch)
{
    ASSERT(putBusy == FALSE);

    WriteFile(writeFileNo, &ch, sizeof(char)); //
    putBusy = TRUE;
    kernel->interrupt->Schedule(this, ConsoleTime, ConsoleWriteInt);
}
```

   當這個 interrupt 被處理完成後會呼叫 ConsoleOutput::Callback()，將 putBusy 設回 FALSE ，因為已經輸出完成，並更新輸出狀態，最後再呼叫 callWhenDone

(SynchConsoleOutput) 的 CallBack() function 執行 waitFor->V() 來告知這個字元已經完成輸出。

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

3. 等待下個字元：

   這邊的 waitFor->P() 會一直等到前一點的 waitFor->V() 結束(也就是前一個字元輸出完成)才可以繼續執行 (因為 waitFor 是 Semaphore)。

4. 釋放 lock：

   在輸出完成之後就可以將 lock 釋放出去讓下個 thread 使用。首先要檢查這個 lock 的主人是否依然是 currentThread，若不是表示有錯誤，會報錯。再來將 lockHolder 設為 NULL 並且 以 semaphore->V() 來讓 lock 變成無人取得的狀態。

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

- Semaphore::P()

  因為 Semaphore 的操作必須是 atomic 的，所以要先將 interrupt disable 掉。之後檢查 Semaphore value，若大於 0 表示有可使用的資源，那就直接使用 (value--)，否則表示需要等到有資源可使用，因此把 currentThread 放進等待 Semaphore 的 queue 並且將它 put to sleep。操作結束要記得將 interrupt enable 設回原來的狀態。

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

```
    Thread *currentThread = kernel->currentThread;

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

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

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

- SynchList<T>::Append(T)

  先來看看 SynchList<T> 這個 object，它的功能基本上與一般的 list 一致，提供了 append 與 remove 的操作，唯一不同的是這些動作都必須是 synchronized 的 (monitor style，一次只能有一個 T object(thread) 進行操作)，而實作的方法是使用 lock 來達到 mutual exclusion 的效果，且用 condition variable (Wait/Signal) 來操作 (所有 condition variable 的操作都必須由持有 lock 的那一個 thread 來呼叫，而同時可能有多個 thread 被這個 lock 所保護)。

```
template <class T>
SynchList<T>::SynchList()
{
    list = new List<T>;
    lock = new Lock("list lock");
    listEmpty = new Condition("list empty cond");
}
```

  再來看 Append(T)，因為對 SynchList<T> 操作必須是 mutual exclusive 的，所以要先取得 lock，然後再把 item append 到 list 尾端，做完動作之後因為不用了，所以必須要 signal 其他被此 lock 保護的 thread 來 wake up 下一個使用者(若沒有則此行 code 等同無效)，最後記得把 lock 使用權釋放。

  - Condition Variable 的 Signal，必須確認是否是 lock 持有者在操作。

```
void Condition::Signal(Lock* conditionLock)
{
    Semaphore *waiter;

    ASSERT(conditionLock->IsHeldByCurrentThread());

    if (!waitQueue->IsEmpty()) {
        waiter = waitQueue->RemoveFront();
      waiter->V();
```

```
    }
}
```

```
template <class T>
void
SynchList<T>::Append(T item)
{
    lock->Acquire();    // enforce mutual exclusive access to the list
    list->Append(item);
    listEmpty->Signal(lock);  // wake up a waiter, if any
    lock->Release();
}
```

- Thread::Sleep(bool)

  會呼叫到此 function 表示目前執行的 thread (currentThread) 碰到以下兩種情況之一：

  1. 已經執行完成，那 currentThread 就會被 block 住等待 nextThread 呼叫 CheckToBeDestroyed() 來把它刪掉。若沒有 nextThread 則會陷入無限迴圈，一直跑kernel->interrupt->Idle()，並且因為持續有 pending interrupt (timer) 而導致 NachOS 不會 shutdown。
  2. 尚未執行完成 (finishing = FALSE)，表示 timer 時間到了，該換下一個 thread (nextThread) 使用 CPU 了。一樣會先 block 住 currentThread，之後在找出 ready queue 的第一個 thread 使其成為 nextThread，並且執行它。

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

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

    status = BLOCKED;

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

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

- Scheduler::FindNextToRun()
  - 在1-2.已經有講解過了，這邊說明就略過。

- Scheduler::Run(Thread*, bool)
    - 在1-2.已經有講解過了，這邊說明就略過。

1-4. Waiting → Ready

- 目的:
  這個階段的process由Waiting state進入到Ready state。在I/O或是某個event完成後，這些sleep的thread需要Semaphore::V()來wake up，再重新進入到Ready queue中。
- Semaphore::V()

  與 Semaphore::P() 一樣必須是 atomic 的，所以先把 interrupt disable掉。之後檢查是否有 thread 在等待使用 Semaphore，若有就將它移出等待 Semaphore 的 queue，同時把它放到 ready queue 裡使其能夠使用 Semaphore (注意若有 thread 在等待，則表示在 Semaphore::P() 時已將它 put to sleep，所以當這邊 wake up它時，它就會從等待的迴圈跳出並且取得 Semaphore 使用權)。再來 increment Semaphore 的 value，使其他人能使用。最後再把 interrupt 設回原本狀態。

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

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

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

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

- Scheduler::ReadyToRun(Thread*)
    - 在1-1.已經有講解過了，這邊說明就略過。

1-5. Running → Terminated

- 目的:
  這個階段的process由Running state進入到Terminated state。將thread terminated需要ExceptionHandler的SC_Exit，再由Thread::Finish()裡的Sleep()將thread put to sleep。最後尋找下一個thread。

- ExceptionHandler(ExceptionType) case SC_Exit

  ExceptionHandler(ExceptionType) 在 MP1 有解析過了，因此這邊就只簡單講解一下流程：

  1. 在執行 user program 時若發生 exception，則會呼叫此函式，藉此進入 kernel mode 處理 exception。
  2. 首先會根據傳入的 ExceptionType 來辨別是哪種 exception，例如：SyscallException, PageFaultException, ReadOnlyException 等(定義在machine.h 中)
  3. 再來依據 ExceptionType 來做相應的處理，這邊目前只有實作 SyscallException 的部分：
     a. SyscallException 處理user program 呼叫 system call 的 exception，會根據各個 case 執行相應的 service routine。
     b. case SC_Exit

        已經執行完的 program 結束前會呼叫此 system call，然後 exception handler 透過此窗口來處理。先印出此 program 欲回傳的 return value (使用 system call 時會把此 argument存在 r4)，並且呼叫 Thread::Finish()，結束目前正在執行(已經執行完成)的 thread (currentThread)。

```
void
ExceptionHandler(ExceptionType which)
{
    char ch;
    int val;
    int type = kernel->machine->ReadRegister(2);
    int status, exit, threadID, programID, fileID, numChar;

    switch (which) {
      case SyscallException:
        switch(type) {
          .
          .
          // other SyscallException cases
          .
          .
          case SC_Exit:
            DEBUG(dbgAddr, "Program exit\n");
            val=kernel->machine->ReadRegister(4);
            cout << "return value:" << val << endl;
            kernel->currentThread->Finish();
          break;
          default:
            cerr << "Unexpected system call " << type << "\n";
          break;
        }
```

```
        break;
        default:
          cerr << "Unexpected user mode exception " << (int)which << "\n";
        break;
      }
      ASSERTNOTREACHED();
  }
```

- Thread::Finish()

  當一個 thread 上的 program 執行完成後就會呼叫此函式。它會先把 interrupt disable 掉(因為之後的 Thread::Sleep(bool) 要在 interrupt off 的狀態下執行)，再來呼叫 Sleep(TRUE) 來把 currentThread put to sleep (因為它已執行完)。

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

    DEBUG(dbgThread, "Finishing thread: " << name);
    Sleep(TRUE);
}
```

- Thread::Sleep(bool)

  會呼叫到此 function 表示目前執行的 thread (currentThread) 碰到以下兩種情況之一：

  1. 已經執行完成， currentThread 就會被 block 住等待 nextThread 呼叫 CheckToBeDestroyed() 來把它刪掉。若沒有 nextThread 則會陷入無限迴圈，一直跑kernel->interrupt->Idle()，並且因為持續有 pending interrupt (timer) 而導致 NachOS 不會 shutdown。
  2. 尚未執行完成 (finishing = FALSE)，表示 timer 時間到了，該換下一個 thread (nextThread) 使用 CPU 。一樣會先 block 住 currentThread，之後找出 ready queue 的第一個 thread 使其成為 nextThread，並且執行它。

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

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

    status = BLOCKED;
```

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

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

- Scheduler::FindNextToRun()
  - 在1-2.已經有講解過了，這邊說明就略過。
- Scheduler::Run(Thread*, bool)
  - 在1-2.已經有講解過了，這邊說明就略過。

1-6. Ready → Running

- 目的:
  這個階段的process由Ready state進入到Running state。一開始需要找到下一個thread，由Scheduler::Run()來判斷context switch的正確性。回到Running state後，則是在Machine::Run()裡的for的infinite loop裡執行。
- Scheduler::FindNextToRun()
  - 在1-2.已經有講解過了，這邊說明就略過。
- Scheduler::Run(Thread*, bool)
  - 在1-2.已經有講解過了，這邊說明就略過。
- SWITCH(Thread*, Thread*)

  SWITCH 是以組合語言寫成的，而為了要讓他能與 C++ language 連結，在 thread.h 裡面有宣告一個 extern 函式來宣告 SWITCH 跟 ThreadRoot。

  SWITCH 主要工作有兩個：
  - 儲存 oldThread 的 register state 和 address space 狀態並且回復 newThread 的 register state 和 address space 狀態。其流程大致如下：
    1. push current eax 到 stack 上當作 thread t1 的 pointer 以便操作。
    2. 先儲存 t1(oldThread) 的 registers —> stack pointer —> return address from stack (存到存 PC 的位置上，因為之後 return 要從這邊繼續執行)。
    3. 把 t2(newThread) 的 stack pointer 暫存進 eax 當作 thread t2 的 pointer。
    4. 再回復 t2 的 registers —> stack pointer —> return address (可能是 ThreadRoot or 上次執行中止的地方，取決於 t2 是否是第一次被執行)

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

        .globl  SWITCH
  .globl  _SWITCH
_SWITCH:
SWITCH:
        movl    %eax,_eax_save          # save the value of eax
        movl    4(%esp),%eax            # move pointer to t1 into eax
        movl    %ebx,_EBX(%eax)         # save registers
        movl    %ecx,_ECX(%eax)
        movl    %edx,_EDX(%eax)
        movl    %esi,_ESI(%eax)
        movl    %edi,_EDI(%eax)
        movl    %ebp,_EBP(%eax)
        movl    %esp,_ESP(%eax)         # save stack pointer
        movl    _eax_save,%ebx          # get the saved value of eax
        movl    %ebx,_EAX(%eax)         # store it
        movl    0(%esp),%ebx            # get return address from stack into
 ebx
        movl    %ebx,_PC(%eax)          # save it into the pc storage

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

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

        ret
```

- 將 program counter 移到 ThreadRoot or newThread 上次執行中止的地方
    1. 先來看看 switch.h 中定義的值

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

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

再回憶 Thread::StackAllocate(VoidFunctionPtr, void*) 中做的事

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

我們可以發現在 allocate stack 時其實就是先設置好一連串的 procedure：

- 把 PCState 設成 ThreadRoot 讓 newThread 在回覆玩registers 後可以跳到此 ThreadRoot 執行後續動作。
- 把 StartupPCState 設成 ThreadBegin 其實就是在準備執行 newThread。
- 把 InitialPCState, InitialArgState 分別設成 func, arg(Kernel::ForkExecute) 可以讓整個執行的 thread 真正轉移到 newThread。
- 把 WhenDonePCState 設成 ThreadFinish 可以讓 newThread 在結束執行之後進入 sleep 進而被 block 住。

2. 而 ThreadRoot 的組合語言部分其實就是在實現依序執行上面三個 function 的動作。

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

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

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

- (depends on the previous process state, e.g., [New,Running,Waiting]→Ready)
  - New → Running → Ready：

    這表示這個 oldThread 第一次執行後被打斷，然後換 newThread 執行，因此當 newThread 結束執行，這個 oldThread 會回到原本進行 SWITCH 的這一行繼續執行。

  - Running → Waiting → Ready

    這表示 oldThread 結束這一次的 CPU burst，因此會跳到 I/O device 執行，等到他結束那邊的工作就會再次被放到 ready queue。

- for loop in Machine::Run()

  Machine::Run() 裡的 for loop 就是持續不斷地讀入指令，並且把時間往前調進，檢查 interrupt；若為 debug 模式則輸出相關資料。

```
void
Machine::Run()
{
    .
    .
    .
    for (;;) {
        OneInstruction(instr);
        kernel->interrupt->OneTick();
        if (singleStep && (runUntilTime <= kernel->stats->totalTicks))
          Debugger();
    }
}
```

2. Implementation

2-1

- (a)
  - 在Scheduler::Scheduler()建立好三種level queue。

```
Scheduler::Scheduler()
{
    readyList_L1 = new SortedList<Thread *>(compareBurstTime);
    readyList_L2 = new SortedList<Thread *>(comparePriority);
    readyList_L3 = new List<Thread *>;
    toBeDestroyed = NULL;
}
```

  - L1是以burst time做比較，實作的部分在Scheduler.cc。at與bt的值為這個thread所需要的時間(burst time)減去實際所需花的時間(used time)。在at或bt大於0的情況下，會將其設為其值，否則就設為0。最後，如果at值等於bt則回傳0，at大於bt則回傳1，at小於bt則回傳-1。

```
int compareBurstTime(Thread *a, Thread *b){
    // less-burst-time thread is smaller
    double at = a->getBurstTime() - a->getUsedTime();
    double bt = b->getBurstTime() - b->getUsedTime();
    at = (at > 0 ? at : 0);
    bt = (bt > 0 ? bt : 0);
    return ((at == bt) ? 0 : ((at > bt) ? 1 : -1));
}
```

  - L2則以priority做比較，實作的部分一樣在Scheduler.cc。取得Thread a跟b取得priority值後，如果a的priority值等於b則回傳0，a大於b則回傳-1，a小於b則回傳1。

```
int comparePriority(Thread *a, Thread *b){
    // higher-priority thread is smaller
    int ap = a->getPriority();
    int bp = b->getPriority();
    return ((ap == bp) ? 0 : ((ap > bp) ? -1 : 1));
}
```

- (b)
  - 當Kernel::ExecAll()讀進新的thread後，Kernel::Exec()會將thread按造idx做區分，如果execfilePriority[idx] 大於0的話，會進入到Thread constructor初始化它的name, ID還有priority，其他的話，則會由Kernel::Initialize()裡的Thread()建立好，priority值會預設為0。

```
int Kernel::Exec(int idx, char* name)
{
    // 要執行execfile就要建一個新的child thread(參見thread.cc)
    if(execfilePriority[idx] > 0) {
        t[threadNum] = new Thread(name, threadNum, execfilePriority[idx]);
    }
    else t[threadNum] = new Thread(name, threadNum);
    .
    .
    .
}
```

```
Thread::Thread(char* threadName, int threadID)
{
    ID = threadID;
    name = threadName;
    priority = 0;
    burstTime = 0; // t0 = 0
    burstStartTime = 0;
    usedTime = 0;
    accumulatedUsedTime = 0;
    waitedTime = 0;
    waitStartTime = 0;
```

```
    .
    .
    .
    space = NULL;
}
Thread::Thread(char* threadName, int threadID, int p)
{
    ID = threadID;
    name = threadName;
    priority = p;
    burstTime = 0;
    burstStartTime = 0;
    usedTime = 0;
    accumulatedUsedTime = 0;
    waitedTime = 0;
    waitStartTime = 0;
    .
    .
    .
    space = NULL;
}
```

- (c)
  - 實作的部分在Scheduler::ReadyToRun()裡，這裡會按造各個thread的priority(由getPriority()取得)的順序安排下去，大於等於100的insert在L1，大於等於50小於100的則在L2，剩下小於50則在L3。

```
if(thread->getPriority() >= 100) {
        DEBUG(dbgScheduler, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" << thread->getID()
                                    << "] is inserted into queue L[1]");
        readyList_L1->Insert(thread);
    }
    else if(thread->getPriority() >= 50) {
        DEBUG(dbgScheduler, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" << thread->getID()
                                    << "] is inserted into queue L[2]");
        readyList_L2->Insert(thread);
    }
    else {
        DEBUG(dbgScheduler, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" << thread->getID()
                                    << "] is inserted into queue L[3]");
        readyList_L3->Append(thread);
    }
```

- (d)
  - 因為burst time會在waiting state計算，所以將burst time的更動在Thread::Sleep()實作。

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

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

    DEBUG(dbgThread, "Sleeping thread: " << name);
    DEBUG(dbgTraCode, "In Thread::Sleep, Sleeping thread: " << name << ", " << kernel->stats->totalTicks);

    this->setAccumulatedUsedTime(this->getAccumulatedUsedTime() + (kernel->stats->userTicks - this->burstStartTime));
    if(!finishing && this->usedTime != 0) {
        double oldThreadNewBurstTime = 0.5 * this->burstTime + 0.5 * this->usedTime;

        DEBUG(dbgScheduler, "[D] Tick [" << kernel->stats->totalTicks << "]: Thread [" << this->getID()
                            << "] update approximate burst time, from: [" << this->burstTime
                            << "], add [" << this->usedTime  << "], to [" << oldThreadNewBurstTime << "]");
        this->setUsedTime(0);
        this->setBurstTime(oldThreadNewBurstTime);
    }
    .
    .
    .
}
```

- (d)、(e)、(f)
  -

```
Thread *
Scheduler::FindNextToRun ()
{
```

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

    if (!readyList_L1->IsEmpty()) {
        Thread *thread = readyList_L1->Front();
    DEBUG(dbgScheduler, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" << thread->getID()
                                    << "] is removed from queue L[1]");
        return readyList_L1->RemoveFront();
    }
    else if (!readyList_L2->IsEmpty()) {
        Thread *thread = readyList_L2->Front();
        DEBUG(dbgScheduler, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" << thread->getID()
                                    << "] is removed from queue L[2]");
    return readyList_L2->RemoveFront();
    }
    else if (!readyList_L3->IsEmpty()) {
        Thread *thread = readyList_L3->Front();
        DEBUG(dbgScheduler, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" << thread->getID()
                                    << "] is removed from queue L[3]");
    return readyList_L3->RemoveFront();
    }
    else {
        return NULL;
    }
}
```

- (g)
  - Alarm::CallBack()會計算時間，每次的time interval都會更新一次，所以在這裡加入scheduler->updateThreadPriority()。此函式的下方則用來判斷是否為L1或L3，如果是的話再來呼叫interrupt->YieldOnReturn()。

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

    if(kernel->currentThread->getPriority() < 50 || kernel->currentThread->getPriority() >= 100 || !scheduler->isL1Empty()) {
        if (status != IdleMode) {
            interrupt->YieldOnReturn();
        }
    }
    else {
        kernel->currentThread->waitStartTime = kernel->stats->totalTicks;
    }
}
```

  - Scheduler::updateThreadPriority()裡會分別檢查L1, L2, L3裡的thread是否有需要做更動。當L2, L3在經過Thread::aging()的計算後有需要upgrade時，會對L2, L3進行更動。

```
void
Scheduler::updateThreadPriority()
{
    SortedList<Thread *> *tmpList_L1 = new SortedList<Thread *>(compareBurstTime);
    SortedList<Thread *> *tmpList_L2 = new SortedList<Thread *>(comparePriority);
    SortedList<Thread *> *tmp1 = readyList_L1, *tmp2 = readyList_L2;

    List<Thread *> *tmpList_L3 = new List<Thread *>();
    List<Thread *> *tmp3 = readyList_L3;

    ListIterator<Thread *> iter1(this->readyList_L1);

  for (; !iter1.IsDone(); iter1.Next()) {
      iter1.Item()->aging(kernel->stats->totalTicks - iter1.Item()->waitStartTime);
    }

    ListIterator<Thread *> iter2(this->readyList_L2);
  for (; !iter2.IsDone(); iter2.Next()) {
      bool upgrade = iter2.Item()->aging(kernel->stats->totalTicks - iter2.Item()->waitStartTime);
        if(upgrade) {
            if(iter2.Item()->getID() > 0) {

                DEBUG(dbgScheduler, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" << iter2.Item()->getID()
                                            << "] is removed from queue L[1]");

                readyList_L2->Remove(iter2.Item());
                kernel->scheduler->ReadyToRun(iter2.Item());
            }
        }
```

```
        else {
            tmpList_L2->Insert(iter2.Item());
        }
    }

    ListIterator<Thread *> iter3(this->readyList_L3);

  for (; !iter3.IsDone(); iter3.Next()) {
      bool upgrade = iter3.Item()->aging(kernel->stats->totalTicks - iter3.Item()->waitStartTime);
        if(upgrade) {
            if(iter3.Item()->getID() > 0) {

                DEBUG(dbgScheduler, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" << iter3.Item()->getID()
                                        << "] is removed from queue L[2]");
                readyList_L3->Remove(iter3.Item());
                kernel->scheduler->ReadyToRun(iter3.Item());
            }
        }
        else {
            tmpList_L3->Append(iter3.Item());
        }
   }
}
```

- 在Thread::aging()裡，new priority看還有幾個1500 ticks乘上10後，再加上old priority，就可以得到new priority，如果new priority大於149，仍然將其值設為149。如果old priority晉升一個queue level的話，就return True，反之return False。

```
bool
Thread::aging(int ageTime)
{
    this->setWaitedTime(ageTime);

    if(this->waitedTime < 1500) {
        return FALSE;
    }

    this->waitStartTime = kernel->stats->totalTicks;
    int oldPriority = this->priority;
    int newPriority = oldPriority + (this->waitedTime/1500) * 10;
    newPriority = (newPriority > 149 ? 149 : newPriority);

    if(newPriority != oldPriority) {
        DEBUG(dbgScheduler, "[C] Tick [" << kernel->stats->totalTicks << "]: Thread [" << this->getID()
                                    << "] changes its priority from [" << oldPriority << "] to ["
                                    << newPriority << "]");
        this->setWaitedTime(this->waitedTime%1500);
        this->priority = newPriority;
    }

    if((oldPriority < 100 && newPriority >= 100) || (oldPriority < 50 && newPriority >= 50)) {
        return TRUE;
    }
    return FALSE;
}
```

2-2

- 在Kernel::Kernel()新增一個argument “-ep”，會藉由execfile[]與execfilePriority[]來讀取thread的name與priority。

```
Kernel::Kernel(int argc, char **argv)
{
    .
    .
    .
    for (int i = 1; i < argc; i++) {
        .
        .
        .
         else if (strcmp(argv[i], "-ep") == 0) { // MP3 implementation 2-2
            priorityFlag = TRUE;
            execfile[++execfileNum]= argv[++i];
            execfilePriority[execfileNum] = atoi(argv[++i]);
        }
        .
        .
```

```
        .
    }
}
```

2-3

- (a)
  - 首先到debug.h的全域設定const char dbgScheduler為z。
  - 需要安插這種DEBUG()資訊，像是Scheduler::ReadyToRun()內有需要insert的地方，queue level會依造不同的狀態賦予不同的值。

```
DEBUG(dbgScheduler, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" << thread->getID()
                                    << "] is inserted into queue L[queue level]");
```

- (b)
  - 需要安插這種DEBUG()資訊，包含了Scheduler::ReadyToRun()，還有Scheduler::updateThreadPriority()裡即將需要Remove()的地方，而queue level會依造不同的狀態賦予不同的值。

```
DEBUG(dbgScheduler, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" << thread->getID()
                                    << "] is removed from queue L[queue level]");
```

- (c)
  - 需要安插這種DEBUG()資訊，像是Thread::aging()，在這裡會將priority的順序做調換，

```
DEBUG(dbgScheduler, "[C] Tick [" << kernel->stats->totalTicks << "]: Thread [" << this->getID()
                                    << "] changes its priority from [" << oldPriority << "] to ["
                                    << newPriority << "]");
```

- (d)
  - 需要安插這種DEBUG()資訊有Thread::Sleep()，這裡會將burst time更新成為oldThreadNewBurstTime，也就是0.5*burstTime+0.5 * usedTime。

```
DEBUG(dbgScheduler, "[D] Tick [" << kernel->stats->totalTicks << "]: Thread [" << this->getID()
                                 << "] update approximate burst time, from: [" << this->burstTime
                                 << "], add [" << this->usedTime  << "], to [" << oldThreadNewBurstTime << "]");
```

- (e)
  - 需要安插這種DEBUG()資訊的像Scheduler::Run()，在這裡會有對SWITCH()的呼叫。下方的oldID在Run()中已經設為oldThread→getID()。

```
DEBUG(dbgScheduler, "[E] Tick [" << kernel->stats->totalTicks << "]: Thread [" << nextThread->getID()
                                 << "] is now selected for execution, thread [" << oldID
                                 << "] is replaced, and it has executed [" << oldUsedTime << "] ticks");
```

Difficulties & Feedback

羅稚涵

這次要trace code的部份之前大多有看過，整體進行還算順暢，卡比較久的點是Semaphore，但搭配講義後就比較看得懂。這次我負責的implement 2-3，但在insert與remove那邊漏掉了不少的細節，感謝隊友的強力幫忙。

簡志宇

這次作業頗為繁雜，雖然很多 code 有 trace 過了，但是其實有很多細節是之前沒注意到的，尤其是 Semaphore 以及 SWITCH 這兩個部分

另外 implementation 的部分真的讓我很頭大，整體的複雜程度完全不是前兩次作業能相比的，我在 aging 的部分花了非常多時間 debug，但也因此搞懂了很多原本誤會的細節。