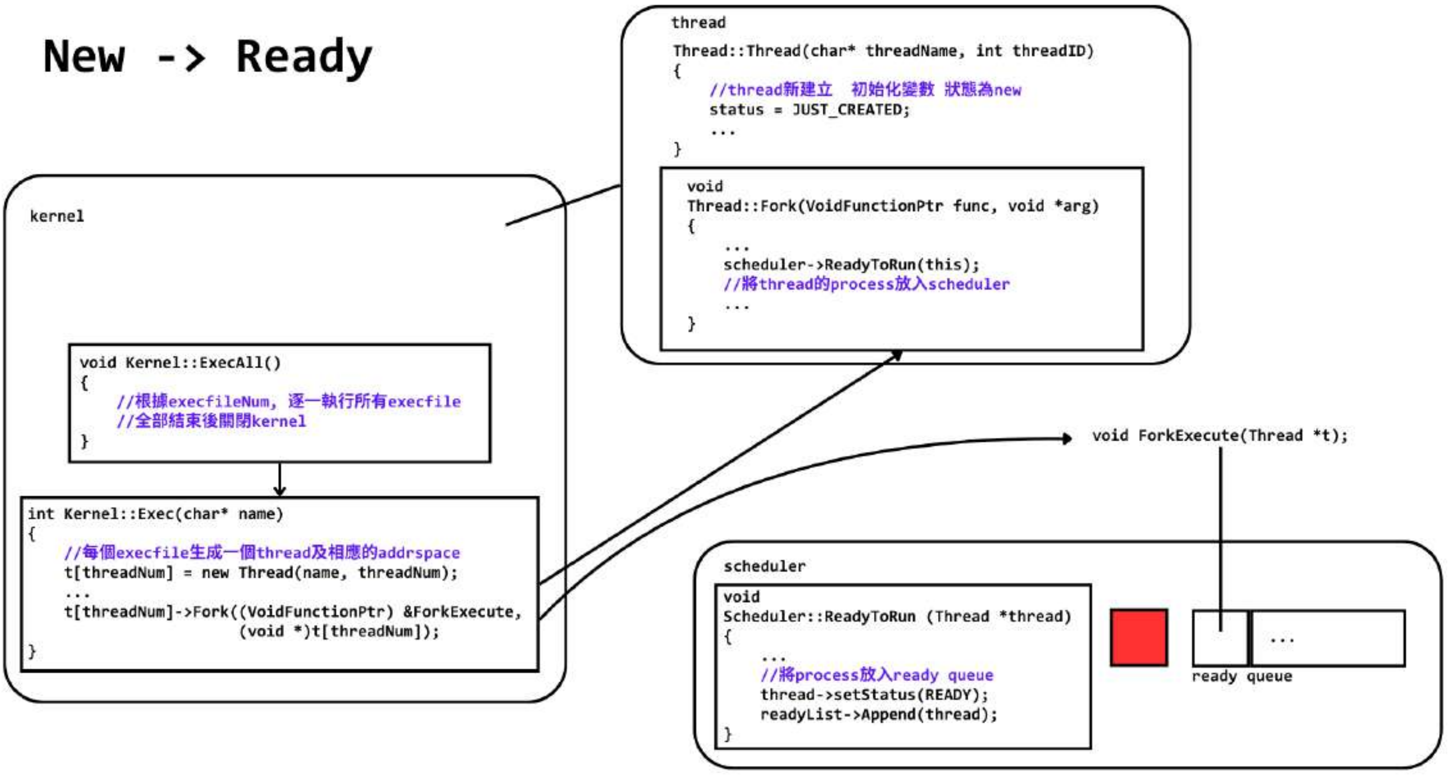
110705017 何翊華 110705063 廖偉辰

Part 1:Trace Code

1. Explain following path



New -> Ready

```
void Kernel::ExecAll()
{
    //根據execfileNum, 逐一執行所有execfile
    //全部結束後關閉kernel
}
```

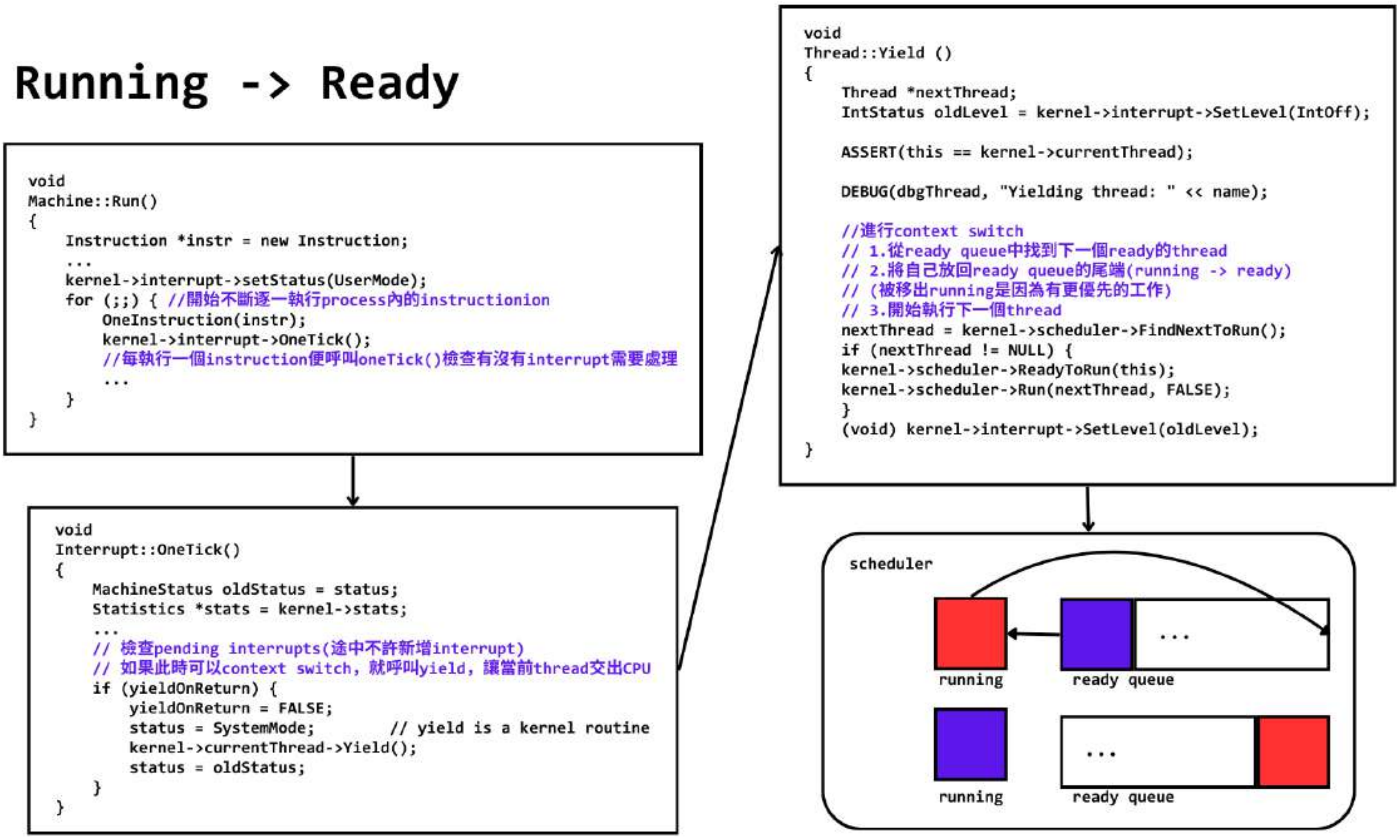
```
int Kernel::Exec(char* name)
{
    //每個execfile生成一個thread及相應的addrspace
    t[threadNum] = new Thread(name, threadNum);
    ...
    t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute,
                        (void *)t[threadNum]);
}
```

```
Thread::Thread(char* threadName, int threadID)
{
    //thread新建立  初始化變數 狀態為new
    status = JUST_CREATED;
    ...
}
```

```
void
Thread::Fork(VoidFunctionPtr func, void *arg)
{
    ...
    scheduler->ReadyToRun(this);
    //將thread的process放入scheduler
    ...
}
```

```
void ForkExecute(Thread *t);
```

```
void
Scheduler::ReadyToRun (Thread *thread)
{
    ...
    //將process放入ready queue
    thread->setStatus(READY);
    readyList->Append(thread);
}
```



Running -> Ready

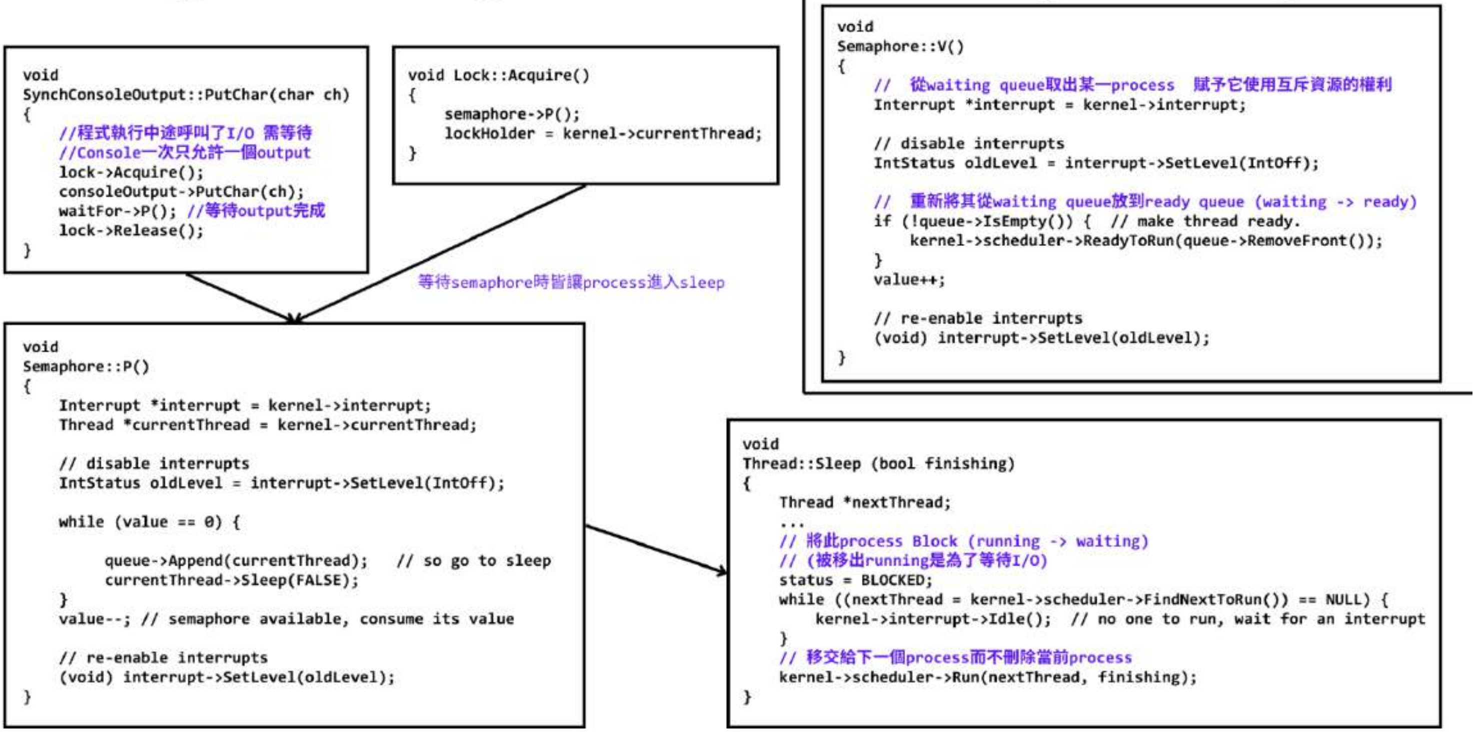
```
void
Machine::Run()
{
    Instruction *instr = new Instruction;
    ...
    kernel->interrupt->setStatus(UserMode);
    for (;;) { //開始不斷逐一執行process內的instructionion
        OneInstruction(instr);
        kernel->interrupt->OneTick();
        //每執行一個instruction便呼叫oneTick()檢查有沒有interrupt需要處理
        ...
    }
}
```

```
void
Interrupt::OneTick()
{
    MachineStatus oldStatus = status;
    Statistics *stats = kernel->stats;
    ...
    // 檢查pending interrupts(途中不許新增interrupt)
    // 如果此時可以context switch, 就呼叫yield, 讓當前thread交出CPU
    if (yieldOnReturn) {
        yieldOnReturn = FALSE;
        status = SystemMode;        // yield is a kernel routine
        kernel->currentThread->Yield();
        status = oldStatus;
    }
}
```

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

    ASSERT(this == kernel->currentThread);

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

    //進行context switch
    // 1.從ready queue中找到下一個ready的thread
    // 2.將自己放回ready queue的尾端(running -> ready)
    // (被移出running是因為有更優先的工作)
    // 3.開始執行下一個thread
    nextThread = kernel->scheduler->FindNextToRun();
    if (nextThread != NULL) {
    kernel->scheduler->ReadyToRun(this);
    kernel->scheduler->Run(nextThread, FALSE);
    }
    (void) kernel->interrupt->SetLevel(oldLevel);
}
```

Running -> Waiting

Waiting -> Ready



```
void
SynchConsoleOutput::PutChar(char ch)
{
    //程式執行中途呼叫了I/O 需等待
    //Console一次只允許一個output
    lock->Acquire();
    consoleOutput->PutChar(ch);
    waitFor->P(); //等待output完成
    lock->Release();
}
```

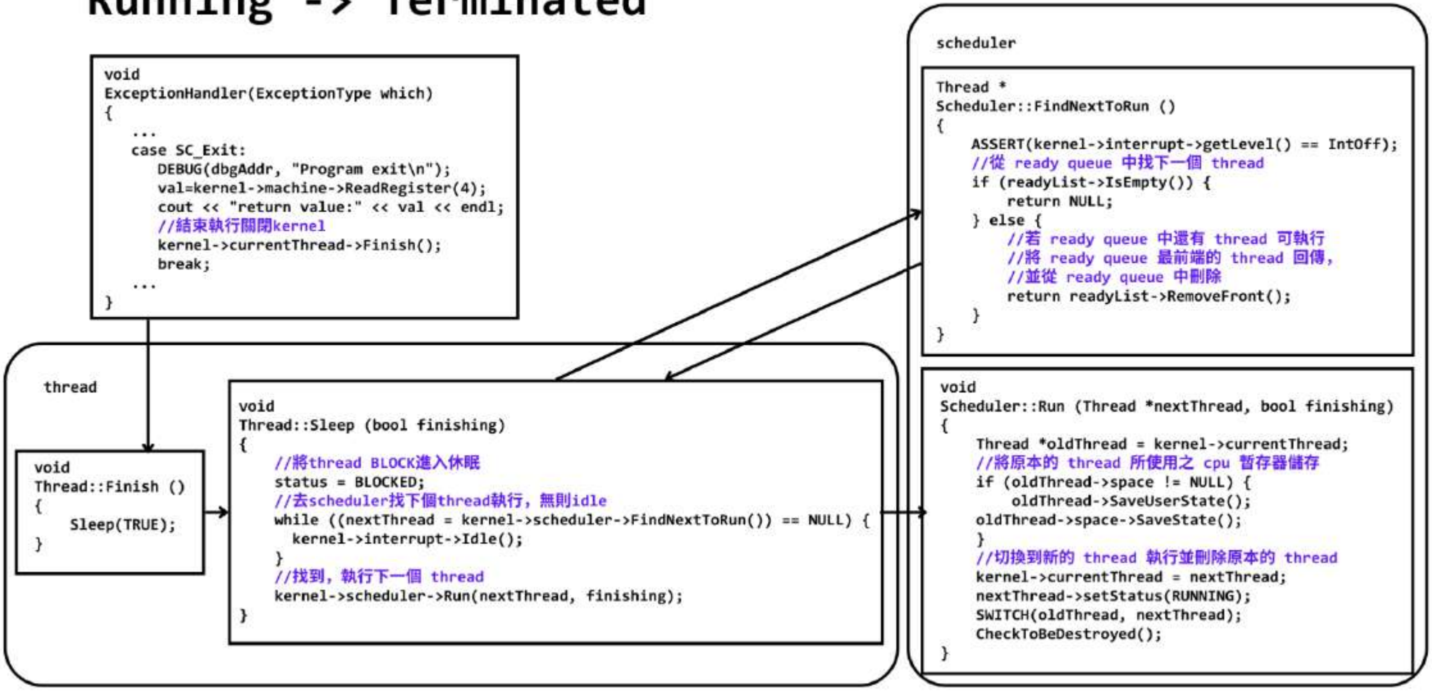
```
void Lock::Acquire()
{
    semaphore->P();
    lockHolder = kernel->currentThread;
}
```

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

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

    while (value == 0) {

         queue->Append(currentThread);   // so go to sleep
         currentThread->Sleep(FALSE);
    }
    value--; // semaphore available, consume its value

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

```
void
Semaphore::V()
{
    //  從waiting queue取出某一process  賦予它使用互斥資源的權利
    Interrupt *interrupt = kernel->interrupt;

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

    //  重新將其從waiting queue放到ready queue (waiting -> ready)
    if (!queue->IsEmpty()) {  // make thread ready.
        kernel->scheduler->ReadyToRun(queue->RemoveFront());
    }
    value++;

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

```
void
Thread::Sleep (bool finishing)
{
    Thread *nextThread;
    ...
    // 將此process Block (running -> waiting)
    // (被移出running是為了等待I/O)
    status = BLOCKED;
    while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
        kernel->interrupt->Idle();  // no one to run, wait for an interrupt
    }
    // 移交給下一個process而不刪除當前process
    kernel->scheduler->Run(nextThread, finishing);
}
```

Running -> Terminated



```
void
ExceptionHandler(ExceptionType which)
{
   ...
   case SC_Exit:
      DEBUG(dbgAddr, "Program exit\n");
      val=kernel->machine->ReadRegister(4);
      cout << "return value:" << val << endl;
      //結束執行關閉kernel
      kernel->currentThread->Finish();
      break;
   ...
}
```

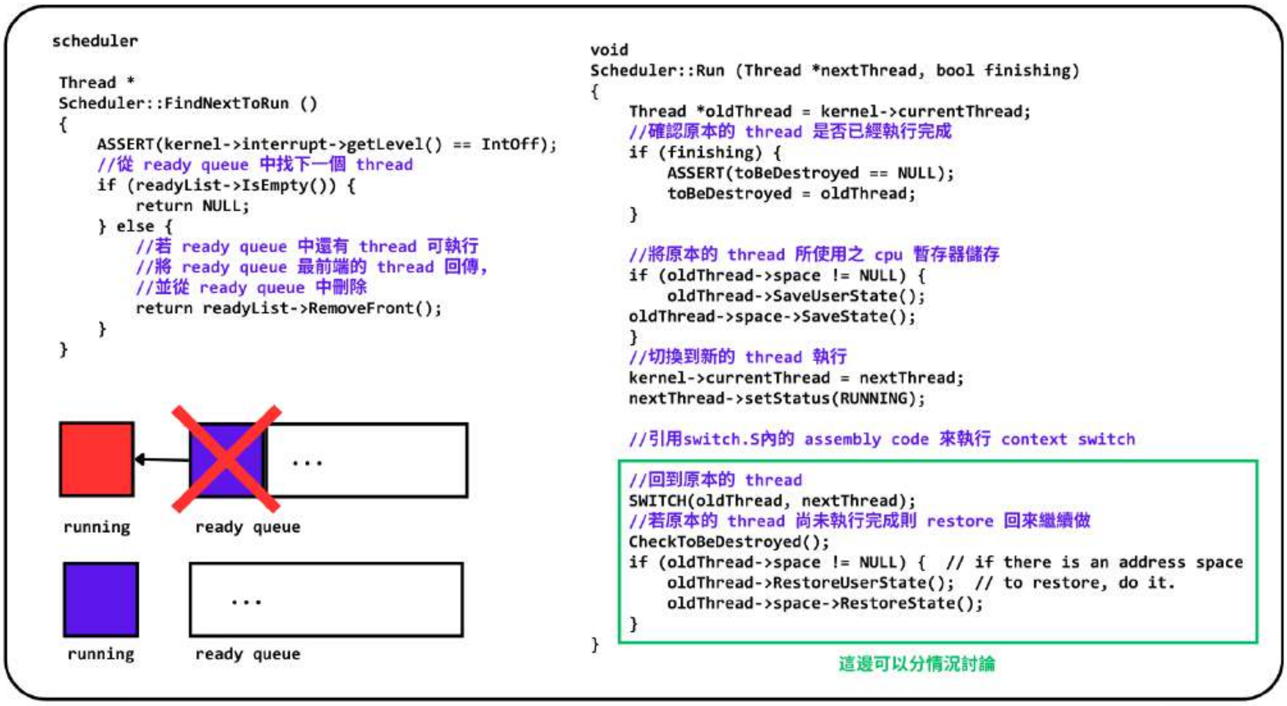
```
void
Thread::Finish ()
{
    Sleep(TRUE);
}
```

```
void
Thread::Sleep (bool finishing)
{
    //將thread BLOCK進入休眠
    status = BLOCKED;
    //去scheduler找下個thread執行，無則idle
    while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
      kernel->interrupt->Idle();
    }
    //找到，執行下一個 thread
    kernel->scheduler->Run(nextThread, finishing);
}
```

```
Thread *
Scheduler::FindNextToRun ()
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    //從 ready queue 中找下一個 thread
    if (readyList->IsEmpty()) {
        return NULL;
    } else {
        //若 ready queue 中還有 thread 可執行
        //將 ready queue 最前端的 thread 回傳，
        //並從 ready queue 中刪除
        return readyList->RemoveFront();
    }
}
```

```
void
Scheduler::Run (Thread *nextThread, bool finishing)
{
    Thread *oldThread = kernel->currentThread;
    //將原本的 thread 所使用之 cpu 暫存器儲存
    if (oldThread->space != NULL) {
         oldThread->SaveUserState();
    oldThread->space->SaveState();
    }
    //切換到新的 thread 執行並刪除原本的 thread
    kernel->currentThread = nextThread;
    nextThread->setStatus(RUNNING);
    SWITCH(oldThread, nextThread);
    CheckToBeDestroyed();
}
```

Ready -> Running

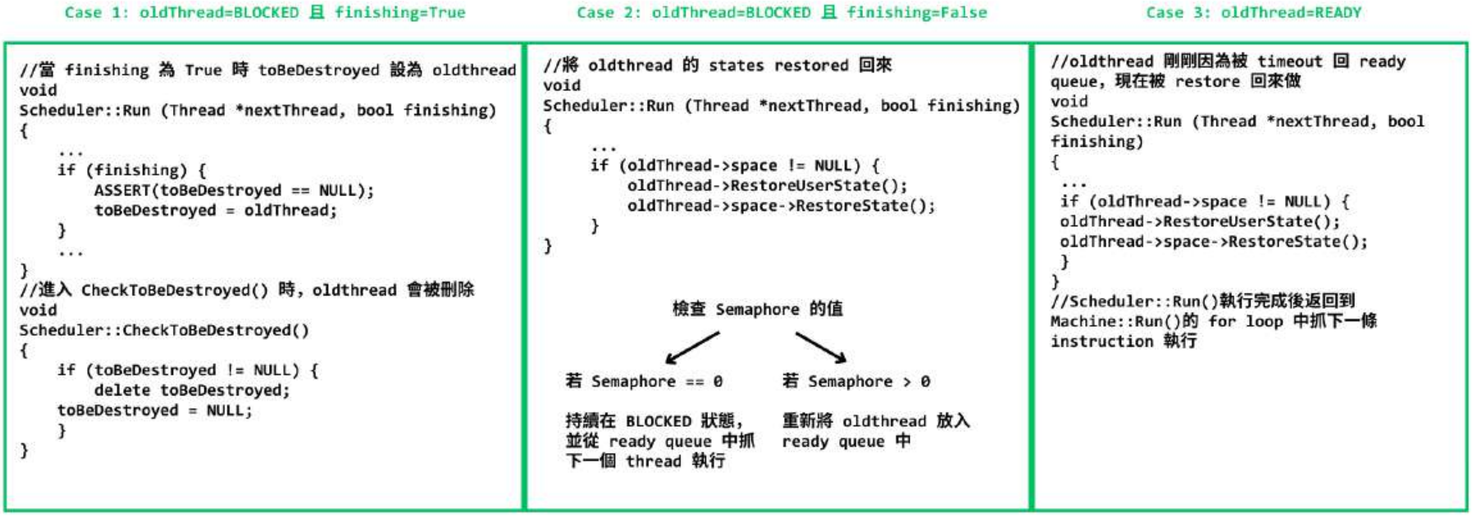


scheduler

```
Thread *
Scheduler::FindNextToRun ()
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    //從 ready queue 中找下一個 thread
    if (readyList->IsEmpty()) {
        return NULL;
    } else {
        //若 ready queue 中還有 thread 可執行
        //將 ready queue 最前端的 thread 回傳,
        //並從 ready queue 中刪除
        return readyList->RemoveFront();
    }
}
```

```
void
Scheduler::Run (Thread *nextThread, bool finishing)
{
    Thread *oldThread = kernel->currentThread;
    //確認原本的 thread 是否已經執行完成
    if (finishing) {
        ASSERT(toBeDestroyed == NULL);
        toBeDestroyed = oldThread;
    }

    //將原本的 thread 所使用之 cpu 暫存器儲存
    if (oldThread->space != NULL) {
        oldThread->SaveUserState();
    oldThread->space->SaveState();
    }
    //切換到新的 thread 執行
    kernel->currentThread = nextThread;
    nextThread->setStatus(RUNNING);

    //引用switch.S內的 assembly code 來執行 context switch

    //回到原本的 thread
    SWITCH(oldThread, nextThread);
    //若原本的 thread 尚未執行完成則 restore 回來繼續做
    CheckToBeDestroyed();
    if (oldThread->space != NULL) {  // if there is an address space
        oldThread->RestoreUserState();  // to restore, do it.
        oldThread->space->RestoreState();
    }
}
```

這邊可以分情況討論



Case 1: oldThread=BLOCKED 且 finishing=True

```
//當 finishing 為 True 時 toBeDestroyed 設為 oldthread
void
Scheduler::Run (Thread *nextThread, bool finishing)
{
    ...
    if (finishing) {
        ASSERT(toBeDestroyed == NULL);
        toBeDestroyed = oldThread;
    }
    ...
}
//進入 CheckToBeDestroyed() 時, oldthread 會被刪除
void
Scheduler::CheckToBeDestroyed()
{
    if (toBeDestroyed != NULL) {
        delete toBeDestroyed;
    toBeDestroyed = NULL;
    }
}
```

Case 2: oldThread=BLOCKED 且 finishing=False

```
//將 oldthread 的 states restored 回來
void
Scheduler::Run (Thread *nextThread, bool finishing)
{
     ...
     if (oldThread->space != NULL) {
         oldThread->RestoreUserState();
         oldThread->space->RestoreState();
     }
}
```

檢查 Semaphore 的值

若 Semaphore == 0

持續在 BLOCKED 狀態, 並從 ready queue 中抓下一個 thread 執行

若 Semaphore > 0

重新將 oldthread 放入 ready queue 中

Case 3: oldThread=READY

```
//oldthread 剛剛因為被 timeout 回 ready
queue, 現在被 restore 回來做
void
Scheduler::Run (Thread *nextThread, bool
finishing)
{
 ...
 if (oldThread->space != NULL) {
 oldThread->RestoreUserState();
 oldThread->space->RestoreState();
 }
}
//Scheduler::Run()執行完成後返回到
Machine::Run()的 for loop 中抓下一條
instruction 執行
```

Part 2:Contribution

1. Describe details and percentage of each member’s contribution.

姓名	負責項目	貢獻度
何翊華	Trace code part (i), (v), (vi) + assist part (ii), (iii), (iv)	50%
廖偉辰	Trace code part (ii), (iii), (iv) + assist part (i), (v), (vi)	50%