作業系統 - MP3 – CPU scheduling

tags NachOS

---

studentID: 1102003S

workshop account: os23s77

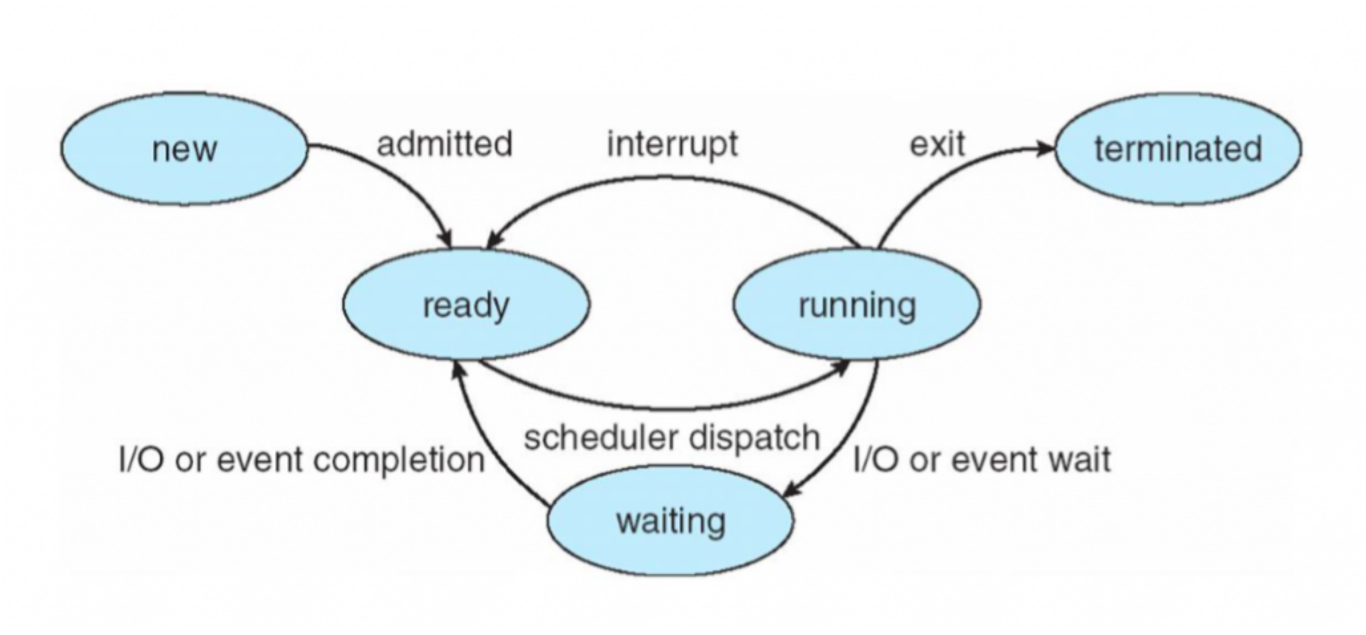
teamID: 陳觀宇 (kcem104@gmail.com)

name: 陳觀宇

HackMD link: LINK

Part 1 Trace Code

Process(Thread) State Diagram



- Process 在其生命週期中，共有 5 個狀態

1. New：當 process 被創建時，將 code 讀到記憶體中，並將前述記憶體狀態初始化
2. Ready： Process 競爭的資源是 CPU 的核心，管理的方式為佇列 (Queue)，在等待被執行的階段，會被放在佇列當中，此狀態稱為 Ready

3. Running：被 CPU 排程選到，執行 instructions。OS 為了確保主控權，一段時間就會將 CPU switch 給 scheduler 而==打斷(interrupt)==程序
4. Waiting：有些指令不會直接使用 CPU (如 I/O)，因為不需要 CPU ，又需等待某些指令完成，所以進入 Waiting 狀態，待完成後重新放回 Ready
5. Terminated：Process 完成執行，釋放資源給 OS。一個 Processor 一次只執行一個 Process；但同時可能有多個 Process 處於 Ready 或 Waiting 的狀態。

1-1. New->Ready

1-1. New→Ready

$$\begin{tikzcd}
\text{Kernel::ExecAll()} \arrow[d] \\
\text{Kernel::Exec(char*)} \arrow[d] \\
\text{Thread::Fork(VoidFunctionPtr, void*)} \arrow[d] \\
\text{Thread::StackAllocate(VoidFunctionPtr, void*)} \arrow[d] \\
\text{Scheduler::ReadyToRun(Thread*)}
\end{tikzcd}$$

1-1. threads/kernel.cc Kernel::ExecAll()

- 根據`threads/kernel.cc`文件說明，`Kernel::ExecAll()`是讓每個文件調用Exec()函數執行所有已載入的可執行文件(execfile)，並在所有文件被執行後終止當前線程。
- 簡述實作步驟:
  - 1.執行所有已載入的可執行文件(execfile)
  - 2.所有文件執行後終止當前thread

```
void Kernel::ExecAll()
{
    for (int i=1;i<=execfileNum;i++) {
        // 利用for迴圈遍歷系統中每個可執行的文件，
        // 從1到execfileNum（已加載的可執行文件的總數）。
        // 對於每個文件，該方法使用相應的文件名作為參數調用Exec()函數。
        int a = Exec(execfile[i]);
                // Exec()函數創建一個新的線程來執行指定的可執行文件，並返回一個整數值，
                // 指示操作的成功或失敗。每次迭代中，該整數值存儲在變量"a"中。
    }
    currentThread->Finish();
        // 當所有可執行文件都被執行後，通過調用Finish()函數終止當前線程。
        // 該函數將當前線程的狀態設置為THREAD_ZOMBIE，並安排下一個線程運行。
}
```

1-1. threads/kernel.cc Kernel::Exec(char* name)

- 根據`threads/kernel.cc`文件說明，`Kernel::Exec()`函式創建一個新的執行緒來執行指定的可執行檔，在分叉該執行緒，在新執行緒中執行該文件之前設置其地址空間。
- 實作步驟依序為:
  - 1.建立新的thread
  - 2.設定新thread的地址空間AddrSpace
  - 3.呼叫Fork傳遞該thread參數
  - 4.回傳新建立的thread對應的threadNum

```
int Kernel::Exec(char* name)
{
    t[threadNum] = new Thread(name, threadNum);
        // 使用 "new" 為新執行緒創建內存，並將執行緒名稱和編號作為參數傳遞。
    t[threadNum]->space = new AddrSpace();
        // 使用 "new" 為新執行緒的地址空間分配內存。
    t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);
        // 函式在新執行緒上調用 Fork() 方法，
        // 將 ForkExecute() 函式的地址作為第一個參數，
        // 將新執行緒的指針作為第二個參數傳遞。
        // ForkExecute() 是負責在新執行緒中執行指定可執行檔的函式。
    threadNum++;
        // 增加 threadNum 變數的值

    return threadNum-1;
        // 返回新創建的執行緒的編號。
}
```

1-1. threads/thread.cc Thread::Fork(VoidFunctionPtr, void*)

- 根據`threads/thread.cc`文件說明，`Thread::Fork()`負責創建一個新線程並將其添加到ready queue就緒隊列中，從而允許caller與callee線程並行執行。
- Fork()有兩個參數，一個是VoidFunctionPtr類型的funtion pointer func，表示new thread需要執行的function；另一個是void類型的pointer arg，表示func function的參數。
- 實作步驟依序為:
  - 1. 分配出一塊stack空間 (Allocate a stack)。
  - 2. 初始化stack以便call去switch讓它執行procedure (Initialize the stack so that a call to SWITCH will cause it to run the procedure)。
  - 3. 設定interrupt Level為IntOff(禁用)，並且用oldLevel記錄原本中斷狀態。
  - 4. 將thread放入ready queue (Put the thread on the ready queue)。
  - 5. 恢復interrupt Level為oldLevel(原本中斷狀態)。

```
//----------------------------------------------------------------------
// Thread::Fork
//  Invoke (*func)(arg), allowing caller and callee to execute
//  concurrently.
//
//  NOTE: although our definition allows only a single argument
//  to be passed to the procedure, it is possible to pass multiple
//  arguments by making them fields of a structure, and passing a pointer
//  to the structure as "arg".
//
//  Implemented as the following steps:
//      1. Allocate a stack
//      2. Initialize the stack so that a call to SWITCH will
//      cause it to run the procedure
//      3. Put the thread on the ready queue
//
//  "func" is the procedure to run concurrently.
//  "arg" is a single argument to be passed to the procedure.
//----------------------------------------------------------------------


void
Thread::Fork(VoidFunctionPtr func, void *arg)
{
    Interrupt *interrupt = kernel->interrupt;
    Scheduler *scheduler = kernel->scheduler;
    // 建立interrupt與scheduler的pointer變數
    // 個別紀錄指向當前kernel的中斷interrupt和排程scehduler的pointer。
    IntStatus oldLevel;
    // 建立oldLevel變數，用來記錄中斷級別原本的狀態
    StackAllocate(func, arg);
    // StackAllocate() 函數來為線程分配一個新堆棧stack，
    // 並初始化它以運行指定的函數和給定的參數。

    oldLevel = interrupt->SetLevel(IntOff);
    // 透過SetLevel(IntOff)將中斷級別設置為禁用，
    // 並且回傳原本中斷級別狀態用oldLevel變數做紀錄
    scheduler->ReadyToRun(this);
    // ReadyToRun assumes that interrupts
    // are disabled!
    // Scheduler::ReadyToRun() 將新線程添加到就緒隊列中
    (void) interrupt->SetLevel(oldLevel);
    // 透過SetLevel(oldLevel)恢復中斷級別到其先前的狀態。
}
```

1-1. threads/thread.cc Thread::StackAllocate(VoidFunctionPtr, void*)

- 根據`threads/thread.cc`文件說明，`Thread::StackAllocate()`負責為新thread分配和初始化執行棧，函數最後返回一個指向新執行棧的pointer，並且該pointer將用於在後續的thread創建中指定執行棧。
- 實作步驟依序為:
    - 1. 使用AllocBoundedArray分配一塊大小為 StackSize * sizeof(int) 的內存給新線程的執行棧。
    - 2. 然後依據不同的平台設置執行棧的大小和結構。 其中將stackTop設定為stack頂端位置。 在某些平台上有利用FENCEPOST值做stack溢出檢查做安全措施。
    - 3. 函式將初始化好的機器狀態（machine state）寫入新線程對應的 machineState 數組。其中包括：
        - (1) PCState：程序計數器（Program Counter），指向執行棧中下一條要執行的指令。
        - (2) StartupPCState：啟動程序計數器，指向 ThreadBegin 函式。
        - (3) InitialPCState：初始程序計數器，指向要 fork 的函式 func。

- (4) InitialArgState：初始引數狀態，傳給 func 的參數 arg。
- (5) WhenDonePCState：執行完畢程式計數器狀態，當線程完成時跳轉的指令，指向 ThreadFinish 函式。

```
//----------------------------------------------------------------------
// Thread::StackAllocate
//  Allocate and initialize an execution stack.  The stack is
//  initialized with an initial stack frame for ThreadRoot, which:
//      enables interrupts
//      calls (*func)(arg)
//      calls Thread::Finish
//
//  "func" is the procedure to be forked
//  "arg" is the parameter to be passed to the procedure
//----------------------------------------------------------------------

void
Thread::StackAllocate (VoidFunctionPtr func, void *arg)
{
    stack = (int *) AllocBoundedArray(StackSize * sizeof(int));
    // 使用AllocBoundedArray分配了一個大小為StackSize的整數數組stack

    // 接著根據不同的架構（PARISC、SPARC、PowerPC、DECMIPS、ALPHA、x86）
    // 來定位棧頂stackTop，
    // 並將STACK_FENCEPOST（一個值為0xc5c5c5c5的特殊標記）存儲到棧的最頂端。
#ifdef PARISC
    // HP stack works from low addresses to high addresses
    // everyone else works the other way: from high addresses to low addresses
    stackTop = stack + 16;
    // HP requires 64-byte frame marker
    stack[StackSize - 1] = STACK_FENCEPOST;
#endif

#ifdef SPARC
    stackTop = stack + StackSize - 96;
    // SPARC stack must contains at
    // least 1 activation record
    // to start with.
    *stack = STACK_FENCEPOST;
#endif

#ifdef PowerPC // RS6000
    stackTop = stack + StackSize - 16;
    // RS6000 requires 64-byte frame marker
    *stack = STACK_FENCEPOST;
#endif

#ifdef DECMIPS
    stackTop = stack + StackSize - 4;
    // -4 to be on the safe side!
    *stack = STACK_FENCEPOST;
#endif
#ifdef ALPHA
    stackTop = stack + StackSize - 8;
    // -8 to be on the safe side!
    *stack = STACK_FENCEPOST;
#endif


#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

    // 根據當前架構的不同，將一些機器狀態（如PCState、StartupPCState等）
    // 初始化為一些特定的地址（如ThreadRoot、ThreadBegin、func等）。
    // 這個新線程的機器狀態被存儲在machineState數組中，以便在線程切換時使用。
#ifdef PARISC
    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
}
```

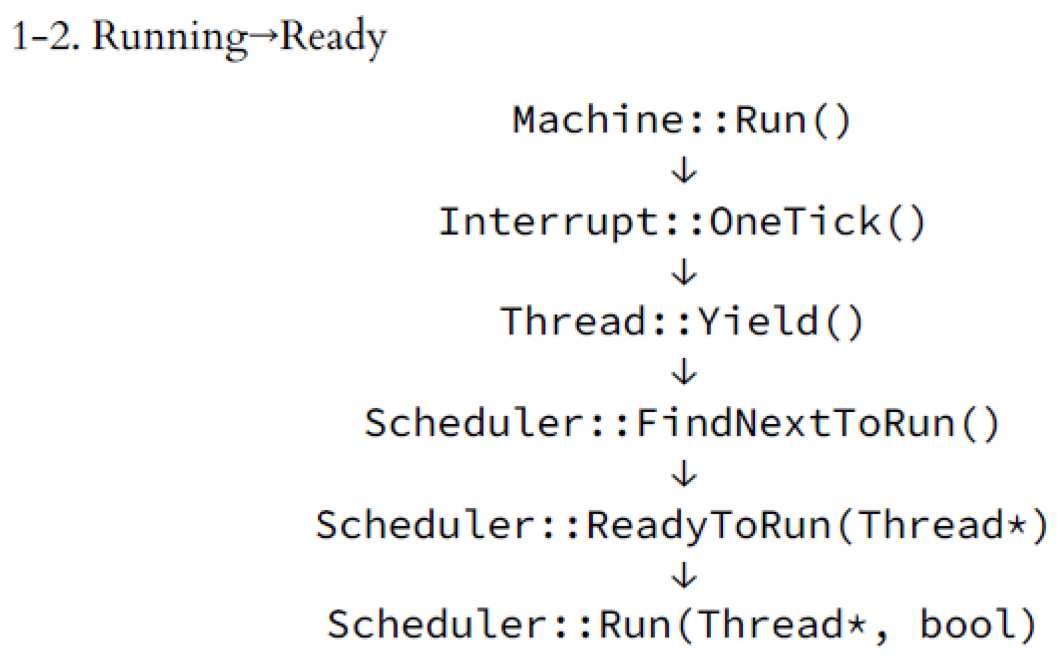
1-1. threads/thread.cc Scheduler::ReadyToRun(Thread*)

- 根據`threads/scheduler.cc`文件說明，`Scheduler::ReadyToRun()`是將指定的 thread 標記為 ready（準備執行），但還不會被執行，並將它放到 readyList 上等待後續被schedular安排到 CPU 上執行。
- ReadyToRun()傳入參數是要標記為 ready 的 thread。
- 實作步驟依序為:
  - 1. 檢查中斷是否已禁用，因為如果啟用了中斷，修改就準備好的列表可能會導致競爭條件。
  - 2. 將線程的狀態設置為 READY，表示此thread已經準備好執行。
  - 3. 將該線程加到準備列表readyList的末尾。

```
//----------------------------------------------------------------------
// Scheduler::ReadyToRun
//  Mark a thread as ready, but not running.
//  Put it on the ready list, for later scheduling onto the CPU.
```

```
//
//  "thread" is the thread to be put on the ready list.
//----------------------------------------------------------------------

void
Scheduler::ReadyToRun (Thread *thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    // 檢查當前中斷的狀態是否為關閉狀態。
    // 在加入就緒隊列之前必須禁用中斷，以避免競爭條件和不一致的狀態。
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
    //cout << "Putting thread on ready list: " << thread->getName() << endl ;
    thread->setStatus(READY);
    // 將 thread 的狀態設為 READY
    readyList->Append(thread);
    // 將 thread 添加到 readyList 之中，
    // 等待schedular決定何時執行該 thread
}
```

1-2. Running->Ready


1-2. Running→Ready
Machine::Run()
Interrupt::OneTick()
Thread::Yield()
Scheduler::FindNextToRun()
Scheduler::ReadyToRun(Thread*)
Scheduler::Run(Thread*, bool)

1-2. machine/mipssim.cc Machine::Run()

- 根據`mipssim.cc`文件說明，`Machine::Run()`是模擬Nashos執行user-level program，當program啟動時，會由kernel呼叫Machine::Run()執行，且此函數不會返回(CPU不斷讀取指令執行)。
- 實作步驟依序為:
  - (1) 建立Instruction class pointer instr紀錄instruction的物件記憶體位置。
  - (2) 將Mode設定為UserMode。
  - (3) 進入無窮迴圈，透過OneInstruction(instr)function傳入instruction，模擬MIPS CPU逐行執行指令。
  - (4) 利用OneTick()確認是否有其他interrupt已經到達預定執行時間，需要優先處理。

```
//----------------------------------------------------------------------
// Machine::Run
//  Simulate the execution of a user-level program on Nachos.
//  Called by the kernel when the program starts up; never returns.
//
//  This routine is re-entrant, in that it can be called multiple
//  times concurrently -- one for each thread executing user code.
//----------------------------------------------------------------------

void
Machine::Run()
{
    Instruction *instr = new Instruction;
    // storage for decoded instruction
    // Instruction對象被實例化為解碼指令所需的存儲區域。

    if (debug->IsEnabled('m')) {
        cout << "Starting program in thread: " << kernel->currentThread-
>getName();
    cout << ", at time: " << kernel->stats->totalTicks << "\n";
    }
    kernel->interrupt->setStatus(UserMode);
    // 將mode設定為UserMode
    for (;;) {
    // 該迴圈是一個無限迴圈，一直到程式執行完畢，
    // 也就是當 `OneInstruction()` 函數返回時，程式會結束。
    DEBUG(dbgTraCode, "In Machine::Run(), into OneInstruction " << "== Tick " <<
kernel->stats->totalTicks << " ==");
        OneInstruction(instr);
        // 每次執行會調用 OneInstruction()將下一條指令讀入並解碼，
        // 並根據指令類型執行相應的操作。
    DEBUG(dbgTraCode, "In Machine::Run(), return from OneInstruction  " << "==
Tick " << kernel->stats->totalTicks << " ==");

    DEBUG(dbgTraCode, "In Machine::Run(), into OneTick " << "== Tick " << kernel-
>stats->totalTicks << " ==");
    kernel->interrupt->OneTick();
        // 調用 OneTick()模擬系統的時鐘中斷。
    DEBUG(dbgTraCode, "In Machine::Run(), return from OneTick " << "== Tick " <<
kernel->stats->totalTicks << " ==");
    if (singleStep && (runUntilTime <= kernel->stats->totalTicks))
    // 如果 singleStep標誌被設置為 true 並且運行時間達到 runUntilTime，
    // Debugger()` 函數將被調用，允許用戶進入調試模式。
        Debugger();

    }
}
```

1-2. machine/interrupt.cc Interrupt::OneTick()

- 根據interrupt.cc文件說明，Interrupt::OneTick()是將計算模擬未來的某個時間點，是否有任何pending interrupt需要被呼叫。因此Interrupt::OneTick()具有模擬時間未來時間的行為，並且根據情況設定interrupt狀態，並釋放目前Thread，然後改成執行下一個Thread。
- 實作步驟依序為:
    - (1) 根據當前模擬器的狀態更新總時間。
    - (2) 檢查是否有任何待處理的中斷，如果有的話會將中斷層級從IntOff轉換為IntOn，以觸發中斷處理程序。
    - (3) 檢查是否有到期的計時器中斷，如果有，則調用中斷處理程序。
    - (4) 恢復中斷層級，使程序能夠接收其他中斷。
    - (5) 如果yieldOnReturn標誌被設置，則表示定時器設備處理程序要求上下文切換，這時就可以進行切換，並且會恢復到先前的中斷層級狀態。

```
//----------------------------------------------------------------------
// Interrupt::OneTick
//  Advance simulated time and check if there are any pending
//  interrupts to be called.
//
//  Two things can cause OneTick to be called:
//      interrupts are re-enabled
//      a user instruction is executed
//----------------------------------------------------------------------
void
Interrupt::OneTick()
{
    MachineStatus oldStatus = status;
    Statistics *stats = kernel->stats;

    // advance simulated time
    // 根據當前模擬器的狀態更新總時間
    if (status == SystemMode) {
        // 如果當前的狀態是系統模式（即核心代碼執行期間），則會增加SystemTick
        stats->totalTicks += SystemTick;
    stats->systemTicks += SystemTick;
    } else {
        // 如果當前的狀態是使用者模式，則增加`UserTick`。
    stats->totalTicks += UserTick;
    stats->userTicks += UserTick;
    }
    DEBUG(dbgInt, "== Tick " << stats->totalTicks << " ==");

    // 檢查是否有任何待處理的中斷，
    // 如果有的話會將中斷層級從IntOff轉換為`IntOn，
    // 以觸發中斷處理程序。
    // 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
        // 如果yieldOnReturn標誌被設置，
        // 則表示定時器設備處理程序要求上下文切換，這時就可以進行切換，
        // 並且會恢復到先前的中斷層級狀態。
    yieldOnReturn = FALSE;
    status = SystemMode;
    kernel->currentThread->Yield();
        // yield is a kernel routine
    status = oldStatus;
    }
}
```

1-2. threads/thread.cc Thread::Yield()

- `Thread::Yield()`是當前執行緒主動放棄CPU，讓其他已經準備好的執行緒進行運行的功能。當前執行緒被插入就緒佇列的末尾，等待後續被重新調度。
- 實作步驟依序為:
  - (1) 關閉中斷:關閉中斷是為了讓下面的操作能原子地執行，避免多線程並發的情況。
  - (2) 確認當前執行緒是正在運行的線程:使用 `ASSERT` 斷言語句確認當前正在運行的線程是否是調用該函數的線程。
  - (3) 查找下一個準備運行的線程:使用`kernel->scheduler->FindNextToRun()`方法查找下一個準備運行的線程。
  - (4) 如果找到了下一個線程，將當前線程插入到就緒佇列的末尾，執行下一個線程。
  - (5) 恢復中斷原本的狀態，並結束函數:使用`kernel->interrupt->SetLevel(oldLevel)`方法恢復中斷原本的狀態，並結束函數。

```
//----------------------------------------------------------------------
// Thread::Yield
//  Relinquish the CPU if any other thread is ready to run.
//  If so, put the thread on the end of the ready list, so that
//  it will eventually be re-scheduled.
//
//  NOTE: returns immediately if no other thread on the ready queue.
//  Otherwise returns when the thread eventually works its way
//  to the front of the ready list and gets re-scheduled.
//
//  NOTE: we disable interrupts, so that looking at the thread
//  on the front of the ready list, and switching to it, can be done
//  atomically.  On return, we re-set the interrupt level to its
//  original state, in case we are called with interrupts disabled.
//
//  Similar to Thread::Sleep(), but a little different.
//----------------------------------------------------------------------

void
Thread::Yield ()
```

```
{
    Thread *nextThread;
    IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);
    // 為了避免多線程並發的情況，
    // 用kernel->interrupt->SetLevel(IntOff)關閉中斷，
    // 使得該操作可以原子執行。

    ASSERT(this == kernel->currentThread);

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

    nextThread = kernel->scheduler->FindNextToRun();
    // 從調度器中查找下一個準備運行的執行緒

    if (nextThread != NULL) {
        // 若查找到了下一個執行緒，
    kernel->scheduler->ReadyToRun(this);
        // 當前執行緒被插入到就緒佇列的末尾
    kernel->scheduler->Run(nextThread, FALSE);
        // 調用 kernel->scheduler->Run(nextThread, FALSE)執行下一個執行緒
    }
    (void) kernel->interrupt->SetLevel(oldLevel);
        // 恢復中斷原本的狀態，並結束函數。
}
```

1-2. threads/scheduler.cc Scheduler::FindNextToRun()

- `Scheduler::FindNextToRun()`用於找到下一個準備被調度到 CPU 執行的線程。
- 實作步驟依序為:
    - (1) 檢查中斷是否已經被關閉，確保該操作可以原子地進行。
    - (2) 如果就緒佇列中沒有任何線程，直接返回NULL，表示沒有任何線程需要被執行。
    - (3) 否則，返回readyList的頭部，表示下一個需要被執行的線程。同時，該線程從readyList中被移除，

```
//----------------------------------------------------------------------
// Scheduler::FindNextToRun
//  Return the next thread to be scheduled onto the CPU.
//  If there are no ready threads, return NULL.
// Side effect:
//  Thread is removed from the ready list.
//----------------------------------------------------------------------

Thread *
Scheduler::FindNextToRun ()
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    // 檢查中斷是否已經被關閉，確保該操作可以原子地進行。
    // 在Nachos中，調度程序運行在中斷上下文之外，
    // 所以必須先關閉中斷以便能夠原子地查找下一個需要運行的線程。

    if (readyList->IsEmpty()) {
    return NULL;
        // 如果就緒佇列中沒有任何線程，
        // 直接返回NULL，表示沒有任何線程需要被執行。
    } else {
        return readyList->RemoveFront();
        // 否則，返回readyList的頭部，表示下一個需要被執行的線程。
        // 同時，該線程從readyList中被移除，
        // 防止在下次調度時再次被選中。
    }
}
```

1-2. threads/scheduler.cc Scheduler::ReadyToRun(Thread*)

- `Scheduler::ReadyToRun(Thread*)`是將一個線程標記為就緒，但尚未運行。也就是將該線程放在就緒列表（ready list）中，以便稍後被調度到CPU上運行。此函數中，傳遞給它的參數thread是一個指向線程的指針，它被設置為就緒狀態，並附加到就緒列表的末尾。
- 實作步驟依序為:
    - (1) 檢查中斷是否已經被關閉，確保該操作可以原子地進行。
    - (2) 設置thread為就緒狀態。
    - (3) 附加thread到就緒列表的末尾。

```
//----------------------------------------------------------------------
// Scheduler::ReadyToRun
//  Mark a thread as ready, but not running.
//  Put it on the ready list, for later scheduling onto the CPU.
//
//  "thread" is the thread to be put on the ready list.
//----------------------------------------------------------------------

void
Scheduler::ReadyToRun (Thread *thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    // 此函數的前提是當前中斷被禁用，
    // 因為在操作系統核心代碼的關鍵區域，任何時候都不能使調度器被打斷。
    // 當有新的線程被標記為就緒並被添加到就緒列表時，
    // 調度器將尋找一個最佳的線程來運行。
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
    //cout << "Putting thread on ready list: " << thread->getName() << endl ;
    thread->setStatus(READY);
    // 設置thread為就緒狀態
    readyList->Append(thread);
    // 附加thread到就緒列表的末尾
}
```

1-2. threads/scheduler.cc Scheduler::Run(Thread*, bool)

- `Scheduler::Run(Thread*, bool)`用於切換 CPU 上的執行緒。這個函數需要傳入一個 Thread 指針 `nextThread`，表示下一個要執行的線程，以及一個 bool 變量 `finishing`，表示當前線程是否需要在切

換時被刪除。
- 實作步驟依序為:
    - (1) 獲取當前線程的指針。
    - (2) 檢查當前中斷是否處於關閉狀態，並使用 ASSERT 斷言確保該情況為真。
    - (3) 如果標記為finishing的布爾變量為true，則將toBeDestroyed指向當前線程，以便在後續運行時刪除它。
    - (4) 如果當前線程是用戶程序，那麼它的CPU寄存器和地址空間的狀態將被保存。
    - (5) 檢查當前線程是否發生了堆棧溢出，以防止未檢測到的堆棧溢出。
    - (6) 將 currentThread 指針切換到下一個應當被執行的線程，並將其狀態設置為 RUNNING。
    - (7) 使用 switch.s 中定義的機器依賴的組語代碼完成上下文切換，把控制權轉移到下一個線程。
    - (8) 當前線程的狀態應已更改，並且應該在這裡進行斷言以驗證中斷是否處於關閉狀態。
    - (9) CheckToBeDestroyed函數用於檢查之前運行的線程是否完成並需要清理。
    - (10) 如果當前線程是用戶程序，則還需要恢復它的CPU寄存器和地址空間的狀態。

```
//----------------------------------------------------------------------
// Scheduler::Run
//  Dispatch the CPU to nextThread.  Save the state of the old thread,
//  and load the state of the new thread, by calling the machine
//  dependent context switch routine, SWITCH.
//
//      Note: we assume the state of the previously running thread has
//  already been changed from running to blocked or ready (depending).
// Side effect:
//  The global variable kernel->currentThread becomes nextThread.
//
//  "nextThread" is the thread to be put into the CPU.
//  "finishing" is set if the current thread is to be deleted
//      once we're no longer running on its stack
//      (when the next thread starts running)
//----------------------------------------------------------------------

void
Scheduler::Run (Thread *nextThread, bool finishing)
{
    Thread *oldThread = kernel->currentThread;
    // 獲取當前線程的指針。
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    // 確保當前中斷已經被關閉（因為調度函數需要在中斷上下文之外執行）。
    if (finishing) {
        // mark that we need to delete current thread
         ASSERT(toBeDestroyed == NULL);
         toBeDestroyed = oldThread;
        // 如果 finishing 為 true，將 oldThread 標記待銷毀。
    }

    if (oldThread->space != NULL) {
        // if this thread is a user program,
        oldThread->SaveUserState();
        // save the user's CPU registers
        oldThread->space->SaveState();
    }
    // 如果當前線程是用戶程序，
    // 那麼它的CPU寄存器和地址空間的狀態將被保存。

    oldThread->CheckOverflow();
    // check if the old thread
    // had an undetected stack overflow
    // 檢查當前線程是否發生了堆棧溢出，以防止未檢測到的堆棧溢出。

    kernel->currentThread = nextThread;
    // switch to the next thread
    nextThread->setStatus(RUNNING);
    // nextThread is now running
    // 將當前線程設置為已阻止或已就緒。
    // 將當前線程設置為下一個線程，將下一個線程設置為運行狀態。

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

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

    SWITCH(oldThread, nextThread);
    // we're back, running oldThread
    // interrupts are off when we return from switch!
    // 調用 SWITCH 函數，將控制權交給下一個線程。
    // 在SWITCH完成後，線程執行完畢並返回。
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    // 當前線程的狀態應已更改，
    // 並且應該在這裡進行斷言以驗證中斷是否處於關閉狀態。

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

    CheckToBeDestroyed();
    // check if thread we were running
    // before this one has finished
    // and needs to be cleaned up
    // CheckToBeDestroyed函數用於檢查之前運行的線程是否完成並需要清理。

    if (oldThread->space != NULL) {
        // if there is an address space
        oldThread->RestoreUserState();
        // to restore, do it.
        oldThread->space->RestoreState();
        //如果當前線程是用戶程序，
        //則還需要恢復它的CPU寄存器和地址空間的狀態。
    }
}
```

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

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

SynchConsoleOutput::PutChar(char)
↓
Semaphore::P()
↓
List<T>::Append(T)
↓
Thread::Sleep(bool)
↓
Scheduler::FindNextToRun()
↓
Scheduler::Run(Thread*, bool)

1-3. userprog/synchconsole.cc SynchConsoleOutput::PutChar(char)

- `SynchConsoleOutput::PutChar(char ch)`實現了同步的輸出控制臺。
- 實作步驟依序為:
    - (1) 首先獲取一個鎖，以確保同一時間只有一個線程可以訪問控制臺。
    - (2) 調用consoleOutput的PutChar方法來寫入一個字符到控制臺中。
    - (3) waitFor的P()操作等待信號，當信號到達時，該線程再次獲取鎖並退出。
- 在所有線程完成之前，調用PutChar的線程都會被阻塞，以確保輸出被正確地序列化到控制臺。該方法遵循嚴格的獲取鎖和釋放鎖協議，以避免死鎖和其他同步問題的發生。

```
//----------------------------------------------------------------------
// SynchConsoleOutput::PutChar
//      Write a character to the console display, waiting if necessary.
//----------------------------------------------------------------------

void
SynchConsoleOutput::PutChar(char ch)
{
    lock->Acquire();
    // 首先獲取一個鎖，以確保同一時間只有一個線程可以訪問控制臺。
    consoleOutput->PutChar(ch);
    // 調用consoleOutput的PutChar方法來寫入一個字符到控制臺中。
    waitFor->P();
    // waitFor的P()操作等待信號
    lock->Release();
    // 當信號到達時，該線程再次獲取鎖並退出。
}
```

1-3. threads/synch.cc Semaphore:😛()

- `Semaphore::P()`用來減少信號量的值，如果當前的信號量值為0，那麼當前執行緒會進入睡眠狀態，直到有其他執行緒釋放信號量，將其值增加至大於0，被阻塞的執行緒才會被喚醒。在此期間，中斷會被關閉，以保證原子操作。waitFor的P()操作等待信號，當信號到達時，該線程再次獲取鎖並退出。
- 實作步驟依序為:
  - (1) 紀錄當前中斷狀態與線程。
  - (2) 將中斷關閉，直到有其他執行緒調用V操作釋放了信號量。
  - (3) 進入while循環，如果信號量的值等於0，當前的執行緒會被加入到等待隊列中，調用Sleep()方法進入睡眠狀態。
  - (4) 直到信號量的值大於0才退出循環，減少信號量的值並將中斷恢復到原來的狀態。

```
//----------------------------------------------------------------------
// Semaphore::P
//  Wait until semaphore value > 0, then decrement.  Checking the
//  value and decrementing must be done atomically, so we
//  need to disable interrupts before checking the value.
//
//  Note that Thread::Sleep assumes that interrupts are disabled
//  when it is called.
//----------------------------------------------------------------------

void
Semaphore::P()
{
    DEBUG(dbgTraCode, "In Semaphore::P(), " << kernel->stats->totalTicks);
    Interrupt *interrupt = kernel->interrupt;
    // 取得中斷處理物件
    Thread *currentThread = kernel->currentThread;
    // 取得當前的執行緒
    // disable interrupts
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    // 關閉中斷
    // 如果信號量的值等於0，則進入while循環等待，
    while (value == 0) {
        // 如果信號量的值等於0，
        // 當前的執行緒會被加入到等待隊列中，
        // semaphore not available
    queue->Append(currentThread);
        // 進入睡眠狀態
        // so go to sleep
    currentThread->Sleep(FALSE);

    }
    value--;
    // semaphore available, consume its value
    // 如果信號量的值大於0，則可以消耗信號量，並且減少信號量的值

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

1-3. lib/list.cc List< T > ::Append(T)

- `List< T >Append()`用於將一個元素item加到列表的末尾。
- 實作步驟依序為:
  - (1) 創建一個ListElement< T >元素，用於存儲要添加的item。
  - (2) ASSERT()檢查確保item不在List內
  - (3) 如果列表是空的，則將first和last指向這個元素。
  - (4) 將該元素插入到最後一個元素後面，然後將last指向這個元素。
  - (5) 增加列表的計數器numInList。
  - (6) ASSERT()檢查是否已經成功添加了元素item。

```
//----------------------------------------------------------------------
// List<T>::Append
//      Append an "item" to the end of the list.
//
//  Allocate a ListElement to keep track of the item.
//      If the list is empty, then this will be the only element.
//  Otherwise, put it at the end.
//
//  "item" is the thing to put on the list.
//----------------------------------------------------------------------

template <class T>
void
List<T>::Append(T item)
{
    ListElement<T> *element = new ListElement<T>(item);
    // 創建一個ListElement< T >元素，用於存儲要添加的item。

    ASSERT(!IsInList(item));
    // ASSERT()檢查確保item不在List內
    if (IsEmpty()) {
        // list is empty
        // 如果列表是空的，
        // 則將first和last指向這個元素。
    first = element;
    last = element;
    } else {
        // else put it after last
        // 如果列表不是空的，
        // 將該元素插入到最後一個元素後面，
        // 然後將last指向這個元素。
    last->next = element;
    last = element;
    }
    numInList++;
    // 增加列表的計數器numInList。
    ASSERT(IsInList(item));
    // ASSERT()檢查是否已經成功添加了元素item。
}
```

1-3. threads/scheduler.cc Thread::Sleep(bool)

- `Thread::Sleep(bool)`會讓當前執行緒釋放CPU，因為當前執行緒可能已經完成工作，或者因為它正在等待一個同步變數(Semaphore、Lock、Condition)而被阻塞。如果是後者，最終某個執行緒會喚醒該線程，並將其放回就緒隊列中，以便可以重新調度。
- 如果沒有任何就緒的執行緒，這意味著我們沒有可用的執行緒運行。此時會調用`Interrupt::Idle`函式，意味著我們應該空閒CPU，直到下一個I/O中斷發生(這是唯一可能導致執行緒變為就緒狀態的事件)。
- 我們假設中斷已經被禁用，因為該函式是從同步例程中調用的，這些例程必須禁用中斷以實現原子操作。我們需要關閉中斷，這樣在從就緒列表中提取第一個線程和切換到它之間不會有時間片。
- 實作步驟依序為:
  - (1) 斷言(assertion)的檢查，確保當前的執行緒是正在運行的線程，並且中斷已經被禁用。
  - (2) 將當前線程的狀態設置為BLOCKED，表示當前線程已經被阻塞。
  - (3) 函式進入一個while循環，不斷地查找下一個可運行的線程(nextThread)。
  - (4) 如果沒有可運行的線程，則調用kernel->interrupt->Idle()函式，進入空閒狀態，直到下一個I/O中斷到來。
  - (5)當下一個可運行線程被找到時，函式將調用kernel->scheduler->Run()函式，將下一個線程(nextThread)加入就緒隊列中，並且將當前線程的狀態設置為READY，以便它可以重新被調度運行。

```
//----------------------------------------------------------------------
// Thread::Sleep
//  Relinquish the CPU, because the current thread has either
//  finished or is blocked waiting on a synchronization
//  variable (Semaphore, Lock, or Condition).  In the latter case,
//  eventually some thread will wake this thread up, and put it
//  back on the ready queue, so that it can be re-scheduled.
//
//  NOTE: if there are no threads on the ready queue, that means
//  we have no thread to run.  "Interrupt::Idle" is called
//  to signify that we should idle the CPU until the next I/O interrupt
//  occurs (the only thing that could cause a thread to become
//  ready to run).
//
//  NOTE: we assume interrupts are already disabled, because it
//  is called from the synchronization routines which must
//  disable interrupts for atomicity.   We need interrupts off
//  so that there can't be a time slice between pulling the first thread
//  off the ready list, and switching to it.
//----------------------------------------------------------------------
void
Thread::Sleep (bool finishing)
{
    Thread *nextThread;

    ASSERT(this == kernel->currentThread);
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    // 斷言(assertion)的檢查，
    // 確保當前的執行緒是正在運行的線程，並且中斷已經被禁用。

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

    status = BLOCKED;
    // 將當前線程的狀態設置為BLOCKED，表示當前線程已經被阻塞。
    //cout << "debug Thread::Sleep " << name << "wait for Idle\n";
    while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
        //函式進入一個while循環，不斷地查找下一個可運行的線程(nextThread)。
        kernel->interrupt->Idle();
        // no one to run, wait for an interrupt
        // 如果沒有可運行的線程，則調用kernel->interrupt->Idle()函式，進入空閒狀態，直
到下一個I/O中斷到來。

    }
    // returns when it's time for us to run
    kernel->scheduler->Run(nextThread, finishing);
    // 當下一個可運行線程被找到時，函式將調用kernel->scheduler->Run()函式，
    // 將下一個線程(nextThread)加入就緒隊列中，
    // 並且將當前線程的狀態設置為READY，以便它可以重新被調度運行。
}
```

1-3. threads/scheduler.cc Scheduler::FindNextToRun()

- `Scheduler::FindNextToRun()`用來從就緒隊列中取出下一個要被調度到 CPU 上運行的線程，如果就緒隊列是空的，則返回 `NULL`，否則就獲取就緒隊列的第一個元素（下一個要運行的線程）並將其從就緒隊列中移除，然後返回該線程的指針。
- 實作步驟依序為:
  - (1) 斷言（`ASSERT`）確保中斷已經被禁用
  - (2) 如果就緒隊列是空的，則返回 `NULL`
  - (3) 否則就獲取就緒隊列的第一個元素（下一個要運行的線程）並將其從就緒隊列中移除，然後返回該線程的指針。

```
//----------------------------------------------------------------------
// Scheduler::FindNextToRun
//  Return the next thread to be scheduled onto the CPU.
//  If there are no ready threads, return NULL.
// Side effect:
//  Thread is removed from the ready list.
//----------------------------------------------------------------------

Thread *
Scheduler::FindNextToRun ()
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    // 斷言（`ASSERT`）確保中斷已經被禁用

    if (readyList->IsEmpty()) {
        return NULL;
        // 如果就緒隊列是空的，則返回 `NULL`
    } else {
        return readyList->RemoveFront();
        // 否則就獲取就緒隊列的第一個元素（下一個要運行的線程）
        // 並將其從就緒隊列中移除，然後返回該線程的指針。
    }
}
```

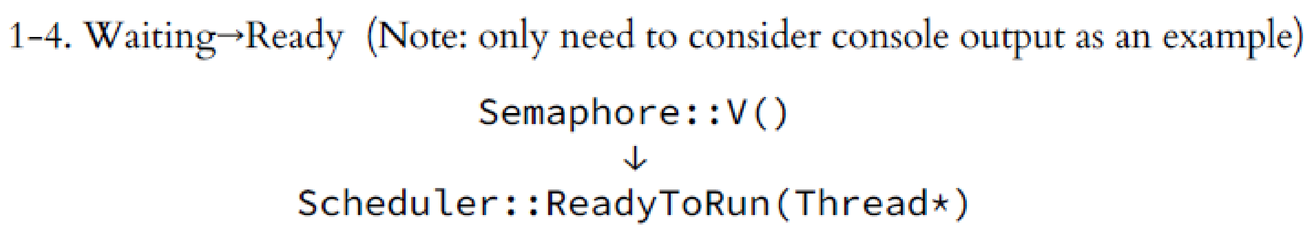
1-3. threads/scheduler.cc Scheduler::Run(Thread*, bool)

- `Scheduler::Run(Thread*, bool)`用於切換 CPU 上的執行緒。這個函數需要傳入一個 Thread 指針 `nextThread`，表示下一個要執行的線程，以及一個 bool 變量 `finishing`，表示當前線程是否需要在切換時被刪除。
- 實作步驟依序為:
  - (1) 獲取當前線程的指針。
  - (2) 檢查當前中斷是否處於關閉狀態，並使用 ASSERT 斷言確保該情況為真。
  - (3) 如果標記為finishing的布爾變量為true，則將toBeDestroyed指向當前線程，以便在後續運行時刪除它。
  - (4) 如果當前線程是用戶程序，那麼它的CPU寄存器和地址空間的狀態將被保存。
  - (5) 檢查當前線程是否發生了堆棧溢出，以防止未檢測到的堆棧溢出。
  - (6) 將 currentThread 指針切換到下一個應當被執行的線程，並將其狀態設置為 RUNNING。
  - (7) 使用 switch.s 中定義的機器依賴的組語代碼完成上下文切換，把控制權轉移到下一個線程。
  - (8) 當前線程的狀態應已更改，並且應該在這裡進行斷言以驗證中斷是否處於關閉狀態。
  - (9) CheckToBeDestroyed函數用於檢查之前運行的線程是否完成並需要清理。
  - (10) 如果當前線程是用戶程序，則還需要恢復它的CPU寄存器和地址空間的狀態。

```
//----------------------------------------------------------------------
// Scheduler::Run
//  Dispatch the CPU to nextThread.  Save the state of the old thread,
//  and load the state of the new thread, by calling the machine
//  dependent context switch routine, SWITCH.
//
//      Note: we assume the state of the previously running thread has
//  already been changed from running to blocked or ready (depending).
// Side effect:
//  The global variable kernel->currentThread becomes nextThread.
//
//  "nextThread" is the thread to be put into the CPU.
//  "finishing" is set if the current thread is to be deleted
//      once we're no longer running on its stack
//      (when the next thread starts running)
//----------------------------------------------------------------------

void
Scheduler::Run (Thread *nextThread, bool finishing)
{
    Thread *oldThread = kernel->currentThread;
    // 獲取當前線程的指針。
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    // 確保當前中斷已經被關閉（因為調度函數需要在中斷上下文之外執行）。
    if (finishing) {
        // mark that we need to delete current thread
         ASSERT(toBeDestroyed == NULL);
         toBeDestroyed = oldThread;
        // 如果 finishing 為 true，將 oldThread 標記待銷毀。
    }

    if (oldThread->space != NULL) {
        // if this thread is a user program,
        oldThread->SaveUserState();
        // save the user's CPU registers
        oldThread->space->SaveState();
    }
    // 如果當前線程是用戶程序，
    // 那麼它的CPU寄存器和地址空間的狀態將被保存。

    oldThread->CheckOverflow();
    // check if the old thread
    // had an undetected stack overflow
    // 檢查當前線程是否發生了堆棧溢出，以防止未檢測到的堆棧溢出。

    kernel->currentThread = nextThread;
    // switch to the next thread
    nextThread->setStatus(RUNNING);
    // nextThread is now running
    // 將當前線程設置為已阻止或已就緒。
    // 將當前線程設置為下一個線程，將下一個線程設置為運行狀態。

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

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

    SWITCH(oldThread, nextThread);
    // we're back, running oldThread
    // interrupts are off when we return from switch!
    // 調用 SWITCH 函數，將控制權交給下一個線程。
    // 在SWITCH完成後，線程執行完畢並返回。
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    // 當前線程的狀態應已更改，
    // 並且應該在這裡進行斷言以驗證中斷是否處於關閉狀態。

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

    CheckToBeDestroyed();
    // check if thread we were running
    // before this one has finished
    // and needs to be cleaned up
    // CheckToBeDestroyed函數用於檢查之前運行的線程是否完成並需要清理。
```

```
        if (oldThread->space != NULL) {
            // if there is an address space
            oldThread->RestoreUserState();
            // to restore, do it.
            oldThread->space->RestoreState();
            //如果當前線程是用戶程序，
            //則還需要恢復它的CPU寄存器和地址空間的狀態。
        }
    }
```

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



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

$$\begin{tikzcd}\texttt{Semaphore::V()} \arrow[d] \\ \texttt{Scheduler::ReadyToRun(Thread*)}\end{tikzcd}$$

1-4. threads/synch.cc Semaphore::V()

- `Semaphore::V()`用於增加信號量的值，並在必要時喚醒等待的線程。
- 在V()方法和P()方法中，都需要禁用中斷，以確保操作是原子性的。
- 實作步驟依序為:
  - (1) 獲取中斷控制器的指針
  - (2) 使用Interrupt::SetLevel()方法禁用中斷，以確保該操作是原子性的。
  - (3) 如果等待隊列不為空，則從中刪除一個線程，並將其添加到可運行隊列中，使其準備運行
  - (4) 增加信號量的值
  - (5) 使用Interrupt::SetLevel()方法重新啟用中斷，將中斷的狀態設置為方法開始時的狀態。

```
//----------------------------------------------------------------------
// Semaphore::V
//  Increment semaphore value, waking up a waiter if necessary.
//  As with P(), this operation must be atomic, so we need to disable
//  interrupts.  Scheduler::ReadyToRun() assumes that interrupts
//  are disabled when it is called.
//----------------------------------------------------------------------

void
Semaphore::V()
{
    DEBUG(dbgTraCode, "In Semaphore::V(), " << kernel->stats->totalTicks);
    Interrupt *interrupt = kernel->interrupt;
    // 獲取中斷控制器的指針
    // disable interrupts
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
    // 使用Interrupt::SetLevel()方法禁用中斷，以確保該操作是原子性的。

    if (!queue->IsEmpty()) {
        // make thread ready.
        // 如果等待隊列不為空，
    kernel->scheduler->ReadyToRun(queue->RemoveFront());
        // 則從中刪除一個線程，並將其添加到可運行隊列中，使其準備運行
    }
    value++;
    // 增加信號量的值

    // re-enable interrupts
    (void) interrupt->SetLevel(oldLevel);
    // 使用Interrupt::SetLevel()方法重新啟用中斷，將中斷的狀態設置為方法開始時的狀態。
}
```

1-4. threads/scheduler.cc Scheduler::ReadyToRun(Thread*)

- `Scheduler::ReadyToRun(Thread*)`是將一個線程標記為就緒，但尚未運行。也就是將該線程放在就緒列表（ready list）中，以便稍後被調度到CPU上運行。此函數中，傳遞給它的參數thread是一個指向線程的指針，它被設置為就緒狀態，並附加到就緒列表的末尾。
- 實作步驟依序為:
  - (1) 檢查中斷是否已經被關閉，確保該操作可以原子地進行。
  - (2) 設置thread為就緒狀態。
  - (3) 附加thread到就緒列表的末尾。

```
//----------------------------------------------------------------------
// Scheduler::ReadyToRun
//  Mark a thread as ready, but not running.
//  Put it on the ready list, for later scheduling onto the CPU.
//
//  "thread" is the thread to be put on the ready list.
//----------------------------------------------------------------------

void
Scheduler::ReadyToRun (Thread *thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    // 此函數的前提是當前中斷被禁用，
    // 因為在操作系統核心代碼的關鍵區域，任何時候都不能使調度器被打斷。
    // 當有新的線程被標記為就緒並被添加到就緒列表時，
    // 調度器將尋找一個最佳的線程來運行。
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
    //cout << "Putting thread on ready list: " << thread->getName() << endl ;
    thread->setStatus(READY);
    // 設置thread為就緒狀態
    readyList->Append(thread);
    // 附加thread到就緒列表的末尾
}
```

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

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

```
ExceptionHandler(ExceptionType) case SC_Exit
↓
Thread::Finish()
↓
Thread::Sleep(bool)
↓
Scheduler::FindNextToRun()
↓
Scheduler::Run(Thread*, bool)
```

1-5. userprog/exception.cc ExceptionHandler(ExceptionType) case SC_Exit (擷取部分程式碼)

- `ExceptionHandler()`為進入NachOS kernel的進入點。當user program需要kernel協助執行syscall或是產生addressing exception或 arithmetic exception時，則會呼叫ExceptionHandler()進行處理。
- 呼叫 SC_Exit 系統調用時，表示當前執行緒已經完成它的工作，需要退出。
- function傳入參數:
  - (1) which: 造成kernel trap的原因，而which的型別為ExceptionType，其定義在Machine.h，由於此引發Exception是system call，因此傳入which參數為SyscallException。
- 實作步驟依序為:
  - (1) 讀取register r2的資料(對應system call stub code)，作為type參數。根據system call stub定義，因此type為SC_Exit對應的stub code(0)。因此system call SC_Exit的type參數是由user program將參數寫入registers，再透過kernel讀取registers r4獲得。(pass parameters in registers)
  - (2) 利用傳入參數which判斷是哪一種Exception，因為指令為system call，因此which為SyscallException。
  - (3) 第一層switch case判斷which會對應到SyscallException。
  - (4) 第二層switch case判斷type會對應到SC_Exit。
  - (5) 呼叫SysHalt()進行system call SC_Exit。

```
//----------------------------------------------------------------------
// ExceptionHandler
//  Entry point into the Nachos kernel.  Called when a user program
//  is executing, and either does a syscall, or generates an addressing
//  or arithmetic exception.
//
//  For system calls, the following is the calling convention:
//
//  system call code -- r2
//      arg1 -- r4
//      arg2 -- r5
//      arg3 -- r6
//      arg4 -- r7
//
//  The result of the system call, if any, must be put back into r2.
//
// If you are handling a system call, don't forget to increment the pc
// before returning. (Or else you'll loop making the same system call forever!)
//
//  "which" is the kind of exception.  The list of possible exceptions
//  is in machine.h.
//----------------------------------------------------------------------
void
ExceptionHandler(ExceptionType which)
{
    char ch;
    int val;
    int type = kernel->machine->ReadRegister(2);
    //從register r2取出system call code存入type
    int status, exit, threadID, programID, fileID, numChar;
    DEBUG(dbgSys, "Received Exception " << which << " type: " << type << "\n");
    DEBUG(dbgTraCode, "In ExceptionHandler(), Received Exception " << which << "
type: " << type << ", " << kernel->stats->totalTicks);
    switch (which) {
    // 進行which判斷
    case SyscallException:
    // 當which為SyscallException
    switch(type) {
        case SC_Exit:
            // 當type為SC_Exit
            DEBUG(dbgAddr, "Program exit\n");
                    val=kernel->machine->ReadRegister(4);
                    cout << "return value:" << val << endl;
            kernel->currentThread->Finish();
            // 調用currentThread的Finish()方法，
            // 這個方法會將當前線程的狀態設置為完成，
            // 並且從調度隊列中刪除。
            // 這樣就實現了當前執行緒的退出。
            break;
        default:
        cerr << "Unexpected system call " << type << "\n";
        break;
    }
    break;
    default:
        cerr << "Unexpected user mode exception " << (int)which << "\n";
        break;
    }
    ASSERTNOTREACHED();
}
```

1-5. threads/thread.cc Thread::Finish()

- `Thread::Finish()`是由執行緒(Thread)物件的`ThreadRoot`函數在執行緒完成時調用的。這個函數的作用是讓目前執行緒進入Sleep狀態，等待被調度器(Scheduler)結束，進而結束執行緒的生命週期。
- 實作步驟依序為:
    - (1) 在進入Sleep狀態之前，先將中斷關閉，以避免在Sleep期間被中斷並產生問題。
    - (2) 透過ASSERT函數驗證目前執行緒物件是否為核心(Kernel)物件的currentThread屬性，以確保在正確的執行緒內完成。

    - (3) 呼叫Sleep函數，傳入TRUE參數，以將執行緒設置為Sleep狀態並告知調度器結束執行緒。

```
//----------------------------------------------------------------------
// Thread::Finish
//  Called by ThreadRoot when a thread is done executing the
//  forked procedure.
//
//  NOTE: we can't immediately de-allocate the thread data structure
//  or the execution stack, because we're still running in the thread
//  and we're still on the stack!  Instead, we tell the scheduler
//  to call the destructor, once it is running in the context of a different
thread.
//
//  NOTE: we disable interrupts, because Sleep() assumes interrupts
//  are disabled.
//----------------------------------------------------------------------

//
void
Thread::Finish ()
{
    (void) kernel->interrupt->SetLevel(IntOff);
    // 在進入Sleep狀態之前，先將中斷關閉，
    // 以避免在Sleep期間被中斷並產生問題。
    ASSERT(this == kernel->currentThread);
    // 透過ASSERT函數驗證目前執行緒物件是否為核心(Kernel)物件的currentThread屬性，
    // 以確保在正確的執行緒內完成。

    DEBUG(dbgThread, "Finishing thread: " << name);
    Sleep(TRUE);
    // invokes SWITCH
    // not reached
    // 呼叫Sleep函數，傳入TRUE參數，
    // 以將執行緒設置為Sleep狀態並告知調度器結束執行緒。
}
```

1-5. threads/thread.cc Thread::Sleep(bool)

- `Thread::Sleep(bool)`會讓當前執行緒釋放CPU，因為當前執行緒可能已經完成工作，或者因為它正在等待一個同步變數(Semaphore、Lock、Condition)而被阻塞。如果是後者，最終某個執行緒會喚醒該線程，並將其放回就緒隊列中，以便可以重新調度。
- 如果沒有任何就緒的執行緒，這意味著我們沒有可用的執行緒運行。此時會調用`Interrupt::Idle`函式，意味著我們應該空閒CPU，直到下一個I/O中斷發生(這是唯一可能導致執行緒變為就緒狀態的事件)。
- 我們假設中斷已經被禁用，因為該函式是從同步例程中調用的，這些例程必須禁用中斷以實現原子操作。我們需要關閉中斷，這樣在從就緒列表中提取第一個線程和切換到它之間不會有時間片。
- 實作步驟依序為:
    - (1) 斷言(assertion)的檢查，確保當前的執行緒是正在運行的線程，並且中斷已經被禁用。
    - (2) 將當前線程的狀態設置為BLOCKED，表示當前線程已經被阻塞。

- (3) 函式進入一個while循環，不斷地查找下一個可運行的線程(nextThread)。
- (4) 如果沒有可運行的線程，則調用kernel->interrupt->Idle()函式，進入空閒狀態，直到下一個I/O中斷到來。
- (5)當下一個可運行線程被找到時，函式將調用kernel->scheduler->Run()函式，將下一個線程(nextThread)加入就緒隊列中，並且將當前線程的狀態設置為READY，以便它可以重新被調度運行。

```
//----------------------------------------------------------------------
// Thread::Sleep
//  Relinquish the CPU, because the current thread has either
//  finished or is blocked waiting on a synchronization
//  variable (Semaphore, Lock, or Condition).  In the latter case,
//  eventually some thread will wake this thread up, and put it
//  back on the ready queue, so that it can be re-scheduled.
//
//  NOTE: if there are no threads on the ready queue, that means
//  we have no thread to run.  "Interrupt::Idle" is called
//  to signify that we should idle the CPU until the next I/O interrupt
//  occurs (the only thing that could cause a thread to become
//  ready to run).
//
//  NOTE: we assume interrupts are already disabled, because it
//  is called from the synchronization routines which must
//  disable interrupts for atomicity.   We need interrupts off
//  so that there can't be a time slice between pulling the first thread
//  off the ready list, and switching to it.
//----------------------------------------------------------------------
void
Thread::Sleep (bool finishing)
{
    Thread *nextThread;

    ASSERT(this == kernel->currentThread);
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    // 斷言(assertion)的檢查，
    // 確保當前的執行緒是正在運行的線程，並且中斷已經被禁用。

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

    status = BLOCKED;
    // 將當前線程的狀態設置為BLOCKED，表示當前線程已經被阻塞。
    //cout << "debug Thread::Sleep " << name << "wait for Idle\n";
    while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
        //函式進入一個while循環，不斷地查找下一個可運行的線程(nextThread)。
        kernel->interrupt->Idle();
        // no one to run, wait for an interrupt
        // 如果沒有可運行的線程，則調用kernel->interrupt->Idle()函式，進入空閒狀態，直
到下一個I/O中斷到來。

    }
    // returns when it's time for us to run
    kernel->scheduler->Run(nextThread, finishing);
    // 當下一個可運行線程被找到時，函式將調用kernel->scheduler->Run()函式，
    // 將下一個線程(nextThread)加入就緒隊列中，
    // 並且將當前線程的狀態設置為READY，以便它可以重新被調度運行。
}
```

1-5. threads/scheduler.cc Scheduler::FindNextToRun()

- `Scheduler::FindNextToRun()`用來從就緒隊列中取出下一個要被調度到 CPU 上運行的線程，如果就緒隊列是空的，則返回 `NULL`，否則就獲取就緒隊列的第一個元素（下一個要運行的線程）並將其從就緒隊列中移除，然後返回該線程的指針。
- 實作步驟依序為:
  - (1) 斷言（`ASSERT`）確保中斷已經被禁用
  - (2) 如果就緒隊列是空的，則返回 `NULL`
  - (3) 否則就獲取就緒隊列的第一個元素（下一個要運行的線程）並將其從就緒隊列中移除，然後返回該線程的指針。

```
//----------------------------------------------------------------------
// Scheduler::FindNextToRun
//  Return the next thread to be scheduled onto the CPU.
//  If there are no ready threads, return NULL.
// Side effect:
//  Thread is removed from the ready list.
//----------------------------------------------------------------------

Thread *
Scheduler::FindNextToRun ()
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    // 斷言（`ASSERT`）確保中斷已經被禁用

    if (readyList->IsEmpty()) {
        return NULL;
        // 如果就緒隊列是空的，則返回 `NULL`
    } else {
        return readyList->RemoveFront();
        // 否則就獲取就緒隊列的第一個元素（下一個要運行的線程）
        // 並將其從就緒隊列中移除，然後返回該線程的指針。
    }
}
```

1-5. threads/scheduler.cc Scheduler::Run(Thread*, bool)

- `Scheduler::Run(Thread*, bool)`用於切換 CPU 上的執行緒。這個函數需要傳入一個 Thread 指針 `nextThread`，表示下一個要執行的線程，以及一個 bool 變量 `finishing`，表示當前線程是否需要在切換時被刪除。
- 實作步驟依序為:

- (1) 獲取當前線程的指針。
- (2) 檢查當前中斷是否處於關閉狀態，並使用 ASSERT 斷言確保該情況為真。
- (3) 如果標記為finishing的布爾變量為true，則將toBeDestroyed指向當前線程，以便在後續運行時刪除它。
- (4) 如果當前線程是用戶程序，那麼它的CPU寄存器和地址空間的狀態將被保存。
- (5) 檢查當前線程是否發生了堆棧溢出，以防止未檢測到的堆棧溢出。
- (6) 將 currentThread 指針切換到下一個應當被執行的線程，並將其狀態設置為 RUNNING。
- (7) 使用 switch.s 中定義的機器依賴的組語代碼完成上下文切換，把控制權轉移到下一個線程。
- (8) 當前線程的狀態應已更改，並且應該在這裡進行斷言以驗證中斷是否處於關閉狀態。
- (9) CheckToBeDestroyed函數用於檢查之前運行的線程是否完成並需要清理。
- (10) 如果當前線程是用戶程序，則還需要恢復它的CPU寄存器和地址空間的狀態。

```
//----------------------------------------------------------------------
// Scheduler::Run
//  Dispatch the CPU to nextThread.  Save the state of the old thread,
//  and load the state of the new thread, by calling the machine
//  dependent context switch routine, SWITCH.
//
//      Note: we assume the state of the previously running thread has
//  already been changed from running to blocked or ready (depending).
// Side effect:
//  The global variable kernel->currentThread becomes nextThread.
//
//  "nextThread" is the thread to be put into the CPU.
//  "finishing" is set if the current thread is to be deleted
//      once we're no longer running on its stack
//      (when the next thread starts running)
//----------------------------------------------------------------------

void
Scheduler::Run (Thread *nextThread, bool finishing)
{
    Thread *oldThread = kernel->currentThread;
    // 獲取當前線程的指針。
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    // 確保當前中斷已經被關閉（因為調度函數需要在中斷上下文之外執行）。
    if (finishing) {
        // mark that we need to delete current thread
         ASSERT(toBeDestroyed == NULL);
         toBeDestroyed = oldThread;
        // 如果 finishing 為 true，將 oldThread 標記待銷毀。
    }

    if (oldThread->space != NULL) {
        // if this thread is a user program,
        oldThread->SaveUserState();
        // save the user's CPU registers
        oldThread->space->SaveState();
    }
    // 如果當前線程是用戶程序，
    // 那麼它的CPU寄存器和地址空間的狀態將被保存。
    oldThread->CheckOverflow();
    // check if the old thread
    // had an undetected stack overflow
    // 檢查當前線程是否發生了堆棧溢出，以防止未檢測到的堆棧溢出。

    kernel->currentThread = nextThread;
    // switch to the next thread
    nextThread->setStatus(RUNNING);
    // nextThread is now running
    // 將當前線程設置為已阻止或已就緒。
    // 將當前線程設置為下一個線程，將下一個線程設置為運行狀態。

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

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

    SWITCH(oldThread, nextThread);
    // we're back, running oldThread
    // interrupts are off when we return from switch!
    // 調用 SWITCH 函數，將控制權交給下一個線程。
    // 在SWITCH完成後，線程執行完畢並返回。
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    // 當前線程的狀態應已更改，
    // 並且應該在這裡進行斷言以驗證中斷是否處於關閉狀態。

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

    CheckToBeDestroyed();
    // check if thread we were running
    // before this one has finished
    // and needs to be cleaned up
    // CheckToBeDestroyed函數用於檢查之前運行的線程是否完成並需要清理。

    if (oldThread->space != NULL) {
        // if there is an address space
        oldThread->RestoreUserState();
        // to restore, do it.
        oldThread->space->RestoreState();
        //如果當前線程是用戶程序，
        //則還需要恢復它的CPU寄存器和地址空間的狀態。
    }
}
```

1-6. Ready→Running (Note: start from the Exit system call is called)

1-6. Ready→Running

```
Scheduler::FindNextToRun()
↓
Scheduler::Run(Thread*, bool)
2.5↓
SWITCH(Thread*, Thread*)
↓
(depends on the previous process state, e.g.,
[New,Running,Waiting]→Ready)
↓
for loop in Machine::Run()
```

1-6. threads/scheduler.cc Scheduler::FindNextToRun()

- `Scheduler::FindNextToRun()`用來從就緒隊列中取出下一個要被調度到 CPU 上運行的線程，如果就緒隊列是空的，則返回 `NULL`，否則就獲取就緒隊列的第一個元素（下一個要運行的線程）並將其從就緒隊列中移除，然後返回該線程的指針。
- 實作步驟依序為:
  - (1) 斷言（`ASSERT`）確保中斷已經被禁用
  - (2) 如果就緒隊列是空的，則返回 `NULL`
  - (3) 否則就獲取就緒隊列的第一個元素（下一個要運行的線程）並將其從就緒隊列中移除，然後返回該線程的指針。

```
//----------------------------------------------------------------------
// Scheduler::FindNextToRun
//  Return the next thread to be scheduled onto the CPU.
//  If there are no ready threads, return NULL.
// Side effect:
//  Thread is removed from the ready list.
//----------------------------------------------------------------------

Thread *
Scheduler::FindNextToRun ()
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    // 斷言（`ASSERT`）確保中斷已經被禁用

    if (readyList->IsEmpty()) {
        return NULL;
        // 如果就緒隊列是空的，則返回 `NULL`
    } else {
        return readyList->RemoveFront();
        // 否則就獲取就緒隊列的第一個元素（下一個要運行的線程）
        // 並將其從就緒隊列中移除，然後返回該線程的指針。
    }
}
```

1-6. threads/scheduler.cc Scheduler::Run(Thread*, bool)

- `Scheduler::Run(Thread*, bool)`用於切換 CPU 上的執行緒。這個函數需要傳入一個 Thread 指針 `nextThread`，表示下一個要執行的線程，以及一個 bool 變量 `finishing`，表示當前線程是否需要在切換時被刪除。
- 實作步驟依序為:
  - (1) 獲取當前線程的指針。
  - (2) 檢查當前中斷是否處於關閉狀態，並使用 ASSERT 斷言確保該情況為真。
  - (3) 如果標記為finishing的布爾變量為true，則將toBeDestroyed指向當前線程，以便在後續運行時刪除它。
  - (4) 如果當前線程是用戶程序，那麼它的CPU寄存器和地址空間的狀態將被保存。
  - (5) 檢查當前線程是否發生了堆棧溢出，以防止未檢測到的堆棧溢出。
  - (6) 將 currentThread 指針切換到下一個應當被執行的線程，並將其狀態設置為 RUNNING。
  - (7) 使用 switch.s 中定義的機器依賴的組語代碼完成上下文切換，把控制權轉移到下一個線程。
  - (8) 當前線程的狀態應已更改，並且應該在這裡進行斷言以驗證中斷是否處於關閉狀態。
  - (9) CheckToBeDestroyed函數用於檢查之前運行的線程是否完成並需要清理。
  - (10) 如果當前線程是用戶程序，則還需要恢復它的CPU寄存器和地址空間的狀態。

```
//----------------------------------------------------------------------
// Scheduler::Run
//  Dispatch the CPU to nextThread.  Save the state of the old thread,
//  and load the state of the new thread, by calling the machine
//  dependent context switch routine, SWITCH.
//
//      Note: we assume the state of the previously running thread has
//  already been changed from running to blocked or ready (depending).
// Side effect:
//  The global variable kernel->currentThread becomes nextThread.
//
//  "nextThread" is the thread to be put into the CPU.
//  "finishing" is set if the current thread is to be deleted
//      once we're no longer running on its stack
//      (when the next thread starts running)
//----------------------------------------------------------------------

void
Scheduler::Run (Thread *nextThread, bool finishing)
{
    Thread *oldThread = kernel->currentThread;
    // 獲取當前線程的指針。
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    // 確保當前中斷已經被關閉（因為調度函數需要在中斷上下文之外執行）。
    if (finishing) {
        // mark that we need to delete current thread
         ASSERT(toBeDestroyed == NULL);
         toBeDestroyed = oldThread;
        // 如果 finishing 為 true，將 oldThread 標記待銷毀。
    }

    if (oldThread->space != NULL) {
        // if this thread is a user program,
        oldThread->SaveUserState();
        // save the user's CPU registers
        oldThread->space->SaveState();
    }
    // 如果當前線程是用戶程序，
    // 那麼它的CPU寄存器和地址空間的狀態將被保存。

    oldThread->CheckOverflow();
    // check if the old thread
    // had an undetected stack overflow
    // 檢查當前線程是否發生了堆棧溢出，以防止未檢測到的堆棧溢出。

    kernel->currentThread = nextThread;
    // switch to the next thread
    nextThread->setStatus(RUNNING);
    // nextThread is now running
    // 將當前線程設置為已阻止或已就緒。
    // 將當前線程設置為下一個線程，將下一個線程設置為運行狀態。

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

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

    SWITCH(oldThread, nextThread);
    // we're back, running oldThread
    // interrupts are off when we return from switch!
    // 調用 SWITCH 函數，將控制權交給下一個線程。
    // 在SWITCH完成後，線程執行完畢並返回。
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    // 當前線程的狀態應已更改，
    // 並且應該在這裡進行斷言以驗證中斷是否處於關閉狀態。

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

    CheckToBeDestroyed();
    // check if thread we were running
    // before this one has finished
    // and needs to be cleaned up
    // CheckToBeDestroyed函數用於檢查之前運行的線程是否完成並需要清理。

    if (oldThread->space != NULL) {
        // if there is an address space
        oldThread->RestoreUserState();
        // to restore, do it.
        oldThread->space->RestoreState();
        //如果當前線程是用戶程序，
        //則還需要恢復它的CPU寄存器和地址空間的狀態。
    }
}
```

1-6. threads/thread.h SWITCH(Thread*, Thread*)

- 這是一個機器dependency的組合語言程式碼，由 `oldThread` 切換到 `newThread`。在 Nachos 中，這個函數實現了上下文切換`context switch`。
- 實作步驟依序為:
  - (1) 保存當前線程的 CPU 狀態（如寄存器）到 `oldThread` 中
  - (2) 加載 `newThread` 的 CPU 狀態，以便開始執行它。
- 因為這是一個機器相依的函數，因此其詳細內容可能因應不同的硬體而有所不同。

```
// Stop running oldThread and start running newThread
void SWITCH(Thread *oldThread, Thread *newThread);
// 此SWITCH function定義在threads/thread.h
```

1-6. threads/switch.h

- 以x86平台舉例說明context switch所需一些偏移量和定義等相關參數。在x86平台上，編譯器需要知道每個寄存器相對於線程對象的開始位置的偏移量。這樣編譯器才能將編譯後的程式碼中的寄存器指令與對應的線程實例中的實際寄存器進行匹配。
- 參數定義如下:
  - (1) 線程堆棧分配的常量。這些常量表示線程堆棧中相關狀態的位置，例如程序計數器`(Program Counter)`、`Frame Pointer`等等。在進行線程切換的時候，需要將這些狀態的值保存到線程的堆棧上或從堆棧上恢復，以便讓線程可以正確地運行。
  - (2) 定義了一些方便使用的`macro`來訪問線程狀態的偏移量和寄存器。

```
#ifdef x86

/* the offsets of the registers from the beginning of the thread object */
// 用來訪問線程狀態的偏移量
#define _ESP     0
#define _EAX     4
#define _EBX     8
#define _ECX     12
#define _EDX     16
#define _EBP     20
#define _ESI     24
#define _EDI     28
#define _PC      32

/* These definitions are used in Thread::AllocateStack(). */
// 線程堆棧分配的常量。這些常量表示線程堆棧中相關狀態的位置
#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)
```

```
// 定義了一些方便使用的`macro`來訪問線程狀態的偏移量和寄存器。
#define InitialPC       %esi
#define InitialArg      %edx
#define WhenDonePC      %edi
#define StartupPC       %ecx

#endif // x86
```

1-6. threads/switch.S

- 以x86平台舉例說明以`context switch`實現多執行緒的實作細節。
- 提供了兩個函式 `ThreadRoot` 和 `SWITCH`，以及一些相關的指令和`macro`定義。
- `ThreadRoot`是一個線程的進入點（entry point），當一個線程被創建時，它的起始函式會被傳遞給 ThreadRoot 作為參數。`ThreadRoot`會初始化一些寄存器，然後依次調用啟動函式（StartupPC）、線程函式（InitialPC）和完成函式（WhenDonePC），然後返回。
- `ThreadRoot`實作步驟依序為:
    - (1) 把 %ebp 壓入堆疊
    - (2) 把堆疊頂指標 %esp 賦值給 %ebp
    - (3) 把 InitialArg 壓入堆疊
    - (4) 調用 StartupPC，InitialPC 和 WhenDonePC 函式
    - (5) 恢復堆疊和 %ebp，返回
- `SWITCH` 函式用於在不同線程之間切換。它接受兩個線程的指針作為參數，把目前的寄存器狀態保存到第一個線程的堆疊中，然後恢復第二個線程的寄存器狀態，從而實現線程切換。
- `SWITCH`實作步驟依序為:
    - (1) 把 %eax 的值保存到 _eax_save 變量中
    - (2) 把第一個線程的指針放到 %eax 中，保存寄存器和堆疊指針，把返回地址和 %eax 的值保存到第一個線程中
    - (3) 把第二個線程的指針放到 %eax 中，從堆疊中恢復寄存器和堆疊指針，以及返回地址和 %eax 的值
    - (4) 把 _eax_save 中的值存回 %eax 中，返回。實現了兩個線程之間的切換。

```
#ifdef x86

        .text
        .align  2

        .globl  ThreadRoot
        .globl  _ThreadRoot

/* 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



/* 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 stack
        movl    _eax_save,%eax

        ret

#endif // x86
```

1-6. (depends on the previous process state, e.g.,[New,Running,Waiting]→Ready)

- 中間過程可參考上述對應process state轉換過程:
- 1-1. New→Ready
- 1-2. Running→Ready
- 1-4. Waiting→Ready

1-6. machine/mipssim.cc for loop in Machine::Run()

- 根據`mipssim.cc`文件說明，`Machine::Run()`是模擬Nashos執行user-level program，當program啟動時，會由kernel呼叫Machine::Run()執行，且此函數不會返回(CPU不斷讀取指令執行)。
- 實作步驟依序為:
    - (1) 建立Instruction class pointer instr紀錄instruction的物件記憶體位置。
    - (2) 將Mode設定為UserMode。
    - (3) 進入無窮迴圈，透過OneInstruction(instr)function傳入instruction，模擬MIPS CPU逐行執行指令。
    - (4) 利用OneTick()確認是否有其他interrupt已經到達預定執行時間，需要優先處理。

```
//----------------------------------------------------------------------
// Machine::Run
//  Simulate the execution of a user-level program on Nachos.
//  Called by the kernel when the program starts up; never returns.
//
//  This routine is re-entrant, in that it can be called multiple
//  times concurrently -- one for each thread executing user code.
//----------------------------------------------------------------------

void
Machine::Run()
{
    Instruction *instr = new Instruction;
    // storage for decoded instruction
    // Instruction對象被實例化為解碼指令所需的存儲區域。
    if (debug->IsEnabled('m')) {
        cout << "Starting program in thread: " << kernel->currentThread-
>getName();
    cout << ", at time: " << kernel->stats->totalTicks << "\n";
    }
    kernel->interrupt->setStatus(UserMode);
    // 將mode設定為UserMode
    for (;;) {
    // 該迴圈是一個無限迴圈，一直到程式執行完畢，
    // 也就是當 `OneInstruction()` 函數返回時，程式會結束。
    DEBUG(dbgTraCode, "In Machine::Run(), into OneInstruction " << "== Tick " <<
kernel->stats->totalTicks << " ==");
        OneInstruction(instr);
        // 每次執行會調用 OneInstruction()將下一條指令讀入並解碼，
        // 並根據指令類型執行相應的操作。
    DEBUG(dbgTraCode, "In Machine::Run(), return from OneInstruction  " << "==
Tick " << kernel->stats->totalTicks << " ==");

    DEBUG(dbgTraCode, "In Machine::Run(), into OneTick " << "== Tick " << kernel-
>stats->totalTicks << " ==");
    kernel->interrupt->OneTick();
        // 調用 OneTick()模擬系統的時鐘中斷。
    DEBUG(dbgTraCode, "In Machine::Run(), return from OneTick " << "== Tick " <<
kernel->stats->totalTicks << " ==");
    if (singleStep && (runUntilTime <= kernel->stats->totalTicks))
    // 如果 singleStep標誌被設置為 true 並且運行時間達到 runUntilTime，
    // Debugger()` 函數將被調用，允許用戶進入調試模式。
        Debugger();

    }
}
```

Part 2 Implement page table in NachOS

2-1. Implement a Preemptive Shortest Job First scheduling algorithm

分析spec說明，需要依序修改以下function完成SJF實作

kernel.h Kernel::Kernel()

- 在kerenl.h的kernel class新增一個bool值ContextSwitchFlag,藉此判斷是否需要做context switch

```
class Kernel {
  public:
    bool ContextSwitchFlag;
    // control timer alarm
};
```

,

alarm.cc Alarm::CallBack()

- 在alarm.cc的CallBack()原本是固定time quantum會呼叫YieldOnReturn,使目前的thread做yield進入ready state,為round robin的scheduling
- 在if判斷增加當ContextSwitchFlag為true時,代表只有需要做ContextSwitch時,才會做YieldOnReturn,因此才能實現Shortest Job First scheduling
- 進入判斷後ContextSwitchFlag要關閉,這樣ContextSwitchFlag開啟後才會只做一次YieldOnReturn

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

    // <Modified>
    if (status != IdleMode && kernel->ContextSwitchFlag) {
        // change the condition for yield
        interrupt->YieldOnReturn();
        kernel->ContextSwitchFlag = false;
        // turn off the ContextSwitch flag
    }
    // </Modified>
}
```

thread.h Thread::Thread()

- 在thread.h的Thread class(也就是thread control block)新增一些時間相關參數與function,用來做SJF scheduling 判斷與設定:
  - (1) ti : 預測的approximate burst time
  - (2) T : 上次的實際burst time
  - (3) last_Running_Start : 紀錄thread上次進入running state的time stamp
  - (4) get_T & set_T : 得到&設定預測的approximate burst time
  - (5) get_T & set_T : 得到&設定實際burst time
  - (6) get_last_Running_Start & set_last_Running_Start : 得到&設定thread上次進入running state的time stamp

```
class Thread {


  private:
    // add time parameter
    double ti;
    // predict approximation burst time (use formula to calculate, so use double)
    int T;
    // previous real burst time (clock tik decide, so use int)
    int last_Running_Start;
    // the last Time Stamp into running state
  public:
    double get_ti() {return ti;}
    // get predict burst time
    void set_ti(double x) {ti = x;}
    // set predict burst time
    int get_T(){return T;}
    // get previous real burst time
    void set_T(int x) { T = x;}
    // set previous real burst time
    int get_last_Running_Start() {return last_Running_Start;}
    // get the last Time Stamp into running state
    void set_last_Running_Start(int x) {last_Running_Start = x;}
    // set the last Time Stamp into running state

};
```

scheduler.h Scheduler::Scheduler()

- 在scheduler.h的Scheduler class將readyList的type改成SortedList,這樣才能讓readyList具有priority queue的性質

```
class Scheduler {
  private:
    SortedList<Thread *> *readyList;
    // change the type to SortedList
    // queue of threads that are ready to run, but not running
};
```

scheduler.cc Scheduler() & SortingFuntion(Thread *t1, Thread *t2)

- 在scheduler.cc的Scheduler class將readyList的sort compare function 設定為SortingFuntion,這樣才能讓readyList具有priority queue的性質
- SortingFuntion需要按照list.h定義:
    - x < y 則回傳 -1
    - x = y 則回傳 0
    - x > y 則回傳 1
- SortingFuntion傳入兩個thread的pointer,而將thread插入readyList後readyList就會按照thread的remaining burst time(ti-T)大小自動進行排序,而排序後的readyList的開頭就會是最小值,因此之後將readyList進行RemoveFront()後就能得到remaining time最小的thread

```
static int SortingFuntion(Thread *t1, Thread *t2){
    double t1_ti = t1->get_ti();
    // thread 1 predict time
    double t2_ti = t2->get_ti();
    // thread 2 predict time
    int t1_T = t1->get_T();
    // thread 1 real cpu burst time
    int t2_T = t2->get_T();
    // thread 2 real cpu burst time
    double t1_remaining_time = max(0.0, t1_ti-t1_T);
    // thread 1 remaing burst time = max(0,ti-t1_T);
    double t2_remaining_time = max(0.0, t2_ti-t2_T);
    // thread 2 remaing burst time = max(0,ti-t2_T);

    if(t1_remaining_time < t2_remaining_time){
        return -1; // x < y
    }else if(t1_remaining_time == t2_remaining_time){
        return 0; // x = y
    }else if(t1_remaining_time > t2_remaining_time){
        return 1; // x > y
    }
}

Scheduler::Scheduler()
{
    readyList = new SortedList<Thread *> (SortingFuntion);
    // use SortingFunction to do priority queue by remaining time
    toBeDestroyed = NULL;
}
```

scheduler.cc Scheduler::ReadyToRun (Thread*)

- Scheduler::ReadyToRun之中,將readyList function改成SortedList Function Insert,因為原本的append是list的function
- 透過readyList->Insert(thread)可將新進入ready state的thread放入readylist,就能做remaing burst time的排序比較
- 透過比較新進入ready state的thread跟當前執行的thread兩者的remaining burst time,若新進入ready state的thread的remaining burst time小於當前執行的thread的remaining burst time,則開啟ContextSwitchFlag,代表需要在下次呼叫alarm callback時會做YieldOnReturn(),讓當前執行的thread讓出CPU,從running state返回ready state,進行一次context switch.

```
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->Insert(thread);
    // use SortedList function Insert to find the thread with smallest remaining
burst time in priority queue
    DEBUG(dbgSJF, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" <<
thread->getID() << "] is inserted into queue");
    // Whenever a process is inserted into a queue,print debug [A]
    if (max(0.0, thread->get_ti()-thread->get_T()) < max(0.0, kernel-
>currentThread->get_ti()-kernel->currentThread->get_T())){
        // new thread preempt currentthread if new thread'remaining time is less
than current thread's
        kernel->ContextSwitchFlag = true;
    }
}
```

thread.cc Thread::Yield ()

- 新增set_T(kernel->stats->totalTicks - get_last_Running_Start() + get_T()),其代表需要更新T(total burst time),T要增加目前到上次進入到running之間的時間
- 當set_T一旦執行,remaining burst time會跟著改變,因此readylist會按照新的remaining time進行重新排序

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

    ASSERT(this == kernel->currentThread);

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

    set_T(kernel->stats->totalTicks - get_last_Running_Start() + get_T());
    // accumulate the real burst time T because the thread jump out of the running
state, and get into ready state
    // when the real burst time is changed, the readylist will use the updated
real burst time to do the sort again
    // so the remaining time is changed too

    double ti = get_ti();
    // get current approximate burst time
    int T = get_T();
    // get current real burst time
    DEBUG(dbgSJF, "[C] Tick [" << kernel->stats->totalTicks << "]: Thread [" <<
getID() << "] update approximate burst time, from: [" << ti << "], add [" << T <<
    "], to [" << max(0.0, ti-T) << "]");
    // Whenever a process updates its approximate burst time:
    // because the T is changed, so the approximate burst time(remaining burst
time) is changed too.
    // this debug definition is unclear
    // ti: approximate burst time
    // T: real burst time
    // max(0.0, ti-T): remaining burst time

    nextThread = kernel->scheduler->FindNextToRun();
    // get the nextThread

    if (nextThread != NULL) {
        DEBUG(dbgSJF, "[E] Tick [" << kernel->stats->totalTicks << "]: Thread ["
<< nextThread->getID() <<
```

```
        "] is now selected for execution, thread [" << getID() << "] is preempted,
and it has executed [" << T << "] ticks")
        // Whenever a context switch occurs with preemption
        // the current thread has executed T (real burst time)
        kernel->scheduler->ReadyToRun(this);
        // put this thread into ready state and readylist
        kernel->scheduler->Run(nextThread, FALSE);
        // run nextThread
    }
    (void) kernel->interrupt->SetLevel(oldLevel);
}
```

Scheduler.cc Scheduler::Run(Thread*, bool)

- 新增nextThread->set_last_Running_Start(kernel->stats->totalTicks),代表當有新的thread從ready state再次進入running state時,需要更新此thread上一次進入running state的time stamp l(ast_Running_Start)

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

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

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

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

    oldThread->CheckOverflow();         // check if the old thread
                        // had an undetected stack overflow

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

    nextThread->set_last_Running_Start(kernel->stats->totalTicks);
    // update the last time stamp when thread get into the running state

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

    // This is a machine-dependent assembly language routine defined
    // in switch.s.  You may have to think
    // a bit to figure out what happens after this, both from the point
    // of view of the thread and from the perspective of the "outside world".
    SWITCH(oldThread, nextThread);
    // we're back, running oldThread

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

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

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

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

thread.cc Thread::Sleep (bool finishing)

- 新增set_T(kernel->stats->totalTicks - get_last_Running_Start() + get_T()),其代表需要更新T(total burst time),T要增加目前到上次進入到running之間的時間
- 當set_T一旦執行,remaining burst time會跟著改變,因此readylist會按照新的remaining time進行重新排序
- 因為thread進入到sleep代表已完成一個完整的cpu burst time,因此已經得到一個完整的total real burst time T,因此需要依照ti = 0.5*T* + *0.5*ti-1的公式更新approximate predict burst time ti,並且在計算完後將T歸零,之後T需要重新累積計算下一次的total real burst time

```
void
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);

    status = BLOCKED;

    set_T(kernel->stats->totalTicks - get_last_Running_Start() + get_T());
    // accumulate the real burst time T because the thread jump out of the running
state, and get into ready state
    // when the real burst time is changed, the readylist will use the updated
real burst time to do the sort again
    // so the remaining time is changed too
    double ti = get_ti();
    // get current approximate burst time
    int T = get_T();
    // get current real burst time
    DEBUG(dbgSJF, "[C] Tick [" << kernel->stats->totalTicks << "]: Thread [" <<
getID() << "] update approximate burst time, from: [" << ti << "], add [" << T <<
    "], to [" << 0.5*ti + 0.5*T << "]");
    // Whenever a process updates its approximate burst time:
    // because the T is changed, so the approximate burst time(remaining burst
time) is changed too.
    // this debug definition is unclear
    // ti: approximate burst time
    // T: real burst time
    // max(0.0, ti-T): remaining burst time
    set_ti(0.5*get_ti() + 0.5*get_T());
    // use the updated total real burst time T to get the current predict burst
time ti
    set_T(0);
    // reset the total real burst time T to 0

    //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
    }
    // returns when it's time for us to run
    DEBUG(dbgSJF, "[D] Tick ["<< kernel->stats->totalTicks << "]: Thread ["<<
nextThread->getID() <<
    "] is now selected for execution, thread [" << getID() << "] starts IO, and it
has executed [" <<
    T << "] ticks");
    // Whenever a context switch occurs without preemption
    // because the current thread is from running state to waiting
    kernel->scheduler->Run(nextThread, finishing);
}
```

2-2. Add a debugging flag z and use the DEBUG('z', expr) macro (defined in debug.h) to print following messages. Replace {...} to the corresponding value.

debug.h

- 在debug.h新增開啟SJF相關debug message指令'z'

```
const char dbgSJF = 'z';        // SJF debug messages
```

- (a) Whenever a process is inserted into a queue: `[A] Tick [{current total tick}]: Thread [{thread ID}] is inserted into queue`
- 當thread進入到ready state時(執行Scheduler::ReadyToRun (Thread *thread)),印出debug message [A]

```
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->Insert(thread);
    // use SortedList function Insert to find the thread with smallest remaining
burst time in priority queue
    DEBUG(dbgSJF, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" <<
thread->getID() << "] is inserted into queue");
    // Whenever a process is inserted into a queue,print debug [A]
    if (max(0.0, thread->get_ti()-thread->get_T()) < max(0.0, kernel-
>currentThread->get_ti()-kernel->currentThread->get_T())){
        // new thread preempt currentthread if new thread'remaining time is less
than current thread's
        kernel->ContextSwitchFlag = true;
    }
}
```

- (b) Whenever a process is removed from a queue: `[B] Tick [{current total tick}]: Thread [{thread ID}] is removed from queue`
- 當thread離開ready state時(執行FindNextToRun ()),印出debug message [B]

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

    if (readyList->IsEmpty()) {
        return NULL;
    } else {
        Thread* thread = readyList->RemoveFront();
        // get thread from readyList's beginning
        if (thread != NULL) {
            // if nextthread is not null
            DEBUG(dbgSJF, "[B] Tick ["<< kernel->stats->totalTicks << "]: Thread
["<< thread->getID() << "] is removed from queue");
            // then we can use getID
        }
        return thread;
        // return the thread
    }
}
```

- (c) Whenever a process updates its approximate burst time: `[C] Tick [{current total tick}]: Thread [{thread ID}] update approximate burst time, from: [{ti-1}], add [{T}], to [{ti}]`
- 當thread的total real burst T有更新時,因此approximate burst time會對應進行更新(雖然不確定這邊定義的approximate burst time是指remaining burst time ti-T還是指predict burst time ti),故執行Thread::Yield

() & Thread::Sleep (bool finishing),皆會印出debug message [c]

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

    ASSERT(this == kernel->currentThread);

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

    set_T(kernel->stats->totalTicks - get_last_Running_Start() + get_T());
    // accumulate the real burst time T because the thread jump out of the running
state, and get into ready state
    // when the real burst time is changed, the readylist will use the updated
real burst time to do the sort again
    // so the remaining time is changed too

    double ti = get_ti();
    // get current approximate burst time
    int T = get_T();
    // get current real burst time
    DEBUG(dbgSJF, "[C] Tick [" << kernel->stats->totalTicks << "]: Thread [" <<
getID() << "] update approximate burst time, from: [" << ti << "], add [" << T <<
    "], to [" << max(0.0, ti-T) << "]");
    // Whenever a process updates its approximate burst time:
    // because the T is changed, so the approximate burst time(remaining burst
time) is changed too.
    // this debug definition is unclear
    // ti: approximate burst time
    // T: real burst time
    // max(0.0, ti-T): remaining burst time

    nextThread = kernel->scheduler->FindNextToRun();
    // get the nextThread

    if (nextThread != NULL) {
        DEBUG(dbgSJF, "[E] Tick [" << kernel->stats->totalTicks << "]: Thread ["
<< nextThread->getID() <<
        "] is now selected for execution, thread [" << getID() << "] is preempted,
and it has executed [" << T << "] ticks")
        // Whenever a context switch occurs with preemption
        // the current thread has executed T (real burst time)
        kernel->scheduler->ReadyToRun(this);
        // put this thread into ready state and readylist
        kernel->scheduler->Run(nextThread, FALSE);
        // run nextThread
    }
    (void) kernel->interrupt->SetLevel(oldLevel);
}
```

```
void
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);

    status = BLOCKED;

    set_T(kernel->stats->totalTicks - get_last_Running_Start() + get_T());
    // accumulate the real burst time T because the thread jump out of the running
state, and get into ready state
    // when the real burst time is changed, the readylist will use the updated
real burst time to do the sort again
    // so the remaining time is changed too
    double ti = get_ti();
    // get current approximate burst time
    int T = get_T();
    // get current real burst time
    DEBUG(dbgSJF, "[C] Tick [" << kernel->stats->totalTicks << "]: Thread [" <<
getID() << "] update approximate burst time, from: [" << ti << "], add [" << T <<
    "], to [" << 0.5*ti + 0.5*T << "]");
    // Whenever a process updates its approximate burst time:
    // because the T is changed, so the approximate burst time(remaining burst
time) is changed too.
    // this debug definition is unclear
    // ti: approximate burst time
    // T: real burst time
    // max(0.0, ti-T): remaining burst time
    set_ti(0.5*get_ti() + 0.5*get_T());
    // use the updated total real burst time T to get the current predict burst
time ti
    set_T(0);
    // reset the total real burst time T to 0

    //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
    }
    // returns when it's time for us to run
    DEBUG(dbgSJF, "[D] Tick ["<< kernel->stats->totalTicks << "]: Thread ["<<
nextThread->getID() <<
    "] is now selected for execution, thread [" << getID() << "] starts IO, and it
has executed [" <<
    T << "] ticks");
    // Whenever a context switch occurs without preemption
    // because the current thread is from running state to waiting
    kernel->scheduler->Run(nextThread, finishing);
}
```

- (d) Whenever a context switch occurs without preemption: `[D] Tick [{current total tick}]: Thread [{new thread ID}] is now selected for execution, thread [{prev thread ID}] starts IO, and it has executed [{accumulated ticks}] ticks`
- thread從running state進入waiting state,因此需要執行Thread::Sleep (bool finishing),標記目前thread為block,並進行context switch,印出debug message [d]

```
void
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);

    status = BLOCKED;

    set_T(kernel->stats->totalTicks - get_last_Running_Start() + get_T());
    // accumulate the real burst time T because the thread jump out of the running
state, and get into ready state
    // when the real burst time is changed, the readylist will use the updated
real burst time to do the sort again
    // so the remaining time is changed too
    double ti = get_ti();
    // get current approximate burst time
    int T = get_T();
    // get current real burst time
    DEBUG(dbgSJF, "[C] Tick [" << kernel->stats->totalTicks << "]: Thread [" <<
getID() << "] update approximate burst time, from: [" << ti << "], add [" << T <<
    "], to [" << 0.5*ti + 0.5*T << "]");
    // Whenever a process updates its approximate burst time:
    // because the T is changed, so the approximate burst time(remaining burst
time) is changed too.
    // this debug definition is unclear
    // ti: approximate burst time
    // T: real burst time
    // max(0.0, ti-T): remaining burst time
    set_ti(0.5*get_ti() + 0.5*get_T());
    // use the updated total real burst time T to get the current predict burst
time ti
    set_T(0);
    // reset the total real burst time T to 0

    //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
    }
    // returns when it's time for us to run
    DEBUG(dbgSJF, "[D] Tick ["<< kernel->stats->totalTicks << "]: Thread ["<<
nextThread->getID() <<
    "] is now selected for execution, thread [" << getID() << "] starts IO, and it
has executed [" <<
    T << "] ticks");
    // Whenever a context switch occurs without preemption
    // because the current thread is from running state to waiting
    kernel->scheduler->Run(nextThread, finishing);
}
```

- (e) Whenever a context switch occurs with preemption: `[E] Tick [{current total tick}]: Thread [{new thread ID}] is now selected for execution, thread [{prev thread ID}] is preempted, and it has executed [{accumulated ticks}] ticks`
- 當前thread在running state執行到一半被迫讓出cpu回到ready state,因此需要執行Thread::Yield (),因此印出debug message [e]

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

    ASSERT(this == kernel->currentThread);

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

    set_T(kernel->stats->totalTicks - get_last_Running_Start() + get_T());
    // accumulate the real burst time T because the thread jump out of the running
state, and get into ready state
    // when the real burst time is changed, the readylist will use the updated
real burst time to do the sort again
    // so the remaining time is changed too

    double ti = get_ti();
    // get current approximate burst time
    int T = get_T();
    // get current real burst time
    DEBUG(dbgSJF, "[C] Tick [" << kernel->stats->totalTicks << "]: Thread [" <<
getID() << "] update approximate burst time, from: [" << ti << "], add [" << T <<
    "], to [" << max(0.0, ti-T) << "]");
    // Whenever a process updates its approximate burst time:
    // because the T is changed, so the approximate burst time(remaining burst
time) is changed too.
    // this debug definition is unclear
    // ti: approximate burst time
    // T: real burst time
    // max(0.0, ti-T): remaining burst time
    nextThread = kernel->scheduler->FindNextToRun();
    // get the nextThread

    if (nextThread != NULL) {
        DEBUG(dbgSJF, "[E] Tick [" << kernel->stats->totalTicks << "]: Thread ["
<< nextThread->getID() <<
        "] is now selected for execution, thread [" << getID() << "] is preempted,
and it has executed [" << T << "] ticks")
        // Whenever a context switch occurs with preemption
        // the current thread has executed T (real burst time)
        kernel->scheduler->ReadyToRun(this);
        // put this thread into ready state and readylist
        kernel->scheduler->Run(nextThread, FALSE);
        // run nextThread
    }
    (void) kernel->interrupt->SetLevel(oldLevel);
}
```

Result

原本測資 hw3t1 & hw3t2

hw3t1

```
#include "syscall.h"

int main() {
  int n;
  for (n = 1; n < 100; ++n);
  PrintInt(1);
  for (n = 1; n < 20000; ++n);
  Exit(1);
}
```

hw3t2

```
#include "syscall.h"

int main() {
  int n;
  for (n = 1; n < 10000; ++n);
  PrintInt(2);
  for (n = 1; n < 10000; ++n);
  Exit(2);
}
```

輸出結果

```
[os23s77@localhost test]$ ../build.linux/nachos -e hw3t1 -e hw3t2 -d z
hw3t1
hw3t2
[A] Tick [10]: Thread [1] is inserted into queue
[A] Tick [20]: Thread [2] is inserted into queue
[C] Tick [30]: Thread [0] update approximate burst time, from: [0], add [30], to
[15]
[B] Tick [30]: Thread [1] is removed from queue
[D] Tick [30]: Thread [1] is now selected for execution, thread [0] starts IO, and
it has executed [30] ticks
1[C] Tick [1162]: Thread [1] update approximate burst time, from: [0], add [1132],
to [566]
[B] Tick [1162]: Thread [2] is removed from queue
[D] Tick [1162]: Thread [2] is now selected for execution, thread [1] starts IO,
and it has executed [1132] ticks
[A] Tick [1172]: Thread [1] is inserted into queue
[C] Tick [111184]: Thread [2] update approximate burst time, from: [0], add
[110022], to [55011]
[B] Tick [111184]: Thread [1] is removed from queue
[D] Tick [111184]: Thread [1] is now selected for execution, thread [2] starts IO,
and it has executed [110022] ticks

[C] Tick [111194]: Thread [1] update approximate burst time, from: [566], add
[10], to [288]
[A] Tick [111195]: Thread [1] is inserted into queue
[B] Tick [111195]: Thread [1] is removed from queue
[D] Tick [111195]: Thread [1] is now selected for execution, thread [1] starts IO,
and it has executed [10] ticks
[A] Tick [111205]: Thread [2] is inserted into queue
return value:1
[C] Tick [331220]: Thread [1] update approximate burst time, from: [288], add
[220025], to [110156]
[B] Tick [331220]: Thread [2] is removed from queue
[D] Tick [331220]: Thread [2] is now selected for execution, thread [1] starts IO,
and it has executed [220025] ticks
2[C] Tick [331230]: Thread [2] update approximate burst time, from: [55011], add
[10], to [27510.5]
[A] Tick [331231]: Thread [2] is inserted into queue
[B] Tick [331231]: Thread [2] is removed from queue
[D] Tick [331231]: Thread [2] is now selected for execution, thread [2] starts IO,
and it has executed [10] ticks

[C] Tick [331241]: Thread [2] update approximate burst time, from: [27510.5], add
[10], to [13760.2]
[A] Tick [331242]: Thread [2] is inserted into queue
[B] Tick [331242]: Thread [2] is removed from queue
[D] Tick [331242]: Thread [2] is now selected for execution, thread [2] starts IO,
and it has executed [10] ticks
return value:2
[C] Tick [441267]: Thread [2] update approximate burst time, from: [13760.2], add
[110025], to [61892.6]
```