MP3_Report_47

HackMD 好讀版：連結 (https://hackmd.io/@OJo2ruXGShKdpuewtwzZcQ/ryawfBira)

團隊分工表

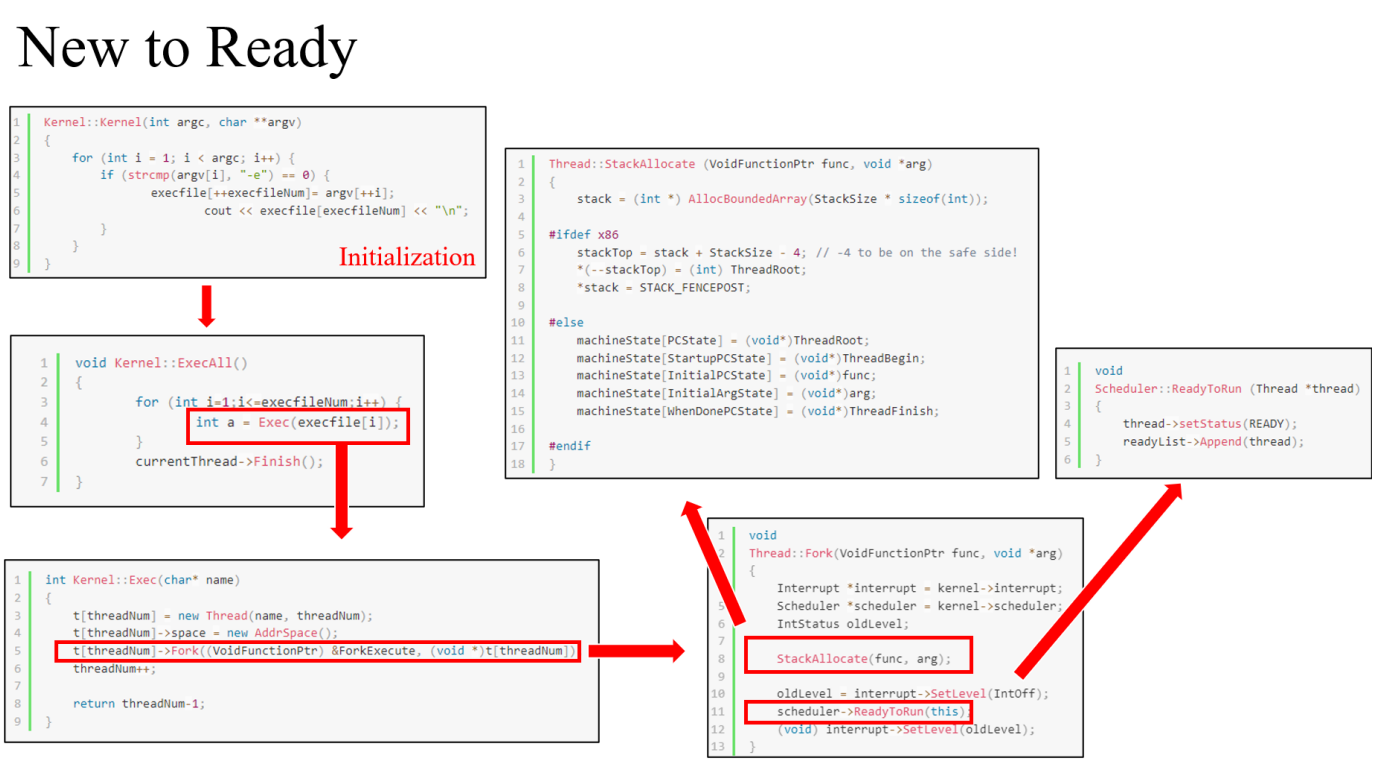
團隊成員	Trace code	文件撰寫	功能實作
111065542 楊智堯	v	v	v
111065531 郭芳妤	v	v	v

Trace code

1-1. New → Ready

```
Kernel::ExecAll()
↓
Kernel::Exec(char*)
↓
Thread::Fork(VoidFunctionPtr, void*)
↓
Thread::StackAllocate(VoidFunctionPtr, void*)
↓
Scheduler::ReadyToRun(Thread*)
```

總覽



1-1-0 Kernel::Kernel

Kernel 初始化時，會先 parse argument，遇到 `-e` 會將檔案存入 `execfile`

```
Kernel::Kernel(int argc, char **argv)
{
    for (int i = 1; i < argc; i++) {
        if (strcmp(argv[i], "-e") == 0) {
                execfile[++execfileNum]= argv[++i];
                        cout << execfile[execfileNum] << "\n";
        }
    }
}
```

1-1-1 Kernel::ExecAll

目的： 執行 kernel `-e` flag 傳過來的所有檔案。

細節：

1. Looping Through Executables： 透過 `for-loop` 遍歷全部存在 `execfile` array 裡的檔案。 `execfileNum` 追蹤可執行檔案的個數。
2. Executing Programs： `Exec()` 負責將 program load to memory 並且建立 user thread to run the program。 其存到 `a` 的 return value 是表示成功或失敗的 thread identifier。
3. Finishing the Current Thread: 任務完成，current thread 會 yield the CPU 給其他 ready to run 的 thread。

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

1-1-2 Kernel::Exec(char*)

目的： creating and starting a new thread to execute a user program。

細節：

1. Thread Creation: 建立一個名字為 `name` 新的 `Thread` object。 `threadNum` 是一個 unique identifier 讓 kernel 能夠追蹤 thread 的數量。

```
t[threadNum] = new Thread(name, threadNum);
```

2. Address Space Allocation： 指定 Program 使用的 memory space。Program's code, data, and stack 皆會 load 到這裡。

```
t[threadNum]->space = new AddrSpace();
```

3. forking the Thread： 傳入兩個參數，第一個是 function pointer `ForkExecute` ，另一個是 thead object itself `t[threadNum]` 。當 scheduler switch 到 new thread 時會直接執行 `ForkExecute` 。通常步驟為 load executable to address space, initialize CPU register, 後開始執行 user program。

```
t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);
```

4. Updating Thread Count：確保 each thread 有 unique identifier 並且記錄 thread 的數量

```
threadNum++;
```

5. Return Value： allows the calling function to keep track of the thread if necessary。會需要 -1 是因為 index 通常從 0 開始。如第一個 thread num 是 1 但實際會存在 thread[0] 的位置。

```
return threadNum-1;
```

1-1-3 Thread::Fork(VoidFunctionPtr, void*)

目的：create new thread，主要有以下三大部分：

1. Concurrency：讓 new thread 開始執行，可以跟 caller thread concurrent 執行。
2. Argument passing：可以傳遞參數到 thread function，如果有多個參數可以 passed as a structure。
3. Execution Initialization：設定初始條件，當 new thread get CPU time 時可執行某些特定的 function。

細節：

1. Stack Allocation：設定 initial stack frame 去 call `func` with provided arguments `arg`。這個設定確保當 thread 第一次被 switch 到時能夠正確的執行 function。

   ```
   StackAllocate(func, arg);
   ```

2. Disable Interrupts: 即將使用到共享的結構（ready queue）之前 disable interrupt，確保 atomicity 並避免 race condition。

   ```
   oldLevel = interrupt->SetLevel(IntOff);
   ```

3. Ready to Run: 將 thread 加入 ready queue，意旨該 thread 已經 ready to run 或被 scheduled by CPU。
   (備註：The assumption here is that interrupts must be disabled when this is done to prevent concurrent access issues.)

   ```
   scheduler->ReadyToRun(this);
   ```

4. Restore Interrupts: 恢復 fork 之前的 interrupt 的狀態。

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

1-1-4 Thread::StackAllocate(VoidFunctionPtr, void*)

目的：
設定新 Thread stack 的必要初始狀態，使得 thread 在被 scheduled 時能夠開始執行。此 function 主要做以下兩件事情：

1. Stack Allocation
   Allocate thread stack 的 memory 空間，確保有 defined area 能夠儲存 local variables, return addresses 或是持續追蹤 function calls。

2. 初始化 Stack Frame
   初始化 Stack frame 讓 Thread 執行在 `ThreadRoot`，並把控制權轉移至 thread function，並確保 function 執行完成後會 call `Thread::Finish`

以上兩件事確保 thread ready to run 會依 `ThreadRoot` 開始，執行欲執行的 function(`func`)，以及在最後結束時呼叫 `Thread::Finish` 完善地結束 thread 執行。

細節：

1. Stack Memory Allocation
   `StackSize` define the size of stack(8192)

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

2. Platform-Specific Stack Top Initialization
   使用 `#ifdef` 區分不同架構(PARISC, SPARC, PowerPC, DECMIPS, ALPHA, x86)。各架構有各自的 convention，如 stack 從高到低或低到高的記憶體位置，以及有多少空間需預留給 stack 等等。

3. 設定 Stack 邊界

```
*stack = STACK_FENCEPOST;
```

4. x86 需初始化 return address：
   將 `ThreadRoot` 位置放在 stack 的頂端，讓程式會從 `ThreadRoot` 開始執行。

```
*(--stackTop) = (int) ThreadRoot;
```

5. 設定機器狀態的 register：確保 thread 在正確的狀態及參數下執行。

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

1-1-5 Scheduler::ReadyToRun(Thread*)

目的：改變 thread 的狀態，從 `BLOCLED` 或 `JUST CREATED` 改到 `Ready`。

細節：

1. Precondition Check: 確保現在是 disable interrupt 的狀態
2. Thread Status Update：將 thread 設定為 `READY` ，意思是準備好被 scheduled 但還沒有執行。
3. Appending to Ready List：通常是 FIFO queue 讓 Scheudler 追蹤 ready to run 的 threads。

> `thread->setStatus(READY);`

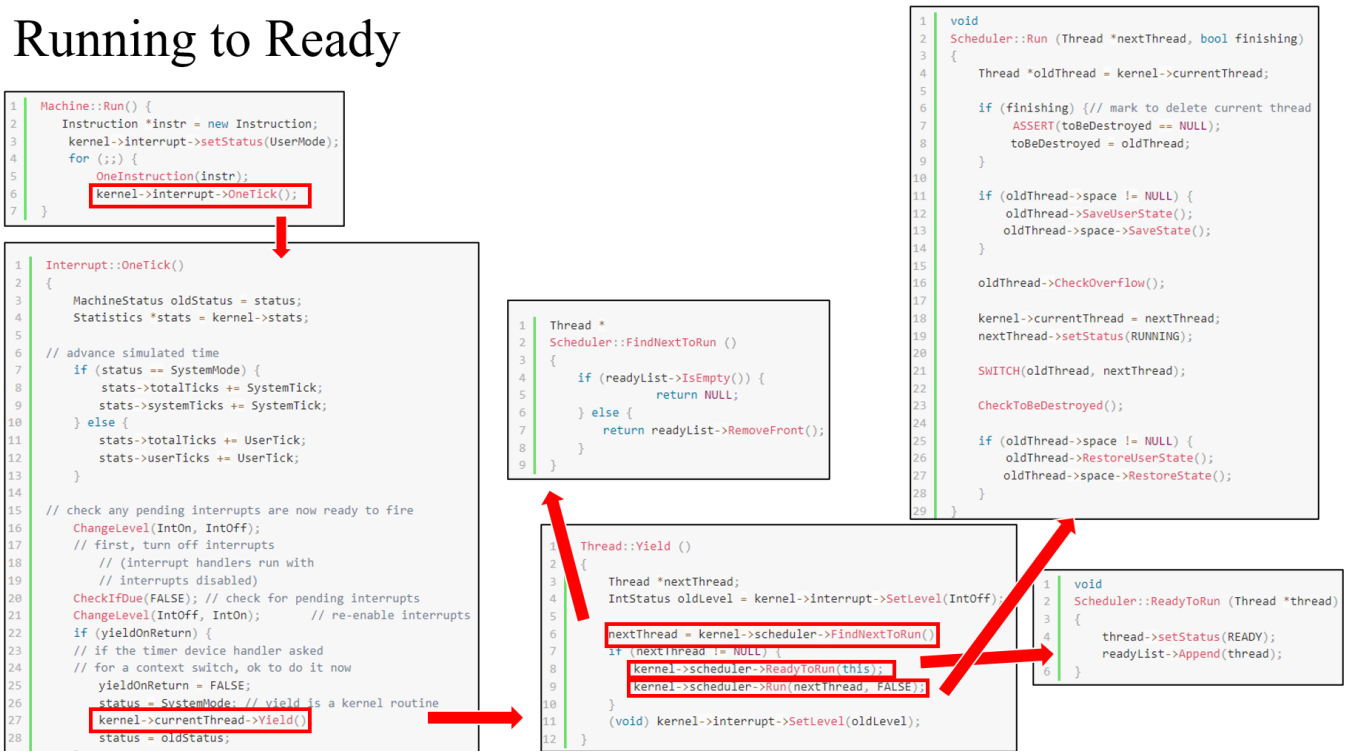
總覽：

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

1-2. Running → Ready

```
Machine::Run()
↓
Interrupt::OneTick()
↓
Thread::Yield()
↓
Scheduler::FindNextToRun()
↓
Scheduler::ReadyToRun(Thread*)
↓
Scheduler::Run(Thread*, bool)
```

總覽



1-2-1 Machine::Run()

目的：模擬 CPU 不停地執行 instruction，直到遇到 Halt 條件停下。 內容主要分成三大部分：

1. 執行迴圈：用無限迴圈 for-loop 不停地執行 instruction。
2. 追蹤和 Debug：追蹤 instruction 執行時間(ticks) 和提供各種 debugging 細節。
3. 狀態調整：將 interrupt 狀態轉成 userMode，提醒機器現在是 user-level 的 instruction 而非 kernel 或 system level。

細節：

1. Allocate 一個已經 decoded instruction 的 memory 空間，供機器執行。

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

2. 將 interrupt 狀態轉成 UserMode，提醒機器現在是 user-level 的 instruction 而非 kernel 或 system level。

```
kernel->interrupt->setStatus(UserMode);
```

3. 開始 `for(;;)` 無限執行程式，透過 OneInstruction(instr) 去獲取、解碼，並執行該 instruction， 並且把原本的 ptr 指向同個程式下一個指令。

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

4. OneTick()，用來紀錄系統時間經過一個單位（ticks），並且檢查有沒有要中斷去執行其他程式的請求，有的話會再 call 一次 run 把它做完。沒有的話就會繼續迴圈直到程式執行完。

參照原文:

```
//----------------------------------------------------------------------
// 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
//----------------------------------------------------------------------
```

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

1-2-2 Interrupt::OneTick()

目的：

1. Time Simulation：
   模擬前進一個模擬時間(one tick)
2. Interrupt Checking：
   檢查在特定時間是否有 interrupt 被觸發
3. Context Switching：
   如果有 interrupt(e.g. from timer)，
   current thread yield its exection

細節：

1. 前進一個模擬時間

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

2. 檢查 Pending Interrupt
   為確保 atomic 執行 operation，故 disabled interrupt 再檢查 pending interrupt，完成後再 enable interrupt。

```
ChangeLevel(IntOn, IntOff);
CheckIfDue(FALSE);
ChangeLevel(IntOff, IntOn);
```

3. Handle Context Switching:
   如果 `yieldOnReturn` 是 `true` ，代表有 context switch 的 request(像是 timer interrupt handler)，執行完再恢復成 oldStatus。

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

1-2-3 Thread::Yield()

目的：

- Context Switching (Context 切換):
  主要將現在的 Thread 放到 ready queue 並 context switch 執行 next queue。

細節：

1. Disale interrpt，並將原先狀態存在 `oldLevel`

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

2. 找到 Ready Queue 下個可以執行的 thread，存在 `nextThread`

```
    nextThread = kernel->scheduler->FindNextToRun();
```

3. 如果有找到 `nextThread` ，將 current thread 加回 Ready Queue，並執行 `nexrThread`

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

4. 執行完畢，調回原先的狀態 `oldLevel`

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

1-2-4 Scheduler::FindNextToRun()

目的：

- Return 下一個 CPU 執行的 thread
  - 若沒有 ready threads, return null。
- 將該 thread 從 ready queue 刪除

細節

```
Thread *
Scheduler::FindNextToRun ()
{
    if (readyList->IsEmpty()) {
                return NULL;
    } else {
        return readyList->RemoveFront();
    }
}
```

1-2-5 Scheduler::ReadyToRun(Thread*)

目的：改變 thread 的狀態，從 `BLOCLED` 或 `JUST CREATED` 改到 `Ready` 。

細節：

1. Precondition Check: 確保現在是 disable interrupt 的狀態

2. Thread Status Update：將 thread 設定為 `READY` ，意思是準備好被 scheduled 但還沒有執行。
3. Appending to Ready List：通常是 FIFO queue 讓 Scheudler 追蹤 ready to run 的 threads。

```
thread->setStatus(READY);
```

總覽：

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

1-2-6 Scheduler::Run(Thread*, bool)

目的：core operation of context switching。主要負責將 current running thread 切到 ready to run 的 new thread，並且 preserve thread state。

細節：

1. Thread Switching Precondition: capture currently thread for later restoration 並確保 operation 不會被中斷。

```
Thread *oldThread = kernel->currentThread;
ASSERT(kernel->interrupt->getLevel() == IntOff);
```

2. Handling Thread Finish: 如果執行完畢，準備 destrcut current thread，確保 cleanup 防止 memory leakage。

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

3. Saving Thread State: 保存 user program 的狀態，包含 CPU registers 和 memory 狀態。

```
if (oldThread->space != NULL) {
    oldThread->SaveUserState();
    oldThread->space->SaveState();
}
oldThread->CheckOverflow();
```

4. Executing the Context Switch: 更新 global current thread reference，並將 new thread 的狀態設為 `RUNNING` ，進一步執行 `SWITCH` 執行 contenxt switch。

```
kernel->currentThread = nextThread;
nextThread->setStatus(RUNNING);
SWITCH(oldThread, nextThread);
```

5. Restoring Thread State Post-Switch: 等到 new thread 完成後檢查是否需回復狀態。

```
ASSERT(kernel->interrupt->getLevel() == IntOff);
CheckToBeDestroyed();
if (oldThread->space != NULL) {
    oldThread->RestoreUserState();
    oldThread->space->RestoreState();
}
```

總覽：

```
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

    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();
    }
}
```

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)
```

總覽

1-3-1 SynchConsoleOutput::PutChar(char)

目的：
從 console 讀取一個 `char` 字元。此處展現 process 再 I/O 操作期間進入 `waiting` 的狀態。

細節：

1. 得到互斥的 `lock` 確保 process 安全進入共享資源區域。(Enter critical section)，並且使用 `semaphore` p，讓 process 再沒有可用字元的時候進入等待(`waiting`)狀態

```
char SynchConsoleInput::GetChar()
{
    char ch;
    lock->Acquire();
    waitFor->P();
```

2. 讀取字元

```
    ch = consoleInput->GetChar();
```

3. Leave Critical section 並且將讀取字元回傳

```
    lock->Release();
    return ch;
}
```

1-3-2 Semaphore::P()

目的：
`P()` 用於等待資源或事件。如果信號 `value` 為零，代表資源現在不可使用，Process 則需要等待。

細節：

1. Disable Interrupt: Checking the value and decrementing must be done atomically, so we need to disable interrupts before checking the value.

```
void
Semaphore::P()
{

    Interrupt *interrupt = kernel->interrupt;
    Thread *currentThread = kernel->currentThread;

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

2. 當 value == 0 意味資源不可使用時，將 process 加到 `waiting` list 並使其睡眠； 若 value != 0，則減少該值。

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

3. 檢查完畢，Enable Interrupt

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

1-3-3 List<T>::Append(T)

這個函數在 `Semaphore::P()` 中用於將當前process 添加到等待列表。

1-3-4 Thread::Sleep(bool)

目的：

- 讓現在的 Thread 放棄交出 CPU 的使用權，讓其他 Thread 可以被 Scheduled。可能發生的情況有：
  - Thread 執行完畢
  - 等待同步化 (如 Semaphore or lock)
- 處理 Idle 的狀況：如果沒有 Threads ready to run，會將系統轉換成 idle state，直到有其他 event 如 I/O 或 interrupt 讓 Thread 變成 ready to run。

細節：

1. 開始前的 Assertion

- 第一個確保現在的 Object 確實是正在執行的 Thread
- 第二個確保 interrupt disabled，該程式能夠 atomic 執行不被 external event 打斷。

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

2. 設定 Thread 狀態：將 Thread 設定為 `BLOCKED` ，表示該 Thread 並非 ready to run 或是正在等待其他資源。

```
status = BLOCKED;
```

3. Loop 檢查 ready Thread：系統持續檢查是否有 ready to run 的 Thread。若無則將系統設定為 `Idle` 。

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

4. 執行 Ready to run 的 Thread： 當系統偵測到有 ready to run 的 Thread，系統會 schedule 該 Thread 並提供其所需的 CPU 資源。 而 `finishing` 這個參數指的是 current thread 的狀態。確保現在 thread 已完成或只是 Blocked，依此做下一步指令，如 cleanup 或 special handling。

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

總覽：

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

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

    status = BLOCKED;
    while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
                kernel->interrupt->Idle(); // no one to run, wait for an interrup
        }
    // returns when it's time for us to run
    kernel->scheduler->Run(nextThread, finishing);
}
```

1-3-5 Scheduler::FindNextToRun()

目的：

- Return 下一個 CPU 執行的 thread
  - 若沒有 ready threads, return null。
- 將該 thread 從 ready queue 刪除

細節

```
Thread *
Scheduler::FindNextToRun ()
{
    if (readyList->IsEmpty()) {
                return NULL;
    } else {
        return readyList->RemoveFront();
    }
}
```

1-3-6 Scheduler::Run(Thread*, bool)

目的：core operation of context switching。主要負責將 current running thread 切到 ready to run 的 new thread，並且 preserve thread state。

細節：

1. Thread Switching Precondition: capture currently thread for later restoration 並確保 operation 不會被中斷。

```
Thread *oldThread = kernel->currentThread;
ASSERT(kernel->interrupt->getLevel() == IntOff);
```

2. Handling Thread Finish: 如果執行完畢，準備 destrcut current thread，確保 cleanup 防止 memory leakage。

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

3. Saving Thread State: 保存 user program 的狀態，包含 CPU registers 和 memory 狀態。

```
if (oldThread->space != NULL) {
    oldThread->SaveUserState();
    oldThread->space->SaveState();
}
oldThread->CheckOverflow();
```

4. Executing the Context Switch: 更新 global current thread reference，並將 new thread 的狀態設為 `RUNNING` ，進一步執行 `SWITCH` 執行 contenxt switch。

```
kernel->currentThread = nextThread;
nextThread->setStatus(RUNNING);
SWITCH(oldThread, nextThread);
```

5. Restoring Thread State Post-Switch: 等到 new thread 完成後檢查是否需回復狀態。

```
ASSERT(kernel->interrupt->getLevel() == IntOff);
CheckToBeDestroyed();
if (oldThread->space != NULL) {
    oldThread->RestoreUserState();
    oldThread->space->RestoreState();
}
```

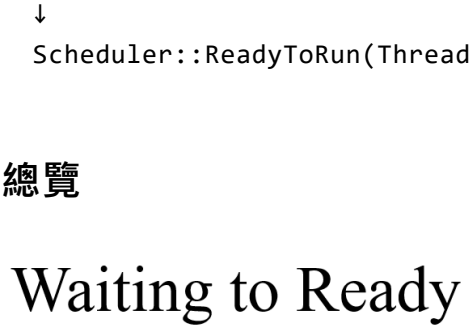
總覽：

```
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

    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();
    }
}
```

1-4. Waiting → Ready

```
Semaphore::V()
↓
Scheduler::ReadyToRun(Thread*)
```

總覽




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

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

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

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

```
void
Scheduler::ReadyToRun (Thread *thread)
{
    thread->setStatus(READY);
    readyList->Append(thread);
}
```

1-4-1 Semaphore::V()

目的：
`V()` 用於釋放資源或通知事件已發生，並允許等待該 value 的 process 繼續執行。

細節：

1. Disable Interrupt

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

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

2. 如果 `Waiting list` 有 process，執行 `ReadytoRun` 將其狀態改為 `ready` ，並且增加信號值 `value`

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

3. Enable Interrupt

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

1-4-2 Scheduler::ReadyToRun(Thread*)

目的：改變 thread 的狀態，從 `BLOCLED` 或 `JUST CREATED` 改到 `Ready`。

細節：

1. Precondition Check: 確保現在是 disable interrupt 的狀態
2. Thread Status Update：將 thread 設定為 `READY`，意思是準備好被 scheduled 但還沒有執行。
3. Appending to Ready List：通常是 FIFO queue 讓 Scheudler 追蹤 ready to run 的 threads。

```
thread->setStatus(READY);
```

總覽：

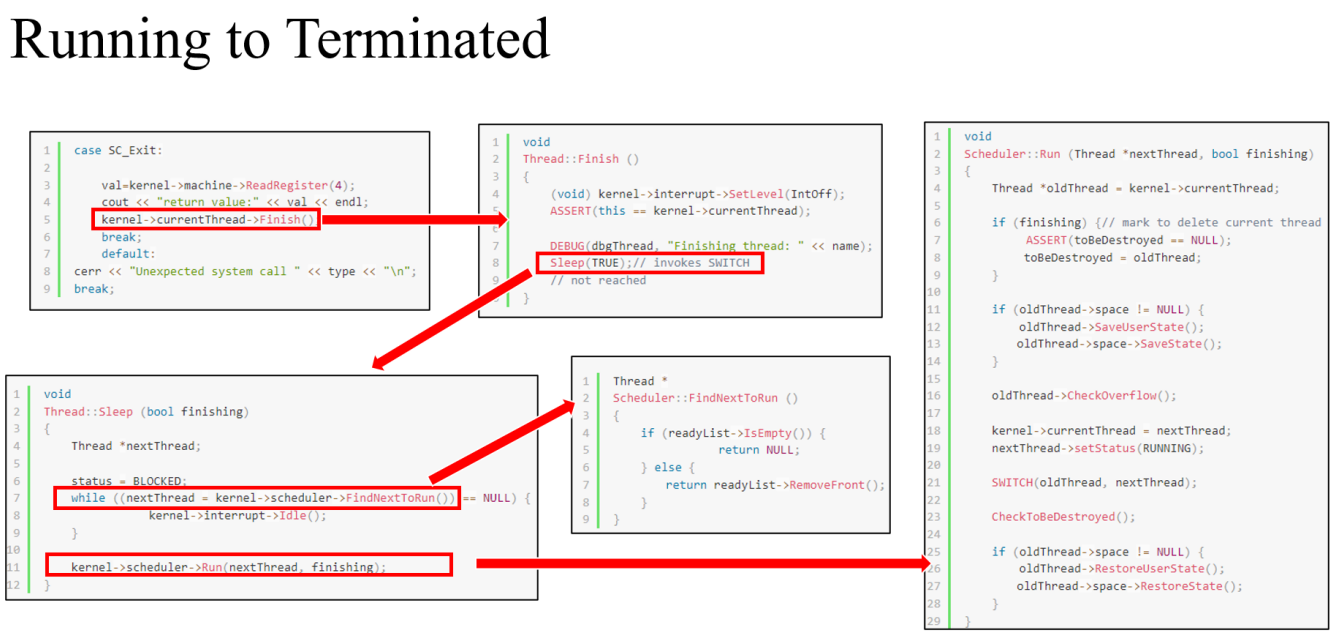
```
void
Scheduler::ReadyToRun (Thread *thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
        //cout << "Putting thread on ready list: " << thread->getName() << endl ;
    thread->setStatus(READY);
    readyList->Append(thread);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

1-5. 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)
```

總覽



1-5-1 ExceptionHandler(ExceptionType) case SC_Exit

目的：當 Process 呼叫 `Exit` ，執行 `currentThread->Finish()` 開始 process 的終止過程。

```
case SC_Exit:

    val=kernel->machine->ReadRegister(4);
    cout << "return value:" << val << endl;
    kernel->currentThread->Finish();
    break;
    default:
cerr << "Unexpected system call " << type << "\n";
break;
```

1-5-2 Thread::Finish()

目的：當 thread 結束工作時能夠正常終止。主要做三件事：

1. Clean Termination: 確保 Thread 可以被清除，各種資源的 deallocation 可以執行。
2. Safe Deallocation: 處理當 Thread 仍在執行無法 deallocate 資源的狀況。設定一個安全的機制讓 Thread 在完全停止的時候能夠 retire the thread resources.
3. 控制權轉移：當 current thread 結束後將控制權交給 scheduler 決定下一個即將執行的 thread(`Sleep` 負責這段)。

細節：

1. Disable Interrupts: 確保 current thread 可以呼叫 `Sleep` 時不擔心被插隊（preempted）。

   ```
   (void) kernel->interrupt->SetLevel(IntOff);
   ```

2. 確保正在呼叫 `Finish` 是正跑在 CPU 上的 current `Thread` object。

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

3. Sleep Call: 呼叫 `Sleep` function 告知該 Thread 已執行完畢可被 destroyed。
   –> `Sleep` 負責將 thread 從 ready queue 移除，並將控制權轉移到下一個 scheduler 選到的 thread。

   ```
   Sleep(TRUE);
   ```

總覽：

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

    DEBUG(dbgThread, "Finishing thread: " << name);
    Sleep(TRUE);// invokes SWITCH
    // not reached
}
```

1-5-3 Thread::Sleep(bool)

目的：

- 讓現在的 Thread 放棄交出 CPU 的使用權，讓其他 Thread 可以被 Scheduled。可能發生的情況有：
  - Thread 執行完畢
  - 等待同步化（如 Semaphore or lock）

- 處理 Idle 的狀況：如果沒有 Threads ready to run，會將系統轉換成 idle state，直到有其他 event 如 I/O 或 interrupt 讓 Thread 變成 ready to run。

細節：

1. 開始前的 Assertion

- 第一個確保現在的 Object 確實是正在執行的 Thread
- 第二個確保 interrupt disabled，該程式能夠 atomic 執行不被 external event 打斷。

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

2. 設定 Thread 狀態：將 Thread 設定為 `BLOCKED` ，表示該 Thread 並非 ready to run 或是正在等待其他資源。

```
status = BLOCKED;
```

3. Loop 檢查 ready Thread：系統持續檢查是否有 ready to run 的 Thread。若無則將系統設定為 `Idle` 。

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

4. 執行 Ready to run 的 Thread： 當系統偵測到有 ready to run 的 Thread，系統會 schedule 該 Thread 並提供其所需的 CPU 資源。 而 `finishing` 這個參數指的是 current thread 的狀態。確保現在 thread 已完成或只是 Blocked，依此做下一步指令，如 cleanup 或 special handling。

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

總覽：

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

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

    status = BLOCKED;
    while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
            kernel->interrupt->Idle(); // no one to run, wait for an interrup
        }
    // returns when it's time for us to run
    kernel->scheduler->Run(nextThread, finishing);
}
```

1-5-4 Scheduler::FindNextToRun()

目的：

- Return 下一個 CPU 執行的 thread
  - 若沒有 ready threads, return null。
- 將該 thread 從 ready queue 刪除

細節

```
Thread *
Scheduler::FindNextToRun ()
{
    if (readyList->IsEmpty()) {
            return NULL;
    } else {
        return readyList->RemoveFront();
    }
}
```

1-5-5 Scheduler::Run(Thread*, bool)

目的：core operation of context switching。主要負責將 current running thread 切到 ready to run 的 new thread，並且 preserve thread state。

細節：

1. Thread Switching Precondition: capture currently thread for later restoration 並確保 operation 不會被中斷。

```
Thread *oldThread = kernel->currentThread;
ASSERT(kernel->interrupt->getLevel() == IntOff);
```

2. Handling Thread Finish: 如果執行完畢，準備 destrcut current thread，確保 cleanup 防止 memory leakage。

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

3. Saving Thread State: 保存 user program 的狀態，包含 CPU registers 和 memory 狀態。

```
if (oldThread->space != NULL) {
    oldThread->SaveUserState();
    oldThread->space->SaveState();
}
oldThread->CheckOverflow();
```

4. Executing the Context Switch: 更新 global current thread reference，並將 new thread 的狀態設為 `RUNNING` ，進一步執行 `SWITCH` 執行 contenxt switch。

```
kernel->currentThread = nextThread;
nextThread->setStatus(RUNNING);
SWITCH(oldThread, nextThread);
```

5. Restoring Thread State Post-Switch: 等到 new thread 完成後檢查是否需回復狀態。

```
ASSERT(kernel->interrupt->getLevel() == IntOff);
CheckToBeDestroyed();
if (oldThread->space != NULL) {
    oldThread->RestoreUserState();
    oldThread->space->RestoreState();
}
```

總覽：

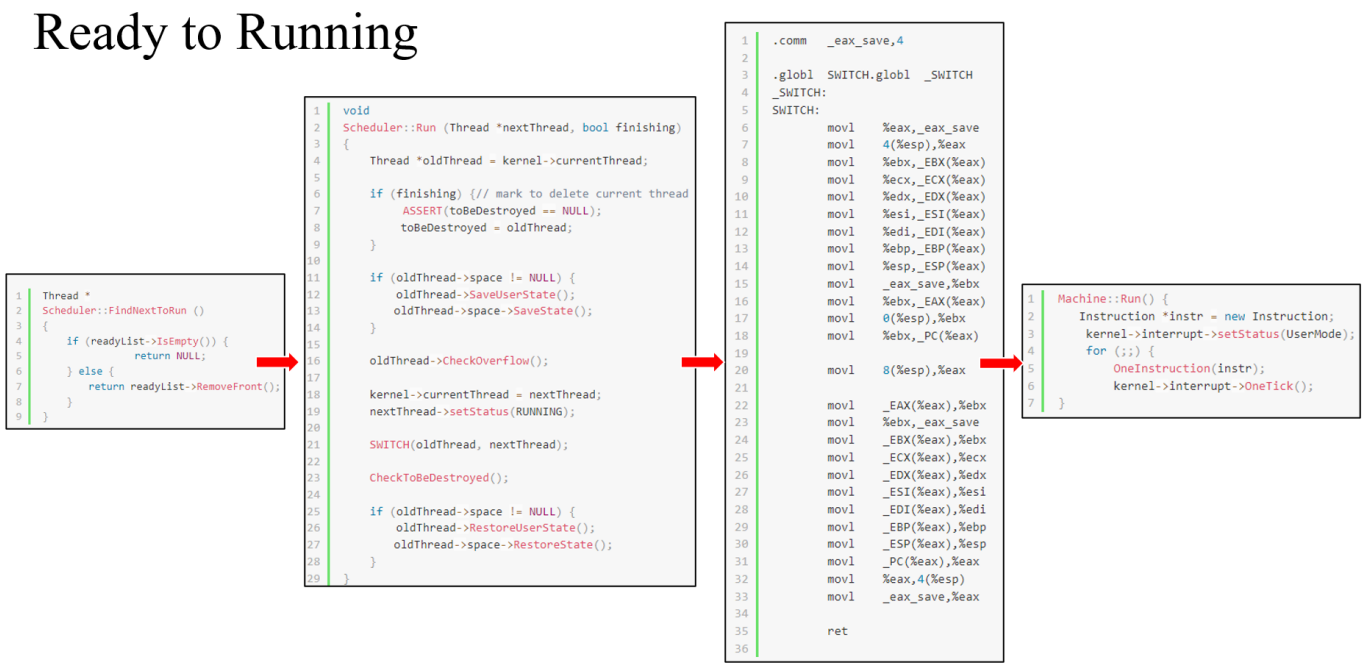
```
void
Scheduler::Run (Thread *nextThread, bool finishing)
{
    Thread *oldThread = kernel->currentThread;

    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

    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();
    }
}
```

1-6. Ready → Running

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

Note: switch.S contains the instructions to perform context switch. You must understand and describe the purpose of these instructions

總覽



1-6-1 Scheduler::FindNextToRun()

目的：

- Return 下一個 CPU 執行的 thread
  - 若沒有 ready threads, return null。
- 將該 thread 從 ready queue 刪除

細節

```
Thread *
Scheduler::FindNextToRun ()
{
    if (readyList->IsEmpty()) {
                return NULL;
    } else {
        return readyList->RemoveFront();
    }
}
```

1-6-2 Scheduler::Run(Thread*, bool)

目的：core operation of context switching。主要負責將 current running thread 切到 ready to run 的 new thread，並且 preserve thread state。

細節：

1. Thread Switching Precondition: capture currently thread for later restoration 並確保 operation 不會被中斷。

```
Thread *oldThread = kernel->currentThread;
ASSERT(kernel->interrupt->getLevel() == IntOff);
```

2. Handling Thread Finish: 如果執行完畢，準備 destrcut current thread，確保 cleanup 防止 memory leakage。

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

3. Saving Thread State: 保存 user program 的狀態，包含 CPU registers 和 memory 狀態。

```
if (oldThread->space != NULL) {
    oldThread->SaveUserState();
    oldThread->space->SaveState();
}
oldThread->CheckOverflow();
```

4. Executing the Context Switch: 更新 global current thread reference，並將 new thread 的狀態設為 `RUNNING` ，進一步執行 `SWITCH` 執行 contenxt switch。

```
kernel->currentThread = nextThread;
nextThread->setStatus(RUNNING);
SWITCH(oldThread, nextThread);
```

5. Restoring Thread State Post-Switch: 等到 new thread 完成後檢查是否需回復狀態。

```
ASSERT(kernel->interrupt->getLevel() == IntOff);
CheckToBeDestroyed();
if (oldThread->space != NULL) {
    oldThread->RestoreUserState();
    oldThread->space->RestoreState();
}
```

總覽：

```
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

    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();
    }
}
```

1-6-3 SWITCH(old Thread*, new Thread*)

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

```
#ifdef x86

        .text
        .align  2

        .globl  ThreadRoot
        .globl  _ThreadRoot

/* void ThreadRoot( void )
**
**      `ThreadRoot` 是新 Process 的入口點
**      主要呼叫 Process 主要執行函數
*/      最後呼叫 `Thread::Finish` 來結束 process
_ThreadRoot:
ThreadRoot:
        # 將舊的指標(base pointer) 儲存到 stack 上
        # 目的是為了在建新的 stack frame 之前保存`上一個`函數 stack frame 的 base ad
        pushl   %ebp
        # 將 stack pointer 的值複製到 base pointer
        # 此行之後 `ebp` 用於指向`新的` stack frame 的底部(固定位置，不移動)
        # 而 `esp` 則向下移動，分配變數或暫時存放數據
        movl    %esp,%ebp
        # 將初始化的參數 push to stack
        # 這個參數會被作為 thread 的啟動函數
        pushl   InitialArg
        # 呼叫 thread 的啟動函數，通常負責進行一些初始化操作
        call    *StartupPC
        # 呼叫 thread 的主函數，這是 thread 主要執行的部分
        call    *InitialPC
        # 呼叫 `Thread::Finish`，用於結束 thread 並進行刪除工作
        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.
**
** eax     points to startup function (interrupt enable)
** ecx     對應到 ThreadBegin
** edx     contains inital argument to thread function
** esi     points to thread function(ForkExecute)
```

```
** edi     point to Thread::Finish()
## esp     ThreadRoot，存新的 Thread PCstate 值
*/

/*
`.comm`: 宣告一個 global variable。
`_eax_save`:該變數用來儲存 `eax` register 的值。
`eax`: 通常儲存 return 值或其他暫存資料。
`4`:指定分配記憶體大小，這裡代表 4 bytes
*/
.comm   _eax_save,4

/*
宣告 `SWITCH` 是 global function，可被其他 code 使用。
*/
.globl  SWITCH
.globl  _SWITCH
_SWITCH:
SWITCH:
        # 將 `eax` reg 值存到全域變數 `_eax_save`
        movl    %eax,_eax_save
        # `4(%esp)` 表示 `esp` 加 4 的位置，即 `t1` 的位置
        # 將 `esp` 指向 `t1`指標的`值`複製到 `eax`(old thread) reg 中
        movl    4(%esp),%eax
        # 以下幾行將當前 process `t1` 的 reg 值保存到對應的記憶體位置上
        # 等於將 register 資料存到 old thread 的 machineState
        # `_EBX(%eax)` 表示 `eax`(old thread) 指向的位址加上 `_EBX` 的 offset
        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)
        # 保留目前 process `t1` stack pointer 的值
        movl    %esp,_ESP(%eax)         # save stack pointer
        # 將之前保存的 `eax` 值從 `_eax_save` 取回
        # 並保存到 `t1` 的對應位置
        movl    _eax_save,%ebx          # get the saved value of eax
        movl    %ebx,_EAX(%eax)         # store it
        # 從 stack 中取出 return address(`esp` 指向的位置)
        # 並且存到 `t1` 的 program counter。
        movl    0(%esp),%ebx            # get return address from stack into ebx
        movl    %ebx,_PC(%eax)          # save it into the pc storage
        # 將 `t2` 的指標從 stack 複製到 `eax`(值指到 new thread)
        movl    8(%esp),%eax            # move pointer to t2 into eax

        # 從 `t2` 的保存位置恢復其 reg 的值
        # 將 new thread 中的資料放到對應 register 中
        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
# 恢復 `t2` 的 stack pointer
movl    _ESP(%eax),%esp         # restore stack pointer
# 將 `t2` 的 return address 複製到 `eax`
# 然後將其複製到 stack 上的返回地址位置。
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
# 返回到 `t2` 的執行位置
ret
```

執行 `SWITCH` 主要將 register 的資料存回 old thread，並把 new thread 的資料拿出來。此時根據 new thread 的狀況會發生兩種情況。

1. 下一個 Thread(`nextThread`) 有 `SWITCH()` 過：

- 前次的 `SWITCH` 會因為 `movl %esp, _ESP(%eax)` ，所以 `stackTop` 已經被 `ret` 覆蓋，於是會執行 `SWITCH()` 下方的程式碼。

2. 下一個 Thread(`nextThread`) 沒有 `SWITCH()` 過：

- 通常指的是第一次執行(`new`)，所以會執行 `stackTop` 的 `ThreadRoot` ，接著建立全新的 thread，並且把 `ret` 放在 `ThreadRoot` 上方，等到 `ThreadRoot` 執行完才會執行到 `nextThread` 。

1-6-4 for loop in Machine::Run()

目的：模擬 CPU 不停地執行 instruction，直到遇到 Halt 條件停下。 內容主要分成三大部分：

1. 執行迴圈：用無限迴圈 for-loop 不停地執行 instruction。
2. 追蹤和 Debug：追蹤 instruction 執行時間(ticks) 和提供各種 debugging 細節。
3. 狀態調整：將 interrupt 狀態轉成 userMode，提醒機器現在是 user-level 的 instruction 而非 kernel 或 system level。

細節：

1. Allocate 一個已經 decoded instruction 的 memory 空間，供機器執行。

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

2. 將 interrupt 狀態轉成 UserMode，提醒機器現在是 user-level 的 instruction 而非 kernel 或 system level。

```
kernel->interrupt->setStatus(UserMode);
```

3. 開始 `for(;;)` 無限執行程式，透過 OneInstruction(instr) 去獲取、解碼，並執行該 instruction， 並且把原本的 ptr 指向同個程式下一個指令。

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

4. OneTick()，用來紀錄系統時間經過一個單位（ticks），並且檢查有沒有要中斷去執行其他程式的請求，有的話會再 call 一次 run 把它做完。沒有的話就會繼續迴圈直到程式執行完。

參照原文:

```
//----------------------------------------------------------------------
// 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
//----------------------------------------------------------------------
```

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

Implement Multi-level feedback queue

Background(Criterion)

- 3 level of queues L1, L2 and L3. . L1 is the highest level queue, and L3 is the lowest level queue
- All processes must have a valid scheduling priority between `0 to 149` . Higher value means higher priority. So 149 is the highest priority, and 0 is the lowest priority.
- A process with priority between `0 - 49` is in the `L3` queue, priority between `50 - 99` is in the `L2` queue, and priority between `100 - 149` is in the `L1` queue.

- For the `L1` queue
  - Preemptive shortest job first
  - Estimate the burst time $t_i = 0.5 * T + 0.5 * t_{i-1}, i > 0, t_0 = 0$
  - Update the approximated burst time when the thread becomes waiting state
  - stop accumulating T when the thread becomes ready state
  - resume accumulating T when the thread moves back to the running state.
- For the `L2` queue
  - non-preemptive priority
  - Thread in L2 first come, first service
  - it will preempt thread in L3 queue
- For the `L3` queue
  - round-robin with the time quantum 100 `ticks`
- Aging mechanism, priority `increased by 10` after waiting more than `1500 ticks`

觀察

1. 根據上述，需要額外紀錄的變數：priority(0-149), approximateBurstTime(t_i), totalBurstTime(T), remainingBurstTime($t_i - T$) 等等。

從其中一種 State Transition 進一步理解需要實作的部分

- New → Ready

1. 原先由 Kernel Parsing '-e'，此處新增 '-ep'，並且在 Thread 新增相關變數紀錄 `Priority` 。

```
// threads/kernel.h

public:
    int Exec(char* name, int priority); //* MP3
private:
// store the priority when creating file
    int filePriority[10];


// threads/kernel.cc
else if (strcmp(argv[i], "-e") == 0) {
    execfile[++execfileNum]= argv[++i];
    filePriority[execfileNum] = 0; // + MP3
    cout << execfile[execfileNum] << "\n";
} else if (strcmp(argv[i], "-ep") == 0) { // + MP3
    ASSERT(i + 2 < argc);
    execfile[++execfileNum]= argv[++i];
    int priority = atoi(argv[++i]);
    ASSERT(priority >= 0 && priority <= 149);
    filePriority[execfileNum] = priority;
    cout << execfile[execfileNum] << "with Priority: " << priority << "\n";
}

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

int Kernel::Exec(char* name, int priority)
{
        t[threadNum] = new Thread(name, threadNum);
        t[threadNum]->space = new AddrSpace();
        t[threadNum]->setPriority(priority); // * MP3
        t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);
        threadNum++;

        return threadNum-1;
}
```

2. 接著到 Exec 會 new Thread，我們此時需要開始記錄各種資料，如設定優先順序、使用時間等等。

```
//threads/thread.h
// begin + MP3
  public:
    void setPriority(int newPriority) { priority = newPriority; } // Set the
priority of the thread
    int getPriority() const { return priority; } // Get the thread's priority
    void setStartExecutionTick(int tick) { executionStartTick = tick; } // Set the
tick count when thread execution started
    void setAgingStartTick(int tick) { agingStartTick = tick; } // Set the tick
count when aging calculation started
    void setTotalAgingTick(int tick) { totalAgingDuration = tick; } // Set the
total duration of aging
    double getRemainingBurstTime() const { return remainingBurst; } // Get
remaining burst time for the thread
    double getLastExecutionTime() const { return lastExecutionDuration; } // Get
last execution duration of the thread
    int getQueueLevel(); // Determine the queue level based on priority
    void updateAgingDuration(); // Update the duration for aging calculation
    void adjustPriority(); // Adjust the thread's priority based on aging

  private:
    int priority; // Priority level of the thread
    double estimatedBurstTime; // Estimated CPU burst time
    double totalExecutedTime; // Total executed time
    double remainingBurst; // Remaining burst time before completion
    double lastExecutionDuration; // Duration of the last execution
    int executionStartTick; // Tick count when the thread last started execution
    int agingStartTick; // Tick count when aging calculation last started
    int totalAgingDuration; // Total duration counted for aging
    void calculateBurstTime(bool transitioningToReady); // Calculate the burst time
  // end + MP3
```

3. 開始到 `Thread.cc` 實現上述功能
   — 觀察完畢 —

結論：需要變更的檔案們有
第一類： Kernel 讀文件(Exec) 的部分

- /threads/kernel.h
- /threads/kernel.cc

第二類：為了 Debug 加上的 flag

- /lib/debug.h

第三類：針對 multi-level queue 實作

- /threads/thread.h -> statistics record
- /threads/thread.cc -> implement .h

- /threads/scheduler.h -> Ready Queue append/remove
- /threads/scheduler.cc -> implement .h
- /threads/alarm.cc -> L3 Round Robin

接下來逐類實作

實作第一類

已於觀察時完成，設定 `-ep` 參數並將該參數提供的 `priority` 值在 Thread 建立時紀錄在該 Class 內

實作第二類

- /lib/debug.h

```
const char dbgSche = 'z';       //+ MP3 scheduler
```

此處要求以下五種錯誤提示，將於實作第三類時實現功能。
(a) Whenever a process is inserted into a queue:
[A] Tick [{current total tick}]: Thread [{thread ID}] is inserted into queue L[{queue level}]

(b) Whenever a process is removed from a queue:
[B] Tick [{current total tick}]: Thread [{thread ID}] is removed from queue L[{queue level}]

© Whenever a process changes its scheduling priority:
[C] Tick [{current total tick}]: Thread [{thread ID}] changes its priority from [{old value}] to [{new value}]

(d)Whenever a process updates its approximate burst time:
[D] Tick [{current total tick}]: Thread [{thread ID}] update approximate burst time, from: [{ti-1}], add [{T}], to [{ti}]

(e)Whenever a context switch occurs:
[E] Tick [{current total tick}]: Thread [{new thread ID}] is now selected for execution, thread [{prev thread ID}] is replaced, and it has executed [{accumulated ticks}] ticks

實作第三類

/threads/thread.h -> statistics record

```
//threads/thread.h
// begin + MP3
  public:
    void setPriority(int newPriority) { priority = newPriority; } // Set the pri
    int getPriority() const { return priority; } // Get the thread's priority
    void setStartExecutionTick(int tick) { executionStartTick = tick; } // Set t
    void setAgingStartTick(int tick) { agingStartTick = tick; } // Set the tick
    void setTotalAgingTick(int tick) { totalAgingDuration = tick; } // Set the t
    double getRemainingBurstTime() const { return remainingBurst; } // Get remai
    double getLastExecutionTime() const { return lastExecutionDuration; } // Get
    int getQueueLevel(); // Determine the queue level based on priority
    void adjustPriority(); // Adjust the thread's priority based on aging
    void updateAgingDuration(); // Update the duration for aging calculation

  private:
    int priority; // Priority level of the thread
    double estimatedBurstTime; // Estimated CPU burst time
    double totalExecutedTime; // Total executed time
    double remainingBurst; // Remaining burst time before completion
    double lastExecutionDuration; // Duration of the last execution
    int executionStartTick; // Tick count when the thread last started execution
    int agingStartTick; // Tick count when aging calculation last started
    int totalAgingDuration; // Total duration counted for aging
    void calculateBurstTime(bool transitioningToReady); // Calculate the burst t
  // end + MP3
```

/threads/thread.cc -> implement .h

目標再細分成兩塊：一是初始化 Thread、二是實作 `.h` 的功能

一、初始化 Thread

- 初始化 Class 的值

```
Thread::Thread(char* threadName, int threadID)
{
    ID = threadID;
    name = threadName;
    stackTop = NULL;
    stack = NULL;
    status = JUST_CREATED;
    for (int i = 0; i < MachineStateSize; i++) {
        machineState[i] = NULL;               /
    }
    //start + MP3
    priority = 0;
    estimatedBurstTime = 0.0;
    totalExecutedTime = 0.0;
    remainingBurst = 0.0;
    lastExecutionDuration = 0.0;
    executionStartTick = 0;
    agingStartTick = 0;
    totalAgingDuration = 0;
    //end + MP3

    space = NULL;
}
```

- 轉回 Ready 前要算一下剩下的 BurstTime

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

    ASSERT(this == kernel->currentThread);

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

    calculateBurstTime(TRUE); // + MP3
    kernel->scheduler->ReadyToRun(this);
    nextThread = kernel->scheduler->FindNextToRun();
    if (nextThread != this) {
        kernel->scheduler->Run(nextThread, FALSE);
    }
    (void) kernel->interrupt->SetLevel(oldLevel);
}
```

- 轉到 Waiting 前也要算一下剩下的 BurstTime

```
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;
    calculateBurstTime(FALSE); // + MP3
        //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
    kernel->scheduler->Run(nextThread, finishing);
}
```

二、實作 header 功能

- 取得任務的 `priority` 並確定其分配的層數

```
int
Thread::getQueueLevel() {
    ASSERT(priority >= 0 && priority <= 149);
    if(priority >= 100) {
        return 1; // High-priority queue (L1)
    } else if (priority >= 50) {
        return 2; // Medium-priority queue (L2)
    } else {
        return 3; // Low-priority queue (L3)
    }
}
```

- Aging：超過 1500 ticks `priority` 提升 10

```
void
Thread::adjustPriority() {
    if(priority < 149 && totalAgingDuration >= 1500) {
        int oldPriority = priority;
        priority = min(149, priority + 10);
        DEBUG(dbgSche, "[C] Tick[" << kernel->stats->totalTicks << "]: Thread ["
        totalAgingDuration -= 1500;
    }
}
```

- 看看已經等了多久，是否需要 Aging

```
void
Thread::updateAgingDuration() {
    totalAgingDuration += (kernel->stats->totalTicks - agingStartTick);
}
```

- 從 Running state 離開計算剩下執行時間

```
void
Thread::calculateBurstTime(bool transitioningToReady) {
    lastExecutionDuration = (double)(kernel->stats->totalTicks -
executionStartTick);

    if(transitioningToReady) {
        totalExecutedTime += lastExecutionDuration;
        remainingBurst = max(0.0, estimatedBurstTime - totalExecutedTime);
    } else {
        double oldEstimatedBurstTime = estimatedBurstTime;
        totalExecutedTime += lastExecutionDuration;
        estimatedBurstTime = 0.5 * totalExecutedTime + 0.5 * estimatedBurstTime;
        DEBUG(dbgSche, "[D] Tick[" << kernel->stats->totalTicks << "]: Thread [" <<
ID << "] updates estimated burst time, from: [" << oldEstimatedBurstTime << "], add
[" << estimatedBurstTime - oldEstimatedBurstTime << "], to [" << estimatedBurstTime
<< "]");
        totalExecutedTime = 0;
        remainingBurst = estimatedBurstTime;
    }
}
```

- /threads/scheduler.h -> Ready Queue append/remove

```
class Scheduler {
  public:
    Scheduler();               // Initialize list of ready threads
    ~Scheduler();              // De-allocate ready list
     void updateThreadAging(); //+ MP3
     bool isL1Empty() { return L1->IsEmpty(); } //+ MP3
  private:
    // List<Thread *> *readyList;  //-MP3
    // begin + MP3
    List<Thread*> *L1; // Queue for high-priority threads
    List<Thread*> *L2; // Queue for medium-priority threads
    List<Thread*> *L3; // Queue for low-priority threads

    // Helper methods for aging and queue management
    void ageThreadsInQueue(List<Thread*> *queue, int queueLevel);
    void addThreadToQueue(List<Thread*> *queue, Thread* thread, int queueLevel);
    Thread* removeThreadFromQueue(List<Thread*> *queue, Thread* thread, int queu
    Thread* selectThreadFromL1Queue();
    Thread* selectThreadFromL2Queue();
    // end + MP3
};
```

/threads/scheduler.cc -> implement .h

一、初始化 Multilevel Feedback Queue

```
Scheduler::Scheduler()
{
    // readyList = new List<Thread *>; - MP3
    // begin + MP3
    L1 = new List<Thread *>;
    L2 = new List<Thread *>;
    L3 = new List<Thread *>;
    // end + MP3
    toBeDestroyed = NULL;
}

Scheduler::~Scheduler()
{
    // delete readyList; // -MP3
    // begin + MP3
    delete L1;
    delete L2;
    delete L3;
    // end + MP3
}
```

二、設定 Aging 機制

```
//----------------------------------------------------------------------
// Scheduler::ageThreadsInQueue
// add the Aging tick in each Thread
// + MP3
//----------------------------------------------------------------------

void
Scheduler::updateThreadAging() {
    ageThreadsInQueue(L1, 1);
    ageThreadsInQueue(L2, 2);
    ageThreadsInQueue(L3, 3);
}

void
Scheduler::ageThreadsInQueue(List<Thread *> *queue, int queueLevel) {
    ListIterator<Thread*> iter(queue);
    while(!iter.IsDone()) {
        Thread *currentThread = iter.Item();
        currentThread->updateAgingDuration();
        currentThread->setAgingStartTick(kernel->stats->totalTicks);
        currentThread->adjustPriority();
        iter.Next();

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

三、設定 Thread Enqueue, Dequeue 機制。

```
//----------------------------------------------------------------------
// Scheduler::addThreadToQueue
// add the Thread tick to the related level queue
// + MP3
//----------------------------------------------------------------------
void
Scheduler::addThreadToQueue(List<Thread*> *queue, Thread *thread, int queueLevel) {
    DEBUG(dbgSche, "[A] Tick[" << kernel->stats->totalTicks << "]: Thread [" <<
thread->getID() << "] is inserted into queue L[" << queueLevel << "]");
    queue->Append(thread);
}

//----------------------------------------------------------------------
// Scheduler::removeThreadFromQueue
// remove the Thread tick from the related level queue
// + MP3
//----------------------------------------------------------------------
Thread*
Scheduler::removeThreadFromQueue(List<Thread*> *queue, Thread *thread, int
queueLevel) {
    DEBUG(dbgSche, "[B] Tick[" << kernel->stats->totalTicks << "]: Thread [" <<
thread->getID() << "] is removed from queue L[" << queueLevel << "]");
    queue->Remove(thread);
    return thread;
}
```

四、設定印出各個 queue 的狀態

```
//----------------------------------------------------------------------
// Scheduler::Print
//      Print the scheduler state -- in other words, the contents of
//      the ready list.  For debugging.
//----------------------------------------------------------------------
void
Scheduler::Print()
{
    cout << "Ready list contents:\n";
    // readyList->Apply(ThreadPrint); // - MP3
    // start + MP3
    cout << "High-Priority Queue (L1) contents:\n";
    L1->Apply(ThreadPrint);
    cout << "Medium-Priority Queue (L2) contents:\n";
    L2->Apply(ThreadPrint);
    cout << "Low-Priority Queue (L3) contents:\n";
    L3->Apply(ThreadPrint);
    // end + MP3
}
```

五、更改 FindNextToRun 從原先只取 ready queue 到從不同 level 的 queue 挑出對應的 Thread

```
//----------------------------------------------------------------------
// 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.
// helper function selectThreadFromL1Queue() and selectThreadFromL2Queue()
//----------------------------------------------------------------------

Thread* Scheduler::selectThreadFromL1Queue() {
    // For L1 (High-Priority Queue), select based on shortest remaining burst ti
    Thread* selectedThread = NULL;
    double minBurstTime = std::numeric_limits<double>::max();

    ListIterator<Thread*> iter(L1);
    for (; !iter.IsDone(); iter.Next()) {
        Thread* currentThread = iter.Item();
        double burstTime = currentThread->getRemainingBurstTime();
        if (burstTime < minBurstTime) {
            selectedThread = currentThread;
            minBurstTime = burstTime;
        }
    }
    return selectedThread;
}

Thread* Scheduler::selectThreadFromL2Queue() {
    // For L2 (Medium-Priority Queue), select based on highest priority
    Thread* selectedThread = NULL;
    int highestPriority = -1;

    ListIterator<Thread*> iter(L2);
    for (; !iter.IsDone(); iter.Next()) {
        Thread* currentThread = iter.Item();
        int priority = currentThread->getPriority();
        if (priority > highestPriority) {
            selectedThread = currentThread;
            highestPriority = priority;
        }
    }
    return selectedThread;
}

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

    if (!L1->IsEmpty()) {
        nextThread = selectThreadFromL1Queue();
        if (nextThread != NULL) {
            return removeThreadFromQueue(L1, nextThread, 1);
        }
    }
```

```
    if (!L2->IsEmpty()) {
        nextThread = selectThreadFromL2Queue();
        if (nextThread != NULL) {
            return removeThreadFromQueue(L2, nextThread, 2);
        }
    }

    if (!L3->IsEmpty()) {
        // For L3, simply select the front thread (round-robin)
        return removeThreadFromQueue(L3, L3->Front(), 3);
    }

    return NULL; // No thread is ready to run
}
```

六、設定成 Ready 的時候放進對應的 Queue

```
//----------------------------------------------------------------------
// 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);
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
        //cout << "Putting thread on ready list: " << thread->getName() << endl ;
    thread->setStatus(READY);
    // readyList->Append(thread); - MP3
    // start + MP3
    int threadPriority = thread->getPriority();
    if(threadPriority >= 100) {
        addThreadToQueue(L1, thread, 1);
    } else if (threadPriority >= 50 && threadPriority < 100) {
        addThreadToQueue(L2, thread, 2);
    } else {
        addThreadToQueue(L3, thread, 3);
    }

    thread->setAgingStartTick(kernel->stats->totalTicks);
    thread->setTotalAgingTick(0);
    // end + MP3
}
```

六、Context Switch 時設定 stat 值

```
//----------------------------------------------------------------------
// 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)
{

    // being + MP3
    nextThread->setStartExecutionTick(kernel->stats->totalTicks);
    DEBUG(dbgSche, "[E] Tick[" << kernel->stats->totalTicks << "]: Thread [" <<



    SWITCH(oldThread, nextThread);

    // we're back, running oldThread
    oldThread->setStartExecutionTick(kernel->stats->totalTicks);
    // end + MP3
}
```

/threads/alarm.cc -> L3 Round Robin

```
//----------------------------------------------------------------------
// Alarm::CallBack
//      Software interrupt handler for the timer device. The timer device is
//      set up to interrupt the CPU periodically (once every TimerTicks).
//      This routine is called each time there is a timer interrupt,
//      with interrupts disabled.
//
//      Note that instead of calling Yield() directly (which would
//      suspend the interrupt handler, not the interrupted thread
//      which is what we wanted to context switch), we set a flag
//      so that once the interrupt handler is done, it will appear as
//      if the interrupted thread called Yield at the point it is
//      was interrupted.
//
//      For now, just provide time-slicing.  Only need to time slice
//       if we're currently running something (in other words, not idle).
//----------------------------------------------------------------------

void
Alarm::CallBack()
{
    Interrupt *interrupt = kernel->interrupt;
    MachineStatus status = interrupt->getStatus();
    // start + MP3
    kernel->scheduler->updateThreadAging(); // Update aging for all threads

    if (status != IdleMode) {
            int currentThreadLevel = kernel->currentThread->getQueueLevel();
        bool isL1Empty = kernel->scheduler->isL1Empty();

        if (currentThreadLevel == 3 || (currentThreadLevel == 2 && !isL1Empty) |
            interrupt->YieldOnReturn();
        }
    }
    // end + MP3
}
```

Trace State Flow

New → Ready

```
Kernel:Kernel(int argc, char **argv) {
    for(int i = 1;i < argc; i++) {
        if (strcmp(argv[i], "-e") == 0) {
            execfile[++execfileNum]= argv[++i];
            filePriority[execfileNum] = 0; // + MP3
            cout << execfile[execfileNum] << "\n";
        } else if (strcmp(argv[i], "-ep") == 0) { // + MP3
            ASSERT(i + 2 < argc);
            execfile[++execfileNum]= argv[++i];
            int priority = atoi(argv[++i]);
               ASSERT(priority >= 0 && priority <= 149);
               filePriority[execfileNum] = priority;
               cout << execfile[execfileNum] << "with Priority:
" << priority << "\n";
        }
    }
}
```

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

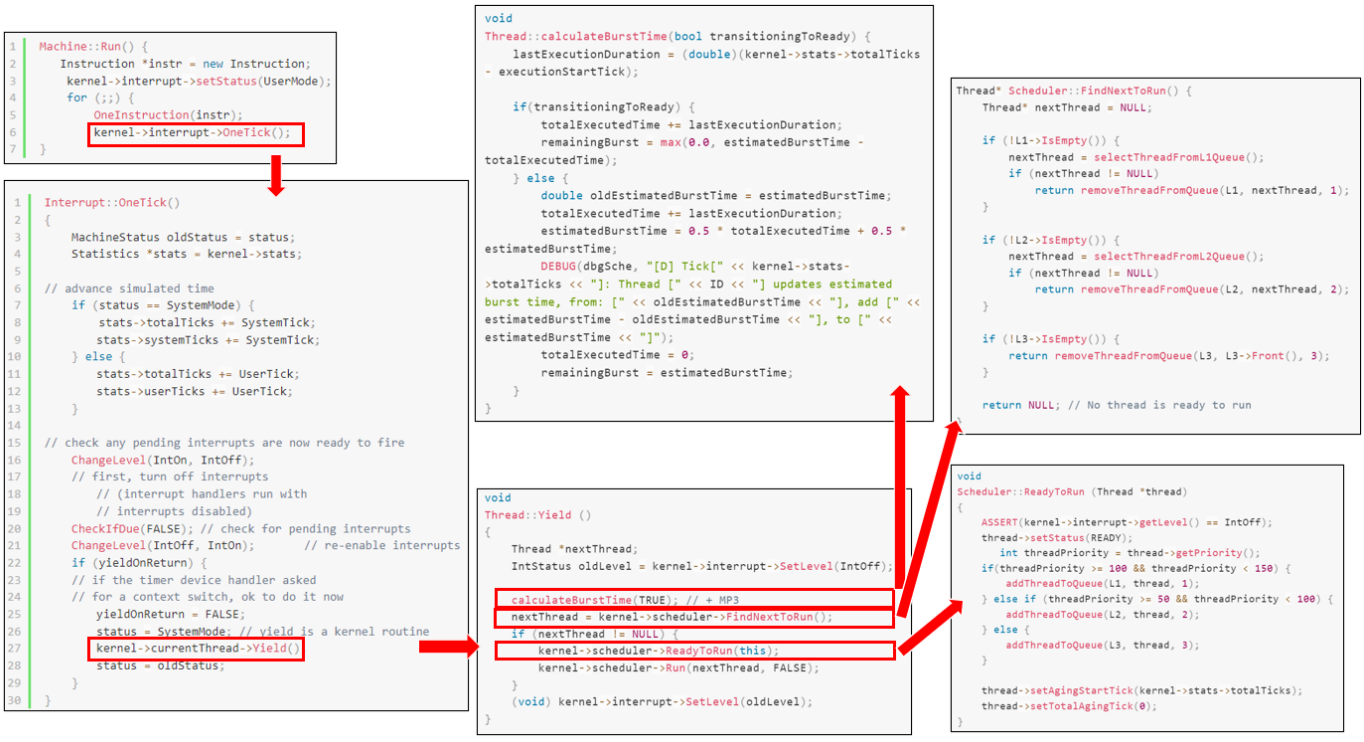
```
int Kernel::Exec(char* name, int priority)
{
        t[threadNum] = new Thread(name, threadNum);
        t[threadNum]->space = new AddrSpace();
        t[threadNum]->setPriority(priority); // * MP3
        t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute,
(void *)t[threadNum]);
        threadNum++;

        return threadNum-1;
}
```

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

    StackAllocate(func, arg);

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

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

```
void
Scheduler::ReadyToRun (Thread *thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    thread->setStatus(READY);
       int threadPriority = thread->getPriority();
    if(threadPriority >= 100 && threadPriority < 150) {
        addThreadToQueue(L1, thread, 1);
    } else if (threadPriority >= 50 && threadPriority < 100) {
        addThreadToQueue(L2, thread, 2);
    } else {
        addThreadToQueue(L3, thread, 3);
    }

    thread->setAgingStartTick(kernel->stats->totalTicks);
    thread->setTotalAgingTick(0);
}
```

```
void
Scheduler::addThreadToQueue(List<Thread*> *queue, Thread
*thread, int queueLevel) {
    DEBUG(dbgSche, "[A] Tick[" << kernel->stats->totalTicks <<
"]: Thread [" << thread->getID() << "] is inserted into queue
L[" << queueLevel << "]");
    queue->Append(thread);
}
```

Running → Ready



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

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

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

// check any pending interrupts are now ready to fire
    ChangeLevel(IntOn, IntOff);
    // first, turn off interrupts
        // (interrupt handlers run with
        // interrupts disabled)
    CheckIfDue(FALSE); // check for pending interrupts
    ChangeLevel(IntOff, IntOn);        // re-enable interrupts
    if (yieldOnReturn) {
    // if the timer device handler asked
    // for a context switch, ok to do it now
        yieldOnReturn = FALSE;
        status = SystemMode; // yield is a kernel routine
        kernel->currentThread->Yield();
        status = oldStatus;
    }
}
```

```
void
Thread::calculateBurstTime(bool transitioningToReady) {
    lastExecutionDuration = (double)(kernel->stats->totalTicks
- executionStartTick);

    if(transitioningToReady) {
        totalExecutedTime += lastExecutionDuration;
        remainingBurst = max(0.0, estimatedBurstTime -
totalExecutedTime);
    } else {
        double oldEstimatedBurstTime = estimatedBurstTime;
        totalExecutedTime += lastExecutionDuration;
        estimatedBurstTime = 0.5 * totalExecutedTime + 0.5 *
estimatedBurstTime;
        DEBUG(dbgSche, "[D] Tick[" << kernel->stats-
>totalTicks << "]: Thread [" << ID << "] updates estimated
burst time, from: [" << oldEstimatedBurstTime << "], add [" <<
estimatedBurstTime - oldEstimatedBurstTime << "], to [" <<
estimatedBurstTime << "]");
        totalExecutedTime = 0;
        remainingBurst = estimatedBurstTime;
    }
}
```

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

    calculateBurstTime(TRUE); // + MP3
    nextThread = kernel->scheduler->FindNextToRun();
    if (nextThread != NULL) {
        kernel->scheduler->ReadyToRun(this);
        kernel->scheduler->Run(nextThread, FALSE);
    }
    (void) kernel->interrupt->SetLevel(oldLevel);
}
```

```
Thread* Scheduler::FindNextToRun() {
    Thread* nextThread = NULL;

    if (!L1->IsEmpty()) {
        nextThread = selectThreadFromL1Queue();
        if (nextThread != NULL)
            return removeThreadFromQueue(L1, nextThread, 1);
    }

    if (!L2->IsEmpty()) {
        nextThread = selectThreadFromL2Queue();
        if (nextThread != NULL)
            return removeThreadFromQueue(L2, nextThread, 2);
    }

    if (!L3->IsEmpty()) {
        return removeThreadFromQueue(L3, L3->Front(), 3);
    }

    return NULL; // No thread is ready to run
}
```

```
void
Scheduler::ReadyToRun (Thread *thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    thread->setStatus(READY);
       int threadPriority = thread->getPriority();
    if(threadPriority >= 100 && threadPriority < 150) {
        addThreadToQueue(L1, thread, 1);
    } else if (threadPriority >= 50 && threadPriority < 100) {
        addThreadToQueue(L2, thread, 2);
    } else {
        addThreadToQueue(L3, thread, 3);
    }

    thread->setAgingStartTick(kernel->stats->totalTicks);
    thread->setTotalAgingTick(0);
}
```

Running → Waiting

```
SynchConsoleOutput::PutChar(char ch)
{
    lock->Acquire();
    consoleOutput->PutChar(ch);
    waitFor->P();
    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) {     // semaphore not available
        queue->Append(currentThread);  // so go to sleep
        currentThread->Sleep(FALSE);
    }
    value--;    // semaphore available, consume its value

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

```
void
Thread::Sleep (bool finishing)
{
    Thread *nextThread;
    status = BLOCKED;
    calculateBurstTime(FALSE);
    while ((nextThread = kernel->scheduler->FindNextToRun())
== NULL) {
        kernel->interrupt->Idle();
    }
    kernel->scheduler->Run(nextThread, finishing);
}
```

```
void
Thread::calculateBurstTime(bool transitioningToReady) {
    lastExecutionDuration = (double)(kernel->stats->totalTicks
- executionStartTick);

    if(transitioningToReady) {
        totalExecutedTime += lastExecutionDuration;
        remainingBurst = max(0.0, estimatedBurstTime -
totalExecutedTime);
    } else {
        double oldEstimatedBurstTime = estimatedBurstTime;
        totalExecutedTime += lastExecutionDuration;
        estimatedBurstTime = 0.5 * totalExecutedTime + 0.5 *
estimatedBurstTime;
        DEBUG(dbgSche, "[D] Tick[" << kernel->stats-
>totalTicks << "]: Thread [" << ID << "] updates estimated
burst time, from: [" << oldEstimatedBurstTime << "], add [" <<
estimatedBurstTime - oldEstimatedBurstTime << "], to [" <<
estimatedBurstTime << "]");
        totalExecutedTime = 0;
        remainingBurst = estimatedBurstTime;
    }
}
```

```
Thread* Scheduler::FindNextToRun() {
    Thread* nextThread = NULL;

    if (!L1->IsEmpty()) {
        nextThread = selectThreadFromL1Queue();
        if (nextThread != NULL)
            return removeThreadFromQueue(L1, nextThread, 1);
    }

    if (!L2->IsEmpty()) {
        nextThread = selectThreadFromL2Queue();
        if (nextThread != NULL)
            return removeThreadFromQueue(L2, nextThread, 2);
    }

    if (!L3->IsEmpty()) {
        return removeThreadFromQueue(L3, L3->Front(), 3);
    }

    return NULL; // No thread is ready to run
}
```

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

    if (IsEmpty()) {            // list is empty
        first = element;
        last = element;
    } else {                    // else put it after last
        last->next = element;
        last = element;
    }
    numInList++;
}
```

```
Thread*
Scheduler::selectThreadFromL1Queue() {
    // For L1 (High-Priority Queue), select based on shortest
remaining burst time
    Thread* selectedThread = NULL;
    double minBurstTime = std::numeric_limits<double>::max();

    ListIterator<Thread*> iter(L1);
    for (; !iter.IsDone(); iter.Next()) {
        Thread* currentThread = iter.Item();
        double burstTime = currentThread-
>getRemainingBurstTime();
        if (burstTime < minBurstTime) {
            selectedThread = currentThread;
            minBurstTime = burstTime;
        }
    }
    return selectedThread;
}
```

```
Thread*
Scheduler::selectThreadFromL2Queue() {
    // For L2 (Medium-Priority Queue), select based on highest
priority
    Thread* selectedThread = NULL;
    int highestPriority = -1;

    ListIterator<Thread*> iter(L2);
    for (; !iter.IsDone(); iter.Next()) {
        Thread* currentThread = iter.Item();
        int priority = currentThread->getPriority();
        if (priority > highestPriority) {
            selectedThread = currentThread;
            highestPriority = priority;
        }
    }
    return selectedThread;
}
```

```
Thread*
Scheduler::removeThreadFromQueue(List<Thread*> *queue, Thread
*thread, int queueLevel) {
    DEBUG(dbgSche, "[B] Tick[" << kernel->stats->totalTicks <<
"]: Thread [" << thread->getID() << "] is removed from queue
L[" << queueLevel << "]");
    queue->Remove(thread);
    return thread;
}
```

Waiting → Ready

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

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

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

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

```
void
Scheduler::ReadyToRun (Thread *thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    thread->setStatus(READY);
       int threadPriority = thread->getPriority();
    if(threadPriority >= 100 && threadPriority < 150) {
        addThreadToQueue(L1, thread, 1);
    } else if (threadPriority >= 50 && threadPriority < 100) {
        addThreadToQueue(L2, thread, 2);
    } else {
        addThreadToQueue(L3, thread, 3);
    }

    thread->setAgingStartTick(kernel->stats->totalTicks);
    thread->setTotalAgingTick(0);
}
```

```
void
Scheduler::addThreadToQueue(List<Thread*> *queue, Thread
*thread, int queueLevel) {
    DEBUG(dbgSche, "[A] Tick[" << kernel->stats->totalTicks <<
"]: Thread [" << thread->getID() << "] is inserted into queue
L[" << queueLevel << "]");
    queue->Append(thread);
}
```

Running → Terminated

```
case SC_Exit:

    val=kernel->machine->ReadRegister(4);
    cout << "return value:" << val << endl;
    kernel->currentThread->Finish();
    break;
    default:
cerr << "Unexpected system call " << type << "\n";
break;
```

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

    DEBUG(dbgThread, "Finishing thread: " << name);
    Sleep(TRUE);// invokes SWITCH
    // not reached
}
```

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

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

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

```
Thread* Scheduler::FindNextToRun() {
    Thread* nextThread = NULL;

    if (!L1->IsEmpty()) {
        nextThread = selectThreadFromL1Queue();
        if (nextThread != NULL)
            return removeThreadFromQueue(L1, nextThread, 1);
    }

    if (!L2->IsEmpty()) {
        nextThread = selectThreadFromL2Queue();
        if (nextThread != NULL)
            return removeThreadFromQueue(L2, nextThread, 2);
    }

    if (!L3->IsEmpty()) {
        return removeThreadFromQueue(L3, L3->Front(), 3);
    }

    return NULL; // No thread is ready to run
}
```

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

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

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

    oldThread->CheckOverflow();

    kernel->currentThread = nextThread;
    nextThread->setStatus(RUNNING);

    SWITCH(oldThread, nextThread);

    CheckToBeDestroyed();

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

Ready → Running

Result

- $ .../build.linux/nachos -ep hw3t1 0 -ep hw3t2 0 #L3

```
[os23team47@localhost test]$ ../build.linux/nachos -ep hw3t1 0 -ep hw3t2 0
hw3t1 with Priority: 0
hw3t2 with Priority: 0
1
2
1
2
1
2
1
2
1
2
1
2
1
2
1
2
1
2
return value:1
return value:2
```

- $ .../build.linux/nachos -ep hw3t1 50 -ep hw3t2 50 #L2

```
[os23team47@localhost test]$ ../build.linux/nachos -ep hw3t1 50 -ep hw3t2 50
hw3t1 with Priority: 50
hw3t2 with Priority: 50
1
1
1
1
1
1
1
1
1
return value:1
2
2
2
2
2
2
2
2
2
return value:2
```

- $ .../build.linux/nachos -ep hw3t1 50 -ep hw3t2 90 #L2

```
[os23team47@localhost test]$ ../build.linux/nachos -ep hw3t1 50 -ep hw3t2 90
hw3t1 with Priority: 50
hw3t2 with Priority: 90
2
2
2
2
2
2
2
2
2
return value:2
1
1
1
1
1
1
1
1
1
return value:1
```

- $ .../build.linux/nachos -ep hw3t1 100 -ep hw3t2 100 #L1

```
[os23team47@localhost test]$ ../build.linux/nachos -ep hw3t1 100 -ep hw3t2 100
hw3t1 with Priority: 100
hw3t2 with Priority: 100
1
1
2
2
2
1
1
1
2
2
2
1
1
1
2
2
2
1
return value:1
return value:2
```

- $ .../build.linux/nachos -ep hw3t1 40 -ep hw3t2 55 #L3 → L2

```
[os23team47@localhost test]$ ../build.linux/nachos -ep hw3t1 40 -ep hw3t2 55
hw3t1 with Priority: 40
hw3t2 with Priority: 55
2
2
2
2
2
2
2
2
2
return value:2
1
1
1
1
1
1
1
1
1
return value:1
```

- $ .../build.linux/nachos -ep hw3t1 40 -ep hw3t2 90 #L3 → L2

```
[os23team47@localhost test]$ ../build.linux/nachos -ep hw3t1 40 -ep hw3t2 90
hw3t1 with Priority: 40
hw3t2 with Priority: 90
2
2
2
2
2
2
2
2
2
return value:2
1
1
1
1
1
1
1
1
1
return value:1
```

- $ .../build.linux/nachos -ep hw3t1 90 -ep hw3t2 100 #L2 → L1

```
[os23team47@localhost test]$ ../build.linux/nachos -ep hw3t1 90 -ep hw3t2 100
hw3t1 with Priority: 90
hw3t2 with Priority: 100
2
2
2
2
2
2
2
2
2
return value:2
1
1
1
1
1
1
1
1
1
return value:1
```

- $ .../build.linux/nachos -ep hw3t1 60 -ep hw3t3 50 #L2

```
[os23team47@localhost test]$ ../build.linux/nachos -ep hw3t1 60 -ep hw3t3 50
hw3t1 with Priority: 60
hw3t3 with Priority: 50
1return value:3

1
1
1
1
1
1
1
1
return value:1
```