

NACHOS-4.0

MP3



第21組

106062173 吳東弦(一起 trace code,負責整理 report)

106062271 郭子豪(一起 TRACE CODE,負責實作 CODING)

### trace code

# New→Ready

## Kernel::ExecAll()

　　在ExecAll()中，先跑一個回圈（Exec()）將要執行之程式的執行緒建立好，接著準備將main thread結束掉（Finish()）。

## Kernel::Exec(char\*)

　　在Exec()中，模擬利用main thread複製（Fork()）出將要執行之程式的thread，並分配各自所需的分頁表（AddrSpace()）。

## Thread::Fork(VoidFunctionPtr, void\*)

　　在Fork()中，先把thread的PCB資訊初始化（StackAllocate()），再將其加入ready queue中（ReadyToRun()）。

## Thread::StackAllocate(VoidFunctionPtr, void\*)

　　在StackAllocate()中，先分配記憶體空間（AllocBoundedArray()），在將PCB的暫存器狀態（machineState[]）初始化為對應stack位置。PCstate對應到ThreadRoot（stack top）；StartupPCState對應到ThreadBegin⋯⋯。

## Scheduler::ReadyToRun(Thread\*)

　　在ReadyToRun()中，把thread的狀態設定為ready（setStatus()），並將其加入ready queue中（readyList）。

# Running→Ready

## Machine::Run()

　　在Run()中，會不斷去讀取thread的指令（OneInstruction()），每讀取一次指令，就會遞進一次tick（OneTick()）。

## Interrupt::OneTick()

　　在OneTick()中，會先依照kernel mode或user mode遞進totalTicks，再去檢查是否有I/O做完，如果有的話要去對硬體做中斷排程（CheckIfDue()），最後去檢查Round-Robin的time quantum是否用完（yieldOnReturn），如果用完的話要將context switch加入排程（Yield()）。

## Thread::Yield()

　　在Yield()中，會去檢查是否有其他的thread要執行（FindNextToRun()），如果有的話就會將它加入排程（Run）。

## Scheduler::FindNextToRun()

　　在FindNextToRun()中，會去檢查ready queue是否有其他thread要去執行（IsEmpty()），如果有的話就將它取出來（RemoveFront()）。

## Scheduler::ReadyToRun(Thread\*)

　　在ReadyToRun()中，把要switch出去的thread狀態設定為ready（setStatus()），並將其放回ready queue中（readyList）。

## Scheduler::Run(Thread\*, bool)

　　在Run()中，先將thread狀態存回記憶體中，並檢查是否發生堆疊溢位（CheckOverflow()），再switch出去；當switch回來後，則將thread狀態讀回暫存器中。而在實體機器進行context switch後（SWITCH），虛擬機中的下一個thread也會開始執行。

# Running→Waiting

## SynchConsoleOutput::PutChar(char)

　　在PutChar()中，先取得設備的mutex（lock）後，執行I/O（PutChar()），等到I/O結束發出中斷指令（waitFor）再釋放mutex。

## Semaphore::P()

　　在P()中，就是semaphore的wait機制，當value大於零表示有資源可以使用；當value等於零表示目前沒有資源，會把thread加入此semaphore的等待隊列（Append()），並且讓他進waiting queue睡眠（Sleep()）。

## SynchList<T>::Append(T)

　　在Append()中，實作加入semaphore等待隊列（List）的過程，如果queue是空的，加入的thread就是第一個元素；如果queue不是空的，就會排到隊列的最後面。

## Thread::Sleep(bool)

　　在sleep()中，先把當前thread設為blocked，並加入waiting queue，再去檢查是否有其他的thread要執行（FindNextToRun()），如果有的話就會將它加入排程（Run）。

## Scheduler::FindNextToRun()

　　在FindNextToRun()中，會去檢查ready queue是否有其他thread要去執行（IsEmpty()），如果有的話就將它取出來（RemoveFront()），準備進行context switch。

## Scheduler::Run(Thread\*, bool)

　　在Run()中，先將thread狀態存回記憶體中，並檢查是否發生堆疊溢位（CheckOverflow()），再switch出去；當switch回來後，則將thread狀態讀回暫存器中。而在實體機器進行context switch後（SWITCH），虛擬機中的下一個thread也會開始執行。

# Waiting→Ready

## Semaphore::V()

　　在V()中，當thread做完I/O，發出中斷訊號後，OS就會將他放回ready queue（ReadyToRun()），並釋出自己所佔用的資源（value）。

## Scheduler::ReadyToRun(Thread\*)

　　在ReadyToRun()中，把thread的狀態設定為ready（setStatus()），並將其放回ready queue中（readyList）。

# Running→Terminated

## ExceptionHandler(ExceptionType) case SC\_Exit

　　在SC\_Exit中，當CPU暫存器指向（ReadRegister()）結束程式的system call時，就會去將thread給結束掉（Finish()）。

## Thread::Finish()

　　在 Finish()中將結束thread的訊號(TRUE)傳給了 Sleep()。

## Thread::Sleep(bool)

　　在sleep()中，先把當前thread設為blocked，再去檢查是否有其他的thread要執行（FindNextToRun()），如果有的話就會將它加入排程（Run），並將結束訊號傳下去（finishing）。

## Scheduler::FindNextToRun()

　　在FindNextToRun()中，會去檢查ready queue是否有其他thread要去執行（IsEmpty()），如果有的話就將它取出來（RemoveFront()），準備進行context switch。

## Scheduler::Run(Thread\*, bool)

　　在Run()中，先標記當前要結束的thread（toBeDestroyed），並將thread狀態存回記憶體中，檢查是否發生堆疊溢位（CheckOverflow()），再switch出去；當switch回來將thread狀態讀回暫存器後，就將剛剛的thread給結束掉（CheckToBeDestroyed()）。而在實體機器進行context switch後（SWITCH），虛擬機中的下一個thread也會開始執行。

# Ready→Running

## Scheduler::FindNextToRun()

　　在FindNextToRun()中，會去檢查ready queue是否有其他thread要去執行（IsEmpty()），如果有的話就將它取出來（RemoveFront()），準備進行context switch。

## Scheduler::Run(Thread\*, bool)

　　在Run()中，要執行的thread在實體機器context switch後（SWITCH），虛擬機中的下一個thread也會開始執行。

## SWITCH(Thread\*, Thread\*)

* 先將要switch出去的thread暫存器資訊存回PCB

movl %eax,\_eax\_save // 先把eax的data存起來

movl 4(%esp),%eax // 借eax，讓eax取得4(%esp)的值

movl %ebx,\_EBX(%eax) // 依照stack top對應偏移值將data存回PCB

(ecx、edx、esi、edi、ebp（base pointer）、esp（stack pointer）)

movl \_eax\_save,%ebx // 借ebx，讓ebx讀取eax原本的data

movl %ebx,\_EAX(%eax) // eax原本的data也要記得存回PCB

movl 0(%esp),%ebx // 借ebx，讓ebx指向stack top

movl %ebx,\_PC(%eax) // 讓PCB的\_PC取得stack top

* 再將要switch進來的thread之PCB資訊讀進暫存器

movl 8(%esp),%eax // 借eax，讓eax指向8(%esp)

movl \_EAX(%eax),%ebx // 借ebx，讓ebx取得PCB的EAX資訊

movl %ebx,\_eax\_save // 先把PCB的EAX資訊存起來

movl \_EBX(%eax),%ebx // 依照PCB對應偏移資訊讀進暫存器中

(\_ECX、\_EDX、\_ESI、\_EDI、\_EBP（base）、\_ESP（stack）)

movl \_PC(%eax),%eax // 借eax，讓eax取得\_PC資訊

movl %eax,4(%esp) // 讓4(%esp)取得eax的值

movl \_eax\_save,%eax // PCB的EAX資訊也要記得讀進eax

ret // return同時，PC正指向新thread的ThreadRoot()

* ThreadRoot()呼叫對應的函數

call \*StartupPC // 對應ThreadBegin()

call \*InitialPC // 對應ForkExcute()

call \*WhenDonePC // 對應ThreadFinish()

* ForkExecute(Thread \*t)

t->space->Execute(t->getName()); // 準備執行新thread

* AddrSpace::Execute(char\* fileName)

kernel->machine->Run(); // 開始執行新thread

## Machine::Run()

　　thread回到running狀態，形成一個context switch循環。

### Implementation

# 流程圖

一張含有 文字, 地圖 的圖片

自動產生的描述

# 黑色部分

　　黑色部分流程為nachos原本的code，從Machine::Run()之後分別有兩個部分會影響thread在queue間的變動。

　　左邊表示Round-Robin機制觸發timer發出中斷，FindNextToRun會從ready queue挑一個thread準備切換至running state，ReadyToRun會把原本在running state的thread放回ready queue。

　　右邊表示有thread想做I/O，當thread要做I/O時，FindNextToRun也會從ready queue挑一個thread準備切換至running state（不在圖中），做完I/O後觸發中斷，ReadyToRun會把原本在waiting queue的thread放回ready queue。

# 藍色部分（實作thread要切換state / queue）

## FindNextToRun

一張含有 螢幕擷取畫面 的圖片

自動產生的描述

* old thread準備要離開running state，可能是因為time quantum用完，或是準備要做I/O。
* 接著從ready queue中挑出new thread，L[1]優先於L[2]優先於L[3]。
* 如果new thread存在，且old thread不是要進行I/O，new thread就與old thread比較優先序位決定是否要preemptive。
* 如果要preemptive，紀錄new thread開始running的時間

## ReadyToRun

一張含有 文字 的圖片

自動產生的描述

* old thread準備要回ready queue，可能是因為time quantum用完（來自running state），或是I/O做完（來自waiting queue）。
* 先將old thread狀態設為ready，並記錄回到ready queue的時間。
* 再將old thread放回對應的ready queue（L[1]、L[2]、L[3]）。

# 紅色部分（Update T / t / Aging）一張含有 文字 的圖片 自動產生的描述

* old thread準備要離開running state，紀錄目前累計cpu burst time（T），同時紀錄aging。
* 如果要做I/O，先用剛剛紀錄的累計cpu burst time（T）計算新的approximate burst time（t），再將累計cpu burst time（T）歸零。

### Summary & FeedBack

　　本次trace code的部分與MP1、MP2多有重疊，並沒有很難。在實作的部分，遇到主要的瓶頸是排程的詳細規則，看完Ch5後即使知道大致的理論，然而在實務上，可以有不同的理解，從iLMS討論區的熱絡程度可見一斑。其一是對於T、t、aging理解錯誤，導致code多次修改，其二是對於preemptive時機不確定，導致不知道要把code寫在哪裡，後來知道實作允許I/O做完的interrupt在alarm後才觸發，那麼基本上都把code寫在schedular.cc就可以了。