11220CS342300 Operating System Final Project

Group 41 IPS25 110020007 施淙綸

Team member contribution

施淙綸 - ALL

Part 1. Trace code

Note: 所有函式原則上僅在一處說明，如有重複的函式，將以上方為主，並於下方重複處標註。

1-1. New→Ready

初始化所有 threads，並將它們的 `ForkExecute` 放進 execute stack 準備將 user program 的內容讀取進來，最後將這些 threads 放入 ready queue。

- userprog/userkernel.cc `UserProgKernel::InitializeAllThreads()`
  - 對所有 execfile 調用 `UserProgKernel::InitializeOneThread(char*, int, int)`，藉此建立各自的實例 `Thread{}` 以完成初始化。
- userprog/userkernel.cc `UserProgKernel::InitializeOneThread(char*, int, int)`
  - 根據得到的參數建立對應的 `Thread{}` 實例，包含該 thread 的基本資訊、流水號和屬性初始化都在這裡處理。
- threads/thread.cc `Thread::Fork(VoidFunctionPtr, void*)`
  - 將該 thread 的 `interrupt` 與 `scheduler` 和 kernel 的綁定。
  - 這裡的引入函式和參數是 `ForkExecute` 和 `threadNum`，會被丟到 `Thread::StackAllocate(VoidFunctionPtr, void*)`。
  - 在使用 `Scheduler::ReadyToRun(Thread*)` 放入 ready queue 之前，將 `interrupt` 暫時關閉，完成後復原。
- threads/thread.cc `Thread::StackAllocate(VoidFunctionPtr, void*)`
  - 來自 `Thread::Fork(VoidFunctionPtr, void*)` 的 `ForkExecute` 和 `threadNum` 會被丟到 execution stack 等待執行。而 `ForkExecute` 做的事情就是把 user program 的 instructions 撈進來 thread 的 `AddrSpace` 並開始執行。
- threads/scheduler.cc **`Scheduler::ReadyToRun(Thread*)`**
  - 根據參數中 `thread` 的優先級，把它丟到 ready queue 裡面排隊。Preemptive 說明見 1-2 **`Scheduler::ReadyToRun(Thread*)`**。

1-2 Running→Ready

在 `UserMode` 執行 user program。若過程中遇到 yield 請求時（i.e. context switch），更新當前 thread 的 `RemainingBurstTime` 後，找到下一個要執行的 thread, context switch then run。

- machine/mipssim.cc `Machine::Run()`
  - 模擬 user-level 的程式呼叫，所以同時也會將 `interrupt` 改為 `UserMode`，以利時間計算。
  - 每次會模擬運行一行 instruction 並前進一個 tick。而這個函式和 threads 一一對應，實際上就是代表該 thread 一直運行下去。
- machine/interrupt.cc `Interrupt::OneTick()`
  - 根據當前呼叫的 `status`（`UserMode` or `SystemMode`）將對應的 tick 增加。
  - 之後將 interrupt 關閉後，確認在排隊的 interrupt 有沒有到時間要做的？如果有，那就讓他跑，反之無事發生。
  - 如果 timer 請求進行 context switch 的話（`yieldOnReturn == True`），切換為 `SystemMode` 呼叫 1-2 **`Thread::Yield()`** 進行 context switch。
- threads/thread.cc **`Thread::Yield()`**
  - 將現正運行之 thread 調用 1-2 **`Scheduler::ReadyToRun(Thread*)`** 放入 ready queue 中，再調用 1-2 **`Scheduler::FindNextToRun()`** 取出下一個要運行的 thread（`nextThread`）。
  - 如果 `nextThread` 不為空或與原先不同，則代表 context switch 發生了。
  - 更新好 `RemainingBurstTime` 後 context switch，並開始執行 `nextThread`。
- threads/scheduler.cc **`Scheduler::FindNextToRun()`**
  - 從 L1 ready queue 找起，若 L1 為空，則依序查找 L2, L3，直至找到 `NextThread` 為止。
  - 找到 `NextThread` 後，將其移出 ready queue，並回傳 `NextThread`。反之，僅回傳 `NULL`。
- threads/scheduler.cc **`Scheduler::ReadyToRun(Thread*)`**
  - 同 1-1 **`Scheduler::ReadyToRun(Thread*)`** 所述。
  - 當該 thread 進入的是 L1 ready queue 時，需檢查是否發生 preemptive，有兩種情況：其一是新加入的 thread 的 `RemainingBurstTime` 較當前小，其二是新加入的 thread 的優先度較高（此處實現允許新加入的 L1 thread 搶奪現正執行之 L2 或 L3 thread，若僅考慮 L1 為 preemptive SRTN 的情況下不需實作此部分）。
- threads/scheduler.cc `Scheduler::Run(Thread*, bool)`
  - 將當前執行的 thread 和 `nextThread` 做 context switch，此處假設跑新的 thread 時，舊的 thread 已經被撈到 ready queue 或是 blocked。
  - 同時，若參數指定當前執行的 thread 已經完成，將在 context switch 完成後，調用 `CheckToBeDestroyed()` 函式將其刪除。

1-3. Running→Waiting

如果 user program 要進行 I/O（此處皆為 output），發現沒有可用的 `"console out"` semaphore，將自己這個 thread 放入 semaphore 的 waiting list 中，暫時 1-3 **`Thread::Sleep(bool)`**，直到有可用資源時再次被喚醒（見 1-4 Waiting→Ready）。

- userprog/exception.cc `ExceptionHandler(ExceptionType) case SC_PrintInt`

    - 如果碰到需要輸出 int 的 system call 時，呼叫 `SC_PrintInt` 將要輸出的數字從 register 讀到 `val` 中，再藉由 1-3 `SynchConsoleOutput::PutInt()` 將其放至 console。
- userprog/synchconsole.cc `SynchConsoleOutput::PutInt()`
    - 將要輸出的整數轉成 string。
    - 要求 `"console out"` 的 lock 後，調用 1-3 `ConsoleOutput::PutChar(char)` 將轉成字串的整數的一個 char 放到 console，並等待 `"console out"` 的 semaphore（即 1-3 `Semaphore::P()`）可用，再繼續放至下一個 char，直到將該整數全部放到 console 為止。
- machine/console.cc `ConsoleOutput::PutChar(char)`
    - 模擬將一個要顯示的 char 放置到檔案（包含 stdout）的過程，並排定未來在 console 上要顯示的 interrupt。
- threads/synch.cc `Semaphore::P()`
    - 暫時將 interrupt 關閉，如果現在沒有此 semaphore 可用，將這個 thread 的請求丟到此 semaphore 的 waiting queue 裡面等，並暫時讓目前的 thread sleep（被丟到 waiting）。
- threads/synchlist.cc `SynchList<T>::Append(T)`
    - 將一個型態為 `T` 的物件在確保 mutex 的情況下，放進此 list 的後方並喚醒任一 waiter 處理（若有）。（註：似乎在此份作業沒有明確用途？）
- threads/thread.cc **`Thread::Sleep(bool)`**
    - 在當前 thread 必須等待 lock, semaphore 或完成時調用此函式。
    - 先進入 `BLOCKED` 狀態，直到有用 1-2 **`Scheduler::FindNextToRun()`** 找到下一個排隊中可以執行的 thread。
    - 如果當前 thread 已經有跑過一段時間的話，結算並更新 `RemainingBurstTime`，再 context switch 至 `nextThread`。
    - 此處有 `bool finishing` 參數會被 pass 到 1-2 **`Scheduler::Run(Thread*, bool)`** 函式，決定是否在 context switch 後刪除此 thread。
- threads/scheduler.cc **`Scheduler::FindNextToRun()`**
    - 同 1-2 **`Scheduler::FindNextToRun()`** 所述。
- threads/scheduler.cc `Scheduler::Run(Thread*, bool)`
    - 同 1-2 **`Scheduler::Run(Thread*, bool)`** 所述。

1-4. Waiting→Ready

當 `"console out"` semaphore 可用資源增加時，喚醒一個原先在排隊的 thread，將其放回 ready queue 等待執行。

- threads/synch.cc `Semaphore::V()`
    - 先停止 interrupt 後，如果此 semaphore（此指 `"console out"` 的 semaphore）有東西在等待的話，喚醒它（在這裡指的是把原先在 waiting state 的 thread 丟回去 ready state，也就是放進 ready queue），再增加此 semaphore 代表的資源量。
- threads/scheduler.cc **`Scheduler::ReadyToRun(Thread*)`**
    - 同 1-2 **`Scheduler::ReadyToRun(Thread*)`** 所述。

1-5. Running→Terminated

當 user program 呼叫該程式要離開的 system call 時，找到下一個要運行的 thread 後 context switch，並刪除要離開的 program 的 thread。

- userprog/exception.cc `ExceptionHandler(ExceptionType) case SC_Exit`
  - user program 呼叫該程式要離開的 system call。
  - 輸出程式回傳值後（像 C++ 的 `int main()` 回傳值），結束該程式（thread），i.e. 呼叫 1-5 `Thread::Finish()`。
- threads/thread.cc `Thread::Finish()`
  - 調用 1-3 **`Thread::Sleep(bool)`**，並標記該 thread 可以刪除，在 1-3 **`Thread::Sleep(bool)`** 中 context switch 結束後將會刪除此 thread。
- threads/thread.cc **`Thread::Sleep(bool)`**
  - 同 1-3 **`Thread::Sleep(bool)`** 所述。
- threads/scheduler.cc **`Scheduler::FindNextToRun()`**
  - 同 1-2 **`Scheduler::FindNextToRun()`** 所述。
- threads/scheduler.cc `Scheduler::Run(Thread*, bool)`
  - 同 1-2 **`Scheduler::Run(Thread*, bool)`** 所述。

1-6. Ready→Running

決定下一個要執行的 thread，並 context switch 後，逐行執行 user program isntructions。

- threads/scheduler.cc **`Scheduler::FindNextToRun()`**
  - 同 1-2 **`Scheduler::FindNextToRun()`** 所述。
- threads/scheduler.cc `Scheduler::Run(Thread*, bool)`
  - 同 1-2 **`Scheduler::Run(Thread*, bool)`** 所述。
- threads/switch.s `SWITCH(Thread*, Thread*)`
  - 將舊的 thread (pointer a0) 原先放在 register 的 callee saves (s0~s7), stack pointer (sp), frame pointer (fp), and program counter (PC) 放回 a0 相應的 memory 空間，再將新的 thread (pointer a1) 放在 memory 裡面的 s0~s7, sp, fp, ra(原先的 PC) 放到 register 中，最後跳回 PC 接續執行原先的 program instructions。
  - 這些 s0~s7, sp, fp, and ra(PC) 就是所謂的 context，所以兩 thread 的 context switch 就是在交換 register 裡面的 context 和 memory 裡面的。
  - 註：Spec 裡面說明提到「Try to understand the x86 instructions first. Concept in MIPS version is equivalent to x86 in switch.s, ...」，但我覺得 MIPS 版本較 x86 好理解，所以我是看 MIPS 而非 x86。
- Machine/mipssim.cc `for loop in Machine::Run()`
  - 同 1-2 `Machine::Run()` 所述。

其他上述未提及的函式

這裡提供上面 trace code 未要求說明，但在檔案中卻標記 `<REPORT>` 的程式碼片段的說明。

- lib/debug.h `const char dbgMLFQ = 'z';`
  - 將 `z` tag 命名為 `dbgMLFQ`，輸出時可以用 `DEBUG('z', ...);` 或 `DEBUG(dbgMLFQ, ...);`。
- test/Makefile
  - 把 test 用的 use program 編譯起來。（似乎並非此 report 的範圍？）
- threads/main.cc `kernel->InitializeAllThreads();`
  - 同 1-1 `UserProgKernel::InitializeAllThreads()` 所述。
- threads/scheduler.cc `void UpdatePriority();`
  - 用來更新 aging 時增加的 priority，每 100 ticks 會被 `Alarm::CallBack()` 調用一次。
- threads/scheduler.cc `SortedList<Thread*>* L1ReadyQueue;`, `SortedList<Thread*>* L2ReadyQueue;`, and `List<Thread*>* L3ReadyQueue;`
  - Three level ready queues.
- threads/thread.h
  - `int ID;` 該 thread 的 ID。
  - `int Priority;` 該 thread 的優先級。
  - `int WaitTime;` 該 thread 已經等待多久，用來判斷是否 aging 而提高 priority。
  - `int RemainingBurstTime;` 該 thread 預期剩下多少 burst time，預期 burst time 的初始值由一開始調用 `userprog/nachos -epb <file_name1> <init_priority> <init_burst_time> ...` 那裡指定。
  - `int RunTime;` 該 thread 跑了多久了。
  - `int RRTime;` 該 thread 在 RR 模式中跑了多久了，供 L3 queue 切換。