NTHU Operating System Final Project: CPU Scheduling

Group: 21, 109033132 李臻茵, 109033151 李耀丞, 109033209 陳俊丞

Part I. Trace Code

(程式碼截圖與紫色字標示為<TODO>或更動說明)

1-1. New → Ready

- UserProgKernel::InitializeAllThreads()

  Explanation: 目的是初始化並啟用所有的 threads

  steps:

  1. For 迴圈遍歷所有需要執行的 file，每個 file 對應一個 threads
  2. int a...是利用 InitializeOneThread()為每一個 file 創建並初始一個 new thread，而這裡傳入了包括：為哪一個 file 做的 thread、此 thread 的優先級、thread 剩餘運行時間
  3. current...當所有 thread 初始化並啟動進程後，開始運作。

- UserProgKernel:: InitializeOneThread(char*, int, int)

  Explanation: 目的是為每個 process 創建並初始化一個 Thread

  steps:

  1. t[threadNum]->space = new AddrSpace();當前 thread 分配一個新的地址空間。AddrSpace()是管理程序內存地址空間的程式，負責分配和管理程序。
  2. t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);使用 Fork 函數啟動 new thread，並指定 ForkExecute 函數作為此 thread 的執行程式，ForkExecute 函數在 new thread 第一次運行時會被呼叫。
  3. threadNum++;創建完 thread 之後，總 thread 數量增加

4. return threadNum - 1;回傳此 thread 的編號，因為編號是從 0 開始標示所以-1。

5. <*cf*>ForkExecute function

   //加入此行確保 thread 有加入 CPU memory

   if(!t->space->Load(t->getName())) return;
   t->space->Execute(t->getName());

- Thread::Fork(VoidFunctionPtr, void*)

  Explanation: 方法負責創建一個新的線程，並將其添加到 ready queue 以準備運行，也用了 StackAllocate 來分配和初始化線程的 stack。

- Thread::StackAllocate(VoidFunctionPtr, void*)

  Explanation: 方法負責為線程分配 stack 並進行初始化，使得線程能夠正確地開始執行指定的函數；

  Conclusion: Fork(VoidFunctionPtr, void*)跟 StackAllocate(VoidFunctionPtr, void*)協作，來確保剛剛創建的 thread 可以正確地準備運行。

- Scheduler::ReadyToRun(Thread*)

  Explanation: 根據線程的優先級將其放入不同的準備隊列(L1, L2, L3);如果是放入 L1 ReadyQueue，還需要檢查是否需要搶占當前運行的線程。

```
100     // L1 queue: priority between 100 - 149
101     if (thread->getPriority() < 150 && thread->getPriority() >= 100){
102         DEBUG('z' , "[InsertToQueue] Tick [" << kernel->stats->totalTicks << "]:
    Thread [" << thread->getID() << "] is inserted into queue L[1]");
103         L1ReadyQueue->Insert(thread);
104     }
105     // L2 queue: priority between 50 - 99
106     else if (thread->getPriority() < 100 && thread->getPriority() >= 50) {
107         DEBUG('z' , "[InsertToQueue] Tick [" << kernel->stats->totalTicks << "]:
    Thread [" << thread->getID() << "] is inserted into queue L[2]");
108         L2ReadyQueue->Insert(thread);
109     }
110     //L3 queue: priority between 0 - 49
111     else if (thread->getPriority() < 50) {
112         DEBUG('z' , "[InsertToQueue] Tick [" << kernel->stats->totalTicks << "]:
    Thread [" << thread->getID() << "] is inserted into queue L[3]");
113         L3ReadyQueue->Append(thread);
114     }
```

1-2. Running → Ready

- Machine::Run()

  Machine::Run 模擬了在 Nachos 上執行 User process 的過程。這段程式碼負責執行每一條指令並管理中斷。

  1. *Instruction *instr = new Instruction;*創建一個 Instruction 類型的指針變數 instr，用來儲存解碼後的指令。
  2. *if (debug->IsEnabled('m')){//cout << "Starting program in thread: " << kernel->currentThread->getName();cout << ", at time: " << kernel->stats->totalTicks << "\n";}*，如果除錯模式啟用了'm'（通常代表與機器執行有關的除錯），則輸出當前時間。這部分程式碼的註釋掉的部分可顯示當前執行的 thread 名稱。
  3. *kernel->interrupt->setStatus(UserMode);*將中斷狀態設置為 UserMode，表示接下來的執行是在使用者模式下進行的，而不是內核模式。

- Interrupt::OneTick()

  目的：提前模擬時間並檢查是否有任何待呼叫的中斷。

  Two things can cause OneTick to be called:
  (1) interrupts are re-enabled
  (2) a user instruction is executed

- 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.
  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.

```
224     this->setRunTime(this->getRunTime() + this->getRRTime());
225     this->setRemainingBurstTime(this->getRemainingBurstTime() - this->getRRTime());
226     DEBUG('z',"[UpdateRemainingBurstTime] Tick [" << kernel->stats->totalTicks <<
    "]: Thread [" << this->getID()  << "] update approximate burst time, from: [" <<
    this->getRemainingBurstTime() + this->getRRTime() << "] - [" << this-
    >getRemainingBurstTime() << "], to [" << this->getRemainingBurstTime() << "]");
227     this->setRRTime(0);
228     nextThread = kernel->scheduler->FindNextToRun();
229     if (nextThread != NULL && this != nextThread) {
230         DEBUG('z',"[ContextSwitch] Tick [" << kernel->stats->totalTicks << "]:
    Thread [" << nextThread->getID()  << "] is now selected for execution, thread [" <<
    this->getID() << "] is replaced, and it has executed [" << this->getRunTime() << "]
    ticks");
231         kernel->scheduler->ReadyToRun(this);
232         kernel->scheduler->Run(nextThread, FALSE);
233     }
```

1. Put current_thread in running state to ready state
2. Then, find next thread from ready state to push on running state
3. After resetting some value of current_thread, then context switch

- Scheduler::FindNextToRun()

Explanation: FindNextToRun()回傳下一個要被調度到 CPU 上運行的 thread。如果沒有準備好的 thread，返回 NULL，我們讓他從 L1ReadyQueue, L2ReadyQueue, L3ReadyQueue 中按照優先級順序找出下一個 thread，如下所述。

```
132     if(!L1ReadyQueue->IsEmpty()){
133         thread = L1ReadyQueue->RemoveFront();
134         thread->setWaitTime(0);
135         DEBUG('z', "[RemoveFromQueue] Tick [" << kernel->stats->totalTicks << "]:
    Thread [" << thread->getID() << "] is remove from queue L[1]");
136     }
137     else if(!L2ReadyQueue->IsEmpty()){
138         thread = L2ReadyQueue->RemoveFront();
139         thread->setWaitTime(0);
140         DEBUG('z', "[RemoveFromQueue] Tick [" << kernel->stats->totalTicks << "]:
    Thread [" << thread->getID() << "] is remove from queue L[2]");
141     }
142     else if(!L3ReadyQueue->IsEmpty()){
143         thread = L3ReadyQueue->RemoveFront();
144         thread->setWaitTime(0);
145         DEBUG('z', "[RemoveFromQueue] Tick [" << kernel->stats->totalTicks << "]:
    Thread [" << thread->getID() << "] is remove from queue L[3]");
146     }
147     else{
148         thread = NULL;
149     }
150     return thread;
151 }
```

1. 找到欲執行的 thread 之後，先將其 WaitTime 設成 0
2. 如果在 L1, L2, L3ReadyQueue 都無欲執行的 thread，返回 NULL。

- Scheduler::ReadyToRun(Thread*)

  同 1-1.所述。

- Scheduler::Run(Thread*, bool)

  用於當一個執行緒尚未完成時要切換到另一個執行，若狀態為已完成則標記刪除

  當前執行緒，則須保存舊狀態並載入新的執行緒狀態。

  1. 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.
  2. In general case, we assume the state of the previously running thread has already been changed from running to blocked or ready (depending).
  3. 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)

1-3. Running → Waiting

- ExceptionHandler(ExceptionType) case SC_PrintInt

  先從 register-2 中讀取出系統調用類型:

  int type = kernel->machine->ReadRegister(2);

  syscal.h 中定義了各 system call 的編碼，其中 #define SC_PrintInt 11

  SC_PrintInt 用於從 register-4 中讀取整數值 val 打印到控制台。

  ```
  case SC_PrintInt:
    val = kernel->machine->ReadRegister(4);
    kernel->synchConsoleOut->PutInt(val);
    DEBUG(dbgMLFQ, "\n");
    return;
  ```

- SynchConsoleOutput::PutInt()

  synchconsole.h 用於解決同步問題，其中 PutInt()用於將整數輸出到控制台，透過

  lock->Acquire() 獲取鎖進入臨界區 critical section 以防止多個線程同時訪問共享

資源 consoleOutput，即控制台的輸出設備，並使用 waitFor->P()等待控制台信號，確保在輸出下一個字符之前控制台已輸出完成。

- ConsoleOutput::PutChar(char)

  首先檢查是否有其他寫入正在進行，若沒有則調用 WriteFile 將字符寫入文件中，並將 putBusy 設置為 true，最後安排一個行程以在 ConsoleTime 後觸發中斷清除 putBusy 標誌

- Semaphore::P()

  儲存當前中斷狀態後禁用，確保操作信號量 value 時的原子性，若是信號量為零則表示資源不可用，將當前線程加入 waiting queue 中後休眠；若大於零則減 1 表示消耗一個資源。

- SynchList<T>::Append(T)

  先獲取鎖後向同步列表中添加新項目 item，通知等待中的執行緒其條件已滿足，可以喚醒執行操作。

- Thread::Sleep(bool)

  檢查執行緒的一致性後將該狀態設為 Blocked ，並尋找下一個準備運行的執行緒，若無則進入 idle，否則開始運行下一個任務。

```
275     this->setRemainingBurstTime(this->getRemainingBurstTime() - this->getRRTime());
276     this->setRunTime(this->getRunTime() + this->getRRTime());
277     // if(!finishing)
278     DEBUG('z',"[UpdateRemainingBurstTime] Tick [" << kernel->stats->totalTicks <<
        "]: Thread [" << this->getID()  << "] update approximate burst time, from: [" <<
        this->getRemainingBurstTime() + this->getRRTime() << "] - [" << this->getRRTime() <<
        "], to [" << this->getRemainingBurstTime() << "]");
279     this->setRRTime(0);
280     if (nextThread != NULL && this != nextThread) {
281         DEBUG('z',"[ContextSwitch] Tick [" << kernel->stats->totalTicks << "]:
        Thread [" << nextThread->getID()  << "] is now selected for execution, thread [" <<
        this->getID() << "] is replaced, and it has executed [" << this->getRunTime() << "]
        ticks");
282         kernel->scheduler->Run(nextThread, finishing);
283     }
```

- Scheduler::FindNextToRun()

  同 1-2.及程式碼截圖所呈現。

- Scheduler::Run(Thread*, bool)

  同 1-2.所述。

1-4. Waiting → Ready

- Semaphore::V()

  用於增加訊號量 value，須確保操作原子性，在結束離開 critical section 時呼叫。

- Scheduler::ReadyToRun(Thread*)

  同 1-2.所述。

1-5. Running → Terminated(Start from Exit system call is called)

- ExceptionHandler(ExceptionType) case SC_Exit

  由 2 號 Register 中得到 Exception 類別細節。

  int type = kernel->machine->ReadRegister(2);

  當得到的類別是 SC_Exit 時會執行以下程式，可以知道程式持行的回傳狀態存放

  在 4 號 Register，並在最後把該 Thread 停下。

  ```
  case SC_Exit:
      DEBUG(dbgAddr, "Program exit\n");
      val=kernel->machine->ReadRegister(4);
      cout << "return value:" << val << endl;
      kernel->currentThread->Finish();
      break;
  ```

- Thread::Finish()

  在被 fork 出來的 thread 要結束時將資源釋放。因為不能直接釋放正在執行 thread

  的資源，因此會呼叫 scheduler 去使用 destructor 來協助處理。

- Thread::Sleep(bool)

  同 1-3.所述

- Scheduler::FindNextToRun()

  同 1-2.所述。

- Scheduler::Run(Thread*, bool)

  同 1-2.所述。

1-6. Ready → Running

- Scheduler::FindNextToRun()

  同 1-2.所述。

- Scheduler::Run(Thread*, bool)

  同 1-2.所述。

- SWITCH(Thread*, Thread*)

  SWITCH(Thread*, Thread*)是用來執行 context switch 的功能。在前半(第 16~28 行)會先將所有的 register 存到屬於要 swap out 的 thread 的儲存空間，接著更新目前使用 stack 的指標位置到新的 thread 的程式開頭(第 30 行)，並且在後半(第 32~43 行)把要 swap in 的 thread 資料更新到 CPU register。最後使用 ret 把程式跳到目前 stack 頂端(已在第 30 行更新到要 swap in 的 thread 的程式最開始處)

```
/* void SWITCH( thread *t1, thread *t2 )
**
** on entry, stack looks like this:
**      8(esp)  ->              thread *t2
**      4(esp)  ->              thread *t1
**       (esp)  ->              return address      teresasa0731, 2 days ago • Fir
**
** 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
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
```

- for loop in Machine::Run()

  這邊是無限迴圈，每次迴圈視為一個 cycle(這邊使用 tick 來代替)。每個 tick 都會執行一個新的指令，並且呼叫 kernel->interrupt->OneTick();來更新 tick 計數並且檢查其他每個 tick 需要更新的內容如排程演算法。

Part II. Implement

2.1 Multilevel queue

在 sortedList 中 cmp function 的定義：

(1) L1 ready queue - RemainingBurstTime 較短者優先

(2) L2 ready queue - PID 較小者優先

```
41  // Function 1. Function definition of sorting rule of L1 ReadyQueue
42  static int RemainingTimecmp(Thread* t1,Thread* t2){
43      return t1->getRemainingBurstTime() - t2->getRemainingBurstTime(); // preemptive SRTN
44  }
45
46  // Function 2. Function definition of sorting rule of L2 ReadyQueue
47  static int PIDcmp(Thread* t1,Thread* t2){
48      return t1->getID() - t2->getID(); // smaller pid do first
49  }
```

2.2 Aging

(1) alarm() 呼叫 updatePriority() 來更新 waiting time，在函式中分別對 L1, L2, L3 呼叫子函式 Aging(List<Thread *> *list) 進行更新與判斷。

```
274 void Scheduler::Aging(List<Thread *> *list)
275 {
276     ListIterator<Thread *> *iter = new ListIterator<Thread *>((List<Thread *> *)list);
277
278     for(;iter->IsDone() != true; iter->Next()){
279         Thread *iterThread = iter->Item();
280         // waiting time update
281         int oldWaitingTime = iterThread->getWaitTime();
282         iterThread->setWaitTime(oldWaitingTime + 100);
283         int oldPriority = iterThread->getPriority();
284         // aging detection
285         if((oldPriority >= 0 && oldPriority < 150) && oldWaitingTime >= 400) {
286             iterThread->setWaitTime(0);
287             iterThread->setPriority((oldPriority + 10 > 149) ? 149 : oldPriority + 10);
288             DEBUG('z' ,"[UpdatePriority] Tick [" << kernel->stats->totalTicks << "]: Thr
289             list->Remove(iterThread);
290             ReadyToRun(iterThread);
```

(2) 在子函式中做 waiting time + 100 後，判斷是否有超過 400 ticks，超過則將 priority +10 ，並移出當前 ready queue 後執行 ReadyToRun() 重新放回。

```
if(iterThread->getPriority() >= 100){
    // preemption condition
    // 1. current running Thread in L2 or L3
    // 2. current running Thread in L1 & Remaining Burst Time is greater
    if(kernel->currentThread->getPriority() < 100){
        kernel->interrupt->YieldOnReturn();
    }else if(kernel->currentThread->getPriority() >= 100 \
        && kernel->currentThread->getRemainingBurstTime() > iterThread->getRemainingBurstTime()){
        kernel->interrupt->YieldOnReturn();
    }
}
// aging thread put into L2 ready queue
else if(iterThread->getPriority() >= 50){
    // preemption condition
    // 1. currentThread in L3
    if(kernel->currentThread->getPriority() < 50){
        kernel->interrupt->YieldOnReturn();
    }
// aging thread put back into L3 ready queue
}else{
    L3ReadyQueue->Append(iterThread);
}
```

(3) 之後判斷是否有 preemption 情況：

A. 上移到不同 ready queue

B. 在 L1 且 Remaining burst time 較 current thread 小

C. 在 L2 且 PID 較 current thread 小

以上須提出 YieldOnReturn()，並在 OneTick() 中執行搶占。

2.3 Time counting & analysis

A. Time Parameter

```
int WaitTime;
int RemainingBurstTime;
int RunTime;
int RRTime;
```

我們定義 WaitTime 是在 ReadyQueue 中等待的時間長度，會在每次確認是否需要 Aging 之前更新，當超過 400 的時候就會執行 Aging 並且歸零 WaitTime。

RemainingBurstTime 是剩下需要執行的時間，我們會在每次要放棄 CPU 時更新。

RunTime 是目前總執行的時長，在 context switch 的時候 Debug 會印出目前的總執行

時長，會在每次要放棄 CPU 時更新。RRTime 是 Thread 把持 CPU 的時間，我們會在 Alarm::CallBack 中去更新。

B. Alarm function

我們在 Alarm::CallBack 中會更新 RRTime、UpdatePriority 跟檢查 RR 是否需要切換 Thread。我們透過 UpdatePriority 中的機制去更新 WaitTime，並且用 kernel->interrupt->YieldOnReturn()來在 RR 需要切換的時候讓系統去在下次執行 OneTick 切換 Thread。

2.4 epb function

我們因為看了 InitializeAllThreads 得知輸入指令要存的地方，並且自己確認一開始的資料輸入是否正確。

```
// Get execfile & its priority & burst time from argv, then save them.
else if (strcmp(argv[i], "-epb") == 0) {
    execfile[++execfileNum] = argv[++i];
    threadPriority[execfileNum] = atoi(argv[++i]);
    threadRemainingBurstTime[execfileNum] = atoi(argv[++i]);
    cout << "epb func get var : " << endl;
    cout << " execfile : " << execfile[execfileNum] << endl;
    cout << " current threadNum : " << execfileNum << endl;
    cout << " Priority : " << threadPriority[execfileNum] << endl;
    cout << " RemainingBurstTime : " << threadRemainingBurstTime[execfileNum] << endl;
}
```

Part III. Contribution

姓名	學號	貢獻
李臻茵	109033132	Trace code、Coding
李耀丞	109033151	Trace code、Coding
陳俊丞	109033209	Trace code、書面報告