MP3: CPU Rescheduling

110000132 詹振暘, 110021121陶威綸

Function Explanation

1-1. New→Ready

- **Kernel::ExecAll()**

  呼叫Excel執行所有的file。
- **Kernel::Exec(char*)**

  生成Thread、Addrspace空間，並且呼叫Fork。
- **Thread::Fork(VoidFunctionPtr, void*)**

  將輸入的函數傳到Stack中，並且呼叫ReadyToRun。
- **Thread::Stack Allocate(VoidFunctionPtr, void*)**

  分配一段大小為StackSize的stack(int)用於保存執行緒的局部變數、返回地址和其他運行時的資料。下面會根據不同CPU架構去初始化stackTop。
- **Scheduler::ReadyToRun(Thread*)**

  將Thread狀態設置為Ready，並將其加入到ready queue中，等待被schedular選中執行。

Summary:

ExecAll迭代所有檔案並且呼叫Exec執行，Exec生成Thread還有Addrspace空間，將Thread作為PCB紀錄資訊。接著呼叫Fork，傳入ForkExecute將其透過StackAllocate讀取到Stack中，最後再將此Thread透過ReadyToRun丟到ReadyList準備執行。

1-2. Running→Ready

- **Machine::Run()**

  模擬NachOS中CPU的循環執行(for(;;)) 並在其中(UserMode)模擬指令執行及電腦時序。
- **Interrupt::OneTick()**

  模擬時間（UserMode+1, KernelMode+10），在yeildOnReturn == True時，呼叫Yield。
- **Thread::Yield()**

  呼叫FindNextToRun找下一個Thread，並且呼叫Run做ContextSwitch。
- **Scheduler::FindNextToRun()**

  從 readyList 中選擇並返回準備執行的thread，若無thread需執行則返回NULL。
- **Scheduler::ReadyToRun(Thread*)**

  將Thread object放到ReadyList中，並將其設為running。
- **Scheduler::Run(Thread*, bool)**

  將當前thread的執行狀態保存並執行context switch，在完成放資源並從前面存取的狀態繼續執行。

Summary:

Machine::Run模擬CPU逐行執行instruction，並且呼叫OneTick，OneTick會呼叫Yield使用FindNextToRun找尋下一個Thread呼叫ReadyToRun將其放到readyList中，並在要使用時使用做ContextSwitch開始執行該程式。
當程式需要執行Context Switch的時候，currentThread->Yield會將正在執行中的thread透過ReadyToRun丟回ReadList並呼叫Run執行Context Switch跳到新的程式，如果舊的程式執行完畢會直接將它刪除。

1-3. Running→Waiting

(Note: only need to consider console output as an example)

- **SynchConsoleOutput::PutChar(char)**

首先透過Acquire()確保只有一個thread可以使用PutChar功能，接著輸出字元到consol，最後當輸出完成時，consoleOutput會釋放該Semaphore，允許其他thread繼續執行。

PutChar 會呼叫consoleOutput的putChar，呼叫Interrupt把ConsoleOutput的函式傳入pending(Waiting queue)，並傳入interrupt將發生的時間。

- **Semaphore::P()**

  當信號量的value為0時，阻塞當前thread（有其他Thread在scheduler->readyList則會把currentThread設成該Thread再進入一次while迴圈），直到信號量的value>0，一旦信號量可用，value--並繼續執行Thread。在執行時會把interrupt設為IntOff，以防Race或其他同步化問題發生。

- **List<T>::Append(T)**

  將T(item)放入list中，如果為空則first = last = item。否則把item接在last後面並設定last = item。放如前會確定該item不在list之中，並且放入後會確認list中有該元素。

- **Thread::Sleep(bool)**

  將Thread設為Blocked開始等待。如果readyList有其他process則會執行Run切到下個Process。如果沒有元素的話會執行kernel->interrupt->Idle()允許其他interrupt發生。

- **Scheduler::FindNextToRun()**

  從 readyList 中選擇並返回準備執行的thread，若無thread需執行則返回NULL。

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

  將當前thread的執行狀態保存並執行context switch，在完成放資源並從前面存取的狀態繼續執行。

Summary:

SynchConsoleOutput 呼叫 ConsoleOutput::PutChar() 時，會利用 Lock 確保只有一個 Thread 能執行。進入後，ConsoleOutput::PutChar() 檢查 putBusy 是否為 FALSE，若為空閒，將其設為 TRUE，然後執行 WriteFile 寫入字元。同時，ConsoleOutput 會將 SynchConsoleOutput 放入 kernel->interrupt->pending，並透過 kernel->interrupt->Schedule 設定中斷處理。接著，程式呼叫 Semaphore->P() 判斷是否有空閒 Thread，若沒有，將 Process 推入WaitingQueue，等待中斷發生；若有空閒，則消耗一個 Semaphore value 繼續執行。

1-4. Waiting→Ready

(Note: only need to consider console output as an example)

- Semaphore::V()

  當consol output結束時，會呼叫Semaphore::V()，檢查Semiphore，並且將裡面的thread丟到scheduler。

- Scheduler::ReadyToRun(Thread*)

  將Thread object放到ReadyList中，並將其設為running。

Summary:

當consol完成輸出時，Semaphore::V() 會移除queue中最前面等待的thread，並呼叫ReadyToRun()，將狀態設為 Ready，並將該thread加入 ReadyList。V()會被呼叫的時機有兩個，其一是在SynchDisk::CallBack()時（磁碟操作完成後喚醒等待該操作完成的thread），另一個是在Lock::Release()時，而Lock->Release()會在許多I/O, interrupt等等的情況被呼叫。

1-5. Running→Terminated

(Note: start from the Exit system call is called)

- **ExceptionHandler(ExceptionType) case SC Exit**

  根據switch(which)執行不同類型的exception，如SC_Halt、SC_Create、SC_Write等等，並在每次處理完成後更新程式計數器。

- **Thread::Finish()**

  終止thread並通過調用 Sleep(TRUE)，執行context switch。若所有thread都執行完成則呼叫Halt()終止整個NachOS。

- **Thread::Sleep(bool)**

  將當前thread設置為 BLOCKED，執行context switch。
- **Scheduler::FindNextToRun()**

  從 readyList 中選擇並返回準備執行的thread，若無thread需執行則返回NULL。
- **Scheduler::Run(Thread*, bool)**

  將當前thread的執行狀態保存並執行context switch，在完成放資源並從前面存取的狀態繼續執行。

Summary:

Thread 執行完成的 system call 會由 ExceptionHandler 捕獲，並調用 Finish() 終止此thread。Finish() 首先將thread設置為BLOCKED 狀態(不可恢復)，然後通過 Sleep(TRUE) 執行context switch。在context switch的過程中，Scheduler::FindNextToRun() 會負責從 ReadyList 中選擇下一個準備執行的thread，若無 executable thread，系統會進入IDLE state，等待中斷發生。

當新的thread被選中後，Scheduler::Run() 負責保存當前thread的context並加載新thread state，完成上 context switch。如果當前thread是系統中的最後一個，Finish() 會呼叫 kernel->interrupt->Halt()，終止 NachOS 的運行，關閉整個系統。

1-6. Ready→Running

- **Scheduler::FindNextToRun()**

  從 readyList 中選擇並返回準備執行的thread，若無thread需執行則返回NULL。
- **Scheduler::Run(Thread*, bool)**

  將當前thread的執行狀態保存並執行context switch，在完成放資源並從前面存取的狀態繼續執行。
- **SWITCH(Thread*, Thread*)**

  Switch內的兩個參數在實際運用時應是代表oldThread跟nextThread，它的作用即是執行 context switch，保存當前thread state並恢復下一個thread state。當 SWITCH() 被執行時，register 的資料會被存進 oldThread，並將 nextThread 的資料拿出來，將其加載到 CPU 的對應register中。此過程會更新extended stack pointer & program counter，使CPU從nextThread 的上一次執行位置繼續執行。

  此外，oldThread 的執行環境會被完整保存到其內部結構中，確保在未來需要切換回該 thread時，可以準確地恢復其執行的狀態。nextThread 則會被設置為current thread，並開始執行它的 context switch，完成thread之間的切換。這個過程是Thread Scheduler的運作中不可或缺的。

  在 x86 架構中，與指標相關的寄存器主要包括ESP (Extended Stack Pointer)、EBP (Extended Base Pointer)、ESI (Extended Source Index)、EDI (Extended Destination Index)、EBX(Extended Base Register)而在 SWITCH 的組合語言中，x86指令包含以下:

  1. movl %esp, offset(%eax): 將ESP的值存儲到 oldThread 的PCB（machineState）
  2. movl %ebp, offset(%eax): 將base address pointer register EBP的值存儲到oldThread 的 PCB。
  3. movl %ebx, offset(%eax): 將EBX(Extended Base Register)的值保存到PCB。
  4. movl offset(%ebx), %esi: 恢復來源Pointer Register ESI。
  5. movl offset(%ebx), %edi: 恢復目標Pointer Register EDI。
  6. movl offset(%ebx), %esp: 從nextThread的PCB中恢復stack pointer ESP的值，以切換到 new thread的stack。
  7. ret: 完成control flow的切換，跳轉到new thread的execution position。
- **[New,Running,Waiting]→Ready→Running)**

  1. **New→Ready→Running**

     Thread在初始化時會將自己的狀態設置為 New，並加入到 readyList，接著void Scheduler::ReadyToRun(Thread* thread) 將thread加入到readyList 並設置狀態為 Ready。在加入ReadyList後OS通過 FindNextToRun 找到readyList中優先度最高的thread，找到thread後通過void Scheduler::Run()來完成context switch並執行。

2. **Running→Ready→Running**

   有三種原因會造成Running->Ready，分別是Time slice expiration、Yield()、Higher priority threads enter Ready queue，MultiLevelFeedBackQueue::ShouldPreempt() 會判定是否需要切換執行的thread，並且通過 Thread::Yield() 正在執行的thread可以主動讓出 CPU。當在Ready狀態的thread需要重新執行時，做法會如上面(1) Ready→Running所提及的方法實現。

3. **Waiting→Ready→Running**

   thread在Waiting 狀態時，可以通過void Semaphore::Signal()中的ReadyToRun()將thread重新加入ready list中，並在Thread* Scheduler::FindNextToRun()中從ready list提取highest priority的thread。最後透過Schedular::Run()將選中的thread切換為 Running狀態：

- **for loop in Machine::Run()**

  此處的for迴圈在模擬CPU的運行，首先呼叫 OneInstruction，執行當前指令，並通過 OneTick 模擬硬體clk的前進，並處理可能的中斷或事件，不斷重複上面的工作，直到被終止system call 終止。

Summary:

當thread從 New 狀態初始化後，會首先透過 Scheduler::ReadyToRun() 被加入到 ready list，並將其狀態設置為 Ready。當NachOS需要選擇下一個執行的thread時，scheduler會調用 Scheduler::FindNextToRun() 從ready list中拿優先級最高的thread，檢查 ready list是否為空的，若有可用 thread，則根據schedular策略return最適合執行的thread。一旦選擇好下一個thread，scheduler接著會接著透過 Scheduler::Run() 進行上context switch，保存當前thread的狀態，將其寄存器內容（如 ESP、EBP、EBX 等）存入thread的 PCB 中，然後執行 SWITCH() 切換到下一個thread。

SWITCH()是context switch的關鍵部分，它保存當前thread的完整執行狀態，包括stack pointer和PC，然後恢復下一個thread的狀態。通過 movl 指令保存當前寄存器的值，再用 ret 指令跳轉到新thread的執行位置，完成從一個thread到另一個thread的切換。此外，若thread因等待 I/O 等資源而處於 Waiting 狀態，當條件滿足後，可以通业過 Semaphore::Signal() 喚醒thread，將其重新加入ready list，隨後重複上述過程進入 Running 狀態。

Implementation Explain

這次的實作，我們主要更改了threads/thread.*, threads/scheduler.*,threads/alarm.cc。其餘部分，我們變更了debug.*(新增flag), kernel.cc(處理 -ep, priority 傳輸)。

先從Thread開始，因為L1需要bustTime estimation，所以新增相關資訊（accTime, lastTick, apprxBurstTime, shouldPreempt）。更改Yeild, Sleep, Thread三個函數更新burstTime。

```
class Thread {  (省略未更改的部分)
  public:
   Thread(char *debugName, int threadID);  // initialize a Thread
   ~Thread();                              // deallocate a Thread
   void Yield();
   void Sleep(bool finishing);

   void setBurstTime(int bt){burstTime=bt;}
   int getBurstTime(){burstTime = approxBurstTime - accTime; return burstTime;}
   void setPriority(int threadPriority){priority = threadPriority;}
   int getPriority(){ return (priority);}
   void updateLastTick(int tick){lastTick = tick;};
   int accTime; // accumulate running time.
   int lastTick; // for calculating accTime
   int approxBurstTime;
```

```
    int burstTime; // burst time for scheduler
private:
    int priority;
};
```

BurstTime的管理如下:

當一個Thread被新增時, 所有值均為0, 遇到Yield則暫時切斷accTime, 並且在重新呼叫後繼續累積accTime。如果遇到Sleep, 更新approxBurstTime並且重設其他的值。每當程式需要判斷L1優先順序時, 我們需要知道remainingBurstTime, 所以呼叫getBurstTime(), 剩餘時間為approxBurstTime(這個Thread在這次運行被預估的burstTime)扣掉accTime(目前已經跑了多久), 即approxBurstTime - accTime。

```
Thread::Thread(char *threadName, int threadID) {
    accTime = 0;
    burstTime = 0;
    approxBurstTime = 0;
    shouldPreempt = FALSE;
}
```

Thread::Yield()
accTime += (totalTicks - lastTick) 現在的時間減去這個Thread開始跑的時間(這個Thread跑了多久)。
burstTime = this->apprxBurstTime - this->accTime概念同getBurstTime()。

```
void Thread::Yield() {
    if (nextThread != NULL) {
        this->accTime += kernel->stats->totalTicks - this->lastTick;
        this->burstTime = this->approxBurstTime - this->accTime;
}}
```

Thread::Sleep(bool finishing)
更新accTime後, 在Sleep(Running -> Waiting)時重新計算下次分配給Thread的apprxBurstTime。
DEBUG Message。
重設accTime, burstTime, approxBurstTime。

```
void Thread::Sleep(bool finishing) {
    // update accTime, renew burstTime reset accTime
    this->accTime += (kernel->stats->totalTicks - this->lastTick);
    int newApproxBurstTime = 0.5*this->approxBurstTime + 0.5*this->accTime;

    DEBUG(dbgScheduler, "[D] Tick [" << kernel->stats->totalTicks << "]: Thread ["
<< this->getID() << "] update approximate burst time, from: [" <<
newApproxBurstTime << "], add [" << accTime << "], to ["<< newApproxBurstTime
<<"]");

    this->approxBurstTime = newApproxBurstTime;
    this->burstTime = this->approxBurstTime - this->accTime;
    this->accTime = 0;
}
```

上述程式中有提到lastTick, 由於lastTick從程式開始執行時計算, 因此更新lastTick的地點在Scheduler::Run。

Scheduler大部分沒有做什麼更動, 因為新的ReadyList跟原本的要Call的函式一樣, 有改動的是在Run的時間要重設Thread lastTick, 還有ShouldPreempt跟Aging這兩個函式。

```
class Scheduler { (省略未更改的部分)
```

```
  public:
    Scheduler();   // Initialize list of ready threads
    void Run(Thread* nextThread, bool finishing);
    bool ShouldPreempt(){return readyList->ShouldPreempt();}
    void Aging();
  private:
    MultiLevelFeedBackQueue* readyList;
};

Scheduler::Scheduler() {
    // readyList = new List<Thread *>;
    readyList = new MultiLevelFeedBackQueue();
    toBeDestroyed = NULL;
}
```

Scheduler::readyToRun(Thread* thread)
更新lastTick紀錄他在readyList等待多久。(如果太久會在alarm::CallBack執行ContextSwitch。)

```
void Scheduler::ReadyToRun(Thread *thread) {
    thread->lastTick = kernel->stats->totalTicks;
    thread->setStatus(READY);
    readyList->Append(thread);
}
```

Scheduler::Run(nextThread, finishing) DEBUG Message, 因為執行Run時, currentThread(oldThread)已經經過Yield或是Sleep設定了accTime, 因此我們只需要把nextThread的lastTick記錄下來(重啟accTime), 就可以執行ContextSwitch了。

```
void Scheduler::Run(Thread *nextThread, bool finishing) {
    DEBUG(dbgScheduler, "[E] Tick [" << kernel->stats->totalTicks << "]: Thread ["
<< nextThread->getID() << "] is now selected for execution, thread [" <<
oldThread->getID() << "] is replaced, and it has executed [" << oldThread->accTime
<< "] ticks");
    nextThread->updateLastTick(kernel->stats->totalTicks);
    DEBUG(dbgThread, "Switching from: " << oldThread->getName() << " to: " <<
nextThread->getName());
    SWITCH(oldThread, nextThread);
}
```

Aging跟Preept一樣留在後面說明。

MultiLevelFeedBackQueue Implementation

我們在scheduler.cc放了一個MultiLevelFeedBackQueue取代原本的scheduler, 我們實作了跟原本List<Thread*>一樣的API, 並且新增getLevel, Aging這兩個Function。在這個Queue中, 我們設定L1, L2, L3三種小的queue分別對應三種演算法。我們使用SortedQueuet當作L1, L2, 因為SortedQueue自動將最小值放在首位, 我們透過實作簡單的CompareFunction, 就完成L1, L22的結構。L1的CompareFunction為CompareBurstTime, L2的function則為CompareL2Priority。L3為round-robin不需要排序因此用基本的List<Thread*>即可。

scheduler.h:

```
class MultiLevelFeedBackQueue{
public:
  MultiLevelFeedBackQueue();
  ~MultiLevelFeedBackQueue();

  Thread* Front();
  Thread* RemoveFront();
  void Append(Thread* item);
  void Aging();
  bool IsEmpty();
  bool ShouldPreempt();
  void Apply(void (*f)(Thread*)) const;

protected:
  int getLevel(int priority);
  int numInList;
  int l1ApproxBurst;
  int L3TimeQuantum;
  int maxWaitTime;

  SortedList<Thread*>* L1;
  SortedList<Thread*>* L2;
  List<Thread*>* L3;
};
```

以下為各個Function的解釋:

Aging, ShouldPreempty因為較為複雜, 留到之後做解釋。

1. MultiLevelFeedBackQueue::Append()
   呼叫getLevel(t->getPriority())找到Thread的level, 再依照三個Level把thread丟到對應的Queue上。
2. MultiLevelFeedBackQueue::IsEmpty()
   回傳 L1.IsEmpty() && L2.IsEmpty() && L3.IsEmpty(), 三個必須同時為Empty才是Empty。
3. MultiLevelFeedBackQueue::Front()
   Front回傳即將執行的Thread地址, 因為L1, L2為SortedList我們只需要依序抓去L1, L2, L3的元素, 若有則立馬回傳就可以拿到我們想要的Thread。
4. MultiLevelFeedBackQueue::RemoveFront()
   跟Font極為相似, 差別在RemoveFront呼叫Li->RemoveFront而非Li->Front。
5. MultiLevelFeedBackQueue::Apply(func)
   L1->Apply(func); L2->Apply(func); L3->Apply(func);
6. MultiLevelFeedBackQueue::getLevel(int priority)
   依照分界情況拿取priority

Scheduler.cc:

```
int CompareBurstTime(Thread* t1, Thread* t2){
   int bt1 = t1->getBurstTime();
   int bt2 = t2->getBurstTime();
   if (bt1 == bt2){
   bt1 = t1->getID();
```

```
    bt2 = t2->getID();
    }
    if (bt1 > bt2) return 1;
    else if (bt1 < bt2) return -1;
    return 0;
}

int CompareL2Priority(Thread* t1, Thread* t2) {
    int p1 = t1->getPriority();
    int p2 = t2->getPriority();

    if (p1 != p2) {
        return p2 - p1;
    }

    return t1->getID() - t2->getID();
}

MultiLevelFeedBackQueue::MultiLevelFeedBackQueue(){
    l1ApproxBurst = 0;
    maxWaitTime = 1500;
    L1 = new SortedList<Thread*>(CompareBurstTime);
    L2 = new SortedList<Thread*>(CompareL2Priority);
    L3 = new List<Thread*>();
}

MultiLevelFeedBackQueue::~MultiLevelFeedBackQueue(){
    delete L1;
    delete L2;
    delete L3;
}

void MultiLevelFeedBackQueue::Append(Thread* item){
    int level = getLevel(item->getPriority());
    switch (level){
        case 1:
        L1->Insert(item);
        break;
    case 2:
        L2->Insert(item);
        break;
    case 3:
        L3->Append(item);
        break;
    }
```

```
    DEBUG(dbgScheduler, "[A] Tick [" << kernel->stats->totalTicks  << "]: Thread ["
<< item->getID() << "] is inserted into queue L[" << level << "]");
}

bool MultiLevelFeedBackQueue::IsEmpty(){
    return L1->IsEmpty() && L2->IsEmpty() && L3->IsEmpty();
}

Thread* MultiLevelFeedBackQueue::Front(){
    Thread* selected = NULL;
    if (!L1->IsEmpty()){
    selected = L1->Front();
    }
    if (!selected && !L2->IsEmpty()){
    selected = L2->Front();
    }
    if (!selected && !L3->IsEmpty()){
    selected = L3->Front();
    }
    return selected;
}



Thread* MultiLevelFeedBackQueue::RemoveFront(){
    Thread* toBeRemoved;
    int level = -1;
    if (!L1->IsEmpty()){
    toBeRemoved = L1->RemoveFront();
    level = 1;
    }else if (!L2->IsEmpty()){
    toBeRemoved = L2->RemoveFront();
    level = 2;
    }else if (!L3->IsEmpty()){
    toBeRemoved = L3->RemoveFront();
    level = 3;
    }
    DEBUG(dbgScheduler, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread ["
<< toBeRemoved->getID() << "] is removed from queue L[" << level << "]");
    ASSERT(toBeRemoved);
    return toBeRemoved;
}

void MultiLevelFeedBackQueue::Apply(void (*f)(Thread*)) const{
    L1->Apply(f);
```

```
    L2->Apply(f);
    L3->Apply(f);
}

int MultiLevelFeedBackQueue::getLevel(int priority){
    if (priority >= 100){
        return 1;
    }else if (priority >= 50){
        return 2;
    }else if (priority >= 0){
        return 3;
    }
    ASSERT(TRUE);
    return -1;
}
```

L1 Queue: Preemptive Shortest Job First (SJF)

每當Thread從WaitingState或是任何狀態進入到Ready，在Thread我們已經解釋burstTime是如何更新。現在只需要處理判斷Thread先後順序還有Preempt。就可以確定1. Front, RemoveFront函式可以拿到優先次序最高的Thread，2. 該Thread會被立馬執行。

1. 判斷先後順序：

根據Spec3，優先回傳較小remainingBurstTime且較小ID（如果前者相同）的Thread。

```
int CompareBurstTime(Thread* t1, Thread* t2){
    int bt1 = t1->getBurstTime();
    int bt2 = t2->getBurstTime();
    if (bt1 == bt2){
    bt1 = t1->getID();
    bt2 = t2->getID();
    }
    if (bt1 > bt2) return 1;
    else if (bt1 < bt2) return -1;
    return 0;
}
```

2. Thread立馬被執行：

根據Spce3，這裡的程式會等到Alarm被呼叫才會做Preempt，所以我們直接定義一個shouldPreempt放在MultiLevelFeedBackQueue跟Scheduler上，便於alarm呼叫 kernel->scheduler->ShouldPreempt()。
不過ShouldPreempt機制涵蓋L2跟L3所以在後面介紹，這邊只放code。

```
bool MultiLevelFeedBackQueue::ShouldPreempt() {
    Thread* curThread = kernel->currentThread;
    int curLevel = getLevel(curThread->getPriority());
    if (curLevel == 3 && curThread->accTime >= L3TimeQuantum) {return TRUE;}
    // curLevel < nextThread
    Thread* next = Front();
    if (!next) return FALSE;
    int nextLevel = getLevel(next->getPriority());
```

```
    // nextLevel < curLevel then should be preempted.
    if (nextLevel < curLevel) return TRUE;
    if (nextLevel == curLevel && curLevel == 1) {
        return CompareBurstTime(curThread, next) == 1;
    }
    return FALSE;
}
```

L2 Queue: Non-Preemptive Priority Scheduling

L2Queue實作non-preemptive priority scheduling，同L1，我們只需要確定L2Queue排序正確，就能每次拿取最優先的值。因此我們也使用SortedList，在這裡附上CompareFunction。
取priority高的Thread，如果一樣則取ID比較小的Thread。

```
int CompareL2Priority(Thread* t1, Thread* t2) {
    int p1 = t1->getPriority();
    int p2 = t2->getPriority();

    if (p1 != p2) {
        return p2 - p1;
    }

    return t1->getID() - t2->getID();
}
```

L3 Queue: Round Robin

最後一個則是round-robin，這個演算法是FIFO所以只需要用一般的List<Thread*>。每當單一個Thread執行，如果以執行時間（accTime）大於TimeQuantum則會switch到下一個process並重新且新增到L3Queue中。從L3到L3可以視為從L3Preempt到L3，因此我們把這個功能也寫在ShouldPreempt。

Aging

因為只有在alarmCallBack才會觸發ContextSwitch，我們可以直接把Aging功能放在Callback裏面。
Aging實作方式就是看L1, L2, L3，檢查每一個Thread的WaitTime(=stats->totalTicks - t->lastTick)，如果WaitTime>=1500則將priority+=10並且重新設定lastTick（新一輪等待）。

```
static void agingForQueue(Thread* t){

    int waitTime = kernel->stats->totalTicks - t->lastTick;
    // int oldLevel = getLevel(t->getPriority());
    int oldPriority = t->getPriority();
    if (waitTime >= 1500){
        t->lastTick = kernel->stats->totalTicks;
        t->setPriority(oldPriority + 10);
        DEBUG(dbgScheduler, "[C] Tick [" << t->lastTick  << "]: Thread ["<<
t->getID()  << "] changes its priority from [" << oldPriority << "] to [" <<
t->getPriority() << "]");
```

```
    }
}
```

更新完priority後，有可能會有level改變，我們需要迭代L2, L3，找到這些應該更換Queue的Thread並且將他們新增到新的Queue上面。

```
void MultiLevelFeedBackQueue::Aging(){
    L1->Apply(agingForQueue);
    L2->Apply(agingForQueue);
    L3->Apply(agingForQueue);

    Thread * to_ch = NULL;
    for (ListIterator<Thread *> it(L3); ; it.Next()) {
    if (to_ch) {
        L3->Remove(to_ch);
            Append(to_ch);
            to_ch = NULL;
        }
        if (it.IsDone()){
            break;
    }
        if (getLevel(it.Item()->getPriority()) != 3){
            to_ch = it.Item();
    }
    }

    for (ListIterator<Thread *> it(L2); ; it.Next()) {
    if (to_ch){
        L2->Remove(to_ch);
            Append(to_ch);
            to_ch = NULL;
    }

        if (it.IsDone()){
        break;
        }
        if (getLevel(it.Item()->getPriority()) != 2){
            to_ch = it.Item();
        }
    }
}
```

更新完後，因為所有的Queue元素都正確，因此下次ShouldPreempt呼叫Front抓取元素或是其他物件呼叫RemoveFront時，MultiLevelFeedBackQueue可以正確地將最重要的thread丟出。

ShouldPreempt

每當L1, L2, L3出現需要preempt的情況我們都會呼叫ShouldPreempt判斷，並且執行ContextSwitch，因此ShouldPreempt就是用來判斷MultiLevelQueue是否有需要立即處理的情況。

依照Spec3要求，每次CallBack我們都會再檢查一次preempt條件，因此我們將此函式放在CallBack中，如果shouldPreempt則呼叫YieldOnReturn再之後執行ConetxtSwitch。
注意在Call ShouldPreempt前，我們需要呼叫scheduler->Aging()，我們需要把所有的狀態更新後呼叫ShouldPreempt判斷才會滿足Spec上的要求。

Alarm::CallBack()

```
void Alarm::CallBack() {
   Interrupt *interrupt = kernel->interrupt;
   MachineStatus status = interrupt->getStatus();
   kernel->scheduler->Aging();
   if (status != IdleMode && kernel->scheduler->ShouldPreempt()) {
       interrupt->YieldOnReturn();
   }}
```

（這就是一個接口而已）
Scheduler::ShouldPreempt()

```
bool ShouldPreempt(){return readyList->ShouldPreempt();}
```

依照Spec3，發生preempt的情況，

1. Level優先（新進來的Thread的level大於currentThread的level）
2. Level相等=1，但current的remainingBurstTime
3. Level相等=3，但currentThread的累計時間大於L3TimeQuantum

MultiLevelFeedBackQueue::ShouldPreempt()

```
bool MultiLevelFeedBackQueue::ShouldPreempt() {
   Thread* curThread = kernel->currentThread;
   int curLevel = getLevel(curThread->getPriority());
3.
   if (curLevel == 3 && curThread->accTime >= L3TimeQuantum) {return TRUE;}
   // curLevel < nextThread
   Thread* next = Front();
   if (!next) return FALSE;
   int nextLevel = getLevel(next->getPriority());
   // nextLevel < curLevel then should be preempted.
1.
   if (nextLevel < curLevel) return TRUE;
2.
   if (nextLevel == curLevel && curLevel == 1) {
       return CompareBurstTime(curThread, next) == 1;
   }
   return FALSE;
}
```

在Kernel::Exec(char *name, char* priority)中新增了char* priority變數，並在function中加入setPriority(atoi(priority))用於將新建立的thread設定其在ready list中的priority。

```
int Kernel::Exec(char *name, char* priority) {
   t[threadNum] = new Thread(name, threadNum);
   t[threadNum]->setPriority(atoi(priority));
```

```
    t[threadNum]->setIsExec();
    t[threadNum]->space = new AddrSpace();
    t[threadNum]->Fork((VoidFunctionPtr)&ForkExecute, (void *)t[threadNum]);
    threadNum++;

    return threadNum - 1;
}
```

kernel.cc的Kernel::Kernel()中，新增了以下一個case，處理 -ep，解析命令列中指定的執行檔案及其priority，並將這些資訊存入execfile[execfileNum] = argv[++i], priority[execfileNum] = argv[++i]陣列，提供給multi thread scheduling使用。

```
else if (strcmp(argv[i], "-ep") == 0){
        execfileNum++;
        execfile[execfileNum] = argv[++i];
        priority[execfileNum] = argv[++i];
        cout << execfile[execfileNum] << " with priority: " << priority[execfileNum]
<< "\n";
```

在debug.h中，新增了一個dbgSchedular = 'z'，用來表示Scheduler相關的Debug Message。

```
const char dbgScheduler = 'z';
```

[A] 實作在scheduler.cc的MultiLevelFeedBackQueue::Append(Thread* item)中

```
DEBUG(dbgScheduler, "[A] Tick [" << kernel->stats->totalTicks  << "]: Thread [" <<
item->getID() << "] is inserted into queue L[" << level << "]");
```

[B] 實作在scheduler.cc的MultiLevelFeedBackQueue::RemoveFront()中

```
DEBUG(dbgScheduler, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" <<
toBeRemoved->getID() << "] is removed from queue L[" << level << "]");
```

[C] 實作在scheduler.cc的agingForQueue(Thread* t)中

```
DEBUG(dbgScheduler, "[C] Tick [" << t->lastTick  << "]: Thread ["<< t->getID()  <<
"] changes its priority from [" << oldPriority << "] to [" << t->getPriority() <<
"]");
```

[D] 實作在scheduler.cc的Thread::Sleep(bool finishing)中

```
DEBUG(dbgScheduler, "[D] Tick [" << kernel->stats->totalTicks << "]: Thread [" <<
this->getID() << "] update approximate burst time, from: [" << approxBurstTime <<
"], add [" << accTime << "], to ["<< newApproxBurstTime <<"]");
```

[E] 實作在scheduler.cc的Scheduler::Run(Thread *nextThread, bool finishing)中

```
DEBUG(dbgScheduler, "[E] Tick [" << kernel->stats->totalTicks << "]: Thread [" <<
nextThread->getID() << "] is now selected for execution, thread [" <<
oldThread->getID() << "] is replaced, and it has executed [" << oldThread->accTime
<< "] ticks");
```

分工

陶威倫：Code tracing, Report, Coding(L1, L3, aging)

詹振暘：Code tracing, Report, Coding(L2)