Operating System

MP3: CPU scheduling

Team 73

Trace Code & Implementation: 1121036s 郭逸洪

Contents

Part I - Trace Code

1-1. New -> Ready

a. threads/kernel.cc, void Kernel::ExecAll()
   - 將nachos執行指令中-e的引數逐一傳入Kernel::Exec
   - 完成後呼叫Thread::Finish結束main thread

b. threads/kernel.cc, int Kernel::Exec(char *name)
   - 傳入名稱與編號創建Thread，執行緒初始狀態為JUST_CREATED（即New）
   - 創建該執行緒的AddrSpace
   - 呼叫該執行緒的Fork方法，傳入一個指到ForkExecute的function pointer與指向該執行緒的指標
   - 增加kernel的執行緒編號
   - 回傳稍早創建的執行緒編號

c. threads/thread.cc, void Thread::Fork(VoidFunctionPtr func, void *arg)
   - 將自己的兩個參數再傳入Thread::StackAllocate以配置此執行緒的stack
   - 關閉interrupt
   - 呼叫Scheduler::ReadyToRun
   - 恢復舊的interrupt狀態

d. threads/thread.cc, void Thread::StackAllocate(VoidFunctionPtr func, void *arg)
   - 呼叫sysdep.cc的AllocBoundedArray直接在host的UNIX配置stack所需的空間
   - 將stackTop指向stack最後一個整數的位置
   - 將組合語言程序ThreadRoot push到stack，此程序定義在switch.S
   - 在stack底部設置STACK_FENCEPOST，用於檢查stack是否發生overflow
   - 將執行緒初始化方法、執行方法、執行參數、結束方法等指標設定到machineState，這些指標將在ThreadRoot程序中被使用

e. threads/scheduler.cc, void Scheduler::ReadyToRun(Thread *thread)
   - 確保interrupt被關閉
   - 將傳入的執行緒的狀態更新為READY
   - 呼叫List<T>::Append將傳入的執行緒放入readyList後面

1-2. Running -> Ready

a. machine/mipssim.cc, void Machine::Run()
   - 切換到UserMode
   - 在無窮迴圈中呼叫Machine::OneInstruction，在執行完指令後若未exit，呼叫Interrupt::OneTick()使kernel的時間前進

b. machine/interrupt.cc, void Interrupt::OneTick()
   - 使kernel的時間前進SystemTick（10）或UserTick（1），取決於當下是在UserMode或SystemMode
   - 關閉interrupt，使用CheckIfDue檢查目前是否有需要執行的的pending interrupt後，重新打開interrupt
   - 如果目前可以進行context switch，即yieldOnReturn被Timer設定為TRUE，則將yieldOnReturn設為FALSE，切換到SystemMode後呼叫kernel->currentThread->Yield()，當此執行緒又可以執行時，恢復MachineStatus

c. threads/thread.cc, void Thread::Yield()
   - 關閉interrupt確保接下來的操作不被context switch中斷
   - 確保執行yield的執行緒為kernel的currentThread
   - 呼叫Scheduler::FindNextToRun取得下一個可執行的執行緒
   - 如果有下一個可執行的執行緒，調用Scheduler::ReadyToRun將目前的執行緒重新放回Scheduler的readyList後，呼叫Scheduler::Run執行context switch，因原先的執行緒還未結束，傳入Scheduler::Run的finishing參數為FALSE
   - 恢復interrupt

d. threads/scheduler.cc, Thread *Scheduler::FindNextToRun()
   - 對readyList進行操作前，先確保interrupt被關閉
   - 如果readyList為空，回傳NULL，否則從readyList前面pop一個執行緒並回傳

e. threads/scheduler.cc, void Scheduler::ReadyToRun(Thread *thread)
   - 參見先前的段落

f. threads/scheduler.cc, void Scheduler::Run(Thread *nextThread, bool finishing)
   - 將kernel的currentThread保存於oldThread變數
   - 如果傳入的finishing flag為TRUE，將toBeDestroyed設為oldThread

- 如果oldThread的space不為空，表示其為user program，呼叫Thread::SaveUserState及AddrSpace::SaveState，保存該執行緒的CPU暫存器和address space
- 檢查oldThread是否發生stack overflow
- 將currentThread設為要執行的nextThread，並將其狀態改為RUNNING
- 呼叫以組合語言定義的SWITCH，在host UNIX機器上切換oldThread和nextThread的context
- 確保interrupt已被關閉
- 呼叫Scheduler::CheckToBeDestroyed檢查是否有任何已完成的執行緒需要被清理
- 如果oldThread的address space需被復原，則呼叫Thread::RestoreUserState及AddrSpace::RestoreState，復原oldThread的CPU暫存器和page table狀態

1-3. Running -> Waiting

a. userprog/synchconsole.cc, void SynchConsoleOutput::PutChar(char ch)

- 將SynchConsoleOutput的lock鎖上，進入critical section
- 調用ConsoleOutput::PutChar輸出一個字元到console，並排程一個ConsoleTime後發生的interrupt模擬IO裝置輸出完畢通知kernel的情境
- 調用waitFor->P()，因為建構SynchConsoleOutput::waitFor的初始值為0，此時currentThread會被放入Semaphore的queue中，並且執行緒的狀態被更新為BLOCKED（即Waiting），直到ConsoleTime後的interrupt觸發SynchConsoleOutput::CallBack及waitFor->V()才會恢復為READY狀態
- 釋放SynchConsoleOutput的lock

b. threads/synch.cc, void Semaphore::P()

- 關閉interrupt確保接下來的操作不被context switch中斷
- 如果現在的值為0，將Append到queue中，並使其Sleep（傳入的finishing為FALSE）
- 將值-1
- 恢復interrupt

c. lib/list.cc, void List<T>::Append(T item)

- 建立ListElement包裹傳入的item
- 確保這個item不在目前的list中

    - 操作指標將新的element加到linked list後面
    - 增加list中的元素計數
    - 確保這個item在目前的list中
d. threads/thread.cc, void Thread::Sleep(bool finishing)
    - 確保要被sleep的執行緒為kernel的currentThread
    - 確保interrupt已被關閉
    - 將該執行緒的狀態改為BLOCKED(即Waiting)
    - 調用Scheduler::FindNextToRun取得下一個可執行的執行緒，如果scheduler的readyList中沒有任何執行緒，呼叫Interrupt::Idle直到有其他執行緒可執行
    - Scheduler安排其他執行緒執行，呼叫Scheduler::Run執行context switch，並傳入finishing參數
e. threads/scheduler.cc, Thread *Scheduler::FindNextToRun()
    - 參見先前的段落
f. threads/scheduler.cc, void Scheduler::Run(Thread *nextThread, bool finishing)
    - 參見先前的段落

1-4. Waiting -> Ready

a. threads/synch.cc, void Semaphore::V()
    - 關閉interrupt確保接下來的操作不被context switch中斷
    - 若queue非空，取出最前面的執行緒，調用Scheduler::ReadyToRun將其狀態更新為READY
    - 將值+1
    - 恢復interrupt
b. threads/scheduler.cc, void Scheduler::ReadyToRun(Thread *thread)
    - 參見先前的段落

1-5. Running -> Terminated

a. userprog/exception.cc, void ExceptionHandler(ExceptionType which), case SC_Exit
    - 從4號暫存器讀出程序的回傳值
    - 調用Thread::Finish終結currentThread
b. threads/thread.cc, void Thread::Finish()
    - 關閉interrupt確保接下來的操作不被context switch中斷

- 確保被終止的執行緒是kernel的currentThread
- 呼叫Thread::Sleep並傳入TRUE以終止執行緒

c. threads/thread.cc, void Thread::Sleep(bool finishing)
- 參見先前的段落

d. threads/scheduler.cc, Thread *Scheduler::FindNextToRun()
- 參見先前的段落

e. threads/scheduler.cc, void Scheduler::Run(Thread *nextThread, bool finishing)
- 參見先前的段落

1-6. Ready -> Running

a. threads/scheduler.cc, Thread *Scheduler::FindNextToRun()
- 參見先前的段落

b. threads/scheduler.cc, void Scheduler::Run(Thread *nextThread, bool finishing)
- 參見先前的段落

c. threads/thread.h, void SWITCH(Thread *oldThread, Thread *newThread)
- 儲存oldThread的%ebx、%ecx、%edx、%esi、%edi、%ebp、%esp、%eax及return address
- 載入newThread的%ebx、%ecx、%edx、%esi、%edi、%ebp、%esp、return address及%eax
- 呼叫ret返回newThread的PC執行newThread

d. (depends on the previous process state, e.g., [New, Running, Waiting]→Ready)
- 從New->Ready的執行緒context switch轉為Running後出現在Thread::Begin()，之後開始執行其對應的function，進入Machine::Run
- 從Waiting->Ready的執行緒context switch轉為Running後出現在Semaphore::P()的while迴圈外第一行，將value -1並恢復interrupt後return
- 從Running->Ready的執行緒context switch轉為Running後出現在Machine::Run的for迴圈中繼續執行

e. for loop in Machine::Run()
- newThread通過ForkExecute及AddrSpace::Execute進入Machine::Run的無窮迴圈中執行指令

Part II - Implementation

1. code/threads/kernel.h

新增私有成員threadPriorities儲存各執行緒的初始priority設定。

```
class Kernel
{
   // ......
private:
  int threadPriorities[10];
};
```

2. code/threads/kernel.cc

在Kernel建構式新增-ep命令參數以設定各執行緒的初始priority，執行緒預設的priority為0。

```
Kernel::Kernel(int argc, char **argv) : frameTable(Bitmap(NumPhysPages))
{
    for (int i = 1; i < argc; i++)
    {   // ......
        else if (strcmp(argv[i], "-e") == 0)
        {
            execfile[++execfileNum] = argv[++i];
            threadPriorities[execfileNum] = 0;
            cout << execfile[execfileNum] << "\n";
        }
        else if (strcmp(argv[i], "-ep") == 0)
        {
            ASSERT(i + 2 < argc);
            execfile[++execfileNum] = argv[++i];
            threadPriorities[execfileNum] = atoi(argv[++i]);
            DEBUG(dbgBeta, "thread " << execfile[execfileNum] << " initial
priority " << threadPriorities[execfileNum]);
        }
        // ......
    }
}
```

在Kernel::Exec建構執行緒時，設定初始priority。

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

    return threadNum - 1;
}
```

因應Thread::setStatus的調整，在Kernel::Initialize先建構stats再依序轉換狀態。

```
void Kernel::Initialize()
{
    // ......
    stats = new Statistics();
    currentThread = new Thread("main", threadNum++);
    currentThread->setStatus(READY, stats->totalTicks);
    currentThread->setStatus(RUNNING, stats->totalTicks);
    // ......
}
```

3. code/threads/thread.h, class Thread

新增私有成員priority及公開成員ts(用於管理執行緒的統計資訊)。新增私有方法getStatusString將ThreadStatus轉換為字串供除錯使用。調整Thread::setStatus以封裝執行緒狀態變化時統計資訊的維護。將部分舊方法加上const修飾詞，以便在其他const方法中使用。新增公開方法getRemainingCpuBurstTime供Scheduler取得預估剩餘的CPU burst時間。

```
class Thread
{
private:
  // ......
  int priority;
  const char *getStatusString(ThreadStatus st);
  // ......
public:
  // ......
  ThreadStatistics *ts;
  // change thread status and update statistics info
  void setStatus(ThreadStatus st, const int totalTicks);
```

```
  ThreadStatus getStatus() const { return (status); }
  int getPriority() const { return priority; }
  void setPriority(int p);
  double getRemainingCpuBurstTime() const;
  char *getName() const { return (name); }
  int getID() const { return (ID); }
  // ......
};
```

4. code/threads/thread.h, class ThreadStatistics

全新的class，用以封裝執行緒的統計資訊，approxCpuBurstTime為公式中的t_i，runningTicks為T，preRunningTicks則是給除錯訊息E使用的T(進入waiting狀態時，仍存有先前的T資訊，不會立刻歸零)，remainingCpuBurstTime即預估剩餘的CPU burst時間(t_i - T)，但只會在離開Running狀態時更新，runningStartTime則是為了更新runningTicks(T)的所需資訊，而readyStartTime則是為了aging的必要資訊。

```
// Routines for managing statistics about thread
class ThreadStatistics
{
  friend class Thread;

private:
  // approximated CPU burst time (t_i)
  double approxCpuBurstTime;
  // The total running ticks within a CPU burst (T). This will be zero out
when thread switches to waiting.
  // When the thread switches to ready, this value will only stop accumulating
and will not be reset to zero.
  int runningTicks;
  // Same as runningTicks (T), but won't reset when the thread switches to
waiting. Just for debug messages.
  int preRunningTicks;
  // t_i - T. This value could be negative, but we don't need to handle it.
Note that the value is only
  // updated after the running state, during the running state this value is
not the real remaining estimated
  // cpu burst time.
  double remainingCpuBurstTime;
  int runningStartTime;
  void updateRemainingCpuBurstTime();
  void updateApproxCpuBurstTime(const int threadId, const char *threadName,
```

```
const int totalTicks);
  void endRunning(const int totalTicks);

public:
  int readyStartTime;
  ThreadStatistics() : approxCpuBurstTime(0.0), runningTicks(0),
preRunningTicks(0), remainingCpuBurstTime(0.0), runningStartTime(0),
readyStartTime(0){};
  int getRunningTicks() const { return runningTicks; }
  int getPreRunningTicks() const { return preRunningTicks; }
  int getRunningStartTime() const { return runningStartTime; }
};
```

5. code/threads/thread.cc, class Thread

建構式創建ThreadStatistics，使每個執行緒都有執行狀態的統計資訊。解構時則將其delete避免memory leak。

```
Thread::Thread(char *threadName, int threadID) : ts(new ThreadStatistics())
{
    // ......
    setStatus(JUST_CREATED, kernel->stats->totalTicks);
    // ......
}

Thread::~Thread()
{
    // ......
    if (space != NULL)
    {
        delete space;
    }
    delete ts;
}
```

Thread::setPriority在設定前檢查priority值是否在合法範圍內。

```
void Thread::setPriority(int p)
{
    ASSERT(PRI_L3_MIN <= p && p <= PRI_L1_MAX);
    priority = p;
}
```

Thread::setStatus封裝了執行緒狀態變化時的檢查和統計資訊的維護，從Running狀態離開時，呼叫[ThreadStatistics::endRunning](#)更新T，若進入Ready狀態，更新readyStartTime，若進入Ready狀態，更新runningStartTime，若進入Waiting狀態，調用[ThreadStatistics::updateApproxCpuBurstTime](#)更新t_i並重置T，最後更新執行緒狀態。

```
void Thread::setStatus(ThreadStatus st, const int totalTicks)
{
    DEBUG(dbgBeta, "Tick [" << totalTicks << "]: Thread [" << this->getID() <<
"] is " << this->getName() << ", "
                            << getStatusString(this->status) << " -> " <<
getStatusString(st));
    if (this->status == RUNNING && st != RUNNING)
    {
        this->ts->endRunning(totalTicks);
    }

    if (st == READY)
    {
        ASSERT(this->status == JUST_CREATED || this->status == BLOCKED ||
this->status == RUNNING);
        this->ts->readyStartTime = totalTicks;
    }
    else if (st == RUNNING)
    {
        ASSERT(this->status == READY);
        this->ts->runningStartTime = totalTicks;
    }
    else if (st == BLOCKED)
    {
        ASSERT(this->status == RUNNING);
        this->ts->updateApproxCpuBurstTime(this->getID(), this->getName(),
totalTicks);
    }
    else if (st == ZOMBIE)
    {
        ASSERT(this->status == RUNNING);
    }
    this->status = st;
}
```

Thread::Sleep對應Thread::setStatus進行調整，並且區分Waiting（即BLOCKED）和Terminated（即ZOMBIE）狀態。

```
void Thread::Sleep(bool finishing)
{
    // ......
    if (finishing)
    {
        setStatus(ZOMBIE, kernel->stats->totalTicks);
    }
    else
    {
        setStatus(BLOCKED, kernel->stats->totalTicks);
    }
    // ......
}
```

Thread::getStatusString轉換執行緒狀態的列舉為字串，方便除錯時印出使用。

```
const char *Thread::getStatusString(ThreadStatus st)
{
    switch (st)
    {
    case JUST_CREATED:
        return "JUST_CREATED";
    case RUNNING:
        return "RUNNING";
    case READY:
        return "READY";
    case BLOCKED:
        return "BLOCKED";
    case ZOMBIE:
        return "ZOMBIE";
    default:
        ASSERTNOTREACHED();
    }
}
```

由於實作使得t_i只在進入Waiting狀態時更新、T只在離開Running和進入Waiting狀態時更新，在執行緒Running時有部分T仍未結算，故Thread::getRemainingCpuBurstTime在執行緒Running時，必須將T未計入的部分扣除，得到的才是正確的預估剩餘CPU burst時間，此方法封裝上述實作細節，使Scheduler能直接取得正確的t_i - T。

```
double Thread::getRemainingCpuBurstTime() const
{
    if (this->status != RUNNING)
    {
```

```
        return this->ts->remainingCpuBurstTime;
    }
    // the ticks passed in this running state, but not yet counted
    int curRunningTicks = kernel->stats->totalTicks -
this->ts->runningStartTime;
    return this->ts->remainingCpuBurstTime - curRunningTicks;
}
```

6. code/threads/thread.cc, class ThreadStatistics

ThreadStatistics::endRunning會被每次執行緒離開Running狀態時被調用，將本次Running的時間加入T，更新除錯用的preRunningTicks後，呼叫updateRemainingCpuBurstTime更新t_i - T。

```
void ThreadStatistics::endRunning(const int totalTicks)
{
    // the ticks passed in this running state
    int curRunningTicks = totalTicks - runningStartTime;
    ASSERT(curRunningTicks >= 0);
    runningTicks += curRunningTicks;
    preRunningTicks = runningTicks;
    updateRemainingCpuBurstTime();
}
```

ThreadStatistics::updateApproxCpuBurstTime會在每次執行緒進入Waiting狀態時被調用，使用公式更新預估CPU burst時間(t_i)後，將T歸零，並且呼叫updateRemainingCpuBurstTime更新t_i - T。

```
void ThreadStatistics::updateApproxCpuBurstTime(const int threadId, const char
*threadName, const int totalTicks)
{
    double from = approxCpuBurstTime;
    approxCpuBurstTime = 0.5 * runningTicks + 0.5 * approxCpuBurstTime;
    // The formula implied by this debug message is different from the formula
in the spec,
    // but spec requires that debug messages be printed in this format.
    DEBUG(dbgBeta, "[D] Tick [" << totalTicks << "]: Thread [" << threadId <<
" " << threadName << "] update approximate burst time, from: ["
                                << from << "], add [" << runningTicks << "],
to [" << approxCpuBurstTime << "]");
    DEBUG(dbgTs, "[D] Tick [" << totalTicks << "]: Thread [" << threadId << "]
update approximate burst time, from: ["
                              << from << "], add [" << runningTicks << "], to
```

```
[" << approxCpuBurstTime << "]");
    runningTicks = 0;
    updateRemainingCpuBurstTime();
}

void ThreadStatistics::updateRemainingCpuBurstTime()
{
    remainingCpuBurstTime = approxCpuBurstTime - runningTicks;
}
```

7. code/threads/scheduler.h

新增執行緒priority範圍定義、排序L1與L2 ready queue所需的compare function。新增私有成員readyList1、readyList2、readyList3作為不同層級的ready queue。新增公開方法aging、isPreempted、shouldDoRoundRobin供Alarm::CallBack()調用。新增私有方法qLv、pushToQ、aging、popFromQMsg以支援內部ready queue的維護。

```
// ......
// thread priority
#define PRI_L3_MIN 0
#define PRI_L2_MIN 50
#define PRI_L1_MIN 100
#define PRI_L1_MAX 149

int CompareSjf(Thread *a, Thread *b);
int ComparePriority(Thread *a, Thread *b);

class Scheduler
{
public:
  // ......
  void aging(const int totalTicks);
  // if the currentThread is preempted
  bool isPreempted(const Thread *currentThread, const int totalTicks) const;
  // if the current L3 thread has run more than 100 ticks and L3 queue is not
empty, return true
  bool shouldDoRoundRobin(const Thread *currentThread, const int totalTicks)
const;

private:
  // SJF
  SortedList<Thread *> *readyList1;
```

```
  // non-preemptive
  SortedList<Thread *> *readyList2;
  // round-robin
  List<Thread *> *readyList3;
  // ......
  /**
   * @brief the corresponding ready queue level of this thread
   *
   * @param thread
   * @return int 1, 2 or 3
   */
  int qLv(const Thread *thread) const;
  void pushToQ(Thread *thread);
  void aging(const int totalTicks, List<Thread *> *readyList, const int
readyListLevel);
  void popFromQMsg(Thread *thread, const int q);
};
```

8. code/threads/scheduler.cc

建構式和解構式移除舊的readyList，並增加3個新ready queue的建立及移除，其中readyList1使用CompareSjf進行排序、readyList2使用ComparePriority進行排序。Scheduler::Print也改為印出3個ready queue。

```
Scheduler::Scheduler()
{
    readyList1 = new SortedList<Thread *>(CompareSjf);
    readyList2 = new SortedList<Thread *>(ComparePriority);
    readyList3 = new List<Thread *>;
    toBeDestroyed = NULL;
}

Scheduler::~Scheduler()
{
    delete readyList1;
    delete readyList2;
    delete readyList3;
}

void Scheduler::Print()
{
    cout << "Ready list contents:\n";
```

```
    readyList1->Apply(ThreadPrint);
    readyList2->Apply(ThreadPrint);
    readyList3->Apply(ThreadPrint);
}
```

Scheduler::ReadyToRun改用私有方法pushToQ將執行緒放入3個ready queue的其中一個。

```
void Scheduler::ReadyToRun(Thread *thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());

    thread->setStatus(READY, kernel->stats->totalTicks);
    pushToQ(thread);
}
```

Scheduler::FindNextToRun改為從優先度高的ready queue開始尋找可執行的執行緒，若有可執行的執行緒，使用私有方法popFromQMsg印出除錯訊息。

```
Thread *Scheduler::FindNextToRun()
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    Thread *next = NULL;
    int q;
    if (!readyList1->IsEmpty())
    {
        q = 1;
        next = readyList1->RemoveFront();
    }
    else if (!readyList2->IsEmpty())
    {
        q = 2;
        next = readyList2->RemoveFront();
    }
    else if (!readyList3->IsEmpty())
    {
        q = 3;
        next = readyList3->RemoveFront();
    }
    if (next != NULL)
    {
        popFromQMsg(next, q);
    }
    return next;
}
```

```
void Scheduler::popFromQMsg(Thread *thread, const int q)
{
    DEBUG(dbgBeta, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread ["
<< thread->getID() << " " << thread->getName() << "] is removed from queue L["
<< q << "]");
    DEBUG(dbgTs, "[B] Tick [" << kernel->stats->totalTicks << "]: Thread [" <<
thread->getID() << "] is removed from queue L[" << q << "]");
}
```

Scheduler::Run對應[Thread::setStatus]進行調整，並增加除錯訊息

```
void Scheduler::Run(Thread *nextThread, bool finishing)
{
    // ......
    nextThread->setStatus(RUNNING, kernel->stats->totalTicks); // nextThread
is now running

    DEBUG(dbgThread, "Switching from: " << oldThread->getName() << " to: " <<
nextThread->getName());
    DEBUG(dbgBeta, "[E] Tick [" << kernel->stats->totalTicks << "]: Thread ["
<< nextThread->getID() << " " << nextThread->getName()
                                << "] is now selected for execution, thread ["
<< oldThread->getID() << " " << oldThread->getName()
                                << "] is replaced, and it has executed [" <<
oldThread->ts->getPreRunningTicks() << "] ticks");
    DEBUG(dbgTs, "[E] Tick [" << kernel->stats->totalTicks << "]: Thread [" <<
nextThread->getID()
                              << "] is now selected for execution, thread ["
<< oldThread->getID()
                              << "] is replaced, and it has executed [" <<
oldThread->ts->getPreRunningTicks() << "] ticks");
    // ......
}
```

Scheduler::qLv根據執行緒的priority判斷該執行緒應該屬於哪個層級的ready queue

```
int Scheduler::qLv(const Thread *thread) const
{
    int p = thread->getPriority();
    if (PRI_L1_MIN <= p && p <= PRI_L1_MAX)
    {
        return 1;
    }
    else if (PRI_L2_MIN <= p)
```

```
    {
        return 2;
    }
    else
    {
        return 3;
    }
}
```
Scheduler::pushToQ將執行緒放入對應層級的ready queue中。
```
void Scheduler::pushToQ(Thread *thread)
{
    int q = qLv(thread);
    ASSERT(1 <= q && q <= 3);
    DEBUG(dbgBeta, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread ["
<< thread->getID() << " " << thread->getName() << "] is inserted into queue
L[" << q << "]");
    DEBUG(dbgTs, "[A] Tick [" << kernel->stats->totalTicks << "]: Thread [" <<
thread->getID() << "] is inserted into queue L[" << q << "]");
    if (q == 3)
    {
        readyList3->Append(thread);
    }
    else if (q == 2)
    {
        readyList2->Insert(thread);
    }
    else
    {
        readyList1->Insert(thread);
    }
}
```
公開方法Scheduler::aging調用私有方法aging分別對3個ready queue進行操作，完成操作後使用SanityCheck確保每個ready queue的狀態和排序正確。私有方法aging則遍歷傳入的ready queue，如果發現有執行緒維持在READY狀態超過1500 ticks，將其priority + 10（上限149），priority提升後，如果對應的ready queue層級發生變化或該執行緒屬於L2（此ready queue以priority排序），則將該執行緒移出原先的ready queue，再放回對應層級的ready queue。注意遍歷ready queue時，如果有執行緒需要從原先的ready queue移除，我們必須等到iterator前進之後再移除，否則原先iterator指向的對象已經被delete，取得其next指標可能會出錯。

```
void Scheduler::aging(const int totalTicks)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    aging(totalTicks, this->readyList1, 1);
    aging(totalTicks, this->readyList2, 2);
    aging(totalTicks, this->readyList3, 3);
    // check the implementation of aging
    // Print();
    readyList1->SanityCheck();
    readyList2->SanityCheck();
    readyList3->SanityCheck();
}

void Scheduler::aging(const int totalTicks, List<Thread *> *readyList, const
int readyListLevel)
{
    ListIterator<Thread *> *it = new ListIterator<Thread *>(readyList);
    Thread *to_delete, *t;
    int waitingTime, oldPriority, newPriority, q;
    while (true)
    {
        if (to_delete) // delay the deletion to avoid null pointer error
        {
            readyList->Remove(to_delete);
            popFromQMsg(to_delete, readyListLevel);
            pushToQ(to_delete);
            to_delete = NULL;
        }
        if (it->IsDone())
        {
            break;
        }
        t = it->Item();
        waitingTime = totalTicks - t->ts->readyStartTime;
        q = readyListLevel;
        if (waitingTime > 1500)
        {
            oldPriority = t->getPriority();
            newPriority = min(oldPriority + 10, PRI_L1_MAX);
            if (oldPriority != newPriority)
            {
                t->setPriority(newPriority);
```

```
                t->ts->readyStartTime = totalTicks; // reset the waiting time
                DEBUG(dbgBeta, "[C] Tick [" << totalTicks << "]: Thread [" <<
t->getID() << " " << t->getName() << "] changes its priority from [" <<
oldPriority << "] to [" << newPriority << "]");
                DEBUG(dbgTs, "[C] Tick [" << totalTicks << "]: Thread [" <<
t->getID() << "] changes its priority from [" << oldPriority << "] to [" <<
newPriority << "]");
                if (qLv(t) != readyListLevel || readyListLevel == 2)
                { // Remove and append the thread again to make sure the
ordering is correct. If a thread has elevated priority in L2,
                    // we must reorder the thread, since L2 is sorted by
priority.
                    to_delete = t;
                }
            }
        }
        it->Next();
    }
    delete it;
}
```

Scheduler::isPreempted判斷目前的執行緒是否會被其他執行緒搶占，如果目前的執行緒不在Running，則不可能被其他執行緒搶占，如果目前的執行緒在Running且有其他對應ready queue層級較高的執行緒正在ready，則目前的執行緒被搶占，又或者目前的執行緒屬於L1層級且L1中有其他預估剩餘的CPU burst時間較低的執行緒，也會發生搶占。

```
bool Scheduler::isPreempted(const Thread *currentThread, const int totalTicks)
const
{
    if (currentThread->getStatus() != RUNNING)
    {
        return FALSE;
    }
    int q = qLv(currentThread);
    if (q == 3 && (!readyList1->IsEmpty() || !readyList2->IsEmpty()))
    {
        // L1 or L2 preempts L3
        DEBUG(dbgBeta, "Tick " << totalTicks << " current L3 thread " <<
currentThread->getID() << " " << currentThread->getName()
                               << " is preempted by other L1/L2 thread");
        return TRUE;
    }
```

```
    else if (q == 2 && !readyList1->IsEmpty())
    { // L1 preempts L2
        DEBUG(dbgBeta, "Tick " << totalTicks << " current L2 thread " <<
currentThread->getID() << " " << currentThread->getName()
                              << " is preempted by other L1 thread");
        return TRUE;
    }
    else if (!readyList1->IsEmpty() &&
readyList1->Front()->getRemainingCpuBurstTime() <
currentThread->getRemainingCpuBurstTime())
    { // L1 preempts L1
        DEBUG(dbgBeta, "Tick " << totalTicks << " current L1 thread " <<
currentThread->getID() << " " << currentThread->getName()
                              << " (remaining " <<
currentThread->getRemainingCpuBurstTime() << ")"
                              << " is preempted by other L1 thread "
                              << readyList1->Front()->getID() << " " <<
readyList1->Front()->getName()
                              << " (remaining " <<
readyList1->Front()->getRemainingCpuBurstTime() << ")");
        return TRUE;
    }
    return FALSE;
}
```

Scheduler::shouldDoRoundRobin判斷目前正在執行的L3層級執行緒是否該yield給其他L3層級執行緒，條件為目前的L3層級執行緒維持Running狀態100 ticks以上。

```
bool Scheduler::shouldDoRoundRobin(const Thread *currentThread, const int
totalTicks) const
{
    if (currentThread->getPriority() < PRI_L2_MIN && !readyList3->IsEmpty())
    {
        // the ticks passed in this running state
        int curRunningTicks = totalTicks -
currentThread->ts->getRunningStartTime();
        ASSERT(curRunningTicks >= 0);
        if (curRunningTicks >= 100)
        {
            DEBUG(dbgBeta, "current L3 thread " << currentThread->getID() << "
" << currentThread->getName()
                                              << " has run " <<
curRunningTicks << " ticks, yields on return");
```

```
            return TRUE;
        }
    }
    return FALSE;
}
```

CompareSjf和ComparePriority判斷兩執行緒的排序順序，兩者相同時回傳0，如果回傳負數，則a排在b前面，反之則b排在a前面。

```
int CompareSjf(Thread *a, Thread *b)
{
    double ra = a->getRemainingCpuBurstTime();
    double rb = b->getRemainingCpuBurstTime();
    if (ra < rb)
    {
        return -1;
    }
    else if (ra == rb)
    {
        return 0;
    }
    else
    {
        return 1;
    }
}

int ComparePriority(Thread *a, Thread *b)
{
    return b->getPriority() - a->getPriority();
}
```

9. code/threads/alarm.cc

如果目前Nachos在IdleMode，表示沒有任何執行緒可執行，不需要進行aging和preemption的判斷，反之如果有任何執行緒在ready狀態，呼叫Scheduler::aging調整在ready狀態等待超過1500 ticks的執行緒priority，並在aging後調用Scheduler::isPreempted和Scheduler::shouldDoRoundRobin判斷目前的執行緒是否要yield給其他執行緒。

```
void Alarm::CallBack()
{
    Interrupt *interrupt = kernel->interrupt;
    MachineStatus status = interrupt->getStatus();
```

```
    // No thread to run. Hence we don't need to check aging and preemption
    if (status == IdleMode)
    {
        return;
    }
    Scheduler *scheduler = kernel->scheduler;
    Thread *currentThread = kernel->currentThread;
    scheduler->aging(kernel->stats->totalTicks);
    // after aging, some threads may preempt the current thread
    if (scheduler->isPreempted(currentThread, kernel->stats->totalTicks))
    {
        interrupt->YieldOnReturn();
    }
    else if (scheduler->shouldDoRoundRobin(currentThread,
kernel->stats->totalTicks))
    {
        interrupt->YieldOnReturn();
    }
}
```

10. Implementation summary

以上執行緒狀態切換的流程可概括為下圖: