CS342301: Operating System
MP3: CPU Scheduling
Team Number: 71

Team Members & Contributions

- Members: 108022138 楊宗諺、111062649 許峻源
- Contributions: 合力製作MP3 report、討論如何實做multilevel feedback queue scheduler 與 debugging message

Trace Codes

New -> Ready

1. **Kernel::ExecAll()**

- 在設置好kernel之後，就會呼叫ExecAll()來執行user program。
- ExecAll()會根據先前設置好的execfile陣列呼叫Exec()函數來執行各個file，為每個program創建一個thread。

2. **Kernel::Exec()**

- Exec()主要在幫每個要執行的user program創建一個thread，並透過thread的constructor初始化thread的相關參數（stack、thread status等等）。另外也會透過AddrSpace::AddrSpace()來創建thread的address space。
- 接著呼叫Thread::Fork()函數載入真正要執行的程式碼，這邊把ForkExecute()的function pointer與thread的pointer當作參數傳入。

3. **Thread::Fork()**

- Fork()會透過Thread::StackAllocate()函數來allocate thread的stack，這邊會把參數function pointer跟thread的pointer傳入StackAllocate()裡面，為了之後呼叫Switch()函數時可以正確執行function pointer指向的function。
- 在Allocate好stack之後，會呼叫Scheduler::ReadyToRun()函數把thread丟進ready queue裡面等待執行。（Note: 呼叫Scheduler之前會先disable所有的interrupt、被丟進ready queue裡面的thread還沒有load好memory）

4. **Thread::StackAllocate()**

- 根據定義好的StackSize為thread創建一個bounded array for stack，並用stackTop pointer來追蹤當前stack的top的位置。
- 初始化且設值所有需要的CPU registers（machine state），這邊會把function pointer跟thread的pointer設給CPU的registers。

5. **Scheduler::ReadyToRun()**

- 將thread的status從just_created設置成ready，並把創建好的thread丟進ready queue裡面等待執行。

Running → Ready

1. **Machine::Run()**

- 被AddrSpace::Execute()呼叫, current thread 開始執行User program
- 執行User program的時候需要在user mode下執行，Run()函數中的setStatus(UserMode)可以將處理器的status轉為user mode。
- 物件Instruction指標instr可以用來儲存instruction的相關資訊，instr會被當作參數傳入Machine::OneInstruction()函數中。

2. **Interrupt::OneTick()**

- OneTick()函數處理Context switch的情況, 如果Alarm::CallBack() 設定YieldOnReturn為true, 則呼叫Yield()進行Context switch。

3. **Thread::Yield()**

- 先設定interrupt關閉, 執行FindNextToRun() 尋找下一個可執行的threads
- 如果找到nextthreads, 呼叫scheduler::ReadyToRun() 把其放入ready queue中, 再呼叫scheduler::Run()進行context switch, 開始執行此threads
- 重新開啟interrupt

4. **Scheduler::FindNextToRun()**

- 只要ready queue不為空, 就從queue中pop出一個thread 作為下一個要執行的thread, 並回傳此thread

5. **Scheduler::ReadyToRun()**

- 把thread 的狀態設為ready, 並把其加入ready queue中

6. **Scheduler::Run()**

- Run()函數會利用switch()函數將CPU的使用權從old thread切換到next thread，而在switch之前會先儲存old thread的register states跟address space並且check old thread的stack有沒有overflow，然後把next thread的status設成running。
- 而從next thread switch回來之後，會先利用CheckToBeDestroyed()函數check說old thread是否已經執行完要被delete掉。如果仍要繼續執行old thread的話，就會restore先前儲存的register states和address space。

Running → Waiting

1. **SynchConsoleOutput::PutChar()**

- PutChar()一開始會先利用class Lock中所定義的Acquire()函數先去取得writer, 如果沒有閒置的writer的話則需要等待。
- 取得writer之後會去呼叫console.cc中的ConsoleOutput::PutChar()函數。
- ConsoleOutput::PutChar()函數會登記一個interrupt。此interrupt會在print完要print的character之後呼叫登記此interrupt物件的CallBack()函數。
- waitFor->P()函數會用來等待前一個要print的character print完之後再繼續執行, 在Semaphore::P()函數的內容中也可以發現有一個while loop在重複執行直到Semaphore是available的狀態。
- 從waitFor->P()函數回來之後, 代表說print的工作完成了，接下來會利用class Lock中定義的Release()函數把lock住的writer釋放掉, 讓其他需要writer的thread可以來使用。

2. **Semaphore::P()**

- 如果value值等於0時, 就把currentThread放入waiting queue中, 呼叫sleep()把thread 的狀態變為block, 並找到nextThread進行執行, value值減1

3. **List::Append(T)**

- 把Thread 加入 由List構成的queue中, 首先為Thread創建list element, 把last element 的指摽指向新的element, 如果list 為空, 則新的element 同時設為list 的first and last element, numInList 加1

4. **Thread::Sleep()**

- Sleep()函數會把current thread的status從running設成blocked, 並利用Scheduler::FindNextToRun()函數去ready queue裡面找下一個要被執行的thread。
- 如果ready queue裡面沒有任何thread等待被執行, 那麼就會呼叫Interrupt::Idle()函數將系統status設為IdleMode並檢查有沒有interrupt要執行，沒有的話就會呼叫Halt()函數。反之, 如果有thread要被執行的話, 就會呼叫Scheduler::Run()函數來執行。

5. **Scheduler::FindNextToRun()**

- 因為currentThread 狀態為blocked, 所以要找到一個thread 來執行, 只要ready queue不為空, 就從queue中pop出一個thread 作為下一個要執行的thread, 並回傳此thread

6. **Scheduler::Run()**

- Run()函數會利用switch()函數將CPU的使用權從old thread切換到next thread, 而在switch之前會先儲存old thread的register states跟address space並且check old thread的stack有沒有overflow, 然後把next thread的status設成running。

- 而從next thread switch回來之後，會先利用CheckToBeDestroyed()函數check說old thread是否已經執行完要被delete掉。如果仍要繼續執行old thread的話，就會restore先前儲存的register states和address space。

Waiting → Ready

1. **Semaphore::V()**

- I/O執行完畢, 把queue的最前端的thread 從List中pop出來, 呼叫ReadyToRun() 把thread 的狀態改成ready, 放入ready queue中

2. **Scheduler::ReadyToRun(Thread*)**

- 把thread 的狀態設為ready, 並把其加入ready queue中

Running→Terminated

1. **ExceptionHandler(ExceptionType) case SC_Exit**

- 在case SC_Exit 中, currentThread會呼叫 Finish() function, 結束 currentThread

2. **Thread::Finish()**

- Finish()函數會先disable所有的interrupt，接著呼叫Thread::Sleep()函數（參數傳入True）

3. **Thread::Sleep(bool)**

- Sleep()函數會把current thread的status從running設成blocked，並利用Scheduler::FindNextToRun()函數去ready queue裡面找下一個要被執行的thread。
- 如果ready queue裡面沒有任何thread等待被執行，那麼就會呼叫Interrupt::Idle()函數將系統status設為IdleMode並檢查有沒有interrupt要執行，沒有的話就會呼叫Halt()函數。反之，如果有thread要被執行的話，就會呼叫Scheduler::Run()函數來執行(finishing參數傳入true)。

4. **Scheduler::FindNextToRun()**

- 因為currentThread 準備要被terminated, 所以要找下一個要執行的thread, 只要ready queue不為空, 就從queue中pop出一個thread 作為下一個要執行的thread, 並回傳此thread

5. **Scheduler::Run(Thread*, bool)**

- 這邊的Run() 除了進行context switch 外, 還要將oldthread destroy, 如果finishing 為true, 則設oldthread 為toBeDestroyed 物件, 呼叫CheckToBeDestroy() 函數來delete 被設定要destroy 的thread

Ready→Running

1. **Scheduler::FindNextToRun()**

- 從ready queue 中, 找到下一個要執行的thread

2. **Scheduler::Run(Thread*, bool)**

- Run()函數會利用switch()函數將CPU的使用權從old thread切換到next thread, 而在switch之前會先儲存old thread的register states跟address space並且check old thread的stack有沒有overflow, 然後把next thread的status設成running。
- 而從next thread switch回來之後, 會先利用CheckToBeDestroyed()函數check說old thread是否已經執行完要被delete掉。如果仍要繼續執行old thread的話, 就會restore先前儲存的register states和address space。

3. **SWITCH(Thread*, Thread*)**

- 在switch() 中主要執行的是將CPU Registers中的data 儲存回oldThread 的 machineState[]中, 再將newThread 的machineState[] load 到CPU Registers內
- 因為不同的processor architecture不同, 為了讓switch() 能在不同的hardware中執行, switch()根據不同的hardware 做了個別的定義, 其中都包含了兩種routine: ThreadRoot, SWITCH
- 在ThreadRoot 中, 先把frame pointer 歸0 (stack 的原點), jump 到StartUpPc的位置, 開始執行startup procedure (執行checkToBeDestroy(), 並enable interrupt), 接著把 InitialArg存入 a0 register 中, 之後 jump 到 InitialPC, 也就是要被執行的function 的位置, 直到function 執行完畢後, 再跳到 whendonePC 執行結束程序的procedure (呼叫 Thread::sleep(finishing) finishing 設為true)
- 在SWITCH中, 把CPU Register中的值store回OldThread 的 MachineState[]中 , 並且把 NewThread 的 MachineState[] load 到CPU Register內, 在最後jump 到儲存NewThread 上次執行到的address, 也就是儲存在ra register 內的address

4. **(depends on the previous process state, e.g., [New,Running,Waiting]→Ready)**

- process 會先被ReadyToRun() 放入Ready queue 中, 直到被FindNextToRun() 移出ready queue 執行

5. **for loop in Machine::Run()**

- 在machine:: Run()中, 會執行OneInstruction() 一行一行執行userprog的assembly language, 直到halt() 被執行

Implementation Details

1. Files Modified

- NachOS-4.0_MP3/code/threads/alarm.cc
- NachOS-4.0_MP3/code/threads/kernel.cc
- NachOS-4.0_MP3/code/threads/kernel.h
- NachOS-4.0_MP3/code/threads/scheduler.cc
- NachOS-4.0_MP3/code/threads/scheduler.h
- NachOS-4.0_MP3/code/threads/thread.cc
- NachOS-4.0_MP3/code/threads/thread.h

2. Modification of Thread

- **Thread.h**
  因為thread多了priority且此次實作的目標為multilevel feedback queue, 所以在TCB裡我們多加了幾個相關參數:

  ➔ **priority** : thread 的priority
  ➔ **burstTime** : 預估的CPU burst time (t)
  ➔ **actualburstTime** : thread 實際執行的burst time (T)
  ➔ **waitingTime** : thread 在ready queue內等待的時間
  ➔ **startTime** : thread 剛進入CPU的時間
  ➔ **startWaitingTime** : thread 剛進入ready queue內的時間

  上述參數均設有對應的get跟set函數, 方便我們取用和更新參數的值。

- **Thread:: Thread()**
  有了新的參數, 我們也在constructor中新增了新參數的初始值。此外，我們也利用constructor overloading的方式新增了一個新的thread constructor, 此constructor提供了thread priority的argument, 用來傳入priority且初始化priority的值。

```
Thread::Thread(char* threadName, int threadID) {
    ID = threadID;
    name = threadName;
    priority = 70;
    burstTime = 0;
    actualBurstTime = 0;
    waitingTime = 0;
    startTime = 0;
    startWaitingTime = 0;
    stackTop = NULL;
    stack = NULL;
    status = JUST_CREATED;
    for (int i = 0; i < MachineStateSize; i++) {
    machineState[i] = NULL;     // not strictly necessary, since
                    // new thread ignores contents
                    // of machine registers
    }
    space = NULL;
}
```

```
Thread::Thread(char* threadName, int threadID, int threadPriority)
{
    ID = threadID;
    name = threadName;
    priority = threadPriority;
    burstTime = 0;
    actualBurstTime = 0;
    waitingTime = 0;
    startTime = 0;
    startWaitingTime = 0;
    stackTop = NULL;
    stack = NULL;
    status = JUST_CREATED;
    for (int i = 0; i < MachineStateSize; i++) {
    machineState[i] = NULL;     // not strictly necessary, since
                    // new thread ignores contents
                    // of machine registers
    }
    space = NULL;
}
```

- **Threads:: UpdateBurstTime(bool check) — newly added function**
  因為此次作業要求L1 queue內部需實作preemptive SJF algorithm，所以此處我們新增了UpdateBurstTime函數，用來計算approximate burst time (t) 與actual burst time (T) 。在thread的life cycle裡有兩個時間點會呼叫到此函數，這邊我們根據這兩個時間點做了不同的實作內容。

  首先，第一個呼叫此函數的時間點會是thread從running state轉換到waiting state或terminated的時候 (會呼叫Sleep()函數)，此時UpdateBurstTime()函數會更新actual burst time (T，利用當前total ticks去減掉記錄到的start time) 和更新approximate burst time (t，根據spec上的公式)。**Note**: 計算完後會將actual burst time reset為0。

  第二個呼叫此函數的時間點會是thread在running state被timer打斷的時候，這邊UpdateBurstTime()函數會做的事情主要是計算approximate remaining burst time和更新actual burst time (T)。**Note**: 這邊會重新reset一次start time。

  除此之外，因為呼叫此函數的時間點有兩個，而不同的時間點有不同的實作內容。因此我們在argument的部分加入了check參數，用來判別說在哪個時間點呼叫了UpdateBurstTime()函數。

```
290 // check: 0 for yield, 1 for sleep.
291 void Thread::UpdateBurstTime(bool check) {
292     int currentTime = kernel->stats->totalTicks;
293     burstTime -= (currentTime - startTime);
294     actualBurstTime += (currentTime - startTime);
295     //TQ -= (currentTime - startTime);
296     if(check) {
297         double newBurstTime = (burstTime + 2 * actualBurstTime) / 2;
298         DEBUG(dbgSchedule, "Tick [" << kernel->stats->totalTicks << "]
299         burstTime = newBurstTime;
300     } else {
301         startTime = currentTime;
302     }
303 }
```

- **Thread:: Yield()**
  Yield()函數會在current thread確認要被context switch的時候被呼叫 (細節會在alarm的callback函數中解釋更清楚)。此函數的主要功能是把current thread丟到ready queue中，並從ready queue中找出下一個要被執行的thread，呼叫Scheduler::Run()函數執行context switch。

```
227 void
228 Thread::Yield ()
229 {
230     Thread *nextThread;
231     IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);
232
233     ASSERT(this == kernel->currentThread);
234
235     DEBUG(dbgThread, "Yielding thread: " << name);
236
237     kernel->scheduler->ReadyToRun(this);
238     nextThread = kernel->scheduler->FindNextToRun();
239     if (nextThread != NULL) {
240     DEBUG(dbgSchedule, "Tick [" << kernel->stats->totalTicks << "]: Thre
241     kernel->scheduler->Run(nextThread, FALSE);
242     }
243     (void) kernel->interrupt->SetLevel(oldLevel);
244 }
```

- **Thread:: Sleep()**
  Sleep()函數會在current thread要執行I/O、wait或terminated的時候被呼叫，在此函數中我們新增了剛剛提及的UpdateBurstTime()函數，用以計算actual burst time (T) 和更新approximate burst time (t)。

```
266 void
267 Thread::Sleep (bool finishing)
268 {
269     Thread *nextThread;
270
271     ASSERT(this == kernel->currentThread);
272     ASSERT(kernel->interrupt->getLevel() == IntOff);
273
274     DEBUG(dbgThread, "Sleeping thread: " << name);
275     DEBUG(dbgTraCode, "In Thread::Sleep, Sleeping thread: " << name << ", " << ke
276
277     status = BLOCKED;
278     UpdateBurstTime(true);
279     //cout << "debug Thread::Sleep " << name << "wait for Idle\n";
280     while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL) {
281         kernel->interrupt->Idle();  // no one to run, wait for an interrupt
282     }
283     // returns when it's time for us to run
284     DEBUG(dbgselfdef, "Thread ["<< getID()<< "] is blocked"<<endl);
285     DEBUG(dbgSchedule, "Tick [" << kernel->stats->totalTicks << "]: Thread [" <<
286     actualBurstTime = 0;
287     kernel->scheduler->Run(nextThread, finishing);
288 }
```

3. Modification of Ready Queue

- **Scheduler::Scheduler() , Scheduler::~Scheduler()**
  因為此次實作的目標為multilevel feedback queue，所以我們在scheduler裡面新增了三個sorted list，分別代表L1、L2和L3 queue，並移除原先的ready list。同時，我們也相對應修改了destructor。

```
122 Scheduler::Scheduler()
123 {
124     //readyList = new List<Thread *>;
125     L1 = new SortedList<Thread*>(SJF_cmp);
126     L2 = new SortedList<Thread*>(Priority_cmp);
127     L3 = new SortedList<Thread*>(RR_cmp);
128     toBeDestroyed = NULL;
129 }
```

```
136 Scheduler::~Scheduler()
137 {
138     //delete readyList;
139     delete L1;
140     delete L2;
141     delete L3;
142 }
```

此外，我們也定義了三個sorted list的compare function，分別對應到三個queue中使用的不同的scheduling algorithm。

- ➔ **L1(SJF_cmp)** : 用approximate remaining burst time排序
- ➔ **L2(Priority_cmp)** : 用thread priority排序
- ➔ **L3(RR_cmp)** : 用進入queue的順序排序

```
30 int SJF_cmp(Thread* t1, Thread* t2) {
31     double bt1=t1->getBurstTime(), bt2=t2->getBurstTime();
32     if(bt1==bt2)
33         return 0;
34     return bt1<bt2?-1:1;
35 }
36
37 int Priority_cmp(Thread* t1, Thread* t2) {
38     int p1=t1->getPriority(), p2=t2->getPriority();
39     if(p1==p2)
40         return 0;
41     return p1>p2?-1:1;
42 }
43
44 int RR_cmp(Thread* t1, Thread* t2) {
45     return 1;
46 }
```

- **Scheduler::ReadyToRun()**
  我們在此函數上做了些小調整，根據thread的priority來決定說thread該放進哪個level的ready queue中。此外，這邊我們也reset了一次start waiting time，因為呼叫此函數的時間點會是有thread要進入ready queue的時候。

```
152 void
153 Scheduler::ReadyToRun (Thread *thread)
154 {
155     ASSERT(kernel->interrupt->getLevel() == IntOff);
156     DEBUG(dbgThread, "Putting thread on ready list: " << thread->getName());
157     //cout << "Putting thread on ready list: " << thread->getName() << endl ;
158     thread->setStatus(READY);
159     thread->setStartWaitingTime(kernel->stats->totalTicks);
160     //readyList->Append(thread);
161     int p=thread->getPriority();
162     if(p>=0&&p<=49) {
163         L3->Append(thread);
164         DEBUG(dbgSchedule, "Tick [" << kernel->stats->totalTicks << "]: Thread [
165     }
166     else if(p>=50&&p<=99) {
167         L2->Append(thread);
168         DEBUG(dbgSchedule, "Tick [" << kernel->stats->totalTicks << "]: Thread [
169     }
170     else if(p>=100&&p<=149) {
171         L1->Append(thread);
172         DEBUG(dbgSchedule, "Tick [" << kernel->stats->totalTicks << "]: Thread [
173     }
174     // CheckIfPreempt()
175 }
```

- **Scheduler::FindNextToRun()**
  因為此次實作為multilevel feedback queue，此函數會從三個ready queue中挑一個thread出來執行。然而又以L1 queue的priority為最高，L3 queue為最低，所以在實作上，我們新增了幾個if-else判斷式來判斷說當前應該取用哪個ready queue中的thread(按照queue的priority)。

```
185 Thread *
186 Scheduler::FindNextToRun ()
187 {
188     ASSERT(kernel->interrupt->getLevel() == IntOff);
189 
190     if(L1->IsEmpty()==false) {
191         DEBUG(dbgSchedule, "Tick [" << kernel->stats->t
192         return L1->RemoveFront();
193     }
194     else if(L2->IsEmpty()==false) {
195         DEBUG(dbgSchedule, "Tick [" << kernel->stats->t
196         return L2->RemoveFront();
197     }
198     else if(L3->IsEmpty()==false) {
199         DEBUG(dbgSchedule, "Tick [" << kernel->stats->t
200         return L3->RemoveFront();
201     }
202     else
203         return NULL;
204 }
```

- **Scheduler::Run()**
  因為呼叫此函數的時間點會是有thread要進到CPU裡執行的時候，所以我們在此新增了reset start time和waiting time的statements。**Note**: reset start time為當前total ticks，reset waiting time為0。

```
244     kernel->currentThread = nextThread;  // switc
245     nextThread->setStatus(RUNNING);      // nextT
246     nextThread->setStartTime(stats->totalTicks);
247     nextThread->setWaitingTime(0);
```

4. Implementation of Aging and Preemption

- **Alarm::CallBack()**
  根據spec，所有preemption和priority update都必須delay到下一個timer interrupt發生的時候。因此，這邊我們把preemption的檢查和priority update都放到alarm的callback函數裡面。**Note**: priority update之前會先更新ready queue中所有thread的waiting time，再根據waiting time來判斷說是否要增加priority，而這些operation都會在Scheduler::Aging()函數中被執行。

```
46 void
47 Alarm::CallBack()
48 {
49     Interrupt *interrupt = kernel->interrupt;
50     MachineStatus status = interrupt->getStatus();
51     Scheduler* scheduler = kernel->scheduler;
52
53     scheduler->Aging();
54     if (status != IdleMode) {
55         kernel->currentThread->UpdateBurstTime(false);
56         if(scheduler->CheckIfPreempt())
57             interrupt->YieldOnReturn();
58     }
59 }
```

  簡單說明一下此函數的運行方式，此函數首先會透過Aging()函數更新ready queue中所有thread的waiting time且判斷是否要update priority。再來，此函數會檢查說current thread是否會被preempt，如果確認會被preempt，則呼叫YieldOnReturn()函數順著下去進行context switch。然而如果current thread不會被preempt，則直接return，不進行context switch。**Note**: 在檢查preemption前需要先更新current thread的approximate remaining burst time，因為current thread可能會需要跟L1 queue中的thread作比較，所以需要先更新。

- **Scheduler::Aging() — newly added function**
  為了實做aging機制，我們在scheduler裡面新增了Aging()函數，用來更新ready queue中所有thread的waiting time與priority。這邊我們使用了list中定義的Apply()函數來對ready queue中的每個thread作檢查 (傳入UpdateWaitingTime()函數)。

```
88 void Scheduler::Aging() {
89     L1->Apply(UpdateWaitingTime);
90     L2->Apply(UpdateWaitingTime);
91     L3->Apply(UpdateWaitingTime);
92 }
```

- **UpdateWaitingTime() — newly added function**
  此函數的主要功能是更新ready queue中thread的waiting time，並對更新後waiting time超過1500 ticks的thread作priority update。而如果在update priority後，造成了thread不屬於當前queue的情況，此時會呼叫Jump()函數來進行thread在queue之間的移動 (升到priority較高的queue)。**Note**: 不管有沒有update priority，都會reset一次start waiting time，因為計算的方便。

```
71 void UpdateWaitingTime(Thread* t) {
72     int currentTime = kernel->stats->totalTicks;
73     t->setWaitingTime(t->getWaitingTime()+currentTime-t->getStartWaitingTime());
74     int waitingTime = t->getWaitingTime();
75     if(waitingTime>=1500) {
76         int p=t->getPriority();
77         if(p+10<=149)
78             t->setPriority(p+10);
79         else
80             t->setPriority(149);
81         DEBUG(dbgSchedule, "Tick [" << kernel->stats->totalTicks << "]: Thread [" <<
82         t->setWaitingTime(waitingTime-1500);
83         Jump(t);
84     }
85     t->setStartWaitingTime(currentTime);
86 }
```

- **Jump() — newly added function**
  我們新增此函數來檢查是否有thread在update priority後不屬於當前queue的情況。如果有則進行queue之間的移動。在此函數中，我們分別處理了L3->L2、L2->L1的情況。**Note**: L2中的thread在update priority後如果還在L2中的話，必須要重新insert進去L2，重新排序一次，才會維持sorted的情況。

```
void Jump(Thread* t) {
    Scheduler* scheduler = kernel->scheduler;
    int p=t->getPriority(), op=p-10;
    if(op>=50&&op<=99) {
        if(p>=100&&p<=149) {
            scheduler->L2->Remove(t);
            DEBUG(dbgSchedule, "Tick [" << kernel->stats->totalTicks << "]: Thread [" << t->getID() << "] is removed from queue L[2]");
            scheduler->L1->Append(t);
            DEBUG(dbgSchedule, "Tick [" << kernel->stats->totalTicks << "]: Thread [" << t->getID() << "] is inserted into queue L[1]");
        } else {
            scheduler->L2->Remove(t);
            scheduler->L2->Append(t);
        }
    } else if(op>=0&&op<=49) {
        if(p>=50&&p<=99) {
            scheduler->L3->Remove(t);
            DEBUG(dbgSchedule, "Tick [" << kernel->stats->totalTicks << "]: Thread [" << t->getID() << "] is removed from queue L[3]");
            scheduler->L2->Append(t);
            DEBUG(dbgSchedule, "Tick [" << kernel->stats->totalTicks << "]: Thread [" << t->getID() << "] is inserted into queue L[2]");
        }
    }
}
```

- **Scheduler::CheckIfPreempt() — newly added function**
  新增此函數來檢查說current thread是否應該被preempt，根據current thread原先的所屬queue來判斷說目前ready queue中有沒有人可以preempt它。主要分為以下情況：

  Current thread原先屬於L1 queue：
  在此情況下，只有L1 queue中有thread的approximate remaining burst time小於current thread的approximate remaining burst time才會preempt current thread (因為queue priority跟preemptive SJF algorithm的緣故)。否則，current thread應該繼續執行。

  Current thread原先屬於L2 queue：
  在此情況下，只有L1 queue中有thread才會preempt current thread (因為queue priority)。L2中的thread不會打斷current thread，因為L2的scheduling algorithm為non preemptive。

  Current thread原先屬於L3 queue：
  在此情況下，L1、L2和L3 queue中有thread都會preempt current thread。如果ready queue中都沒有thread的話，current thread可以繼續執行。

```
bool Scheduler::CheckIfPreempt() {
    Thread* currentThread = kernel->currentThread;
    int p=currentThread->getPriority();
    if(p>=0&&p<=49) {
        if(L1->IsEmpty()==false || L2->IsEmpty()==false || L3->IsEmpty()==false)
            return true;
        else
            return false;
    } else if(p>=50&&p<=99) {
        if(L1->IsEmpty()==false)
            return true;
        else
            return false;
    } else if(p>=100&&p<=149) {
        if(L1->IsEmpty()==false && L1->Front()->getBurstTime()<currentThread->getBurstTime())
            return true;
        else
            return false;
    }
    return false;
}
```

5. Add Command "-ep"

- **Kernel::Kernel()**
  因為新增了一個command，所以在kernel的constructor中，我們多加入了-ep的if-else判斷。

```
57          } else if (strcmp(argv[i], "-ep") == 0) {
58              execfile[++execfileNum] = argv[++i];
59              threadPriority[execfileNum] = atoi(argv[++i]);
60              cout << execfile[execfileNum] << "\n";
61              cout << threadPriority[execfileNum] << "\n";
```

- **Kernel.h**
  因為-ep command會多輸入thread的priority，所以除了原先的execfile以外，我們新增了一個threadPriority陣列，同時記錄-ep command中指定的thread priority。

```
81      char* execfile[10];
82      int threadPriority[10];
```

- **Kernel::ExecAll()**
  在呼叫Exec()函數的時候，我們多傳入了thread的priority作為參數。

```
271 void Kernel::ExecAll()
272 {
273     for (int i=1;i<=execfileNum;i++) {
274         int a = Exec(execfile[i], threadPriority[i]);
275     }
276     currentThread->Finish();
277     //Kernel::Exec();
278 }
```

- **Kernel::Exec()**
  改為呼叫新建立的thread constructor且傳入thread的priority。

```
281 int Kernel::Exec(char* name, int threadPriority)
282 {
283     t[threadNum] = new Thread(name, threadNum, threadPriority);
284     t[threadNum]->space = new AddrSpace();
285     t[threadNum]->Fork((VoidFunctionPtr) &ForkExecute, (void *)t[threadNum]);
286     threadNum++;
287
288     return threadNum-1;
```

6. Add Debug Messages

- **Debug.h**
  根據spec上的要求，我們在debug.h中新增了debugging flag 'z'。

```
33 const char dbgSchedule = 'z';
```

- **Location of Debugging Messages**
  因為debug message的statements多數都很長，所以這邊就不附圖。我們主要將debug messages放在以下地方：

  ➔ **Scheduler::ReadyToRun()**：此函數會把thread丟進ready queue中，所以這邊有加入thread insertion的debug message。
  ➔ **Scheduler::FindNextToRun()**：此處有加入thread removal的debug message。
  ➔ **UpdateWaitingTime()**：此處有加入priority update的debug message。
  ➔ **Jump()**：此處有加入thread insertion和removal的debug message。
  ➔ **Thread::Sleep()**：此處有新增context switch的debug message。
  ➔ **Thread::Yield()**：此處有新增context switch的debug message。
  ➔ **Thread::UpdateBurstTime()**：此處有加入approximate burst time update的debug message。