

线程调度实习报告

姓名 张煌昭 学号 1400017707
日期 2017.10.5

目录

内容一：总体概述.....	3
内容二：任务完成情况.....	3
任务完成列表（Y/N）	3
具体 Exercise 的完成情况.....	3
内容三：遇到的困难以及解决方法.....	10
内容四：收获及感想.....	10
内容五：对课程的意见和建议.....	10
内容六：参考文献.....	11

内容一：总体概述

本次 Lab 对 Nachos 进行线程调度的改进，实现了比 FIFO 相对更为高级的优先级调度，时间片轮转调度和多级队列调度算法。在多级队列实现时，充分分析了多级队列的特点，并利用了操作系统层不对用户开放的特性，利用单一队列和动态优先级实现了多级队列。

内容二：任务完成情况

任务完成列表 (Y/N)

	Exercise1	Exercise2	Exercise3	Challenge 1
第一部分	Y	Y	Y	Y

具体 Exercise 的完成情况

Exercise1 调研 Linux 或 Windows 中采用的进程/线程调度算法。

参考 IBM Developer Works 的文档^[1]对 Linux 的调度器进行简单的调研，下面简单地介绍 Linux 2.4.18 调度器，Linux 2.6 的 O(1)调度器。

Linux 2.4.18 调度器是基于优先级和时间片的调度器——每个进程在创建时被赋予时间片和优先级，时钟中断递减当前运行进程的时间片，当进程的时间片用完则进入队列等待重新赋予时间片。

进程分为两类——实时进程和普通进程。实时进程的优先级静态设定，并且始终高于普通进程的优先级，如此便保证实时进程一定在普通进程之前被优先调度；实时进程的调度策略有 FIFO 和轮转两种，相比较而言轮转会更加公平。普通进程的优先级是动态变化的，其优先级主要由 PCB 中的 Counter 字段加上 nice 设定的静态优先级确定，当当前进程的时间片用完后，调度器重新计算所有进程的 Counter 从而得到各个进程的优先级；计算 Counter 时，由于处于挂起状态的进程 Counter 没有用完，因而其优先级相应地被提高，而交互式进程经常由于 I/O 而被挂起，因此调度器倾向于优先调度交互式进程以便提高用户体验；额外需要说明的是，一个普通进程 Fork 出两个进程时，子进程的 Counter 减半，从而保证进程不可能依靠不断 Fork 来抢占 CPU 资源。

Linux 2.4.18 调度器需要线性时间更新 Counter，因而当进程数很大时，更新 Counter 的开销也很大；调度器优先调度 I/O 频繁进程，而非真正的交互式进程，一个非交互但 I/O 频繁的进程可以长时间占据 CPU 而使得真正的交互式进程被挂起甚至是饥饿；并且更重要的是该内核是非抢占式的，对于实时进程而言，当进程进入内核态就不会发生抢占，这是不可接受的。

Linux 2.6 的 O(1)调度器调度开销成为常数，支持内核态抢占，并且对于交互式进程有了更好的支持。O(1)调度器对实时进程采用 FIFO 或者轮转的调度策略，对普通进程采用 Normal 的调度策略。

普通进程的优先级依旧是动态计算的，计算公式如下。其中 `bonus` 一项取决于进程的平均睡眠时间——平均睡眠时间越长，`bonus` 越大，动态优先级相应越高。

$$\text{Dynamic priority} = \max(100, \min(\text{Static priority} - \text{bonus} + 5, 139))$$

额外的，通过优先级也可以判断出一个进程是否是交互式进程。如果满足下面的不等式，则被调度器视为交互式进程。

$$\text{Dynamic priority} \leq 3 \times \text{Static priority} / 4 + 28$$

由于优先级总数为 140（普通进程优先级 `prio` 和实时进程优先级 `rt_prio`），调度器为每个 CPU 维护两个进程队列数组，`active` 和 `expire`，数组元素为某个优先级的进程队列的指针。当需要选择最高优先级时， $O(1)$ 调度器直接从 `active` 中选择当前最高优先级队列中的第一个进程即可；为了得知当前最高优先级队列是哪一个，还维护了一个 `bitmap`，当某个优先级上有进程插入数组时，该 `bit` 就被置为 1。交互式进程和实时进程时间片用尽后，会重置时间片并插入 `active` 中，其他进程被插入 `expire` 中；当交互式进程和实时进程占用 CPU 时间达到某个阈值后，会被插入到 `expire` 中，以防 `expire` 中的其他进程饥饿。

Exercise2 源代码阅读

仔细阅读下列源代码，理解 Nachos 现有的线程调度机制。

- `code/threads/scheduler.h` 和 `code/threads/scheduler.cc`
- `code/threads/switch.s`
- `code/machine/timer.h` 和 `code/machine/timer.cc`

Nachos 现有的线程调度机制为 FIFO，只有一个等待队列，先进入等待队列的先执行，一个线程一旦抢占到 CPU，会一直占据直到运行完毕才会释放。

各个源码文件的介绍如下。

`code/threads/scheduler.cc` 和 `code/threads/scheduler.h` 中定义了 Nachos 的线程调度器 `Scheduler` 类，该类具有一个私有的 `List` 类型的等待队列 `readyList`，和公有方法 `ReadyToRun`，`FindNextToRun` 和 `Run`。`ReadyToRun` 将线程直接置于等待队列队尾；`FindNextToRun` 将等待队列队首元素取出并返回；`Run` 将正在执行的线程（旧线程）与待执行的线程（新线程）进行切换，具体步骤为检查旧线程栈溢出，设置新线程状态调用汇编切换 `SWITCH`，删除待删除线程。

`code/threads/switch.s` 为汇编代码实现的 `SWITCH` 过程。其中有对多种 ISA 的不同实现，寻找到 `i386` 的部分，发现其先进入第一个线程的内存区域，将全部寄存器内容按规则移入内存中，之后进入第二个线程的内存区域，将内存内容按照规则放入寄存器中，如此完成线程的上下文切换。

`code/machine/timer.h` 和 `code/machine/timer.cc` 中定义了 Nachos 的模拟时钟，该时钟基于中断，每当发生一次中断，便使得时钟加 1，并在中断处理函数中定下下一次的中断。

Exercise3 扩展线程调度算法，实现基于优先级的抢占式调度算法。

由于需要添加基于优先级抢占的调度，首先需要向 `code/threads/thread.h` 和 `code/threads/thread.cc` 中的 `Thread` 类添加私有变量优先级 `priority`，规定 `priority` 是 0-255 的整数，且数值越低优先级越大（`code/threads/list.cc` 中链表排序，所有元素按 `key` 从小到大排，`key` 小者在队前，因而优先级如此定义比较方便），还需要添加公有方法 `getPriority()` 读取优先级；此外，需要修改构造函数，使之带有优先级参数并根据该参数设置优先级，优先

级参数默认为 255，以避免默认构造的线程（比如 main 线程）抢占其他线程的资源。

首先考虑线程优先级抢占的实质，就是 `currentThread` 始终是最高优先级，或者说是最优 `priority` 线程（之一）。那么下面考虑线程进行抢占的时机：1. `currentThread` 执行完毕或被 `Yield`，需要将 `currentThread` 插入等待队列，之后从等待队列中选取优先级最高的线程（此线程优先级一定不高于 `currentThread`，或者说此线程的 `priority >= currentThread->priority`）；2. 新的线程开始执行，并且其优先级高于 `currentThread`，此时需要将 `currentThread` 重新插入等待队列，将新线程设置为 `currentThread`。

继续考虑上述两种情况，`currentThread` 执行完毕后，如果等待队列本身就是按优先级排序的，不需要任何改动，直接取出队首线程即可；`currentThread` 被 `Yield` 后，目前版本的 Nachos 会先从队列中取出队首线程，之后将调用 `ReadyToRun` 函数将 `currentThread` 插入等待队列中，因此这里需要将二者调换顺序；当新的线程开始执行（即 `Fork`）后，会调用 `ReadyToRun` 函数将该线程插入等待队列中，为了进行抢占，应将 `ReadyToRun` 函数改为若新线程优先级高于 `currentThread` 则将 `currentThread Yield`，否则将新线程插入等待队列。

进行修改后，发现如果通过 `currentThread->Yield()` 进入 `ReadyToRun` 函数，会再次调用 `Yield`，如此二者循环调用进入死循环，因此在 `ReadyToRun` 函数调用 `Yield` 前再加入判断排除循环调用的情况。

在 `code/threads/theradTest.cc` 中添加新的测试如下：定义各个线程执行的函数为循环打印 5 次，在首次打印时若没有达到边界条件（优先级达到阈值）则再创建一个优先级高于自身的线程，并令该线程调用此函数；创建一个线程，使该执行如上函数。预期情况下，每个新创建的线程都将直接抢占父线程的 CPU 资源，最终打印结果呈现为递归结构。实验结果如下图所示，符合预期。

```

C:\ 管理员: 命令提示符 - vagrant ssh
vagrant@precise32:/vagrant/nachos/nachos-3.4/code$ python onekeyrun.py -n -q 4
*****
Without-Make Mode
*****
New thread:      name: Forked Thread, UID: 1000, TID: 1, priority: 10
Thread 1 Loop 0
New thread:      name: Forked Thread, UID: 1000, TID: 2, priority: 9
Thread 2 Loop 0
New thread:      name: Forked Thread, UID: 1000, TID: 3, priority: 8
Thread 3 Loop 0
New thread:      name: Forked Thread, UID: 1000, TID: 4, priority: 7
Thread 4 Loop 0
New thread:      name: Forked Thread, UID: 1000, TID: 5, priority: 6
Thread 5 Loop 0
New thread:      name: Forked Thread, UID: 1000, TID: 6, priority: 5
Thread 6 Loop 0
Thread 6 Loop 1
Thread 6 Loop 2
Thread 6 Loop 3
Thread 6 Loop 4
Thread 5 Loop 1
Thread 5 Loop 2
Thread 5 Loop 3
Thread 5 Loop 4
Thread 4 Loop 1
Thread 4 Loop 2
Thread 4 Loop 3
Thread 4 Loop 4
Thread 3 Loop 1
Thread 3 Loop 2
Thread 3 Loop 3
Thread 3 Loop 4
Thread 2 Loop 1
Thread 2 Loop 2
Thread 2 Loop 3
Thread 2 Loop 4
Thread 1 Loop 1
Thread 1 Loop 2
Thread 1 Loop 3
Thread 1 Loop 4
No threads ready or runnable, and no pending interrupts.

```

Challenge1 可实现“时间片轮转算法”、“多级队列反馈调度算法”，或将 Linux 或 Windows 采用的调度算法应用到 Nachos 上。

我准备实现多级队列反馈调度算法。为了实现该算法，首先需要实现单队列的时间片轮转算法。

时间片轮转算法的实现如下。

首先，向 code/threads/thread.cc 和 code/threads/thread.h 的 Thread 类中添加私有成员变量时间片数 timeSlices 和已用时间片计数 usedTimeSlices，并相应添加公有方法 getTimeSlices 和 setTimeSlices 以便设定和读取时间片数，resetUsedTimeSlices 和 TickTack 来重置已用时间片计数和给已用时间片计数加一。由于时间片轮转算法中时间片数由创建线程时直接确定，之后不会更改，因而在构造函数中添加时间片数参数（默认为 1）。

再考虑已用时间片数，时钟每固定（也有可能会有随机延迟）时间便会发出一个中断，即为一个时间片，因而在该中断的处理函数中给当前线程 usedTimeSlices 加一并判断是否用尽时间片即可。一个线程下 CPU 有两种情况，一是当线程用尽时间片后，会通过 ReadyToRun

函数插入等待队列，这种情况下需要在 `ReadyToRun` 中将该线程 `usedTimeSlices` 归零；二是运行结束，此时会直接运行下一个线程（或队列为空，等待下一进程），这种情况下不需要修改 `usedTimeSlices`，因为只要进入等待队列的线程都会在 `ReadyToRun` 函数中将 `usedTimeSlices` 归零。

对 Nachos 的时钟中断机制再进行进一步的研究，发现 `code/threads/system.cc` 中定义了时钟中断处理函数，该函数会直接调用 `code/machine/interrupt.h` 和 `code/machine/imterrupt.cc` 中定义的 `YieldOnReturn` 函数，该函数设定 `yieldOnReturn` 标志位后退出，之后在 `OneTick` 函数中，检查标志位，若为真则调用 `currentThread->Yield()` 是当前线程下 CPU 进入等待队列。

因此在上述修改的基础之上再进行如下更改：1. 修改 `code/threads/system.cc` 中的 `TimerInterruptHandler` 时钟中断处理函数，在该函数中调用 `currentThread->TickTack()` 使当前线程已用时间片数加一，之后检查已用时间片数是否达到时间片数的限制，若达到则调用 `interrupt->YieldOnReturn()` 函数设定 `yieldOnReturn` 标志位注明将 `currentThread` Yield；2. 修改 `code/threads/scheduler.cc` 中的 `ReadyToRun` 函数，删去用于直接优先级抢占的 `currentThread->Yield()` 部分，添加 `thread->resetUsedTimeSlices()` 重置已用时间片数；3. 仿照 `Exercise3` 编写 `ThreadTest4` 测试函数，并向 `TimerInterruptHandler` 时钟中断处理函数中添加打印“Tick!”的语句，以便测试该时间片轮转算法的实现。

进行实验，发现没有打印过“Tick”，即时间片从未用尽，将测试函数中的循环设置为死循环，发现时间片仍旧从未用尽，推测是 `timer` 时钟没有运行。

阅读代码，首先发现 `code/threads/system.cc` 中的 `Initialize` 函数中只有在 `randomYield` 为真时才构造时钟 `timer`，修改此处使得无论何时都构造时钟。此外阅读 `code/machine/timer.h` 和 `code/machine/timer.cc` 发现只有一次中断开关，才能触发一次 `OneTick` 函数，相当于需要开关一次中断，才能手动拨一下时钟。

修改测试函数，在循环中加入一次中断开关，之后恢复原中断等级。再次进行测试，结果如下。

```
管理: 命令提示符 - vagrant ssh
vagrant@precise32:/vagrant/nachos/nachos-3.4/code$ python one
*****
Without-Make Mode
*****
Time Slices = 4
New thread 1
Thread 1 BEGIN!
New thread 2
Tick! Tick! Thread 1 Loop 19, priority 3, time-slice 2/4
Tick! Tick! Thread 2 BEGIN!
New thread 3
Tick! Tick! Thread 2 Loop 19, priority 2, time-slice 2/4
Tick! Tick! Thread 3 BEGIN!
Tick! Tick! Thread 3 Loop 19, priority 1, time-slice 2/4
Tick! Tick! Thread 3 Loop 39, priority 1, time-slice 0/4
Tick! Tick! Thread 3 Loop 59, priority 1, time-slice 2/4
Tick! Tick! Thread 3 Loop 79, priority 1, time-slice 0/4
Tick! Tick! Thread 3 Loop 99, priority 1, time-slice 2/4
Thread 3 END!
Thread 2 Loop 39, priority 2, time-slice 0/4
Tick! Tick! Thread 2 Loop 59, priority 2, time-slice 2/4
Tick! Tick! Thread 2 Loop 79, priority 2, time-slice 0/4
Tick! Tick! Thread 2 Loop 99, priority 2, time-slice 2/4
Thread 2 END!
Tick! Thread 1 Loop 39, priority 3, time-slice 1/4
Tick! Tick! Thread 1 Loop 59, priority 3, time-slice 3/4
Tick! Tick! Thread 1 Loop 79, priority 3, time-slice 1/4
Tick! Tick! Thread 1 Loop 99, priority 3, time-slice 3/4
Thread 1 END!
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

管理: 命令提示符 - vagrant ssh
vagrant@precise32:/vagrant/nachos/nachos-3.4/code$ python one
*****
Without-Make Mode
*****
Time Slices = 16
New thread 1
Thread 1 BEGIN!
New thread 2
Tick! Tick! Thread 1 Loop 19, priority 3, time-slice 2/16
Tick! Tick! Thread 1 Loop 39, priority 3, time-slice 4/16
Tick! Tick! Thread 1 Loop 59, priority 3, time-slice 6/16
Tick! Tick! Thread 1 Loop 79, priority 3, time-slice 8/16
Tick! Tick! Thread 1 Loop 99, priority 3, time-slice 10/16
Thread 1 END!
Thread 2 BEGIN!
New thread 3
Tick! Tick! Thread 2 Loop 19, priority 2, time-slice 2/16
Tick! Tick! Thread 2 Loop 39, priority 2, time-slice 4/16
Tick! Tick! Thread 2 Loop 59, priority 2, time-slice 6/16
Tick! Tick! Thread 2 Loop 79, priority 2, time-slice 8/16
Tick! Tick! Thread 2 Loop 99, priority 2, time-slice 10/16
Thread 2 END!
Thread 3 BEGIN!
Tick! Tick! Thread 3 Loop 19, priority 1, time-slice 2/16
Tick! Tick! Thread 3 Loop 39, priority 1, time-slice 4/16
Tick! Tick! Thread 3 Loop 59, priority 1, time-slice 6/16
Tick! Tick! Thread 3 Loop 79, priority 1, time-slice 8/16
Tick! Tick! Thread 3 Loop 99, priority 1, time-slice 10/16
Thread 3 END!
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

在时间片数为 4 时，各个线程都不可能在 4 个时间片的限制内完成，因而出现了抢占，打印结果呈现为类似递归的形式；当时间片数为 16 时，各个线程可以在限制内完成，因而

没有出现抢占，各个线程顺序执行完毕。

考虑到一个问题，即若一个线程在用尽时间片之前结束了，那么将切换到下一个线程继续使用该时间片，那么在该时间片耗尽时时钟中断，是否应该给之后的线程 TickTack？我认为后继线程并没有完整地使用一个时间片，因而应当将这“半个”时间片“赠送”给后继线程。所以，我向 Thread 类中添加了共有方法 `extraUsedTimeSlices`，设定 `usedTimeSlices` 为 -1（即将当前时间片不计入其中），之后继续修改了 `code/threads/thread.cc` 中的 `Sleep` 函数，使之在运行下一个线程之前，先调用 `extraUsedTimeSlices` 将这半个时间片赠送给下一线程。

多级队列反馈调度算法的实现如下。

Nachos 的线程调度只有一个等待队列，因此要实现多级队列的前提就是将等待队列变为多个队列，而这需要将队列包装为与 `code/threads/list.cc` 和 `code/threads/list.h` 中一致的多级队列，提供相同的方法才可以。这一工作比较复杂，因此我考虑是否有可行的替代方法。

分析多级队列的机制，假设有多级队列中有两个队列，所有线程直接先进入队列 A 等待，等待到 CPU 后上 CPU，在 CPU 执行 t 个时间片之后下 CPU，进入队列 B，当从队列 B 出队上 CPU 后，在 CPU 上执行 $2t$ 个时间片后下 CPU，再次进入队列 B；各个队列之中按照线程优先级排队。假设线程的优先级为从 0 到 $(p-1)$ 的整数（共 p 个优先级）且数值越低优先级越高，假设队列 A 和 B 中由 0 到 $(p-1)$ 优先级的线程填满，那么一定可以得到队列 B 中的 0 优先级的线程一定在队列 A 中的 $(p-1)$ 优先级的线程之后等到 CPU；因此在实际运行时，队列 B 中的线程的实际的优先级 p_{real} 和其数值的优先级 p_{num} 满足 $p_{\text{real}} \leq p_{\text{num}} + p$ ，即所有队列 B 中的线程，在实际运行时相当于其数值的优先级加上优先级登记数目。

因此我们可以用单一的队列，配合变化的优先级，实现多级队列反馈调度算法的效果——线程的初始优先级为 -256 到 -1（依旧符合 255 个优先级的要求），每当线程进入队列时将优先级 `priority` 加 256，并将时间片数 `timeSlices` 乘 2，对处于边界条件（处于多级队列最后一个队列）的线程，其 `priority` 和 `timeSlices` 不变，其余与时间片轮转调度算法一致。

对于代码的变动如下。在 `code/threads/thread.h` 和 `code/threads/thread.cc` 中的 Thread 类中添加公有方法 `getPriorityLevel` 用于计算该线程逻辑上处于哪个队列之中，修改公有方法 `getPriority` 使之返回初始优先级数值（对 256 求模即可得到），再添加最为重要的公有方法 `priorityLevelDown`，每次将线程入队等待时调用该方法使得线程的 `priority` 和 `timeSlices` 发生上述变化。在 `code/threads/scheduler.cc` 的 `ReadyToRun` 函数中添加对 `priorityLevelDown` 的调用，并将插入队列时的优先级改为 `getPriority()+255*getPriorityLevel()`（用实际优先级数值，即 `priority` 排队）。仿照之前的测试函数编写测试函数 `threadTest6`，创建三个线程 `t1`，`t2` 和 `t3`，分别给予优先级 10，0 和 5，之后 Fork 执行函数，循环打印 10000 次。

测试结果如下图（截图为部分结果）。符合多级队列算法的特征——队列等级高的（ $p\text{-level}$ 数值越低，队列等级越高）先抢占 CPU，相同队列时优先级（`priority` 数值越低，优先级越高）高的先抢占 CPU；队列等级降低一级，时间片数乘 2；最后一个队列（ $p\text{-level}=7$ ）的线程再次进入该队列时，队列等级，优先级数值，时间片数不变。


```
管理: 命令提示符 - vagrant ssh
vagrant@precise32: /vagrant/nachos/nachos-3.4/code$ python onekeyrun.py -n -q
6
*****
Without-Make Mode
*****
thread 2 loop 0, p-level 0 priority 0, time-slice 0/2
thread 3 loop 0, p-level 0 priority 5, time-slice 0/2
thread 1 loop 0, p-level 0 priority 10, time-slice 0/2
thread 2 loop 100, p-level 2 priority 0, time-slice 4/8
thread 3 loop 100, p-level 2 priority 5, time-slice 4/8
thread 1 loop 100, p-level 2 priority 10, time-slice 4/8
thread 2 loop 200, p-level 3 priority 0, time-slice 6/16
thread 3 loop 200, p-level 3 priority 5, time-slice 6/16
thread 1 loop 200, p-level 3 priority 10, time-slice 6/16
thread 2 loop 300, p-level 4 priority 0, time-slice 0/32
thread 3 loop 400, p-level 4 priority 0, time-slice 10/32
thread 2 loop 500, p-level 4 priority 0, time-slice 20/32
thread 3 loop 600, p-level 4 priority 0, time-slice 30/32
thread 2 loop 300, p-level 4 priority 5, time-slice 0/32
thread 3 loop 400, p-level 4 priority 5, time-slice 10/32
thread 3 loop 500, p-level 4 priority 5, time-slice 20/32
thread 3 loop 600, p-level 4 priority 5, time-slice 30/32
thread 1 loop 400, p-level 4 priority 10, time-slice 10/32
thread 1 loop 500, p-level 4 priority 10, time-slice 20/32
thread 1 loop 600, p-level 4 priority 10, time-slice 30/32
thread 2 loop 700, p-level 5 priority 0, time-slice 9/64
thread 3 loop 800, p-level 5 priority 0, time-slice 19/64
thread 2 loop 900, p-level 5 priority 0, time-slice 29/64
thread 2 loop 1000, p-level 5 priority 0, time-slice 39/64
thread 2 loop 1100, p-level 5 priority 0, time-slice 49/64
thread 2 loop 1200, p-level 5 priority 0, time-slice 59/64
thread 3 loop 700, p-level 5 priority 5, time-slice 8/64
thread 3 loop 800, p-level 5 priority 5, time-slice 18/64
thread 3 loop 900, p-level 5 priority 5, time-slice 28/64
thread 3 loop 1000, p-level 5 priority 5, time-slice 38/64
thread 3 loop 1100, p-level 5 priority 5, time-slice 48/64
thread 3 loop 1200, p-level 5 priority 5, time-slice 58/64
thread 1 loop 700, p-level 5 priority 10, time-slice 8/64
thread 1 loop 800, p-level 5 priority 10, time-slice 18/64

管理: 命令提示符 - vagrant ssh
thread 1 loop 96300, p-level 7 priority 10, time-slice 165/256
thread 1 loop 96400, p-level 7 priority 10, time-slice 175/256
thread 1 loop 96500, p-level 7 priority 10, time-slice 185/256
thread 1 loop 96600, p-level 7 priority 10, time-slice 195/256
thread 1 loop 96700, p-level 7 priority 10, time-slice 205/256
thread 1 loop 96800, p-level 7 priority 10, time-slice 215/256
thread 1 loop 96900, p-level 7 priority 10, time-slice 225/256
thread 1 loop 97000, p-level 7 priority 10, time-slice 235/256
thread 1 loop 97100, p-level 7 priority 10, time-slice 245/256
thread 1 loop 97200, p-level 7 priority 10, time-slice 255/256
thread 1 loop 97300, p-level 7 priority 10, time-slice 9/256
thread 1 loop 97400, p-level 7 priority 10, time-slice 19/256
thread 1 loop 97500, p-level 7 priority 10, time-slice 29/256
thread 1 loop 97600, p-level 7 priority 10, time-slice 39/256
thread 1 loop 97700, p-level 7 priority 10, time-slice 49/256
thread 1 loop 97800, p-level 7 priority 10, time-slice 59/256
thread 1 loop 97900, p-level 7 priority 10, time-slice 69/256
thread 1 loop 98000, p-level 7 priority 10, time-slice 79/256
thread 1 loop 98100, p-level 7 priority 10, time-slice 89/256
thread 1 loop 98200, p-level 7 priority 10, time-slice 99/256
thread 1 loop 98300, p-level 7 priority 10, time-slice 109/256
thread 1 loop 98400, p-level 7 priority 10, time-slice 119/256
thread 1 loop 98500, p-level 7 priority 10, time-slice 129/256
thread 1 loop 98600, p-level 7 priority 10, time-slice 139/256
thread 1 loop 98700, p-level 7 priority 10, time-slice 149/256
thread 1 loop 98800, p-level 7 priority 10, time-slice 159/256
thread 1 loop 98900, p-level 7 priority 10, time-slice 169/256
thread 1 loop 99000, p-level 7 priority 10, time-slice 179/256
thread 1 loop 99100, p-level 7 priority 10, time-slice 189/256
thread 1 loop 99200, p-level 7 priority 10, time-slice 199/256
thread 1 loop 99300, p-level 7 priority 10, time-slice 209/256
thread 1 loop 99400, p-level 7 priority 10, time-slice 219/256
thread 1 loop 99500, p-level 7 priority 10, time-slice 229/256
thread 1 loop 99600, p-level 7 priority 10, time-slice 239/256
thread 1 loop 99700, p-level 7 priority 10, time-slice 249/256
thread 1 loop 99800, p-level 7 priority 10, time-slice 3/256
thread 1 loop 99900, p-level 7 priority 10, time-slice 13/256
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

修复删除线程的 BUG

Nachos 现有的线程删除机制为：使用指针 `threadToBeDestroyed` 指示待删除线程，线程结束后调用 `Finish` 函数将 `threadToBeDestroyed` 指示为自身，之后在 `ReadyToRun` 函数中检查该指针并删除。如果两个线程先后结束，使得指针被替换，将导致第一个结束的线程丢失，而始终无法删除。

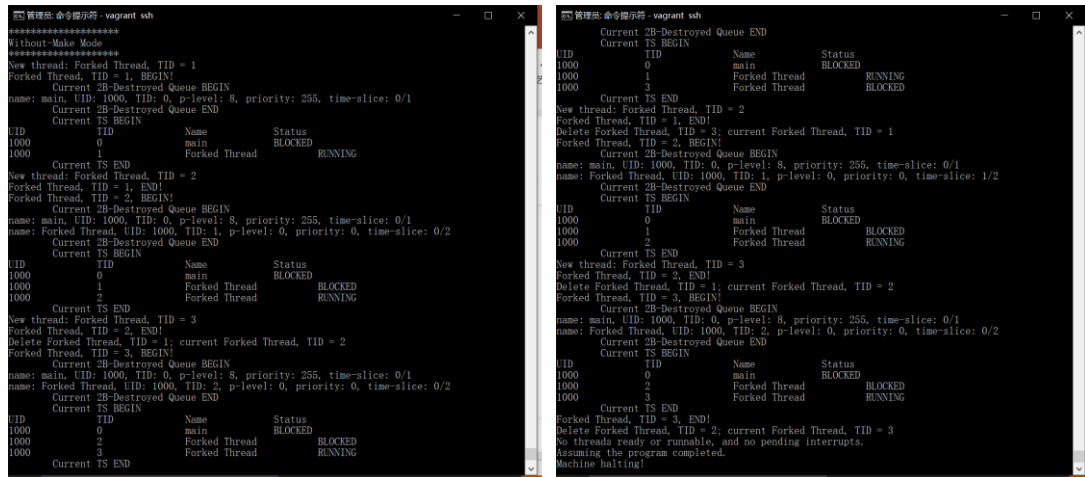
为了使得指针不被替换，将该指针更换为 List 链表，将所有赋值操作更换为插入操作，将所有删除操作更换为删除各个元素即可。

代码变动如下：将 `code/threads/system.cc` 和 `code/threads/system.h` 中的 `threadToBeDestroyed` 的类型更换为 List；之后按照上述规则将操作进行替换；此外，考虑到尽量将对 `threadToBeDestroyed` 的操作放在临近位置里，因而将 `code/threads/scheduler.cc` 中的 `ReadyToRun` 中的删除操作移放在 `code/threads/thread.cc` 中的 `Finish` 函数之中。

额外的，由于 `main` 线程是作为 `Idle` 线程存在的，`main` 线程不能被删除；并且，`currentThread` 也不能被删除，因为一个对象不可以析构自身。因此在删除时对这两种情况进行特殊处理，不进行删除，而是重新插入 `threadToBeDestroyed` 队列之中。

测试函数 `ThreadTest7` 为，创建线程并 Fork 执行函数 `DeleteThread`，该函数打印待删除列表 `threadToBeDestroyed` 和 `TS`，之后如果满足条件则再创建一个新的线程并 Fork 执行 `DeleteThread`，最后 `Finish` 结束。

实验结果如下，基本符合预期。



内容三：遇到的困难以及解决方法

在实现时间片轮转时，测试时发现时钟没有自动运行，通读 `code/machine/timer.h` 与 `code/machine.cc` 和 `code/threads/interrupt.h` 与 `code/threads/interrupt.cc`，以及 `code/threads/system.cc` 后，发现 Nachos 系统的时钟需要手动开关一次中断（`interrupt->setLevel(IntOff); interrupt->setLevel(IntOn);`）才能加一，并且在默认情况下 Nachos 中没有时钟。通过更改 `code/threads/system.cc` 中时钟 `timer` 的部分，添加时钟；再在测试代码中手动开关中断使得时钟加一运行。之后该问题得以解决。

内容四：收获及感想

本次试验总体而言比较简单，但比上一次 Lab 1 难度有所提升，尤其是时钟的部分，在没有阅读代码的情况下，基本不可能想到。总而言之，我的收获和得意之处有下。

利用单队列和动态优先级实现多级队列，比较充分地体现了计算机系统各层之间的关系，如果在用户层面，封闭系统代码而只看结果，我的实现方式与多级队列几乎没有任何区别；然而在系统层面，我利用了动态优先级实现了多级队列，但是相对的，在我的理解里，多级队列的本质想法是利用静态优先级实现动态优先级。

如此转变，是颇为有趣的；几乎没有证据可以表明静态优先级和动态优先级孰优孰劣，优劣完全取决于环境和思路。我认为在 `code/threads/list.h` 和 `code/threads/list.cc` 的单队列的限制之下，我的实现方法最符合奥卡姆剃刀原则，即“如无必要，勿增实体”。

内容五：对课程的意见和建议

Nachos 的时钟设计，是绝对出乎常人意料的。我认为，正常思路应该是将时钟独立于 Nachos 线程，利用多线程/多进程的方式，实现时钟和 Nachos 系统的并行。因此我认为应当修改 Nachos，或者将时钟并行作为一个独立的 Lab 或这 challenge 之一。

内容六：参考文献

[1] 刘明. Linux 调度器发展简述 [DB/OL]

<https://www.ibm.com/developerworks/cn/linux/l-cn-scheduler/>