1.调研 Linux 或 Windows 中采用的进程/线程调度算法。

Windows 的任务调度以线程为单位，基于抢占式的时间片轮转算法。

Windows 的线程分为 32 个优先级，其中 15-31 是实时线程，1-14 是普通线程，0 优先级被操作系统保留。线程的基本优先级取值范围时它的进程的优先级加减 2，线程的动态优先级取值范围下界时它的进程的优先级减 2，上界是 31。如果一个优先级高于 14 的线程发生了等待时间，则它的优先级会被重置（降低）。

Windows 的线程具有 CPU 亲和性，即可以被限制只能在某些 CPU 上运行。对于给定的 CPU，如果某个高优先级线程被创建，或某个线程的优先级被提高，高于现在在该 CPU 上运行的进程，则进行线程切换，高优先级线程***抢占***CPU，被抢占的线程放回队首；如果现在运行的线程进入等待状态，则降低它的优先级，并调度最高优先级的就绪线程，当该等待线程就绪后，赋予其原来的三分之一的时间配额；如果现在运行的线程用尽了一个时间片，则把它放回该优先级的线程队列的队尾，令队首线程上 CPU。

Linux 的任务调度基于进程，进程调度策略分为三种。对于实时进程，有 SCHED_FIFO 先到先服务，和 SCHED_RR 时间片轮转两种策略；对于普通进程，使用 SCHED_OTHER 分时调度策略。

实时进程将得到优先调用，实时进程根据实时优先级决定调度权值。使用 FIFO 策略时，相同优先级的进程不会被赶下 CPU，直到它主动放弃 CPU；使用 RR 策略时，进程的时间片用完，系统将为它重新分配时间片，并置于就绪队列尾。

没有就绪的实时进程时，系统才会调用分时进程。分时进程通过它的 counter 和 nice 的差值决定进程权值，nice 越小，counter 越大，被调度的概率越大，也就是曾经使用了 cpu 最少的进程将会得到优先调度。当运行的进程用完了 counter 的时间，或者主动放弃 CPU，则被放到队尾。

2.代码阅读。

Timer.cc/Timer.h: 用来模拟计时器，每隔一段时间发出一次时间中断，每次中断调用相同的用户指定的处理函数。

通过构造函数可以指定中断处理函数和中断处理函数的参数，可以设置中断间隔是否随机。构造函数先创建一个中断放入等待队列，对应的中断处理函数是 TimerHandler()，是用户中断函数的一个封装。

TimerHandler() 间接调用 TimerExpired()，TimerExpired()首先创建一个新的时间中断放入等待队列，然后再调用用户的中断处理函数。

TimeOfNextInterrupt()用来计算中断时间间隔，有固定和随机两种模式，固定的话是 100 个时间单位，随机的范围是 1 到 200 个时间单位。

Scheduler.cc/Scheduler.h: 用于线程调度

ReadyToRun() 向线程队列尾部插入就绪的线程

FindNextToRun() 返回队首线程

Run() 进行线程切换，检查上一个线程的是否栈溢出，销毁一个结束的线程。

switch.s: 用于线程切换，使用汇编语言实现，支持四种体系结构

ThreadRoot() 负责规划这个线程的执行流程，进行线程创建，线程的函数的调用和线程的返回。

SWITCH() 负责保存上一个线程的机器状态（例如栈指针，寄存器值，运行状态等）到它的 TCB 中，并且把下一个线程的 TCB 中内容拷贝到机器中。

3. 抢占式线程调度：

在 Thread 类中增加属性 prior，值越小优先级越高
在 Scheduler 类中增加 JustCreate 函数对 ReadyTo，用于决定刚刚创建的线程是否抢占 CPU，方法是先用刚创建的线程调用 ReadyToRun，再比较刚创建线程与当前线程的优先级，如果刚创建线程的优先级更高，则令当前线程 Yield()
把 Scheduler 类中的 ReadyToRun 函数的 list->Append 改为 list->SortedInsert

在 threadTest.cc 中实现下面 3 个函数，首先 main 线程调用 ThreadTest3 函数，然后 ThreadTest3 新建线程调用 ThreadTest4 函数，然后 ThreadTest4 新建线程调用 myThread 函数，后两个线程的优先级与调用顺序相反（prior 越小优先级越高）

```
void myThread(int which)
{
    printf("*** thread %d with priority %d\n", currentThread->getTid() , currentThread->prior );
    return;
}
void
ThreadTest4(int none)
{
    DEBUG('t', "Entering ThreadTest4");

    Thread *t = createThread("forked thread", 4);
    t->Fork(myThread, 1);
    printf("*** thread %d with priority %d\n", currentThread->getTid() , currentThread->prior );

}

void
ThreadTest3()
{
    DEBUG('t', "Entering ThreadTest3");

    Thread *t = createThread("forked thread", 5);
    t->Fork(ThreadTest4, 1);
    printf("*** thread %d with priority %d\n", currentThread->getTid() , currentThread->prior );
}
```

运行效果图，可以看到 ThreadTest4 所在线程成功抢占了 ThreadTest3 所属线程

```
vagrant@precise32:/vagrant/nachos/nachos-3.4/code/threads$ ./nachos -q 3
*** thread 1 with priority 0
*** thread 3 with priority 4
*** thread 2 with priority 5
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

4. 时间片轮转调度

总体实现思路是，每个线程上 CPU 时给自己预定一个中断，中断到来后使自己 Yield()，加入就绪队列尾部。

具体而言，在 Scheduler::Run() 中规划一个中断事件，该中断的时间为时间片长度，中段发生代表线程时间片用尽。中断处理函数有两个参数，分别代表线程 tid 和执行次数，防止一个不属于当前线程的中断使当前线程下 CPU，这两个参数合并成一个参数传给中断处理函数。

```
interrupt->Schedule(ShouldYield, nextThread->getTid()+
        (++nextThread->version)*1000, TimeTick, TimerInt);
```

ShouldYield() 是中断处理函数，它首先判断当前线程是不是要下 CPU 的线程，如果是，则 YieldOnReturn()，不直接 Yield 是因为中断不能嵌套。当前线程如果不是预定中断的线程，则什么也不做。这种情况有可能因为上个线程提前返回而发生。

```
void ShouldYield(int arg)
{
    int tid = arg%1000 ;
    int version = arg/1000 ;
    //printf("t:%d, v:%d\n", tid, version) ;
    if(tid == currentThread->getTid() && version == currentThread->version)
    {
        //printf("Y\n") ;
        interrupt->YieldOnReturn();
    }
}
```

测试函数 ThreadTest5() 比较复杂。因为 nachos 模拟的硬件计时器不能及时调用自己，所以我使用自己的计时器 TimeAdvance()。TimeAdvance() 从命令行读入一个时间，并且把系统时间加上这些时间，然后判断此时有没有中断事件。为了防止 nachos 计时器干扰，我把它的时间增量 SystemTick 变成了 0。我设置的时间片长度 TimeTick 是 50。主线程会 fork 两个 TimeAdvance 线程，使用时间片轮转进行调度。

```
void
ThreadTest5()
{
    DEBUG('t', "Entering ThreadTest5");

    Thread *t = createThread("time thread 2", 5);
    t->Fork(TimeAdvance, 1);
    Thread *h = createThread("time thread 3", 5);
    h->Fork(TimeAdvance, 1);
```

```
void TimeAdvance(int none)
{
    int advance = 0 ;
    for(int i = 1 ; i <= 5 ; i ++)
    {
        printf("Thread %d working for: ", currentThread->getTid()) ;
        scanf("%d", &advance) ;
        stats->totalTicks += advance;
        stats->systemTicks += advance;
        //interrupt -> ChangeLevel(IntOn, IntOff) ;
        interrupt->OneTick() ;
        //interrupt -> ChangeLevel(IntOff, IntOn) ;
    }
}
```

```
#define UserTick    1   // advance for each user-level instruction
#define SystemTick  0   // advance each time interrupts are enabled
#define RotationTime    500     // time disk takes to rotate one sector
#define SeekTime    500     // time disk takes to seek past one track
#define ConsoleTime     100 // time to read or write one character
#define NetworkTime     100     // time to send or receive one packet
#define TimerTicks  20      // (average) time between timer interrupts
#define TimeTick    50
```

```
Switching from thread "main" to thread "time thread 2"
Scheduling interrupt handler the timer at time = 50
Thread 2 working for: 20
Thread 2 working for: 30
Switching from thread "time thread 2" to thread "time thread 3"
Scheduling interrupt handler the timer at time = 100
Thread 3 working for: 50
Switching from thread "time thread 3" to thread "time thread 2"
Scheduling interrupt handler the timer at time = 150
Thread 2 working for: 49
Thread 2 working for: 1
Switching from thread "time thread 2" to thread "time thread 3"
Scheduling interrupt handler the timer at time = 200
Thread 3 working for: 50
Switching from thread "time thread 3" to thread "time thread 2"
Scheduling interrupt handler the timer at time = 250
Thread 2 working for: 10
Switching from thread "time thread 2" to thread "time thread 3"
Scheduling interrupt handler the timer at time = 260
Thread 3 working for: 10
Thread 3 working for: 40
Thread 3 working for: 40
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 300, idle 0, system 300, user 0
```

从实验结果可以看出，每个线程上 CPU 的时候，会给自己预订一个中断。总共执行一个时间片长度后，中断被触发，发生线程切换。

5. Challenge：多优先级反馈队列

多个优先级队列可以看做有两个关键字的单个优先级队列，主键是优先级，副键是到来时间，优先级越高，越早被执行；相同优先级，到来时间越早，越早被执行。

任意线程用尽当前时间片后，优先级加 1，时间片长度由优先级*10 给定。

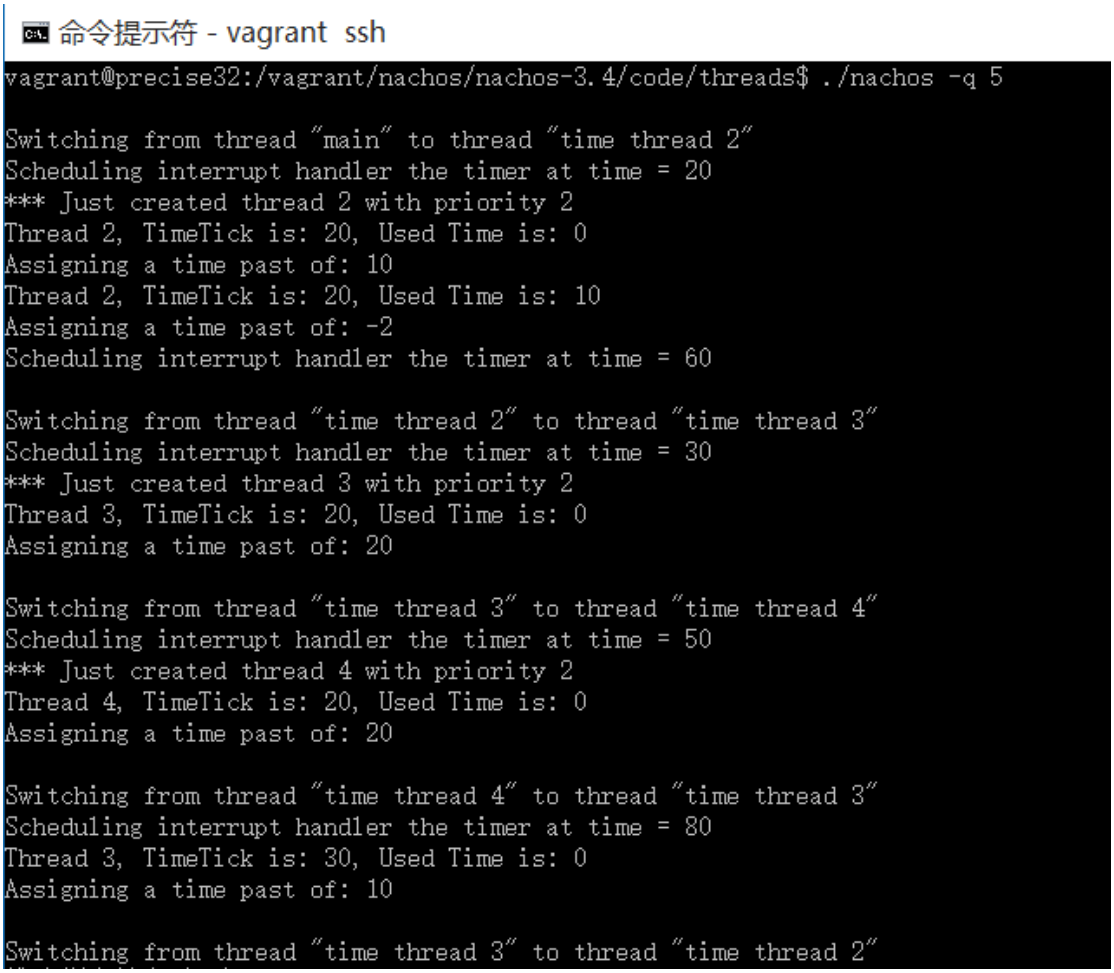
任意线程 Yield() 后，优先级不变，下一时间片变为现在时间片的剩余时间，插入队列。

任意线程 Sleep() 后，优先级不变，下一时间片长度与现在时间片长度相同，插入队列。

其他程序实现与时间片轮转 exercise 相同，不再赘述。

测试程序中，输入数字为当前线程的时间消耗。输入-1 代表当前线程 Yield()，输入-2 代表当前线程 Sleep()，并在 50 ticks 后中断唤醒。

使用三个初始优先级为 2 的线程，每个线程执行 5 次，测试结果如下：



```
Scheduling interrupt handler the timer at time = 80
Thread 2, TimeTick is: 20, Used Time is: 0
Assigning a time past of: 10
Thread 2, TimeTick is: 20, Used Time is: 10
Assigning a time past of: -1

Switching from thread "time thread 2" to thread "time thread 2"
Scheduling interrupt handler the timer at time = 80
Thread 2, TimeTick is: 10, Used Time is: 0
Assigning a time past of: 10

Switching from thread "time thread 2" to thread "time thread 4"
Scheduling interrupt handler the timer at time = 110
Thread 4, TimeTick is: 30, Used Time is: 0
Assigning a time past of: 30

Switching from thread "time thread 4" to thread "time thread 3"
Scheduling interrupt handler the timer at time = 130
Thread 3, TimeTick is: 20, Used Time is: 0
Assigning a time past of: 20
```

重点观察 Thread2 的行为，发现它在 Yield()和 Sleep()后的时间片变化符合预期。当 Thread3 和 Thread4 用尽时间片后，优先级下降，时间片变长。Thread 在 Yield() 后，处于最高优先级的线程只有它自己，所以它自己切换到了自己。

实验难点：

1. 中断的开关容易引起 bug。在中断处理函数中不能调用 Yield()，但是 Yield()中没有相关的 assert 语句，从而引发了 segmentation falut。找到这个问题的原因需要较长的调试时间。

2. nachos 模拟硬件时钟不理想。如果一个函数始终在运行，时钟甚至不能自己前进。只有发生中断和其他系统时间时，OneTick()才会被调用，时间才能前进。这就导致做时间片轮转调度时没有一个可以参照的系统时间。为此，我在测试函数中自己模拟了时钟的前进，验证了调度代码的正确性。

3. 测试函数比较难以设计，需要测试到 Yield()和 Sleep()和抢占等情况，睡眠之后的唤醒也需要自己进行。

杨东升
1400012898