

线程调度实习报告

姓名： 学号：

日期： 2020/10/31

目录

内容一：任务完成情况.....	3
任务完成列表（Y/N）	3
具体 Exercise 完成情况	3
Exercise1	3
Exercise2	9
Exercise3	13
Challenge	18
内容二：遇到的困难以及收获.....	26
内容三：对课程或 Lab 的意见和建议.....	26
内容四：参考文献.....	26

内容一：任务完成情况

任务完成列表 (Y/N)

	Exercise1	Exercise2	Exercise3	Challenge
第一部分	Y	Y	Y	Y

具体 Exercise 完成情况

Exercise1

在这部分练习中，我调研了 Linux 的进程调度机制，现将这一机制进行简要的阐述。调研的内核版本基于 3.1.10-g05b777c。

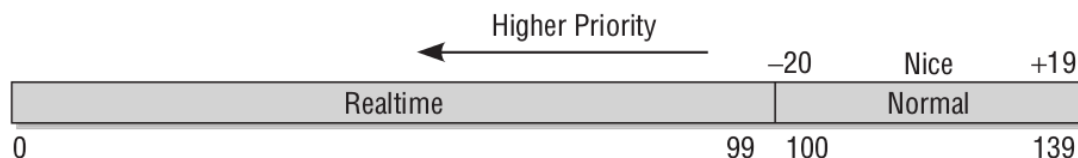
(一) 进程的分类

在描述进程调度机制之前，让我们首先对 linux 的进程做一个简要的分类。我们有三种视角来为进程分类。

首先，我们可以把进程分类为 CPU 密集型进程或者 I/O 密集型进程。CPU 密集型进程会进行大量的 CPU 运算，而另一些则会花费很多时间等待相对较慢的 I/O 操作完成。I/O 操作可以是用户输入、磁盘输入，或者网络输入。Linux 的调度目标是：及时响应 I/O 密集型进程，且提高 CPU 密集型进程的效率。调度算法需要在运行时长（时间片长度）和切换频率之间进行权衡取舍，在两个需求之间达到平衡。

其次，Linux 进程还可以分为实时进程（RT）和普通进程。实时进程有严格的时序要求，因此实时进程的优先级也要更高一些。在 Linux 中，为 RT 进程和普通进程使用了不同的调度器。

除此之外，我们也可以按照优先级来分类进程。Linux 中，优先级越高，优先级的数值表示越小。实时进程的优先级为 0-99，普通进程的优先级为 100-139。



(二) Linux 中的进程调度器

Linux 的进程调度器是模块化的，它是用不同的策略来调度不同类别的进程。每个调度算法都被封装在 Linux 中的 sched_class 结构体中，它提供接口给主调度器框架，框架就会根据算法的实现来处理对应的任务。这个数据结构被存放在 include/linux/sched.h 中，内容如下：

```

struct sched_class {

    const struct sched_class *next;
    void (*enqueue_task) (struct rq *rq, struct task_struct *p, int flags);
    void (*dequeue_task) (struct rq *rq, struct task_struct *p, int flags);
    void (*yield_task) (struct rq *rq);
    bool (*yield_to_task) (struct rq *rq, struct task_struct *p, bool preempt);
    void (*check_preempt_curr) (struct rq *rq, struct task_struct *p, int flags);
    struct task_struct * (*pick_next_task) (struct rq *rq);
    void (*put_prev_task) (struct rq *rq, struct task_struct *p);
#ifdef CONFIG_SMP
    int (*select_task_rq) (struct task_struct *p, int sd_flag, int flags);
    void (*pre_schedule) (struct rq *this_rq, struct task_struct *task);
    void (*post_schedule) (struct rq *this_rq);
    void (*task_waking) (struct task_struct *task);
    void (*task_woken) (struct rq *this_rq, struct task_struct *task);
    void (*set_cpus_allowed) (struct task_struct *p, const struct cpumask *newmask);
    void (*rq_online) (struct rq *rq);
    void (*rq_offline) (struct rq *rq);
#endif
    void (*set_curr_task) (struct rq *rq);
    void (*task_tick) (struct rq *rq, struct task_struct *p, int queued);
    void (*task_fork) (struct task_struct *p);
    void (*switched_from) (struct rq *this_rq, struct task_struct *task);
    void (*switched_to) (struct rq *this_rq, struct task_struct *task);
    void (*prio_changed) (struct rq *this_rq, struct task_struct *task, int oldprio);
    unsigned int (*get_rr_interval) (struct rq *rq, struct task_struct *task);
#ifdef CONFIG_FAIR_GROUP_SCHED
    void (*task_move_group) (struct task_struct *p, int on_rq);
#endif
};

```

除了第一个成员，其余成员全部都是函数指针，可以被调度器框架用来调用对应的策略。

内核中所有的 sched_class 结构按照优先级放在一个链表上。结构体中的 next 成员指向下一个结构体。在调研的内核版本汇总，链表内容依次为：

```
stop_sched_class → rt_sched_class → fair_sched_class → idle_sched_class → NULL
```

stop 类用来调度每个 CPU 上抢占一切任务且不可被其他任务抢占的 stop task；idle 类调度每个 CPU 上没有可运行任务时才会运行的空闲任务；rt 类调度实时任务；fair 类调度普通任务。

（三）主运行队列

主运行队列在每个 CPU 上都保存一份，其数据结构定义在 kernel/sched.c 中，记录了特定 CPU 上所有可运行的任务，主要内容有：

1. 一个锁，用于同步当前 CPU 的调度操作：

```
raw_spinlock_t lock;
```

2. 指向当前运行的任务、空闲任务、停止任务的 task_struct（Linux 的 PCB）结构体指针：

```
struct task_struct *curr, *idle, *stop;
```

3. 公平调度（CFS）和实时调度类运行队列数据结构

```
struct cfs_rq cfs;
struct rt_rq rt;
```

（四）调度框架

进入调度器的主入口是函数 `schedule()`，定义在 `kernel/sched.c`。内核调用这个函数，来决定接下来要运行的进程，其代码如下：

```
static void __sched __schedule(void)
{
    struct task_struct *prev, *next;
    unsigned long *switch_count;
    struct rq *rq;
    int cpu;

    need_resched:
    preempt_disable();
    cpu = smp_processor_id();
    rq = cpu_rq(cpu);
    rcu_note_context_switch(cpu);
    prev = rq->curr;
    schedule_debug(prev);
    if (sched_feat(HRTICK))
        hrtick_clear(rq);
    raw_spin_lock_irq(&rq->lock);
    switch_count = &prev->nvcsw;
    if (prev->state && !(preempt_count() & PREEMPT_ACTIVE)) {
        if (unlikely(signal_pending_state(prev->state, prev))) {
            prev->state = TASK_RUNNING;
        } else {
            deactivate_task(rq, prev, DEQUEUE_SLEEP);
            prev->on_rq = 0;
            /*
             * If a worker went to sleep, notify and ask workqueue
             * whether it wants to wake up a task to maintain
             * concurrency.
             */
            if (prev->flags & PF_WQ_WORKER) {
                struct task_struct *to_wakeup;
                to_wakeup = wq_worker_sleeping(prev, cpu);
                if (to_wakeup)
                    try_to_wake_up_local(to_wakeup);
            }
        }
        switch_count = &prev->nvcsw;
    }
    pre_schedule(rq, prev);

    if (unlikely(!rq->nr_running))
        idle_balance(cpu, rq);
    put_prev_task(rq, prev);
    next = pick_next_task(rq);
    clear_tsk_need_resched(prev);
    rq->skip_clock_update = 0;
    if (likely(prev != next)) {
        rq->nr_switches++;
        rq->curr = next;
        ++*switch_count;
        context_switch(rq, prev, next); /* unlocks the rq */
        /*
         * The context switch have flipped the stack from under us
         * and restored the local variables which were saved when
         * this task called schedule() in the past. prev == current
         * is still correct, but it can be moved to another cpu/rq.
         */
        cpu = smp_processor_id();
        rq = cpu_rq(cpu);
    } else
        raw_spin_unlock_irq(&rq->lock);
    post_schedule(rq);
    preempt_enable_no_resched();
    if (need_resched())
        goto need_resched;
}
```

由于 Linux 内核可抢占，所以在内核空间执行的进程代码是可以被高优先级的进程随时打断的，被抢占的进程在内核空间没执行完的操作需要等待下次调度时执行。因此，调度函数首先要调用 `preempt_disable()` 禁止抢占，这样在调度中的进程就不会在执行临界区操作时被打断。

接下来系统锁定当前 CPU 的运行队列锁来建立另一个保护，因为某一时刻只允许一个线程来修改运行队列。

然后，`schedule()` 检查 `prev` 指向的先进程。若它在内核态，不可运行，且没被抢占，那么这个进程会被移出运行队列；如果它又非阻塞的等待信号，那么把它的运行状态设置为 `TASK_RUNNING` 然后继续留在运行队列里。这就意味着 `prev` 指向的进程有获取了一次被调度执行的机会。

下一步，函数检查 CPU 中的运行队列中是否还有可以运行的进程。如果没有的话，调用 `idle_balance()` 从其他 CPU 获取可运行的进程。选定进程后，调用 `put_prev_task()` 告诉当前进程即将被切换出 CPU。接下来，调用 `pick_next_task()` 命令相应的调度类挑选下一个进程，并清除 `need_resched` 标志。`pick_next_task()` 会遍历全部调度类，找出拥有可运行进程的优先级最高的调度类，然后调用对应的函数。

现在，`schedule()` 检查是否找出可运行的仅仅从。如果找出的进程和当前进程相同，不进行切换，继续执行；如果找到一个新进程，会调用 `context_switch()` 进行切换，交换执行状态。

完成之后，解锁运行队列，重新允许优先级抢占。如果抢占被禁止时还有抢占发起，那么 `schedule()` 会立即执行。

在内核代码中，有三个主要时机会发生进程切换。

1. 周期性更新当前的进程

函数 `scheduler_tick()` 被时钟中断周期性调用，以更新运行队列、CPU 负载、当前进程的运行时计数器。这个函数调用 `task_tick()`，来进行周期性的进程更新。在 `task_tick()` 内部，调度类可以决定一个新进程是否要被调度，以及为进程设置 `need_resched` 来通知内核尽快调用 `sched()`。除此之外，如果内核设置了 **SMP**，还会调用负载均衡。

```
/*
 * This function gets called by the timer code, with HZ frequency.
 * We call it with interrupts disabled.
 */
void scheduler_tick(void)
{
    int cpu = smp_processor_id();
    struct rq *rq = cpu_rq(cpu);
    struct task_struct *curr = rq->curr;
    sched_clock_tick();
    raw_spin_lock(&rq->lock);
    update_rq_clock(rq);
    update_cpu_load_active(rq);
    curr->sched_class->task_tick(rq, curr, 0);
    raw_spin_unlock(&rq->lock);
    perf_event_task_tick();
#ifdef CONFIG_SMP
    rq->idle_at_tick = idle_cpu(cpu);
    trigger_load_balance(rq, cpu);
#endif
}
```

2. 当前进程需要睡眠

如果一个进程要等待事件发生，那么它会进入睡眠状态。进程创建等待队列，并把自己加入其中，然后启动一个循环来等待某个条件成真，且进程会设置为可中断（**TASK_INTERRUPTIBLE**）或者不可中断（**TASK_UNINTERRUPTIBLE**）的睡眠状态，前者可被它能够处理的信号唤醒。如果需要的事件仍然没有发生，调用 `schedule()`，然后睡眠。`schedule()` 把当前进程从队列中移出，它总在睡眠之前被调用。

```
/* 'q' is the wait queue we wish to sleep on */
DEFINE_WAIT(wait);
add_wait_queue(q, &wait);
while (!condition) /* condition is the event that we are waiting for */
{
    prepare_to_wait(&q, &wait, TASK_INTERRUPTIBLE);
    if (signal_pending(current))
        /* handle signal */
        schedule();
}
finish_wait(&q, &wait);
```

3. 唤醒睡眠进程

主要函数为 `try_to_wake_up()`。它会做三件事：（1）把将要唤醒的进程放回运行队列；（2）把任务的状态设为 **TASK_RUNNING** 来唤醒进程；（3）如果被唤醒的进程优先级更高，设置 `need_resched` 调用 `schedule()`。

这个函数在执行一些初始错误检查和 **SMP** 处理后，会调用 `ttwu_queue()` 进行唤醒工作。`ttwu_queue()` 函数锁定了运行队列并调用 `ttwu_do_activate()`，而 `ttwu_do_activate()` 又会调用 `ttwu_activate()` 执行步骤 1，而 `ttwu_do_wakeup()` 会去执行步骤 2 和 3。

`ttwu_activate()` 会执行 `enqueue_task()`，把相应进程放回运行队列。`ttwu_do_wakeup()` 会检查当前运行的进程是否要被刚唤醒的现在处于运行队列的任务抢占。其中的 `check_preempt_curr()` 函数内部最后会调用相应函数，可能会设置 `need_resched` 标志。在任务的标志被设置成 **TASK_RUNNING** 之后，唤醒过程就结束了。

```

static int
try_to_wake_up(struct task_struct *p, unsigned int state, int wake_flags)
{
    unsigned long flags;
    int cpu, success = 0;
    smp_wmb();
    raw_spin_lock_irqsave(&p->pi_lock, flags);
    if (!(p->state & state))
        goto out;
    success = 1; /* we're going to change ->state */
    cpu = task_cpu(p);
    if (p->on_rq && ttwu_remote(p, wake_flags))
        goto stat;
#ifdef CONFIG_SMP
    /*
     * If the owning (remote) cpu is still in the middle of schedule() with
     * this task as prev, wait until its done referencing the task.
     */
    while (p->on_cpu) {
#ifdef __ARCH_WANT_INTERRUPTS_ON_CTXSW
        /*
         * In case the architecture enables interrupts in
         * context_switch(), we cannot busy wait, since that
         * would lead to deadlocks when an interrupt hits and
         * tries to wake up @prev. So bail and do a complete
         * remote wakeup.
         */
        if (ttwu_activate_remote(p, wake_flags))
            goto stat;
#else
        cpu_relax();
#endif
    }
    /*
     * Pairs with the smp_wmb() in finish_lock_switch().
     */
    smp_rmb();
    p->sched_contributes_to_load = !task_contributes_to_load(p);
    p->state = TASK_WAKING;
    if (p->sched_class->task_waking)
        p->sched_class->task_waking(p);
    cpu = select_task_rq(p, SD_BALANCE_WAKE, wake_flags);
    if (task_cpu(p) != cpu) {
        wake_flags |= WF_MIGRATED;
        set_task_cpu(p, cpu);
    }
#endif /* CONFIG_SMP */
    ttwu_queue(p, cpu);
stat:
    ttwu_stat(p, cpu, wake_flags);
out:
    raw_spin_unlock_irqrestore(&p->pi_lock, flags);
    return success;
}

```

(五) CFS 调度算法

这个算法是用于一般进程的调度算法。算法的调度目标为：（1）任何进程获得的处理器时间是由它自己和其他所有可运行进程的 **nice** 值的相对差值决定的。（2）任何 **nice** 值对应的时间不是绝对时间，而是处理器的使用比。（3）**nice** 值对时间片的作用不再是算术级增加，而是几何级增加。（4）公平调度，确保每个进程有公平的处理器使用比。

这个算法的基本原理是，设定一个调度周期（**sched_latency_ns**），目标是让每个进程在这个周期内至少有机会运行一次。换一种说法就是，等待 CPU 的时间最长不超过这个调度周期。根据进程的数量，每个进程评分这个调度周期内的 CPU 使用权。由于进程的优先级（**nice** 值）不同，分割时需要加权。每个进程加权后的累计运行时间保存在自己的 **vruntime** 字段中，**哪个进程的 vruntime 最小就获得本轮运行的权利**。

为了避免频繁切换造成过大的调度开销，Linux 引入了调度延迟和最小运行时间粒度的概念。调度延迟（**TSL**）是用来计算最小任务运行时间的常量。同等级别的进程会均分这个时间；如果进程数量增加，**TSL** 不变，那么运行时长会不断减少，因此 **CSF** 引入了“最小粒度”，根据运行中进程数量来调节 **TSL**。

调度中，进程的 **vruntime** 增长速度决定了进程的优先级。时间差产生的原因是进程的

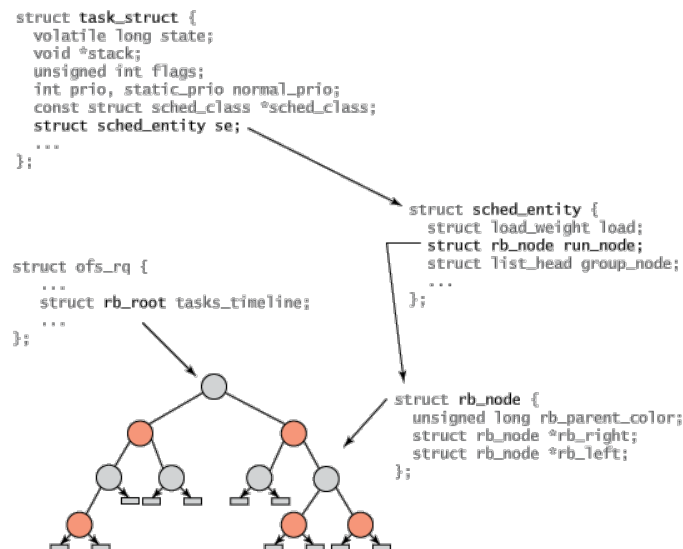
调度是根据进程的优先级决定的，也就是说，高优先级进程的 `vruntime` 增长慢于低优先级。

对于 CFS 对于 I/O 密集进程和 CPU 密集进程的处理，不妨来看这样一个例子。一个系统有一个文本编辑器进程（I/O）和视频编码器进程（CPU）。为了给用户更好的交互体验，量的调度应该给予文本编辑器更多的处理器时间。因此，我们希望文本编辑器：（1）有大量的处理器时间（2）被唤醒时可以抢占视频编码器。

CFS 是这样处理的：调度器保证文本编辑器有特定的处理器时间比例。因为文本编辑器把大部分的时间花费到了等待输入的阻塞上，所以它没有用尽自己的处理器时间比例。相对的，视频编码器能够自由使用超过分配给自己的 50% 的处理器，这就使它能够快速的完成编码工作。

为了给新进程更快的调度机会，CFS 维护了 `vruntime_min` 值，它是单调递增的，记录了运行队列中全部的进程的最小 `vruntime` 值。当新进程创建时，这个值会赋给新进程。

CFS 在运行队列中使用红黑树，操作复杂度为 $O(\log n)$ ，允许快速、高效的插入和删除节点的操作。树中每个节点代表一个进程，根据 `vruntime` 排序。因此，红黑树每个节点的左儿子指向 `vruntime` 更小的进程，它们更需要处理器。因此，缓存最左边的节点可以更快、更方便地访问节点。



考虑到服务器会生成很多进程来并行处理收到的接入连接，CFS 引入了组来描述这种行为，而不是统一的公平对待全部进程。生成子进程的服务器进程在组内（根据层级）共享他们的 `vruntime`，而单一进程会维护自己独立的 `vruntime`。在这种办法中，单一进程和组进程的调度时间粗略相等。CFS 会首先尝试对组公平，然后对组内进程公平。

CFS 中，分配给进程的运行时间计算公式为：

$$\text{分配给进程的运行时间} = \text{调度周期} * \text{进程权重} / \text{所有可运行进程权重之和}$$

`vruntime` 计算公式为：

$\begin{aligned} \text{vruntime} &= \text{实际运行时间} * \text{NICE_0_LOAD} / \text{进程权重} \\ &= \text{实际运行时间} * 1024 / \text{进程权重} \\ &\quad (\text{NICE_0_LOAD} = 1024, \text{表示nice值为0的进程权重}) \end{aligned}$	$\begin{aligned} \text{vruntime} &= \text{进程在一个调度周期内的实际运行时间} * \text{NICE_0_LOAD} / \text{进程权重} \\ &= (\text{调度周期} * \text{进程权重} / \text{所有进程总权重}) * \text{NICE_0_LOAD} / \text{进程权重} \\ &= \text{调度周期} * \text{NICE_0_LOAD} / \text{所有可运行进程总权重} \end{aligned}$
---	---

因此，进程优先级越高，权重越大，`vruntime` 增长得越慢。且一个进程在一个调度周期内的 `vruntime` 值和自己的权重无关。但是，`vruntime` 小，说明之前占用 CPU 时间短，受到了“不公平”对待。为了确保公平，我们总是选出 `vruntime` 最小的进程运行，形成一种“追赶”的局面。这样既能公平选择进程，又能保证高优先级进程获得较多的运行时间。

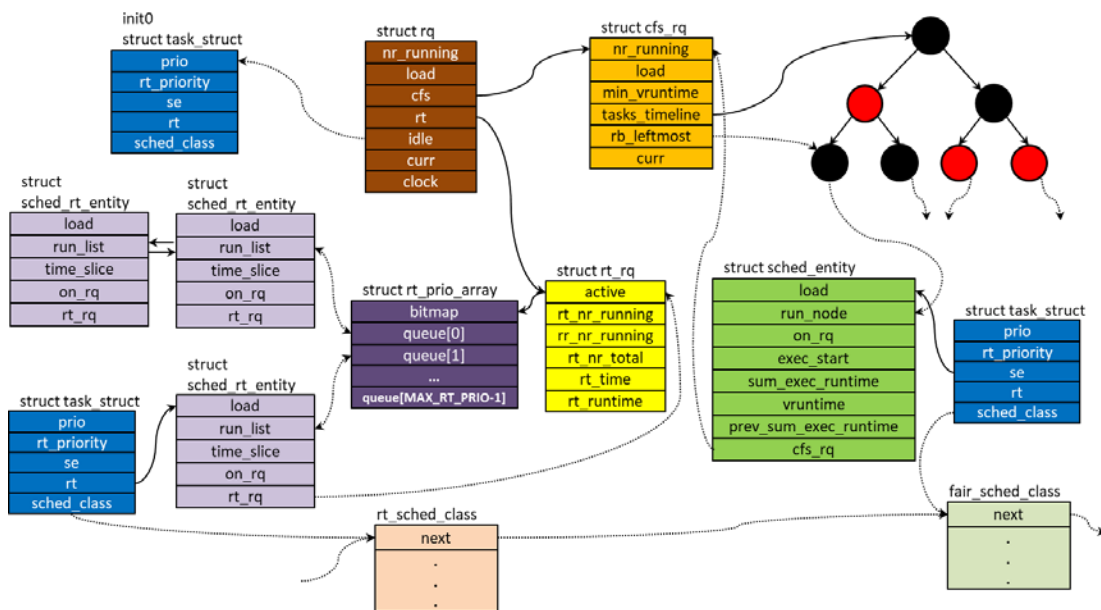
(六) 软实时调度

Linux 支持软实时任务调度，因此内核可以有效地调度有严格时间限制的进程。对应的调度类是在 kernel/sched_rt.c 中实现的 `rt_sched_class`。RT 进程比 CFS 调度的进程优先级更高。

由 RT 调度器处理的进程可以配置成两种不同的模式：（1）`SCHED_FIFO`，先入先出模式调度，没有时间片的概念，只会在进程终止或者自己放弃处理器时被切换；（2）`SCHED_RR`，每个进程运行固定的时间片，然后按照循环方式被相同优先级的进程抢占。时间片用完之前，进程也可能被高优先级的进程抢占。`SCHED_RR` 实现了“理想中的完全公平”——每个进程平均地使用 CPU

RT 调度类中，也使用了不同优先级的运行队列，添加进程、删除进程、查找最高优先级的进程操作都可以在 $O(1)$ 时间内完成。

单核视角下，调度类数据结构间的关联如下：



Exercise2

这一部分要求阅读五份源代码，下面我将逐一分析各部分的具体内容。

(一) code/threads/scheduler.h 和 code/threads/scheduler.cc

这两份代码文件中包含了 Scheduler 这一调度类的声明和实现。scheduler.h 中进行了类的声明，scheduler.cc 中实现了这个类的成员方法。

```
class Scheduler {
public:
    Scheduler(); // Initialize list of ready threads
    ~Scheduler(); // De-allocate ready list

    void ReadyToRun(Thread* thread); // Thread can be dispatched.
    Thread* FindNextToRun(); // Dequeue first thread on the ready
                             // list, if any, and return thread.

    void Run(Thread* nextThread); // Cause nextThread to start running
    void Print(); // Print contents of ready list

private:
    List *readyList; // queue of threads that are ready to run,
                    // but not running
};
```

Scheduler 中，只有一个私有成员变量 readyList，代表就绪队列，储存 RUNNABLE 的线程。其余的 public 成员方法均实现在 scheduler.cc 中。下面让我们来逐一分析其功能。

1.构造函数和析构函数

构造函数初始化就绪队列，析构函数删除当前的就绪队列。就绪队列的类型是 Nachos 自己的 List 类型。

```
//-----
// Scheduler::Scheduler
// Initialize the list of ready but not running threads to empty.
//-----

Scheduler::Scheduler()
{
    readyList = new List;
}

//-----
// Scheduler::~Scheduler
// De-allocate the list of ready threads.
//-----

Scheduler::~Scheduler()
{
    delete readyList;
}
```

2.ReadyToRun

把参数中传入的线程的状态标记为 READY，表示当前线程处于就绪状态，可以准备运行了。然后，把这个线程插入就绪队列的队尾（readyList->Append）。

```
//-----
// Scheduler::ReadyToRun
// Mark a thread as ready, but not running.
// Put it on the ready list, for later scheduling onto the CPU.
// "thread" is the thread to be put on the ready list.
//-----

void
Scheduler::ReadyToRun (Thread *thread)
{
    DEBUG('t', "Putting thread %s on ready list.\n", thread->getName());

    thread->setStatus(READY);
    readyList->Append((void *)thread);
}
```

3.FindNextToRun

这个函数从就绪队列中取出将要调度的线程，并把它从队列中移出。目前的实现是直接调用 List 类的 Remove 方法，这个方法取出队首元素，并将其从队列中删除。如果 List 为空，则返回空指针 NULL。

```
//-----
// Scheduler::FindNextToRun
// Return the next thread to be scheduled onto the CPU.
// If there are no ready threads, return NULL.
// Side effect:
// Thread is removed from the ready list.
//-----

Thread *
Scheduler::FindNextToRun ()
{
    return (Thread *)readyList->Remove();
}
```

4.Run

这个函数把 CPU 调度给下一个线程（参数中的 nextThread）。

首先，函数对当前的线程进行处理：如果当前线程是用户级程序，那么储存它的用户状态（寄存器）和空间状态，然后调用线程类中的 CheckOverflow()函数检查是否有栈溢出。

```

    Thread *oldThread = currentThread;

#ifdef USER_PROGRAM
    if (currentThread->space != NULL) { // ignore until running user programs
        // if this thread is a user program,
        currentThread->SaveUserState(); // save the user's CPU registers
        currentThread->space->SaveState();
    }
#endif

    oldThread->CheckOverflow(); // check if the old thread
                              // had an undetected stack overflow

```

如果一切正常，那么把 `currentThread` 切换到下一个线程，同时设置其状态为 `RUNNING`，再调用 `SWITCH` 汇编函数，来进行上下文切换。

```

DEBUG('t', "Switching from thread \"%s\" to thread \"%s\"",
      oldThread->getName(), nextThread->getName());

// This is a machine-dependent assembly language routine defined
// in switch.s. You may have to think
// a bit to figure out what happens after this, both from the point
// of view of the thread and from the perspective of the "outside world".

SWITCH(oldThread, nextThread);

DEBUG('t', "Now in thread \"%s\"", currentThread->getName());

```

上下文切换完成后，由于先前的线程可能是因为执行完程序，进入终止状态而选择调度，我们还需要检查待销毁线程指针是否为空，如果不为空，需要把它指向的线程对象 `delete` 掉。

```

if (threadToBeDestroyed != NULL) {
    delete threadToBeDestroyed;
    threadToBeDestroyed = NULL;
}

```

最后，如果新线程也运行用户级程序，那么需要恢复它的状态。

```

#ifdef USER_PROGRAM
    if (currentThread->space != NULL) {
        currentThread->RestoreUserState();
        currentThread->space->RestoreState();
    }
#endif

```

5.Print

这个函数打印当前就绪队列中所有线程的状态。`List` 类中的 `Mapcar` 方法可以对其中的每个成员调用参数中传入的函数，因此会按顺序打印线程信息。

```

void
Scheduler::Print()
{
    printf("Ready list contents:\n");
    readyList->Mapcar((VoidFunctionPtr) ThreadPrint);
}

```

6.小结

以上我们分析了 `Nachos` 中的调度类。可以看到，`Nachos` 目前的调度实现是简单的 `FIFO`，且没有优先级机制，需要等待当前线程自己让出 `CPU` 或者执行完毕以后才会进行切换。

（二）code/threads/switch.s

这一部分实现了上下文切换的汇编代码。由于上下文切换的工作因体系结构的不同会有所出入，这份代码实现了 `MIPS`、`SUN`、`PA-RISC`、`I386` 下的上下文切换。总体来说，切换的主入口是 `SWITCH`，主要工作流程为：保存原线程的上下文（压入栈中）——载入新线程的上下文——跳转到新线程对应的入口处，准备执行新线程。

除此之外，`switch.S` 中还有 `ThreadRoot` 汇编函数，在第一次启动线程的时候使用。

(三) code/machine/timer.h 和 code/machine/timer.cc

这两份代码中主要实现了对时钟硬件的仿真。时钟生成周期性信号，并周期性触发时钟中断。它可以用于实现时间片机制，也可以为线程的睡眠统计时长。Nachos 使用的时钟模拟由 Timer 类实现：

```
class Timer {
public:
    Timer(VoidFunctionPtr timerHandler, int callArg, bool doRandom);
    // Initialize the timer, to call the interrupt
    // handler "timerHandler" every time slice.

    ~Timer() {}

    // Internal routines to the timer emulation -- DO NOT call these

    void TimerExpired(); // called internally when the hardware
                        // timer generates an interrupt

    int TimeOfNextInterrupt(); // figure out when the timer will generate
                            // its next interrupt

private:
    bool randomize; // set if we need to use a random timeout delay
    VoidFunctionPtr handler; // timer interrupt handler
    int arg; // argument to pass to interrupt handler
};
```

其中包含三个私有成员：randomize 用于引入随机化机制。如果这个变量被设置为 true，那么时钟中断会在随机的 tick 之后执行，这允许了时间片的随机长度。handler 是一个函数指针，指向时钟中断处理函数。arg 则是传递给时钟中断处理函数的参数。

Timer 类里还包含 4 个公有方法：

1.构造函数和析构函数

Timer 的构造函数会设置是否执行随机化时间片机制，设置时钟中断处理函数以及它的参数。初始化参数以后，函数会调用 interrupt 的 Schedule 方法来规划第一个来自时钟设备的中断。Timer 的析构函数为空函数。

```
Timer::Timer(VoidFunctionPtr timerHandler, int callArg, bool doRandom)
{
    randomize = doRandom;
    handler = timerHandler;
    arg = callArg;

    // schedule the first interrupt from the timer device
    interrupt->Schedule(timerHandler, (int) this, TimeOfNextInterrupt(),
                        TimerInt);
}
```

2.TimerExpired

这个函数用于仿真硬件生成的时钟中断。它首先规划下一次时钟中断的时间，然后执行时钟中断处理函数。

```
void
Timer::TimerExpired()
{
    // schedule the next timer device interrupt
    interrupt->Schedule(timerHandler, (int) this, TimeOfNextInterrupt(),
                        TimerInt);

    // invoke the Nachos interrupt handler for this device
    (*handler)(arg);
}
```

3.TimeOfNextInterrupt

这个函数给出下一次执行时钟中断的时间。如果开启随机化机制，函数返回一个范围在[1, 2*TimeTicks]的随机数，否则以 TimeTicks 的固定时间间隔执行时钟中断。

```

int
Timer::TimeOfNextInterrupt()
{
    if (randomize)
        return 1 + (Random() % (TimerTicks * 2));
    else
        return TimerTicks;
}

```

除了三个成员函数以外，timer.cc 中还包含一个 dummy function——TimerHandler。由于 C++ 不允许使用成员函数做函数指针，因此使用这个函数包装 TimerExpired 成员函数，来作为 Schedule 函数的参数使用。

```

static void TimerHandler(int arg)
{ Timer *p = (Timer *)arg; p->TimerExpired(); }

```

Exercise3

这一部分要求我们实现基于优先级的抢占式调度算法。

Exercise2 中，我们分析了目前 Nachos 的线程调度机制。我们可以发现，目前的线程没有优先级机制，因此我们首先需要添加优先级机制。这一过程有**三步**：添加私有成员和 setter-getter 方法——修改构造函数——测试效果。

首先添加成员变量和 setter-getter 方法，代码如下：

```

class Thread {
private:
    // NOTE: DO NOT CHANGE the order of these first two members.
    // THEY MUST be in this position for SWITCH to work.
    int* stackTop;           // the current stack pointer
    void *machineState[MachineStateSize]; // all registers except for stackTop

    /***** added by Li Cong 1800012826 *****/

    int UID; // User ID
    int TID; // Thread ID

    int priority; // Lab2, thread priority for scheduling

public:
    int getUID();
    int getTID();
    void setUID(int newUID);

    int getPriority();
    void setPriority(int p);

    ThreadStatus getStatus();

    Thread(char* debugName, int p);

//-----
// Thread::getPriority
// Get thread's scheduling priority.
//-----

int
Thread::getPriority(){return priority;}

//-----
// Thread::setPriority
// Set thread's scheduling priority.
//-----

void
Thread::setPriority(int p){priority = p;}

```

这里由于还不知道优先级是否有规定，故未对优先级设置范围限制。

下一步，我们需要修改构造函数和析构函数。实际上，我们只需要对构造函数进行修饰。首先，对于初始的构造函数，我们应该添加优先级的初始化值；其次，我们还应该允许构造时设置优先级，因此需要修改原构造函数，添加新构造函数，代码如下：

```

Thread::Thread(char* threadName)
{
    name = threadName;
    stackTop = NULL;
    stack = NULL;
    status = JUST_CREATED;
#ifdef USER_PROGRAM
    space = NULL;
#endif

    /***** added by Li Cong 1800012826 *****/
    this->priority = 0; // for lab2
    this->TID = -1;
    for(int i=0; i<MAX_THREAD_NUM; ++i)
    {
        if(!used_TID[i])
        {
            this->TID = i;
            used_TID[i] = true;
            Thread_Pointer[i] = this;
            break;
        }
    }
    this->UID = 0;
    if(this->TID==-1)
    {
        printf("Assertion failure: reached max thread number!\n");
    }
    ASSERT(this->TID>=0 && this->TID<MAX_THREAD_NUM);

    /***** added by Li Cong 1800012826 *****/
}

//-----for lab 2-----
// Thread::Thread
// Initialize a thread control block, so that we can then call
// Thread::Fork.
//
// "threadName" is an arbitrary string, useful for debugging.
// "p" initializes the priority of current thread
//-----for lab 2-----

Thread::Thread(char* threadName, int p)
{
    name = threadName;
    stackTop = NULL;
    stack = NULL;
    status = JUST_CREATED;
#ifdef USER_PROGRAM
    space = NULL;
#endif

    this->priority = p; // for lab2

    this->TID = -1;
    for(int i=0; i<MAX_THREAD_NUM; ++i)
    {
        if(!used_TID[i])
        {
            this->TID = i;
            used_TID[i] = true;
            Thread_Pointer[i] = this;
            break;
        }
    }
    this->UID = 0;
    if(this->TID==-1)
    {
        printf("Assertion failure: reached max thread number!\n");
    }
    ASSERT(this->TID>=0 && this->TID<MAX_THREAD_NUM);
}

```

下面对优先级的添加进行测试。首先修改 Lab1 中的 TS，使得它可以显示优先级：

```

void
TS()
{
    DEBUG('t', "Entering TS.");

    char *State2String[] = {"JUST_CREATED", "RUNNING", "READY", "BLOCKED"};

    printf("UID\tTID\tPRI\tNAME\tSTAT\n");
    for(int i=0; i<MAX_THREAD_NUM; ++i)
    {
        if(used_TID[i])
        {
            printf("%d\t%d\t%d\t%s\t%s\n", Thread_Pointer[i]->getUID(), Thread_Pointer[i]->getTID(), Thread_Pointer[i]->getPriority(), Thread_Pointer[i]->getName(),
                State2String[Thread_Pointer[i]->getStatus()]);
        }
    }
}

```

接下来编写测试函数。这里用到了 Lab1 编写的 SetStatus 函数来为线程设置状态。为了能够在 TS 中显示所有的测试进程，因此把它们全部阻塞住。测试函数与命令行结果如下。可以看到，setter-getter 方法、构造函数都可以正常运行。

```

void
Lab2_3_1_Test()
{
    DEBUG('t', "Entering Lab2_3_1_Test");

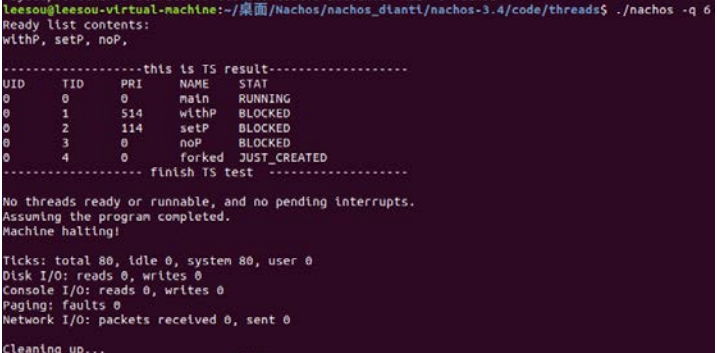
    Thread *t1 = new Thread("withP", 514);
    t1->Fork(setStatus, (void *)1);
    Thread *t2 = new Thread("setP");
    t2->setPriority(114);
    t2->Fork(setStatus, (void *)1);
    Thread *t3 = new Thread("noP");
    t3->Fork(setStatus, (void *)1);

    Thread *t4 = new Thread("forked");

    setStatus(0);

    printf("\n\n-----this is TS result-----\n");
    TS();
    printf("----- finish TS test ----- \n\n");
}

```



```

leesoul@leesoul-virtual-machine:~/桌面/nachos/nachos_dianti/nachos-3.4/code/threads$ ./nachos -q 6
Ready list contents:
withP, setP, noP,

-----this is TS result-----
UID  TID  PRI  NAME  STAT
0     0    0   main  RUNNING
0     1   514  withP  BLOCKED
0     2   114  setP   BLOCKED
0     3    0   noP   BLOCKED
0     4    0  forked JUST_CREATED
----- finish TS test -----

No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 80, idle 0, system 80, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
Cleaning up...

```

接下来我们便可以考虑实现基于优先级的抢占式调度算法。

注意到，选择新线程的函数中，原来的实现只是简单地选择队首元素。又注意到，List 类中有 SortedInsert 函数：

```

void
List::SortedInsert(void *item, int sortKey)
{
    ListElement *element = new ListElement(item, sortKey);
    ListElement *ptr;          // keep track

    if (IsEmpty()) {          // if list is empty, put
        first = element;
        last = element;
    } else if (sortKey < first->key) {
        // item goes on front of list
        element->next = first;
        first = element;
    } else {                  // look for first elt in list bigger than item
        for (ptr = first; ptr->next != NULL; ptr = ptr->next) {
            if (sortKey < ptr->next->key) {
                element->next = ptr->next;
                ptr->next = element;
                numInList++;
                return;
            }
        }
        last->next = element;          // item goes at end of list
        last = element;
    }
    numInList++;
}

```

它可以实现由根据一个 SortedKey 小到大排序 List 中的元素，因此我们可以规定，priority 越小，优先级越高，且把 Scheduler 类中的 ReadyToRun 方法中的 Append 修改成 SortedInsert，具体修改如下：

```

void
Scheduler::ReadyToRun (Thread *thread)
{
    DEBUG('t', "Putting thread %s on ready list.\n", thread->getName());

    thread->setStatus(READY);

    // change for lab2
    //readyList->Append((void *)thread);
    readyList->SortedInsert((void *)thread, thread->getPriority());
}

```


这样，每次调度时，便会按照等待队列中优先级顺序来进行调度了。下面来编写测试程序验证这一修改的正确性，代码如下：

```
void
Lab2_3_2_Test()
{
    DEBUG('t', "Entering Lab2_3_2_Test");

    Thread *t1 = new Thread("low", 810);
    Thread *t2 = new Thread("mid", 514);
    Thread *t3 = new Thread("high", 114);

    t1->Fork(SetStatus, (void *)0);
    t2->Fork(SetStatus, (void *)0);
    t3->Fork(SetStatus, (void *)0);

    SetStatus(0);

    printf("\n\n-----this is TS result-----\n");
    TS();
    printf("----- finish TS test ----- \n\n");
}
```

其中使用的 SetStatus 与之前相比修改为如下内容（增加了一些输出）：

```
void
SetStatus(int which)
{
    printf("***** current thread (uid=%d, tid=%d, priority=%d, name=%s) => \n", currentThread->getUID(), currentThread->getTID(), currentThread->getPriority(), currentThread->getName());
    switch(which)
    {
        case 0:
            scheduler->Print();
            printf("\n\n");
            currentThread->Yield();
            break;
        case 1:
            intStatus oldlevel = interrupt->SetLevel(IntOff);
            currentThread->Sleep();
            (void) interrupt->SetLevel(oldlevel);
            break;
        case 2:
            currentThread->Finish();
            break;
        default:
            currentThread->Yield();
            break;
    }
    printf("thread %d finished\n\n", currentThread->getTID());
}
```

运行测试代码，结果如下：

```
leesou@leesou-virtual-machine:~/桌面/Nachos/nachos_dianti/nachos-3.4/code/threads$ ./nachos -q 7
***** current thread (uid=0, tid=0, priority=0, name=main) =>
Ready list contents:
high, mid, low,

***** current thread (uid=0, tid=3, priority=114, name=high) =>
Ready list contents:
main, mid, low,

thread 0 finished

-----this is TS result-----
UID   TID   PRI   NAME   STAT
0      0      0    main  RUNNING
0      1     810    low   READY
0      2     514    mid   READY
0      3     114    high  READY
----- finish TS test -----

thread 3 finished

***** current thread (uid=0, tid=2, priority=514, name=mid) =>
Ready list contents:
low,

***** current thread (uid=0, tid=1, priority=810, name=low) =>
Ready list contents:
mid,

thread 2 finished
thread 1 finished

No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 110, idle 0, system 110, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0
```

可以看到，系统先执行 main 线程，main 创建三个新线程后调用 SetStatus，让出 CPU 给就绪队列中优先级最高的线程，这个线程执行 SetStatus 函数，再次让出 CPU，执行调度后，main 从 SetStatus 中返回，打印线程表，结束执行。这时，调度器寻找就绪队列中优先级最高的线程，继续运行，这个线程输出结束语句后终止运行，然后调度器继续调度，与上面的过程相似，直到所有线程执行完毕。注意，每个线程是在 SetStatus 内部保存上下文的，所以 thread x finished 需要等待再次切换回当前线程后才会输出。

每次触发调度时，总会选择优先级最高的那个线程进行切换。在这里，为了维持一定的公平性，当优先级最高的线程执行 yield 时，我的设计仍然是从调度队列中选择优先级最高的那个线程，以避免一直执行这个最高优先级的线程。（由于目前没有对优先级高低的明确规定，故暂时如此设计。想要始终执行所有线程中优先级最高的那个，需要修改 yield 函数中的语句顺序）

更进一步地，由于抢占式的要求，当更高优先级的线程插入就绪队列时，应该立即进行线程切换。因此，对 ReadyToRun 再次修改：

```
void
Scheduler::ReadyToRun (Thread *thread)
{
    DEBUG('t', "Putting thread %s on ready list.\n", thread->getName());

    thread->setStatus(READY);

    // change for lab2
    //readyList->Append((void *)thread);
    readyList->SortedInsert((void *)thread, thread->getPriority());
    if(thread->getPriority() > currentThread->getPriority())
    {
        currentThread->Yield();
    }
}
```

对这部分代码测试如下：

```
void
p3(int which)
{
    for(int i=0; i<5; ++i)
    {
        printf("***** thread name %s priority %d looped %d times\n", currentThread->getName(), currentThread->getPriority(), i);
    }
}

void
p2(int which)
{
    for(int i=0; i<5; ++i)
    {
        printf("***** thread name %s priority %d looped %d times\n", currentThread->getName(), currentThread->getPriority(), i);
        if(i==0)
        {
            Thread *t3 = new Thread("mid", 514);
            t3->Fork(p3, (void *)0);
        }
    }
}

void
p1(int which)
{
    for(int i=0; i<5; ++i)
    {
        printf("***** thread name %s priority %d looped %d times\n", currentThread->getName(), currentThread->getPriority(), i);
        if(i==0)
        {
            Thread *t2 = new Thread("high", 114);
            t2->Fork(p2, (void *)0);
        }
    }
}

void
Lab2_3_2_Test1()
{
    DEBUG('t', "Entering Lab2_3_2_Test1");

    Thread *t1 = new Thread("low", 810);

    t1->Fork(p1, (void *)0);

    printf("\n\n-----this is TS result-----\n\n");
    TS();
    printf("----- finish TS test ----- \n\n");
}
```

第一个线程优先级最低，它执行 5 次循环，第一次循环时创建一个高优先级线程。高优先级线程也是循环 5 次，第一次循环创建一个中优先级线程。main 线程优先级最高。运行结果如下：

```
leesou@leesou-virtual-machine:~/桌面/Nachos/nachos_dianti/nachos-3.4/code/threads$ ./nachos -rs -q 8

-----this is TS result-----
UID      TID      PRI      NAME      STAT
0         0         0      main      RUNNING
0         1        810     low       READY
----- finish TS test -----

**** thread name low priority 810 looped 0 times
**** thread name high priority 114 looped 0 times
**** thread name high priority 114 looped 1 times
**** thread name high priority 114 looped 2 times
**** thread name high priority 114 looped 3 times
**** thread name high priority 114 looped 4 times
**** thread name mid priority 514 looped 0 times
**** thread name mid priority 514 looped 1 times
**** thread name mid priority 514 looped 2 times
**** thread name mid priority 514 looped 3 times
**** thread name mid priority 514 looped 4 times
**** thread name low priority 810 looped 1 times
**** thread name low priority 810 looped 2 times
**** thread name low priority 810 looped 3 times
**** thread name low priority 810 looped 4 times
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 184, idle 114, system 70, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

可以看到，low 线程创建 high 线程后，high 的优先级更高，会抢占 low 的运行；high 创建 mid 后，mid 优先级低，不会抢占。high 执行完后，根据调度，会先运行 mid，最后再运行 low。而 main 由于优先级最高，会最先运行完毕（输出线程表后退出）。

至此，抢占式调度算法实现完成。

Challenge

这一部分我选择实现 Round Robin 调度机制。

在实现之前，首先来分析一下目前 Nachos 的时钟中断机制。在分析源代码后可以发现，时钟中断由 Interrupt 类中的 OneTick 方法来控制。这个方法会更新全局对象 stats 中的各个 Ticks 变量，来模拟时钟进程。

```
void
Interrupt::OneTick()
{
    MachineStatus old = status;

    // advance simulated time
    if (status == SystemMode) {
        stats->totalTicks += SystemTick;
        stats->systemTicks += SystemTick;
    } else {
        stats->totalTicks += UserTick;
        stats->userTicks += UserTick;
    }
    DEBUG('i', "\n== Tick %d ==\n", stats->totalTicks);

    // check any pending interrupts are now ready to fire
    ChangeLevel(IntOn, IntOff);
    // first, turn off interrupts
    // (interrupt handlers run with
    // interrupts disabled)
    while (CheckIfDue(FALSE))
        ;
    // check for pending interrupts
    ChangeLevel(IntOff, IntOn);
    // re-enable interrupts
    // if the timer device handler asked
    // for a context switch, ok to do it now
    if (yieldOnReturn) {
        yieldOnReturn = FALSE;
        status = SystemMode;
        currentThread->Yield();
        status = old;
    }
}
```

这个函数中，标黄色的变量 `yieldOnReturn` 由 Nachos 的时钟中断处理函数设置。这个变量初始为 `false`，当触发时钟中断时，处理函数会把这个变量变为 `true`，这样，当调用 `OnTick` 更新时钟数时，便会触发 `Yield` 函数实现时钟中断带来的线程切换，以模拟时间片的效果。

```
static void
TimerInterruptHandler(int dummy)
{
    if (interrupt->getStatus() != IdleMode)
        interrupt->YieldOnReturn();
}

void
Interrupt::YieldOnReturn()
{
    ASSERT(inHandler == TRUE);
    yieldOnReturn = TRUE;
}
```

除此之外，注释中说明，`OneTick` 会在两种情形下被调用：（1）重新恢复中断响应（`Interrupt::SetLevel()`）（2）用户指令被执行。因此，目前只有 `Fork`，`Yield`，`Finish` 的时候会调用这个函数，因为它们进行了中断的 `disable` 和 `enable`。

```
//-----
// Interrupt::OneTick
// Advance simulated time and check if there are any pending
// interrupts to be called.
//
// Two things can cause OneTick to be called:
//   interrupts are re-enabled
//   a user instruction is executed
//-----
```

更进一步地，为了实现时间片的随机时长分配，Nachos 原有的机制是，需要在运行时加上 `-rs` 命令，这样会把 `system.cc` 的 `Initialize` 函数中的 `randomYield` 设为 `true`，并创建一个 `Timer` 类，来控制时间片的分配。

```
if (randomYield)
    timer = new Timer(TimerInterruptHandler, 0, randomYield); // start the timer
```

默认的固定时长定义在 `machine/stats.h` 中，由 `Statistics` 类来维护。`threads/system.cc` 中声明了一个全局变量

```
class Statistics {
public:
    int totalTicks; // Total time running Nachos
    int idleTicks; // Time spent idle (no threads to run)
    int systemTicks; // Time spent executing system code
    int userTicks; // Time spent executing user code
                  // (this is also equal to # of
                  // user instructions executed)

    int numDiskReads; // number of disk read requests
    int numDiskWrites; // number of disk write requests
    int numConsoleCharsRead; // number of characters read from the keyboard
    int numConsoleCharsWritten; // number of characters written to the display
    int numPageFaults; // number of virtual memory page faults
    int numPacketsSent; // number of packets sent over the network
    int numPacketsRecv; // number of packets received over the network

    Statistics(); // initialize everything to zero
    void Print(); // print collected statistics
};

// Constants used to reflect the relative time an operation would
// take in a real system. A "tick" is a just a unit of time -- if you
// like, a microsecond.
//
// Since Nachos kernel code is directly executed, and the time spent
// in the kernel measured by the number of calls to enable interrupts,
// these time constants are none too exact.

#define UserTick 1 // advance for each user-level instruction
#define SystemTick 10 // 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 100 // (average) time between timer interrupts

extern Statistics *stats; // performance metrics
```

因此，我们要做的主要是：（1）时间片用尽时调用 `Yield`（2）启动一个定时器，不断地关闭---重启中断响应，以调用 `OneTick()`，推动系统时钟的前进。

为了实现 `Round Robin` 算法，我们首先需要修改 `Scheduler` 类，添加一个 `prevTick` 变量来记录上次切换的 `Tick` 标号。除此之外，还要实现一个 `Round Robin` 的 `Handler` 函数。

```
public:
    int prevTick; // used for lab2 challenge--RR algorithm

};

extern void RRHandler(int dummy); // implement RR algorithm
```

RRHandler 的实现如下:

```
static void
RRHandler(int dummy)
{
    int duration = stats->totalTicks - scheduler->prevTick;
    printf("Timer interrupt's duration is %d\n", duration);
    if(duration >= TimerTicks)
    {
        if(interrupt->getStatus() != IdleMode) // ready queue is not empty
        {
            printf("Time up and switch!\n");
            scheduler->Print();
            printf("\n");
            interrupt->YieldOnReturn();
            scheduler->prevTick = stats->totalTicks;
        }
        else
        {
            printf("readyList is empty!\n");
        }
    }
    else
    {
        printf("\n");
    }
}
```

这个函数类似于先前的 TimerHandler。当触发中断时,检查是否到达运行时长。如果到达了且就绪队列不为空,那么标记 yieldOnReturn 变量,等待 OnTick 函数切换线程,同时更新 prevTick 以记录最后一次切换的 TimeTick,用于后续的时间片计算;否则输出对应的状态信息。由于 Timer 类的构造方法要求处理函数必须有一个参数,所以记为 dummy(与 Nachos 的无用变量名称保持一致)。

下一步,为 RR 调度算法设置命令行选项。-rr 参数代表执行 RR 调度算法,相应地修改 Initialize 函数如下:

```
bool randomYield = FALSE;
bool RoundRobbin = FALSE;

/***** added by Li Cong 1800012826 *****/

for(int i=0; i<MAX_THREAD_NUM; ++i)
{
    used_TID[i] = false;
    Thread_Pointer[i] = NULL;
}

/***** added by Li Cong 1800012826 *****/

#ifdef USER_PROGRAM
bool debugUserProg = FALSE; // single step user program
#endif
#ifdef FILESYS_NEEDED
bool format = FALSE; // format disk
#endif
#ifdef NETWORK
double rely = 1; // network reliability
int netname = 0; // UNIX socket name
#endif

for (argc--, argv++; argc > 0; argc -= argCount, argv += argCount) {
    argCount = 1;
    if (!strcmp(*argv, "-d")) {
        if (argc == 1)
            debugArgs = "+"; // turn on all debug flags
        else {
            debugArgs = *(argv + 1);
            argCount = 2;
        }
    } else if (!strcmp(*argv, "-rs")) {
        ASSERT(argc > 1);
        RandomInit(atoi(*(argv + 1))); // initialize pseudo-random
                                        // number generator
        randomYield = TRUE;
        argCount = 2;
    }

    if (!strcmp(*argv, "-rr")) // added for Round Robbin algorithm
        RoundRobbin = TRUE;
}

if (RoundRobbin) // timer device for Round Robbin
    timer = new Timer(RRHandler, 0, false);
```

对于 Round Robbin 算法，由于每个时间片长度相同，故 Timer 构造函数中的第三个参数为 false，前面实现的 RRHandler 函数作为 Timer 的中断处理函数。

除此之外，由于 Round Robbin 是轮盘式调度，为了与先前的抢占式调度兼容，我们需要把 RoundRobbin 变量声明为全局变量，并且修改 Scheduler 的 ReadyToRun 函数：

```
void
Scheduler::ReadyToRun (Thread *thread)
{
    DEBUG('t', "Putting thread %s on ready list.\n", thread->getName());

    thread->setStatus(READY);

    // change for lab2
    if (RoundRobbin)
    {
        readyList->Append((void *)thread);
    }
    else
    {
        readyList->SortedInsert((void *)thread, thread->getPriority());
        if (thread->getPriority() < currentThread->getPriority())
        {
            currentThread->Yield();
        }
    }
}
```

现在，Round Robbin 算法已经基本实现，我们可以编写时钟模拟函数和测试函数来检验实现的正确性，测试代码及输出如下：

```
void
ClockSimulator(int loops)
{
    for (int i=0; i<loops*SystemTick; i++) {
        printf("*** thread with running time %d looped %d times (stats->totalTicks: %d)\n", loops, i+1, stats->totalTicks);
        scheduler->Print();
        printf("\n");
        interrupt->SetLevel(IntOff);
        interrupt->SetLevel(IntOn);
    }
    currentThread->Finish();
}

// -----
// Lab2_3_ch_Test
// test function for challenge in lab2
// -----

void
Lab2_3_ch_Test()
{
    DEBUG('t', "Entering Lab2_3_ch_Test");

    Thread *t1 = new Thread("low", 810);
    Thread *t2 = new Thread("mid", 514);
    Thread *t3 = new Thread("high", 114);

    t1->Fork(ClockSimulator, (void *)3);
    t2->Fork(ClockSimulator, (void *)2);
    t3->Fork(ClockSimulator, (void *)4);

    printf("\n\n-----this is TS result-----\n");
    TS();
    printf("----- finish TS test ----- \n\n");

    ClockSimulator(1);
}
```

为了便于观察轮盘的表现，我把四个线程的循环次数设置为互不相同的次数。可以看到，每个线程都不断地请求时钟中断响应，直到时间片用尽，调度函数才会轮盘式地选择并切换下一个线程。注意到 main 线程在开始执行 ClockSimulator 前有 40 个 ticks 的时间片消耗，因为 main 线程调用 initialize 函数的时候，会进行一次中断的 disable—enable 切换，且 main 线程调用三次 Fork 时，又会进行三次中断的 disable—enable 切换。每次会消耗 10 个 tick 的时长。可见，每个线程都会使用相同次数的 tick（Nachos 默认的宏定义为 100），然后激发时钟中断时，中断处理函数会轮盘式地逐个选择下一个执行的线程。循环结束时，由于调用了 Finish 函数，还会跳过 10 个 tick 的时钟周期。可见，上述 Round Robbin 机制实现应该是正确的。


```

leesou@leesou-virtual-machine:~/桌面/Nachos/nachos_diantl/nachos-3.4/code/threads$ ./nachos -rr -q 9
-----this is TS result-----
UID    TID    PRI    NAME    STAT
0       0       0      main    RUNNING
0       1      810    low     READY
0       2      514    mid     READY
0       3      114    high    READY
----- finish TS test -----

*** thread with running time 1 looped 1 times (stats->totalTicks: 40)
Ready list contents:
low, mid, high,
*** thread with running time 1 looped 2 times (stats->totalTicks: 50)
Ready list contents:
low, mid, high,
*** thread with running time 1 looped 3 times (stats->totalTicks: 60)
Ready list contents:
low, mid, high,
*** thread with running time 1 looped 4 times (stats->totalTicks: 70)
Ready list contents:
low, mid, high,
*** thread with running time 1 looped 5 times (stats->totalTicks: 80)
Ready list contents:
low, mid, high,
*** thread with running time 1 looped 6 times (stats->totalTicks: 90)
Ready list contents:
low, mid, high,
Timer interrupt's duration is 100
Time up and switch!
Ready list contents:
low, mid, high,
*** thread with running time 3 looped 1 times (stats->totalTicks: 110)
Ready list contents:
mid, high, main,
*** thread with running time 3 looped 2 times (stats->totalTicks: 120)
Ready list contents:
mid, high, main,
*** thread with running time 3 looped 3 times (stats->totalTicks: 130)
Ready list contents:
mid, high, main,
*** thread with running time 3 looped 4 times (stats->totalTicks: 140)
Ready list contents:
mid, high, main,
*** thread with running time 3 looped 5 times (stats->totalTicks: 150)

Ready list contents:
mid, high, main,
*** thread with running time 3 looped 6 times (stats->totalTicks: 160)
Ready list contents:
mid, high, main,
*** thread with running time 3 looped 7 times (stats->totalTicks: 170)
Ready list contents:
mid, high, main,
*** thread with running time 3 looped 8 times (stats->totalTicks: 180)
Ready list contents:
mid, high, main,
*** thread with running time 3 looped 9 times (stats->totalTicks: 190)
Ready list contents:
mid, high, main,
Timer interrupt's duration is 100
Time up and switch!
Ready list contents:
mid, high, main,
*** thread with running time 2 looped 1 times (stats->totalTicks: 210)
Ready list contents:
high, main, low,
*** thread with running time 2 looped 2 times (stats->totalTicks: 220)
Ready list contents:
high, main, low,
*** thread with running time 2 looped 3 times (stats->totalTicks: 230)
Ready list contents:
high, main, low,
*** thread with running time 2 looped 4 times (stats->totalTicks: 240)
Ready list contents:
high, main, low,
*** thread with running time 2 looped 5 times (stats->totalTicks: 250)
Ready list contents:
high, main, low,
*** thread with running time 2 looped 6 times (stats->totalTicks: 260)
Ready list contents:
high, main, low,
*** thread with running time 2 looped 7 times (stats->totalTicks: 270)
Ready list contents:
high, main, low,
*** thread with running time 2 looped 8 times (stats->totalTicks: 280)
Ready list contents:
high, main, low,
*** thread with running time 2 looped 9 times (stats->totalTicks: 290)
Ready list contents:
high, main, low,
Timer interrupt's duration is 100

```



```

Time up and switch!
Ready list contents:
high, main, low,
*** thread with running time 4 looped 1 times (stats->totalTicks: 310)
Ready list contents:
main, low, mid,
*** thread with running time 4 looped 2 times (stats->totalTicks: 320)
Ready list contents:
main, low, mid,
*** thread with running time 4 looped 3 times (stats->totalTicks: 330)
Ready list contents:
main, low, mid,
*** thread with running time 4 looped 4 times (stats->totalTicks: 340)
Ready list contents:
main, low, mid,
*** thread with running time 4 looped 5 times (stats->totalTicks: 350)
Ready list contents:
main, low, mid,
*** thread with running time 4 looped 6 times (stats->totalTicks: 360)
Ready list contents:
main, low, mid,
*** thread with running time 4 looped 7 times (stats->totalTicks: 370)
Ready list contents:
main, low, mid,
*** thread with running time 4 looped 8 times (stats->totalTicks: 380)
Ready list contents:
main, low, mid,
*** thread with running time 4 looped 9 times (stats->totalTicks: 390)
Ready list contents:
main, low, mid,
Timer interrupt's duration is 100
Time up and switch!
Ready list contents:
main, low, mid,
*** thread with running time 1 looped 7 times (stats->totalTicks: 410)
Ready list contents:
low, mid, high,
*** thread with running time 1 looped 8 times (stats->totalTicks: 420)
Ready list contents:
low, mid, high,
*** thread with running time 1 looped 9 times (stats->totalTicks: 430)
Ready list contents:
low, mid, high,
*** thread with running time 1 looped 10 times (stats->totalTicks: 440)
Ready list contents:
low, mid, high,

*** thread with running time 3 looped 10 times (stats->totalTicks: 460)
Ready list contents:
mid, high,
*** thread with running time 3 looped 11 times (stats->totalTicks: 470)
Ready list contents:
mid, high,
*** thread with running time 3 looped 12 times (stats->totalTicks: 480)
Ready list contents:
mid, high,
*** thread with running time 3 looped 13 times (stats->totalTicks: 490)
Ready list contents:
mid, high,
Timer interrupt's duration is 100
Time up and switch!
Ready list contents:
mid, high,
*** thread with running time 2 looped 10 times (stats->totalTicks: 510)
Ready list contents:
high, low,
*** thread with running time 2 looped 11 times (stats->totalTicks: 520)
Ready list contents:
high, low,
*** thread with running time 2 looped 12 times (stats->totalTicks: 530)
Ready list contents:
high, low,
*** thread with running time 2 looped 13 times (stats->totalTicks: 540)
Ready list contents:
high, low,
*** thread with running time 2 looped 14 times (stats->totalTicks: 550)
Ready list contents:
high, low,
*** thread with running time 2 looped 15 times (stats->totalTicks: 560)
Ready list contents:
high, low,
*** thread with running time 2 looped 16 times (stats->totalTicks: 570)
Ready list contents:
high, low,
*** thread with running time 2 looped 17 times (stats->totalTicks: 580)
Ready list contents:
high, low,
*** thread with running time 2 looped 18 times (stats->totalTicks: 590)
Ready list contents:
high, low,
Timer interrupt's duration is 100
Time up and switch!
Ready list contents:

```

```

high, low,
*** thread with running time 4 looped 10 times (stats->totalTicks: 610)
Ready list contents:
low, mid,
*** thread with running time 4 looped 11 times (stats->totalTicks: 620)
Ready list contents:
low, mid,
*** thread with running time 4 looped 12 times (stats->totalTicks: 630)
Ready list contents:
low, mid,
*** thread with running time 4 looped 13 times (stats->totalTicks: 640)
Ready list contents:
low, mid,
*** thread with running time 4 looped 14 times (stats->totalTicks: 650)
Ready list contents:
low, mid,
*** thread with running time 4 looped 15 times (stats->totalTicks: 660)
Ready list contents:
low, mid,
*** thread with running time 4 looped 16 times (stats->totalTicks: 670)
Ready list contents:
low, mid,
*** thread with running time 4 looped 17 times (stats->totalTicks: 680)
Ready list contents:
low, mid,
*** thread with running time 4 looped 18 times (stats->totalTicks: 690)
Ready list contents:
low, mid,
Timer interrupt's duration is 100
Time up and switch!
Ready list contents:
low, mid,
*** thread with running time 3 looped 14 times (stats->totalTicks: 710)
Ready list contents:
mid, high,
*** thread with running time 3 looped 15 times (stats->totalTicks: 720)
Ready list contents:
mid, high,
*** thread with running time 3 looped 16 times (stats->totalTicks: 730)
Ready list contents:
mid, high,
*** thread with running time 3 looped 17 times (stats->totalTicks: 740)
Ready list contents:
mid, high,
*** thread with running time 3 looped 18 times (stats->totalTicks: 750)
Ready list contents:

```

```

mid, high,
*** thread with running time 3 looped 19 times (stats->totalTicks: 760)
Ready list contents:
mid, high,
*** thread with running time 3 looped 20 times (stats->totalTicks: 770)
Ready list contents:
mid, high,
*** thread with running time 3 looped 21 times (stats->totalTicks: 780)
Ready list contents:
mid, high,
*** thread with running time 3 looped 22 times (stats->totalTicks: 790)
Ready list contents:
mid, high,
Timer interrupt's duration is 100
Time up and switch!
Ready list contents:
mid, high,
*** thread with running time 2 looped 19 times (stats->totalTicks: 810)
Ready list contents:
high, low,
*** thread with running time 2 looped 20 times (stats->totalTicks: 820)
Ready list contents:
high, low,
*** thread with running time 4 looped 19 times (stats->totalTicks: 840)
Ready list contents:
low,
*** thread with running time 4 looped 20 times (stats->totalTicks: 850)
Ready list contents:
low,
*** thread with running time 4 looped 21 times (stats->totalTicks: 860)
Ready list contents:
low,
*** thread with running time 4 looped 22 times (stats->totalTicks: 870)
Ready list contents:
low,
*** thread with running time 4 looped 23 times (stats->totalTicks: 880)
Ready list contents:
low,
*** thread with running time 4 looped 24 times (stats->totalTicks: 890)
Ready list contents:
low,
Timer interrupt's duration is 100
Time up and switch!
Ready list contents:
low,
*** thread with running time 3 looped 23 times (stats->totalTicks: 910)

```

```

Ready list contents:
high,
*** thread with running time 3 looped 24 times (stats->totalTicks: 920)
Ready list contents:
high,
*** thread with running time 3 looped 25 times (stats->totalTicks: 930)
Ready list contents:
high,
*** thread with running time 3 looped 26 times (stats->totalTicks: 940)
Ready list contents:
high,
*** thread with running time 3 looped 27 times (stats->totalTicks: 950)
Ready list contents:
high,
*** thread with running time 3 looped 28 times (stats->totalTicks: 960)
Ready list contents:
high,
*** thread with running time 3 looped 29 times (stats->totalTicks: 970)
Ready list contents:
high,
*** thread with running time 3 looped 30 times (stats->totalTicks: 980)
Ready list contents:
high,
Timer interrupt's duration is 100
Time up and switch!
Ready list contents:

*** thread with running time 4 looped 25 times (stats->totalTicks: 1010)
Ready list contents:

*** thread with running time 4 looped 26 times (stats->totalTicks: 1020)
Ready list contents:

*** thread with running time 4 looped 27 times (stats->totalTicks: 1030)
Ready list contents:

*** thread with running time 4 looped 28 times (stats->totalTicks: 1040)
Ready list contents:

*** thread with running time 4 looped 29 times (stats->totalTicks: 1050)
Ready list contents:

*** thread with running time 4 looped 30 times (stats->totalTicks: 1060)
Ready list contents:

*** thread with running time 4 looped 31 times (stats->totalTicks: 1070)

```

```

Ready list contents:

*** thread with running time 4 looped 32 times (stats->totalTicks: 1080)
Ready list contents:

*** thread with running time 4 looped 33 times (stats->totalTicks: 1090)
Ready list contents:

Timer interrupt's duration is 100
Time up and switch!
Ready list contents:

*** thread with running time 4 looped 34 times (stats->totalTicks: 1110)
Ready list contents:

*** thread with running time 4 looped 35 times (stats->totalTicks: 1120)
Ready list contents:

*** thread with running time 4 looped 36 times (stats->totalTicks: 1130)
Ready list contents:

*** thread with running time 4 looped 37 times (stats->totalTicks: 1140)
Ready list contents:

*** thread with running time 4 looped 38 times (stats->totalTicks: 1150)
Ready list contents:

*** thread with running time 4 looped 39 times (stats->totalTicks: 1160)
Ready list contents:

*** thread with running time 4 looped 40 times (stats->totalTicks: 1170)
Ready list contents:

No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 1200, idle 20, system 1180, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...

```

内容二：遇到的困难以及收获

首先，我对 Nachos 的时钟中断机制的理解上遇到了一些问题。起初我并不理解什么时候会触发时钟中断，以及时钟中断是怎样触发线程切换的，这是我实现 Round Robbin 算法遇到的一个瓶颈。后来，我逐个分析代码文件，发现 OnTick 这个方法会管辖 stats 全局变量中的各种 tick 的记录，且 Nachos 的时钟中断处理函数 TimeHandler 对 yieldOnReturn 变量的设置也是在这里恢复的（同时会调用 Yield 切换线程），这一定程度上地解决了我的疑惑。更进一步地，我通过阅读代码知道了 OnTick 的触发时机，以及 Nachos 已有的触发机会，这启发了我使用类似的想法去实现 Round Robbin——实现对应的时钟中断处理函数，进而实现线程的调度。这样可以减少对 Scheduler 内部方法的修改，也可以实现 Exercise3 与 Challenge 的调度方法的兼容。

其次，我对“抢占式”和“非抢占式”的调度有了更好的了解。抢占式是指只要就绪队列中出现了更高优先级的线程，就会进行调度。由于就绪队列的更新必然伴随事件的发生，因此只要出现了事件，抢占式调度就应该运行调度算法；而非抢占式则只有在当前运行的进程终止了，或者当前运行的进程阻塞/自动让出 CPU (yield) 的时候，才会进行进程的调度。我们需要实现的是抢占式调度算法。在不考虑时钟中断的情况下，Yield 函数时一定会进行调度，除此之外，当某个线程 Fork 了一个更高优先级的线程（模拟就绪队列出现了更高优先级的线程）时，也需要进行调度。后者是我在最初完成 Exercise3 的时候忽略掉了的，这个练习从这一角度加深了我对“抢占式”的理解。

内容三：对课程或 Lab 的意见和建议

希望可以把引导手册进一步完善，现在每次阅读代码的时候需要我们递归阅读和寻找其他额外的代码，但我们往往不知道应该去哪里能找到，这样浪费了许多不必要的时间。我觉得可以把某些常用函数（或者重要辅助函数）的位置标注出来，这样可以节省遍历搜索的时间。

也希望可以为调研类的作业提供更多的提示和说明。现在我们只能漫无目的地去遍历搜索百度、google 的结果，我们无法确认博客等网站表述内容的正确性和严谨性，而且有的调研题目很难找到相关博客。让我们在短时间内查询所有体系结构/操作系统的手册，对于还有许多其他课程的本科生来说在时间上不是很现实。所以希望课程可以提供更多相关资源，或者提示我们应该去看手册的哪些章节，去找什么样的权威资料。这样既可以让我们学到准确的知识，也可以提升我们的学习效率。

内容四：参考文献

1.Linux 调度机制：

<https://github.com/oska874/process-scheduling-in-linux>

2.Linux 调度机制:

<https://github.com/freelancer-leon/notes/tree/master/kernel/sched>