线程调度实习报告

姓名： 马少鹏 学号： 1500012893

日期： 2018.3.18

目录

内容一：总体概述

本次实习首先通过对 Linux 中采用的进程调度算法的调研，和对 Nachos 系统平台的底层源代码的阅读，理解进程/线程的调度过程与采用的算法。

在理解的基础上，通过对 Nachos 源代码的修改，达到“扩展调度算法”的目标。

内容二：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 的完成情况

第一部分

Exercise1 调研 Linux 中采用的进程调度算法、进程状态及转换关系

Linux 采用的进程调度算法为 CFS 组调度（完全公平调度算法）。

CFS 的原理：

CFS 给 run_queuq 中的每一个进程安排一个虚拟时钟 vruntime。如果一个进程得以执行，随着时间的增长，其 vruntime 将不断增大。没有得到执行的进程 vruntime 不变。而调度其总是选择 vruntime 最小的那个进程来执行。这就是“完全公平”。为了区别不同优先级的进程，优先级高的进程 vruntime 增长的慢，这样便可以得到更多的运行机会。

CFS 的数据结构：

每个 task_struct 中都有一个 sched_entity，调度实体，进程的 vruntime 和权重都保存在这个结构中。所有的 sched_entity 通过红黑树组织在一起，以 vruntime 为 key 插入到红黑树中。并且缓存最左侧的节点，也就是 vruntime 最小的节点，这样可以迅速选中 vruntime 最小的进程。

Linux 进程状态及转换关系：

TASK_RUNNING：

进程当前正在运行，或者正在运行队列中等待调度。

TASK_INTERRUPTIBLE：

进程处于睡眠状态，正在等待某些事件发生。进程可以被信号中断。接收到信号或被显式的唤醒呼叫唤醒之后，进程将转变为 TASK_RUNNING 状态。

TASK_UNINTERRUPTIBLE：

此进程状态类似于 TASK_INTERRUPTIBLE，只是它不会处理信号。中断处于这种状态的进程是不合适的，因为它可能正在完成某些重要的任务。 当它所等待的事件发生时，进程将被显式的唤醒呼叫唤醒。

TASK_STOPPED：

进程已中止执行，它没有运行，并且不能运行。接收到 SIGSTOP 和 SIGTSTP 等信号时，进程将进入这种状态。接收到 SIGCONT 信号之后，进程将再次变得可运行。

TASK_TRACED：

正被调试程序等其他进程监控时，进程将进入这种状态。

EXIT_ZOMBIE：

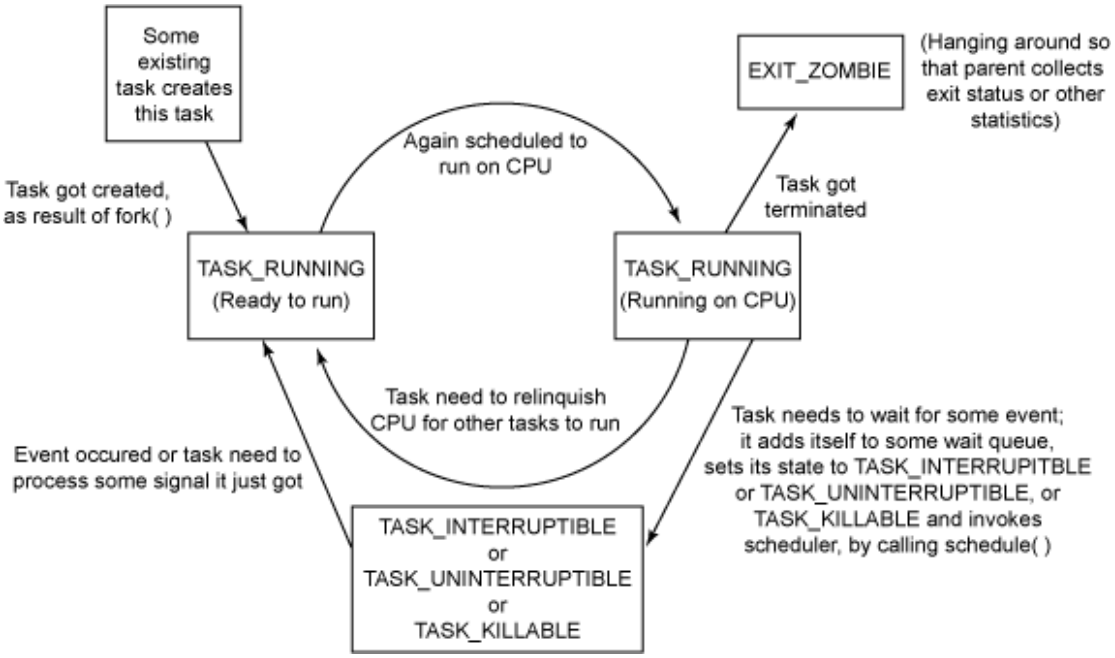
进程已终止，它正等待其父进程收集关于它的一些统计信息。

EXIT_DEAD：

最终状态（正如其名）。将进程从系统中删除时，它将进入此状态，因为其父进程已经通过 wait4() 或 waitpid() 调用收集了所有统计信息。

Linux Kernel 2.6.25 引入了一种新的进程睡眠状态，TASK_KILLABLE：当进程处于这种可以终止的新睡眠状态中，它的运行原理类似于 TASK_UNINTERRUPTIBLE，只不过可以响应致命信号。

状态转换图：



Exercise2 源代码阅读，理解 Nachos 现有的线程机制

code/threads/scheduler.h 和 code/threads/scheduler.cc

schedule.h 中定义了线程调度的数据结构 Scheduler 类，它的成员有：

构造函数与析构函数。

成员函数 void ReadyToRun(Thread* thread) // 设置进程状态为 ready，并把该进程添加到 readyList 队列的末尾。

成员函数 Thread* FindNextToRun() // 返回 readyList 中第一个线程的指针，如果不存在，则返回 NULL。

成员函数 Run(Thread* nextThread) // 执行线程 nextThread，包括设置线程状态为 running，通过调用 SWITCH 函数进行与上一个执行的线程之间的上下文切换。此外，将

需要杀死的线程对象 delete 掉。

成员函数 Print() // 打印 readyList 信息。

成员变量 readyList // ready 状态的线程指针的队列。

code/threads/switch.s

switch 中定义了执行两个进程之间的上下文切换的过程 SWITCH。在栈中保存了老线程的返回地址、被调用者保存的寄存器信息、栈指针等信息，并且加载了新线程的对应的信息，完成线程间上下文的切换。

值得注意的是，在 i386 中，老线程的指针位置在栈中有一个 4 字节的偏移量，并不在栈顶，这是因为在 bss 段中声明了 4 字节的内存空间用来存储寄存器 eax 的信息，eax 中本来保存的是返回地址，也就是 startup function 的指针。在栈中保存了 eax 之后，便可以使用 eax 寄存器了，代码中 eax 被用来保存线程 t1，也就是老线程的指针，用于构成保存老线程信息的操作的操作数。

code/machine/timer.h code/machine/timer.cc

timer.h 中定义了计时器的数据结构 timer 类。

它的成员变量有：

Randomize // 表示计时器是否以随机时间片产生中断

Handler // 计时器中断管理程序

Arg // 传给中断管理程序的参数

构造函数，初始化成员变量，调用 interrupt->Schedule()函数在中断等待队列中添加一个项，这个项就是计时器产生的第一个中断。

TimerExpired()函数，该函数调用 interrupt->Schedule()函数在中断等待队列中添加下一个中断项，并且调用了计时器的中断管理函数。

成员函数 TimerOfNextInterrupt()返回距计时器下一次产生中断的时间。

综上，Nachos 的线程调度机制为先来先服务调度算法。即维护一个 readyLlist 队列，记录所有已经就绪但并没有正在执行的线程。当 cpu 空闲，寻找下一个要执行的线程时，即调用 FindNextToRun()函数时，选择队列首部的线程执行。当 fork 新线程时，只需要将线程状态设置为 ready，并把该线程添加到 readyLiist 队列的尾部。

Exercise3 线程调度算法扩展 实现基于优先级的抢占式调度算法

基于优先级：

首先在 Thread 类中增加成员变量 priority，即线程自身的优先级，priority 的范围为 0-128，值越大，则优先级越高。主线程 main 优先级默认为 128，即最高优先级。

为了在新建线程的时候初始化优先级，对 Thread 类的构造函数做出如下修改：增加一个缺省的参数 Priority，缺省值为 0。当线程 ID 为 0，即创建的是主线程时，priority 置为 128，否则 priority 等于参数 Priority。

为了调用成员变量 priority，在 Thread 类中增加成员函数 get_priority()，返回 priority 的值。

此外，需要修改 readyList 中线程的排序，需要根据 priority 的值从大到小排列。之前向 readyList 中添加项的时候使用的是 list 类的 append 函数，添加到队尾，现在改成使用成员函数 SortedInsert 添加，该函数始终维护 readyList 中的项按照 priority 的值从大到小排列（Nachos 源代码中该函数维护队列从小到大排列，稍作修改即可）。故当调

用 FindNextToRun 的时候，仍旧找到 readyList 队列的首项，但此时首项即优先级最高的项。即线程切换时，始终寻找优先级最高的项。

值得注意的是，如果使用 Yield 切换下 cpu 的线程仍旧为优先级最高的线程，那么该线程将不会让出 cpu 的使用权，即 Yield 操作不会把 cpu 的使用权限让给比自身优先级低的线程。要实现此功能，需要修改 Thread 类中的成员函数 Yield()：找到 readyList 中优先级最高的线程后，与当前线程的优先级进行比较，如果下一个线程的优先级大于等于当前线程的优先级，才进行切换，否则继续执行当前线程，并把下一个线程放回 readyList 的首部。

实现功能后的运行结果：

```
msp@msp-virtual-machine:~/os/os_lab/nachos_dianti/nachos-3.4/code/threads$ ./nac
hos
thread 0 priority: 128
thread 4 priority: 40
thread 3 priority: 30
thread 1 priority: 20
thread 2 priority: 10
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 90, idle 0, system 90, 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...
msp@msp-virtual-machine:~/os/os_lab/nachos_dianti/nachos-3.4/code/threads$
```

主线程 main 依次创建优先级为 20、10、30、40 的线程，可以看到线程执行顺序并不是按照创建时间顺序，而是按照优先级从高到低的顺序执行。

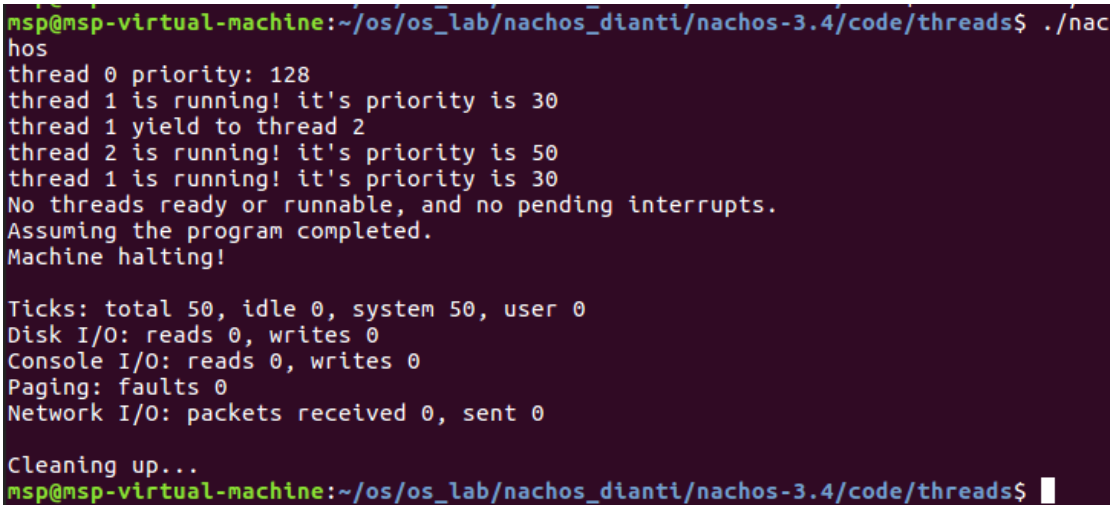
抢占式调度算法的实现：

所谓“抢占”，即如果有一个优先级更高的线程就绪，需要中断当前线程，执行优先级更高的线程。实现方法：

修改 Thread 类中的成员函数 Fork()，当创建新线程时，比较新线程与当前线程的优先级，如果新线程的优先级高，则当前线程调用 Yield()函数，让出 cpu 的使用权。

如果需要正在执行的高优先级线程让出 cpu 给低优先级的线程，当前线程调用 Sleep()函数即可。

实现功能后的运行结果：



可以看出，线程 1 执行时，创建了优先级更高的线程 2，此时线程 1 停止执行，切换到线程 2。

综上，基于优先级的抢占式调度算法实现成功。

Challenge： 实现时间片轮转算法

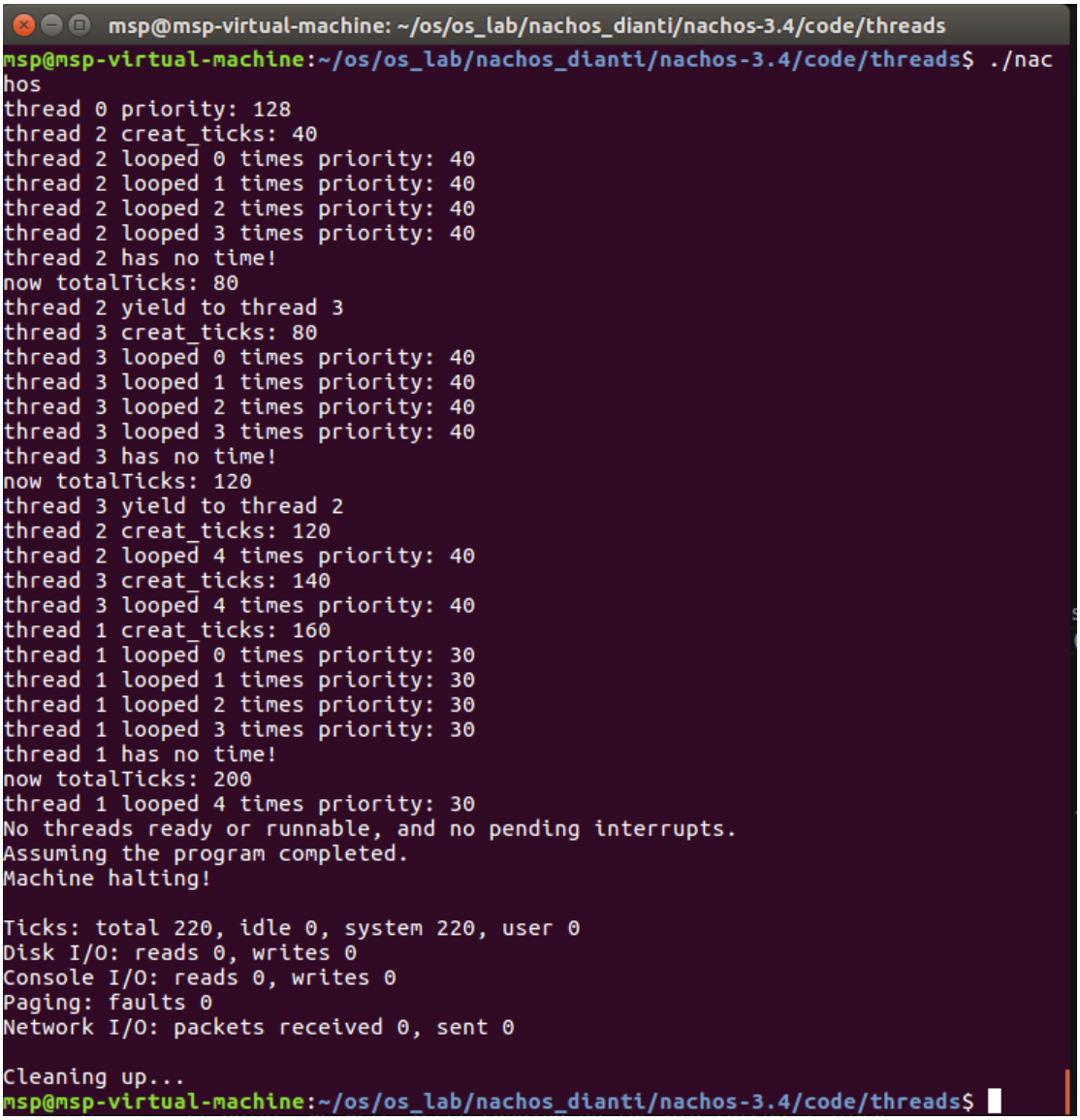
经过对代码的仔细阅读，在了解 Nachos 实现中断的原理和之前实现的基于优先级的抢占式调度算法的基础上，实现了时间片轮转算法，具体实现如下：

为了方便调试，设置每隔线程的时间片长度为 40ticks。线程执行时，分配给该线程长度为 40ticks 的时间片。当时间片耗尽时，让出 cpu，根据优先级选择下一个要执行的线程。即如果下一个线程的优先级与当前线程优先级相同，则让出 cpu 的使用权，若下一个线程的优先级低于当前线程的优先级，则当前线程继续执行，重新分配时间片给该线程。值得注意的是，不会出现下一个线程的优先级高于当前线程的优先级，因为这是在已经实现抢占式调度算法的基础上实现的。具体的实现方式是：

在 scheduler.cc 中的 Scheduler 类的成员函数 Run()中，调用 interrupt 类的 Schedule()函数，Schedule()的功能时新建一个中断，并按中断发生的时间顺序添加到中断等待队列中。并设置参数 when=40，即中断在 40ticks 后发生，也就是时间片的长度。参数 arg 为创建中断的线程 ID。

在 system.cc 中，新建计时器中断管理程序 void TimerInterruptHandler2(int dummy)，与 Nachos 原本的计时器中断管理程序相似，但增加对线程 ID 的判断，只有当当前要发生的中断的成员变量 arg 与当前线程 ID 相等时，才进行上下文的切换，进行线程切换，这样做是为了保证线程不受其他线程创建的中断的影响。如果判断相等，则表示当前线程时间片耗尽，应该进行线程切换。

运行结果如下：



测试程序中主线程 main 顺序创建了三个线程，线程 ID 分别为 1、2、3，优先级分别为 30、40、40。三个线程各自循环 5 次打印自身信息。由图中可以看出，综合优先级和时间顺序，线程 2 先运行，循环 4 次后时间片耗尽，切换到优先级相同的线程 3。线程 3 同样耗尽时间片后，切换到线程 2，线程 2 未用尽时间片就执行完毕，直接切换到线程 3。线程 3 未用尽时间片就执行完毕，直接切换到优先级更低的线程 1。线程 1 循环 4 次耗尽时间片后，因为不存在其他线程，所以重新分配时间片，继续执行。说明时间片轮转算法实现成功。

内容三：遇到的困难以及解决方法

困难 1 对 Nachos 实现计时器的理解

开始时我以为在 Nachos 初始化时打开计时器后，始终就自动运行，但是发现代码运行结果与未开计时器时没有区别。仔细研究代码后发现，在 OneTick()函数中，“时间才会往前

走”。而 OneTick()函数又在开关中断时调用，所以我修改了我的测试代码，在每次循环中开关中断，才解决了问题。

在实现时间片轮转算法时，开始时我理解错了概念，实现的结果是系统始终每过固定时间就产生中断，而不是线程执行固定时间让出 cpu。后来和讨论小组队友讨论后发现我概念理解错了，就重新做了一遍。

内容四：收获及感想

通过这次实习，我理解了常见的以及 Linux 采用的进程调度算法，并自己简单实现了基于优先级的抢占式调度算法和时间片轮转算法。尤其在实现时间片轮转算法时，我对 Nachos 实现计时器的原理有了更加深刻的理解。

内容五：对课程的意见和建议

希望课堂上对于上次 lab 的回顾可以更加详细。

内容六：参考文献

[1] Andrew S. Tanenbaum 著．陈向群 马洪兵 译 .现代操作系统［M］. 北京：机械工业出版社，2011：47-95