

Lab2 线程调度实 习报告

姓名 姜慧强 学号 1801210840
日期 2019.03.08

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 | 14 |
| 内容四：收获及感想 | 14 |
| 内容五：对课程的意见和建议 | 15 |
| 内容六：参考文献 | 15 |

内容一：总体概述

本次实验室操作系统高级课程的第二次实验。在 `Lab1` 阅读线程相关代码并完成修改 `Thread Class` 数据结构任务的基础上，主要完成了线程调度相关代码，时间管理代码，`switch` 汇编代码的阅读。并在代码阅读的基础上，完成了基于优先级抢占式调度算法，完成带有时间片的优先级抢占调度算法。通过 `lab2` 的实践，强化了对线程类，调度类等代码的理解。

内容二：任务完成情况

任务完成列表 (Y/N)

| EXERCISE1 | EXERCISE2 | EXERCISE3 | CHANELLAGE1 |
|-----------|-----------|-----------|-------------|
| Y | Y | Y | Y |

具体 Exercise 的完成情况

Exercise1

`Linux` 中的线程调度算法相对而言比较复杂，一般 `OS` 的线程调度算法对 `I/O` 密集型的线程都比较友好，`Linux` 也不例外，其更倾向于优先调度高 `I/O` 的进程。

`Linux` 使用的是带时间片的动态优先级抢占式调度模式，被称之为公平调度 `CFS` 的算法。其利用 `nice` 值+实时优先级+时间片共同维护线程的优先级。

在 `Unix` 中如果有两个同 `nice` 值的进程，那么他们都将分配到一半的时间片，一般为 `5ms` 的时间，在这段时间内 CPU 完全属于占用的进程。理想状态下应该是两个相同优先级的进程共同使用这段时间片 `10ms`，各占有 CPU 一半的能力。

`CFS` 通过计算所有就绪态进程的需要 CPU 时间，计算出一个总的 CPU 需要时间，从而尽可能根据各个进程的实际需要来进行分配，而原来直接当做优先级的 `nice` 值现在用于分配各个进程实际使用权重的标准。

其具体的计算公式见右 `weight = 1024/(1.25^nice)`。

通过上述的论述，可以发现，这样的操作能保证各个进程间权重比与 `nice` 的差值之间保持一致。这样就能减小原来在 `Unix` 中单纯使用 `nice` 值进程权重划分造成的权重与 `nice` 值绝对大小有较大关系的情况。

`CFS` 的具体实现细节，需要对 `Linux kernel` 的源码进行阅读。

通过对 `Linux kernel4.19.25` 代码的阅读，发现 `Linux` 关于线程调度的代码大致可分为时间记录，进程选择，调度器入口，睡眠唤醒，抢占五部分。

- **时间记录**

调度器需要记录当前调度周期内，进程还剩下多少时间片可用。

`Linux` 中的调度器实体 `class` 定义于 `<include/linux/sched.h>` 文件中。

```
struct sched_entity {
```

```

/* For load-balancing: 负责使得调度尽量均衡的模块 */
struct load_weight          load; // 优先级
unsigned long                runnable_weight; // 就绪态中的权值
struct rb_node               run_node; // 红黑树节点
struct list_head              group_node; // 所在进程组
unsigned int                 on_rq; // 是否在红黑树队列中

u64                          exec_start; // 线程开始时间
u64                          sum_exec_runtime; // 线程总运行时间
u64                          vruntime; // 虚拟运行时间
u64                          prev_sum_exec_runtime; // 上个调度周期总运行时间

u64                          nr_migrations;

struct sched_statistics  statistics;
};


```

上面的代码中有一项叫做 `vruntime`, 直译就是虚拟运行时间, 简单的理解可以认为是带权的运行时间, 利用一个权值来控制时间的快慢(好像有点恐怖)。CFS 利用 `vruntime` 来记录当前进程运行时间以及还需要运行的时间。其源码位于<kernel/sched/fair.c>

```

/*
 * Update the current task's runtime statistics.
 */
static void update_curr(struct cfs_rq *cfs_rq)
{
    struct sched_entity *curr = cfs_rq->curr;
    u64 now = rq_clock_task(rq_of(cfs_rq));
    u64 delta_exec;

    if (unlikely(!curr))
        return;
    // 获得最后一次 Switch Thread 至今耗时
    delta_exec = now - curr->exec_start;
    if (unlikely((s64)delta_exec <= 0))
        return;

    curr->exec_start = now; // 更新开始执行时间

    schedstat_set(curr->statistics.exec_max,
                  max(delta_exec, curr->statistics.exec_max));

    curr->sum_exec_runtime += delta_exec;
    schedstat_add(cfs_rq->exec_clock, delta_exec);

    curr->vruntime += calc_delta_fair(delta_exec, curr); // 计算 vruntime
}

```

```

update_min_vruntime(cfs_rq);

if (entity_is_task(curr)) {
    struct task_struct *curtask = task_of(curr);

    trace_sched_stat_runtime(curtask, delta_exec, curr->vruntime);
    cgroup_account_cputime(curtask, delta_exec);
    account_group_exec_runtime(curtask, delta_exec);
}

account_cfs_rq_runtime(cfs_rq, delta_exec);
}

```

上述函数计算当前进程的已执行时间，存放于变量 `delta_exec`，然后调用函数 `calc_delta_fair` 更新 `vruntime` 值。而计算好的 `vruntime` 值将会在后面用作 `FindNextToRun` 函数的判断。

- **Next 进程选择**

选择 `NextThread` 是调度算法的核心。在 `Linux` 中，通过计算 `vruntime` 值来实现 `CFS` 算法。

在 `Linux` 中利用红黑树来维护可运行进程的队列。红黑树因为其自平衡的特性，在这个变 `vruntime` 的场景下特别适用，而且红黑树维护代价，遍历代价都比较低。

在这个红黑树上存储了 `Linux` 系统中所有可运行的进程，每个节点的值就是他们的 `vruntime` 值，那么这棵红黑树上最小的节点，就是其最左节点。这一部分搜索红黑树寻找最小节点的代码也在`<kernel/sched/fair.c>`中。可以看出其维护了一些规则，比如说以前换出去过的进程优先级比较好，刚入队的进程需要单独比较一下（`vruntime` 值可能还没有更新）。

```

1. vim kernel/sched/fair.c (docker)
* 4) do not run the "skip" process, if something else is available
*/
static struct sched_entity *
pick_next_entity(struct cfs_rq *cfs_rq, struct sched_entity *curr)
{
    struct sched_entity *left = __pick_first_entity(cfs_rq); // 取红黑树左下节点
    struct sched_entity *se;

    /*
     * If curr is set we have to see if its left of the leftmost entity
     * still in the tree, provided there was anything in the tree at all.
     */
    if (!left || (curr && entity_before(curr, left))) // 如果左节点不存在，或者设了 curr
        left = curr; // 则需要检查是否是最左节点

    se = left; /* ideally we run the leftmost entity 理论上找到了最左节点 */

    /*
     * 如果这个进程被skip 且下一个进程vruntime没有差太多的话，选下一个
     * Avoid running the skip buddy, if running something else can
     * be done without getting too unfair.
     */
    if (cfs_rq->skip == se) {
        struct sched_entity *second;
        if (se == curr) {
            second = __pick_first_entity(cfs_rq);
        } else {
            second = __pick_next_entity(se);
            if (!second || (curr && entity_before(curr, second)))
                second = curr;
        }

        if (second && wakeup_preempt_entity(second, left) < 1)
            se = second;
    }

    /*
     * 对待主动放弃CPU的进程 有优先权(好孩子卡)
     * Prefer last buddy, try to return the CPU to a preempted task.
     */
    if (cfs_rq->last && wakeup_preempt_entity(cfs_rq->last, left) < 1)
        se = cfs_rq->last;

    /*
     * 比较刚入队的节点，刚入队的节点可能vruntime还没更新
     * Someone really wants this to run. If it's not unfair, run it.
     */
    if (cfs_rq->next && wakeup_preempt_entity(cfs_rq->next, left) < 1)
        se = cfs_rq->next;

    clear_buddies(cfs_rq, se);

    return se;
}

```

就绪态进程队列的新增相对于红黑树插入新的节点。这一部分代码位于<kernel/sched/fair.c>的enqueue_entity函数中。

```

1. vim kernel/sched/fair.c (docker)
static void
enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
{
    bool renorm = !(flags & ENQUEUE_WAKEUP) || (flags & ENQUEUE_MIGRATED);
    bool curr = cfs_rq->curr == se;

    /*
     * If we're the current task, we must renormalise before calling
     * update_curr().
     */
    if (renorm && curr)
        se->vruntime += cfs_rq->min_vruntime;

    update_curr(cfs_rq); // 更新当前运行调度的实体的vruntime信息

    /*
     * Otherwise, renormalise after, such that we're placed at the current
     * moment in time, instead of some random moment in the past. Being
     * placed in the past could significantly boost this task to the
     * fairness detriment of existing tasks.
     */
    if (renorm && !curr)
        se->vruntime += cfs_rq->min_vruntime; // 加回task_fork_fir() 中减去的min_vruntime

    /*
     * When enqueueing a sched_entity, we must:
     * - Update loads to have both entity and cfs_rq synced with now.
     * - Add its load to cfs_rq->runnable_avg
     * - For group_entity, update its weight to reflect the new share of
     *   its group cfs_rq
     * - Add its new weight to cfs_rq->load.weight
     */
    update_load_avg(cfs_rq, se, UPDATE_TG | DO_ATTACH);
    update_cfs_group(se);
    enqueue_runnable_load_avg(cfs_rq, se);
    account_entity_enqueue(cfs_rq, se); // 更新就绪队列信息

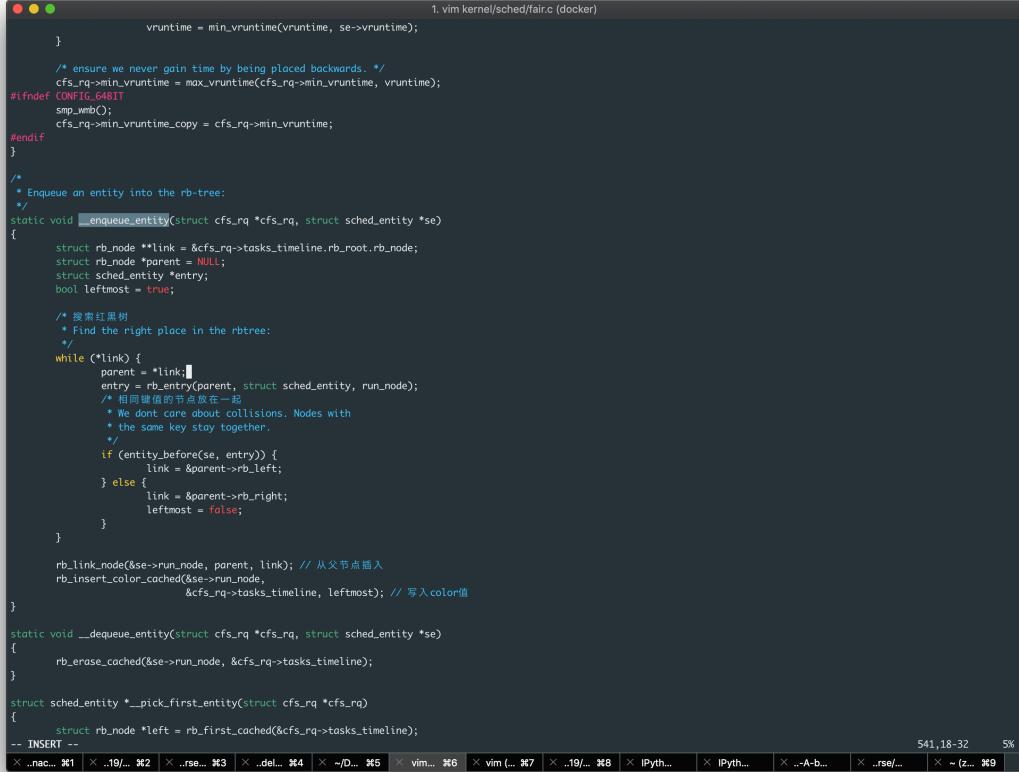
    if (flags & ENQUEUE_WAKEUP)
        place_entity(cfs_rq, se, 0); // 对唤醒进程的特殊处理

    check_schedstat_required();
    update_stats_enqueue(cfs_rq, se, flags);
    check_spread(cfs_rq, se);
    if (!curr)
        __enqueue_entity(cfs_rq, se); // 将se加入红黑树
    se->on_rq = 1; // se成功加入红黑树

    if (cfs_rq->nr_running == 1) {
        list_add_leaf_cfs_rq(cfs_rq);
        check_enqueue_throttle(cfs_rq);
    }
}
-- INSERT --

```

从上面的代码可以看出 `queue_entity()` 函数主要用于更新 `vruntime`, 队列信息等等。真正做红黑树插入的逻辑实际上在 `_queue_entity()` 函数中。



```
1. vim kernel/sched/fair.c (docker)
}
    vruntime = min_vruntime(vruntime, se->vruntime);
}

/* ensure we never gain time by being placed backwards. */
cfs_rq->min_vruntime = max_vruntime(cfs_rq->min_vruntime, vruntime);

#ifndef CONFIG_64BIT
    smp_mb();
cfs_rq->min_vruntime_copy = cfs_rq->min_vruntime;
#endif
}

/*
 * Enqueue an entity into the rb-tree:
 */
static void __enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
    struct rb_node **link = &cfs_rq->tasks_timeline.rb_root.rb_node;
    struct rb_node *parent = NULL;
    struct sched_entity *entry;
    bool leftmost = true;

    /* 搜索红黑树
     * Find the right place in the rbtree:
     */
    while (*link) {
        parent = *link;
        entry = rb_entry(parent, struct sched_entity, run_node);
        /* 相同键值的节点放在一起
         * We dont care about collisions. Nodes with
         * the same key stay together.
         */
        if (entity_before(se, entry)) {
            link = &parent->rb_left;
        } else {
            link = &parent->rb_right;
            leftmost = false;
        }
    }

    rb_link_node(&se->run_node, parent, link); // 从父节点插入
    rb_insert_color_cashed(&se->run_node,
                          &cfs_rq->tasks_timeline, leftmost); // 写入color值
}

static void __dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se)
{
    rb_erase_cashed(&se->run_node, &cfs_rq->tasks_timeline);
}

struct sched_entity *_pick_first_entity(struct cfs_rq *cfs_rq)
{
    struct rb_node *left = rb_first_cashed(&cfs_rq->tasks_timeline);
-- INSERT --
x ..nac... #1 x ..19/... #2 x ..rse... #3 x ..del... #4 x ..D... #5 x vim... #6 x vim (... #7 x ..19/... #8 x ..Pyth... x ..Pyth... x ..-A-b... x ..rsef... x ..(z... #9
```

和插入相似的还有从队列中删除节点，这个就不再赘述。

- **调度器入口**

Linux 在实现进程调度的时候提供了一个统一的调度器入口，在这个入口中选择最高优先级的调度类，每个调度类拥有自己的进程队列，相对于一个多队列调度算法。

关于调度器入口的代码定义在 `<kernel/sched/core.c>`，以优先级为序，依次检查每个调度类中的进程队列。

```

rcu_sleep_check();

profile_hit(SCHED_PROFILING, __builtin_return_address(0));
schedstat_inc(this_rq->sched_count);

/*
 * Pick up the highest-prio task:
 */
static inline struct task_struct *
pick_next_task(struct rq *rq, struct task_struct *prev, struct rq_flags *rf)
{
    const struct sched_class *class;
    struct task_struct *p;

    /* 相对于在这里做了个剪枝：如果当前所有需要调度的进程都是普通进程，就直接用CFS
     * Optimizations: we know that if all tasks are in the fair class we can
     * call that function directly, but only if the @prev task wasn't of a
     * higher scheduling class, because otherwise those loose the
     * opportunity to pull in more work from other CPUs.
     */
    if (likely(!prev->sched_class == &idle_sched_class ||
              prev->sched_class == &fair_sched_class) &&
        rq->nr_running == rq->cfss.h_nr_running)) {
        p = fair_sched_class.pick_next_task(rq, prev, rf);
        if (unlikely(p == RETRY_TASK))
            goto again;
        /* Assumes fair_sched_class->next == idle_sched_class */
        if (unlikely(p))
            p = idle_sched_class.pick_next_task(rq, prev, rf);
    }
    return p;
}

again:
for_each_class(class) { // 遍历各个调度类
    p = class->pick_next_task(rq, prev, rf);
    if (p) {
        if (unlikely(p == RETRY_TASK))
            goto again;
        return p;
    }
}
/* The idle class should always have a runnable task: */
BUG();
}

-- INSERT --

```

● 睡眠&唤醒

在 Linux 中进程的挂起态，分为两种，一种是能收到信号 `signal`，一种是忽略 `signal`。和就绪态用红黑树来维护不一样，这里的挂起态队列用一个简单的链表结构来实现。

具体来说是利用 `wait_queue_head` 的结构来构造一个等待队列。

```

struct wait_queue_head {
    spinlock_t          lock; // 自旋锁保持一致性
    struct list_head    head;
};

```

挂起态和就绪态的转换涉及到红黑树出树+链表入队，链表出队+红黑树插入，部分代码和上面所述的就绪态队列维护一致，就不在此赘述。

● 抢占

Linux 的线程调度是可抢占的，在实际操作过程中抢占的现象十分普遍，比如说在 Linux 系统上开了一个 `vim` 编辑器，然后又在后台跑了一个 `shell` 命令，因为交互的实时性需要，你在 `vim` 中编辑的时候，就发生了抢占现象。

```

/*
 * context_switch - switch to the new MM and the new thread's register state.
 */
static __always_inline struct rq *
context_switch(struct rq *rq, struct task_struct *prev, struct task_struct *next, struct rd_flags *rf)
{
    struct mm_struct *mm, *oldmm;
    prepare_task_switch(rq, prev, next); // 进程切换的准备工作

    mm = next->mm;
    oldmm = prev->active_mm;
    /*
     * For paravirt, this is coupled with an exit in switch_to to
     * combine the page table reload and the switch backend into
     * one hypercall.
     */
    arch_start_context_switch(prev);

    /*
     * If mm is non-NULL, we pass through switch_mm(). If mm is
     * NULL, we will pass through mmdrop() in finish_task_switch().
     * Both of these contain the full memory barrier required by
     * membarrier after storing to rq->curr, before returning to
     * user-space.
     */
    if (!mm) { // 如果是内核进程 <- 内核进程无虚拟地址
        next->active_mm = oldmm; // 内核进程的active_mm 为上一个进程的mm
        mmgrab(oldmm);
        enter_lazy_tlb(oldmm, next); // 通知底层不需要切换虚拟地址空间 -> 惰性TLB
    } else // 如果不是内核进程 就需要切换虚拟地址空间
        switch_mm_irqs_off(oldmm, mm, next);

    if (!prev->mm) { // 如果prev是内核进程 or 正在退出的进程
        prev->active_mm = NULL; // 把active_mm 设为空
        rq->prev_mm = oldmm; // 更新队列的prev_mm信息
    }

    rq->clock_update_flags &= ~(RQCF_ACT_SKIP|RQCF_REQ_SKIP);

    prepare_lock_switch(rq, next, rf);

    /* Here we just switch the register state and the stack. */
    switch_to(prev, next, prev); // 切换运行环境 堆栈+寄存器 返回上一个执行的程序
    barrier(); // 屏障同步 -> 汇编

    return finish_task_switch(prev); // 进程切换之后的一些收尾工作
}

-- INSERT --

```

Exercise2

threads/scheduler.h

在 `scheduler.h` 中声明了 `schedule class` 的一些参数比如说 `readyList`, 和一些静态函数, 比如说一些 `get, set` 方法。其具体实现见 `scheduler.cc` 文件中。

threads/scheduler.cc

`schedule.cc` 文件中除了实现在 `schedule.h` 文件中声明的一些函数, 还定义了一些 `public` 方法, 用于线程调用中使用。

具体来说, 有如下几个函数:

```

void Scheduler::ReadyToRun(Thread *thread) {} // 入就绪队列, 状态改为 READY
Thread *Scheduler::FindNextToRun() {} // 从 readyList 中取出下一个进程
void Scheduler::Run(Thread *nextThread) {} // 实现调度 上 CPU

```

threads/switch.s

进程切换的具体实现, 包括一些屏障同步, 切换 Context 的工作。

machine/timer.h

主要声明了模拟时钟中断的 `class`, 通过 `Timer class` 实现实现模拟硬件的时钟中断。

```

1. vim ..../machine/timer.h (docker)

// timer.h
// Data structures to emulate a hardware timer.
//
// A hardware timer generates a CPU interrupt every X milliseconds.
// This means it can be used for implementing time-slicing, or for
// having a thread go to sleep for a specific period of time.
//
// We emulate a hardware timer by scheduling an interrupt to occur
// every time stats->totalTicks has increased by TimerTicks.
//
// In order to introduce some randomness into time-slicing, if "doRandom"
// is set, then the interrupt comes after a random number of ticks.
//
// DO NOT CHANGE -- part of the machine emulation
//
// Copyright (c) 1992-1993 The Regents of the University of California.
// All rights reserved. See copyright.h for copyright notice and limitation
// of liability and disclaimer of warranty provisions.

#ifndef TIMER_H
#define TIMER_H

#include "copyright.h"
#include "utility.h"

// The following class defines a hardware timer.
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
};

#endif // TIMER_H


```

machine/timer.h

主要完成 timer.h 中声明的函数的实现。

```

// 初始化 Timer
Timer::Timer(VoidFunctionPtr timerHandler, int callArg, bool doRandom) {}
void Timer::TimerExpired() {} // 当发生时钟中断时调用
int Timer::TimeOfNextInterrupt() {} // 计算下一次时钟中断的时间

```

Exercise3

经过前面的源码阅读，可以发现如果要实现基于优先级的抢占式调度算法，首先需要在 `Thread class` 中增加 `priority` 变量。

在线程调度过程中，抢占的时机，可以是当前运行的线程主动放弃 CPU 地位的 `Yield()` 函数。但不能是在线程正在运行的时候，因为线程创建的时候，其 `Status` 是 `Just_ready` 状态。

另外因为需要根据优先级进行调度，所以在 `schedule.h` 维护一个根据 `priority` 自排序的就绪态队列。

总的来说，修改 `schedule.cc` 中的 `void Scheduler::ReadyToRun(Thread *thread) {}` 函数。

```

readyList->SortedInsert((void *)thread,
                        thread->getPriority()); // 通过对 list 的排序插入,
                                         // 维护一个根据优先级排序的 ReadyList

```

在 `thread.cc` 中的 `void Thread::Yield() {}` 函数

```

if (nextThread != NULL) { // lab2 ready run ref prio
    if (nextThread->getPriority() <= this->getPriority()) {
        scheduler->ReadyToRun(this);
        scheduler->Run(nextThread);
    } else {
        scheduler->ReadyToRun(nextThread);
    }
}

```

通过优先级的判断实现 Thread 能否抢占逻辑的实现。

另外还需要在线程初始化的时候，分配相应的优先级。

```

SimpleThread();
}

//-----lab1-Test-Begin-----
void Lab1Thread() {
    for (int i = 0; i <= 3; ++i) {
        printf("threadname: %s tid: %d uid: %d looped %d times\n",
               currentThread->getName(), currentThread->getTid(),
               currentThread->getUid(), i);
        currentThread->Yield();
    }
}

Thread *createThreadTest(int num, char *threadNameList) {
    Thread *temp = new Thread(threadNameList);
    temp->setTid(num);
    temp->Fork(Lab1Thread, (void *)1);
    return temp;
}

void PrintThreadInfo() {
    printf("threadname-----tid-----uid-----status\n");
    for (int i = 0; i < MaxThreadNum; ++i) {
        if (thread[i] != NULL) {
            thread[i]->Print();
        }
    }
}

void Lab1Test() {
    printf("Write by Liang Huiqiang 1801210840\n");
    char threadNameList[MaxThreadNum][20] = {};
    for (int i = 0; i < MaxThreadNum - 1; ++i) {
        char str[20];
        sprintf(str, "%d", i);
        strcat(threadNameList[i], "Thread");
        strcat(threadNameList[i], str);
        createThreadTest(i, threadNameList[i]);
    }
    Lab1Thread();
}

void TestThreadNumExceed() {
    for (int i = 1; i <= 128; ++i) {
        Thread *t = new Thread("testThread");
        printf("creat thread %d\n", i);
    }
}
//-----lab1-Test-End-----

```

通过测试程序，可以发现实现了线程按优先级顺序运行，同级进程实现了互相抢占的功能。

```

1. root@1801210840: ~/nachos/nachos-3.4/code/threads (docker)

..../threads/threadtest.cc: In function 'void TestThreadNumExceed()':
..../threads/threadtest.cc:95:40: warning: deprecated conversion from string constant to 'char*' [-Wwrite-strings]
    Thread *t = new Thread("testThread");
                           ^
..../threads/threadtest.cc:95:13: warning: unused variable 't' [-Wunused-variable]
    Thread *t = new Thread("testThread");
                           ^
..../threads/threadtest.cc: In function 'Thread* createThreadLab2Test(int, int, char*)':
..../threads/threadtest.cc:117:35: warning: invalid conversion from 'void (*)()' to 'VoidFunctionPtr {aka void (*)(int)}' [-fpermissive]
    temp->Fork(&lab2Thread, (void *)i);
                           ^
In file included from ..../threads/scheduler.h:14:0,
                 from ..../threads/system.h:13,
                 from ..../threads/threadtest.cc:14:
..../threads/thread.h:97:8: note: initializing argument 1 of 'void Thread::Fork(VoidFunctionPtr, void*)'
    void Fork(VoidFunctionPtr func, void *arg); // Make thread run (*func)(arg)
                           ^
g++ main.o list.o scheduler.o synch.o synclist.o system.o thread.o utility.o threadtest.o interrupt.o stats.o sysdep.o timer.o elevator.o elevatortest.o switch.o -o nachos
→ 1801210840 threads git:(develop) ✘ /nachos
Write by Jiang Huiqiang 1801210840 in 2019-03-08

threadname: main tid: 0 uid: 0 priority: 4 looped 0 times
threadname: Thread0 tid: 2 uid: 1 priority: 1 looped 0 times
threadname: Thread0 tid: 2 uid: 1 priority: 1 looped 1 times
threadname: Thread0 tid: 2 uid: 1 priority: 1 looped 2 times
threadname: Thread0 tid: 2 uid: 1 priority: 1 looped 3 times
threadname: Thread0 tid: 1 uid: 0 priority: 3 looped 0 times
threadname: Thread0 tid: 1 uid: 0 priority: 3 looped 1 times
threadname: Thread0 tid: 1 uid: 0 priority: 3 looped 2 times
threadname: Thread0 tid: 1 uid: 0 priority: 3 looped 3 times
threadname: Thread0 tid: 0 uid: 0 priority: 4 looped 0 times
threadname: main tid: 4 uid: 3 priority: 4 looped 0 times
threadname: Thread3 tid: 4 uid: 3 priority: 4 looped 1 times
threadname: Thread3 tid: 4 uid: 3 priority: 4 looped 2 times
threadname: main tid: 0 uid: 0 priority: 4 looped 3 times
threadname: Thread3 tid: 4 uid: 3 priority: 4 looped 4 times
threadname: Thread3 tid: 4 uid: 3 priority: 4 looped 5 times
threadname: Thread3 tid: 3 uid: 2 priority: 5 looped 0 times
threadname: Thread2 tid: 3 uid: 2 priority: 5 looped 1 times
threadname: Thread2 tid: 3 uid: 2 priority: 5 looped 2 times
threadname: Thread2 tid: 3 uid: 2 priority: 5 looped 3 times
threadname: Thread2 tid: 3 uid: 2 priority: 5 looped 4 times
threadname: Thread2 tid: 3 uid: 2 priority: 5 looped 5 times
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 290, idle 0, system 290, 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...
→ 1801210840 threads git:(develop) ✘ [1]
× ..... ● #1 | × 19... #2 | × ...rs... #3 | × ...de... #4 | × ~D... #5 | × ...os... #6 | × .../c... #7 | × ...19... #8 | × IPyth... | × IP... #9 | × ...A... | × ...rse... | × ... ~... #9 |
```

Challenge 1

需要实现一个带时间片的抢占调度算法，即时间片未走完的时候，可以根据优先级进行抢占，如果时间片结束了，同级别的进程交替进行。

需要维护一个上次 `switch` 时间戳，如果时间差小于时间片，则需要优先级大于才能抢占；如果时间差大于时间片，则可以同级抢占。

修改 `thread.cc` 中 `void Thread::Yield()` {} 函数

```

1. vim thread.cc (docker)
//----- NOTE: returns immediately if no other thread on the ready queue.
//----- Otherwise returns when the thread eventually works its way
//----- to the front of the ready list and gets re-scheduled.
//----- NOTE: we disable interrupts, so that looking at the thread
//----- on the front of the ready list, and switching to it, can be done
//----- atomically. On return, we re-set the interrupt level to its
//----- original state, in case we are called with interrupts disabled.
//----- Similar to Thread::Sleep(), but a little different.
//-----

void Thread::YieldO {
    Thread *nextThread;
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    ASSERT(this == currentThread);

    DEBUGC('t', "Yielding thread \"%s\"\n", getName());

    nextThread = scheduler->FindNextToRun();

    int interval = stats->systemTicks - lastTick; // lab2 Challenge 1
    printf("Interval Time is: %d ms\n", interval);
    if (nextThread != NULL) {
        if (interval < MinSwitchTick) {
            printf("NextPrio: %d; CurrentPrio: %d\n",
                   this->getPriority());
        }
        if (nextThread->getPriority() < this->getPriority()) {
            lastTick = stats->systemTicks;
            printf("Switch 1\n");
            scheduler->ReadyToRun(this);
            scheduler->Run(nextThread);
        } else {
            // printf("Wait\n");
            scheduler->ReadyToRun(nextThread);
        }
    } else {
        lastTick = stats->systemTicks;
        printf("NextPrio: %d; CurrentPrio: %d\n",
               this->getPriority());
        printf("Switch 2\n");
        scheduler->ReadyToRun(this);
        scheduler->Run(nextThread);
    }

    (void)interrupt->SetLevel(oldLevel);
}

//----- Thread::Sleep
//----- Relinquish the CPU, because the current thread is blocked

```

224,1 58%

通过开两个相同优先级的线程测试，得到的每个时间片交替切换线程，实现了所需要的功能。

```

1. root@1801210840: ~/nachos/nachos-3.4/code/threads (docker)
DEBUGC('t', "Sleeping thread \"%s\"\n", getName());
g++ main.o list.o scheduler.o synch.o synclist.o system.o thread.o utility.o threadtest.o interrupt.o stats.o sysdep.o timer.o elevator.o elevatortest.o switch.o -o nachos
→ 1801210840 threads git:(develop) ✘ ./nachos
Write by Jiang Huiqiang 1801210840 in 2019-03-08
threadname: main tid: 0 uid: 0 priority: 4 looped 1 times
Interval Time is: 150 ms
NextPrio: 3; CurrentPrio: 4
Switch 2
threadname: Thread0 tid: 1 uid: 0 priority: 3 looped 1 times
Interval Time is: 10 ms
NextPrio: 3; CurrentPrio: 3
threadname: Thread0 tid: 1 uid: 0 priority: 3 looped 2 times
Interval Time is: 20 ms
NextPrio: 3; CurrentPrio: 3
threadname: Thread0 tid: 1 uid: 0 priority: 3 looped 3 times
Interval Time is: 30 ms
NextPrio: 3; CurrentPrio: 3
threadname: Thread0 tid: 1 uid: 0 priority: 3 looped 4 times
Interval Time is: 40 ms
NextPrio: 3; CurrentPrio: 3
threadname: Thread0 tid: 1 uid: 0 priority: 3 looped 5 times
Interval Time is: 50 ms
NextPrio: 3; CurrentPrio: 3
Switch 2
threadname: Thread1 tid: 2 uid: 1 priority: 3 looped 1 times
Interval Time is: 10 ms
NextPrio: 3; CurrentPrio: 3
threadname: Thread1 tid: 2 uid: 1 priority: 3 looped 2 times
Interval Time is: 20 ms
NextPrio: 3; CurrentPrio: 3
threadname: Thread1 tid: 2 uid: 1 priority: 3 looped 3 times
Interval Time is: 30 ms
NextPrio: 3; CurrentPrio: 3
threadname: Thread1 tid: 2 uid: 1 priority: 3 looped 4 times
Interval Time is: 40 ms
NextPrio: 3; CurrentPrio: 3
threadname: Thread1 tid: 2 uid: 1 priority: 3 looped 5 times
Interval Time is: 50 ms
NextPrio: 3; CurrentPrio: 3
Switch 2
threadname: Thread0 tid: 1 uid: 0 priority: 3 looped 6 times
Interval Time is: 10 ms
NextPrio: 3; CurrentPrio: 3
threadname: Thread0 tid: 1 uid: 0 priority: 3 looped 7 times
Interval Time is: 20 ms
NextPrio: 3; CurrentPrio: 3
threadname: Thread0 tid: 1 uid: 0 priority: 3 looped 8 times
Interval Time is: 30 ms
NextPrio: 3; CurrentPrio: 3
threadname: Thread0 tid: 1 uid: 0 priority: 3 looped 9 times
Interval Time is: 40 ms
NextPrio: 3; CurrentPrio: 3

```

```

1. root@1801210840: ~/nachos/nachos-3.4/code/threads (docker)
threadname: Thread1 tid: 2 uid: 1 priority: 3 looped 8 times
Interval Time is: 30 ms
NextPrio: 3; CurrentPrio: 3
threadname: Thread1 tid: 2 uid: 1 priority: 3 looped 9 times
Interval Time is: 40 ms
NextPrio: 3; CurrentPrio: 3
threadname: Thread1 tid: 2 uid: 1 priority: 3 looped 10 times
Interval Time is: 50 ms
NextPrio: 3; CurrentPrio: 3
Switch 2
threadname: Thread3 tid: 4 uid: 3 priority: 4 looped 1 times
Interval Time is: 30 ms
NextPrio: 4; CurrentPrio: 4
threadname: Thread3 tid: 4 uid: 3 priority: 4 looped 2 times
Interval Time is: 40 ms
NextPrio: 4; CurrentPrio: 4
threadname: Thread3 tid: 4 uid: 3 priority: 4 looped 3 times
Interval Time is: 50 ms
NextPrio: 4; CurrentPrio: 4
threadname: main tid: 0 uid: 0 priority: 4 looped 2 times
Interval Time is: 10 ms
NextPrio: 4; CurrentPrio: 4
threadname: main tid: 0 uid: 0 priority: 4 looped 3 times
Interval Time is: 20 ms
NextPrio: 4; CurrentPrio: 4
threadname: main tid: 0 uid: 0 priority: 4 looped 4 times
Interval Time is: 30 ms
NextPrio: 4; CurrentPrio: 4
threadname: main tid: 0 uid: 0 priority: 4 looped 5 times
Interval Time is: 40 ms
NextPrio: 4; CurrentPrio: 4
threadname: main tid: 0 uid: 0 priority: 4 looped 6 times
Interval Time is: 50 ms
NextPrio: 4; CurrentPrio: 4
Switch 2
threadname: Thread3 tid: 4 uid: 3 priority: 4 looped 4 times
Interval Time is: 10 ms
NextPrio: 4; CurrentPrio: 4
threadname: Thread3 tid: 4 uid: 3 priority: 4 looped 5 times
Interval Time is: 20 ms
NextPrio: 4; CurrentPrio: 4
threadname: Thread3 tid: 4 uid: 3 priority: 4 looped 6 times
Interval Time is: 30 ms
NextPrio: 4; CurrentPrio: 4
threadname: Thread3 tid: 4 uid: 3 priority: 4 looped 7 times
Interval Time is: 40 ms
NextPrio: 4; CurrentPrio: 4
threadname: Thread3 tid: 4 uid: 3 priority: 4 looped 8 times
Interval Time is: 50 ms
NextPrio: 4; CurrentPrio: 4
Switch 2
threadname: main tid: 0 uid: 0 priority: 4 looped 7 times

```

内容三：遇到的困难以及解决方法

在实现时间片调度算法的时候，一开始的策略是修改 `schedule.cc` 中的 `Thread *Scheduler::FindNextToRun() { return (Thread *)readyList->Remove(); }`

结果会遇到只要是正在执行的进程不是在时间片临界点结束的话，之后的进程被放进 `ReadyList` 之后就没有机会再被执行了，而且因为 `return NULL;` 导致启动了 `idle` 模式 `No threads ready or runnable, and no pending interrupts.`

究其原因就是 `thread` 的 `Yield` 拿不到 `nextThread`，后面就继续不下去了。

思考了一会觉得时间片逻辑应该写在 `schedule` 上面一层，也就是 `thread.cc`。通过每次正在运行的进程请求 `Yield` 的时候进行逻辑判断。

这样就解决了原来线程被 `break` 的情况。

```

1. vim thread.cc (docker)

// NOTE: returns immediately if no other thread on the ready queue.
// Otherwise returns when the thread eventually works its way
// to the front of the ready list and gets re-scheduled.

// NOTE: we disable interrupts, so that looking at the thread
// on the front of the ready list, and switching to it, can be done
// atomically. On return, we re-set the interrupt level to its
// original state, in case we are called with interrupts disabled.
// Similar to Thread::Sleep(), but a little different.
//-----

void Thread::YieldO {
    Thread *nextThread;
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    ASSERT(this == currentThread);

    DEBUGC('t', "Yielding thread \"%s\"\n", getNameO);

    nextThread = scheduler->FindNextToRun();

    int interval = stats->systemTicks - lastTick; // lab2 Challenge 1
    printf("Interval Time is: %d ms\n", interval);
    if (nextThread != NULL) {
        if (interval < MinSwitchTicks) {
            printf("NextPrio: %d; CurrentPrio: %d\n",
                   nextThread->getPriorityO,
                   this->getPriorityO);
            if (nextThread->getPriorityO < this->getPriorityO) {
                lastTick = stats->systemTicks;
                printf("Switch 1\n");
                scheduler->ReadyToRun(this);
                scheduler->Run(nextThread);
            } else {
                // printf("Wait\n");
                scheduler->ReadyToRun(nextThread);
            }
        } else {
            lastTick = stats->systemTicks;
            printf("NextPrio: %d; CurrentPrio: %d\n",
                   nextThread->getPriorityO,
                   this->getPriorityO);
            printf("Switch 2\n");
            scheduler->ReadyToRun(this);
            scheduler->Run(nextThread);
        }
    }
    interrupt->SetLevel(oldLevel);
}

//-----
// Thread::Sleep
// Relinquish the CPU, because the current thread is blocked

```

内容四：收获及感想

这次的作业比 **lab1** 难度有所增加，但做起来反而没有 **lab1** 吃力。实践部分，对线程整体理解提升了挺多的。尤其是写报告的时候，补了一下 **linux** 源码阅读，感觉这块如果做得好，收获会特别大。

内容五：对课程的意见和建议

感谢老师给我们这个机会动手体会 **os** 中线程管理的具体逻辑。

内容六：参考文献

- [1] Stevens, W. R. (2002). UNIX 环境高级编程：英文版. 机械工业出版社.
- [2] RobertLove, et al. Linux 内核设计与实现. 机械工业出版社, 2006.
- [3] Linux 内核调度分析（进程调度）