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

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）。

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。

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。

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。

- **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。

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。其具体实现在 schedule.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。

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。

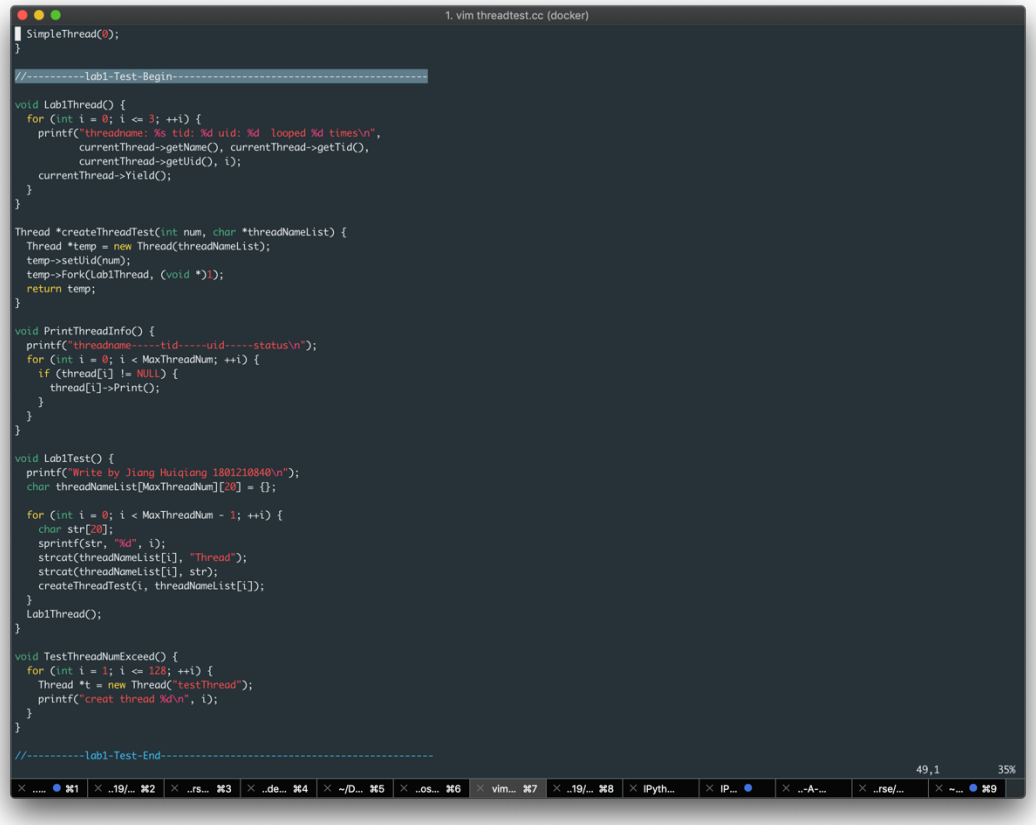
```
readyList->SortedInsert((void *)thread,
                        thread->getPriority()); // 通过对 list 的排序插入，
                                                // 维护一个根据优先级排序的 ReadyList
```

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

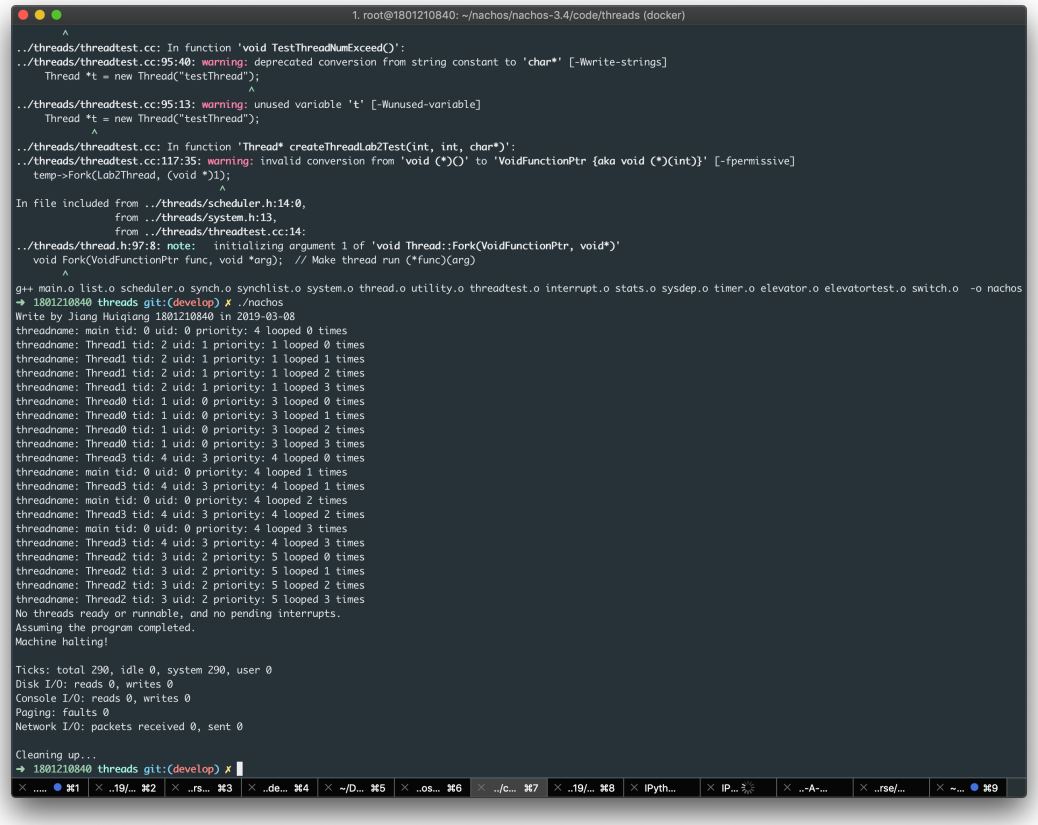
```
if (nextThread != NULL) {  // lab2 ready run ref prio
      if (nextThread->getPriority() <= this->getPriority()) {
        scheduler->ReadyToRun(this);
        scheduler->Run(nextThread);
      } else {
        scheduler->ReadyToRun(nextThread);
      }
    }
```

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。



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。

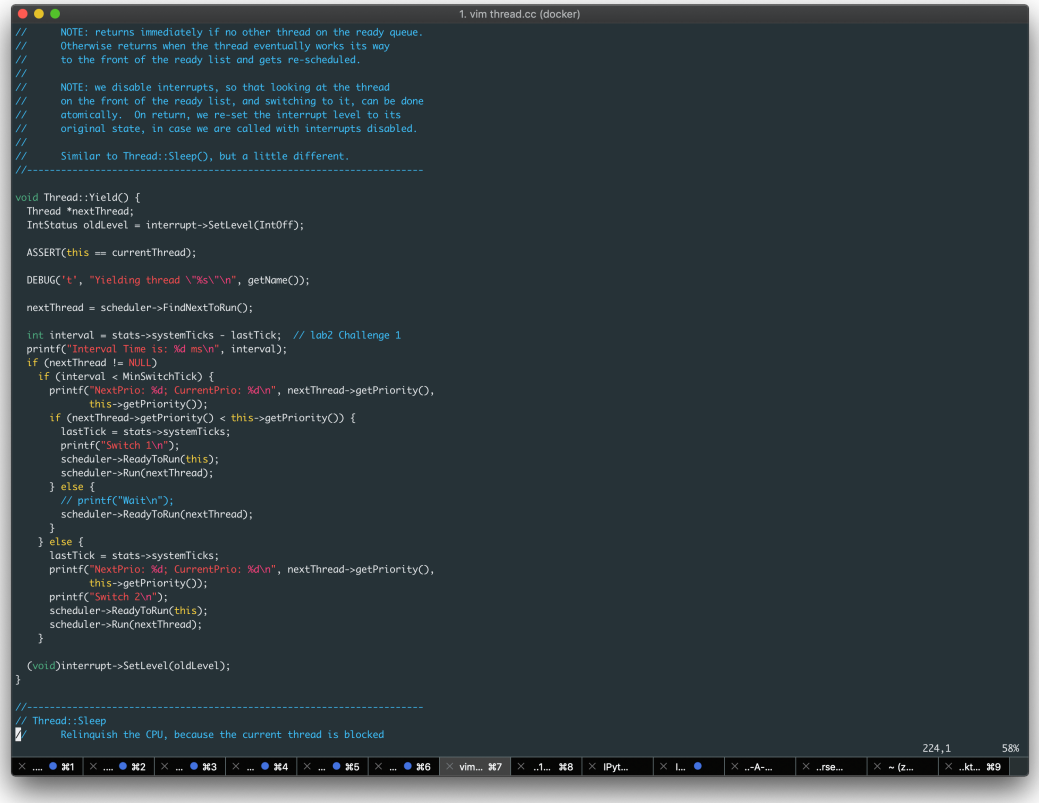


Challenge 1

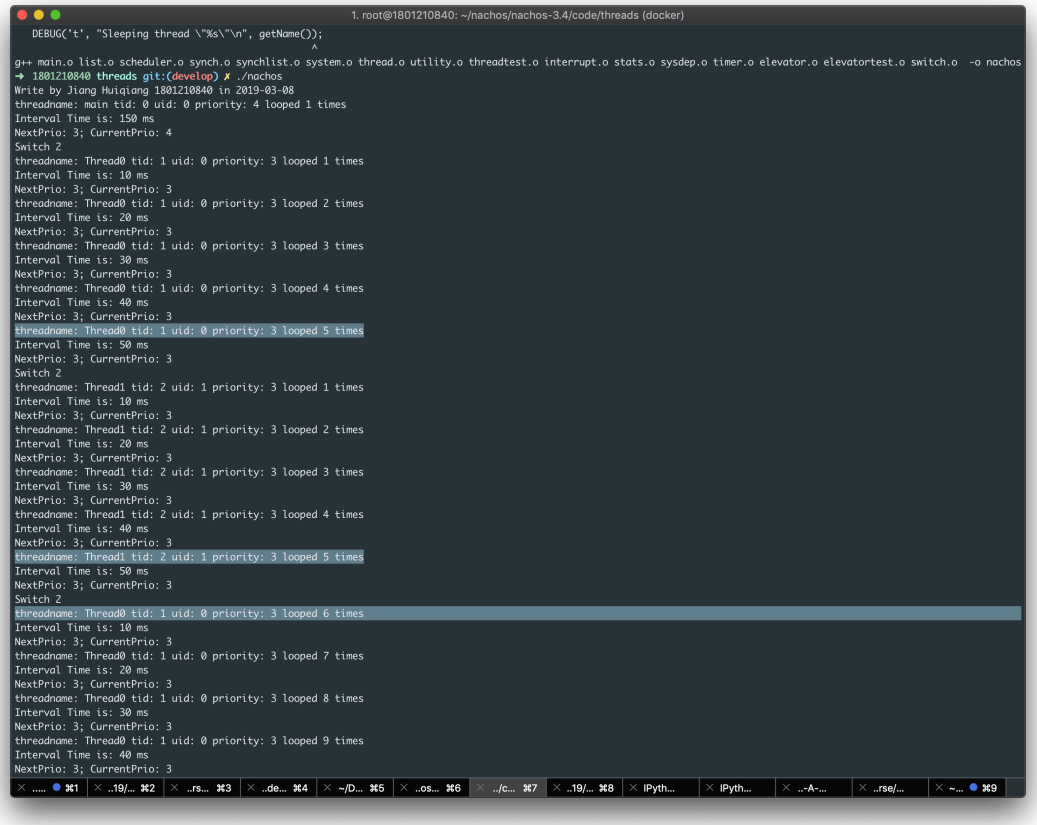
需要实现一个带时间片的抢占调度算法，即时间片未走完的时候，可以根据优先级进行抢占，如果时间片结束了，同级别的进程交替进行。

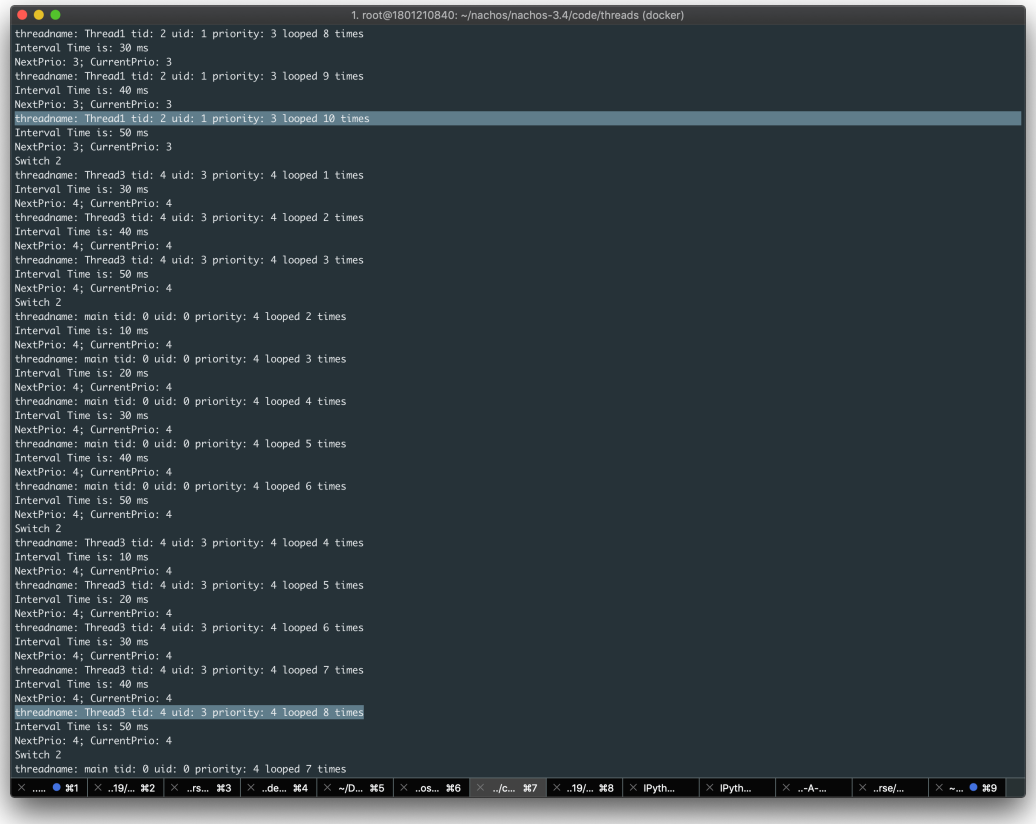
需要维护一个上次 `switch` 时间戳，如果时间差小于时间片，则需要优先级大于才能抢占；如果时间差大于时间片，则可以同级抢占。

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



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。





内容三：遇到的困难以及解决方法

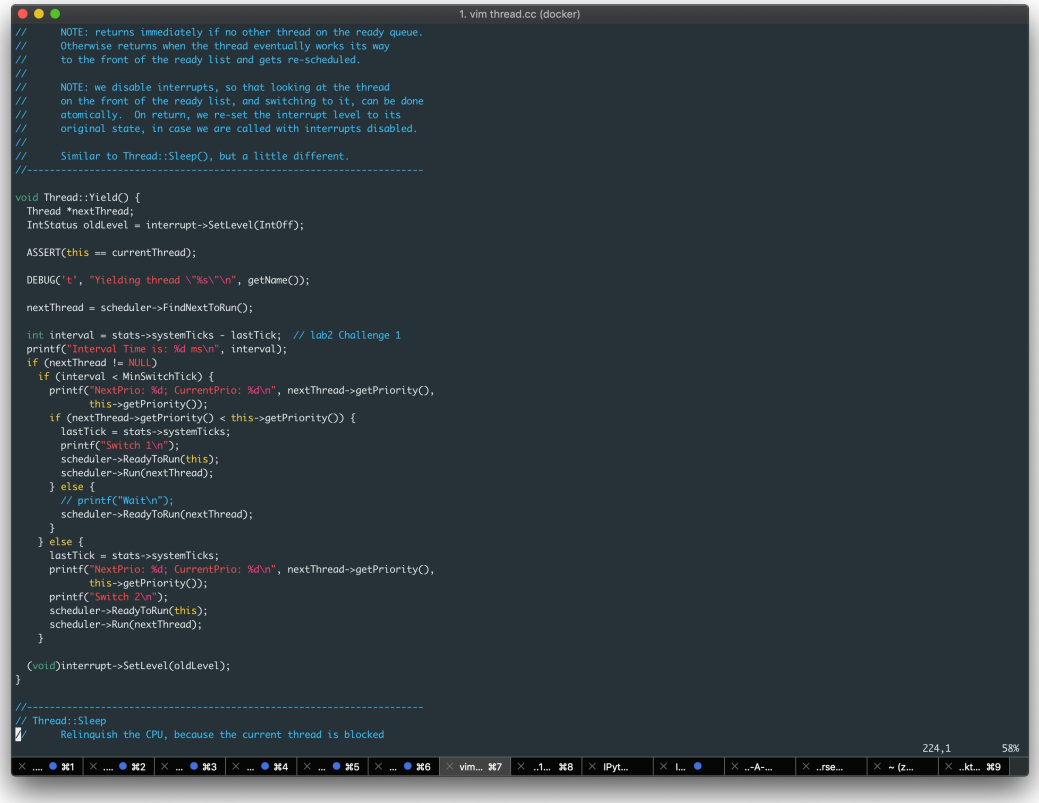
在实现时间片调度算法的时候，一开始的策略是修改 schedule.cc 中的 Thread *Scheduler::FindNextToRun() { return (Thread *)readyList->Remove(); }

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。



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）