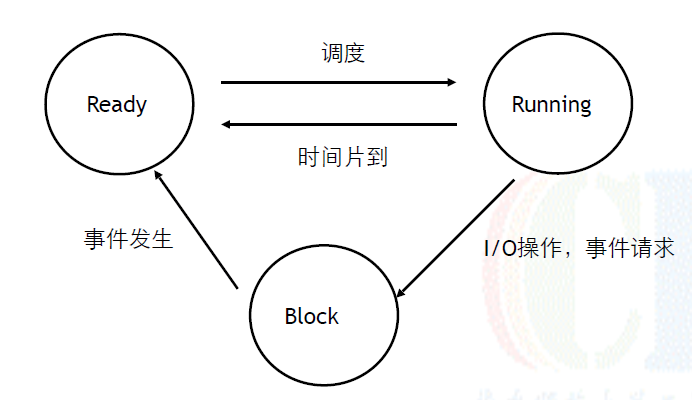
**Project 1 Thread-设计与实现文档**

一、任务需求分析：

Thread 最终任务就是在threads/中跑make check的时候， 27个test能够全部pass

Alarm-Clock：pintos中的线程状态只有两个，就绪态（ready）和运行态（running）

需要给线程添加一个新的状态，阻塞态（block）



Priority Scheduling：

pintos现有的调度机制是采用的FIFO（先进先出）现在需要修改使之实现基于优先级的调度机制。

1.就绪队列中的优先级最高的线程被优先调用

2.当就绪队列中线程优先级比正在运行的线程优先级高时，当前运行线程主动放弃CPU，调用yield()。即可抢占式。

3.当线程等待的锁、信号量或者其他条件得到满足时，阻塞队列中最高优先级的线程被唤醒。

Advanced Scheduler：pintos现有的调度机制是基于时间片的调度方法（轮转法RR）

在任务二中实现了基于时间片的优先级调度，不过该调度方法容易引起线程饥饿。希望能够实现多级反馈队列调度。

1.多个优先级队列，队列间存在优先级关系。

2.优先级队列固定时间片，时间片到后仍未完成的进入下一队列。

3.优先级队列依据先进先出原则，从队列出口处取出线程，分给时间片进行调度。

\*要明确线程优先级的更新策略并且每次更新覆盖所有进程，即更新所有进程的优先级。

\*队里中线程顺序的管理，要明确每次的更新时间，并且更新后需要更改队列中线程的顺序。

二、设计方案

Alarm-Clock：

调用timer\_sleep的时候直接把线程阻塞掉，然后给线程结构体加一个成员ticks\_blocked来记录这个线程被sleep了多少时间，然后利用操作系统自身的时钟中断（每个tick会执行一次）加入对线程状态的检测，每次检测将ticks\_blocked减1，如果减到0就唤醒这个线程。

Priority Scheduling：

1.就绪队列按优先级排序。

（threads/thread.c thread\_unblock）；

2.被上锁时，捐赠优先级；锁被打开时，恢复初始优先级。

（threads/synch.h synch.c）；

3.两个函数的使用（threads/thread.c）：

thread\_set\_priority():给当前进程赋予新的priority，如果当前进程不再拥有最高优先级时，立即使用系统调用yield()进程主动放弃CPU。

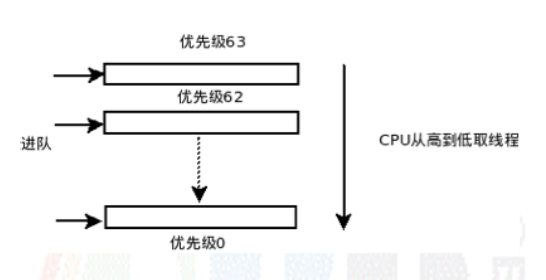
thread\_get\_priority():返回当前进程的priority（不允许修改）

Advanced Scheduler：

1.根据优先级声明64个队列，CPU调度时从高优先级队列开始。

2.CPU仍基于时间片调度，时间片完成的线程yield()放弃CPU，再调度最高优先级队列中第一个进程。

3.每个线程根据已经执行的情况，调整自己的优先级。



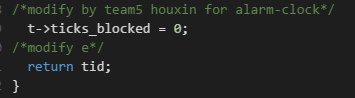
4.浮点运算支持

三、具体实现

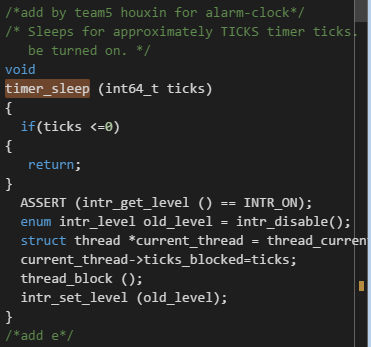
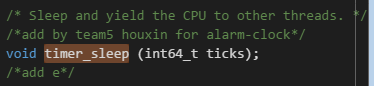
1.线程的结构体中添加ticks\_blocked成员（thread.h）：



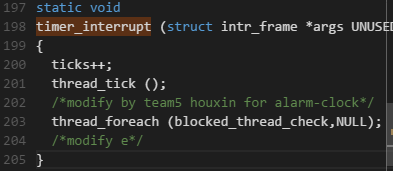
2.初始化ticks\_blocked=0在thread\_create函数中（thread.c）



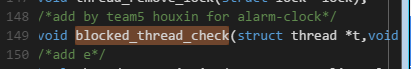
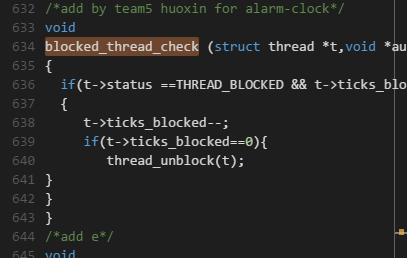
3.timer\_sleep函数声明(timer.h)以及实现（timer.c）：



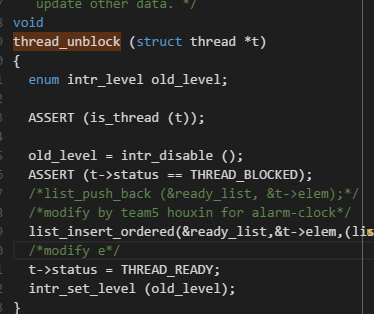
4. timer\_interrupt中加入线程sleep时间的检测（timer.c）



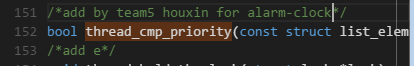
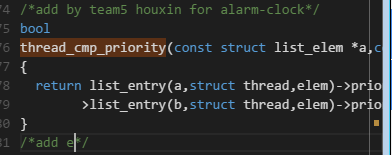
1. thread.c中实现blocked\_thread\_check以及thread.h中声明



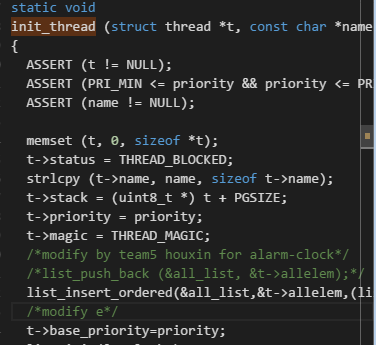
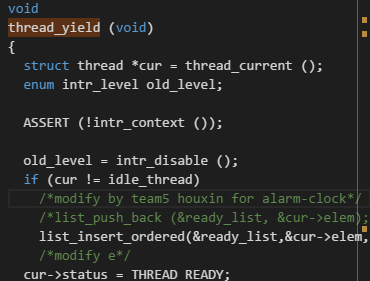
6.thread\_unblock函数中修改list\_push\_back（thread.c）



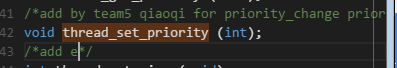
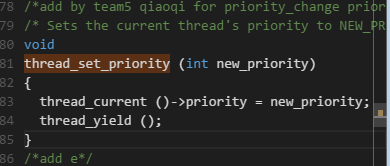
1. thread.c中实现比较函数thread\_cmp\_priority以及thread.h中声明



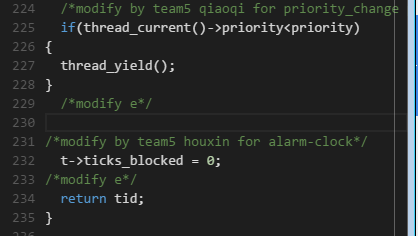
1. 对thread\_yield与init\_thread里的list\_push\_back做同样修改（thread.c）



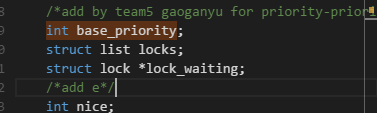
1. 在thread.c中实现thread\_set\_priority以及thread.h的声明



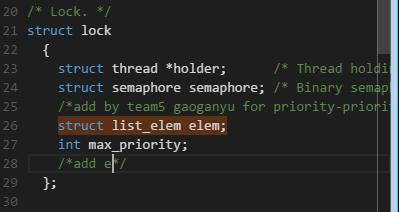
1. thread\_create函数中做修改（thread.c）



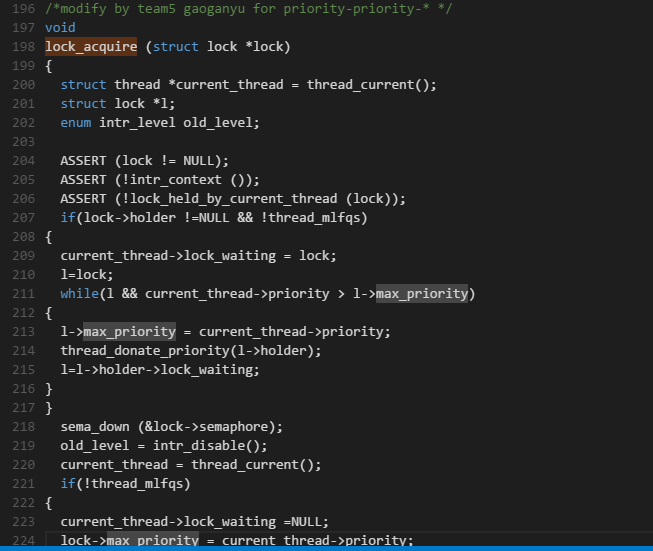
1. thread.h struct中添加成员



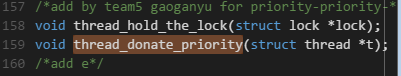
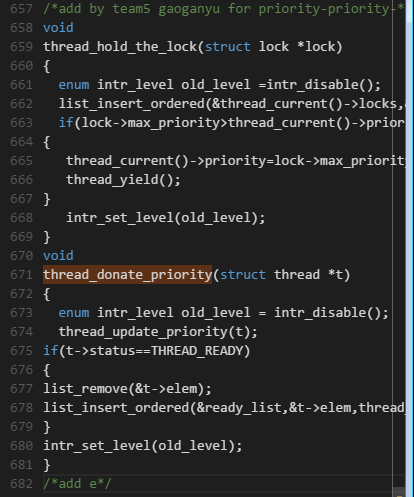
1. synch.h中添加成员



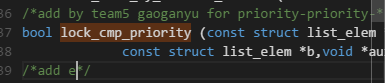
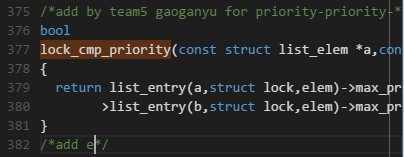
1. 修改lock\_acquire（synch.h）



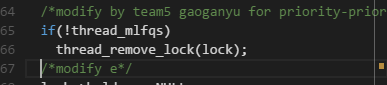
1. thread\_hold\_the\_lock和thread\_donate\_priorit在thread.c中的实现和thread.h的声明



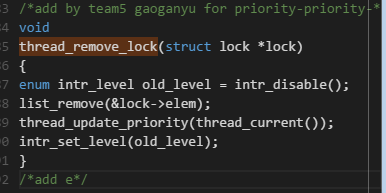
1. lock\_cmp\_priority函数实现（synch.c）与声明（synch.h）



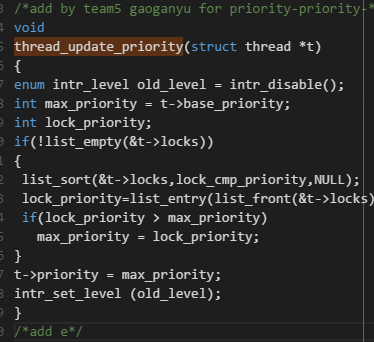
16.lock\_release函数中添加语句



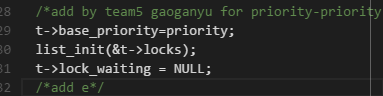
17.thread\_remove\_lock实现在thread.c，声明在thread.h中



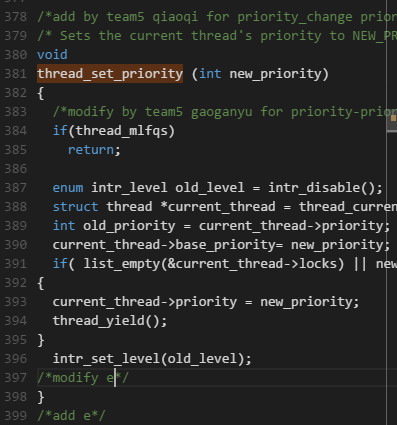
1. thread\_update\_priority函数



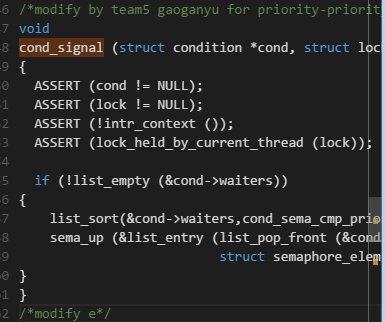
19.static init\_thread中加入初始化



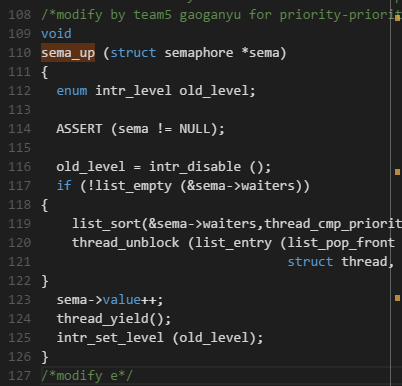
1. 对9中实现的thread\_set\_priority函数进行修改



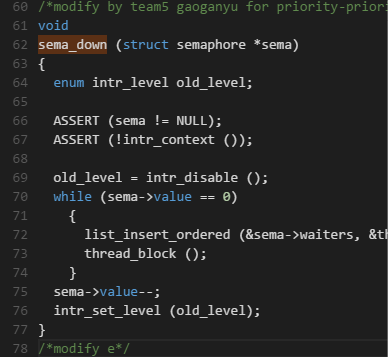
1. cond\_signal函数的修改



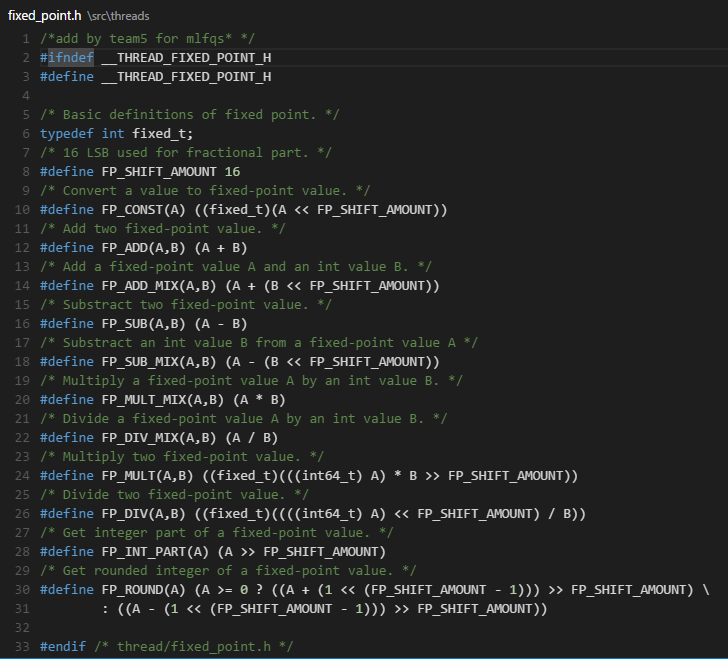
22.sema\_up函数做出修改（synch.c）



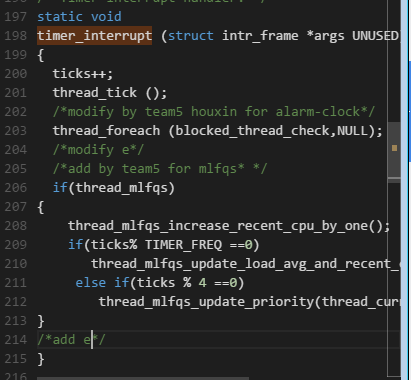
1. sema\_down函数修改（synch.c）



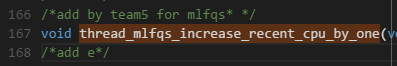
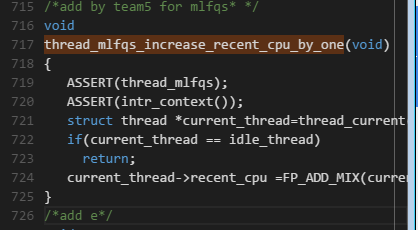
1. 在thread/目录下创建fixed\_point.h并实现浮点运算逻辑



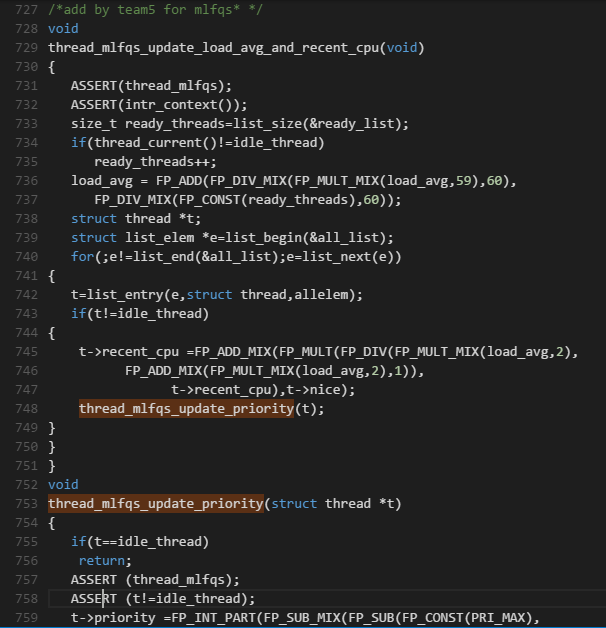
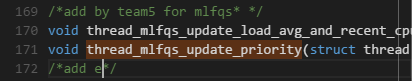
25.对4中的timer\_interrupt函数实现补充



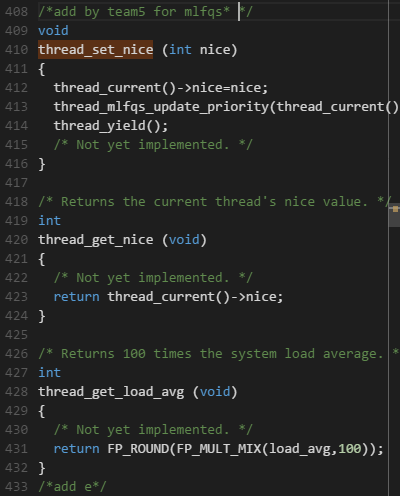
26.thread\_mlfqs\_increase\_recent\_cpu\_by\_on实现（thread.c）声明（thread.h）



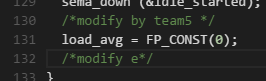
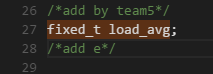
27.同26，实现声明thread\_mlfqs\_update\_priority与thread\_mlfqs\_update\_load\_avg\_and\_recent\_cpu



28.thread\_set\_nice函数相关操作

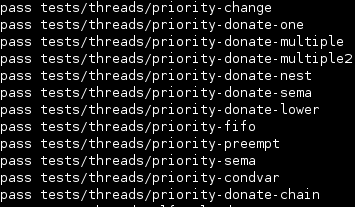
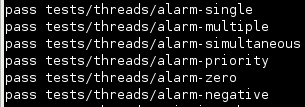


29.fixed\_t load\_avg添加到thread.c并在thread\_start中初始化



四、调试

运行结果，27个tests全部通过



五、实现结果

完成了project1的三块内容，Alarm-Clock，Priority Scheduling以及Advanced Scheduler。

project1是关于线程的实验，我们重新实现了timer\_sleep函数、优先级调度以及多级反馈队列。

六、总结

关于优先级调度的实现，就是我们在插入线程到就绪队列的时候保证这个队列是一个优先级队列即可。thread\_unblock,init\_thread以及thread\_yield都可能会引起一个线程被丢到就绪队列中。所以我们只要在扔的时候维持这个就绪队列是优先级队列即可。1.在设置一个线程优先级要立即重新考虑所有线程执行顺序，重新安排执行顺序。直接在线程设置优先级的时候调用thread\_yield即可，这样就把当前线程重新丢到就绪队列中继续执行，保证了执行顺序。如果在创建进程的时候，新进程比主进程优先级高的话也要调用thread\_yield。2.在一个线程获取一个锁的时候，如果拥有这个锁的线程优先级比自己低就提高它的优先级，然后在这个线程释放掉这个锁之后把原来拥有这个锁的线程改回原来的优先级。释放一个锁的时候，将该锁的拥有者改为该线程被捐赠的第二优先级，若没有其他捐赠者，则恢复原始优先级。

关于多级反馈调度的实现：使用多级反馈调度，可以减少系统的平均响应时间。实验中，系统维持了64个队列，每个队列对应一个优先级，通过一些公式计算出线程当前的优先级，系统调用的时候会从高优先级队列开始选择线程执行， 线程的优先级随着操作系统的运转数据而动态改变。（这里涉及到浮点数的运算问题）