****

****

**Pintos Project 1 设计方案**

**目录**

1. 成员信息及工作总结

1.1 小组成员信息

1.2 工作总结

1.3 GitHub工作截图

1.4 测试点情况

1. 需求分析

2.1 Alarm Clock 需求分析

2.2 Priority Scheduling 需求分析

2.3 Advanced Scheduler 需求分析

1. 设计思路

3.1 Alarm Clock 设计思路

3.1.1 设计思路

3.1.2 优化思路

3.1.3 主要难点

3.2 Priority Scheduling 设计思路

3.3 Advanced Scheduler 设计思路

1. 成员信息及工作总结

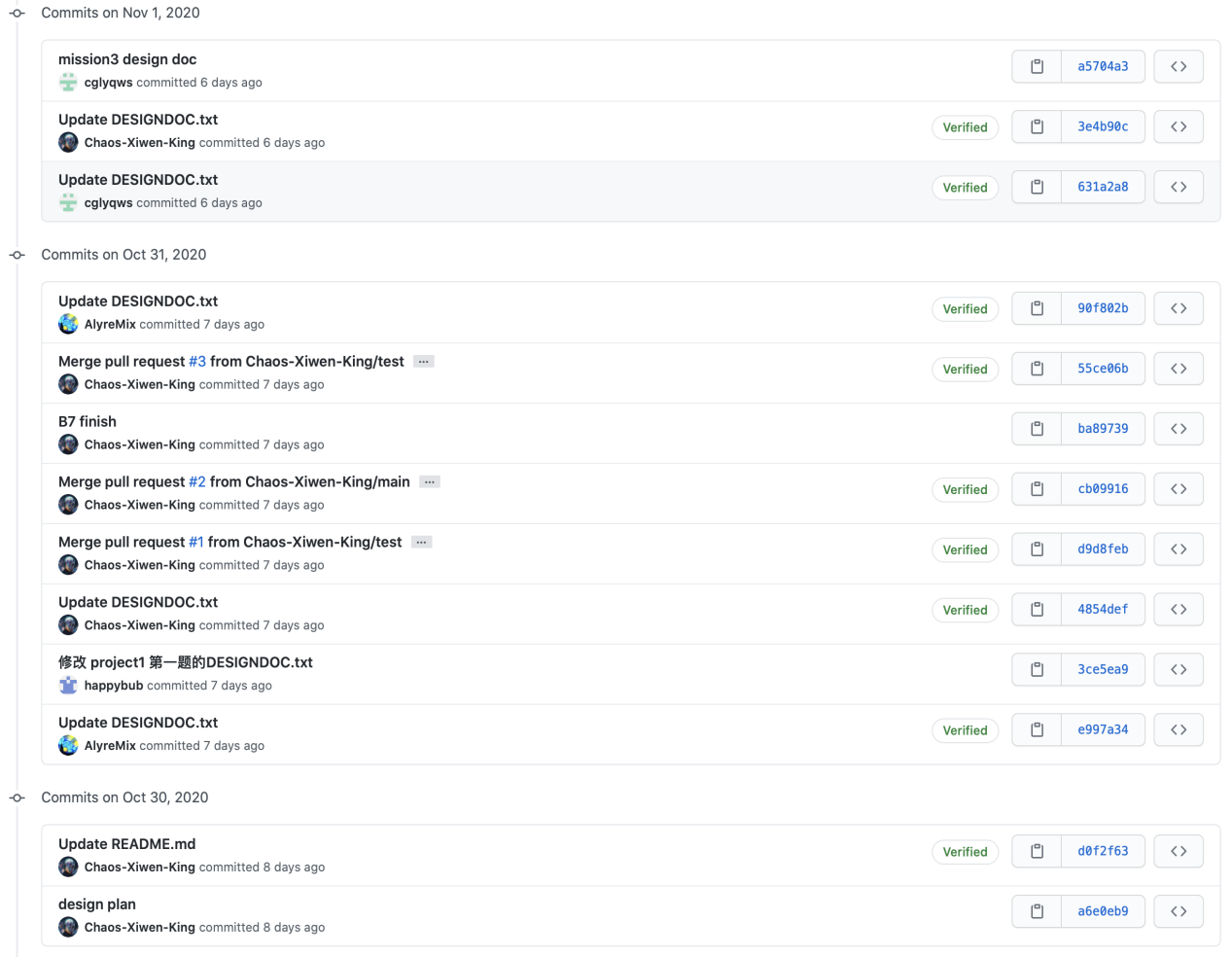
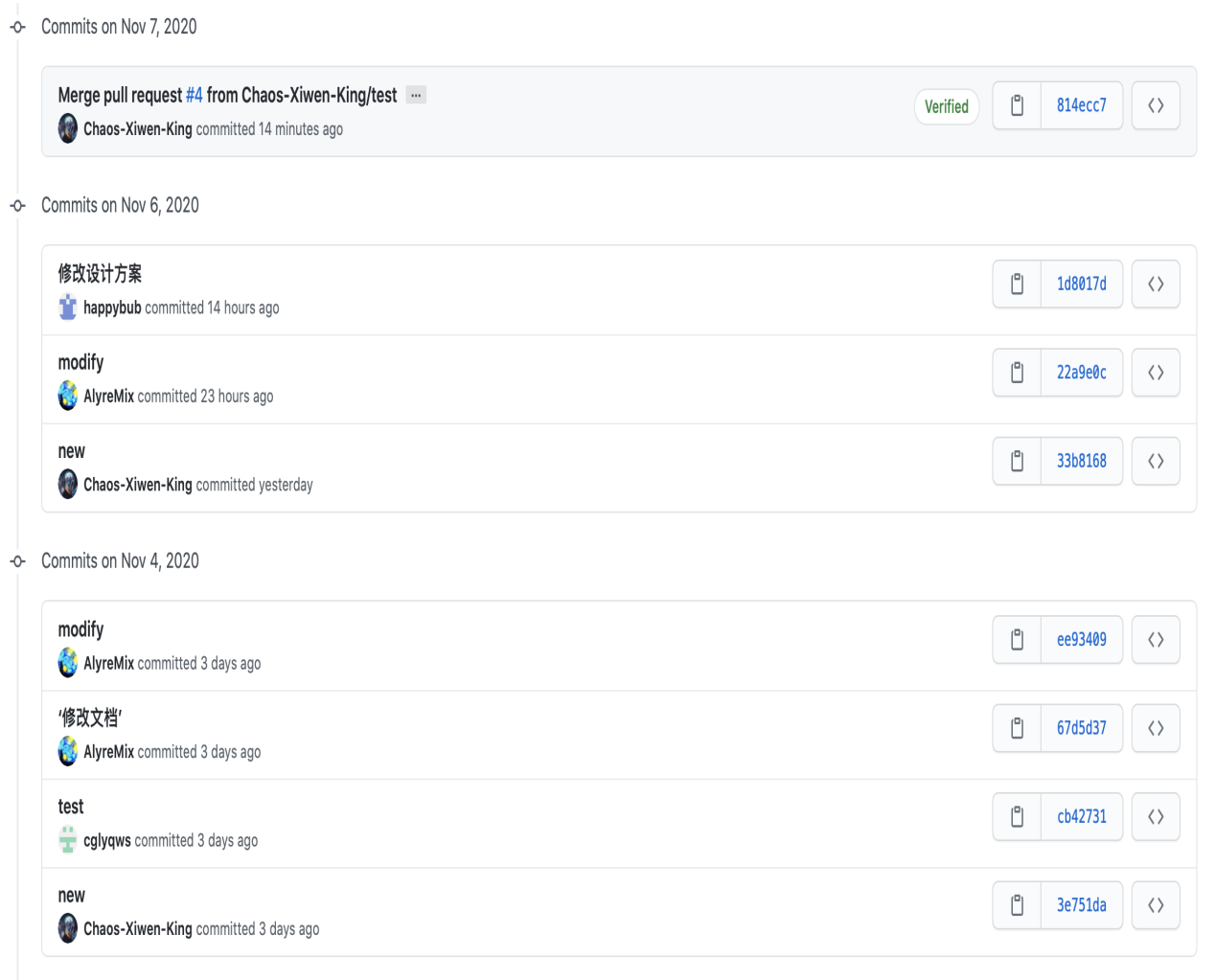
1.1 小组成员信息

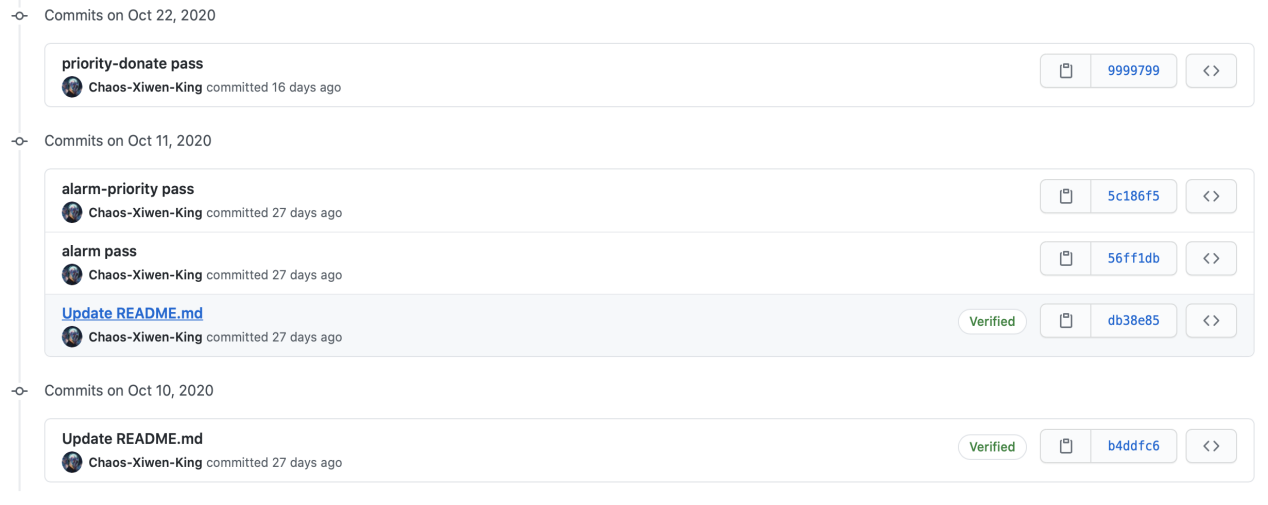
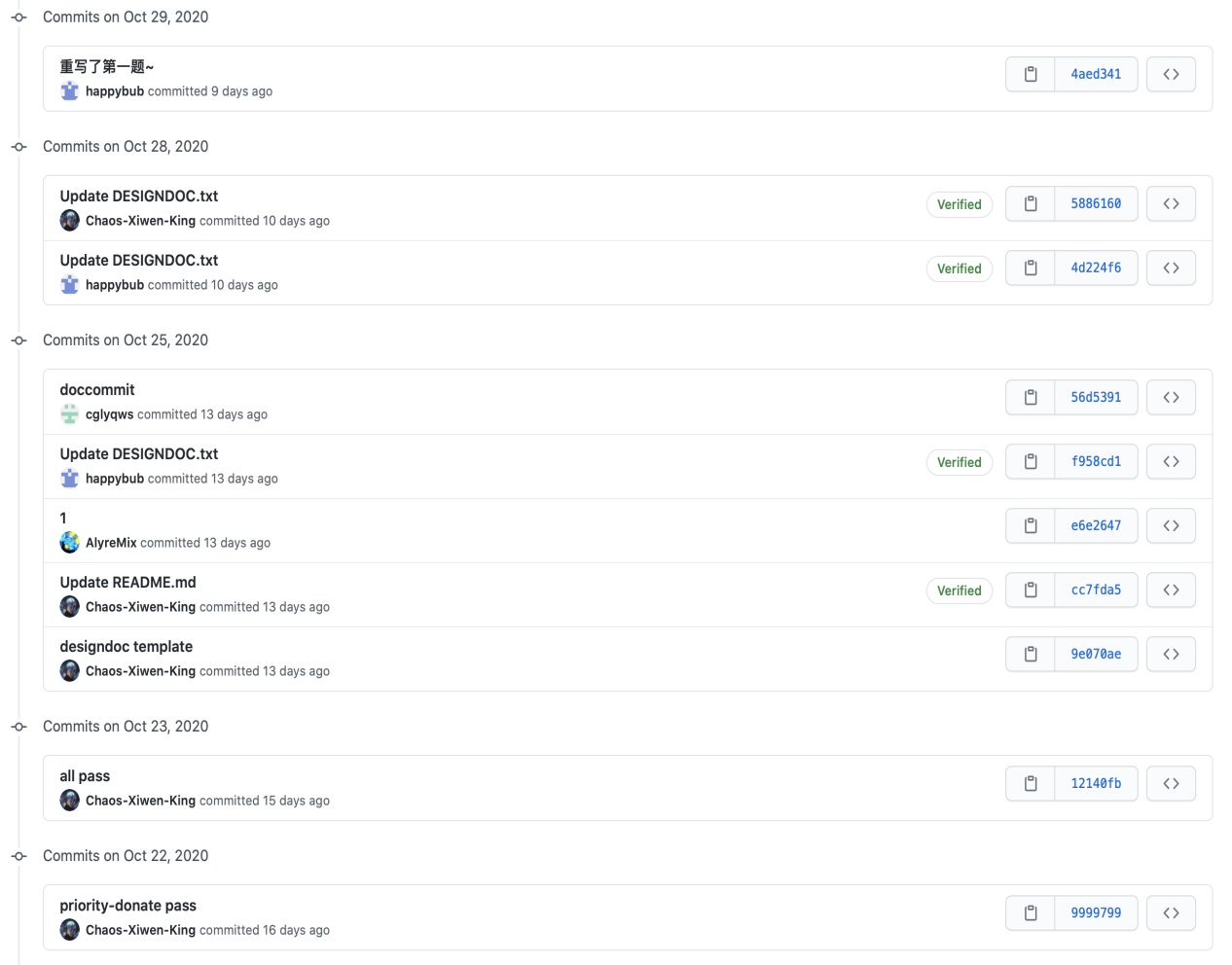
|  |  |  |
| --- | --- | --- |
| 姓名 | 学号 | 贡献 |
| 康曦文 | 18373607 | 100 |
| 郭凌宇 | 1837 | 100 |
| 陈宇畅 | 18373609 | 100 |
| 赵致远 | 18373669 | 100 |

1.2 工作总结

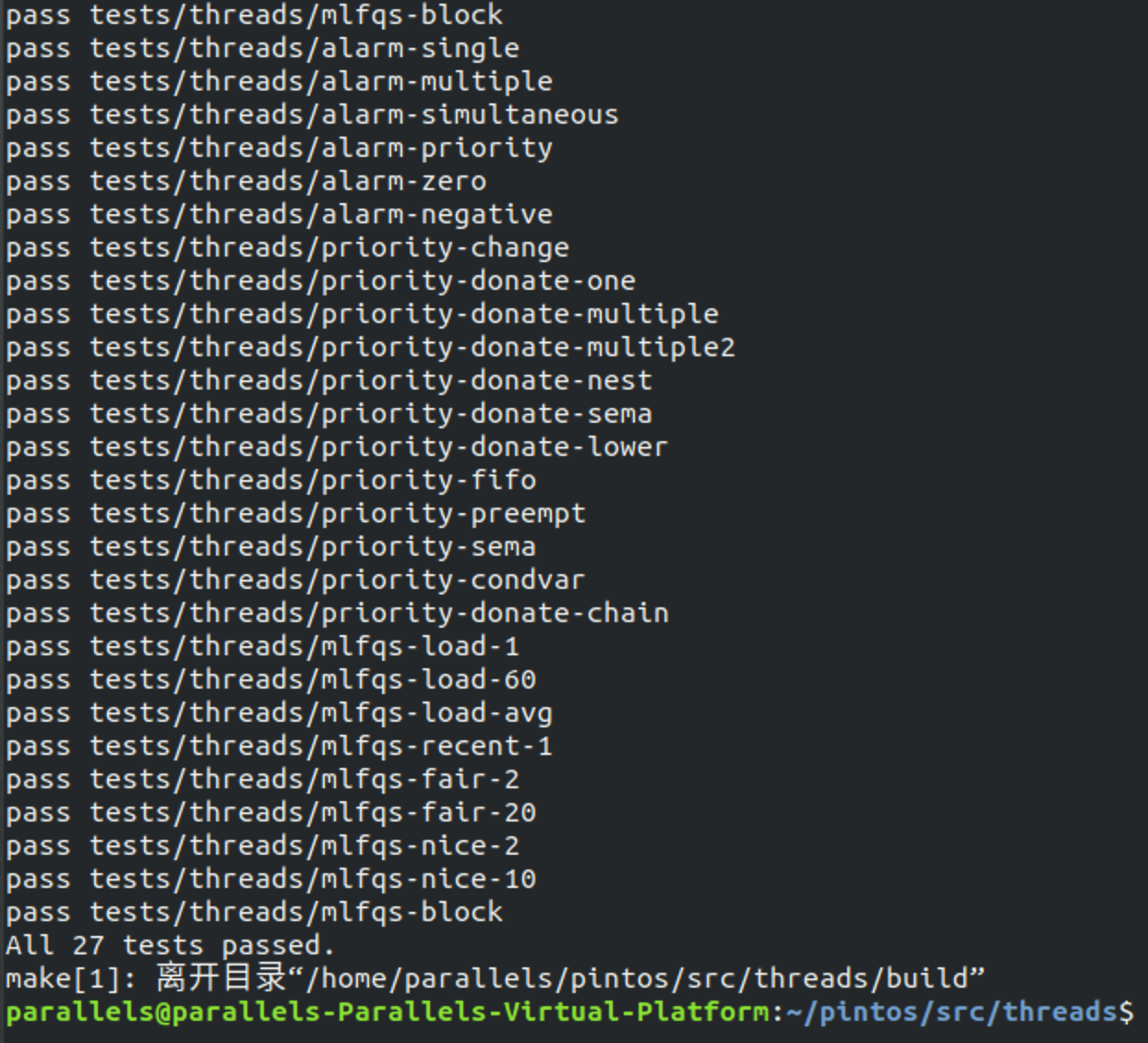
|  |  |
| --- | --- |
| 姓名 | 工作内容 |
| 康曦文 | Alarm Clock、Priority Scheduling、Advanced Scheduler 的代码部分 |
| 郭凌宇 | Alarm Clock 的代码部分、Alarm Clock的Pintos文档和设计文档 |
| 陈宇畅 | Priority Scheduling的Pintos文档和设计文档 |
| 赵致远 | Advanced Scheduler的Pintos文档和设计文档 |

1.3 GitHub工作截图





1.4 测试点情况



1. 需求分析

2.1 Alarm Clock 需求分析

在这个任务中，我们需要对原先的timer\_sleep()函数进行优化。其主要目的是：避免线程不断将自己放进休闲队列中而降低CPU的运行效率。我们需要设计一定的算法，用其他方式来实现线程的休眠，将线程状态变成BLOCKED取代原先原先的READY。

2.2 Priority Scheduling 需求分析

在这个任务中，我们需要解决的问题是，当拥有高优先级的线程等待低优先级的线程时，高优先级线程永远不会得到CPU因为低优先级在有更高优先级的情况下得不到任何CPU时间。

2.2.1 优先级队列以及抢占式调度

在分析timer\_sleep的时候我们已经知道，为了实现优先级调度，最关键的是要实现维持就绪队列为一个优先级队列。而在程序中有一个list\_insert\_ordered的函数，因此，我们要修改这个函数让所有进入就绪队列中的线程都能够实现优先级排序。

接着，我们分析priority-change和priority-preempt这两个测试点，从中得出结论，在设置一个线程的优先级时应该考虑所有线程的执行顺序，重新安排执行顺序，防止出现浪费CPU时间的问题。而通过调用thread\_yield函数我们能够将线程放进就绪队列中。

2.2.2 优先级捐献

继续分析测试代码，发现，当优先级低的线程先运行的话，会产生优先级翻转的问题，造成高优先级任务阻塞，而这也是课程中提出的问题。为了解决这个问题，应当在高优先级任务因为低优先级任务占用资源发生阻塞时，将低优先级任务的优先级提升到它占有资源的最高优先级任务的优先级。

那么在多锁情况下，还应该考虑到优先级逻辑的正确性。在测试点priority-donate-multiple中我们要实现的便是这个，而对应的实现思路为：释放一个锁，将该锁的拥有者改为被捐赠的第二优先级，若没有其他的捐赠者，则改为原始的优先级。

分析priority-donate-nest测试点，在这个测试点中我们要实现的也是课程中提到的优先级嵌套问题，优先级应该具有连环效应，当有多个优先级线程嵌套时，应该提升到和最高优先级线程一样的优先级。因此我们需要知道当前线程被锁于哪个线程。

看下一个测试点priority-donate-sema，在这之中有信号量和锁的混合触发，即信号量在这里起到了作用。而在priority-sema中，也是优先级高的先唤醒，因此对于信号量队列来说，我们也应该实现为优先队列。

分析priority-condvar测试点，在里面有一个waiters队列，而在唤醒的时候，也是按照优先级顺序唤醒，因此condition的waiters队列我们也应该实现为优先队列。

分析最后一个测试点priority-donate-chain，这个测试点测试的仍然为优先级捐赠逻辑的正确性，而在这里有一个逻辑：当释放锁时，若优先级发生改变，则可以发生抢占。这个点是需要我们来实现的。

2.3 Advanced Scheduler 需求分析

此任务是实现一个多级反馈队列调度算法（mlfqs），其的主要目的是：根据不同CPU区间的特点以区分进程，将占用过多CPU时间的进程转移到更低优先级队列，以保证I/O约束和交互进程等占用CPU时间较少的线程留在优先级更高的队列，从而减少系统的平均响应时间。

为了实现这一功能，我们需要设计一定的算法，将占有CPU时间过长的进程转移到优先级低的队列中。

1. 设计思路

3.1 Alarm Clock 设计思路

3.1.1 设计思路

在原先的timer\_sleep()中，核心算法如下：

while (timer\_elapsed (start) < ticks)

thread\_yield ();

函数在线程应该休眠的时间里不断调用thread\_yield()函数，将该线程放入等待队列ready\_list中，如果schedule()函数再次调度到该线程，就会再次进行判断，如果时间仍未到达预设休眠时间，则会再次重复上述过程。该做法有着非常明显的缺点：线程调度大量的消耗CPU的资源，许多时间浪费在了没有意义地调度上。

因此，我们设计了一个全新的算法来实现timer\_sleep()函数。当线程需要进入休眠时，我们调用thread\_block()将线程阻塞，当需要唤醒线程时再调用thread\_unblock()将其放入schedule() 函数负责调度的ready\_list中，这样可以避免原算法无意义调度的问题。

核心算法如下：

enum intr\_level old\_level = intr\_disable ();

thread\_block();

intr\_set\_level(old\_level);

其中，禁止中断是thread\_block()函数所要求的条件。

我们还需要一个方法能够在适当的时刻调用thread\_unblock()将线程唤醒。注意到线程休眠时间以ticks为单位，而ticks是由时间中断例程timer\_interrupt()所修改，因此我们在线程结构体中加入了blocked\_ticks成员变量，标记该线程需要停止休眠的时间，并在调用thread\_block()函数之前将其设置为ticks + tick（timer\_sleep()参数）。在timer\_interrupt()中分析所有线程的blocked\_ticks参数，若blocked\_ticks == ticks，说明该线程休眠时间已到，调用thread\_unblock()函数唤醒该线程。

3.1.2 优化思路

在上述算法中，还存在着可以优化的地方。注意到timer\_interrupt()是一个频繁调用的函数，我们应当尽可能的减少timer\_interrupt()所运行的时间。

我们采取的做法是：建立了一个和ready\_list相似的blocked\_list，当线程调用timer\_sleep()时将线程放入blocked\_list中（按照blocked\_ticks从小到大的顺序），在timer\_interrupt()中遍历blocked\_list，当ticks > blocked\_ticks时就可以停止遍历。这样，在时间中断例程中，就不需要遍历所有线程的列表，只需要遍历因为调用time\_sleep()而被阻塞的线程，同时，在blocked\_list中是blocked\_ticks有序的，意味着时间中断例程只会遍历需要唤醒的线程(blocked\_list == ticks)，大大减小了timer\_interrupt()所运行的时间，达到了优化的目的。

3.1.3 主要难点

在设计思路中，主要难点集中于对于时间中断处理函数的利用。正是由于每个时间周期timer\_interrupt()都会被调用，才使得我们算法的实现成为可能。

在一开始的算法设计中，我们将blocked\_ticks设计为剩余睡眠的时间，但是与随后的优化发生了冲突。于是我们将其改为需要被唤醒的时间，但是这会产生一个新的问题：ticks + tick有可能会超出int64的储存范围。尽管这个可能性微乎其微，但我们还是加入了一个断言来确保不会发生错误。

在一开始的优化中，我们将插入blocked\_list的操作放在了thread\_block()中，但是，并不只有timer\_sleep()函数会调用thread\_block()，这样做会使得列表中插入因为其他原因而被阻塞的线程，这就会影响到blocked\_list列表删除操作。于是我们在插入blocked\_list前做了一个判断，只有当blocked\_ticks>0时，才代表着该操作是由timer\_sleep引起的。

3.2 Priority Scheduling 设计思路

数据结构设计如下：

在thread数据结构中加入成员init\_priority, locks, lock\_waiting，其中init\_priority记录创建时的优先级，locks记录线程所拥有的锁，lock\_waiting记录的时线程申请的锁。

在lock数据结构中加入成员elem, max\_priority成员，其中elem记录等待该锁的线程，max\_priority为上述队列中的最大优先级。通过添加以上成员记录线程捐献优先级的过程。

3.2.1 优先级队列以及抢占式调度的实现

我们实现比较函数thread\_cmp\_priority：

bool

thread\_cmp\_priority (const struct list\_elem \*a, const struct list\_elem \*b, void \*aux UNUSED)

{

return list\_entry(a, struct thread, elem)->priority > list\_entry(b, struct thread, elem)->priority;

}

这个比较函数配合list\_insert\_ordered，我们就可以实现线程的优先级队列，保证就绪队列实现优先级排序。

我们实现thread\_set\_priority函数，直接对线程设置优先级时调用thread\_yield，将线程扔入就绪队列中。

同时，在创建新线程时，如果新线程优先级比主线线程还高，则也应该将新线程扔入就绪队列中，即调用thread\_yield。

3.2.2 优先级捐献的实现

修改lock\_acquire函数，添加如下内容：

if (lock->holder != NULL && !thread\_mlfqs)

{

current\_thread->lock\_waiting = lock;

l = lock;

while (l && current\_thread->priority > l->max\_priority)

{

l->max\_priority = current\_thread->priority;

thread\_donate\_priority (l->holder);

l = l->holder->lock\_waiting;

}

}

添加这段内容让程序能够递归地实现优先级地捐赠。并在后面当前线程得到现在的锁，改变锁的优先级。

添加thread\_donate\_priority和thread\_hold\_the\_lock函数，thread\_donate\_priority函数作用是线程将优先级捐赠给其他线程，并更新其他线程所拥有的锁的优先级，thread\_hold\_the\_lock函数则是让线程拥有锁，并将线程插入锁的elem队列当中，按照优先级从大到小进行排序，这样满足队列中的队头永远都是优先级最大，让之后的程序实现更加方便。

为了实现优先队列，我们加入锁队列排序函数lock\_cmp\_priority，满足当a优先级大于b返回1，否则返回0。

考虑到释放锁后优先级的变化，我们添加thread\_remove\_lock来实现。其中，当一个线程释放锁后，线程就应该从lock中的elem队列中移出，并判断是否更新当前线程的优先级，而更新线程的优先级我们在thread\_update\_priority函数实现。

thread\_update\_priority实现线程优先级的更新。首先应该判断线程是否还有锁，如果没有，当前线程的优先级即为初始的优先级，否则，取出线程拥有的所有锁中的最大优先级与线程的初始优先级进行比较，去两个中的最大值设置为线程的当前优先级。

修改thread\_set\_priority函数，设置判断线程是否处于被捐赠的状态，如果处于，则对init\_priority进行设置，并且如果设置的优先级大于当前的优先级，则改变优先级，否则的话，在无捐赠状态时应该恢复init\_priority。同样，在设置时应调用thread\_yield，将线程扔入就绪队列中。

实现condition队列和信号量的等待队列为优先队列，比较函数如下：

bool

cond\_sema\_cmp\_priority (const struct list\_elem \*a, const struct list\_elem \*b, void \*aux UNUSED)

{

struct semaphore\_elem \*sa = list\_entry (a, struct semaphore\_elem, elem);

struct semaphore\_elem \*sb = list\_entry (b, struct semaphore\_elem, elem);

return list\_entry(list\_front(&sa->semaphore.waiters), struct thread, elem)->priority > list\_entry(list\_front(&sb->semaphore.waiters), struct thread, elem)->priority;

}

在程序中，我们对锁，信号量和condition队列实现为优先队列的原因是保证在等待时拥有最高优先级的线程能够被最先唤醒。

3.3 Advanced Scheduler 设计思路

根据文档，BSD Scheduler的实现思路为：每过一定的时间，重新计算线程的优先级，以降低占用过多CPU时间的线程的优先级，从而实现将其“转移到低优先级队列”。

此调度算法程序可以用以下参数来定义：

（1）队列数量：64

可以认为是每个优先级数值对应一个队列。

（2）每个队列的调度算法：FCFS

由于此64个队列，都属于ready\_list，因此采取的调度算法均为FCFS

（3）升高/降低到其他队列的算法：通过动态计算线程优先级的数值来确定线程队列的 改变。

（4）确定进程在需要服务时应进入的队列：可以认为按优先级的数值插入到对应队列。

其中，（3）的具体做法为：

首先，为每一个线程增加参数nice、recent\_cpu属性用于实现线程优先级的动态计算。设定全局变量load\_avg，用于计算recent\_cpu。

其中nice值的含义为：该线程对其他线程的友好程度，nice值越大，该线程的优先级会越低。

recent\_cpu的作用为：衡量线程所占用cpu时间的多少。

load\_avg的作用为：估计过去1分钟运行线程的平均数量。

其次，改变的优先级算法为：

① 每个时钟周期，正在运行的线程的recent\_cpu增加1。即在时间中断函数time\_interrupt() 中加入功能，每次中断为当前运行线程的recent\_cpu值增加1.

② 每过4个tick( tick % 4 == 0 )，重新计算优先级，公式为

③ 每过100个tick(tick % timer\_freq == 0),通过以下公式更新recent\_cpu和load\_avg

此外，由于pintos不支持浮点数的运算，需按照浮点数运算规则，实现浮点数的运算。