操作系统PINTOS实验PROJECT 1:THREADS

设计文档（测评方式一）

1. 小组成员以及贡献

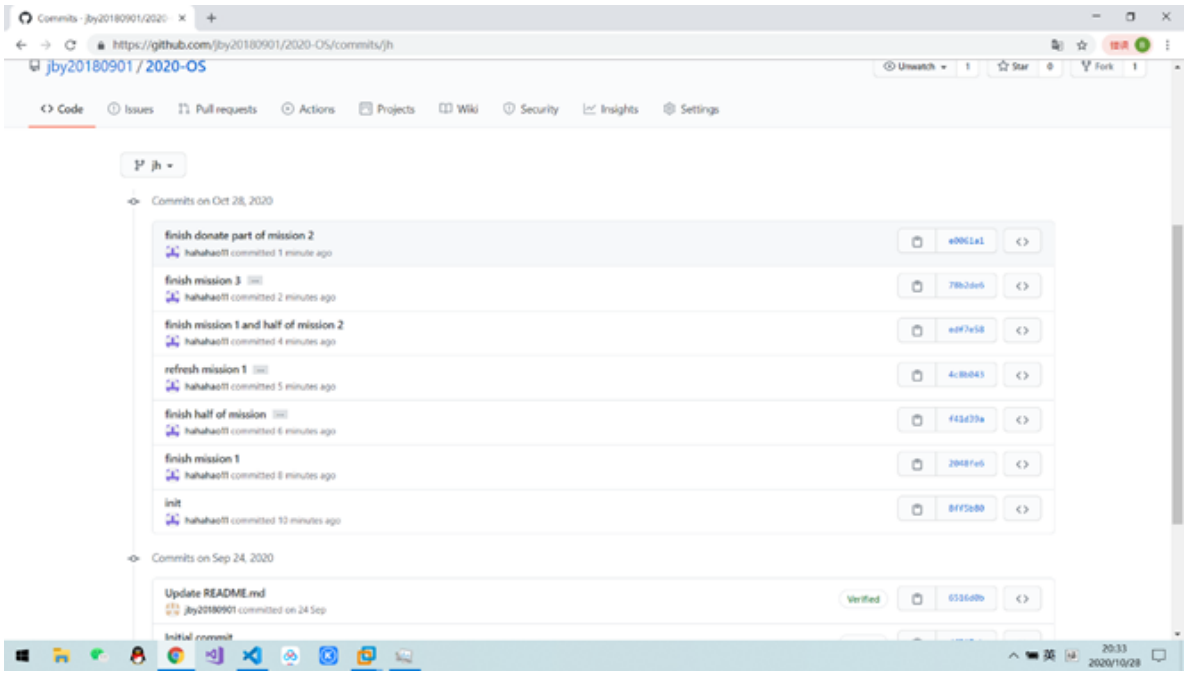
总共100%

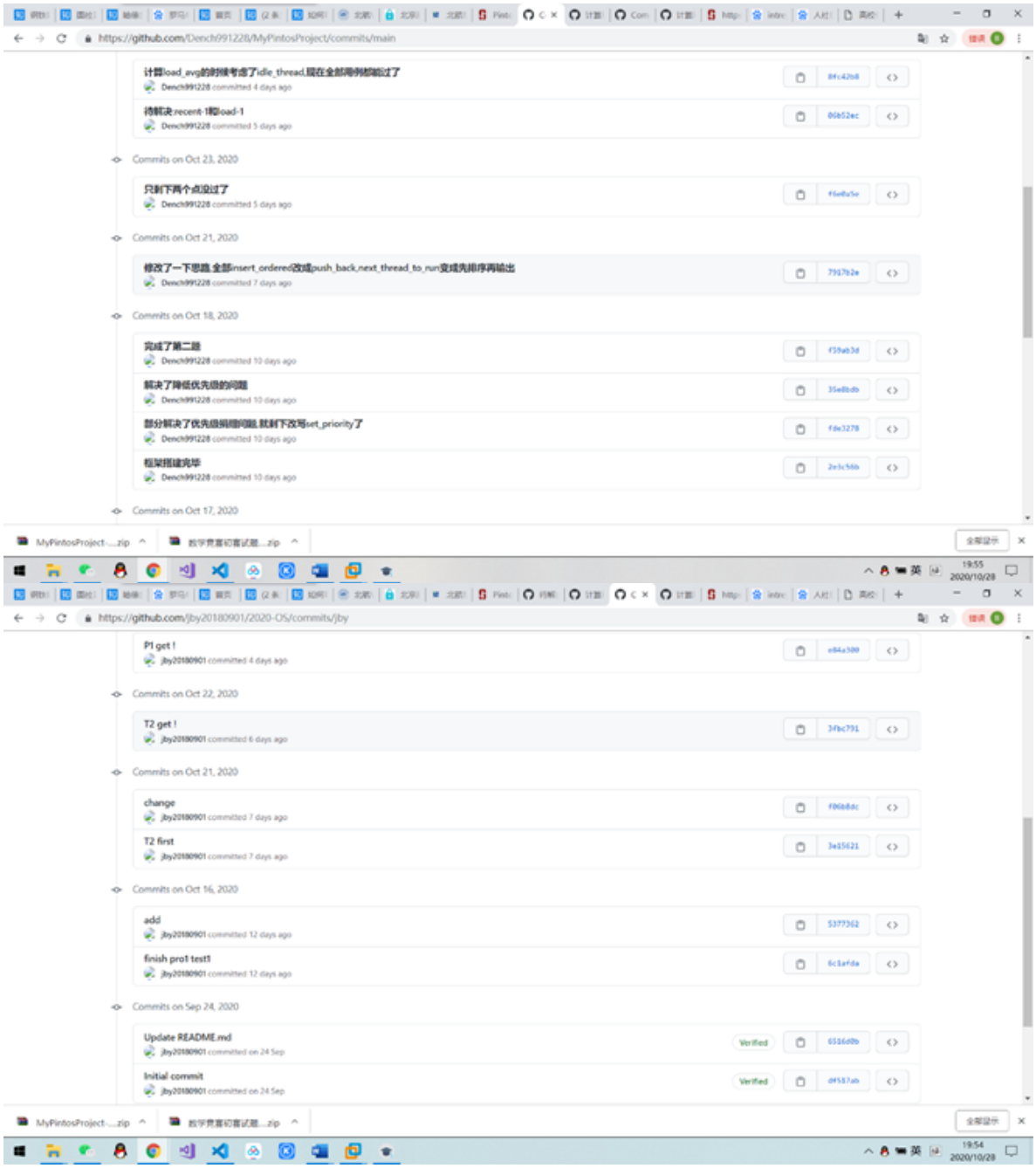
18373403 姜宝洋25%

18373570 邓涵之25%

18373133 方洁25%

18373744 姜昊25%





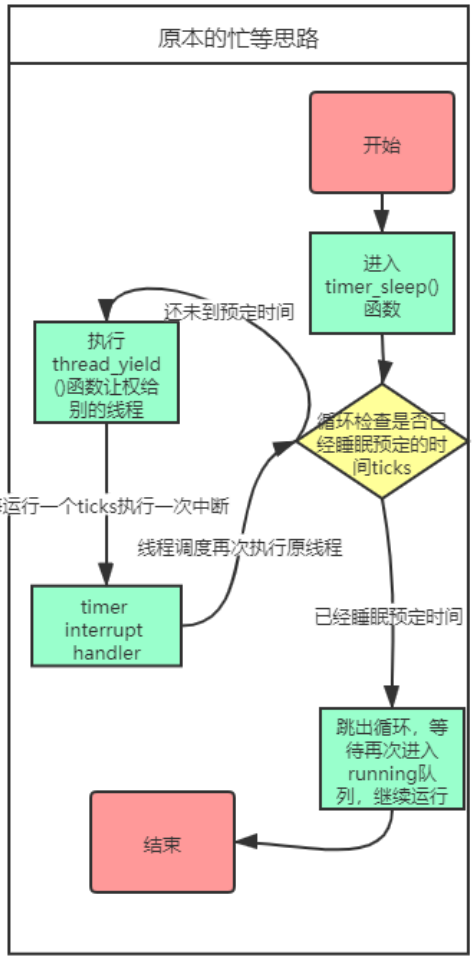
2.实验设计与相关问题

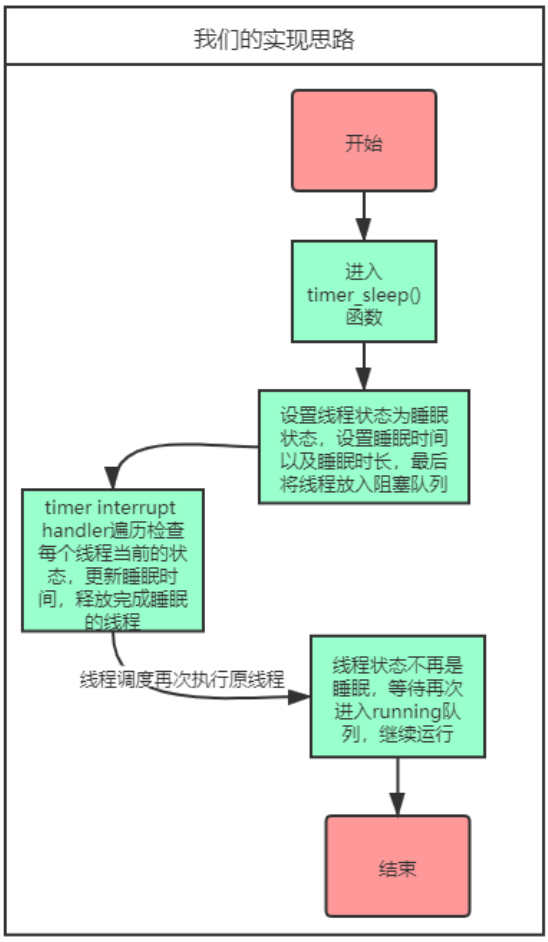
2.1 重构`timer_sleep()`

2.1.1 需求分析

对于该任务而言，需求十分明确，题目中即给出了说明——将原本`devices/timer.c`中的`timer_sleep()`函数重新实现，将原本的忙等机制改为唤醒机制，即可完成作业要求。

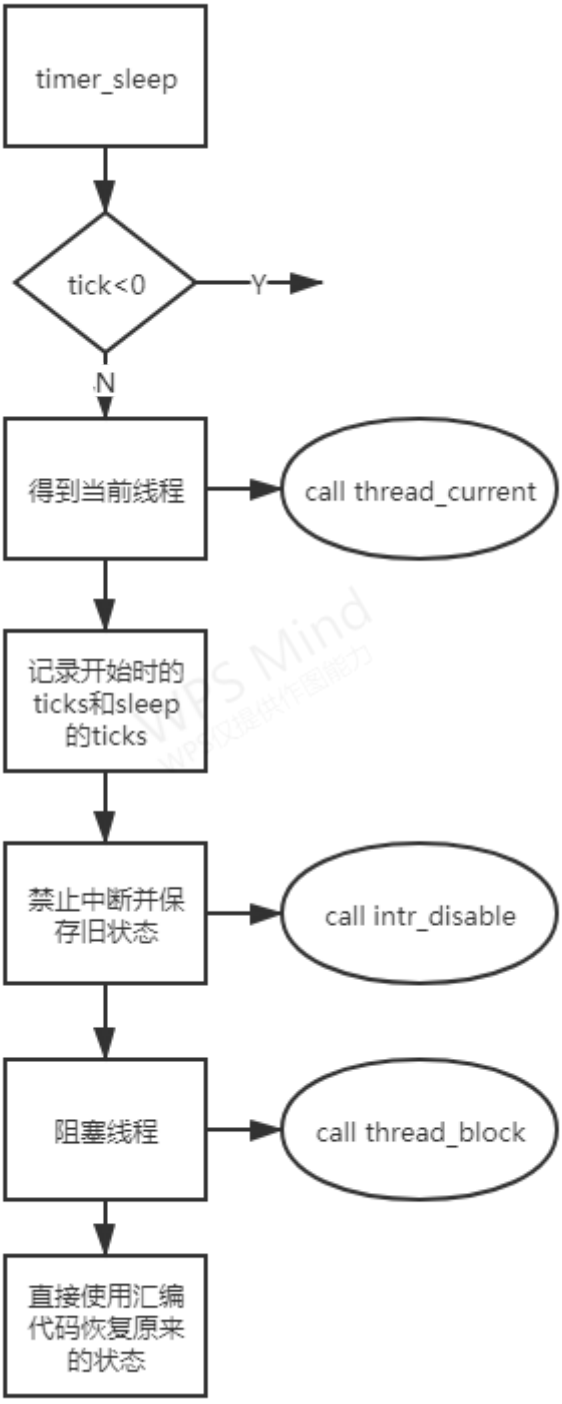
如果睡眠时间未到，线程便会不断在就绪队列以及`running`队列转换，浪费系统资源。为完成唤醒机制，需要记录每一个线程所要睡眠的时间，利用`pintos`自带的每隔一个`ticks`执行一次中断的特性，对所有线程的睡眠时间进行更新，同时如果有线程完成睡眠，将其唤醒放入就绪队列，便完整完成了我们的唤醒机制。





2.1.2 函数设计流程图

(1)`Timer_sleep()`



(2)Timer_interrept()`

(3) `Check_sleep()`

2.1.3 实验相关问题

> (1)数据结构

> A1 在此复制每个新的或更改的`struct`或的声明`struct`成员，全局或静态变量，`typedef`或枚举。用25个字以内的单词来确定每个单词的目的。

在结构体`struct thread`添加了成员：

```
  /*is the thread sleeping*/
  bool isSleep;//线程是否处于睡眠状态，睡眠则为1，反之为0
  /*the time when thread start to sleep*/
  int64_t start_tick;//记录线程开始睡眠的时间
  /*the time thread going to sleep*/
  int64_t sleep_tick;//线程将要睡眠多长时间
```

即：

```
struct thread
 {
  /* Owned by thread.c. */
  tid_t tid;             /* Thread identifier. */
  enum thread_status status;     /* Thread state. */
  char name[16];           /* Name (for debugging purposes). */
  uint8_t *stack;           /* Saved stack pointer. */
  int priority;            /* Priority. */
  struct list_elem allelem;      /* List element for all threads list. */
  /* Shared between thread.c and synch.c. */
  struct list_elem elem;       /* List element. */
\#ifdef USERPROG
  /* Owned by userprog/process.c. */
  uint32_t *pagedir;         /* Page directory. */
\#endif

  /*is the thread sleeping*/
  bool isSleep;//线程是否处于睡眠状态，睡眠则为1，反之为0
  /*the time when thread start to sleep*/
  int64_t start_tick;//记录线程开始睡眠的时间
  /*the time thread going to sleep*/
  int64_t sleep_tick;//线程将要睡眠多长时间
  /* Owned by thread.c. */
  unsigned magic;           /* Detects stack overflow. */
 };
```

> (2)算法

> A2 简要描述`timer_sleep`函数里面发生了什么，包括时间中断处理程序对它的影响

在使用`timer_sleep()`函数时，首先获取当前系统的时间，其次获取当前正在运行的线程的信息，然后将设置 线程结构体内的睡眠状态判断变量为真，表示线程将进入睡眠状态，并设置睡眠起始时间为刚刚获得的当前系统时间，睡眠时长为传给函数的参数`ticks`，最后执行一个原子性操作`thread_block();`，使当前运行的线程进入阻塞状态。

对于`timer interrupt handler`也就是`timer_interrupt ()`函数，增加了一句`thread_foreach(checkSleep,NULL);`，用以遍历所有线程，检查睡眠时间，将结束睡眠时间的线程放进就绪队列里。

> A3 我们可以采取什么措施，使得时间中断造成的开销最小。

在`timer.c`中增加函数`checkSleep(struct thread* t, void* aux)`来检查每个进程中的是否已经到达了唤醒时间。在时间中断函数运行时，遍历查询队列里每个线程的状态，如果线程已到唤醒时间，则唤醒该线程，加入到等待队列中，此时中断函数只需要遍历一次就可以更改所有线程的状态，节约了大量的时间。

```
void checkSleep(struct thread* t, void* aux){
 if(t->status==THREAD_BLOCKED&&t->isSleep){//正在睡眠并且已经阻塞
  if(timer_elapsed(t->start_tick)>=t->sleep_tick){//放入到正在准备的队列
   //list_push_back(&ready_list, &t->allelem);
   t->isSleep=false;
   thread_unblock(t);
  }
 }
}
```

> (3)同步

> A4 当多个线程同时调用`timer_sleep()`时，如何避免竞争条件?

只要是处于一下两端代码之间的操作，均为原子性操作，无法被中断。不论调用函数对线程队列去遍历、插入还是删除，我们的操作内容均放在以下两段代码之间，保证了操作的独立性。

```
asm volatile ("cli" : : : "memory");
...
asm volatile("sti":::"memory");
```

> A5 在调用`timer_sleep()`期间发生计时器中断时，如何避免竞争条件?

调用`timer_sleep()`函数时，我们将主要操作放在了

```
asm volatile ("cli" : : : "memory");
...
asm volatile("sti":::"memory");
```

代码段之间，屏蔽了时钟中断对函数调用的影响，在调用`timer_sleep()`时，时钟是无法被中断的。

> (4)基本原理

> A6 你为什么选择了这个设计？它在哪一方面是优秀的呢?

基于我们的设计，将`timer_sleep()`函数原本的忙等状态修改为设置睡眠状态以及睡眠时间，减少了不必要的在就绪队列和运行队列转换所消耗的时间，更加节省系统资源。

2.2 优先级调度

2.2.1 需求分析

对于`priority_preempt`测试样例，其实就是要求在创建一个线程的时候， 如果线程高于当前线程就先执行创建的线程。

而对于`priority_fifo`测试样例，则要求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。所以两者只需要在`thread_create()`和`thread_set_priority()`函数执行最后运行`thread_yield()`函数的让权，就可以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。

此时， `priority_change`测试样例也可以通过，是因为它的要求本身就是上述两个测试用例的综合，要求创建了一个新线程并要这个线程立刻调用，然后在降低优先级之后不再继续执行它了。

而`priority_sema`测试用例创建了10个优先级不等的线程，并且每个线程调用`sema_down`函数，其他得不到信号量的线程都得阻塞，而最后每次运行的线程释放信号量时必须确保优先级最高的线程继续执行。

而测试样例`priority_condvar`与之类似，只不过信号量变成了条件变量，本质上并没有太大变化。

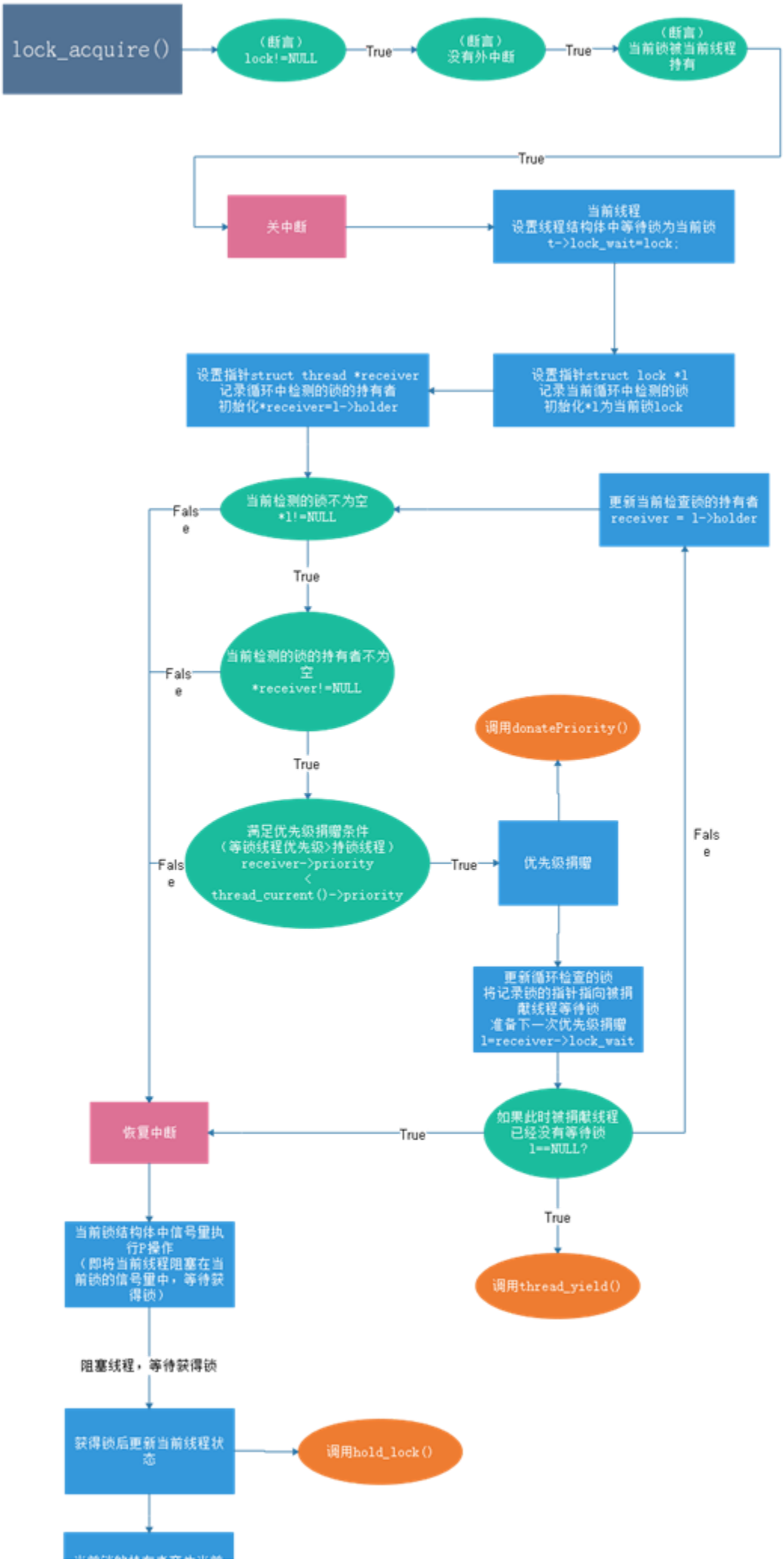
剩余七个测试样例，则是有关优先级捐献的内容，难度由浅入深，逐步提升，给出了一个关于捐赠方式的整体框架。

综上所述，可以得出以下测试整合的逻辑——

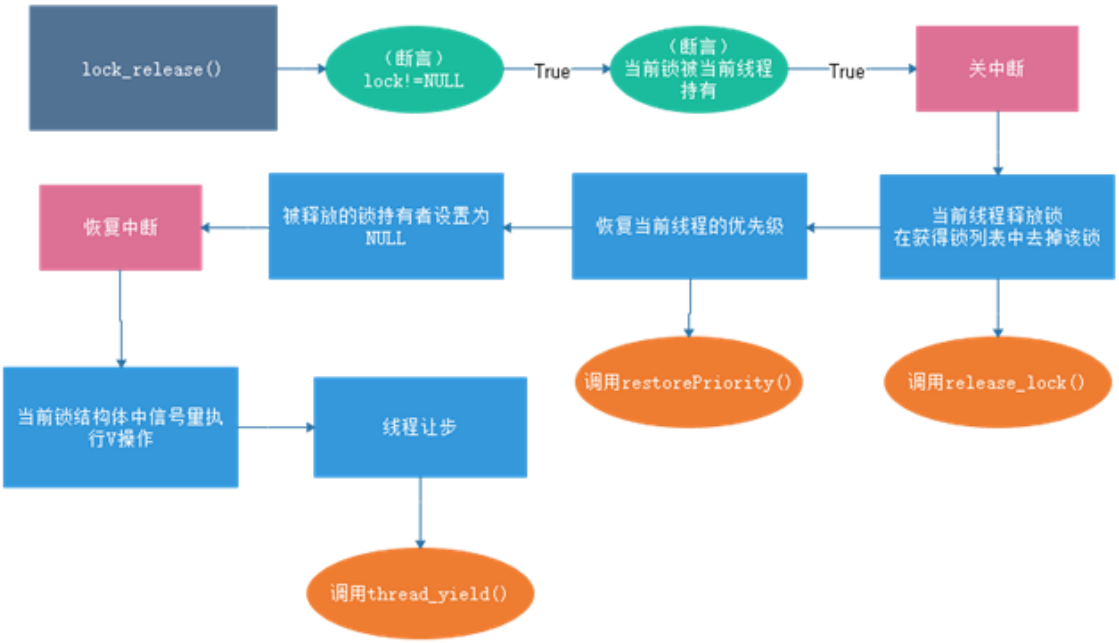
> 1. 在一个线程获取一个锁的时候， 如果拥有这个锁的线程优先级比自己低就提高它的优先级，并且如果这个锁还被别的锁锁着， 将会递归地捐赠优先级， 然后在这个线程释放掉这个锁之后恢复未捐赠逻辑下的优先级。
> 2. 如果一个线程被多个线程捐赠， 维持当前优先级为捐赠优先级中的最大值（`acquire`和`release`之时）。
> 3. 在对一个线程进行优先级设置的时候， 如果这个线程处于被捐赠状态， 则对线程本身的优先级进行设置， 然后如果设置的优先级大于当前优先级， 则改变当前优先级， 否则在捐赠状态取消的时候恢复本身的优先级。
> 4. 在释放锁对一个锁优先级有改变的时候应考虑其余被捐赠优先级和当前优先级。
> 5. 将`semaphore`的`waiters`队列实现为优先级队列。
> 6. 将`condition`的`waiters`队列实现为优先级队列。
> 7. 释放锁的时候若优先级改变则可以发生抢占。

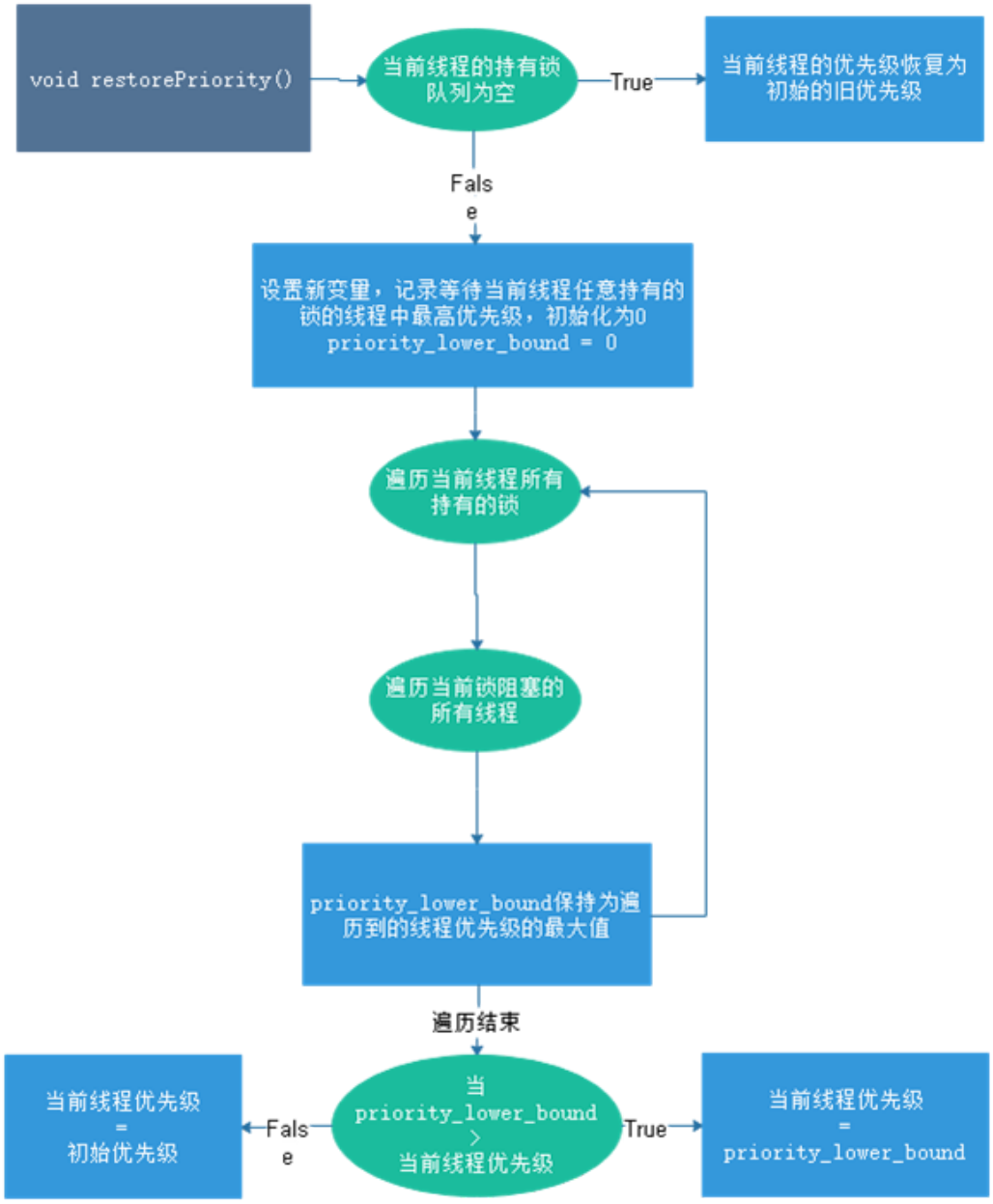
2.2.2 函数设计流程图

(1)`lock_acquire()`

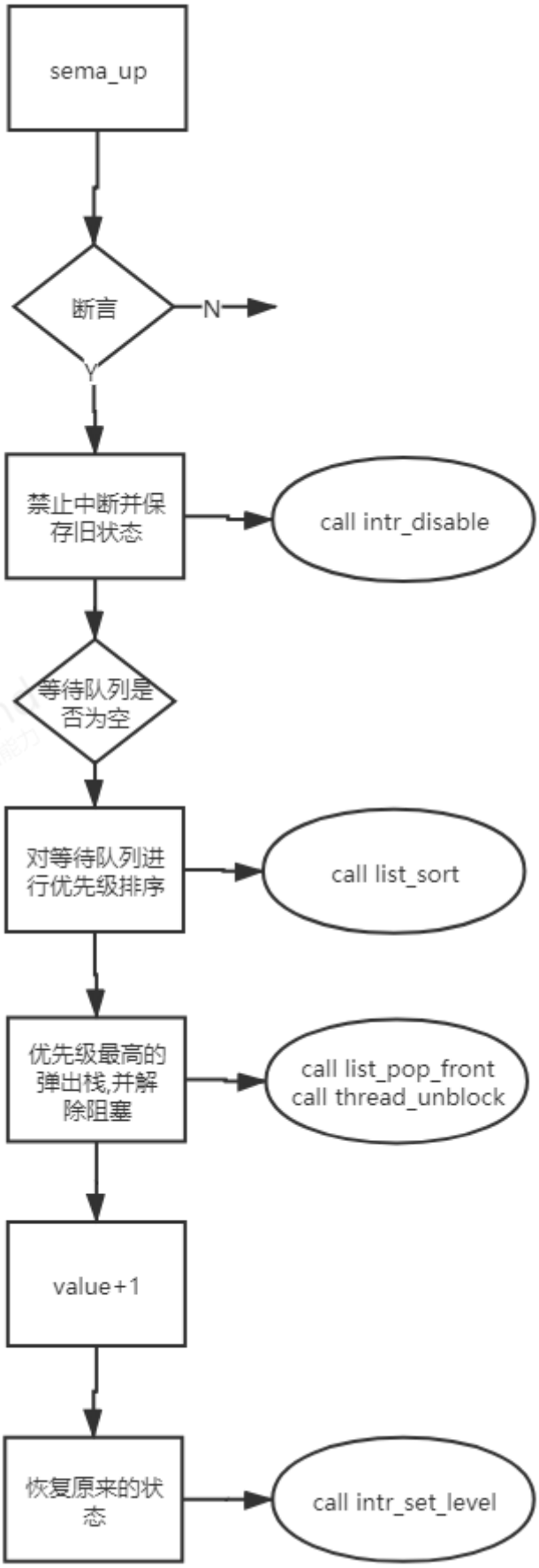

lock_acquire()
（断言）
lock!=NULL
True
（断言）
没有外中断
True
（断言）
当前锁被当前线程
持有
True
关中断
当前线程
设置线程结构体中等待锁为当前锁
t->lock_wait=lock;
设置指针struct lock *l
记录当前循环中检测的锁
初始化*l为当前锁lock
设置指针struct thread *receiver
记录循环中检测的锁的持有者
初始化*receiver=l->holder
更新当前检查锁的持有者
receiver = l->holder
当前检测的锁不为空
*l!=NULL
Fals
e
True
当前检测的锁的持有者不为
空
*receiver!=NULL
Fals
e
True
满足优先级捐赠条件
（等锁线程优先级>持锁线程）
receiver->priority
<
thread_current()->priority
Fals
e
True
优先级捐赠
调用donatePriority()
更新循环检查的锁
将记录锁的指针指向被捐
献线程等待锁
准备下一次优先级捐赠
l=receiver->lock_wait
如果此时被捐献线程
已经没有等待锁
l==NULL?
True
Fals
e
恢复中断
True
当前锁结构体中信号量执
行P操作
（即将当前线程阻塞在当
前锁的信号量中，等待获
得锁）
True
调用thread_yield()
阻塞线程，等待获得锁
获得锁后更新当前线程状
态
调用hold_lock()

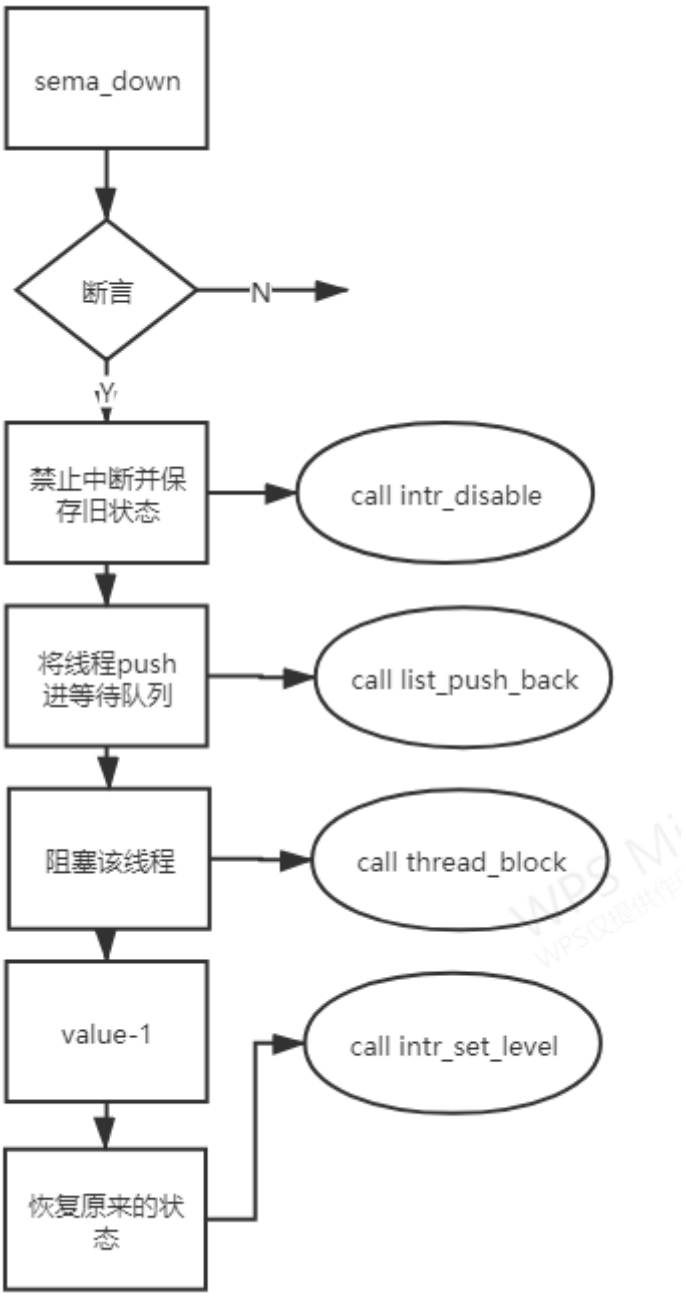
(2) `lock_release()`



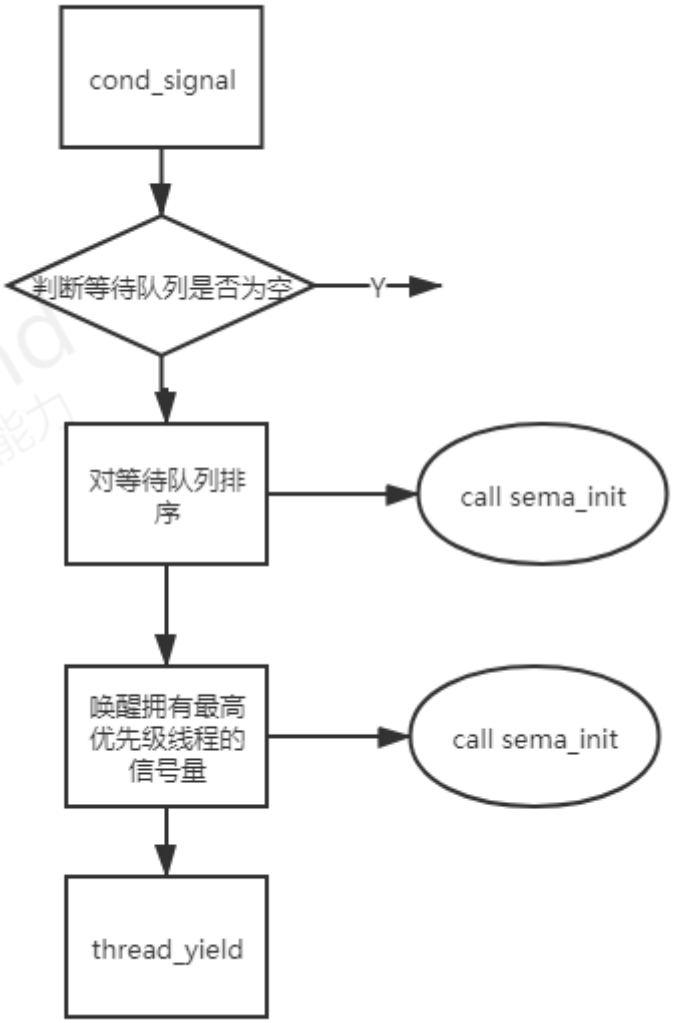


(3) `sema_up` 与 `sema-down`


sema_up
断言
N
Y
禁止中断并保存旧状态
call intr_disable
等待队列是否为空
对等待队列进行优先级排序
call list_sort
优先级最高的弹出栈,并解除阻塞
call list_pop_front
call thread_unblock
value+1
恢复原来的状态
call intr_set_level



(4) `cond_signal()`



2.2.3 实验相关问题

> (1)数据结构

> B1 在此复制每个新的或更改的struct或的声明struct成员，全局或静态变量，“ typedef”或枚举。用25个字以内的单词来确定每个单词的目的。

- thread结构体

```
/*the lock the thread is waiting for*/
struct lock* lock_wait;
/*the list of locks that the thread is holding*/
struct list locks;
/*the original priority*/
int base_priority;
```

- lock结构体

```
struct list_elem elem;
```

> B2 解释用于跟踪优先级捐赠的数据结构。 使用ASCII艺术作品来绘制嵌套捐赠。

```
|Thread 1|low| //首先我们有一个低优先级的线程拥有一个锁lock1
|Thread 2|mid|->|Thread 1|low|//然后一种中优先级的线程被lock1阻塞
|Thread 2|mid|->|Thread 1|mid|//那么thread2就将优先级donate给thread1
|Thread 3|hig|->|Thread 2|mid|->|Thread 1|mid|//现在一个更高优先级的线程想要lock
|Thread 3|hig|->|Thread 2|hig|->|Thread 1|mid|//于是donate优先级
|Thread 3|hig|->|Thread 2|hig|->|Thread 1|hig|//新的高优先级的线程会继续donate给第一个线程
```

> (2)算法

> B3 如何确保等待锁，信号灯或条件变量的优先级最高的线程首先唤醒?

每当一个锁、信号量被释放时，我们就会对所有等待的线程进行检查，并唤醒所有线程中优先级最高的线程。然后将这个线程从等待列表中删除。对于条件变量，我们一样是寻找有最高优先级线程的信号量。

> B4 描述对lock_acquire()的调用导致优先级捐赠时的事件顺序。嵌套捐赠如何处理?

如下处理:

```
/*update the lock we're waiting for*/
  thread_current()->lock_wait = lock;
  struct lock *l = lock;
  struct thread *receiver = l->holder;
  while(l!=NULL&&receiver!=NULL&&receiver->priority<thread_current()->priority){
   donatePriority(receiver, thread_current()->priority);
   l = receiver->lock_wait;
   if(l==NULL){// just sort the ready_list because the donated wasn't waiting for a lock
    thread_yield();
   }
   receiver = l->holder;
  }
```

当当前调用的线程需要的锁有另一个线程持有，且这个线程是低优先级线程，当前线程马上进行优先级捐赠。当被捐赠的线程的优先级仍然高于其等待锁的持有者的优先级，那么就将继续优先级捐赠，直到递归道德线程不等待任何锁，或者比等待锁的持有者的优先级高。

B5 描述在较高优先级线程正在等待的锁上调用lock_release()时的事件顺序。

```
enum intr_level old_level = intr_disable();
 release_lock(lock);
 //TODO: restore the priority before donation
 if(!thread_mlfqs)restorePriority();
 lock->holder = NULL;
 intr_set_level(old_level);
/*remove the lock from acquired locks*/
void release_lock(struct lock *l){
 list_remove(&l->elem);
}

```

```
/*return to the priority whose waiter has the highest priority if it is bigger than base_priority*/
void restorePriority(){
 if(list_empty(&thread_current()->locks)){
  //printf("No locks at hand!\n");
  thread_current()->priority = thread_current()->base_priority;
 }
 else{
  int priority_lower_bound = 0;

  /*run through all the locks holden by the current thread*/
  for(struct list_elem *temp_lock_elem = list_begin(&thread_current()->locks);temp_lock_elem!=list_end(&thread_current()->locks); temp_lock_elem = list_next(temp_lock_elem)){

   struct lock* cur_lock = list_entry(temp_lock_elem, struct lock, elem);
   /*find the waiter with highest priority*/

   for(struct list_elem *temp_thread_elem = list_begin(&cur_lock->semaphore.waiters);temp_thread_elem!= list_end(&cur_lock->semaphore.waiters); temp_thread_elem = list_next(temp_thread_elem)){
   struct thread* cur_thread = list_entry(temp_thread_elem, struct thread, elem);

   priority_lower_bound = priority_lower_bound>cur_thread->priority? priority_lower_bound:cur_thread->priority;
   }
  }
  if(priority_lower_bound>thread_current()->base_priority){
   thread_current()->priority = priority_lower_bound;
  }
  else{
   thread_current()->priority = thread_current()->base_priority;
  }
 }
}
```

首先，我们要明确，释放锁的线程并不一定返回原来的优先级，当他的持有的其他锁（不包括释放的锁）的中存在某个等待锁的线程的优先级，比释放锁的该线程的优先级高时，我们的这个线程的优先级就需要恢复到，它所持有的锁中（不包括释放的锁），等待这些锁的线程的最高优先级。这样，我们就可以避免多次的优先级捐赠。

> (3)同步

> B6 在thread_set_priority()中描述潜在的竞争，并说明您的实现如何避免这种情况。您可以使用锁来避免这场竞争吗?

```
/* Sets the current thread's priority to NEW_PRIORITY. */
void
thread_set_priority (int new_priority)
{
 if(list_empty(&thread_current()->locks)){
  thread_current()->priority = new_priority;
 }
 else{
```

```
  if(new_priority>thread_current()){
   thread_current()->priority = new_priority;
  }
 }
 thread_current()->base_priority = new_priority;
 if(list_entry(list_max(&ready_list, comparePriorityElem,NULL),struct
thread, elem)->priority>new_priority){
  thread_yield();
 }
}
```

当新优先级被赋予，线程中存储的原优先级，立即改变为新的优先级。而正在使用的优先级并不一定要改变。

当这个新优先级大于当前优先级时，改变当前优先级。

当这个线程持有的锁没有阻塞其他线程，改变。

当这个线程持有的锁阻塞其他线程且新优先级小于当前优先级，那么当前优先级变为MAX（新优先级，被阻塞的线程的优先级的最大值）

> (4)基本原理
>
> B7 你为什么选择了这个设计？它在哪一方面是优秀的呢？

这种设计很稳定。相比其他的设计，我们的设计确保了捐赠立即发生，不会对之后的事件产生影响。

2.3 高级调度程序

2.3.1 需求分析

这里我们需要实现一个多级优先反馈队列。

每个线程直接在其控制下的值都在-20到20之间。 每个线程还有一个优先级，介于`0（PRI_MIN）`到`63（PRI_MAX）`之间，每四个刻度使用以下公式重新计算一次：

`priority = PRI_MAX - (recent_cpu / 4) - (nice * 2)`

`Recent_cpu`度量线程“最近”收到的CPU时间。 在每个计时器滴答中，正在运行的线程的最近使用的cpu会增加1。每秒一次，每个线程的最近使用的cpu都将以这种方式更新：

`recent_cpu = (2*load_avg)/(2*load_avg + 1) * recent_cpu + nice`

`load_avg`估计过去一分钟准备运行的平均线程数。 它在启动时初始化为0，并每秒重新计算一次，如下所示：

`load_avg = (59/60)*load_avg + (1/60)*ready_threads`

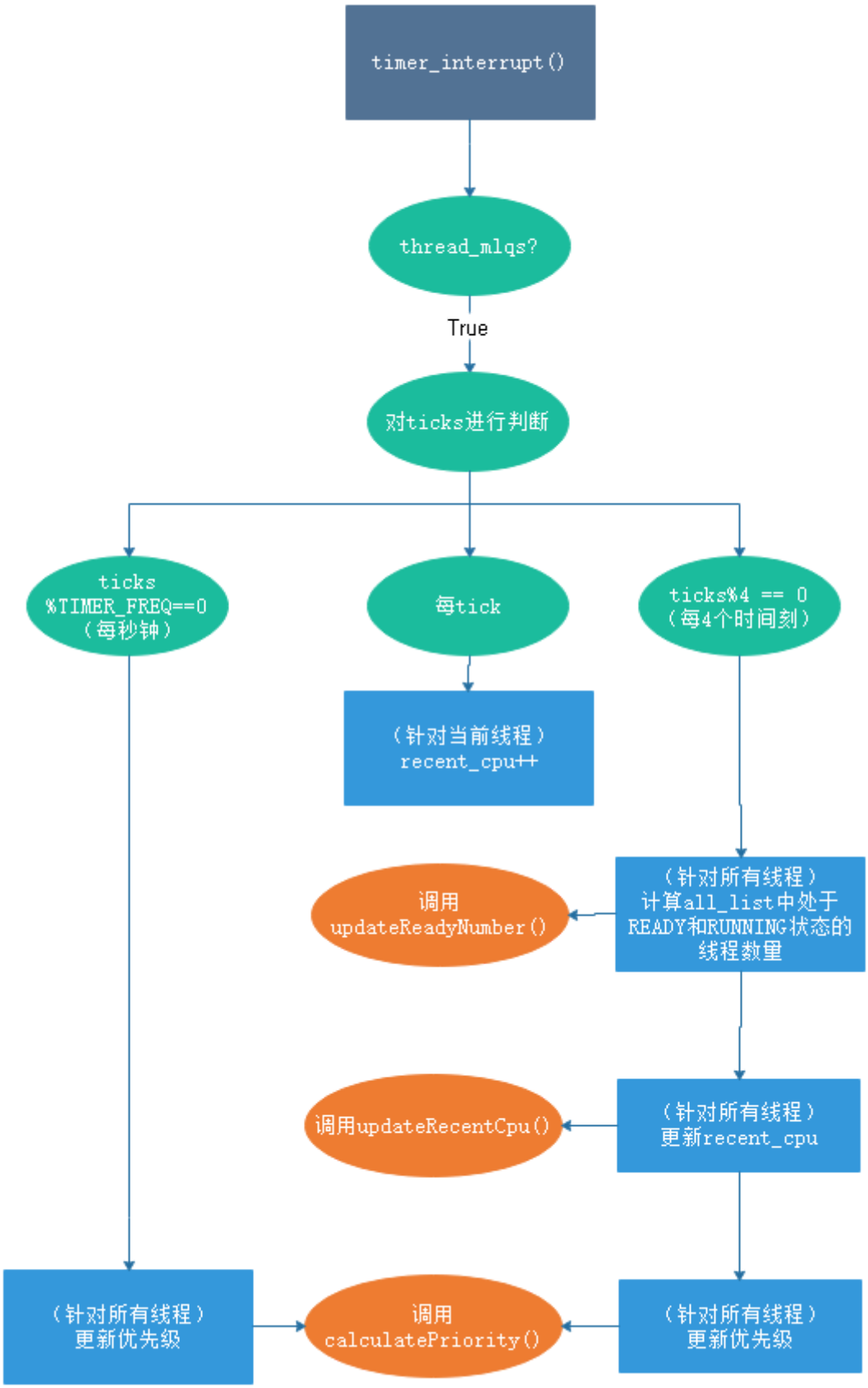
其中`ready_threads`是在更新时正在运行或准备运行的线程数（不包括空闲线程）。

在`timer_interrupt`中固定一段时间计算更新线程的优先级，这里是每`TIMER_FREQ`时间更新一次系统load_avg和所有线程的`recent_cpu`， 每4个`timer_ticks`更新一次线程优先级， 每个`timer_tick running`线程的`recent_cpu`加一， 虽然这里说的是维持64个优先级队列调度， 其本质还是优先级调度， 我们保留之前写的优先级调度代码即可， 去掉优先级捐赠（之前`donate`相关代码已经对需要的地方加了`thread_mlfqs`的判断了）。

另外，我们需要自己设计一套浮点数计算的函数。

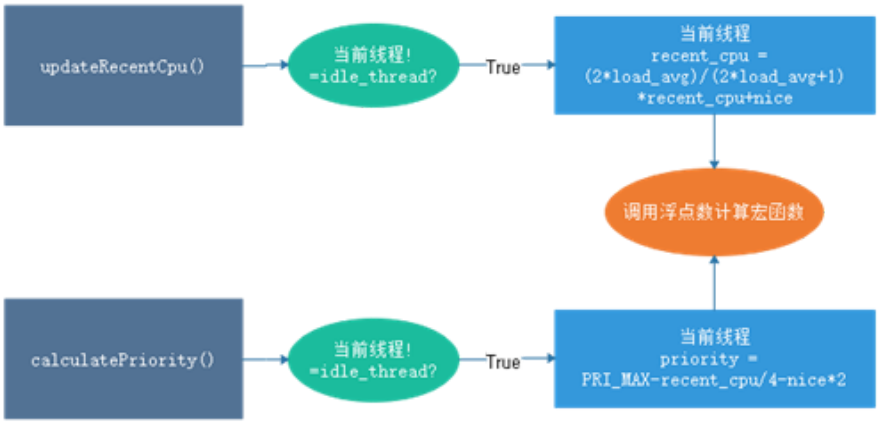
2.3.2 函数设计流程图

(1) `Timer_interrupt`

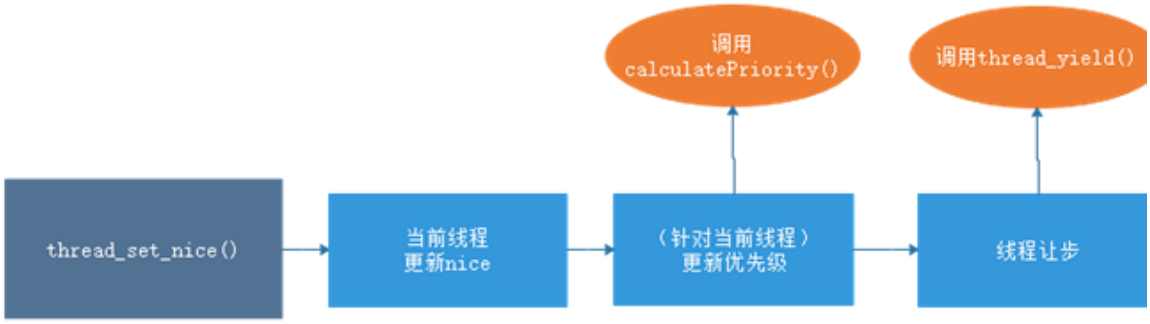


(2) `updateReadyNumber`

(3) `updateRecentCpu & calculatePriority`



(4) `Thread_set_nice`



2.3.3 实验相关问题

> (1)数据结构

> C1：在此复制每个新的或更改的struct或的声明struct成员，全局或静态变量，“ typedef”或枚举。用25个字以内的单词来确定每个单词的目的。

- 浮点数计算

```
typedef int myfloat;
/*偏移量*/
#define shift 16
/*把整数转换为定点数*/
#define convert_float(n) ((myfloat)(n<<shift))
/*把定点数转换为整数*/
#define convert_integer(x) (x>0?(int)((x+(1<<shift)/2)>>shift):(int)((x-(1<<shift)/2)>>shift))
/*定点数相乘*/
#define mult(x,y) (((int64_t)x*(int64_t)y)>>shift)
/*定点数相除*/
#define div(x,y) ((int64_t)x<<shift)/y
```

- thread结构体

```
/*the niceness of a thread*/
int nice;
/*recent cpu of a thread*/
myfloat recent_cpu;
```

- 全局变量

```
myfloat load_avg;
```

> (2)算法

> C2：假设线程A，B和C的nice值分别为0、1和2。每个的recent_cpu值均为0。填写下表，显示调度决策以及每个进程的priority和recent_cpu值

Timer	**Recent_cpu**	**Priority**	`Thread`				
Ticks	A	B	C	A	B	C	To run
0	0	0	0	63	61	59	A
4	4	0	0	62	61	59	A
8	8	0	0	61	61	59	A
12	12	0	0	60	61	59	B
16	12	4	0	60	60	59	B
20	12	8	0	60	59	59	A
24	16	8	0	59	59	59	A
28	20	8	0	58	59	59	B
32	20	12	0	58	58	59	C
36	20	12	4	58	58	58	A

> C3：那些在任何模糊调度程序规范使表中的值不确定？如果是这样，您使用什么规则来解决这些问题？这是否与您的调度程序的行为相符？

同优先级线程运行顺序问题。当出现一下两种情况：当前线程优先级和等待队列中的线程优先级相同时、等待队列中最高优先级线程有多个时，我们应该让那个线程先运行？

实际上我们并不能确定他们的运行关系，因为在给线程优先级排序的时候，我们使用了系统自带的sort函数，而sort函数是不稳定的，我们并不能确定同优先级线程在快排的时候，是否被改变了顺序。

> C4：您如何分摊那些可能会影响性能的，处于中断内部与中断外部的上下文切换的时间花费？

为了避免“关中断”操作中占用的时间过长，影响系统时间中断的正常运行，我们尽量将代码写在关中断之外，只有那些需要被原子地运行的操作，才会写入到关中断代码块中，以减少关中断时间过长对时钟中断的影响。

2.3.3 基本原理

> C5：简要评析您的设计，指出您选择的设计中的优点和缺点。如果您有更多的时间来处理项目的这一部分，您将如何选择改进或改进您的设计？

设计：

（1）、题目中要求：每一个tick，增加当前线程的recent_cpu；每4个ticks，更新所有线程的优先级；每秒钟（100个ticks）更新所有线程的优先级、recent_cpu，并且更新load_avg的值。

因为这个要求与时间中断密切相关，所以我们将更新优先级、recent_cpu、load_avg的函数写在timer_interrupt函数里面，每一个时间中断tick就进行一次操作，进行操作的时候屏蔽中断，达到原子操作的效果。

（2）、我们在调取等待队列的大小值时，没有用list_size函数（因为出现了一些问题）我们利用thread_foreach函数使每个运行中和等待中的线程修改同一个全局变量，达到计数的目的。

改进：在维护各个优先队列的时候，我们发现pintos自带的关于链表的操作函数有一定的缺陷，而且还可能被我们之前修改的代码影响（比如本组在做该部分实验时，list_size函数出现了一些问题）如果再有一些时间，我们可以对pintos自带的list函数进行修改。

> C6：作业详细说明了定点数学的算术，但它使您可以实施它。 您为什么决定以自己的方式实施它？ 如果您为定点数学创建了一个抽象层，即一个抽象数据类型和/或一组函数或宏来操纵定点数，为什么要这么做？ 如果没有，为什么不呢？

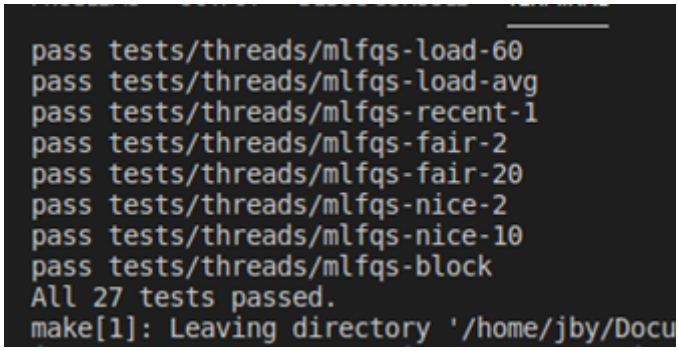
实现浮点数的添加代码如下：

```
typedef int myfloat;
/*偏移量*/
\#define shift 16
/*把整数转换为定点数*/
\#define convert_float(n) ((myfloat)(n<<shift))
/*把定点数转换为整数*/
\#define convert_integer(x) (x>0?(int)((x+(1<<shift)/2)>>shift):(int)((x-(1<<shift)/2)>>shift))
/*定点数相乘*/
\#define mult(x,y) (((int64_t)x*(int64_t)y)>>shift)
/*定点数相除*/
\#define div(x,y) ((int64_t)x<<shift)/y
```

由于在内核中直接实现浮点数的相加需要大量的开销，对整个内核的性能造成了巨大的影响，所以我们还是用整型变量的计算来模拟浮点数的计算。

我们将整型变量向左偏移16个二进制位，来模拟浮点数，留出的16位二进制位就是表示浮点数小数的位置。

2.3.4 实验结果


pass tests/threads/mlfqs-load-60
pass tests/threads/mlfqs-load-avg
pass tests/threads/mlfqs-recent-1
pass tests/threads/mlfqs-fair-2
pass tests/threads/mlfqs-fair-20
pass tests/threads/mlfqs-nice-2
pass tests/threads/mlfqs-nice-10
pass tests/threads/mlfqs-block
All 27 tests passed.
make[1]: Leaving directory '/home/jby/Docu