+——————————————+
| CS 140 |
| PROJECT 1: THREADS |
| DESIGN DOCUMENT |
+——————————————+

---- GROUP ----

何敖南（10132510231@ecnu.cn）
李梦芸（10132510105@ecnu.cn）

---- PRELIMINARIES ----

>>NONE

ALARM CLOCK

==========

---- DATA STRUCTURES ——

>> owned by thread.h
>> int64_t ticks_blocked; // Record the time the thread has been blocked.
>> void blocked_thread_check (struct thread *t, void *aux UNUSED); // Check the blocked thread.

>> owned by timer.c
>> enum intr_level old_level = intr_disable ();
>> struct thread *current_thread = thread_current ();

---- ALGORITHMS ——

>> timer_sleep是在ticks时间内，当线程处于running状态时，不断地把它传至就绪队列，不让其执行。
>> 线程依然不断在CPU就绪队列和running队列之间来回，占用了CPU资源，因此，我们希望用一种唤醒机制来实现timer_sleep函数。
>> 实现方法：
>> 调用timer_sleep的时候，将线程阻塞掉，然后给线程结构体增加一个成员变量ticks_blocked，用来记录这个线程被sleep了多长时间。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自减1，当减至0时就唤醒这个线程。

---- SYNCHRONIZATION ——

>> 显然，当多个线程同时调用timer_sleep()时，它们互相不受影响。
>> 因为，每个线程都有属于自己的ticks_blocked，记录着自己的sleep时间，不会对其他的线程造成影响。

>> intr_level代表能否被中断，而intr_diable调用intr_get_level()，并且直接执行汇编代码，调用汇编指令来保证当前线程不会被中断。
>> 修改timer_interrupt函数，加入线程sleep时间的检测，thread_foreach (blocked_thread_check, NULL)。

---- RATIONALE ——

>> 我们并希望线程在CPU就绪队列和running队列之间来回，这占用了不必要的CPU资源，因此我们使用一种唤醒机制来避免这种情况。

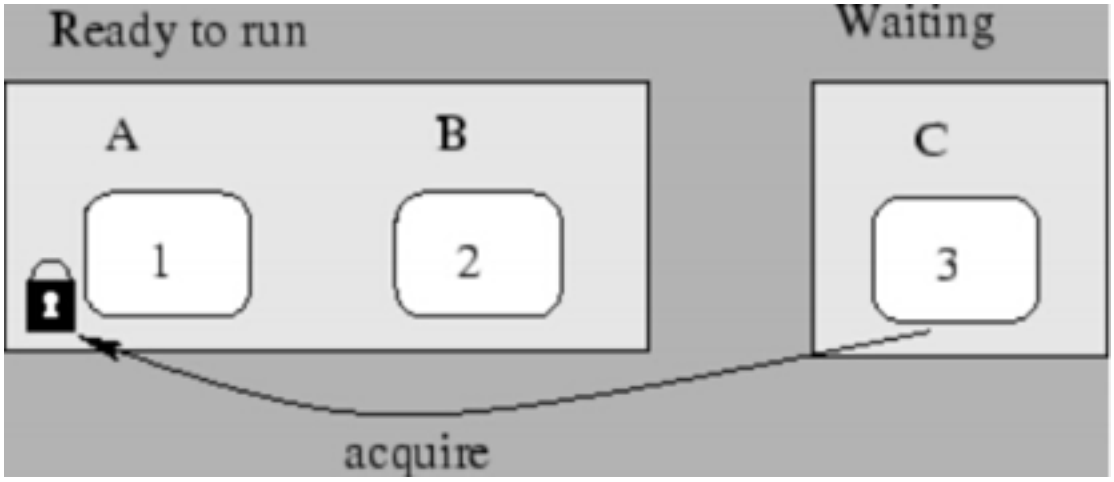
PRIORITY SCHEDULING
==========

---- DATA STRUCTURES ——

```
>> owned by thread.h
>> int base_priority;            // Base priority.
>> struct list locks;            // Locks that the thread is holding.
>> struct lock *lock_waiting;            // The lock that the thread is waiting for.
>> void thread_hold_the_lock (struct lock *);   // Let thread hold a lock
>> void thread_remove_lock (struct lock *);     // Remove a lock
>> void thread_donate_priority (struct thread *);  // Donate current priority to thread t.
>> void thread_update_priority (struct thread *);            // Update priority

>> owned by thread.c
>> enum intr_level old_level = intr_disable ();
>> struct thread *current_thread = thread_current ();
>> int old_priority = current_thread->priority;

>> owned by synch.h
>> struct list_elem elem;      // List element for priority donation.
>> int max_priority;           // Max priority among the threads acquiring the lock.
>> bool lock_cmp_priority (const struct list_elem *a, const struct list_elem *b, void
*aux);
>> bool cond_sema_cmp_priority (const struct list_elem *a, const struct list_elem *b,
void *aux);

>> owned by synch.c
>> struct thread *current_thread = thread_current ();
>> struct lock *l;
>> enum intr_level old_level;
```

>> 线程A、B、C分别具有1、2、3优先级（数字越大表示优先级越高）
>> 线程A、B目前在就绪队列中等待调度，线程A对一个互斥资源拥有线程锁

>> 此时，高优先级的线程C也想要访问这个互斥资源，线程C只好在这个资源上等待，不能进入就绪队列
>> 当调度器开始调度时，只能从A和B中进行选择，根据优先级的调度原理，线程B将会先运行
>> 这时，本来线程C优先级比线程B高，但是线程B却先运行了，从而产生了优先级翻转问题



---- ALGORITHMS ——

>> 当发现高优先级的任务因为低优先级任务占用资源而阻塞时，就将低优先级任务的优先级提升到等待它所占有的资源的最高优先级任务的优先级。
>> 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，若没有其余捐赠者， 则恢复原始优先级。
>> 优先级嵌套问题， 重点在于medium拥有的锁被low阻塞， 在这个前提下high再去获取medium的说阻塞的话， 优先级提升具有连环效应， 就是medium被提升了， 此时它被锁捆绑的low线程应该跟着一起提升。
>> 实现方法：
>> 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。
>> 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）。
>> 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original_priority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original_priority。
>> 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。

>> 将信号量的等待队列实现为优先级队列。
>> 将condition的waiters队列实现为优先级队列。
>> 释放锁的时候若优先级改变则可以发生抢占。

---- SYNCHRONIZATION ——

>> 这部分不使用同步

---- RATIONALE ——

>> 线程需要一个数据结构来记录所有对这个线程有捐赠行为的线程，以及获取这个线程被锁于哪个线程。
>> 因此，我们给struct thread加入数据结构int base_priority、struct list locks、struct lock *lock_waiting，给lock加入数据结构struct list_elem elem、int max_priority。

ADVANCED SCHEDULER

---- DATA STRUCTURES ----

>> owned by thread.h
>> int nice; // Niceness.
>> fixed_t recent_cpu; // Recent CPU.
>> void thread_mlfqs_increase_recent_cpu_by_one (void);
>> void thread_mlfqs_update_priority (struct thread *);
>> void thread_mlfqs_update_load_avg_and_recent_cpu (void);

>> owned by fixed_point.h
/* Basic definitions of fixed point. */
>> typedef int fixed_t;
/* 16 LSB used for fractional part. */
>> #define FP_SHIFT_AMOUNT 16
/* Convert a value to fixed-point value. */
>> #define FP_CONST(A) ((fixed_t)(A << FP_SHIFT_AMOUNT))
/* Add two fixed-point value. */
>> #define FP_ADD(A,B) (A + B)
/* Add a fixed-point value A and an int value B. */
>> #define FP_ADD_MIX(A,B) (A + (B << FP_SHIFT_AMOUNT))
/* Substract two fixed-point value. */
>> #define FP_SUB(A,B) (A - B)
/* Substract an int value B from a fixed-point value A */
>> #define FP_SUB_MIX(A,B) (A - (B << FP_SHIFT_AMOUNT))
/* Multiply a fixed-point value A by an int value B. */
>> #define FP_MULT_MIX(A,B) (A * B)
/* Divide a fixed-point value A by an int value B. */
>> #define FP_DIV_MIX(A,B) (A / B)

/* Multiply two fixed-point value. */
>> #define FP_MULT(A,B) ((fixed_t)(((int64_t) A) * B >> FP_SHIFT_AMOUNT))
/* Divide two fixed-point value. */
>> #define FP_DIV(A,B) ((fixed_t)((((int64_t) A) << FP_SHIFT_AMOUNT) / B))
/* Get integer part of a fixed-point value. */
>> #define FP_INT_PART(A) (A >> FP_SHIFT_AMOUNT)
/* Get rounded integer of a fixed-point value. */
>> #define FP_ROUND(A) (A >= 0 ? ((A + (1 << (FP_SHIFT_AMOUNT - 1))) >> FP_SHIFT_AMOUNT) \
: ((A - (1 << (FP_SHIFT_AMOUNT - 1))) >> FP_SHIFT_AMOUNT))

---- ALGORITHMS ——

>> 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1。
>>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）。

timer ticks	recent_cpu A	B	C	priority A	B	C	thread to run	ready list
0	0	0	0	63	61	59	A	B, C
4	4	0	0	62	61	59	A	B, C
8	8	0	0	61	61	59	B	A, C
12	8	4	0	61	60	59	A	B, C
16	12	4	0	60	60	59	B	A, C
20	12	8	0	60	59	59	A	C, B
24	16	8	0	59	59	59	C	B, A
28	16	8	4	59	59	58	B	A, C
32	16	12	4	59	58	58	A	C, B
36	20	12	4	58	58	58	C	B, A

---- RATIONALE ——

>> 我们参考了pintos.pdf
>> 包括附录B 4.4BSD调度器和2.2.4高级调度等
>> 在网上搜索了关于pintos的配置教程和相关的算法思想