Lab1 Design Doc

Project 1: Threads

Preliminaries

> If you have any preliminary comments on your submission, notes for the
> TAs, please give them here.
> Please cite any offline or online sources you consulted while
> preparing your submission, other than the Pintos documentation, course
> text, lecture notes, and course staff.

Alarm Clock

DATA STRUCTURES

> A1: Copy here the declaration of each new or changed struct or struct member, global or static variable, typedef, or enumeration. Identify the purpose of each in 25 words or less.

```
struct thread
  {
    /* Owned by thread.c. */
    tid_t tid;
    enum thread_status status;
    char name[16];
    uint8_t *stack;
    int priority;

    int sleep_ticks;
```

1. `int sleep_ticks` :A new member added to struct thread,which records the number of timer ticks since this thread fell into asleep.

2. `static struct list sleeping_thread_list` :The list of threads that are sleeping.

ALGORITHMS

> A2: Briefly describe what happens in a call to timer_sleep(),
> including the effects of the timer interrupt handler.

```c
void
timer_sleep (int64_t ticks)
{
  if(ticks<0)return;

  enum intr_level old_level;
  old_level = intr_disable ();

  struct thread*t=thread_current();
  t->sleep_ticks=ticks;
  list_insert_ordered(&sleeping_thread_list,&t->elem,thread_compare_sleepticks,NULL);
  thread_block();

  intr_set_level (old_level);
}
```

```c
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  struct list_elem*e=list_begin(&sleeping_thread_list);
  while(e!=list_end(&sleeping_thread_list))
  {
    struct thread*t=list_entry(e,struct thread,elem);
    --(t->sleep_ticks);
    if(t->sleep_ticks<=0)
    {
      e=list_remove(e);
      thread_unblock(t);
      intr_yield_on_return();
    }
    else
    {
      e=list_next(e);
      break;
    }
  }
  while(e!=list_end(&sleeping_thread_list)){
    struct thread*t=list_entry(e,struct thread,elem);
    --(t->sleep_ticks);
    e=list_next(e);
  }
  thread_tick ();
}
```

1.When `ticks<0` ,the thread who calls `timer_sleep` doesn't need to sleep,this function returns immediately.

2. `ticks>=0`

It can be confusing when `tick=0` .It should be clarified that we can't just ignore `timer_sleep(0)` ,because it should lead to a transfer of control.So we combine these two cases(ticks=0,ticks>0).

We first set current thread's sleep_ticks equaling to ticks,then insert it to `sleeping_thread_list` ,then block the caller thread and transfer control to another ready thread.

As for ending sleep,that's the timer interrupt handler's work.At each timer tick,we traverse the `sleepint_thread_list` and reduce the `sleep_ticks` of each thread in this list by one.If some thread's `sleep_ticks` is less than 0,it's time to unblock the thread and remove it from `sleeping_thread_list` .

> A3: What steps are taken to minimize the amount of time spent in the timer interrupt handler?

An ordered list which is sorted and inserted by sleep_ticks number is used, so that we can check the list from the beginning and stop whenever the sleep_ticks is larger zero, which guarantees the later threads in the sleep list don’t need to be checked. By this means, we can minimize the time spent.

SYNCHRONIZATION

> A4: How are race conditions avoided when multiple threads call timer_sleep() simultaneously?

List operations happen during interrupt is disabled.

> A5: How are race conditions avoided when a timer interrupt occurs during a call to timer_sleep()?

Turn off the interrupts.

RATIONALE

> A6: Why did you choose this design? In what ways is it superior to

> another design you considered?

To achieve sleeping without busy waiting,an obvious and direct idea is to block this thread for `ticks` timer tick instead of 'keeping asking'.And these sleeping thread will be awakened by the only manager.Naturally we choose `timer_interrupt` to act as **the manager**.

To have a sleep list to store the sleeping threads are the straight forward thought, so no other designs are actually been considered.The choice of a sorted list and to insert by order is made to be more efficient.I think this design is safe and reasonable too.

Priority Scheduling

DATA STRUCTURES

> B1: Copy here the declaration of each new or changed struct or struct member, global or static variable, typedef, or enumeration. Identify the purpose of each in 25 words or less.

1.add three new menbers to struct thread

```
int initial_priority;                /**Initial prority*/
struct list lock_list;               /**Locks the thread holds*/
struct lock*wait_lock;               /*the lock this thread waiting to hold*/
```

`int initial_priority` :records the original priority of the thread before any priority donation happens.

`struct list lock_list` :A list of the locks this thread already holds.

`struct lock*wait_lock` :If there is a lock this thread ask to hold,but it is held by another thread,it records the lock.Otherwise,it is `NULL` .

```
struct lock
  {
    struct list_elem elem;
    int priority;
    struct thread *holder;      /**< Thread holding lock (for debugging). */
    struct semaphore semaphore; /**< Binary semaphore controlling access. */
  };
```

`struct list_elem elem` :make `lock` linkable as element of list.

`int priority` :It records the maximun priority of the threads that are waiting to acquire this lock.

> B2: Explain the data structure used to track priority donation.
> Use ASCII art to diagram a nested donation. (Alternately, submit a
> .png file.)

First,there are two locks,**Lock A and Lock B**.Both of locks are vacant.

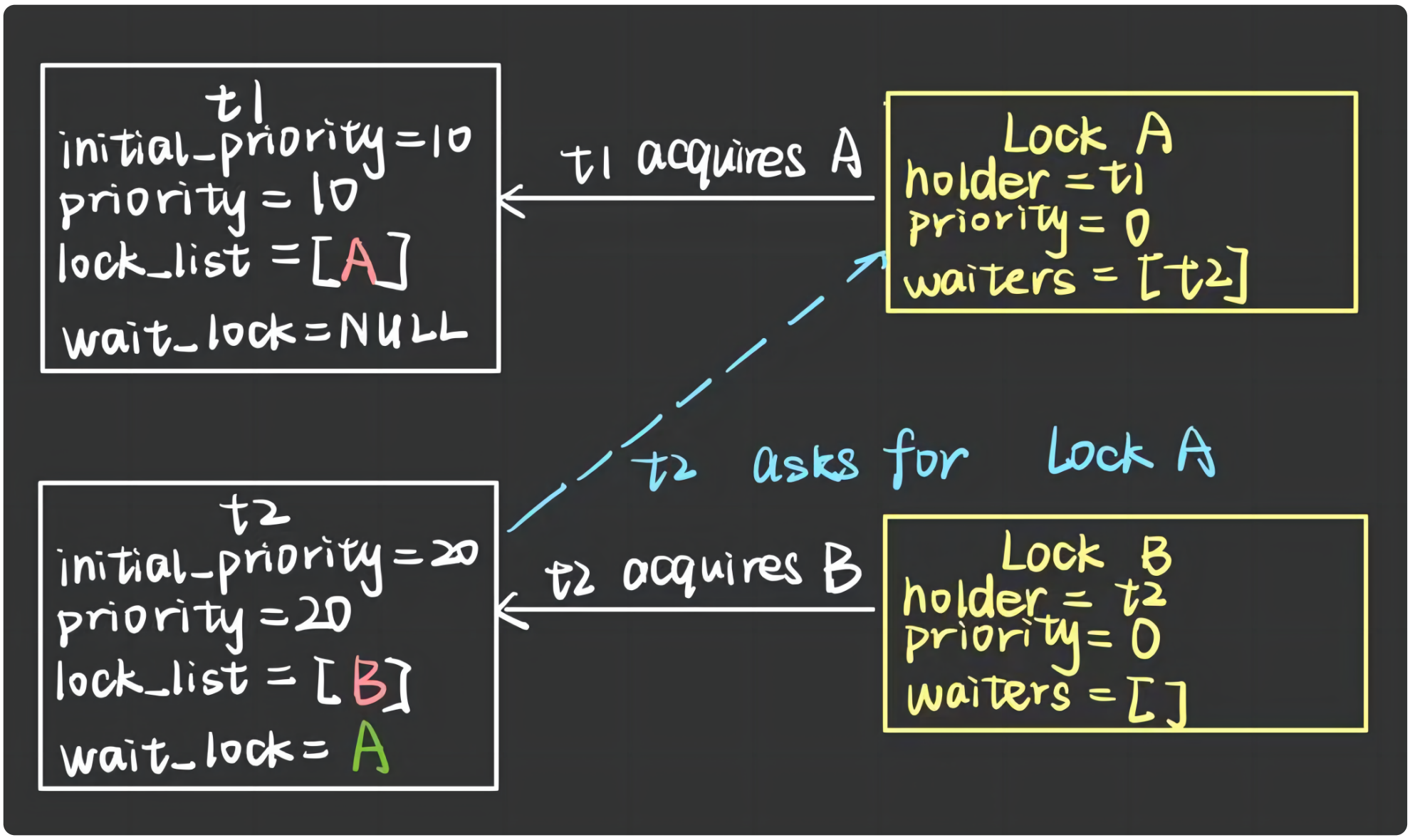
Then we create a thread named **t1**,whose priority is 10.Becase there is only one thread,so **t1** continues running.

Then,**t1** gets **Lock A**.

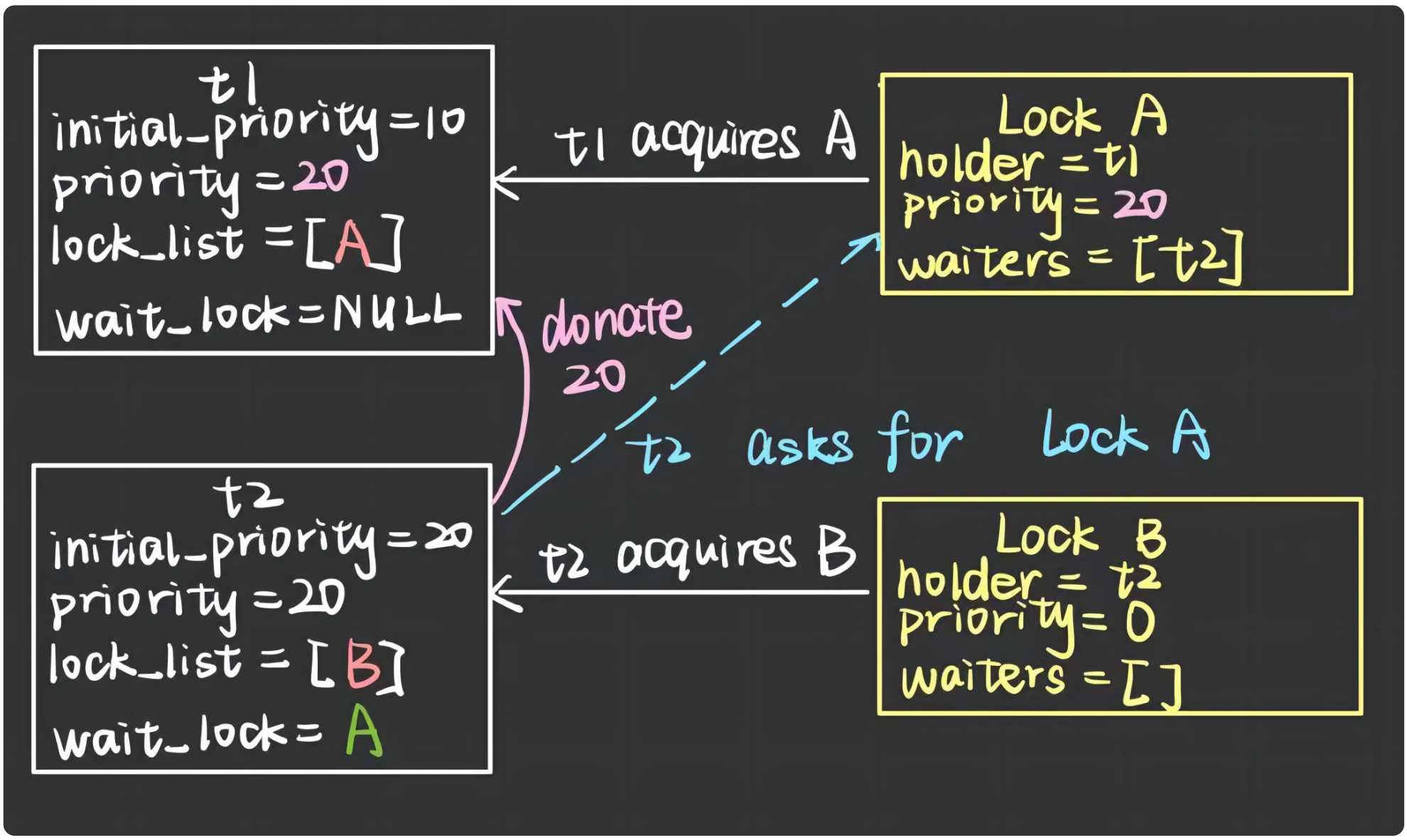
After **t1** acquires **Lock A**,we create a new thread named **t2**,whose priority is 20.So control transfers to t2.

Then,**t2** acquires **Lock B**.

Before **t2** releases **Lock B**,it asks to acquire **Lock A**.

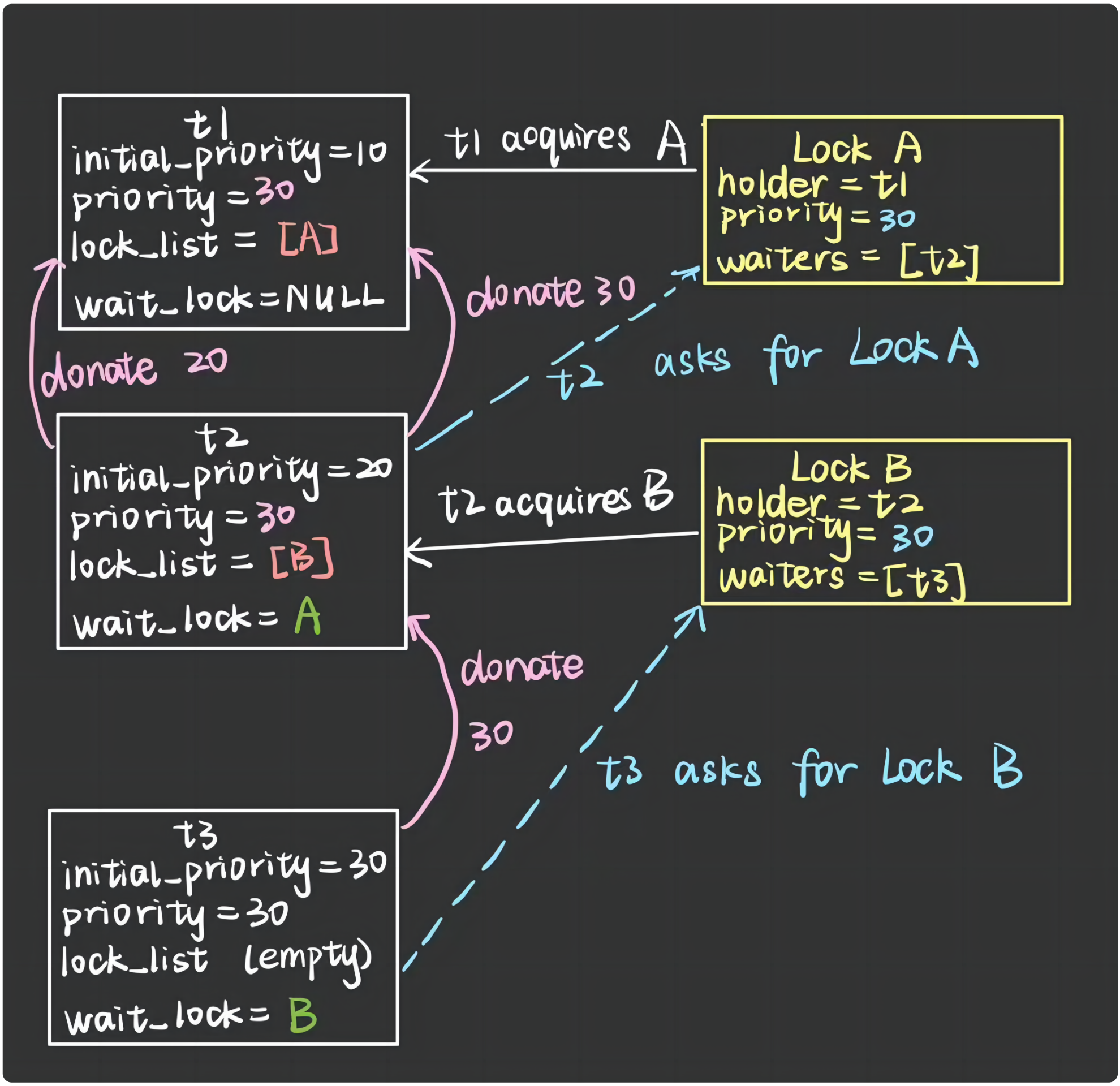


Lock A is held by **t1**,and **t1**'s priority is lower than **t2**.That' where priority donation happens.**t2** donates its priority to **t1**,so **t1**'priority will be 20. **t1** can continue to run.

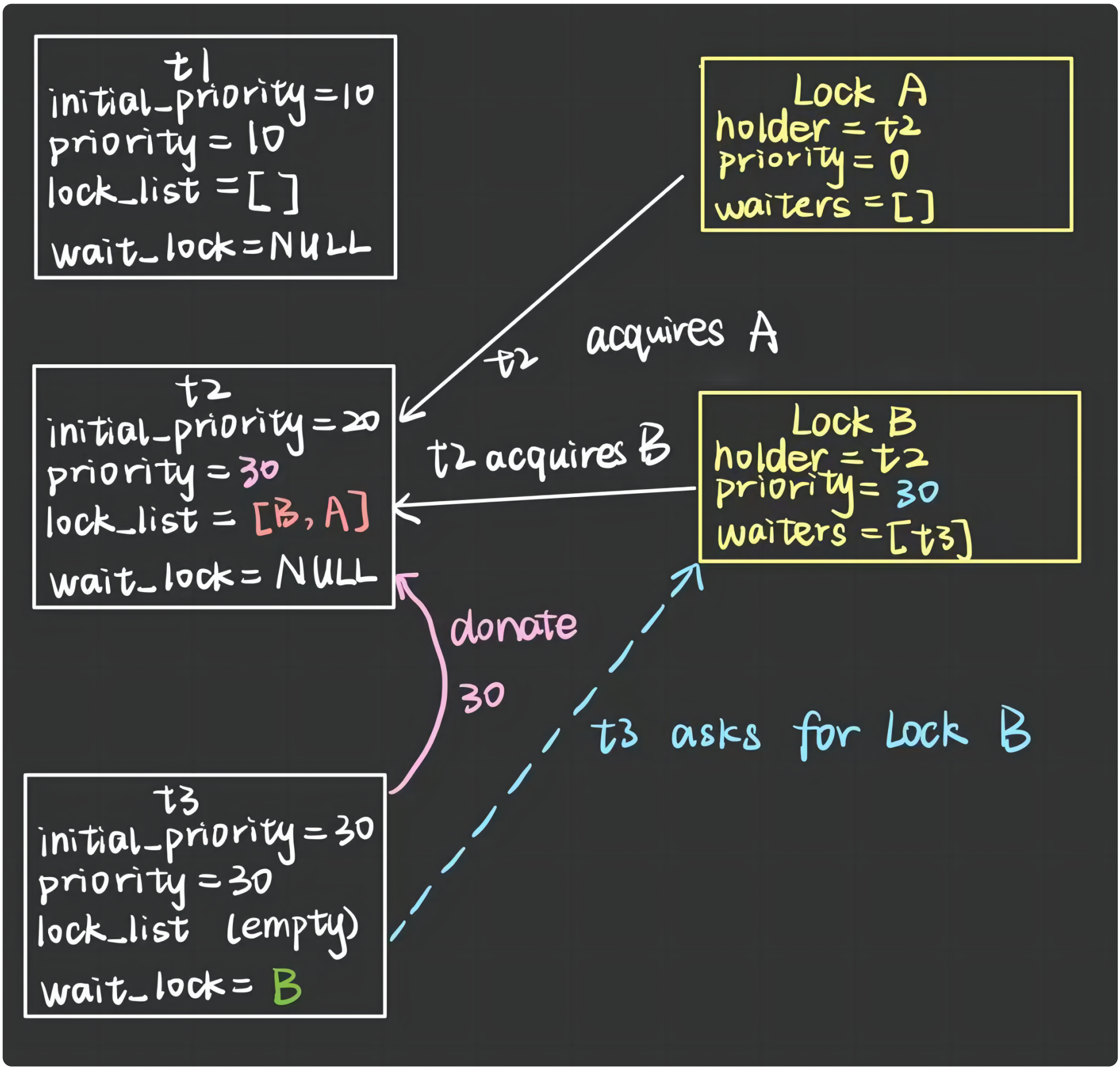


We create **thread t3 with a priority 30**.Then **t3** begins running.At some point,**t3** asks to acquire **Lock B**. But **Lock B** is held by **t2**.**t3 needs to donate its priority to t2.**

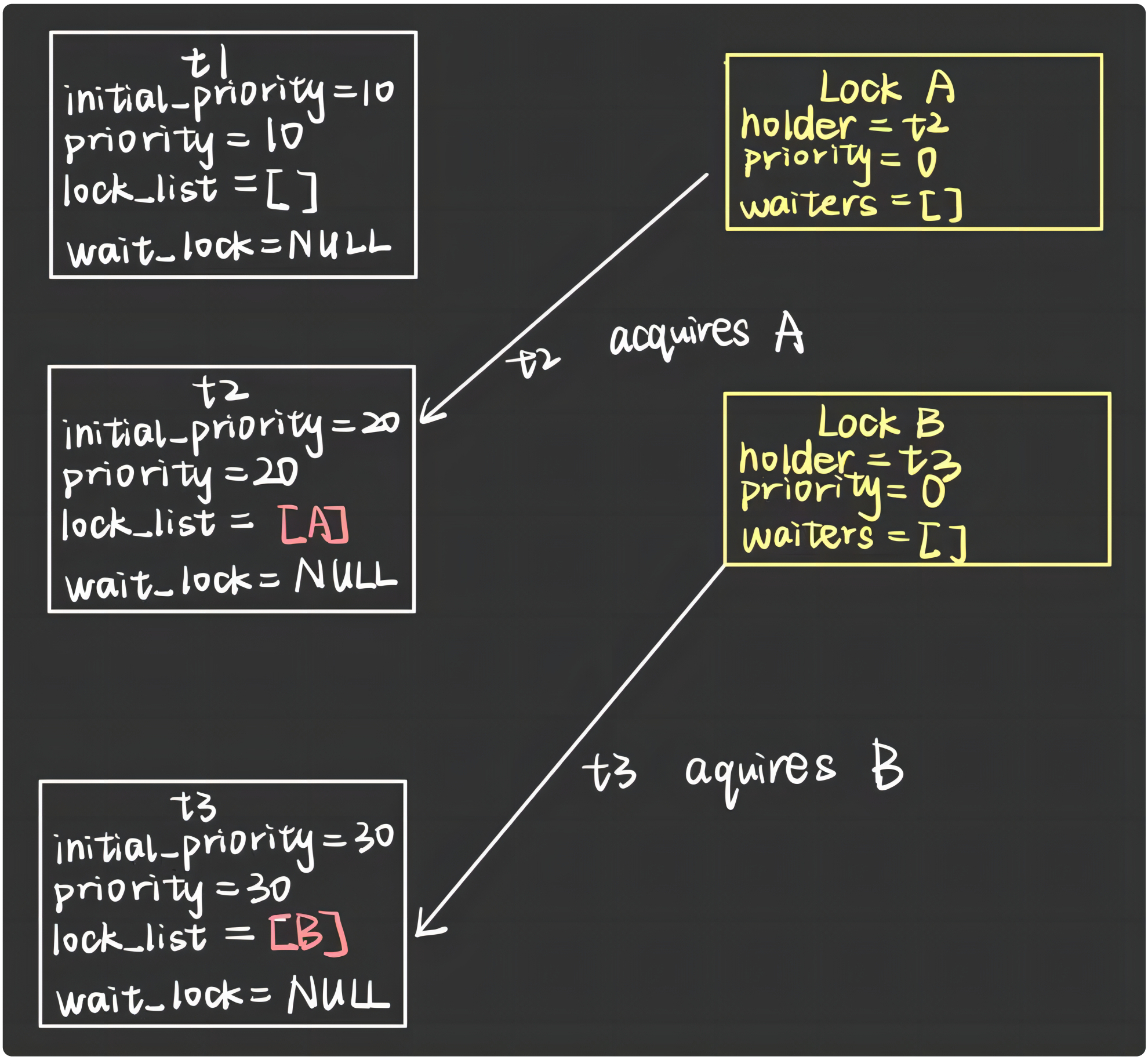
But now **t2** is still blocked,because it is waiting to acquire **Lock A**.So **t2** donates to **t1** again.



Then **t1** continue running.After **t1** releases **Lock A**,its priority will return to its initial priority 10.



After that,**t2** will run and acquire **Lock A**.When **t2** releases **Lock B**,**t3** will get **Lock B** and run.



ALGORITHMS

> B3: How do you ensure that the highest priority thread waiting for

> a lock, semaphore, or condition variable wakes up first?

Lock and Semaphore:

The wake-up of semaphore and the release of lock will both call the fuction `sema_up` .(Lock is implemented through semaphores.)

Then the waiters can be awakened.If there is no waiter,there is nothing to worry.If there are some waiters,we will choose the thread with the highest priority from all the waiters and wake up it.As it shows below:

```
if (!list_empty (&sema->waiters))
{
  t=list_entry(list_min(&sema->waiters,thread_less,NULL),struct thread,elem);
  list_remove(&t->elem);
  thread_unblock(t);
}
```

So we ensure that the highest priority thread waiting for a lock or semaphore wakes up first.

Condition variables:

```
struct semaphore_elem
  {
    struct list_elem elem;              /**< List element. */
    struct semaphore semaphore;         /**< This semaphore. */
    int priority;
  };
```

newly added member:priority

```
struct condition
  {
    struct list waiters;        /**< List of waiting threads. */
  };
```

```
void
cond_wait (struct condition *cond, struct lock *lock)
{
  struct semaphore_elem waiter;

  ASSERT (cond != NULL);
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (lock_held_by_current_thread (lock));

  waiter.priority=thread_current()->priority;

  sema_init (&waiter.semaphore, 0);
  list_push_back (&cond->waiters, &waiter.elem);
  lock_release (lock);
  sema_down (&waiter.semaphore);
  lock_acquire (lock);
}
```

We add a new member `priority` to `struct semaphore_elem` to record the corresponding thread's priority.

```
void
cond_signal (struct condition *cond, struct lock *lock UNUSED)
{
  ASSERT (cond != NULL);
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (lock_held_by_current_thread (lock));

  if (!list_empty (&cond->waiters))
  {
    struct list_elem*e=list_min(&cond->waiters,sema_compare,NULL);
    struct semaphore_elem*waiter=list_entry(e,struct semaphore_elem,elem);
    list_remove(e);
    sema_up(&waiter->semaphore);
  }
}
```

In `cond_signal` ,just as what we do in `sema_up` ,we choose the **waiter** with the highest priority and wake up the corresponding thread.So we ensure that the highest priority thread waiting for a condition variable wakes up first.

> B4: Describe the sequence of events when a call to lock_acquire() causes a priority donation. How is nested donation handled?

Section1:

```
void
lock_acquire (struct lock *lock)
{
  ASSERT (lock != NULL);
  ASSERT (!intr_context ());
  ASSERT (!lock_held_by_current_thread (lock));

  struct thread*cur = thread_current();
  if(list_empty(&lock->semaphore.waiters))
  {
    lock->priority=0;
  }
  else
  {
    lock->priority=list_entry(list_max(&lock->semaphore.waiters,thread_less,NULL)
    ,struct thread,elem)->priority;
  }
  if(lock->holder!=NULL)
  {
    struct semaphore*sema=&lock->semaphore;
      cur->wait_lock=lock;
```

First,we should update `lock` 's priority(We don't update lock's priority in function `lock_release` ,so we need to

update before another thread asks for this lock).Only `lock` 's priority is correct can the priority donation operates well.

Then,we should check if the lock this thread asks for is available.
If the `lock->holder=NULL` ,the lock is vacant.This thread can directly hold the lock.There is no need for priority donation.If `lock->holder!=NULL` ,there are chances that priority donation is needed.

Section 2:

```
if(!thread_mlfqs)
{
  struct thread*rthread=cur;
  struct lock*rlock=rthread->wait_lock;
  while(rlock!=NULL&&rthread!=NULL)
  {
    /*lock->priority needs to be updated too.*/
    if(rthread->priority>rlock->priority)
    {
      rlock->priority=rthread->priority;
    }
    if(rthread->priority > rlock->holder->priority)
    {
      rlock->holder->priority=rthread->priority;
      rthread=rlock->holder;
      rlock=rthread->wait_lock;
    }
    else break;
  }
  /*thread priority change*/
}
```

This section is the critical section to handle priority donation.We use a loop to handle possible nested donation.

`rthead` represents an element in the current donation chain(A chain consists of threads.The chain of ordinary priority donation that isn't nested is a one-element chain.So we can handle ordinary and nested donation together.)
`rlock` represents the lock that `rthead` is waiting to acquire.If `rthead` 's priority is higher than `rlock` 's holder's priority,there will be deadlock.So `rthread` needs to donate its priority to `rlock` 's holder.If `rlock` 's holder is also waiting for a lock,the donation needs to continue until `rthead=NULL` or `rlock=NULL` or `rthead` 's priority is lower than the `rlock` 's holder's.

Not only threads' priority should be updated,but also locks'priority.We declared above that lock's priority records that highest priority among its waiters.Threads' priority changes,so do locks.

Section3:

```
  /*Start waiting*/
  sema_down (&lock->semaphore);
  /*Once thread cur acquire lock,cur's priority will have
  no effect on lock's priority.*/
  cur=thread_current();
  lock->holder = cur;
  cur->wait_lock=NULL;
  /*Add lock to thread's lock_list*/
  list_push_back(&cur->lock_list,&lock->elem);
}
```

Finally,thread will get the lock.Thread's `waitlock` will be `NULL`.Lock's holder should be updated.And `lock` should be put into thread's `lock_list`.

> B5: Describe the sequence of events when lock_release() is called on a lock that a higher-priority thread is waiting for.

Section1:

```
void
lock_release (struct lock *lock)
{
  ASSERT (lock != NULL);
  ASSERT (lock_held_by_current_thread (lock));

  /*the priority are likely to change,we need to remove the current
  running thread from waiters,and also remove lock from thread's lock_list*/
  struct thread*cur=thread_current();
  /*delete this lock from the lock_list*/
  list_remove(&lock->elem);
  lock->holder = NULL;
```

First,we remove `lock` from current thread's `lock_list`.No thread holds the lock,so `lock->holder` should be `NULL`.

Section2:

```
  if(!thread_mlfqs)
  {
    cur->priority=cur->initial_priority;
    if(!list_empty(&cur->lock_list))
    {
      int priority=list_entry(list_max(&cur->lock_list,lock_compare,NULL),
      struct lock,elem)->priority;
      if(priority>cur->priority)
      {
        cur->priority=priority;
      }
    }
  }
  sema_up (&lock->semaphore);
}
```

Because the current thread will no longer hold `lock` ,part of the priority donation doesn't make sense anymore.Only the threads waiting for the rest locks which is held by current thread will donate their priority.

If current thread's `lock_list` is empty,the whole donation will lose effiency.Thread's priority returns to its `initial_priority` .If current thread's `lock_list` isn't empty,we should choose the highest priority among thread's `initial_priority` and its waiters' priority.

Finally,call `sema_up` to wake up one waiter and yield the control.

SYNCHRONIZATION

> B6: Describe a potential race in thread_set_priority() and explain
> how your implementation avoids it. Can you use a lock to avoid
> this race?

Potential race:During priority donation, the lock holder’s priority may be set by it’s donor,at the mean time, the thread itself may want to change the priority.
If the donor and the thread itself set the priority in a different order, may cause a different result.

It can not be avoided using a lock in our implementation, since we didn’t provide the interface and structure to share a lock between donor and the thread itself.

My implementation:
The interrupts shoubld be turned off to avoid this potential race.
But unfortunately,I didn't notice this synchronization problem.So in my code,I don't turn off the interrupts,but it still passes all the tests.This is a bug that has not been detected by testing.

RATIONALE

> B7: Why did you choose this design? In what ways is it superior to
> another design you considered?

There are some questions I think carefully before and during my implementation.

Problem 1:How can we find the thread with the highest priority efficiently?

First,I want to create 64 queues with a unique priority attached to each queue form 63 to 0.Every time we want to find the thread with the highest priority,we can iterate all the queues in descending order of priority(from 63 to 0) until we find a nonempty queue and pop out the first element.This design is intuitive but complex to inplement.

Then I decide to put all ready threads in a list, `ready_list` .To make search more efficiently,I come up with the idea that we can keep `ready_list` in descending order of priority. `list_insert_ordered` can be used to achieve ordered insertion.In this case,we only need to pop out the first element of `ready_list` to get what we want.But this design results in another annoying thing in the follow-up design.
That means every time we change one ready thread's priority,we need to adjust `ready_list` to keep order.

So I finally choose the most simple design:store all the ready threads in `ready_list` and as for insertion just push the new thread at the back of `ready_list` .
We can use `list_max` or `list_min` to find the thread with highest priority.

Problem 2.How can we implement priority donation when priority inversion unluckily happens?

The conflict occurs between a lock's holder and this lock's waiter,so does priority donation.Lock's waiter will donate its priority to lock's holder to avoid deadlock.I finally choose add a new member `priority` to `struct lock` .It records the highest priority of waiters.One unconvenient thing is that we should update lock's priority when priority donation occurs and when adding or removing waiters.

A thread can have more than one lock,so we add a member `lock_list` to record all the locks this thread holds.

`wait_lock` is added to handle nested priority donation.If the waiting thread's priority is higher than `wait_lock` 's holder's priority,the donation will continue.

Problem 3:What's the priority going to be when the current thread releases one lock?
If this thread doesn't have any locks,its priority will return to its initial priority.If this thread still has some locks,its priority will become the highest priority among its current waiters.
So I add `initial_priority` to `struct thread` to record its base priority,and update a thread's priority when this thread releases a lock(in function `lock_release`).
Here also shows the advantage of adding `priority` to `struct lock` .When a thread releases a lock,We can use `list_min` or `list_max` to find the highest priority by comparing the rest locks' priority instead of traversing all the waiters.

Problem 4:In which scenarios should we check the priority queue to determine whether we should enforce the control to yield?

1.A new thread is created.
If the new thread's priority is higher than current thread's priority,control will be transferred to the new thread.

2.Thread's priority is set to a new number.
In function `thread_set_priority` ,we will set current thread's `initial_priority` to a new priority.If current thread's new priority is lower than before,current thread's priority is probably not highest anymore,the control may yield.

3.A lock was released,and kernel wakes up one of its waiters.
This thread will be added to `ready_list` .It may have the highest priority, so control will probably yield.

Advanced Scheduler

DATA STRUCTURES

> C1: Copy here the declaration of each new or changed struct or struct member, global or static variable, typedef, or enumeration. Identify the purpose of each in 25 words or less.

```
int nice;
struct fixed_point recent_cpu;
```

```
static struct fixed_point load_avg;
```

1. `int nice` :A new member added to `struct thread` .Each thread has its own `nice` number.

2. `struct fixed-point recent_cpu` :A new member added to `struct thread` .It will be used to compute thread's priority.

3. `static struct fixed-point load_avg` :A new static variable added to source file thread.c.It will be used to compute thread's priority.

ALGORITHMS

> C2: How is the way you divided the cost of scheduling between code inside and outside interrupt context likely to affect performance?

If the CPU spends too much time on calculations for recent_cpu, load_avg and priority, then it takes away most of the time that a thread before enforced preemption. Then this thread can not get enough running time as expected and it will run longer. This will cause itself got blamed for occupying more CPU time, and raise its load_avg, recent_cpu, and therefore lower its priority. This may disturb the scheduling decisions making. Thus, if the cost of scheduling inside the interrupt context goes up, it will lower performance.

RATIONALE

> C3: Briefly critique your design, pointing out advantages and disadvantages in your design choices. If you were to have extra time to work on this part of the project, how might you choose to refine or improve your design?

One advantage of my design is that it's clear and simple.That's why I choose it.We don't need to worry about how to maintain the order and where we have to resort the `ready_list` .The simpler the design is, the less mistakes we are likely to make.But the biggest problem is its low efficiency.

If I have extra time working on this lab,I will probably choose to apply 64 queues. I used only one queue -- the ready_list that Pintos originally have. But as the same as what we did for the task 2, we didn't arrange `ready_list` in order -- that means we need to traverse the list to find the thread with the highest priority.The time complexity is O(n). Every time we do thread switching,we need to search `all_list` , which will take O(n) time. It will make a thread’s running ticks shorter than it is expected,beacuse thread switching may happen quite often. If we use 64 queues for the ready threads, we can put the 64 queues in an array with index

equaling to its priority value. When the thread is first inserted, it only need to index the queue by this thread's priority. This will take only O(1) time.And we want to find the next thread to run,we don't need to traverse the whole `ready_list` ,so it will be better in performance.

Thus, I'd like to implement 64 queues instead of 1 queue for ready threads,because it is better in performance.

> C4: The assignment explains arithmetic for fixed-point math in detail, but it leaves it open to you to implement it. Why did you decide to implement it the way you did? If you created an abstraction layer for fixed-point math, that is, an abstract data type and/or a set of functions or macros to manipulate fixed-point numbers, why did you do so? If not, why not?

As mentioned in the BSD scheduling manual, recent_cpu and load_avg are real numbers, but pintos disabled float numbers. So we define a new data type `fixed-point` numbers to represent recent_cpu and load_avg.

```
#define FIXED_NUM 14
#define FP (1<<FIXED_NUM)
struct fixed_point
{
  int f;
};
```

We also define a set of inline funciotn to manipulate fixed-point numbers in the new created header fixed-point.h under thread.

Reasons for my design:

1. Data abstraction and inline function supply interfaces we can directly use.
2. Hiding the real calculatation process behind the abstraction barriers makes it easier for us to use it and debug our code.