In device.c:

```
/* Sleeps for approximately TICKS timer ticks.  Interrupts must
be turned on. */
void
timer_sleep (int64_t ticks)
{
 int64_t start = timer_ticks ();
 ASSERT (intr_get_level () == INTR_ON);
 while (timer_elapsed (start) < ticks)
   thread_yield ();
}
```

As we can see on the code above, when the timer_sleep() is called, it goes to a “while” loop until the timer elapsed time is bigger than the ticks. This is a wasted CPU resource as it will remain idle until the timer has expired. To counter this problem, let’s go to thread.h file and introduce a modified thread structure in the file:

thread.h

```
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. */
 int64_t sleep_ticks;                /* time to sleep in ticks */
 #ifdef USERPROG
 /* Owned by userprog/process.c. */
 uint32_t *pagedir;                  /* Page directory. */
 #endif
 /* Owned by thread.c. */
 unsigned magic;                     /* Detects stack overflow. */
};
```

Now if you look closely, we have added sleep_ticks variable in the thread structure to detect how many ticks the thread has to sleep. So, let’s modify the timer_sleep():

timer.c

```
/* Sleeps for approximately TICKS timer ticks.  Interrupts must
be turned on. */
void
timer_sleep (int64_t ticks)
{
 ASSERT (intr_get_level () == INTR_ON);
 // assign requested sleep time to current thread
```

```
 thread_current()->sleep_ticks = ticks;
 // disable interrupts to allow thread blocking
 enum intr_level old_level = intr_disable();
 // block current thread
 thread_block();
 /* set old interrupt level which was used before the current thread was blocked
 to ensure that no other logic crashes */
 intr_set_level(old_level);
}
```

What the code does is that we assign the sleep_ticks variable of the thread with the defined ticks. We disable our interrupt handle to ensure serialization and block the thread in the thread_block(). Now, the timer_interrupt() will always be called for each elapsed tick:

timer.c

```
/* Timer interrupt handler. */
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
 ticks++;
 thread_tick ();
 // Check each thread with wake_threads() after each tick. It is assumed that
 // interrupts are disabled  because timer_interrupt() is an interrupt handler.
 thread_foreach(wake_threads, 0);
}
```

The thread_foreach() will check every thread and execute wake_threads(). And for your information, we have to define that wake_threads() in our timer.c. It basically a function to check whether a thread has to wake up or not.

timer.c

```
/**
* Function for waking up a sleeping thread. It checks
* whether a thread is being blocked. If TRUE, then
* check whether the thread's sleep_ticks has reached 0 or not
* by decrementing it on each conditional statement.
* If the thread's sleep_ticks has reached 0, then unblock the
* sleeping thread.
*/
static void
wake_threads(struct thread *t, void *aux)
{
 if(t->status == THREAD_BLOCKED)
 {
  if(t->sleep_ticks > 0)
  {
   t->sleep_ticks--;
   if(t->sleep_ticks == 0)
   {
    thread_unblock(t);
```

```
   }
  }
 }
}
```

Essentially, when the wake_threads() is called, it checks if the current thread is being blocked. If it is blocked, then we know that the thread is currently asleep. If the ticks of that thread has not expired (not equal to 0), then we need to decrement the ticks of that thread to indicate that one tick has passed. If the ticks of that thread has expired (equal to 0), now we know that it is okay for the thread to wake up (We perform an unblock() operation to that particular thread).