Task

Reimplement `timer_sleep()`, defined in 'devices/timer.c'. Although a working implementation is provided, it "busy waits," that is, it spins in a loop checking the current time
and calling `thread_yield()` until enough time has gone by. Reimplement it to avoid busy waiting.

Pintos Documentation Notes

2.1 Background

2.1.1 Understanding Threads

- When a thread is created, you are creating a new context to be scheduled.
- You provide a function to be run in this context as an argument to `thread_create()`.
- The first time the thread is scheduled and runs, it starts from the beginning of that function and executes in that context.
- When the function returns, the thread terminates.
- Each thread, therefore, acts like a mini-program running inside Pintos, with the function passed to `thread-create()` acting like `main()`.

2.1.2 Source Files

`thread.c` and `thread.h`:
Basic thread support. Much of your work will take place in these files.
'thread.h' defines struct thread, which you are likely to modify in all four projects.
`timer.c` and `timer.h`:
System timer that ticks, by default, 100 times per second. You will modify this code in this project.

A.2 Threads

`enum thread_status status`
The thread's state, one of the following:
THREAD_RUNNING
The thread is running. Exactly one thread is running at a given time. `thread_current()` returns the running thread.
THREAD_READY
The thread is ready to run, but it's not running right now. The thread could be selected to run the next time the scheduler is invoked. Ready threads are kept in a doubly linked list

called ready_list.

THREAD_BLOCKED

The thread is waiting for something, e.g. a lock to become available, an interrupt to be invoked. The thread won't be scheduled again until it transitions to the THREAD_READY state with a call to `thread_unblock()`. This is most conveniently done indirectly, using one of the Pintos synchronization primitives that block and unblock threads automatically

THREAD_DYING

The thread will be destroyed by the scheduler after switching to the next thread.

Code Description

Methods Implemented:

`thread.c`:

```
/*
* Blocks a thred and sets the wakeUpTick of the current thread to the pased -
*/
void our_tsleep(int64 wakinguptick)

/*
* Scans the sleeping list, and checks if the timer has expired or not. If it
*/
void scan_tsleep()
```

`intr_disable()`:

Turns interrupts off, returns the previous interrupt state.

`intr_set_level(prev)`

Turn interrupts on or off according to level. Returns the previous interrupt state.

`list_push_back():`

Insert ELEM at the end of the list, so that it becomes the back in the LIST.

`list_begin():`

Returns the beginning of LIST

`list_end():`

Returns the end of LIST

`timer_ticks():`

Returns the number of timer ticks since the OS booted

`thread_tick():`

Called by the timer interrupt at each timer tick. It keeps track of thread statistics and triggers the scheduler when a time slice expires. Also called `scan_tsleep()` every time as well whenever the tick is increased to check if the timer has expired or not.