User threads

Pintos is a multithreaded kernel (i.e. there can be more than one thread running in the kernel). While working on Project User Programs, you have no doubt worked with the threading interface in the kernel. In `threads/thread.c`, `thread_create` allows us to create a new kernel thread that runs a specific kernel task. Analogously, the `thread_exit` function allows a thread to terminate itself. You should read and understand the kernel threading model.

On the other hand, as it were in Project User Programs, each user process only had one thread of control. In other words, it is impossible for a user program to create a new thread to run another user function. There was no analog of `thread_create` and `thread_exit` for user programs. In a real system, user programs can indeed create their own threads. We saw this via the POSIX `pthread` library.

For this task, you will need to implement a simplified version of the `pthread` library in `lib/user/pthread.h`. User programs should be allowed to create their own threads using the functions `pthread_create` and `pthread_exit`. Threads can also wait on other threads with the `pthread_join` function, which is similar to the `wait` syscall for processes. Threads should be able to learn their thread IDs (TIDs) through a new `get_tid` syscall. You must also account for how the syscalls in Project User Programs are affected by making user programs multithreaded.

In Project User Programs, whenever a user program (which consisted of just a single thread) trapped into the OS, it ran in its own dedicated kernel thread. In other words, user threads had a 1-1 mapping with kernel threads. For this task, you will need to maintain this 1-1 mapping. That is, a user process with n user threads should be paired 1-1 with `n` kernel threads, and each user thread should run in its dedicated kernel thread when it traps into the OS. You should **not** implement green threads , which have a many-to-one mapping between user threads and kernel threads. Green threads are not ideal, because as soon as one user thread blocks, e.g. on IO, all of the user threads are also blocked.

In addition, you must also implement user-level synchronization. After all, threads are not all that useful if we can't synchronize them properly with locks and semaphores. You will be required to implement `lock_init`, `lock_acquire`, `lock_release`, `sema_init`, `sema_down`, and `sema_up` for user programs.

Workflow Recommendations: this task is most easily done in small steps. Start by implementing a barebones `pthread_create` and `pthread_execute` so that you pass `tests/userprog/multithreading/create-simple`. Then, slowly add more and more features. It is easier to augment a working design than to fix a broken one. Make sure to carefully track resources. Everything that you allocate must be freed!

TABLE OF CONTENTS
