Pthread library

TABLE OF CONTENTS

1 Threading
2 User-level synchronization

Threading

A subset of the `pthread` (Pintos thread) library is provided for you in `lib/user/pthread.h`. These functions serve as the glue between the high-level API of `pthread_create`, `pthread_exit`, and `pthread_join` and the low-level system call implementation of these functions. We'll walk you through how the `pthread` library works by starting at the high-level usage in one of our tests, and walk down the stack until we get to the kernel syscall interface.

The test we will walk through is `tests/userprog/multithreading/create-simple.c`.

In the `create-simple` test, we see how the high-level API of the threading library is supposed to work. The main thread of the process first runs `test_main`. It then creates a new thread to run `thread_function` with the `pthread_check_create` call, and waits for that thread to finish with the `pthread_check_join`. The expected output of this test is shown in `tests/userprog/multithreading/create-simple.ck`.

The functions `pthread_check_create` and `pthread_check_join` are simple wrappers (defined in `tests/lib.c`) around the "real" functions, `pthread_create` and `pthread_join`, that take in roughly the same values and return the same values as `pthread_create` and `pthread_join`, and ensure that `pthread_create` and `pthread_join` did not fail. The APIs for `pthread_create` and `pthread_join` are:

- `tid_t pthread_create(pthread_fun fun, void* arg)`

  A `pthread_fun` is simply a pointer to a function that takes in an arbitrary `void*` argument, and returns nothing. This is defined in `user/lib/pthread.h`. So, the arguments to `pthread_create` are a function to run, as well as an argument to give that function. This function creates a new child thread to run the `pthread_fun` with argument `arg`. This function returns to the parent thread the TID of the child thread, or `TID_ERROR` if the thread could not be created successfully.

- `bool pthread_join(tid_t tid)`

The caller of this function waits until the thread with TID `tid` finishes executing. This function returns true if `tid` was valid.

The implementation of `pthread_create` and `pthread_join` are in the file `lib/user/pthread.c`. They each are simple wrappers around the functions `sys_pthread_create` and `sys_pthread_join`, which are syscalls for the OS that you will be required to implement. Their APIs are similar to `pthread_create` and `pthread_join`, and are as follows:

- `tid_t sys_pthread_create(stub_fun sfun, pthread_fun tfun, const void* arg)`

  The `sys_pthread_create` function creates a new thread to run `stub_fun sfun`, and gives it as arguments a `pthread_fun` and a `void*` pointer, which is intended to be the argument of the `pthread_fun`. It returns to the parent the TID of the created thread, or `TID_ERROR` if the thread could not be created.

  What is a stub function? There is only one stub function that we are concerned with here, called `_pthread_start_stub` defined in `lib/user/pthread.c`, and its implementation is copied below. This function returns nothing but takes two arguments: a function to run, and an argument for that function. The stub function runs the function on the argument, then calls `pthread_exit()`. `pthread_exit()` is a system call that simply kills the current user thread.

  ```
  /* OS jumps to this function when a new thread is created.
  OS is required to setup the stack for this function and
  set %eip to point to the start of this function */
  void _pthread_start_stub(pthread_fun fun, void* arg) {
    (*fun)(arg); // Invoke the thread function
    pthread_exit(); // Call pthread_exit
  }
  ```

  Why this extra layer of indirection? You might have noticed in `tests/userprog/multithreading/create-simple.c` that `pthread_exit()` was never called; instead, as soon as the created thread returns from `thread_function`, it is presumed to have been killed. The stub function is how this is implemented: the OS actually jumps to `_pthread_start_stub` instead of directly jumping to `thread_function` when the new thread is created. The stub function then calls `thread_function`. Then, when `thread_function` returns, it returns back into `_pthread_start_stub`. Then, the implementation of `_pthread_start_stub` kills the thread by calling `pthread_exit()`.

- `tid_t sys_pthread_join(tid_t tid)`

The caller of this function waits until the thread with TID `tid` finishes executing. This function returns the TID of the child it waited on, or `TID_ERROR` if it was invalid to wait on that child.

- `void sys_pthread_exit(void) NO_RETURN`

  This function terminates the calling thread. The function `pthread_exit` simply calls this function.

The functions `sys_pthread_create`, `sys_pthread_join`, and `sys_pthread_exit` are system calls that you are required to implement for this project. They have slightly different APIs than the high level `pthread_create`, `pthread_join`, and `pthread_exit` functions defined in `lib/user/pthread.h`, but are fundamentally very similar. We have set up the user-side of the syscall interface for you in `lib/syscall-nr.h`, `lib/user/syscall.c`, and `lib/user/syscall.h`, and it is your job to implement these system calls in `userprog/` in the kernel.

User-level synchronization

Our `pthread` library also provides an interface to user-level synchronization primitives. See `lib/user/syscall.h`. We define the primitives `lock_t` and `sema_t` to represent locks and semaphores in user programs. A `lock_t` (or `sema_t`) is a `char` because we only require processes to support up to 2^8 locks and 2^8 semaphores, and a `char` is 1 byte or 8 bits. A `lock_t` (or `sema_t`) is thus an identifier for the underlying `struct lock` (or `struct semaphore`) kernel-level synchronization primitive that the user-level is tied to. You can change these definitions if you'd like, but we found the current definitions sufficient for our implementation. We provide the following syscall stubs:

- `bool lock_init(lock_t* lock)`

  Initializes `lock` by registering it with the kernel, and returns true if the initialization was successful. In `tests/lib.c`, you will see we define `lock_check_init`, which is analogous to `pthread_check_create` and `pthread_check_join`; it simply verifies that the initialization was successful.

  You do not have to handle the case where `lock_init` is called on the same pointer multiple times; you can assume that the result of doing so is undefined behavior. However, `lock_init` should still create a new lock even if the pointer points to the same value that an earlier argument to the syscall pointed to (as the value is usually unitialized garbage).

- `void lock_acquire(lock_t* lock)`

  Acquires `lock` and exits the process if acquisition failed. The syscall implementation of `lock_acquire` should return a boolean as to whether acquisition failed; the user level implementation of `lock_acquire` in `lib/user/syscall.c` handles termination of the process. You should not update the `lock_acquire` (or for that matter, any of the below functions) code in `lib/user/syscall.c` to remove the `exit` call; it will simply make debugging more difficult.

- `void lock_release(lock_t* lock)`

  Releases `lock`, and exits the process if the release failed. The syscall implementation of `lock_release` should return a boolean as to whether `release` failed.

- `bool sema_init(sema_t* sema, int val)`

  Initializes `sema` to `val` by registering it with the kernel, and returns true if the initialization was successful. In `tests/lib.c`, you will see we define `sema_check_init`, which is analogous to `pthread_check_create` and `pthread_check_join`; it simply verifies that the initialization was successful.

  You do not have to handle the case where `sema_init` is called on the same argument multiple times; you can assume that the result of doing so is undefined behavior. However, `sema_init` should still create a new semaphore even if the pointer points to the same value that an earlier argument to the syscall pointed to (as the value is usually unitialized garbage).

- `void sema_down(sema_t* sema)`

  Downs `sema` and exits the process if the down operation failed. The syscall implementation of `sema_down` should return a boolean as to whether the down operation failed.

- `void sema_up(sema_t* sema)`

  Ups `sema`, and exits the process if the up operation failed. The syscall implementation of `sema_up` should return a boolean as to whether the up operation failed.
