COSC 301: Operating Systems
Lab 5: Threads

0. The `pthreads` API

   `pthreads` is a standard library available on many operating systems that provides a portable API for creating threads, using mutex locks, and other thread-related operations. Item 0 in this lab is just an overview of a couple functions in the `pthreads` library. Exercise 3 introduces a few more functions in the `pthreads` library, and next week's lab will introduce you to a few more.

   A thread is an execution context that exists within a traditional process. You can think of a process as a container of resources for an application (e.g., a heap, some code, open files, network connections). A thread is just an execution context that exists within this container.

   To create a new thread, you use the `pthread_create` function. This function requires a pointer to a `pthread_t`, which is just a variable to hold some thread state, a pointer to function which is where the new execution context will start running, and an argument to the thread function (or `NULL`):

```
#include <pthread.h> // need to do this at the top of a file to use
                     // pthreads functions

...

void *worker_function(void *threadarg) {
    printf("I'm running in a thread!\n");
    printf("And now I'm going away :-(");

    return NULL;
}

int main() {

    pthread_t mythread;  // a special type that represents a thread
    if (pthread_create(&mythread, worker_function, NULL, NULL) < 0) {
        fprintf(stderr, "pthread create failed!\n");
    }
```

   In the above example, `worker_function` is the thread entrypoint. This is the function in which the new thread will start running. This function *must* return a pointer to `void`, and take a pointer to `void` as a parameter. To pass an argument to the function, you need to give it as the last argument to `pthread_create`, and make an explicit cast inside the `worker_function` from `void *` to what ever pointer type you actually gave as the fourth argument to `pthread_create`. Note that within `worker_function`, we explicitly return `NULL` at the end, since the return value of the function requires a pointer type.

   If `pthread_create` is analogous to `fork`, the analogous system call for `wait` is `pthread_join`. It works as follows:

```
pthread_join(&mythread, NULL);
```

Calling `pthread_join` will cause the calling thread to block until the thread represented by `mythread` is finished.

Refer to the Thread API chapter in OSTEP and the `man` pages for more detail.

Lastly, it is important to note that special flags to a compiler are often required to correctly compile a multithreaded program. Typically, this flag is `-pthread` on gcc. You'll notice that in the `Makefile` for this lab, `CFLAGS` includes this flag to tell gcc that we're using the pthreads library.

---

1. Put the code for lab 5 into a (new or existing) `git` repository. For two of the exercises below you'll need to just give the `diff` (difference) between the original code you pull, and your modifications. Using `git` will ensure that creating the diff will be easy. Do this before moving on. Really.

2. What is the output of the following program? (You should just hand-trace it - Don't compile and run it.) Remember that threads share the same address space.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int value = 0;
void *runner(void *param) {
     value = 5;
     return NULL;
}

int main(int argc, char **argv) {
    pid_t pid;
    pthread_t tid;

    pid = fork();
    if (pid == 0) {
        printf("CHILD: before %d\n", value);
        pthread_create(&tid, NULL, runner, NULL);
        pthread_join(tid, NULL);
        printf("CHILD: after %d\n", value);
    } else {
        printf("PARENT: before %d\n", value);
        wait(NULL);
        printf("PARENT: after %d\n", value);
    }
    exit(0);
}
```

   Write up your answers to this question in a text file (which you'll later submit to Moodle).

3. More processes versus threads. Run `make` to compile (among other programs), `procshare` and `threadshare`. Read the code in each program, and then run each program a few times at the command line. Briefly describe and explain the behavior you observe. (You **must** run these programs on a real machine, not in a VM!)

   Write up your answers to this question in your submission file.

4. The program `threadshare` exhibits what is called a *race condition*. Modify the `threadshare.c` to avoid the race condition by inserting locks around the critical section in the worker thread function.

   To do that, you'll need to:

1. Create a *mutex lock* in the main thread (before the worker threads are started). Although I pretty much hate global variables, it's ok for this *one* program to create a global mutex variable, like:

   ```
   pthread_mutex_t happy_mutex;
   ```

   Of course, you can use what ever variable name you'd like.
   In `main`, you'll need to initialize the mutex, using the following:

   ```
   pthread_mutex_init(&happy_mutex, NULL);
   ```

2. Add calls to `pthread_mutex_lock` and `pthread_mutex_unlock` in appropriate places inside the worker thread. The format of these calls is:

   ```
   pthread_mutex_lock(&happy_mutex);   // lock the mutex
   pthread_mutex_unlock(&happy_mutex); // unlock the mutex
   ```

3. Free the mutex at the end of main:

   ```
   pthread_mutex_destroy(&happy_mutex);
   ```

Note: all these function calls *do* return values. You should read the `man` pages to find out more about how their behavior.

Don't forget to test `threadshare` once you've fixed the race.

Run `git diff threadshare.c` to show the difference between your modified version and the original version. Copy/paste this diff into your submission file.

5. The last task for this lab is to modify a linked list library to make it *threadsafe*. That is, once you are done modifying the library, it should permit multiple threads to simultaneously call functions within the library to add, remove, print, or clear the list.

   There are two files that are part of the library: `list.c` and `list.h`. In `list.h`, you'll see that there are two `struct` definitions, and 5 function prototypes. The structure type `list_t` is the only type that a user of the linked list library will use (as you'll see from the prototype definitions of the 5 list-related functions). You'll need to add one or more mutex locks to the library, and ensure that the functions `list_clear`, `list_add`, `list_remove`, and `list_print` are each threadsafe. The initialize function `list_init` does *not* need to be threadsafe - in fact, it is in this function that you'll probably want to initialize and set up any locks you need.

   When you run `make`, a test program called `mtlist` (for "multi-threaded list") will be created. If you type `./mtlist -h`, you'll see some help for running this program: you can use different options given to the program to start up any number of threads, and you can specify the total number of items to add to the list (which will be divided among however many threads you decide to create). Read through `mtlist.c` if you want to see what it's doing in more detail. The only thing you really need to know is that it will be making calls into your (soon-to-be threadsafe) linked list library.

   Run `git diff list.?` to show the difference between your modified list files, and the original versions. Copy/paste this diff into your submission file.