CS355 Lab Notes #2 Threads

Start by reading the manual pages for POSIX threads related Clib functions, such as `pthread_create`, `pthread_join`, etc.

Note that when using these functions, you need to include header file `pthread.h` and compile with flags `-lpthread`.

Reentrancy: POSIX.1 and C-language functions were originally written to work in an environment of single-threaded processes. Reentrancy was not an issue in their design: the possibility of a process attempting to "re-enter" a function through concurrent invocations was not considered, because threads - the enabler of concurrency within a process - were not anticipated. So, as it turns out, some POSIX.1 and C-language functions are inherently non-reentrant with respect to threads. For example, some functions (such as `asctime()`) return a pointer to a result stored in memory space allocated by the function on a per-process basis. Such a function is non-reentrant, because its result can be overwritten by successive invocations. Other POSIX.1 and C-language functions, while not inherently non-reentrant, may be implemented in ways that lead to non-reentrancy. For example, some functions (such as `rand()`) store state information (such as a seed value, which survives multiple function invocations) in memory space allocated by the function on a per-process basis. The implementation of such a function is non-reentrant if the implementation fails to synchronize invocations of the function and thus fails to protect the state information. The problem is that when the state information is not protected, concurrent invocations can interfere with one another (for example, see the same seed value).

Functions must be reentrant in an environment of multithreaded processes, in order to ensure that they can be safely invoked by concurrently executing threads. POSIX.1c takes three actions in the pursuit of reentrancy. First, POSIX.1c imposes reentrancy as a general rule: all functions, unless explicitly singled out as exceptions to the rule, must be implemented in a way that preserves reentrancy. Second, POSIX.1c redefines `errno`. Third, for those functions whose interface specifications preclude reentrancy, POSIX.1c defines alternative "reentrant" versions. The new functions have "_r" appended to the function names (that is, "asctime_r()", and so on). To achieve reentrancy, the new "_r" functions replace library-allocated structures with application-allocated structures that are passed as arguments to the functions at invocation.

When writing threaded C programs, it is a good habit to always include the following:

```c
#ifndef __REENTRANT
    #define __REENTRANT
#endif
```
Lab exercise:

Write a program that takes a list of arguments, and will create a corresponding number of threads, each of which will print the respective argument a random number of times between 2-10. Wait for all threads to finish and report status before exiting.

Note that you will need `rand_r()`, the reentrant version of `rand()`.

Also, the program should be able to take a theoretically unlimited number of arguments, which means you need to dynamically allocate memory based on `argc`. If you are not sure how to do that, start by limiting the number of arguments to, say 5, so that you can use an array instead.

Copy the program `threads` from `~dxu/handouts/cs355`, run and study it so that you get an idea of how it is supposed to work.