CS180 Notes 3-3-15

- Thread Libraries
    - Similar to processes having parent/child, threads have main/worker relationships
    - Threads are managed with three methods: *pthread_create*, *pthread_join*, and *pthread_exit*.
        - *pthread_create* has four parameters
            - A pthread_t ID
            - Specifiers for the thread
            - The new function to be called
            - Parameters to the new function
                - If multiple parameters, must pass as a struct
        - *pthread_exit* can be an expensive call, so sometimes it is best to let the OS reclaim the thread
        - Valgrind can provide a false memory leak with *pthread_exit*.
        - As always, check the return values!
- All thread functions have the same type of signature
    - *void *[name](void *p)*
    - Assists with being generic
    - Does not return to the caller
- All global data is shared between all threads
    - No extra work to share data
    - Safe for reads, but not safe for writing.
- Thread IDs tend to be very long, so that's why we are giving the tid by 1000000 to have more reasonable IDs to print.
- While we are waiting with *pthread_join*, the thread is now blocked until that thread completes.
    - It doesn't matter if other threads finish first. We still wait for the one we designate.
    - After a join returns, the thread is terminated.
- We have the same process ID as threads are inside processes.
    - Thread IDs are (effectively) unique.
    - There is absolutely no way to control what thread runs at a given time.
    - There is a way to make a thread idle.
- In production code, random number generators are not considered thread safe.
- Example where threads are insanely useful: File search on a hard disk
    - Create threads to search each drive in the computer
        - No point in putting multiple threads per drive – the I/O is already the bottleneck
- A way to avoid using global data is to communicate using data structs.

    - When using a thread's parameter, we must cast the data into the proper type we want.
        - Example: (data_struct *)data (from void *data)
    - Since we're scoped into main, we must pass array data by structs (see example in notes)
        - Because the data struct is the name for all threads, we could have passed the one struct to each thread as opposed to a struct for each thread.
        - It is potentially worth it to check for type of data passed into the thread's new function for safety.
- Each thread is passed a pointer to its own data
    - Threads need to know what data to expect.
    - The data must still remain alive while other threads are potentially using it
- **stdio** library functions are thread safe with GNU and Microsoft compilers
    - Related to the concept of re-entry. (Executing the same code at the same time)
    - *localtime* is not thread safe, but *localtime_t* is.
    - *strace* is not thread safe as static structures are used
    - We can still get interleaved output, but nothing will be corrupted.
- Circle, Leibniz, and atan methods
    - Before, we had these functions running serially – one after another.
        - Call, wait to finish, call the next one, wait, etc.
    - With threads (and processes), we can all three functions concurrently, which drastically reduces the run-time of our program.
        - The user time is still the cumulative time of all the process put together.
        - Remember that threads share global data. Fortunately, each method only writes to its own global data, so we are okay.