COMP 3430 Summer 2017 Assignment 2

Due Date: Wednesday, Mar 6, 11:59 PM

Notes and Style

- must be written in C. No C++.
- The prof has provided a working demo. ~young/comp3430/frogger. Run it on aviary
- your assignment code must be handed in electronically using the D2L drop box. Try early and re-submit later.
- Include a Makefile. If "make" doesn't compile and produce an executable, your assignment **will not be marked**
- Your program must run. Programs that do not run, or segfault before working at all, **will not be marked**. Save early and often, and use version control. Submit often, only submit working versions.
- Your program must compile with –Wall, with no warnings showing. You will lose marks for warnings
- Include a "listing" file, a single .txt file with all your source files concatenated into it. Good header text in EACH file is imperative here.
- Confusion over D2L at the last minute is your own fault.
- Use common-sense commenting. Provide function headers, comment regions, etc. No particular format or style is required.
- Error checking is extremely important in real world OS work, and you must do it rigorously. Error handing, however, is hard in C. Try to handle gracefully when you can, but for hard errors (out of memory, etc.), hard fail (exit) is OK.
- Use clang or gcc, make sure your Makefile uses the one you tested.
- Your assignment will be tested on aviary

Frogger

You will make a text version of the classic video game "frogger." (for more info see google). The rules to frogger are simple – get all the frogs home and you win. Run out of lives and you lose. The frog dies if it lands in water (the home bank is safe), and if it lands on a log, it moves with the log. To save explanation, try running a sample of the program (how yours will act) on aviary by typing the following command at the prompt

`~young/comp3430/frogger`

`wasd` controls and `q` quits.

Synopsis:

- Have a player frog that the player can control using w, a, s, d, around the screen (it cannot go off screen). The player frog must animate.
- There are four rows of logs. You generate logs off screen, and they move on screen. Logs appear at random intervals (within some predefined max and min), and higher rows have faster logs. As in the example, log rows alternate directions. Also, logs must animate graphics in addition to move.
- If the frog reaches one of the home spaces, leave a graphic there and move the frog back to home location.

- The player wins if all frog home spaces are filled.
- The program must always quit properly by joining all threads, until only the main one is left, and freeing memory, and quickly. The final "Done" text must appear on the console as in the provided main and sample. This is to highlight that you did not just use an exit() call. Every time you create a thread, you MUST think: who will join it and when? This is to make scalable, reusable programs. For the same reasons, you must free all memory properly.
- There must be no evident race conditions or deadlock conditions.
- Game mechanics will not be a major concern, e.g., exact log speeds

REQUIRED programming components: As this is a threading assignment, you need the following (to make it hard enough!)

- You will create (at least) the following threads. You may find it useful to make additional ones to help you. Threads must be joinable (default), and cannot be detached (man `pthread_create`).
  - a thread for the player. This handles the player animation.
  - a thread for the keyboard. This requires its own thread so it can block while the rest of the program runs. This is quite tricky because you will have to use the "select" command: it blocks until there is data on a stream, or on a timeout. Here is the problem: if you use getchar, it blocks. While it is blocking, the game may end (you get shot). However, until getchar unblocks that thread cannot end. Using select, you can time out regularly to check if the game is over before blocking again. A code snippit is provided on the website.
  - a thread to re-draw the screen / perform a refresh using the `screen_refresh` command. Do not do refreshes each time you draw, only here.
  - An upkeep thread, that puts updated player lives to screen and does regular cleanup (such as delete memory left behind for dead logs).
  - One thread per each log row that spawns new logs at a regular (but somewhat random) interval.
  - a new thread for **EACH log**. The thread lives until the log goes off screen. Using a single thread for all the logs is not acceptable and won't really work well.
  - the main program (not coded in main.c!) that starts everything and then sleeps using a condition variable (`pthread_cond_wait`). When the game ends this thread is woken up (`pthread_cond_signal`) and cleans up. You must use a condition variable and signal here for the assignment.
- Dynamically managed (malloc/free) linked list: these are very thread un-friendly. You need a linked list for storing the logs, one list for all logs is fine.

Handout:

You have been provided with a `screen.h/screen.c` library that provides text-mode functions for you and a sample program. Be sure to compile with the `–lcurses` flag

to link to the curses (text graphics) library, and the `–pthread` flag to link to the posix-threads library.

- This library is simple to use (see the example). There is a screen buffer off screen. You draw to the buffer using the commands detailed in the library. To have the changes reflected to screen, use the refresh command. To *move*, e.g., the player, you first draw a blank square where the player WAS and then draw the player at its new location.
- This library contains a sleep function that sleeps for the given number of ticks, where a tick is 10ms. Use this in your thread loops, e.g., to set the speed of screen redraw (1 tick?), animation, movements, etc. Most of your threads will be loops that sleep before checking for more work.
- note that you can alter speeds simply by making each sleep longer or shorter.
- **this library is not thread safe!** Create a mutex and make sure to lock it before using ANY screen calls. Not ensuring mutual exclusion will yield funny results.
- You do not need to change this library. In fact, don't do it, it's a bad idea.

Hints:

- **KEEP FINE-GRAINED MUTEXING**, e.g., only lock each log, the player, the screen, the list, etc., as you need it. A single, global lock is missing the point (and extremely inefficient) and will yield very few marks, potentially even negative marks.
- What happens to memory when a joinable thread dies (man pthread_create)? What happens if you are making a lot of threads (bullets) and killing them and do not clean up the memory as you go?
- **Suggested order:** Get the player's frog on-screen and animating, to practice threads (create the screen refresh and player threads first). Then get the keyboard thread working to move the player around. Get a single log working, and then a thread for spawning new ones, before scaling to 4 of them. Make the frog interact with the logs last, as this is by far the hardest part. AT EACH STEP make sure you can quit cleanly and join all threads that you created.
- **HINT:** decide globally on a lock order (as discussed in class) and make sure to stick to it! This avoids deadlocks.
- **HINT:** stop the messy practice of working locally and uploading your files. Learn VI or pico (http://blog.interlinked.org/tutorials/vim_tutorial.html - vi is awesome when you learn tabs and split windows, **see my cheat sheet online**!!!). Working on the target machine dramatically lowers your recompile-test time and gets you done faster.
- **HHIINNTT:: USE GDB**. seriously. I'm not joking. Your program segfaults? run it through gdb and do a backtrace and it tells you where. Your program deadlocks? GDB can let you look at where each thread is stopped so you can figure out who is contending. Printfs? Sure, if you don't mind taking hours to sift through threaded output… **seriously.** gdb can turn an hours-long debug session into 5 minutes as you see which threads are blocking on which mutexes. Don't bother coming to see me for help unless you can show me gdb output.

- This is a big assignment, and some students get overwhelmed in the organization. Here is a list of the files I had in my project.

llist.h/c (generic linked list), gameglobals.h/c, log.h/c, frogger.h/c (main game logic), player.h/c, threadwrappers.h/.c (very simple pthread wrappers that check for errors, to simplify everywhere else).

BONUS: make a submission note to the marker about the bonus, if you do it.

Bonus (10%): Your program creates / destroys threads constantly as things are created and destroyed – this is a huge overhead. Implement a thread pool to re-use threads once they are abandoned. **NOTE:** this does not mean making logs simply move to a new location, this means making a generic library that does jobs, so that when a thread dies the library holds onto the thread instead of killing it, and a new job takes a held one instead of creating a new one. E.g., a log thread may be recycled as a bullet next time! This isn't hard once you see it. You can rely on function pointers (as the work to do). Note that you can have multiple listeners on a condition variable (e.g., waiting for work!).