COMP 3413 Fall 2021 Assignment 3

Due: Monday, Nov 15th, 2021, 11:59 PM

Notes and Style

- must be written in C. No C++
- The prof has provided a working demo in D2l (a3demo executable <- change permissions when copied to your FCS machine with chmod u+x <file>)
- 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**
- You make file should also include a "make run" and "make clean" command that runs your executable and removes all files resulting from compilation
- **Your program must run**. Programs that do not run, or segfault before working at all, **will lose 50% automatically**. Save early and often, and use version control. Submit often, only submit working versions. Partial solutions that are commented out to prevent segfaults should be pointed out to the markers in the submission text box, so markers know to read those comments, otherwise they can skip commented code.
- Your program must compile with –Wall, **with no warnings showing**. You will lose marks for warnings and lose marks for not compiling with -Wall in your makefile
- 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. E.g. cat *.h *.c | listing.txt
- Use common-sense commenting and good style, practices. Provide function headers, comment regions, etc. No particular format or style is required but bad programming practices will result in marks docked (e.g., huge complicated functions, unreadable code, magic numbers, bad variable names, etc).
- 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.
- Test on FCS computers with their version of gcc, make sure your Makefile uses the one you tested.
- Your assignment will be tested on FCS computers.

Attack of the Poisonous Fredericton Caterpillars

You will make a text version of the classic video game "centipede." However, when researching Fredericton, before I moved, I found a funny story about venomous caterpillars in the city, so we'll do caterpillars instead. The rules for the game are simple: kill all the caterpillars before they kill you. The problem is, these are space caterpillars (and you are in a space ship) – when you shoot them, they split at the point they were hit, and now you have two smaller caterpillars!

To save explanation, try running a sample of the program (how yours will act) on AN FCS computer using the precompiled executable on D2L. (a3demo). When you copy it over you will probably need to give it executable permissions (chmod u+x a3demo)

`wasd` controls for up/left/down/right), `space` to fire, and `q` quits.

Synopsis:

- Have a player spaceship that the player can control using w, a, s, d, around a restricted space at the bottom of the screen. The player spaceship must animate.
- Generate caterpillars starting at the top of the screen, at some random interval. They go left to right (or opposite), and when they hit an end, they go to the next row and change direction. Note in the sample how they wrap around (tricky logic). Caterpillars must animate graphics in addition to move.
- If the player presses space, a bullet fires from the player toward the top of the screen. If it goes off the screen, it is destroyed. If it hits a caterpillar, the caterpillar splits into two at that spot, and you now have two caterpillars, with the new one going faster than the old one by some multiple. If one of the two caterpillars is below a minimum size (e.g., 5), it dies. The player has a reasonable maximum fire rate (e.g., once per n ticks).
- The caterpillars randomly shoot from their head at the player.
- When the player is hit, pause the game briefly and kill all the bullets (but leave the caterpillars) to avoid unfair re-spawn deaths. If the player is hit, they lose a life. When no lives are left, the game is over.
- The player wins if there are no caterpillars left on screen.
- The program must 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., left-over dead aliens, exact speeds, etc. In fact, I encourage you to have fun with this. Collision detection must be perfect to help test for race conditions.

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 and other things, e.g., what happens when the player is shot.
  - 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 (man select): 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 thread to re-draw the screen / perform a refresh using the command in the provided console code. Do not do refreshes each time you draw, only here.

  - An upkeep thread, that puts updated score and player lives to screen, checks if there are no enemies left, and does regular cleanup (such as delete memory left behind for dead bullets and caterpillars, maybe).
  - A thread that spawns new caterpillars at a regular (but somewhat random) interval.
  - a new thread for **EACH caterpillar**. The thread lives until the caterpillar is shot and killed. Using a single thread for all the caterpillars is not acceptable. When a caterpillar splits, a new thread is started for the new caterpillar.
  - a new thread for each bullet (yours and caterpillars'!), which lives until the bullet gets consumed into an enemy or you, or until it leaves the screen.
  - a main program (not 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 lists: these are very thread un-friendly. At one list for the bullets, and one list for the caterpillars. No storing in arrays, etc.

In a real game engine, you would likely have fewer threads, e.g., one thread for enemy AI, one thread for input, etc. However, games are highly multithreaded and have a lot of concurrency issues. Making a game engine is a lot like making a simple OS as well: there is a core set of functions that control access to game resources and keeps track of states of other parts. Then, the actual game components (player, enemies, etc) will access those through the engine. This assignment should require more thinking about the engineering and planning of your solution than normal, but it is perhaps algorithmically more straight forward than A2.

Have fun! There's a lot up to you (redesign the spaceship or enemy look, manage difficulty, etc). Your caterpillars may just be simple chains of ascii characters that animate a bit, or they may be more like as seen in the example, larger and stranger looking. Want to add other types? Go for it! There are no game aesthetic or marks for "fun" factor, but it can feel good to make something you're proud of. Really cool submissions will be given a few extra marks (but not enough to make up for not threading correctly).

Handout:

You have been provided with a `console.h/console.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. Console.h has a lot of comments to explain how to use each function. **Definitely use these in your solution**. They are provided so you don't have to worry about "graphics" and drawing to screen here. **You should not modify this code.** Look at console.h comments to understand what each function does.

- 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?), alien 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. This must be done in your game, not in the console.c/h files.

Bootstrap code

I have provided some parts of the code from my solution to help you get started. This includes writing a simple unmoving but animated player (thread unsafe!), a look into how I wrapped some pthread calls for cleaner error checking and more. Parts are missing, and there's no makefile. But used together, they can compile and give you a place to start. You're free to use it as you wish in your final assignment, or ignore it.

If you wish to experience this assignment as I originally intended it, don't use the "A3Bootstrap" code. But if you're struggling to conceptualize how animation or gameplay loops may work, give this a look. There are no penalties or bonuses if you use this/don't use this. It's just a resource to leverage if you're having trouble booting up.

Hints:

- **KEEP FINE-GRAINED MUTEXING**, e.g., only lock each caterpillar, the player, the screen, a 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 ship 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 caterpillar working, and then the thread for spawning new ones. Add bullets last, as they are the hardest part. AT EACH STEP make sure you can quit cleanly and join all threads that you created. Take each step slowly, and think "What is the next smallest piece of work I can do to get closer to a finished solution?" E.g., draw a caterpillar first, then think about spawning them, then think about moving them, then worry about wrapping around, etc.
- **HHIINNTT:: USE GDB**. seriously. I'm not joking. I say this all the time because I mean it. Spending an hour or two reading about gdb and playing around/learning it will save you dozens of hours over this course, and good debugging skills will save you so much time throughout your career. 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 a lot of time 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. If you come asking for help with your code, I'll be asking you what steps you've taken to debug (e.g., do you know where the bug is occurring, in what conditions, etc).
  Of course, if you prefer other debugging tools that's fine. The point is, they're extremely useful tools. Printfs help find simple bugs, but if you're stuck, use gdb.

- 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 (linked list for caterpillars, bullets), gameglobals.h/c, enemy.h/c, bullet.h/c, boolean.h, caterpillars.h/c (main game logic), player.h/c, threadwrappers.h/.c (very simple pthread wrappers that check for errors, to simplify everywhere else.

  None of these files are required, but may help you think through your program structure

BONUS: make a submission note to the marker in D2L 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 bullets 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 caterpillar 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!).

Marking

Assignments that do not have a Makefile, do not compile (with a "make" command), or do not run (e.g., no action, or segfault before running) will immediately receive a 50% penalty. Makefiles will not be marked, just get it working. If segfaults happen part way, we will mark up to the segfault. Partial marks may be given for code that segfaults part way through.

Marks in brackets, with resolution to half mark. To get full marks in a category your solution must be "amazing" (e.g., coded well, general, with good algorithmic solutions). Good solutions that have small quirks or could be improved can lose 0.5-2 marks, and on the opposite ends, solutions that have something kind of working may gain 1 mark. Handing in partially working (not segfaulting) solutions that get the gist of things working is a good way to get a reasonable score on the assignment. Remember to develop incrementally!

[4] C quality – defensive programming, error checking, no magic numbers. Your listing file is used for this. No listing file, you get 0.

[3] Basic player – animates, moves, can shoot bullets

[3] Advanced player – dies when hit (respawns, all bullets disappear). Game over when no lives left.

[3] basic caterpillars – animates, move, wraps properly. Shoots bullets

[3] Advanced caterpillars – breaks when hit, spawning new thread. Dies when too small.

[3] bullets – move appropriately, die appropriately

[3] end conditions: quit, win, and lose

[3] joins all threads and quits cleanly