CSC 360 Summer 2023
Assignment 2

Programming Environment

For this assignment, your code must work in the JupyterHub environment (`https://jhub.csc.uvic.ca`). Even if you already have access to your own Unix system, we recommend you work as much as possible with the JupyterHub environment. Bugs in systems programming tend to be platform-specific and something that works perfectly at home may end up crashing on a different computer-language library configuration.

Individual Work

This assignment is to be completed by each individual student (i.e. no group work). You are encouraged to discuss aspects of the problem with your classmates, however **sharing of code is strictly forbidden**. If you are unsure about what is permitted or have other questions regarding academic integrity, please ask the instructor. Code-similarity tools will be run on submitted programs. Any fragments of code found or generated on the web and is used in your solution must be properly cited (citation in the form of a comment in your code).

Assignment Objectives

1. Write a program `kosmos-sem.c` that solves the **Kosmos-ethynyl-radical** problem with the use of semaphores
2. Write a program `kosmos-mcv.c` that solves the **Kosmos-ethynyl-radical** problem with the use of mutexes and condition variables

Submission

Submit your `kosmos-sem.c` and `kosmos-mcv.c` containing your programming solutions to Brightspace.

The Kosmos-ethynyl-radical problem

You are hired by an interstellar civilization who are seeding the universe with the building blocks needed for one of their major projects. The specific tasks you have agreed to help solve for them is to manage the chemical reaction needed to form **ethynyl radicals**, which is made up of **two carbon atoms** and **one hydrogen atom**.

The interstellar civilization is having trouble getting the carbon and hydrogen atoms to combine correctly due to some serious synchronization problems. They are able to create each atom (i.e. one atom equals one thread), and so the challenge is to get two carbon threads and one hydrogen thread together at the same time, regardless of the number or order of thread creation.

Each carbon atom invokes a function named `c_ready()` when it is ready to react, and each hydrogen atom invokes a function named `h_ready()` when it is ready to react. For this problem, you are to complete the code for `c_ready()` and `h_ready()`, adding whatever code and data structures are needed to solve the problem.

Here are some other notes / requirements for solutions:

- Skeleton files that help you get started are provided
- You are provided with a `makefile` in order to help with building executable versions of your programs (compile by executing the `make` command)
- When running your program, you are asked to provide a random seed (a number so that the pseudorandomness in the program can be replicated) and the total number of atoms to create. The code provided will randomly assign the atoms as carbon or hydrogen (55% chance of being carbon).
- The `c_ready()` and `h_ready()` functions must only delay until there are at least two carbon and one hydrogen atoms ready to combine into the radical
- **You are not permitted** to wait until all threads / atoms are created such that you then match up atoms once you know what exists. This means your solution must work when the next atom / thread created is a hydrogen atom when there already exists two carbon atoms, or when the next atom / thread created is a carbon atom when there already exists one carbon atom and one hydrogen atom.
- When an arriving atom / thread determines that a radical can be made, then that atom / thread must cause the radical to be made to indicate the identities of the carbon atoms and the identity of the hydrogen atom. Below is an example showing the format of the report, where each line indicates the atoms in a radical, and the atom / thread in parentheses was the one that initiated the creation of the radical. For example, in the output below, the 7th radical consists of carbon atoms 10 and 14 and the hydrogen atom 7, and it was the hydrogen atom that triggered the creation of the radical.

```
001: c002 c001 h001 (c002)
002: c004 c009 h002 (h002)
003: c005 c006 h003 (h003)
004: c007 c008 h004 (h004)
005: c003 c013 h005 (h005)
006: c011 c012 h006 (h006)
007: c010 c014 h007 (h007)
008: c016 c015 h008 (c016)
009: c018 c017 h009 (c018)
010: c020 c019 h010 (c020)
```

- Because there are several layers of scheduling involved with POSIX threads and the Linux thread library, you will be unable to reason correctly about the fine details of program behavior by looking at `printf` statements. Therefore, all reports of radical creation are actually written to an internal log and that log is output at the end of the program. You have been provided with the log routines (`logging.c` and `logging.h`), which include the use of a mutex to ensure that there are no race conditions when multiple radicals are formed at the same time. Put a little bit more dramatically, debugging `printf` statements may lead you astray as you may try to interpret them to indicate a certain order of instruction operation (such as "before this point" or "after this point"), but the truth will often be very different.

Part 1: `kosmos-sem.c`

For this part you are to solve the problem using only POSIX semaphores. In this part, you are not permitted to use mutexes or condition variables or any other items from the pthreads library.

Code that randomly creates each atom thread has been provided for you. Please read the file carefully as it clearly indicates parts of the program which you are permitted and not permitted to modify.

Part 1: `kosmos-mcv.c`

For this part you are to solve the problem using only POSIX mutexes and condition variables. In this part, you are not permitted to use semaphores or any other items from the pthreads library.

Code that randomly creates each atom thread has been provided for you. Please read the file carefully as it clearly indicates parts of the program which you are permitted and not permitted to modify.

Note on thread scheduling

You will notice, when you write a solution, that the pthread scheduling of threads is a little unpredictable. Some atoms / threads created later in the program will appear in radicals before atoms that were created earlier. You may even find that some atoms created earlier do not even appear in the radical. This is fine. In other words, your solutions are not required to have atoms combine in a strict numeric order.