CS-4710 & CS-1217 Operating Systems, Spring 2025

Test Code:	CS4710_A2 / CS1217_A3
Assignment #	3
Topic:	Process Synchronization and Critical Section Problem
Due Date:	Sunday, April 27, 2025
Total Marks:	40
Weightage:	-

Submission Instructions:

- You have to write a C program for **Part 1** of the assignment and answer the theoretical questions in **Part 2**. Submit a compressed file containing the program for Part 1 and a PDF for Part 2, with the naming format **Firstname_CS4710_A2** or **Firstname_CS1217_A3** (for 4-credit students).
- Comment your program and provide an explanation of your implementation. It will be used to assess your understanding.

Part 1: Programming [6+6+8=20]

You have been hired by a company to do climate modelling of oceans. The inner loop of the program matches atoms of different types as they form molecules. In an excessive reliance on threads, each atom is represented by a thread.

1. Your task is to write code to form water out of two hydrogen threads and one oxygen thread (H_2O). You are to write the two procedures: `HArrives()` and `OArrives()`. A water molecule forms when two H threads are present and one O thread; otherwise, the atoms must wait. Once all three are present, one of the threads calls `MakeWater()`, and only then, all three depart.
2. The company wants to extend its work to handle cloud modelling. Your task is to write code to form ozone out of three oxygen threads (O_3). Each of the threads calls `OArrives()`, and when three are present, one calls `MakeOzone()`, and only then, all three depart.
3. Extending the product line into beer production, your task is to write code to form alcohol (C_2H_6O) out of two carbon atoms, six hydrogens, and one oxygen. You must implement procedures `CArrives()`, `HArrives()`, and `OArrives()` accordingly. Once a group of 2 carbon, 6 hydrogen, and 1 oxygen atoms is ready, one of them must call `MakeAlcohol()`, and only then do all the participating atoms depart.

You must implement synchronization logic that ensures molecules form only when the correct number and type of atoms are present. You should use mutual exclusion locks (mutexes) and Mesa-style condition variables. In Mesa-style synchronization, threads must always re-check the condition in a loop when woken up, as another thread may have taken the resource they are waiting for. You must avoid busy-waiting (spinning) and ensure that no thread waits unnecessarily if a valid group of atoms can form a molecule. An atom that arrives after a molecule has already formed must wait until the next valid group is ready. One and only one thread per group must call the molecule-forming function, such as `MakeWater()`, before any of the atoms in that group proceed. Threads not used in the current molecule formation must remain blocked until enough atoms are present to form the next group.

Part 2: Questions [7+5+8=20]

1. Construct your own example of a shared memory race condition (different from your textbook / slides / internet resources).

2. Consider the following code snippet running on a modern Linux operating system. Assume that there are no other interfering processes in the system. Note that the executable "good_long_executable" runs for 100 seconds, prints the line "Hello from good executable" to screen, and terminates. On the other hand, the file "bad executable" does not exist and will cause the exec system call to fail.

```
1: int ret1 = fork();
2: if ret1 == 0 then
3:    printf("Child 1 started\n");
4:    exec("good_long_executable");
5:    printf("Chīld 1 finished\n");
6: else
7:    int ret2 = fork();
8:    if ret2 == 0 then
9:       sleep(10);
10:      printf("Child 2 started\n");
11:      exec("bad_executable");
12:      printf("Child 2 finished\n");
13:   else
14:      wait();
15:      printf("Child reaped\n");
16:      wait();
17:      printf("Parent finished\n");
18:   end if
19: end if
```

Write down the output of the above program with justification.

3. The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P_0 and P_1, share the following variables:

```
boolean flag[2]; /* initially false */
int turn;
```

The structure of process P_i ($i == 0$ or 1) is shown in the figure below. The other process is P_j ($j == 1$ or 0). Prove that the algorithm satisfies all three requirements for the critical-section problem. (Mutual exclusion, progress, and bounded waiting)

```
while (true) {
    flag[i] = true;
    while (flag[j]) {
        if (turn == j) {
            flag[i] = false;
            while (turn == j)
                ; /* do nothing */
            flag[i] = true;
        }
    }
    /* critical section */
    turn = j;
    flag[i] = false;
    /* remainder section */
}
```