IT308: Synchronization – Part II

The objective of this lab is to gain more familiarity with pthreads synchronization primitives.

Checkoffs:

- Finish Task 1 during your assigned lab slot, and show your work to a TA.
- After getting checked off on Task 1, you can continue working on Task 2. This will be due for checkoff in your next assigned lab slot.

Task 1: Readers-Writers

A classical concurrency problem is the **readers-writers** problem. In this problem, a set of threads access a shared variable. Some of these threads, the **readers**, only read the variable, whereas the remaining threads, the **writers**, only write to the variable.

Since reading operations do not interfere with one another, we can increase concurrency, by allowing simultaneous reading operations. On the other hand, write operations interfere with both reading and writing operations, and therefore a write operation must be executed in mutual exclusion. i.e., when a writer thread accesses to the shared variable, no other thread, either reader or writer, should access the shared variable. (Check Section 31.5 of the OSTEP book for a more detailed discussion of this problem.)

In this problem, you should solve a variant of the readers-writers using mutexes and condition variables. In particular, there are two global integer variables, **m** and **n**, which should be initialized to 0 by the main thread. The main thread creates 2 writer threads and 2 reader threads, and waits for them to exit. After termination of the 4 threads, the main thread should print the final value of the two variables.

Each of the writer threads shall execute a loop 50 million times. In each iteration, a writer shall increment by one each of the two variables, so that their value when the program terminates should be 100 millions.

Each of the reader threads shall execute a loop also 50 million times. In each iteration, a reader shall read the 2 variables and print them after every 1 million iterations. The threads should synchronize in such a way that the values of the two variables printed are identical.

When there is at least one reader accessing the variables, no writer should change the value of these variables, however it is possible for another reader to read them. **So the critical question is:**

If a reader is reading the variables and a writer is waiting to write, should we allow another reader to read the shared variable, further delaying the writer?

The alternatives are clear:

Yes

In this case, reading concurrency is increased, but writers may be forced to wait forever, if a reader arrives before another that has started the reading is finished. i.e. there may be **starvation** (of the writers).

No

In this case, the solution is fairer, but it may reduce concurrency.

Try to solve this problem using both strategies (one at a time). Are the observed results as you were expecting?

Task 2: Animal clinic

An animal clinic cares for some ferocious but endangered animals. The animals are allowed to wander between an enclosed building and an open yard. As part of their recovery efforts, the caretakers need to enter the yard from time to time. However the animals are so ferocious that caretakers should only enter the yard when it is free of animals. Furthermore, because we want to preserve the wild nature of the animals, only one caretaker can be allowed in the yard at one time.

You are tasked with writing a software system that will help manage this animal yard by blocking animals from entering when a caretaker is present, blocking a caretaker from entering until all animals have left, and ensuring only one caretaker is in the yard at one time.

Write a program

```
clinic numAnimals numCaretakers
```

in which the parent spawns *numAnimals* threads to represent the animals, and *numCaretakers* threads to represent the caretakers. Though an animal thread will of course look different from a caretaker thread, each thread should run an infinite loop like the following:

```
while (1)
{
    // Enter the yard (or attempt and wait, subject to
 constraints)
    // Sleep for a random time
    // Exit the yard
    // Sleep for a random time
}
```

You may use any combination of mutex, condition, and semaphore variables that you wish. Implement your program in file clinic.c.

You can sleep a random amount of time with something like the following:

```
usleep( random() % 2e6 ); // Sleep up to two seconds
```

It is up to you how long to make the bounds for each of the four sleep periods.

For instance, the output might look something like this.

```
Creating animal 0
Creating animal 1
Creating animal 2
Creating animal 3
Creating animal 4
Creating caretaker 0
Creating caretaker 1
Animal 4 queues
Creating caretaker 2
Animal 4 enters (1)
Animal 0 queues
Animal 0 enters (2)
Caretaker 2 queues
Animal 2 queues
Animal 2 enters (3)
Caretaker 0 queues
Animal 3 queues
Animal 3 enters (4)
Animal 1 queues
Animal 1 enters (5)
Caretaker 1 queues
Animal 4 exits (4)
Animal 4 queues
Animal 4 enters (5)
Animal 2 exits (4)
Animal 0 exits (3)
Animal 1 exits (2)
Animal 3 exits (1)
Animal 4 exits (0)
Caretaker 2 enters
Animal 4 queues
Animal 2 queues
Animal 3 queues
Animal 0 queues
Caretaker 2 exits
Caretaker 0 enters
Caretaker 2 queues
Animal 1 queues
Caretaker 0 exits
Caretaker 1 enters
Caretaker 1 exits
Animal 4 enters (1)
Animal 2 enters (2)
Animal 3 enters (3)
Animal 0 enters (4)
Animal 1 enters (5)
Animal 4 exits (4)
Animal 3 exits (3)
Animal 0 exits (2)
Animal 4 queues
Animal 4 enters (3)
Animal 1 exits (2)
Animal 2 exits (1)
Animal 2 queues
Animal 2 enters (2)
Animal 2 exits (1)
Animal 4 exits (0)
Caretaker 2 enters
Caretaker 2 exits
Animal 1 queues
Animal 1 enters (1)
Animal 1 exits (0)
Animal 0 queues
Animal 0 enters (1)
Animal 3 queues
Animal 3 enters (2)
Animal 1 queues
Animal 1 enters (3)
Animal 1 exits (2)
Animal 1 queues

...

and so on
```