Assignment 5

Due Apr 27 by 11:59pm **Points** 18 **Submitting** a file upload

Instructions

Submit a gzipped tar file containing your submitted files. They should be called problem_1.c, problem_2.c, and problem_3.c. Also submit a Makefile that builds from your source files three executables called problem_1, problem_2, and problem_3. Your tarball may include extra header files if you like.

You may use functions from the C library but otherwise (with one exception, see below) you must write all your own code. The return code of any library function that might fail must be checked (including pthread_create, but not including printf()).

Your programs must compile correctly on CADE machines using this command:

```
/usr/bin/gcc -pthread -O2 -fmessage-length=0 -pedantic-errors -std=gnu99 -Werror -Wall -Wextra -Wwrite-strings -W
init-self -Wcast-align -Wcast-qual -Wpointer-arith -Wstrict-aliasing -Wformat=2 -Wmissing-include-dirs -Wno-unuse
d-parameter -Wshadow -Wuninitialized -Wold-style-definition problem_n.c -o problem_n
```

Problem 1

You are writing a simulator for a pet motel -- where people's pets can stay while they go on vacation. Cats, dogs, and birds are all supported. The problem is that these pets need to be let out of their cages to get exercise but there is just one playground.

Cats and dogs must not be permitted to play together, nor cats and birds. Any number of cats can play together, any number of dogs may play together, and any number of birds may play together. Also, dogs and birds can play together in any numbers.

Each animal is represented by a thread. Implement a monitor with 6 methods: cat_enter(), cat_exit(), dog_enter(), dog_exit(), bird_enter(), bird_exit(). The monitor should have a condition variable for each kind of animal and (of course) a single mutex. Animals should only be forced to wait when their entrance would break a rule. Sleeping and busy waiting are absolutely not permitted in your animal simulation code (it's fine for the main thread to sleep for 10 seconds before telling the animal threads to quit).

You are not required to implement any higher-level fairness policy. Therefore, it is possible that one kind of animal will not get to play much (or at all) especially when large numbers of animals are being simulated. If you want to solve this, feel free, but include comments explaining what your code is doing.

Your problem 1 executable should take three command line arguments: the number of cats, dogs, and birds. Your main function should create the proper number of threads and then they should repeatedly attempt to enter the playground until 10 seconds have passed, at which point each thread should exit. The main

function should join all of the created threads, then print out how many times the cats, dogs, and birds were able to enter the playground.

Between 0 and 99 of each kind of animal must be supported. You must perform error checking and exit with a friendly error message if the number of command line arguments is incorrect or if an argument isn't a number of is a number that is out of range.

When any animal is allowed into the playground, it should call a play() function -- this function uses the assert() macro to enforce the correctness criteria for this problem: that there are never cats+dogs or cats+birds on the playground. Here's the code that I used, which repeats the check several times for good measure:

```
void play(void) {
  for (int i=0; i<10; i++) {
    assert(cats >= 0 && cats <= n_cats);
    assert(dogs >= 0 && dogs <= n_dogs);
    assert(birds >= 0 && birds <= n_birds);
    assert(cats == 0 || dogs == 0);
    assert(cats == 0 || birds == 0);
   }
}
```

Example run:

```
$ ./problem_1 2 2 2
```

```
cat play = 5726, dog play = 274161, bird play = 274152
```

Problem 2

Write a single-threaded program that takes a single command-line argument: the name of a directory. For each file in that directory, your program should print the name of the file and also its checksum. The file name and checksum should be separated by a space and followed by a newline. The checksum should be printed in uppercase hex including any leading zeroes necessary to pad out to 8 hex digits. Your checksum function must be **CRC32** **(http://www.opensource.apple.com/source/xnu/xnu-1456.1.26/bsd/libkern/crc32.c)** (feel free to reuse the CRC32 implementation found at this link). The initial value of the CRC state variable should be 0. Directories inside the target directory should not be printed: just skip over them. For any file that cannot be opened for reading, print "ACCESS ERROR" where you would have printed the file's checksum. If the directory specified on the command line does not exist or cannot be read, exit with a friendly error message.

By default, readdir() won't give you files in any particular order. You must print them in sorted in the order determined by running strcmp() on file names.

Although of course you may not look at other students code, you may test your code against other students' code by exchanging executables with your friends or acquaintances in the class. Any two correct solutions

for this problem should print the same thing when pointed at the same target directory.

The basis for your directory traversal should be the C library functions opendir(), readdir(), and closedir(). You must not assume any particular maximum number of files in a directory. Make sure your problem 2 solution works regardless of whether the provided directory name contains a trailing '/' character.

Problem 3

Write a multi-threaded version of your problem_2 implementation. The directory walking code should remain single-threaded, but the checksum computations for files should run in multiple threads. Your problem_3 implementation therefore takes a second (mandatory) command line argument: the number of threads to spawn. Numbers of threads between 1 and 99 must be supported and error checking should be performed on the command line argument, aborting the program on non-numeric or out-of-bounds argument.

You must implement the "worker thread" pattern where the specified number of threads are created and then perform work until there is no work left to do. In other words, if the program is run with "2" as the number of threads, then only a total of 2 threads can be created to do all the work. Communication between the main thread and its workers should be through a monitor that you design and implement. Your code should keep all of the worker threads busy as long as there is work to do. This means that you should not assume that workers finish jobs in the order in which these jobs were handed out.

The output of a correct problem_3 implementation, when pointed at any given directory, will be the same as the output of a correct problem_2 implementation. In other words, the threaded execution must not change the output.

Some Rubric (1)		
Criteria	**Ratings**	**Pts**
Problem 1		6.0 pts
Problem 2		6.0 pts
Problem 3		6.0 pts
		Total Points: 18.0