CSSE 332 – Operating Systems
Rose-Hulman Institute of Technology
Computer Science and Software Engineering Department

Exam 1 — Computer Part

Name:______________________________________ Section:______ CM:_________

Instructions: This part of the exam is open notes (same as paper part) and open computer.

- You must disable all chat tools (IM, ICQ, IRC, Lync, etc.) before the exam starts.
- **Any communication with anyone other than an exam proctor during the exam could result in a failing grade for the course**.
- You may refer to programs you have written for the course-both those assigned and those you have written for practice. You may not, however, refer to programs written by others except those provided in your textbook or by the course staff.
- Regarding materials on the web, you may only use the course web site and pages linked directly from it. Of course, search engine use is not allowed.
- Write all answers on these pages.
- Submit all code and support files to your svn repository.
- Read the entire examination before starting, then budget your time.

	Points available	Your marks
1	11	
2	11	
3	10	
4	8	
5	10	
6	11	
7	9	
Total	70	

Problem 1 (11 points)

Look at the example code in ThreadCommander.c. This is a simple program that lets you start (and eventually stop) threads. The first feature you will add is the ability to start threads.

```
Enter 'c <THREAD_NAME>' to create a new thread
Enter 'k' to stop most recently started thread
c foo
Create foo
Hello from thread!
Hello from thread!
c bar
Create bar
Hello from thread!
Hello from thread!
Hello from thread!
Hello from thread!
```

Each started thread should print it's message, then sleep for one second, then print again, etc. It's difficult to tell from above, but if you have 2 threads running the messages should print twice as frequently.

You can make the assumption there will be no more than 100 threads.

Problem 2 (11 points)

Now modify the code so that each thread prints its name along with the message. This will require your thread function to take some parameters.

Each thread name can be up to 100 characters long.

At this stage, you can feel free to pre-allocate 100 thread parameter structures, each with their own 100 character string. In the final step of the problem, I ask you to change this to use malloc.

```
Enter 'c <THREAD_NAME>' to create a new thread
Enter 'k' to stop most recently started thread
c foo
Create foo
Hello from thread foo!
Hello from thread foo!
c bar
Create bar
Hello from thread bar!
Hello from thread foo!
Hello from thread bar!
Hello from thread foo!
```

Problem 3 (10 points)

Now add the "k" command which stops threads. Typing k should stop the most recent thread. Typing k again should stop the second most recent, etc. In essence, there is a stack of threads and k stops the thread on the top of the stack. When no threads are running, typing k should display an error message.

Once a thread had stopped, you should display a message "thread NAME stopped". This print should be in your main thread, after successfully executing a join.

To stop the thread safely, have a boolean parameter passed to the thread like "threadOughtToKeepRunning". Then when the thread needs to stop, have your main thread set this value to false. That should terminate the while loop that keeps the thread running forever.

```
Enter 'c <THREAD_NAME>' to create a new thread
Enter 'k' to stop most recently started thread
c foo
Create foo
Hello from thread foo!
Hello from thread foo!
c bar
Create bar
Hello from thread bar!
Hello from thread foo!
k
thread bar stopped
Hello from thread foo!
Hello from thread foo!
c qqq
Hello from thread qqq!
Hello from thread foo!
k
thread qqq stopped
Hello from thread foo!
k
thread foo stopped
k
No threads running
```

Problem 4 (9 points)

Now we want to improve memory efficiency by using malloc. Rather than pre-allocating all your 100 character name arrays, use malloc to allocate memory for the structures when they are created. Then use free to deallocate them when the thread is stopped. You can feel free to continue to preallocate an array of 100 pointers or thread ids - just not the memory for the thread specific data itself. (though it is possible to avoid that overhead by building a little linked list with your thread data structure)

Problem 5 (10 points)

Now onto the second example. Take a look at the code in UnitTestingFork. There are 2 c files, 1 header file. You can compile the code using the provided Makefile, which produces an excutable called "runTests". If you run runTests it will run one unit test successfully and then segmentation fault on the second test.

But the first problem is the Makefile. Right now we recompile the entire codebase everything there is a change to any file. What we would like to do is produce .o files for each .c file, ensuring we only recompile what is necessary each time a file is updated. Recall that the –c directive will make the compilier compile without linking (i.e. produce .o files).

Modify the Makefile make it construct .o files. It should then use those .o files to compile the final executable. When a specific source file (.h or .c) is updated, your Makefile should only rebuild what is necessary to produce a new version of the executable, reusing the .o files of parts of the system that have not changed.

Problem 6 (11 points)

Ok, now to the code itself. The code exhibits a very common problem with unit testing in C — an error in one test can crash the entire testing program. Our goal is not to fix the segmentation fault, but rather to prevent a segmentation fault from stopping later tests from running.

To do this, we will modify UnitTestingFork.c to fork before we run each unit test function. In the child we will run the test. In the parent, we will wait for the test to complete before continuing. Remember: the goal here is not to run the tests in parallel, just protect our test running code from any bugs that might be discovered in the code we test.

If you do this properly, your tests should look like this:

```
Starting unit test 1 of 4
assert failed.  Expected 10 got 9
assert passed.
Starting unit test 2 of 4
Starting unit test 3 of 4
Starting unit test 4 of 4
assert passed.
```

Note that if we wanted to get fancy, we could look at the return code of the fork to figure out a child crashed and warn the user. But that's beyond the scope of this exam.

As part of this process, you may end up with some fairly ugly duplicated code in your main (feel free to do some cutting and pasting). We'll solve that in our next step.

Problem 7 (9 points)

Writing that repetitive fork code sure is ugly! If only we could somehow remove that duplication. Oh wait, we can remove that duplication — with function pointers. Write a function that takes a function pointer to a test function and runs that given function safely in a fork. When you're finished, your code in main should look like this:

```
runTest(runUnitTest1);
runTest(runUnitTest2);
runTest(runUnitTest3);
runTest(runUnitTest4);
```

Hint: to end your child processes after their test case is run, use exit(0) which can end your process even in a function.

Have you committed your files and changes to svn?