POSIX Threads Programming Exercises

Exercise 1

1. **List the contents of your `pthreads` subdirectory**

   You should notice a number of files as shown in the table below.

File Name	Description
`arrayloops.c` `arrayloops.f`	Data decomposition by loop distribution. Fortran example only works under IBM AIX: see comments in source code for compilation instructions.
`condvar.c`	Condition variable example file. Similar to what was shown in the tutorial
`detached.c`	Demonstrates how to explicitly create pthreads in a detached state.
`dotprod_mutex.c` `dotprod_serial.c`	Mutex variable example using a dot product program. Both a serial and pthreads version of the code are available.
`hello.c`	Simple "Hello World" example
`hello32.c`	"Hello World" pthreads program demonstrating thread scheduling behavior.
`hello_arg1.c`	One correct way of passing the pthread_create() argument.
`hello_arg2.c`	Another correct method of passing the pthread_create() argument, this time using a structure to pass multiple arguments.
`join.c`	Demonstrates how to explicitly create pthreads in a joinable state for portability purposes. Also shows how to use the pthread_exit status parameter.

`mpithreads_serial.c`
`mpithreads_threads.c`
`mpithreads_mpi.c`
`mpithreads_both.c`
`mpithreads.makefile`
	A "series" of programs which demonstrate the progression for a serial dot product code to a hybrid MPI/Pthreads implementation. Files include the serial version, Pthreads version, MPI version, hybrid version and a makefile.

Bug Code	Bug Behavior	Hints/Notes
bug1.c bug1fix.c	Hangs	
bug2.c bug2fix.c	Seg fault/coredump	
bug3.c	Wrong answers	
bug4.c bug4fix.c	Hangs (usually)	
bug5.c	Threads die and never get to do their work	
bug6.c bug6fix.c	Wrong answer - run it several times to prove this	

2. **Create, compile and run a Pthreads "Hello world" program**
    1. Using your favorite text editor (vi/vim, emacs, nedit, gedit, nano...) open a new file - call it whatever you'd like.
    2. Create a simple Pthreads program that does the following:
        - Includes the `pthread.h` header file
        - Main program creates several threads, each of which executes a "print hello" thread routine. The argument passed to that routine is their thread ID.
        - The thread's "print hello" routine accepts the thread ID argument and prints "hello world from thread #". Then it calls `pthread_exit` to finish.
        - Main program calls `pthread_exit` as the last thing it does

        If you need help, see the provided file.

    3. Using your choice of compiler (see above section 4), compile your hello world Pthreads program. This may take several attempts if there are any code errors. For example:

```
gcc -pthread -o hello myhello.c
```

    When you get a clean compile, proceed.

    4. Run your `hello` executable and notice its output. Is it what you expected? As a comparison, you can compile and run the provided example program.
    5. **Notes:**
        - For the remainder of this exercise, you can use the compiler command of your choice unless indicated otherwise.
        - Compilers will differ in which warnings they issue, but all can be ignored for this exercise. Errors are different, of course.

3. **Thread Scheduling**
    1. Review the example code `hello32.c`. Note that it will create 32 threads. A `sleep();` statement has been introduced to help insure that all threads will be in existence at the same time. Also, each thread performs actual work to demonstrate how the OS scheduler behavior determines the order of thread completion.
    2. Compile and run the program. Notice the order in which thread output is displayed. Is it ever in the same order? How is this explained?
4. **Argument Passing**
    1. Review the `hello_arg1.c` and `hello_arg2.c` example codes. Notice how the single argument is passed and how to pass multiple arguments through a structure.
    2. Compile and run both programs, and observe output.
    3. Now review, compile and run the `bug3.c` program. What's wrong? How would you fix it? See the explanation in the bug programs table above.
5. **Thread Exiting**
    1. Review, compile (for gcc include the `-lm` flag) and run the `bug5.c` program.
    2. What happens? Why? How would you fix it?
    3. See the explanation in the bug programs table above.
6. **Thread Joining**
    1. Review, compile (for gcc include the `-lm` flag) and run the `join.c` program.
    2. Modify the program so that threads send back a different return code - you pick. Compile and run. Did it work?
    3. For comparison, review, compile (for gcc include the `-lm` flag) and run the `detached.c` example code.
    4. Observe the behavior and note there is no "join" in this example.

7. **Stack Management**
    1. Review, compile and run the `bug2.c` program.
    2. What happens? Why? How would you fix it?
    3. See the explanation in the bug programs table above. Also review and try out `bug2fix.c` program.

This completes Exercise 1

Exercise 2

1. **Mutexes**
    1. Review, compile and run the `dotprod_serial.c` program. As its name implies, it is serial - no threads are created.
    2. Now review, compile and run the `dotprod_mutex.c` program. This version of the dotprod program uses threads and requires a mutex to protect the global sum as each thread updates it with their partial sums.
    3. Execute the `dotprod_mutex` program several times and notice that the order in which threads update the global sum varies.
    4. Review, compile and run the `bug6.c` program.
    5. Run it several times and notice what the global sum is each time? See if you can figure out why and fix it. The explanation is provided in the bug examples table above, and an example solution is provided by the `bug6fix.c` program.
    6. The `arrayloops.c` program is another example of using a mutex to protect updates to a global sum. Feel free to review, compile and run this example code as well.

2. **Condition Variables**
    1. Review, compile and run the `condvar.c` program. This example is essentially the same as the shown in the tutorial. Observe the output of the three threads.
    2. Now, review, compile and run the `bug1.c` program. Observe the output of the five threads. What happens? See if you can

determine why and fix the problem. The explanation is provided in the bug examples table above, and an example solution is provided by the **`bug1fix.c`** program.

3. The **`bug4.c`** program is yet another example of what can go wrong when using condition variables. Review, compile (for gcc include the **`-lm`** flag) and run the code. Observe the output and then see if you can fix the problem. The explanation is provided in the bug examples table above, and an example solution is provided by the **`bug4fix.c`** program.
4. Run each of the codes and observe their output:

This completes the exercise.