Lab – Shared Memory Parallelism using POSIX PThread

For this lab, you shall write, review, compile and run C/C++ code with the POSIX PThread API. Hence, you must use GCC on Linux machine, e.g. Ubuntu, ideally running on a virtual machine.

In some steps, you are going to review, compile and run a set of predefined C program files that are available for you through “src.zip”. Once unzipped, you will find the files described in Table 1.

Skip reading the complete table for now and jump to the first step given on the next page.

Table 1 List of C source files

File Name	Description	
arrayloops.c	Data decomposition by loop distribution.	
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.	
Bug Code	**Bug Behavior**	**Hints/Notes**
bug3.c	Wrong answers	bug3.c shows an unsafe way to pass the argument parameter in the pthread_create routine. **It passes the address of t**, rather than the value. As a result, each thread looks at the same location for their supposed unique instance of t. By the times the threads start, the thread creation loop is done and t is equal to 8 - usually. This means each thread gets an invalid and non-unique value for t. Compare with hello_arg1.c for one correct way of passing t.
bug5.c	Threads die and never get to do their work	bug5.c is simply missing a pthread_exit call at the end of main. As a result, when

		main finishes, it goes away as do all of its threads. Adding a pthread_exit routine at the end of main will solve the problem.
bug6.c **bug6fix.c**	Wrong answer - run it several times to prove this	bug6.c shows how shared memory can be misused by multiple threads. All are attempting to update the global sum at the same time without any synchronization mechanism. As a result, the answer will almost always vary and be wrong. Using a mutex variable around the update of the global sum variable will solve this problem. The solution bug6fix.c shows one way to do this, but **dotprod_mutex.c** shows a more efficient way to accomplish this.

Step 1. 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 "**`myhello.c`**", or 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 "**`hello.c`**" file.

3. Using the GCC compiler, compile your hello world Pthreads program. This may take several attempts if there are any code errors. For example:

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

    When you get a clean compile, proceed.

4. Run your **`myhello`** executable by calling **`./myhello`** and notice its output. Is it what you expected? As a comparison, you can compile and run the provided "**`hello.c`**" example program.

    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.

Step 2. 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 (for gcc include the **`-lm`** flag[1] at the end of the command line) and run the program. Notice the order in which thread output is displayed. Is it ever in the same order? How is this explained?

Step 3. 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.

Step 4. 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.

Step 5. 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.

Step 6. Mutexes

1. Review, compile and run the **`dotprod_serial.c`** program. As its name implies, it is serial - no threads are created.

[1] The flag -lm asks the GCC compiler to include the math library on runtime.

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.

Step 7. Programming Task

Implement a program that sums up a range of numbers in parallel. The general algorithmic problem is called “parallel reduction”.

Input

Your application must be named “**`parsum`**” and accept three parameters:

1. The number of threads to use,
2. the start index, and
3. the end index (64 bits numbers) of the range to compute.

For example, the command line:

```
./parsum 30 1 10000000000
```

has to result in a parallel summation of the numbers 1, 2, ..., 10.000.000.000, based on 30 threads running in parallel.

Output

Your program must output the computed sum.

You may also opt to produce an output file with the name “output.txt” in the same directory. This file must contain only the computed sum.

Validation

The solution is considered correct if a true parallelized computation takes place (no guess please), if the solution scales based on the number of threads, and if the application produces correct results for all inputs. You may evaluate your solution with different thread counts / summation ranges.