

## 2.

Review the terminology and describe clearly what it means to write "thread safe" functions that are "re-entrant". There are generally three main ways to do this: 1) pure functions that use only stack and have no global memory, 2) functions which use thread indexed global data, and 3) functions which use shared memory global data, but synchronize access to it using a MUTEX semaphore critical section wrapper. Describe each method and how you would code it and how it would impact real-time threads/tasks. Now, using a MUTEX, provide an example using RT-Linux Pthreads that does a thread safe update of a complex state with a timestamp ([pthread mutex lock](#)). Your code should include two threads and one should update a timespec structure contained in a structure that includes a double precision attitude state of {X,Y,Z acceleration and Roll, Pitch, Yaw rates at Sample\_Time} (just make up values for the navigational state and see [http://linux.die.net/man/3/clock\\_gettime](http://linux.die.net/man/3/clock_gettime) for how to get a precision timestamp). The second thread should read the times-stamped state without the possibility of data corruption (partial update).

### ANSWER

#### Problem with Shared Functions

If a shared function updates global data, it can be a problem since two or more threads accessing it can corrupt the data. In such a condition, it is extremely important to make the thread safe such that it is not interfered with by the other thread, while updating common global data.

#### Thread-Safe Functions

Thread-safe functions are those that access shared data in a safe manner. This implies that, multiple calls to a thread-safe function, at the same time, will not have any adverse effect on the variables or components handled by the function.

#### Re-entrant Functions

Re-entrant functions are those that can be interrupted or pre-empted, then called again in a safe manner, without finishing execution of the original call.

Hence, essentially, if a shared function exists such that it can be called by more than one thread and if those threads are concurrently active, the function must provide re-entrant support so that it is thread-safe.

#### Three ways to write Thread-safe functions that are re-entrant

- The first way of creating thread-safe functions is to create pure functions that only use stack variables and have no global data or dynamic memory. This is the easiest way to ensure non-corruption of data, and this is achieved due to the absence of any shared data between two or more threads, each function having its own local data that can be modified without affecting others. Each thread working on the function will have a local

copy of the variable unlike a global declaration which would result in a race condition. This method can hence be implemented by writing simple functions, each with their own local data. All data that every function needs must be passed to it, and the functions should return all processed data. It is necessary, though, to make sure that the threads must be executed in order, with proper priority assigned to each thread, to ensure no issues or problems with data forwarding. While this method is easy, it is cumbersome because it, more often than not, leads to over-sized structures getting passed in to functions unnecessarily. Copying all data every time a function call is made is extremely inefficient. Also, it does not make practical sense, since in most practical cases, sharing of data between real-time threads is required.

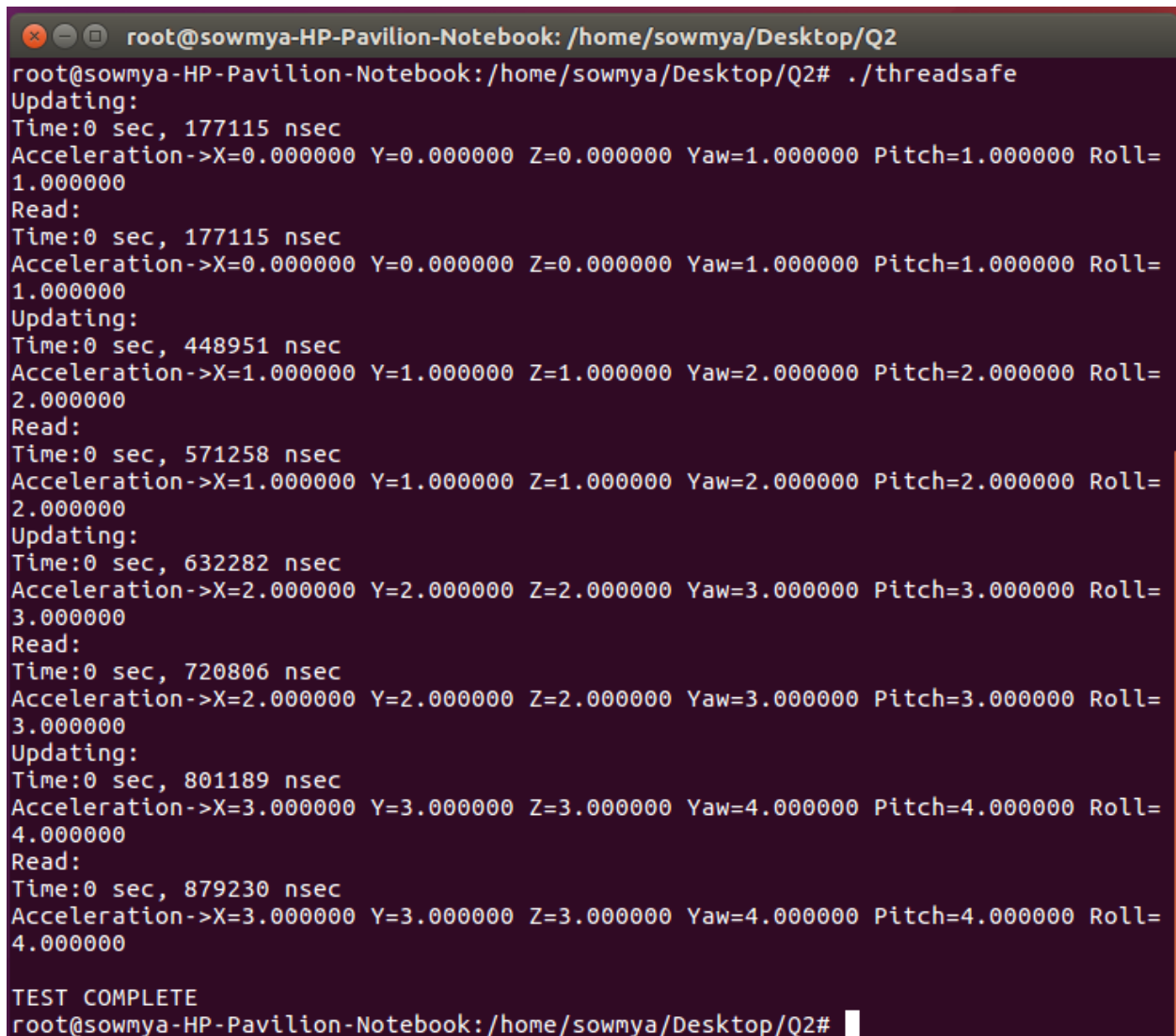
- The second method of implementation of a thread-safe function, while overcoming the limitations of the first, is to use functions which use thread-indexed global data. This ensures that the thread remains safe, while having global variables to share data. Since data is thread-indexed, either many copies of the same memory will have to be made, or proper locking mechanisms will have to be implemented to prevent shared memory errors. Thread-indexed global data defines a mechanism wherein the code can retrieve thread-specific data stored in a global database which can be accessed with the use of a shared key that is known to all threads involved in accessing the function. When a thread requests data using the key, it will get an address back, which can be used to securely access the data. Thus, this method enables threads to communicate in real-time systems. The problem, however, is the synchronization between threads. If the threads need to share data, a shared memory has to be used with mutexes and semaphores. Priorities to each thread must be assigned with care. Transferring data from global memory for computation purposes will result in lesser throughput due to high latency.
- The final method is to use functions which use shared memory global data but synchronize access to it using a MUTEX semaphore critical section wrapper. It is essentially a locking mechanism using semaphores and MUTEXes. This is a good way to code any real-time threads that require data to be shared among them without corruption of it. The idea is to allow only one thread to work on the shared memory at a time. It can be done by implementing a type of counter, which, when zero, makes a thread wait., and when one or more, makes the thread execute on the shared memory. A MUTEX can be created globally and locked by threads in the critical sections where they update or read global data. This ensures that a thread is not interrupted by another thread during critical update and/or read. In the function, it is to be ensured that update and read are safe from one another. The function `mutex_init()` can be used to initialize a MUTEX in the main thread. The corresponding threads can then use `mutex_lock()` and `mutex_unlock()` functions around the critical sections to make them safe. The drawbacks of this

implementation method include unbounded priority inversion or unbounded blocking, which can generally be overcome/fixed with proper coding techniques.

### Implementation using MUTEX and RT-Linux Pthreads

This is a design and implementation of a thread-safe application that modifies contents of a structure with timestamps and reads the changes without corrupting global data.

A code using MUTEX and RT-Linux Pthread APIs was written to generate data and timestamp in one thread and print it out safely in another. It is built using a makefile and executed to obtain the output as shown in the following screenshot.



```
root@sowmya-HP-Pavilion-Notebook: /home/sowmya/Desktop/Q2# ./threadsafe
Updating:
Time:0 sec, 177115 nsec
Acceleration->X=0.000000 Y=0.000000 Z=0.000000 Yaw=1.000000 Pitch=1.000000 Roll=
1.000000
Read:
Time:0 sec, 177115 nsec
Acceleration->X=0.000000 Y=0.000000 Z=0.000000 Yaw=1.000000 Pitch=1.000000 Roll=
1.000000
Updating:
Time:0 sec, 448951 nsec
Acceleration->X=1.000000 Y=1.000000 Z=1.000000 Yaw=2.000000 Pitch=2.000000 Roll=
2.000000
Read:
Time:0 sec, 571258 nsec
Acceleration->X=1.000000 Y=1.000000 Z=1.000000 Yaw=2.000000 Pitch=2.000000 Roll=
2.000000
Updating:
Time:0 sec, 632282 nsec
Acceleration->X=2.000000 Y=2.000000 Z=2.000000 Yaw=3.000000 Pitch=3.000000 Roll=
3.000000
Read:
Time:0 sec, 720806 nsec
Acceleration->X=2.000000 Y=2.000000 Z=2.000000 Yaw=3.000000 Pitch=3.000000 Roll=
3.000000
Updating:
Time:0 sec, 801189 nsec
Acceleration->X=3.000000 Y=3.000000 Z=3.000000 Yaw=4.000000 Pitch=4.000000 Roll=
4.000000
Read:
Time:0 sec, 879230 nsec
Acceleration->X=3.000000 Y=3.000000 Z=3.000000 Yaw=4.000000 Pitch=4.000000 Roll=
4.000000

TEST COMPLETE
root@sowmya-HP-Pavilion-Notebook: /home/sowmya/Desktop/Q2#
```

The code does a thread-safe update of a complex state with a timestamp. It includes two threads - one updates values of all parameters and timestamp contained in a global structure that includes a double precision attitude state, while the other reads the time-stamped state from the same structure and prints out the entire data without data corruption. Both the threads use a

global MUTEX while updating and reading so that the data is never corrupted and the threads remain safe.