ENSC 351: Real-time and Embedded Systems

Craig Scratchley, Fall 2017

Multipart Project Part 5

Please continue working with a partner.

In part 3 of the project we developed code for socketpairs such that when reading from a socket in a POSIX socketpair we only actually called a function for reading when the function call would not block. Instead of blocking on a suitably configured read() function, our implementation of myRead and myReadcond would wait on a condition variable.

Through a 2-step process, I have transformed the solution for Part 3 to avoid making any direct or indirect calls to the POSIX read() and write() functions, and have substituted in instances of a Circular Buffer class to buffer the data being communicated from a write() function to a read() function. The circular buffer template we will use contains my modifications to a class available on the internet: the "Rage Util" CircBuf template found in the StepMania project. Ask if you are interested in the URL to it. My modifications to the class template are primarily changes in the read() and write() member functions to make them more compatible with the POSIX functions: specifically, to have those member functions return the number of bytes that were actually read or written instead of just returning a bool. With my modifications, the read() member function will return a lesser number if elements than requested if the buffer becomes empty, and the write() member function will write a lesser number of elements if the buffer becomes full.

The code as presented online claims at the top that the circular buffer code is threadsafe and lockless. It is misleading to claim the code is threadsafe. What the authors mean, as written a few lines further down from the top, is that a single read and a single write should be able to correctly execute at the same time. Though it is indeed lockless, it is not actually threadsafe for a concurrent read and write on arbitrary computer hardware. Thread safe circular buffer code is not needed by us when we integrate the code into the Part 3 solution, as every time we read or write from the circular buffer we do so with our socketDrainMutex locked and so only one thread can call the circular buffer read or write functions at a time.

Being able to do concurrent reads and writes, however, can be very useful. A lock-free circular buffer can be a great way to transfer data to or from an interrupt routine, like for serial UART hardware. For example, one is not allowed to wait on a mutex in an interrupt routine. An interrupt should finish as quickly as possible and an interrupt routine is not a thread and therefore can't block like a thread can block. On some systems, by default, when an interrupt routine starts other interrupts are disabled. If an interrupt were even to try to block in such a case, deadlock might result.

C++11 and later C++ standards provide for things like atomic variables and such features can allow us to make the circular buffer safe for a concurrent read and write.

These are your tasks.

1) Describe the simplest way to make the Circular Buffer code safe for a concurrent read and write.
2) Modify and submit for grading a version of the code that is not only safe for a concurrent read and write, but also has been updated for concurrency in a way that will be efficient on arbitrary computer hardware. As can be seen in the textbook: "The availability of the distinct memory-

ordering models allows experts to take advantage of the increased performance of the more fine-grained ordering relationships where they're advantageous while allowing the use of the default sequentially-consistent ordering (which is considerably easier to reason about than the others) for those cases that are less critical." Increased performance is definitely advantageous if we are considering using the code in an interrupt routine.

Due date to be announced soon. I will try to get the next part of the project out as soon as possible.