ENSC 351: Real-time and Embedded Systems

Craig Scratchley, Fall 2021

Multipart Project Part 2

Please continue working with a partner.

In part 1 of the project you programmed some code for the YMODEM protocol that generated blocks with CRC16s and responded to (imagined) 'C' characters, ACKs and NAKs. In part 2 we will expand on that code and write additional code to receive blocks and code to hook in a simulator for a communication medium comprising a pair of telephone modems and the telephone system between them.

In the function *Ensc351Part2()* of the provided code, I create at real-time priority 70 a thread identified via *term2Thrd*. That created thread will represent a communication terminal program identified as *term2* (terminal 2). The primary thread will serve as *term1* (terminal 1), and runs at real-time priority 60. Create another thread, identified via *mediumThrd*, for a "kind" medium and have it execute the function *mediumFunc*(). The kind medium should run at real-time priority 40.

We want to send one or more files using the CRC16 YMODEM protocol from *term2Thrd* via the kind medium to the primary thread. That is, *term2Thrd* will generate blocks and send the blocks to the kind medium for forwarding on to the receiver running in the context of the primary thread. I am providing a template project, which contains or will contain elements of the part 1 solution, to use for this part 2 of the project.

The template code as I am giving it to you should compile, run, and actually transfer a file. However, not much checking is done to see what bytes are actually being received on each end. Once the so-called "kind medium" is hooked up, the receiver will not correctly receive the file being sent.

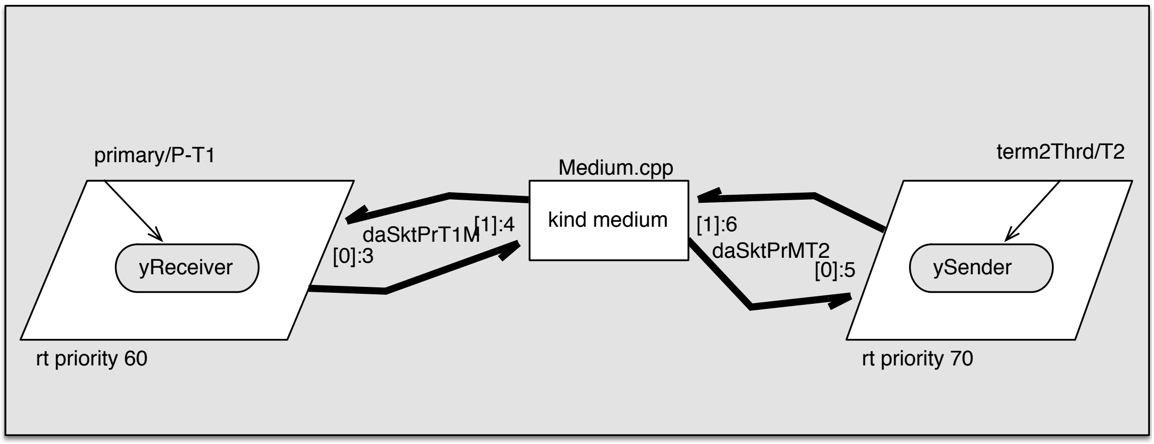
Your mission, should you choose to accept it, is to improve the code so that files can be correctly transferred even when the medium is hooked up. You will need to improve the YMODEM code so that characters like 'C', ACKs and NAKs are properly sent, received, and processed. In general, changes will need to be made where indicated by the markers "*****" in comments in the code.

For now, don't worry about elements of the YMODEM protocol that are not needed for this part of the project. For example, don't worry about timeouts, user cancellation of a session, and purging at the receiver, etc. We will deal with them in later parts of the project. The "kind medium" is kind but not error free. The kind medium complements certain bytes that pass from term2 to term1. The way the medium is written, the first byte of each block for the file we are transmitting should not get complemented. So this means that the SOH byte in our block headers will not be corrupted in this part of the project. The kind medium also introduces some errors when transferring data from terminal 1 to terminal 2: it changes every 9th or so ACK from the YMODEM receiver into a NAK. The kind medium does not drop (i.e. discard) any characters. In any case we consider this medium to be kind and we shouldn't need to worry about timeouts in this part of the project.

In future parts of the project we will have an "evil medium", which may alter the SOH header byte and do other mean things and that will necessitate that further mechanisms be employed in the code to be able to correctly transfer files on most occasions.

When writing the receiver code, consider the StateChart that I will provide for the simplified receiver from this part of the project. There is a document describing StateCharts on Canvas. 1 Before writing the sender code, draw a StateChart for our simplified sender for this part of the project. This term, mostly later in the term, we will be using StateChart software called SmartState. The state *ConditionalTransient* and its outgoing transitions implement in SmartState a "conditional pseudostate" as described on page 7 of the StateCharts document. The entry code 'POST("*",CONT)' in the *ConditionalTransient* state immediately posts a continue (*CONT*) event that immediately kicks the StateChart out of that state (because both transitions out of the state are triggered by that event). 2 You will need to hand in a drawing of your simplified sender StateChart at the time of your Part2 submission. For Part 2 you do not need to use the SmartState software, though it is on Canvas in the Software folder in the Files area if you want to install it. You can draw on paper and scan your page in if you want, or use any drawing program that you may prefer.

To communicate between the threads, we will use socketpairs as indicated in the provided code and in the following diagrams:



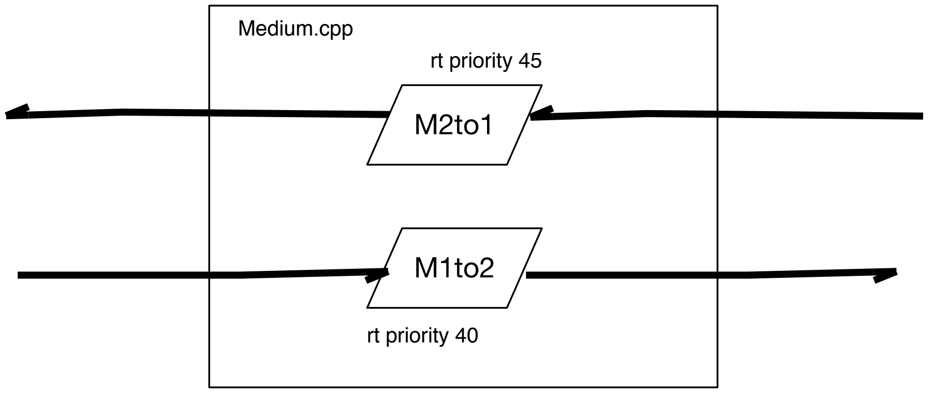


Figure 1: Collaboration graph showing asynchronous communication between threads provided by socketpairs

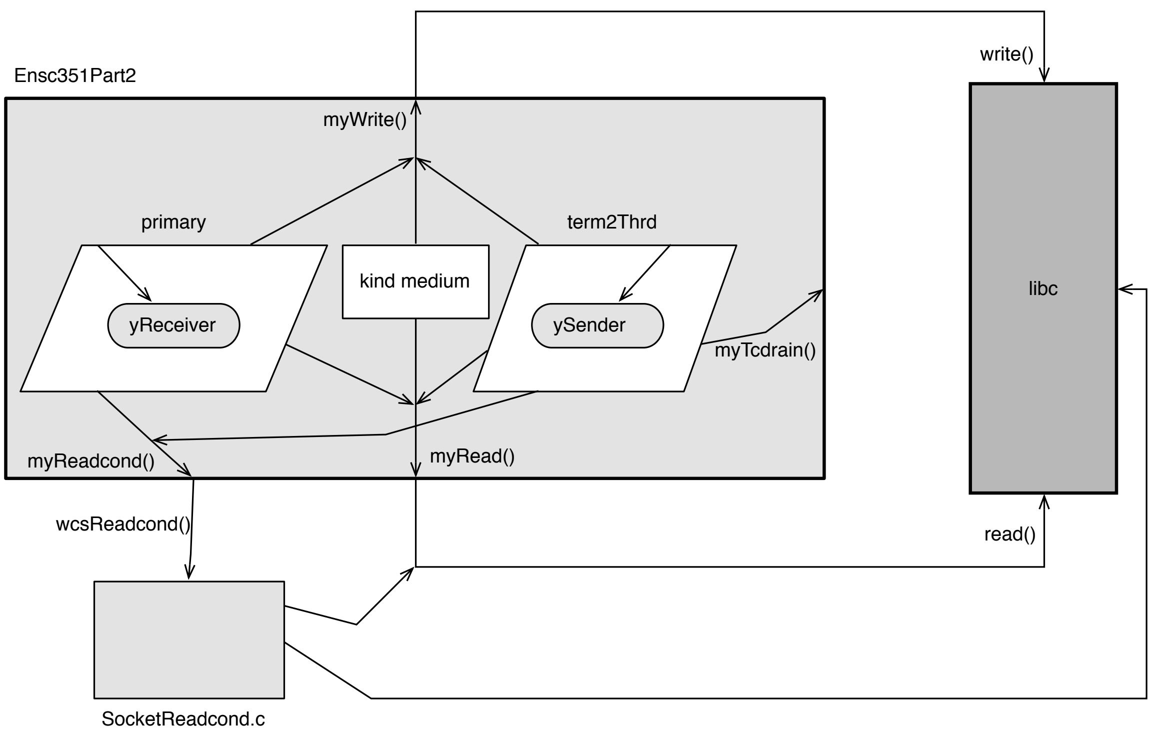


Figure 2: Collaboration Graph showing function calls for socketpair communication

Note that the template code writes to and reads from the socketpairs indirectly using functions *myWrite()*, *myRead()* and *myReadcond()* found in the file *myIO.cpp*. All calls to *myReadcond()*, which provides the functionality of the *readcond()* function ("READ from CONsole Device") available natively in the QNX Real-time Operating System, have the *time* and *timeout* arguments set to 0 for this part of the project. We will need and deal with timeouts later in the course. The supplied code will also call a function *myTcdrain()*, which will in Project Part 3 for socketpairs simulate (and later for serial ports use) the *tcdrain()* function ("Terminal Control: DRAIN"). For this part of the project, myTcdrain() does nothing. myTcdrain() is used so that the YMODEM sender will, in future parts of the project, be able to wait for the medium to have received all the bytes of a block already written to the medium before sending the last byte of the block. That will allow the sender to dump any glitches received before sending that last byte of the block. More on that later.

The YMODEM receiver should create an output file for each file being transferred. The kind medium copies all data it sends in both directions to a special output file ("ymodemData.dat").

You will have two weeks to work on this part of the multipart project. The exact time will be announced shortly on CourSys (courses.cs.sfu.ca).