CSCI 463 Assignment 6 – RISC-V Multithreading

20 Points – Due Friday, December 4, 2020 at 23:59

Abstract
In this assignment, you will create a producer-consumer application that will run on an RV32I simulator based on a multithreaded example called mt01 as presented in lecture.

1 Problem Description
Send a series of ASCII messages from a producer thread to a consumer thread by using an eight-byte lock-free queue as discussed in lecture.

2 Files You Must Write
You will write a C program suitable for execution on the RV32I simulator installed on hopper.cs.niu.edu (or turing.cs.niu.edu.)
Your source file(s) MUST be named exactly as shown below or they will fail to compile and you will receive zero points for this assignment.
Start by copying the mt01 example application (and other examples and Makefiles) from this directory on hopper.cs.niu.edu as shown below:

cp -r /home/hopper/winans/463/rv32i/examples .

As described, this cp command will copy the entire tree of example files into the directory where you ran the cp command into a new directory called examples.
Within these example files, you will find a directory named mt01 as seen in lecture.
You will not need to create any new files for this assignment. You will implement this assignment by editing the main.c file in the mt01 example.

2.1 Alterations to main.cpp
As provided, the mt01 example application configures two harts to call main() where one hart executes the producer() function and a second executes the consumer() function where the producer sends a 26-byte message to the consumer.
You will alter the program so that it will send an arbitrary number of variable-length ASCII text string messages from the producer to the consumer.

2.1.1 main()
Change your main() so that hartid zero will call producer() seven times to send the following text strings:
"111111111111"
"Hello out there"
"aaaaaaaaaaaaa"
"bbbbbbbbbbbb"
"ccc"
"1234"
"\03"  // Note: this is how you can create a string that contains an ETX character.

Note that character code three \03 is the ASCII ETX character (which stands for End Of Text.) We will use it to determine that the producer has finished sending messages.

Change your main() so that hartid one will call consumer() in a loop until the consumer has received a message string with the value "\03".

2.1.2 producer(const char *msg)

Your producer will accept a parameter that is a character pointer to a C string (a series of characters followed by a null-character that terminates the string.) You may assume that the string will never be “be too long.”

The producer must send each string and its null-terminator to the consumer and return.

The producer must be written such that it can be called once for each ASCII message that is to be sent.

2.1.3 consumer()

The consumer must be written such that it can be called once for each ASCII message that is to be received.

Each message must be saved in an array named consumer_buffer, including the null-terminator, after any previous messages such that all of them will remain in the consumer_buffer, one after the other, after the simulation has terminated.

2.1.4 consumer_buffer

Change the definition of consumer_buffer to this:

    char* consumer_buffer = (char *)0x00005000;

Doing so will tell the compiler to use the memory starting at address 0x00005000 as the consumer buffer (as opposed to letting the compiler and/or linker decide where to put it.) Note that writing to this region of memory will clobber anything that might otherwise be there. Make sure to use nm -n prog to make sure that your code, data, and BSS does not grow that high and these two overlap! (If they do, you are likely doing something very wrong.

My solution shows the _end at 00002abc as seen below:

$ nm -n prog
...  
00002ab0 B buffer
00002ab8 B errno
00002abc B __BSS_END__
00002abc B _end

Your program will likely not end in the exact same spot as each solution’s code (and thus number of instructions) is likely to be different. However, they should not differ by multiple kilobytes. So using 0x00005000 should be OK.
3 Output

Your program will be tested by running it and checking to see if the proper values are stored into the consumer's buffer at address 0x00005000 in the memory dump at the end of your program run and by spot checking the values of other variables like ppos, cpos, and buffer.

Running nm on my reference solution includes the following information:

$ nm -n prog
...
00002a74 S cpos
00002a78 S ppos
...
00002ab0 B buffer
...

The post-simulation dump from my reference solution includes with the following values:

make run
...
00002a70: 42 00 00 00 02 00 00 00 02 00 00 00 00 00 00 00 *B.************
00002a80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 *.........
00002a90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 *........
00002aa0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 *........
00002ab0: 03 00 00 31 32 33 34 00 00 00 00 a5 a5 a5 *...1234.....
00002ac0: a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 *.........
...
00004fff0: a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 *...........
00005000: 31 31 31 31 31 31 31 31 31 31 31 00 48 65 6c *11111111111.Hel*
00005010: 6c 6f 20 6f 75 74 20 74 68 65 72 65 00 61 61 61 *lo out there.aaa*
00005020: 61 61 61 61 61 61 61 61 00 62 62 62 62 62 a5 a5 a5 a5 a5 a5 a5 *1234.*
00005030: 62 62 62 62 62 62 62 62 00 00 00 00 00 bbbb bbbb bbbb bbbb bbbb *1234.
00005040: 03 00 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 *...........
00005050: a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 a5 *...........
...

4 How To Hand In Your Program

When you are ready to turn in your assignment, make sure that the only files in your edited mt01 directory is/are the source files defined and discussed above. Then, in the parent of your mt01 directory, use the mailprog.463 command to send the contents of the files in your version of the mt01 project directory in the same manner as we have used in the past.

5 Grading

The grade you receive on this programming assignment will be scored according to the syllabus and its ability to compile and execute on the Computer Science Department’s computer.

It is your responsibility to test your program thoroughly.
When we grade your assignment, we will compile it on hopper.cs.niu.edu by using ONLY your main.c file and original copies of the other mt01 example program source and Makefile files. In other words, make SURE that you save and test your final program with all the other original files from the mt01 example files. (Keep backup files so you don’t clobber your work when making copies of things!)

6 Hints

As always, build up a solution one step at a time. Multithreaded programs can be VERY difficult to think about. Take VERY SMALL steps!

- It is likely that this assignment is impossible to complete without watching the related course lectures first. If you have not yet done so then watch them now!

- Start by making sure that you can run the mt01 example as seen in lecture. If it does not work then something is wrong and you can not proceed. Try checking your copy of the example files and that you have the mt01 files AS WELL as the Makefile and Make.rules files in the parent directory. (It is easiest to take all the files and examples as shown above and in lecture.)

- Note that your producer must be written such that it can be called from main multiple times! If/when producer() is called a second, third,... time, it must pick up by using the ppos value where it left off from the last time producer() was called. In order words, you DON’T want to “reset it” nor “clear” the buffer when producer() is called. (Remember that ppos and cpos are also used by the consumer!)

- Note that the producer has to send all the characters in the text string INCLUDING its null-terminating characters. Otherwise the consumer will not know where the end(s) of each text message(s) is/are. There are multiple ways to do this:

  1. Set a flag in the loop when it sends the null-character from the given msg parameter such that the flag terminates your sending loop and the function returns.

  2. Count the number of characters in the string (including the null-terminator) and then send that number of characters from the given msg parameter.

- Note that your consumer must be made such that it can be called from main() multiple times! Like the producer, note that it does not matter where cpos is when your consumer() is finished. The important thing here is that if/when consumer() is called again, it must pick up by using the cpos value where it left off from the last time it was called. Again, you dont wait to “reset cpos or ppos.”

- Note that the consumer must read one string at a time from the producer. To do so, it must read all the characters until it reaches a null-terminator character (and store the null-terminator into the consumer_buffer too) and then return. I found the easiest way for me to do this was to set a flag in the consumer’s reading loop that terminates said loop after it has received and stored the null-terminator character into the consumer_buffer.

- Recall that this is a C program. You probably want to use functions like strlen() and strcmp() when working with C strings.

- It is now tougher to stop the simulator any time you want to because, while you can halt one hart with an ebreak instruction, the other might continue on indefinitely.
  
  You might have to experiment with the max instruction limit if you are completely in trouble.
• Consider running your program with only one hart and send short (1-character) messages until you can see that your producer is filling in the buffer and managing its ppos variable perfectly before you run with two harts and proceed to working on your consumer. Recall that you can send one or two small messages and let your producer return and halt so that you can get your simulator to halt while focusing on sending those one or two messages in your consumer and checking that the producer() is working perfectly before trying to debug two threads at once.

You can also start your application with cpos and ppos set to any starting values desires as long as they both start with the same value. Consider how this might help verify your producer is operating properly when testing this one-hart scenario.

Note that with only one hart running, you can halt it any time you want by placing an ebreak instruction anywhere desired by using an inline assembler instruction as seen in lecture. (See the _exit() function in stub_stdlib.c for an example how to do this.)

• Recall that if you create a test case where your producer never reaches the end of the buffer (like if it only sends two one-byte messages), that it will enqueue those messages without ever blocking and then it will return and halt.

When in this state, the consumer might still be running to read the messages that the producer has placed into the buffer... and if you have already perfected your producer and know it has halted, then you can toss in an ebreak instruction using inline assembler as seen in lecture to stop your consumer anywhere you want.

• There are many ways to figure out if a message containing ETX (aka \03) has arrived or not. I propose that you keep track of the “next_byte_to_fill index” of the consumer_buffer in a global variable that is used and advanced in your consumer() function so that the code in main() can know where it is after consumer() returns... so you can know where to look and see if the last message was an \03. Recall that "\03" is a two bytes with the values 0x03 and 0x00.

Checking if the last message is an 0x03 and a 0x00 has to be done carefully. Make sure that you first check if the consumer_buffer has at least two bytes in it before you decide to blindly look at a value... such as consumer_buffer[next_byte_to_fill - 2].

7 Stuff You Should Know

• What sort of maximum limit do you suppose is reasonable for the length of any one message that is passed to producer(const char *msg) for sending?

• How many total messages can we send with this application?

• What sum-total number of message bytes can this application send?

• What would happen if this program is executed with only one hart and why?

• What would happen if this program is executed with three or more harts and why?