Bowen Yang/by11
COMP521
February 10, 2020

Lab 1 Report

In the design and implementation of the device driver for the OS, I used three character arrays as the buffer designed as input buffer, echo buffer and output buffer for each terminal. Input buffer is only for *ReadTerminal*() to read from, echo buffer is only for dealing with "echoing" characters back to the terminal, and output buffer is only for *WriteTerminal*() to write on. Theoretically, I think there is a way to only manage a single buffer to deal with all three main operations, which would only take up one third of the space for buffers, but this separation would make my code functionality easy to implement and manage. Also, to make things easier, I have integer values for each buffer that indicate the write position, read position as well as count for number of characters left not dealt with. I have created five condition variables, *WDT_lock*, *WT_num_lock*, *WT_writing_lock*, *RT_num_lock*, and *RT_input_buffer_lock* to control the flow of the thread, and I will elaborate the uses of them below.

My *ReceiveInterrupt*() is responsible for updating the echo buffer and input buffer. This function basically updates the buffers one character at a time, and when special cases appear (such as \b or \n), it updates the two buffers accordingly. When done updating the buffers, if there is any character left in the echo buffer not dealt with and the *WriteDataRegister*() is not busy being called by some other procedures, only then will *ReceiveInterrupt*() kick start the very first *WriteDataRegister*(). One thing worth mentioning is that, whenever the input buffer get a newline character, *ReceiveInterrupt*() would *signal*() *RT_input_buffer_lock* to let *ReadTerminal*() starting reading from input buffer, regardless if there is any *ReadTerminal*() waiting.

TransmitInterrupt() is the function that mostly handles transmitting character to the terminal. When a first *WriteDataRegister*() is started by a *ReceiveInterrupt*() call or a *WriteDataRegister*() call, the *TransmitInterrupt*() will be called consecutively and repeatedly, until there is nothing to transmit to the terminal. A *TransmitInterrupt*() call can only do a *WriteDataRegister*() call a time. In *TransmitInterrupt*(), the echo buffer is always first examined since "echoing" has the highest priority. If there is any character not transmitted in the echo buffer (by checking the count of echo buffer),

the *TransmitInterrupt*() will always transmit the stuff in echo buffer first. When echo buffer is empty, *TransmitInterrupt*() then examines the output buffer and starts transmitting characters there. Only when the count for both buffer is empty, the *TransmitInterrupt*() writes nothing and releases the *WDR_lock* and *WT_writing_lock*, letting other procedures that want to call *WriteDataRegister*() know that it is available now. There is no waiting in neither interrupt functions, as specified by professor.

ReadTerminal() first check if there are any preceding *ReadTerminal*() calls by checking *RT_num*. If there are, wait on *RT_num_lock*. After that, it checks if there is stuff to read from in the input buffer, if there is not, wait on *RT_input_buffer_lock*. When *ReadTerminal*() gains mutual exclusion, I used a for loop read the characters from input buffer. Finally, signal the *RT_num* lock.

For my *WriteTerminal*(), I used three condition variables: *WT_num_lock* to ensure there is only one *WriteTerminal*() being executed; *WDR_lock* to make sure echo buffer is always first dealt with; and *WT_writing_lock* to make sure the output buffer is emptied before a new *WriteDataRegister*() call. When *WriteTerminal*() gains mutual exclusion, I simply copy the buffer into output buffer, leaving it to be handled by *TransmitInterrupt*().

What design choices did you make in your solution to the project that you intended to improve the performance of your solution?

As I have mentioned above, instead of blocking the whole thread with only one condition variable, I used five condition variables to control the flow of the thread. Each condition variable is only reasonable for one circumstance and hence they are independent, which means when there is something requires waiting on the previous writing to finish, the reading should not be affected at all. This decoupling of condition variable only not provides the device driver with atomicity and consistency but also maximally utilizes the hardware.

What design choices would you have made in your solution, if you'd had more time to implement them, that could have improved the performance of your solution?

I would imagine that there is a way to only use one char array as the buffer to interact with the user and the hardware. In that case, I can manage to cut down two thirds of the space I have used for buffering. However, that requires delicate manipulation of the buffer and I would definitely need more time to work on that.

I have also tried to use less condition variables in order to make the logic of my code clearer. I tried make my WriteTerminal() wait on the same condition variable when there is something to "echo" and when the previous output buffer is not fully transmitted. However, in that implementation, when I type really fast I get the error saying *WriteDataRegister*() called when its busy. So this use of condition variable is so far the best I can come up with.

What are the biggest issues affecting the ability of your solution, and what your solution could have been if you'd had more time, to scale to larger sizes and larger systems?

When I need to manage my buffers, I manipulate them directly, which I think is to some extent dangerous and lack of logic decoupling. When it comes to larger systems and larger sizes, I would think a specific struct and corresponding functions that are only responsible for dealing with management of the buffer are necessary, not only for better performance but also for correctness of every buffer manipulation. It would be better for debugging.

Also, I am simply using arrays of Integers and condition variables to handle different terminal, which may require a specific set of struct as well as functions rather than directly manipulating them.