COSC440 Assignment 2 Report

Edward Hills

May 18, 2012

Circular Buffer

The design of the circular buffer. I decided to go with a design that was fairly simple and intuitive. The main design choice in this section was to do overriding rather than waiting when a byte was trying to be written. This was mainly decided by the fact that we have fairly slow devices and that the chance of the overriding occurring is very slim and even then it does not matter too much.

Bottom-half implementation

I chose to implement the bottom half with a Tasklet. This is because using the tasklet provides me with added atomicity due to the fact that only one tasklet can be scheduled at a time by the operating system. Unlike work-queues in which there can be many and so race conditions can more easily occur. Having the tasklet means I can avoid the added complexity of checking for race conditions in most places.

User access

The way I managed that the user only gets one file per process is by simply keeping track of where the null character is when writing in the bottom half to the multiple page list and then having an eof flag which I set once it has read up to but not including the position where the null is stored.

Possible race condition

I did at one stage in my program have the possibility that the consumer process could be reading from the page at the same time as the bottom half was writing to it. This race condition was due to the fact that my consumer process would immediately try and start reading once it had detected there was bytes available for it. The best way to avoid this race condition I decided was to only have the consumer start reading once an entire file had been written to the page list. This means that the consumer and producer would be able to avoid the race condition.

Freeing used pages

When to free the pages was also another important decision. I decided to free the page as I was reading as this allowed a greater amount of memory for producer processes when they start to write rather than having to wait until an entire file was written. This was difficult as I needed to then change the array which stored the position of the nulls to reflect that the page was no longer there. So then when the consumer process started consuming it knew where the null was in the current context of how many pages there are.

General race conditions

Keeping race conditions to a minimum was -fairly- easy. This is because most of the areas were set for atomicity fairly early on in the process. What I mean by this is that we kept reader process race conditions away by making sure that only one process at a time can access the device, all the others are put into the wait-queue and block until the previous process has finished. Another way data races were avoided is due to the fact that I used a tasklet. This means that all of the bottom half is guaranteed by the operating system to run without other tasklets interfeering. This means that during all bottom half actions, you only need to be concerned with the one tasklet and not worry about others interfering.

Read process

In the read() function I make the consumer process wait if there is not a whole file for it to read. I do this by putting the process into a wait queue. This could have been accomplished just as easily as with a mutex or semaphore however I decided to use a wait queue so that if I was to change the program and make it have more than one process accessing the device then the queue could be of use. This is just a form of future proofing, also I found it easiest to implement and think about so I left it as that.