COMP 530 INTRODUCTION TO OPERATING SYSTEMS

Fall 2012
Kevin Jeffay

Homework 3, September 19

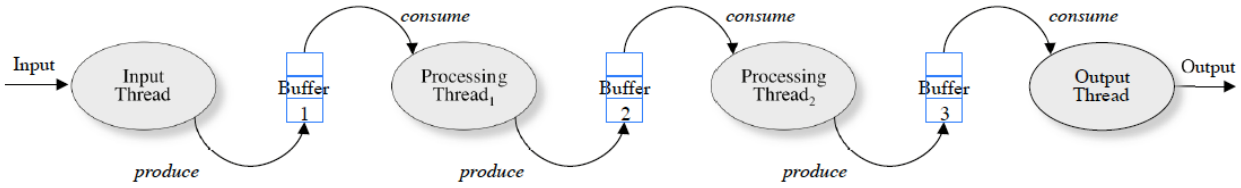
Due: October 1

The "Linux Warmup" Revisited: Producer/Consumer Interaction with a Bounded Buffer

In this exercise you will re-implement your sequential C program from Homework 1 as a multithreaded *C* program using the *ST* threading package.

The goal of this exercise is decompose the processing of Homework 1 into a data-flow "pipeline" and to have a thread responsible for the processing of each stage in the pipeline. As the term is used here, a pipeline refers to an organization of threads wherein each thread inputs data from a single source (either a device or another thread) and outputs data to a single sink (which is again either a device or another thread). In such a pipeline, a pair of adjacent threads acts as a simple producer/consumer system wherein one thread processes data and then "produces" processed data for the next thread to "consume."

In more detail, in a data-flow pipeline *each pair* of communicating threads is constructed as a producer/consumer system with a bounded buffer. The i^{th} thread in the pipeline, T_i, reads (inputs) data that is generated by thread T_{i-1}, and writes (outputs) data for use by thread T_{i+1}. Thus, thread T_i simultaneously acts as a consumer with respect to thread T_{i-1} (with thread T_{i-1} playing the role of thread T_i's producer), and thread T_i acts as a producer with respect to thread T_{i+1} (with thread T_{i+1} playing the role of thread T_i's consumer). A graphical illustration of a producer/consumer processing pipeline is shown below. Note that in this scenario, each thread in the interior of the pipeline will access *two* different sets of buffers and contain *both* producer code and consumer code.



For this assignment, an obvious pipeline to construct would be to assign a thread to each of the main functions of your homework 1 program, namely, reading lines of input from standard input, processing characters, and writing 80 character lines of text to standard output. That is, one *might possibly* have a first thread perform input (*e.g.*, read from standard input on a line by line basis) and generate a stream of characters for a first processing thread that replaced carriage returns with blanks. This first processing thread might forward the processed characters (via a buffer) to a second processing thread that dealt with asterisks, and so on.

For example a high-level conceptual structure for thread T_i might be as follows.

```
loop
   bounded buffer code to consume ("get") a data object from buffer i
   (previously written by thread i–1)

   process data object

   bounded buffer code to produce ("deposit") a data object in buffer i+1
   (to be later read by thread i+1)

   exit when ???
end loop
```

In more detail, each producer/consumer pair of threads should have its own collection of shared buffers. Thus in the code schema above, thread T_i would be operating on two different sets of buffers — one buffer shared with thread T_{i-1}, and one buffer shared with thread T_{i+1}. Semaphores should be used to synchronize accesses by each producer and consumer pair of threads to the buffers they share. A C-based semaphore utility will be provided to you for this purpose.

You should design a bounded-buffer abstract type for use with your program. This data type should encapsulate all of the implementation details of the buffer and simply export functions "deposit" and "remove" (and possibly a function to create/initialize the buffer). The code for your bounded buffer data type should be contained in a separate file from your code for your main program.

From the standpoint of the user, your program should behave *exactly* as a correct solution to Homework 1 does. That is, the program will read from standard input and write 80 character lines to standard output changing carriage returns to spaces, and pairs of asterisks to caret characters. (See the Homework 1 handout for the complete specification of the requirements.) Your program will be tested the same way your homework 1 program was tested.

The purpose of this assignment is to gain experience using basic multiprogramming concepts, specifically, producer/consumer process interaction and synchronization, buffering, and the use of threads as a program structuring mechanism.

A Semaphore Abstract Data Type

The *ST* thread package does not provide semaphores as a synchronization primitive. However, one can easily construct a semaphore class using *ST* primitives (see the extra credit description below). We will make a semaphore abstract data type available to you. The following is the header file of the semaphore data type.

```
/* semaphore.h
 *
 * An ST extension implementing classical general/counting semaphores.
 *
 * Public Interface -
 *
 *    typedef semaphore;
 *
 *    void down(semaphore *s) - blocks the calling thread if the
 *                              value of the of the semaphore is 0,
 *                              otherwise the value of the semaphore
 *                              is decremented.
 *
 *    void up(semaphore *s)   - increments the value of the semaphore.
 *                              If a thread is blocked on the semaphore it
 *                              is woken up.
 *
 *    void createSem(semaphore *s, int value ) - sets the initial integer
 *                              value of the semaphore. 'value' must be
 *                              nonnegative.
 */
```

Grading

"Submit" your program electronically for grading by emailing *mxrider@cs.unc.edu* when the program is ready for grading. Programs whose files are dated after 11:00 AM on October 1 will be considered late. You will submit three (3) files for this homework assignment. Your main program should be in a file called *HW3.c*. Your bounded buffer abstract data type should be in files *buffer.c* and *buffer.h*.

The program should be neatly formatted (*i.e.*, easy to read) and well documented. For this program, approximately 40% of your grade for a program will be for correctness and 60% for "programming style" (appropriate use of language features, including variable/procedure names, implementation of a bounded buffer abstract data type, structure of thread pipeline, termination of threads, *etc.*), and documentation (descriptions of functions, general comments, use of invariants, assertions, pre- and post conditions).

Extra Credit

For extra credit, use the condition variable primitives in *ST* to implement your own semaphore abstract data type using the same interface given above. Put the code for your semaphore implementation in separate files named *semaphore.c* and *semaphore.h*. When you send the email message indicating that your program is ready for grading, please indicate that you did the extra credit portion of the assignment.

And please note that, as always, you should absolutely not even think about the extra credit portion of the assignment until after you have a *perfectly* executing version of HW3.