COP4610 - Operating Systems

Programming Assignment 1

Joseph Allen

10/2/18

**Program 1**

In my program I have 4 classes:

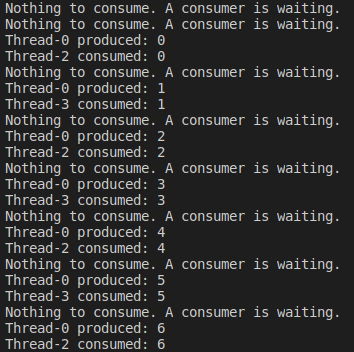
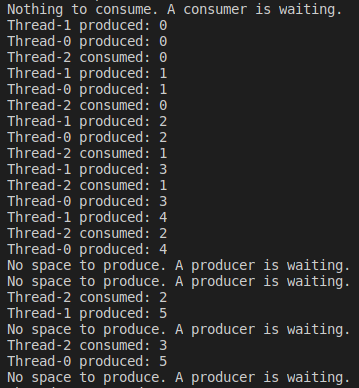
* **P1** – contains main; creates BoundedBuffer object; creates and starts Producers & Consumers.
* **BoundedBuffer** – has a constructor to create a generic-type array of a given size; contains methods to produce or consume items in the array.
* **Producer** – used to create a thread that continuously calls produce() on a BoundedBuffer instance; in this program I pass produce() integers that increment starting from 0.
* **Consumer** – used to create a thread that continuously calls consume() on a BoundedBuffer instance.

The program handles the following conditions:

* A producer (consumer) producing (consuming) past the end (beginning) of the buffer.
  + It handles this by telling producers (consumers) to wait() if the buffer is full (empty).
* A producer writing to an array element that is not EMPTY and a consumer reading (i.e. setting the array element to EMPTY) from an array element that is not FULL.
  + I handle this by keeping track of the index in the array that a producer (consumer) can set to FULL (can set to EMPTY). I use index % BUFFER\_SIZE to keep indices within the buffer.
* Use wait() and notify() to avoid errors producing and consuming and to ensure thread synchronization.
  + My program uses wait() and notifyAll() within synchronized blocks to ensure only one thread at a time can access the critical buffer data.

Examples of output when the BoundedBuffer instance has BUFFER\_SIZE of 5:

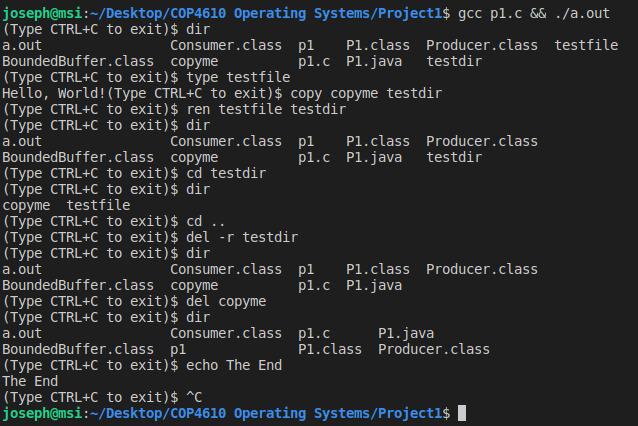
* On the left is an example of the program executing with 2 producers and 1 consumer. The consumer can’t keep up with the 2 producers and the producers end up waiting on it to free up space for them.
* On the right is an example of the program executing with 1 producer and 2 consumers. From the very start, the consumers are waiting for items to consume.



**Program 2**

In my program I have 4 functions:

* **main** – runs program loop that 1) captures user input, 2) calls parse\_input(), 3) calls translate(), and 4) forks the process and calls execute() within the child (the parent waits for child)
* **parse\_input** – accepts an input string and parses it for a command and any number of arguments; returns pointers to each token.
* **translate** – accepts pointers to strings and translates the first one (the command) if appropriate.
* **execute** – calls chdir() if the command is ‘cd’, otherwise it calls execvp(<command>, <args>)

My program functions as a DOS to UNIX interpreter and successfully translates the commands dir, type, del, ren, and copy. Any non-DOS commands skip the translation process and are executed in the shell as normal commands.