Assignment 4: Multi-threaded Programming

Submit Assignment

Due Friday by 11:59pm **Points** 90 **Submitting** a file upload **Available** Aug 5 at 8am - Aug 14 at 11:59pm 10 days

Introduction

In this assignment you will apply the principles of multi-threaded programming to write two programs.

- You will write a C program that uses the Producer-Consumer model to implement a character processing program.
- You will modify a single threaded Rust program provided to you that processes data using the Map-Reduce programming model to create a multi-threaded version.

C Program: Multi-threaded Producer Consumer Pipeline

The program will read in lines of characters from the standard input and write them as 80 character long lines to standard output with the following changes:

1. Every line separator in the input will be replaced by a space
2. Every adjacent pair of plus signs, i.e., "++", is replaced by a "^".

Input

- A “line of input” is
  - A sequence of the allowed characters (see below) that does not include a line separator
  - Followed by a line separator.
- Allowed characters: Other than the line separator, the input will only consist of ASCII characters from space (decimal 32) to tilde (decimal 126). These are sometimes termed printable characters.
- The input will not contain any empty lines, i.e., lines that only have space characters or no characters except the line separator.
- An input line will never be longer than 1000 characters (including the line separator).
- Your program doesn't need to check the input for correctness.
- Your program should process input lines until it receives an input line that contains only the characters DONE (followed immediately by the line separator) at which point the program should terminate.
  - Examples: The following input lines should not terminate the program
    - DONE!
    - ALLDONE

Output

- The “80 character line” to be written to standard output is defined as 80 non-line separator characters plus a line separator.
- Whenever your program has sufficient data for an output line, the output line must be produced. Your program must not wait to produce the output only when the terminating line is received.
- No part of the terminating line should be written to the output
- Your program should output only lines with 80 characters (with a line separator after each line).
  - For example,
    - If your program has some input that has not been written because the program is waiting for more data in order to make an 80 character line for output and the terminating line is received from standard input,

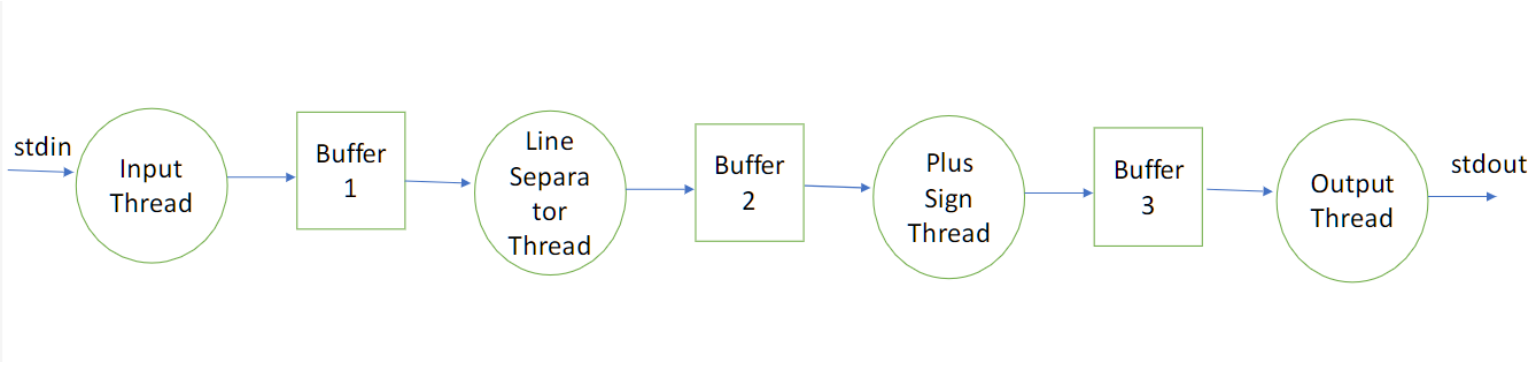
    - Then the program should terminate without printing out the partial output line.
- In addition, your program should not output any user prompts, debugging information, status messages, etc.
- For the second replacement requirement, pairs of plus signs should be replaced as they are seen.
    - Examples:
        - The string “abc+++def” contains only one pair of plus signs and should be converted to the string "abc^+def".
        - The string “abc++++def” contains two pairs of plus signs and should be converted to the string "abc^^def".

Multi-threading Requirement

Your program will create 4 threads as follows:

1. Input Thread: This thread performs input on a line-by-line basis from standard input.
2. Line Separator Thread: This thread replaces line separators with blanks.
3. Plus Sign Thread: This thread performs the required replacement of pair of plus signs.
4. Output Thread: This thread writes output lines to the standard output.

Producer Consumer Pipeline



Pipeline of threads that gets data from stdin, processes it and displays it to stdout

- Each pair of communicating threads should be constructed as a producer/consumer system.
- If a thread T1 gets its input data from another thread T0, and T1 outputs data for use by another thread T2, then
    - T1 acts as a consumer with respect to T0 and T0 plays the role of T1’s producer
    - T1 acts as a producer with respect to T2 and T2 plays the role of T1’s consumer
- Thus each thread in the interior of the pipeline will contain both producer code and consumer code.
- Each producer/consumer pair of threads will have its own shared buffer. Thus, there will be 3 of these buffers in your program, each one shared only by its producer and consumer.
- You must use condition variables for coordination. Your program must never sleep.
- Your program will never receive more than 10 input lines in any 5 second interval.
    - If you size your buffers to hold 10 lines of adequate size, you can model the problem as Producer-Consumer with unbounded buffers which will make your program simpler.
        - **[Clarification added August 7] Adequate size = 1000 characters each**
    - Recall that unbounded buffers are never full, though they can be empty.

Hints

- Create a single threaded implementation of the required processing to get the processing logic ironed out.
    - In the single-threaded version, put the processing to be done by each thread in a separate function. Then you can use that function in your multi-threaded program.
    - Then create a multi-threaded implementation of your program. For the same input, the output of both implementations of the

programs should be the same.

- For the multi-threaded implementation:
  - Start with the **example program** (https://repl.it/@cs344/64prodconscvc#main.c) in **Module 6: Exploration Condition Variables** that uses condition variables for the producer-consumer problem.
  - Modify the program to implement the first producer-consumer interaction, i.e, between the Input Thread and the Line Separator Thread.
  - Now you can duplicate the code and make the adjustments for the next producer-consumer interaction, i.e., between the Line Separator Thread and the Plus Sign Thread.
  - Finally, do the same for the last producer-consumer interaction, i.e., between the Plus Sign Thread and the Output Thread.

Where to Develop

- We recommend that you develop this program directly on our class server! Do not use any other non-class server to develop these programs.
- You can only use C for coding this assignment and you must use the gcc compiler. You can use C99 or GNU99 standard or the default standard used by the gcc installation on os1.
- If you do see ^M characters all over your files, which come from copying a Windows-saved file onto a Unix file system, try this command:

```
$ dos2unix bustedFile
```

What to Submit

- Submit a single zip file named YOUR_ONID_program4.zip with your C program.
- Your C program can be in as many different files as you want.
- Inside that zip file, you must provide a text file called README (or README.txt) that contains instructions on HOW to compile your code.
- The compilation instructions must result in creating an executable file with the name **line_processor**.

Please be aware that neither the Instructor nor the TA(s) are alerted to comments added to the text boxes in Canvas that are alongside your assignment submissions, and they may not be seen. No notifications (email or otherwise) are sent out when these comments are added, so we aren't aware that you have added content!

If you need to make a meta-comment about this assignment, please include it in the README file as part of your submission, or email the person directly who will be grading it (see the Syllabus page for grading responsibilities).

Rust Program: Multi-threaded Map Reduce

According to **the Wikipedia article on MapReduce** (https://en.wikipedia.org/wiki/MapReduce)

> MapReduce is a programming model and an associated implementation for processing and generating big data sets with a parallel, distributed algorithm on a cluster.
>
> A MapReduce program is composed of a map procedure, which performs filtering and sorting (such as sorting students by first name into queues, one queue for each name), and a reduce method, which performs a summary operation (such as counting the number of students in each queue, yielding name frequencies).

For this part of the assignment, we are providing you with a **single-threaded Rust program** (https://repl.it/@cs344/pa4mapreducers#main.rs) that processes input numbers to produce a sum. The program contains extensive comments that explain the functionality and give directions on what parts of code you are allowed to change (look for comments starting with CHANGE CODE).

Here is a description of the program: currently the `main()` function does the following

- Generates data for the rest of the program

  - Calls `generate_data()` to generates a vector of numbers that serves as input for the rest of the program.
- Partitions the data
  - Calls `partition_data_in_two()` which partitions the input numbers into two partitions
- Performs the map step
  - Calls `map_data()` for each of the two partitions, which returns the sum of all the numbers in that partition.
- Performs the reduce step
  - Gathers the intermediate results produced by each call to `map_data()`
  - Calls `reduce_data()` that sums up the intermediate results produced by the map step to produce the final sum of all the input numbers.

You have to modify the program to accomplish the following tasks:

- Modify the program to create 2 threads, each of which runs the `map_data()` function on one of the two partitions created by the program given to you.
- Add code for the function `partition_data()` to partition the data into equal-sized partitions based on the argument `num_partitions`
- Add code to the function `main()` to
  - Partition the data into equal-size partitions
  - Create as many threads as the number of partitions and have each thread run the `map_data()` function to process one partition each
  - Gather the intermediate results returned by each thread
  - Run the reduce step to process the intermediate results and produce the final result

See detailed comments in the provided program to see how you can go about making the required changes. Also, review the Hints section and the Grading Rubric.

Hints

- The function **thread::spawn()** (https://doc.rust-lang.org/std/thread/fn.spawn.html) returns `JoinHandle<T>` where T is the type of the return value of the function the thread runs. This means that
  - Because `map_data()` returns an integer of type `usize`
  - If you spawn a thread that runs `map_data()`
  - `thread::spawn()` will return a value of type `JoinHandle<usize>`

Where to Develop

- Your program will be tested on os1, so we recommend you develop on os1
- To test your program, we will compile the code as follows

```
rustc main.rs
```

- We will run the program as follows

```
./main num_partitions num_elements
```

E.g.,

```
./main 5 150
```

What to Submit

- Upload your code in one file main.rs.
- If you have any meta-comments on this program, create a file README_RUST.txt and put the comments in that file, and upload that file with your submission as well.

Grading

~~This assignment is worth 125 points and contributes 15% of your final grade~~. The C program is worth 75 points. The Rust program is worth 50 points. Further breakdown of the points by functionality is given in the grading rubric.

Update August 9:

We are making part of this assignment extra credit. Specifically

- The maximum points you can get on the assignment is still 125 (the grading rubric remains unchanged)
- However, a score of 90 points will be considered full marks on the assignment and is worth 15% of the grade
- Any points above 90 will give you extra credit.
- E.g., if a student gets 125 points, then
  - They get 15% for the first 90 points
  - Another 5.83% added to their course grade for the 35 points beyond 90

Assignment 4

Criteria	Ratings		Pts
C Program: Input and line separator threads are created and interact correctly as Producer-Consumer	**20.0 pts** **Full Marks**	**0.0 pts** **No Marks**	20.0 pts
C Program: Newlines are correctly replaced by spaces	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
C Program: Plus Sign thread is created and interacts correctly with New Line thread in the pipeline	**10.0 pts** **Full Marks**	**0.0 pts** **No Marks**	10.0 pts
C Program: Pair of plus signs are correctly replaced by the caret symbol	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
C Program: Output thread is created and interacts correctly with the Line Separator thread in the pipeline	**10.0 pts** **Full Marks**	**0.0 pts** **No Marks**	10.0 pts
C Program: A line of output is produced as soon as 80 characters are available to output	**15.0 pts** **Full Marks**	**0.0 pts** **No Marks**	15.0 pts
C Program: Output lines are always 80 character long	**5.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
C Program: The code is well commented	**5.0 to >0.0 pts** **Full Marks**	**0.0 pts** **No Marks**	5.0 pts
Rust Program: Processes the two partitions in the provided program by creating two thread each of which runs map_data() on one partition each	**15.0 pts** **Full Marks**	**0.0 pts** **No Marks**	15.0 pts
Rust Program: partition_data() partitions the data evenly based on the argument num_partitions	**10.0 pts** **Full Marks**	**0.0 pts** **No Marks**	10.0 pts
Rust Program: Implements map-reduce using as many threads as the number of partitions in the argument	**25.0 pts** **Full Marks**	**0.0 pts** **No Marks**	25.0 pts
			Total Points: 125.0