POSIX Thread Programming

**Team name: Group 4**

Name of the VM: 🡨 Centos7VM (under Brogan in the EAS site)

Password: 🡨 IloveCS4500

Team members names:

Jaylen McKinney | Worked on half of part 1 and part 3

Brogan Bewley | Worked on half of part 1 and part 2

# Description of Project: 2 – 3 Sentences

**Here write a few (2-3) sentences describing the purpose of the program. You may be more specific later in the pseudocode or conclusion section of this report. You should write in complete sentences to describe the purpose:**

Task 1: Write a Pthread program to find the number of substrings in string s1 that are the same as the characters in s2. Input strings are read from a file, and the parallel solution involves dividing the task among threads and updating a global variable to represent the total number of matched substrings.

Task 2: Implement a Pthread program using condition variables to solve a producer-consumer problem. The producer reads characters from a file and writes them into a circular queue, while the consumer reads from the queue and prints characters in the same order. The circular queue has a fixed size of 12 characters.

Task 3: Modified code from a provided file, list-forming.c, to improve performance. The program involves multiple threads creating data nodes and attaching them to a global list. The goal is to optimize the program runtime, considering that attaching nodes to the global list needs protection with a lock, and reducing the time spent acquiring the lock can contribute to runtime speed.

# Pseudocode: This can be a picture, drawing, sketch, or digitally produced. For example, if you hand-wrote it on a sheet of paper, convert it into and image and paste it here. If you typed it in notepad, copy and paste it here.

**Pseudocode for part 1**

Readfile function

1. Make sure that the file exists
2. Allocate memory for the string and the substring
3. Read the strings from the file
   1. If the strings are empty, return an error message
4. Create two variables to hold the length of the strings

Num\_substring function

1. Create integer to hold the number of matching characters
2. Create integer variable to hold number of substrings
3. Create a variable for the starting position
4. Create a variable for the ending position
5. For loop from value of start to value of end
   1. Reset count to 0
   2. Nested for loop for substring count
      1. If the two characters at position i of string and substring match, increment count by 1
      2. Else if the count equals the length of the substring, a match has been found
         1. Lock thread
         2. Increment total by 1
         3. Unlock thread

Main function

1. Create array of threads of type pthread\_t
2. Create count integer
3. Jump to readfile function to read from file
4. Create array of thread number
5. Initialize lock for threads
6. For loop for threads from 0 to the current number of threads
   1. Split string into 4 separate strings for threads
   2. Create a thread and run num\_substring function
7. For loop to join the threads from 0 to current number of threads
   1. Wait for the thread to terminate
8. Destroy the lock
9. Exit program

**Pseudocode for part 2**

1. Set up variables for buffer size, message filename, and counting variables for the producer and consumer.
2. Set up condition variables for producer and consumer and a mutex to know when the critical region is locked.
3. Producer thread:
   1. Attempt to open file (error out if empty or nonexistent)
   2. Read from file and add to buffer till full, making sure to lock the mutex while accessing the information.
   3. If the buffer is full, have related condition variable wait and print message alerting that buffer is full.
   4. Do this until the message is empty and close the file upon finishing.
4. Consumer thread:
   1. Constantly check if there is text in the buffer.
   2. Lock the critical region to check.
   3. If the buffer is empty, signal associated condition variable to wait and print a message saying such.
   4. If the buffer is not empty, get a character from it making sure to lock the mutex while accessing the critical region.
   5. Print variable out, return null once the message has finished
5. Put & get helper functions
   1. Put is used to add a character to the buffer and increment the counting variables associated with adding to it for next time.
   2. Get is for retrieving the next character in the buffer and increments the counting variables associated with taking from the buffer.
   3. For both, make sure to not increment over buffer size by using modulus division to see when to start back at 0.
6. Main
   1. Initialize condition variables and mutex.
   2. Create instance of producer and consumer thread.
   3. Let them run and join them together after finishing.
   4. Finally, destroy condition variables and mutex.
   5. Return 0 to signal end of the program.

**Pseudocode for part 3**

1. Create K data nodes
2. Create a struct to create a node for a linked list
3. Create a lock for the pthread
4. Create a list pointer for the linked list

Bind to CPU function (Sets up the CPU affinity)

1. Create a CPU set mask
2. Clear the mask so that it contains no CPUs
3. Set the mask to the current CPU ID
4. If the CPU affinity is not set
   1. Print out an error
   2. Exit

Generate data node function

1. Create a node pointer
2. Allocate size for the node pointer
3. If the pointer’s memory was allocated
   1. Set next to null
4. Else
   1. Print out error message
5. Return the pointer

Producer thread function

1. Run the bind thread to cpu function
2. Create a node pointer and temp pointer
3. Create an integer counter
4. While the counter is less than K
   1. Jump to generate\_data\_node to create node
   2. If node creation was successful
      1. While true
         1. If the lock is acquired
            1. Set the node pointer’s data to 1
      2. If the linked list is empty
         1. Set the list header and tail equal to the pointer
      3. Else
         1. Put the new node at the end of the list
         2. Update tail to be new node
      4. Unlock the thread
      5. Break from the loop
5. Increment counter

Main function

1. Create integer variables for counter I and number of threads
2. Create an integer variable for the current number of CPUs
3. Create an integer pointer for an empty array of CPUs
4. Create a temporary and next node pointer
5. Create variables for a start and end time
6. If the user did not input anything
   1. Return an error message
   2. Exit
7. Set the number of threads equal to user’s input
8. Create a pthread array with num of elements being the num of threads
9. Get the current number of CPUs
10. If there are CPUs present
    1. Allocate memory for the cpu\_array
    2. If memory for the array was not allocated
       1. Print error message
       2. Exit
    3. Else
       1. For loop from I to the number of CPUs
          1. Set the CPU number I to the current CPU\_array index
11. Initialize a mutex
12. Allocate size for the list
13. If memory for list was not allocated
    1. Print end message
    2. Exit
14. Initialize linked list
15. Get the program’s start time
16. For loop from i to the number of threads
    1. Create a pthread and have that thread run through the producer thread function
17. For loop from i to the number of threads
    1. If the thread does not equal 0
       1. Wait for the thread to terminate
18. Get the finish time
19. If the linked list is not empty
    1. Set next and temporary pointer equal to the head of the list
    2. While temporary pointer is not null
       1. Clear and free every node from the linked list
20. If the cpu\_array is not empty
    1. Free memory from the cpu\_array
21. Calculate total runtime

# Conclusion 1- 2 Paragraphs

Part 1 took a bit of time to figure out. Since we did not fully understand how pthreads worked, we spent some time researching different pthread functions. We also had some trouble separating the string amongst the threads. Brogan figured out how to separate the string so that each thread could find substrings. He also implemented the for loop for the substrings. Jaylen wrote the loops to create the pthreads, the loop to wait for the thread to terminate, and added some comments for better understanding of the code.

Part 2 involved some experimentation for accessing the buffer properly and incrementing where to add or take from next. After careful balance with a mutex and condition variables, it was possible to take from the buffer in the right order and not run into any index out of bound errors. This part allowed Brogan to get more experience and research on developing with mutexes since only a description of the program was provided. The code could definitely be more optimized upon further review, but this prompt was a great introduction to multithreading.

Part 3 took a little while to figure out. Reading through all that code and trying to figure out how it all worked was a challenge. Eventually, Jaylen figured out that changing the pthread\_mutex\_trylock function to pthread\_mutex\_lock improved the performance of the program. We also noticed that performance was significantly more improved if there were more threads involved.

K=200

|  |  |  |
| --- | --- | --- |
| Threads | List avg | My-list avg |
| 2 | (548+445+415+320)/4 = 432 | (572+478+538+310)/4 = 474.5 |
| 4 | (1156+771+885+758)/4 = 892.5 | (685+912+717+654)/4 = 742 |
| 6 | (1500+1094+1134+1382)/4 = 1277.5 | (901+1014+1048+1076)/4 = 1009.75 |
| 8 | (1980+1369+1726+1447)/4 = 1630.5 | (1387+1538+1849+1490)/4 = 1566 |

K=400

|  |  |  |
| --- | --- | --- |
| Threads | List avg | My-list avg |
| 2 | (662+408+447+516)/4 = 508.25 | (572+607+478+589)/4 = 561.5 |
| 4 | (1778+1401+1715+1621)/4 = 1628.75 | (1509+1447+951+884)/4 = 1197.75 |
| 6 | (1501+1380+1923+2387)/4 = 1797.75 | (1303+1726+1173+1289)/4 = 1372.75 |
| 8 | (2088+2474+3526+2510)/4 = 2649.5 | (1572+1638+1592+2161)/4 = 1740.75 |

K=600

|  |  |  |
| --- | --- | --- |
| Threads | List avg | My-list avg |
| 2 | (957+794+1068+1009)/4 = 957 | (633+727+620+645)/4 = 656.25 |
| 4 | (2536+2038+2453+2445)/4 = 2368 | (1271+1370+1065+1207)/4 = 1228.25 |
| 6 | (2898+2989+2645+2736)/4 = 2817 | (1658+1707+1690+1612)/4 = 1666.75 |
| 8 | (3634+2682+3307+3233)/4 = 3214 | (2875+3092+2316+1837)/4= 2530 |

In the modified program, the only thing that was changed was using pthread\_mutex\_lock instead of pthread\_mutex\_trylock. At lower K values and thread counts, using trylock was typically faster than using lock. However, as K and the thread count increased, the performance of lock improved as well. The reason that pthread\_mutex\_lock decreases performance time over pthread\_mutex\_trylock is because pthread\_mutex\_trylock is a nonblocking function. If the mutex is currently locked, it will return immediately. However, if a mutex is currently locked with pthread\_mutex\_lock, the calling thread will be blocked until it becomes available. Since the thread is completely blocked until the lock becomes available instead of running in the background until the lock becomes available, that makes pthread\_mutex\_lock more efficient than pthread\_mutex\_trylock.

**Describe in which ways the project could have improved? Do you feel as though you put forth your best efforts? Did your team function well as a group? If not, please share in which ways with the instructor, not in the conclusion of this report. Students who do not participate in group projects may submit a single submission that they created by themselves – it cannot be the same submission the group you are assigned to submitted if you did not participate in the group work.**

We believe that our best efforts were put forth especially since this was the first time using these tools in development for both members of our group. Our team functioned well, assisting each other with issues that arose and completing an even split of the work. Given more time we could probably find some bugs and optimizations but we are proud of the final result.

# Lessons Learned

You are not confined to the following prompts, they are examples of the types of questions you might ponder while determining what your lessons learned were. Each team member can submit a separate “Lessons Learned” section to the dropbox in Canvas, or each student can paste a paragraph response in this document. Were you able to complete the project basically the first time through with little to no errors? Describe how you utilized the hints (if you needed to)? Describe which part of the project you learned the most from, what you learned, and how you feel it could benefit you someday in the future. Can you relate it to a potential career you may choose? How did completing this project contribute to your knowledge of how computing and OS (in general) behave? You may also add general take-aways about the assignment itself, if you would like. You may consider addressing: do you feel the assignment was about the appropriate breadth and depth you would expect from a higher level CS course?

Jaylen

Throughout this project, I learned how threads are applied to programming. I learned how to create a thread, lock a thread, and wait for a thread to terminate in C. I also learned the difference between pthread\_mutex\_trylock and pthread\_mutex\_lock. Lock is faster than trylock because if the mutex object is locked, then the thread will be blocked until the lock is available. This is different from trylock, where if the mutex object is locked, the thread will return immediately. This project was a lot of trial and error. Even if I didn’t do everything perfectly, I still learned a lot from this project. The most difficult part was figuring out how to get the for loop for substrings to work with threads. Brogan really helped me out with that part. I also realized that I should really start writing pseudocode again. I got so comfortable just writing code that when I needed to think more in-depth, I got stuck. Writing pseudocode a little bit earlier in the project would have saved me a lot of time.

# Brogan

This project exposed me to multithreading in c, something I had only briefly experimented with in python. I not only can see the vast benefits of utilizing multiple threads but also can see where it might be necessary in real time systems. Although a difficult tool to use in development, it can save notable time and shows off how much a programmer can control their kernel. This was a different process than writing programs usually goes for me but the documentation out there for it was extremely useful. Wrapping your head around how mutexes and pthreads work can be a difficult process, but breaking it up into small chunks of pseudocode helped. This project felt up to spec for an advanced CS course, adding to our repertoire of skills. It was extremely satisfying once the threads all worked together properly and hopefully in my future I have more reason to practice working with multithreading.