CS 4500 Operating Systems
Section 001
Project #3
POSIX Thread Programming

Chase Bauer, Sean Campbell, Phil Hilt
15 November 2022

Statement: **“**We have neither given nor received unauthorized assistance on this work.”

1 Virtual Machine Information

Directory and Name of Virtual Machine: CS4500/cbauer2/cbauer VM
Password for VM Account[1]: password
Path to Source Code: /home/cbauer/project3

2 Description

2.1 How We Solved the Problems

Task 1

The num_substring function from sequential_substring.c was modified to take an input i and the first for loop was removed from the function. The first loop in the sequential substring function iterated n1-n2 times to find the substrings in the text, where n1 and n2 are the lengths of the input strings. In the pthread implementation, the number of threads to create equals n1-n2 so that the first for loop can be replaced with threads. The calculations performed by each loop in the sequential function are independent, so this allows these calculations to be performed in parallel. The function now increments a global variable, total, instead of returning the total through the function. After all the threads are finished the threads a joined back together with pthread_join and then the total number of substrings are printed out.

Task 2

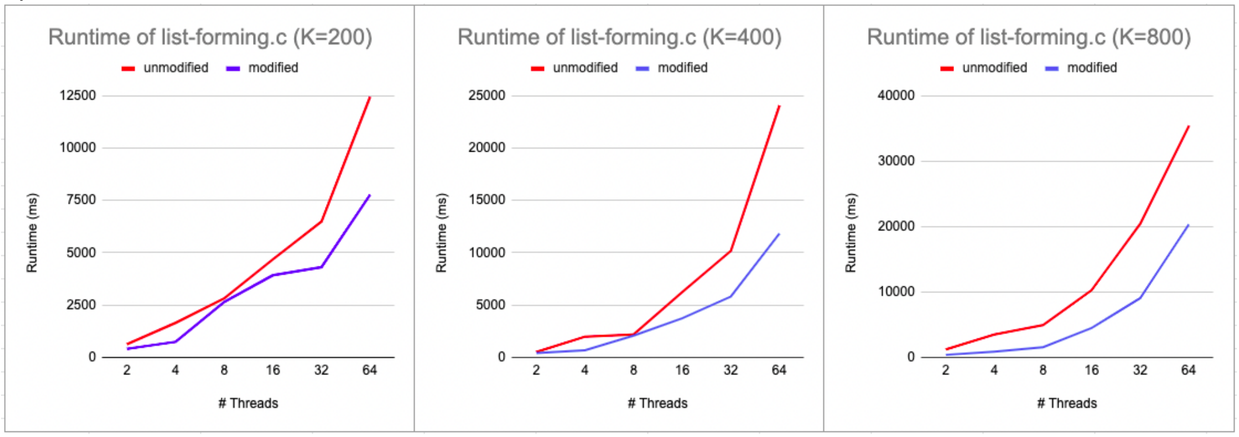
pthread_producer_consumer.c works by creating two threads; a producer thread and a consumer thread. These two threads use a shared mutex to lock and unlock in-order to retain mutual exclusion. Also, these threads have their own condition variable consumer_cond and producer_cond which are used to wait and signal the threads. The consumer thread works by first, locking the mutex, checking if it needs to sleep itself when there are no new characters to output, when the thread does not wait it will loop until it runs out of new characters and the producer thread needs to be signalled. Then, the consumer unlocks the mutex. The producer thread works by locking the mutex, then checking if it needs to wait when there is no more space for more characters in the queue. If it does not wait, the producer thread will loop, adding characters to the queue one by one from message.txt until the consumer thread needs to be signaled or until it runs out of characters from the file. These two threads alternate running and sleeping until every character has been read from the file and those characters have been written to stdout. Then the threads are exited, all pthread variables are destroyed, and the queue memory is cleaned up.

Task 3

For this task, the VM settings had to be changed to increase the number of CPUs to 4. Since multiple CPUs were now available it made sense to let threads run on all CPUs so that caching could be used. To do this, the function bind_thread_to_cpu wasn't used and instead a struct named thread_data was difined that included a CPU id and thread list as members. Then

[1] Depending on the project requirement, you may need to provide the password of a regular user or the root user.

instead of a while true loop that broke once a thread could enter and leave the critical region, a for loop moved through the number of processes. By using a for loop, pthread_mutex_lock could be used on each iteration instead of pthread_mutex_trylock to save resources. Once the process entered the critical region, the global list was updated and then pthread_mutex_unlock was used to exit the critical region. The modified design proved to be faster because as the number of threads and the value for K increased, the runtime of the new design was considerable faster than the original design. This is evident in the graphs below which show sample runs of each method with 2, 4, 8, 16, 32, and 64 threads and 200, 400, and 800 operation values K.



2.2 What We learned

For task1, we had to convert a program that was performing a task sequentially that would perform the same task with a threaded approach. To solve this task, we had to learn about implementing pthreads in C and how they can be effective.

In task 2, we learned how to use mutex variables and condition variables. Within task 2 a mutex variable was used to achieve mutual exclusion between the consumer and producer thread. Also, two condition variables were used to cause the threads to wait and to cause the threads to be signaled. In tasks 2 we learned how to control threads that need to work in the same memory with the mutex and how to handle intricacies within the wait and signal conditions to not get stuck in an infinite loop of waiting and signaling.

Task 3 illustrated how multithreading can be made more efficient in multiple CPU systems by utilizing all CPUs and taking advantage of caching. Using this method, a global list containing each pthread can be traversed executing each process by assigning it to a CPU and ensuring that with each loop iteration the pthread enters the critical region with mutex lock so that CPU cycles are not wasted. My modifying the original program this way, the runtime was drastically improved as the number of threads and operations increased.

For task 3,

2.3 How to Run Our Code

To execute and compile Task 1 compile pthread_substring.c with ‘gcc pthread_substring.c -o pthread_substring –pthread –D GNU_SOURCE’. The program is tested with the file strings.txt.

Task 2 can be run by compiling pthread_producer_consumer.c with the command gcc pthread_producer_consumer.c -o pthread_producer_consumer -pthread -D GNU SOURCE. Then the program can be executed using the command ./pthread_producer_consumer. In order to change what is output by the program you can edit the contents of message.txt.

Task 3 can be run and tested by compiling the programs list-forming.c and my-list-forming.c with the command:

`gcc list-forming.c -o list-forming -pthread -D GNU SOURCE`

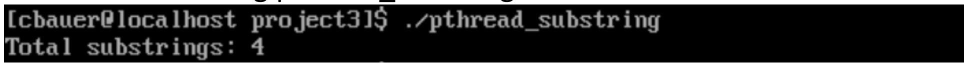
To compile the modified design, run the same command but replace instances of ‘list-forming’ with ‘my-list-forming'. Run each program with their respective executables with the same thread arguments to see the runtime results. Currently K is set to 800 in both versions of the program.

3 Screenshots of Output

3.1 Code Output Screenshots

Task 1

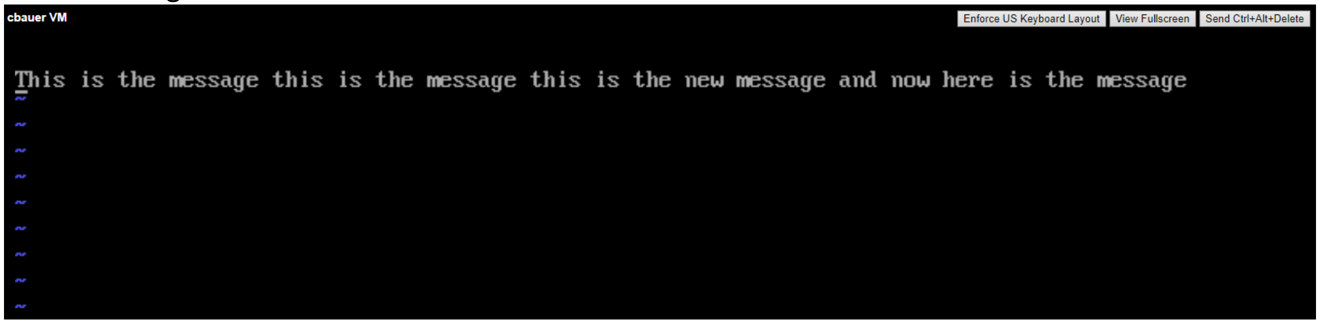
Stdout after running pthread_substring:



```
[cbauer@localhost project3]$ ./pthread_substring
Total substrings: 4
```

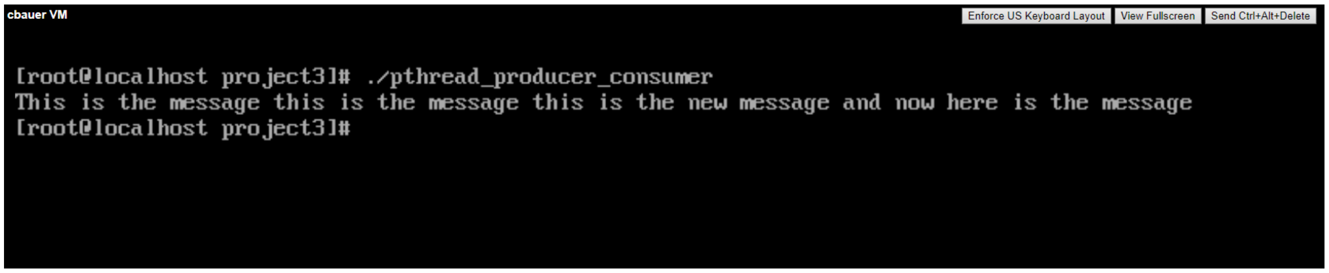
Task 2

Initial message.txt file:



```
This is the message this is the message this is the new message and now here is the message
~
~
~
~
~
~
~
~
~
```

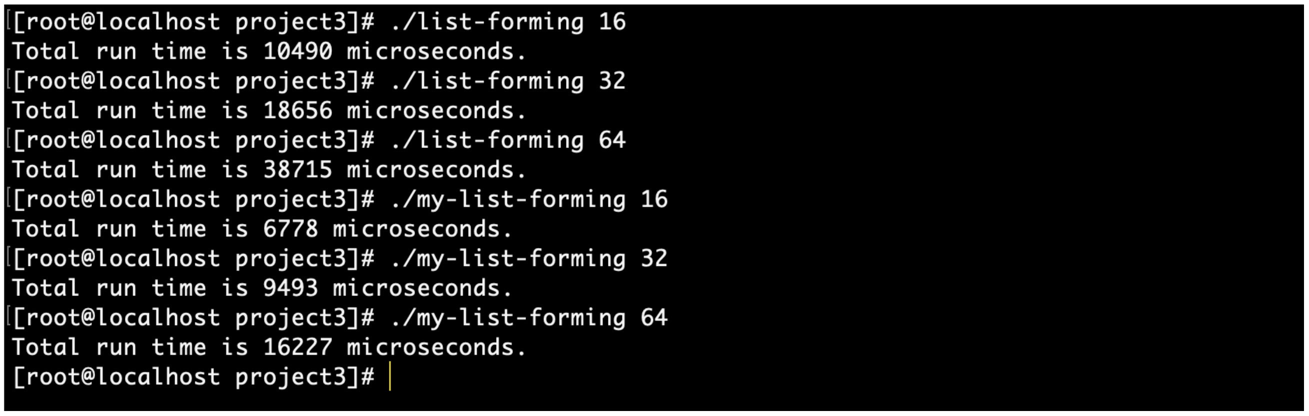
stdout Display after running pthread_producer_consumer:



```
[root@localhost project3]# ./pthread_producer_consumer
This is the message this is the message this is the new message and now here is the message
[root@localhost project3]#
```

Task 3

Sample output with K set to 800 and threads of 16 32 64:



```
[root@localhost project3]# ./list-forming 16
Total run time is 10490 microseconds.
[root@localhost project3]# ./list-forming 32
Total run time is 18656 microseconds.
[root@localhost project3]# ./list-forming 64
Total run time is 38715 microseconds.
[root@localhost project3]# ./my-list-forming 16
Total run time is 6778 microseconds.
[root@localhost project3]# ./my-list-forming 32
Total run time is 9493 microseconds.
[root@localhost project3]# ./my-list-forming 64
Total run time is 16227 microseconds.
[root@localhost project3]#
```