

Multi-threaded word count

32214362 Yoonseo Cho

Free days remaining: 5 / Used: 0

Contents

1. Introduction
2. Concepts
3. Implementation
 - a) 1 producer 1 consumer
 - b) 1 producer N consumer
 - c) N producer M consumer with K shared objects
4. Build
5. Lesson

Introduction

The last assignment, we dealt with process management(`fork()`, `execve()`). In this assignment, we are going to implement a N:M producer and consumer multi-threading program. The goal of the program is to read a text file with a series of producer threads, and write each separated line into a shared object. Then with consumer threads, generate statistics of the read input file.

Concepts

Multi-threading is a technique that allows a single program to execute multiple threads concurrently. A thread is the smallest unit of a process. Unlike processes, which run in separate memory spaces, threads share the same memory space. But, to be more specific, there are types of memory regions (heap, static) that can be shared across multiple threads. This enables sharing data across threads more efficiently compared to that of processes. However, it is still required to coordinate (synchronize) each processes/threads for data consistency as well as avoiding deadlocks.

For implementation, we would be using the library `pthread` for the Linux environment. To be more specific, we will be using `pthread_mutex` as well as `pthread_cond` for sending conditions across multiple threads.

Implementation

Goal 1) Correct the given code to implement 1:1

First of all, I had to fix the primitive *prod_cons.c* code. When I executed the program, I used the LICENSE file as an input and setted up to create with 1 producer and 1 consumer. However, it didn't work because the program couldn't handle the input file properly.

```
choyoonseo@joyunseoui-MacBookAir 2024-os-hw2 % ./a.out ./LICENSE 1 1
main continuing
Cons: 0 lines
main: consumer_0 joined with 0
Prod_6aef7000: 21 lines
main: producer_0 joined with 21
```

<Figure 1 - Failure of 1 producer 1 consumer>

Due to a race condition between the single producer and consumer, I implemented a mutex lock to avoid it. In other words, I synchronized both threads.

Before entering the critical section, I used a mutex lock to guarantee mutual exclusion.

`pthread_mutex_lock()` prevents other threads from entering the critical section. After locking the mutex, it is important to unlock it. `pthread_mutex_unlock()` allows other threads to enter the critical section. Without unlocking, other threads never get their turn, which can lead to deadlock.

Moreover, I used a condition variable because if I had only used a mutex, it would have been inevitable to experience busy waiting, which would waste CPU resources.

`pthread_cond_wait()` makes the thread remain blocked until another thread signals the condition variable using `pthread_cond_signal()` or `pthread_cond_broadcast()`. `pthread_cond_signal()` sends a signal which wakes up the waiting thread so it can continue processing.

It is important to initialize them before using those mutex and condition variables. I used `pthread_mutex_init()` and `pthread_cond_init()`.

After using them, it is crucial to destroy them. `pthread_mutex_destroy()` and `pthread_cond_destroy()` inform the operating system that we've finished using the mutex or condition variable, and help clean up kernel resources.

```

● choyoonseo@joyunseoui-MacBookAir 2024-os-hw2 % ./a.out ./LICENSE 1 1
main continuing
Cons_6b0c7000: [00:00] MIT License
Cons_6b0c7000: [01:01]
Cons_6b0c7000: [02:02] Copyright (c) 2019 mobile-os-dku-cis-mse
Cons_6b0c7000: [03:03]
Cons_6b0c7000: [04:04] Permission is hereby granted, free of charge, to any person obtaining a copy
Cons_6b0c7000: [05:05] of this software and associated documentation files (the "Software"), to deal
Cons_6b0c7000: [06:06] in the Software without restriction, including without limitation the rights
Cons_6b0c7000: [07:07] to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
Cons_6b0c7000: [08:08] copies of the Software, and to permit persons to whom the Software is
Cons_6b0c7000: [09:09] furnished to do so, subject to the following conditions:
Cons_6b0c7000: [10:10]
Cons_6b0c7000: [11:11] The above copyright notice and this permission notice shall be included in all
Cons_6b0c7000: [12:12] copies or substantial portions of the Software.
Cons_6b0c7000: [13:13]
Cons_6b0c7000: [14:14] THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
Cons_6b0c7000: [15:15] IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
Cons_6b0c7000: [16:16] FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
Cons_6b0c7000: [17:17] AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
Cons_6b0c7000: [18:18] LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
Cons_6b0c7000: [19:19] OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
Cons_6b0c7000: [20:20] SOFTWARE.
Prod_6b03b000: 21 lines
Cons: 21 lines
main: consumer_0 joined with 21
main: producer_0 joined with 21

```

<Figure 2 - Success of 1 producer 1 consumer>

Finally, it works!

Cons_6b037000 is the identifier of consumer thread, and Pros_6b03b000 is the identifier of producer thread. Cons_6b037000: [00:00] MIT License displays the work done by the consumer thread. As I have set up 1 consumer, this consumer reads every single line from the input file. Pros_6b03b000: 21 lines indicates that the producer thread has completed its task, which is reading from the input file. Cons: 21 lines shows that the consumer thread has finished its work. main: consumer_0 joined with 21 means that the main thread has completed synchronization with the consumer thread (consumer_0). main: producer_0 joined with 21 means that the main thread has completed synchronization with the producer thread (producer_0).

Goal 2) Add character statistics code

Now, let's count words! I added code to analyze and generate statistics for the string from *char_stat.c* to the consumer function.

```
main continuing
Prod_6fa4b000: 962 lines
Cons: 962 lines
*** print out distributions ***
#ch freq
11: 0
21: 962 *****
31: 0
41: 0
51: 0
61: 0
71: 0
81: 0
91: 0
101: 0
111: 0
121: 0
131: 0
141: 0
151: 0
161: 0
171: 0
181: 0
191: 0
201: 0
211: 0
221: 0
231: 0
241: 0
251: 0
261: 0
271: 0
281: 0
291: 0
301: 0
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
74 74 74 74 74 74 74 74 74 74 74 74 74 74 74 74 74 74 74 74
main: consumer_0 joined with 962
main: producer_0 joined with 962
```

<Figure 3 - Successful character statistics of 1 producer 1 consumer>

Yes it works!

To check the statistics, I created a simple input file that contains 74 instances of each letter of the alphabet.

Goal 3) Use FreeBSD9-orig.tar as an input

Until now, I checked that the program works well with a simple file like README.md or LICENSE. From now on, I'm going to use a 600MB size of FreeBSD9-orig.tar file as an input.

However, when I execute with the FreeBSD9-orig.tar file, the program stops due to the double free memory issue.

```
Cons_6d5b3000: [848553:848570] .\" 1. Redistributions of source code must retain the above copyright
Cons_6d5b3000: [848554:848571] .\" notice, this list of conditions and the following disclaimer.
Cons_6d5b3000: [848555:848572] .\" 2. Redistributions in binary form must reproduce the above copyright
Cons_6d5b3000: [848556:848573] .\" notice, this list of conditions and the following disclaimer in the
Cons_6d5b3000: [848557:848574] .\" documentation and/or other materials provided with the distribution.
Cons_6d5b3000: [848558:848575] .\"
Cons_6d5b3000: [848559:848576] .\" THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
Cons_6d5b3000: [848560:848577] .\" ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
Cons_6d5b3000: [848561:848578] .\" IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
Cons_6d5b3000: [848562:848579] .\" ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
Cons_6d5b3000: [848563:848580] .\" FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
Cons_6d5b3000: [848564:848581] .\" DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
a.out(38872,0x16d5b3000) malloc: Double free of object 0x132a04140
a.out(38872,0x16d5b3000) malloc: *** set a breakpoint in malloc_error_break to debug
Cons_6d5b3000: [848565:848581] zsh: abort ./a.out ./FreeBSD9-orig.tar 1 1
```

<Figure 4 - Double free issue with FreeBSD9-orig.tar in 1 producer 1 consumer>

It is due to the dangling pointer. To deal with this issue, I added `so-> line = NULL` after `free(so->line)` to prevent attempts to free memory that had already been freed. In other words, I set the pointer to NULL so that it no longer points to an invalid memory address.

```
Cons_6d37f000: [911583:911514] print SCRIPT "export CVSROOT=\"$mfclogin@ncvs.freebsd.org:/home/ncvs/\"\\n\\n";
Cons_6d37f000: [911584:911515]
Cons_6d37f000: [911585:911516] if (scalar(keys(%new_files)) or scalar(keys(%dead_files))) {
Cons_6d37f000: [911586:911517]     if (scalar(keys(%new_files))) {
Cons_6d37f000: [911587:911518]         print SCRIPT "cvs add";
Cons_6d37f000: [911588:911519]         print SCRIPT " \\n $." foreach (keys(%new_files));
Cons_6d37f000: [911589:911520]         print SCRIPT "\\n";
Cons_6d37f000: [911590:911521]     }
Cons_6d37f000: [911591:911522]     if (scalar(keys(%dead_files))) {
Cons_6d37f000: [911592:911523]         print SCRIPT "cvs rm -f";
Cons_6d37f000: [911593:911524]         print SCRIPT " \\n $." foreach (keys(%dead_files));
Cons_6d37f000: [911594:911525]         print SCRIPT "\\n";
Cons_6d37f000: [911595:911526]     }
Cons_6d37f000: [911596:911527] }
Cons_6d37f000: [911597:911528]
Cons_6d37f000: [911598:911529] print SCRIPT "cvs diff";
Cons_6d37f000: [911599:911530] print SCRIPT " \\n $." foreach (keys(%mfc_files));
Cons_6d37f000: [911600:911531] if ($diff = `diff`) {
Cons_6d37f000: [911601:911532]     print SCRIPT "\\n";
Cons_6d37f000: [911602:911533] } else {
Cons_6d37f000: [911603:911534]     print SCRIPT " | $diff";
Cons_6d37f000: [911604:911535] }
Cons_6d37f000: [911605:911536]
Cons_6d37f000: [911606:911537] print SCRIPT "cvs ci";
Cons_6d37f000: [911607:911538] print SCRIPT " \\n $." foreach (keys(%mfc_files));
Cons_6d37f000: [911608:911539]
Cons_6d37f000: [911609:911540] *** print out distributions ***
#ch freq
[ 1]: 307855 *****
[ 2]: 346979 *****
[ 3]: 272315 *****
[ 4]: 259126 *****
[ 5]: 159355 *****
[ 6]: 215351 *****
[ 7]: 130111 *****
[ 8]: 112050 *****
[ 9]: 68798 *****
[10]: 62598 *****
[11]: 38323 *****
[12]: 27384 *****
[13]: 26442 *****
[14]: 17938 *****
[15]: 13178 *****
[16]: 10372 *****
[17]: 8888 *****
[18]: 7370 *****
[19]: 5982 *****
[20]: 5026 *****
[21]: 4208 *****
[22]: 3554 *****
[23]: 2523 *****
[24]: 2474 *****
[25]: 2289 *****
[26]: 2022 *****
[27]: 1352 *****
[28]: 1171 *****
[29]: 907 *****
[30]: 11295 *****
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
586083 131248 388863 352693 823337 380474 135450 206443 626077 8530 53886 339358 216406 526859 440394 285760 19728 547382 565789 699184 257310 94984 77179 58895 98792 17175
main: consumer_0 joined with 911528
```

<Figure 5 - Deadlock issue with FreeBSD9-orig.tar in 1 producer 1 consumer>

Right after solving the double free issue, a deadlock issue arose.

The program stopped after joining with the consumer thread. It seems like the producer thread cannot join with the main thread. It keeps waiting.

I thought it was interrupted by some other issue temporarily. So I forced the program to quit, and re-executed it 5 times. But every time I executed it, the total number of lines processed by the consumer thread was different.

So I asked ChatGPT why it is so.

It was because of a spurious wakeup.

From a software perspective, the code should execute normally, but the thread that should not wake up can wake up unexpectedly due to characteristics of the operating system or hardware. So I replaced the if statement with a while loop when I check whether the buffer is full or not in producer and consumer functions.

```
Cons_6f70f000: [21262911:21262911] Cons_6f70f000: [21262912:21262912] # $FreeBSD: release/9.0.0/bin/cat/Makefile 198148 2009-10-15 18:17:29Z ru $
Cons_6f70f000: [21262913:21262913]
Cons_6f70f000: [21262914:21262914] PROG= cat
Cons_6f70f000: [21262915:21262915]
Cons_6f70f000: [21262916:21262916] -include <bsd.prog.mk>
Cons_6f70f000: [21262917:21262917] Prod_6f683000: 21262918 Lines
Cons: 21262918 Lines
*** print out distributions ***
#ch fresa
[ 1]: 9077967 *****
[ 2]: 245746 *****
[ 3]: 5472711 *****
[ 4]: 5700209 *****
[ 5]: 2000375 ****
[ 6]: 7692940 *****
[ 7]: 2824641 ****
[ 8]: 2305460 ***
[ 9]: 1552922 **
[10]: 2170713 **
[11]: 808332 *
[12]: 806050 *
[13]: 706501 *
[14]: 596018
[15]: 526804
[16]: 433856
[17]: 358912
[18]: 354497
[19]: 268007
[20]: 280063
[21]: 201742
[22]: 164014
[23]: 145650
[24]: 125062
[25]: 104175
[26]: 82598
[27]: 70950
[28]: 64521
[29]: 52127
[30]: 50092
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
14239598 4972835 11303608 9549271 22447966 7962974 3715946 4591418 14220229 344062 1483389 8150586 5628639 12350324 10792819 6635469 512006 12521644 13178317 16491358 6070339 2637484 1777558 7532456 2215212 510983
main: consumer_0 joined with 21262918
main: producer_0 joined with 21262918
```

<Figure 6 - Success with FreeBSD9-orig.tar in 1 producer 1 consumer>

Finally it works!

Although it took more than 5 minutes to execute, it worked!

```
./a.out FreeBSD9-orig.tar 1 1 70.16s user 198.04s system 66% cpu 6:43.85 total
```

<Figure 7 - Execution time with FreeBSD9-orig.tar in 1 producer 1 consumer>

I used the `time` command to measure the execution time.

It took 70.16 seconds in user mode, which represents the actual program code execution time, and 198.04 seconds in system mode, which represents time spent on system calls like

file I/O and memory management. The wall clock time, which is the total actual time taken from program start to finish was 6 minutes 43.85 seconds with a CPU utilization of 66%.

```
./a.out ./FreeBSD9-orig.tar 1 1 37.56s user 81.78s system 100% cpu 1:58.48 total
```

<Figure 8 - Reduced time with FreeBSD9-orig.tar in 1 producer 1 consumer>

It took so much time, so I changed the order of `pthread_mutex_unlock()` and `pthread_cond_signal()` to reduce context switching and decrease execution time.

Also, I commented out unnecessary `printf()` statements.

As a result, the total execution time was reduced to $\frac{1}{3}$ of the original time.

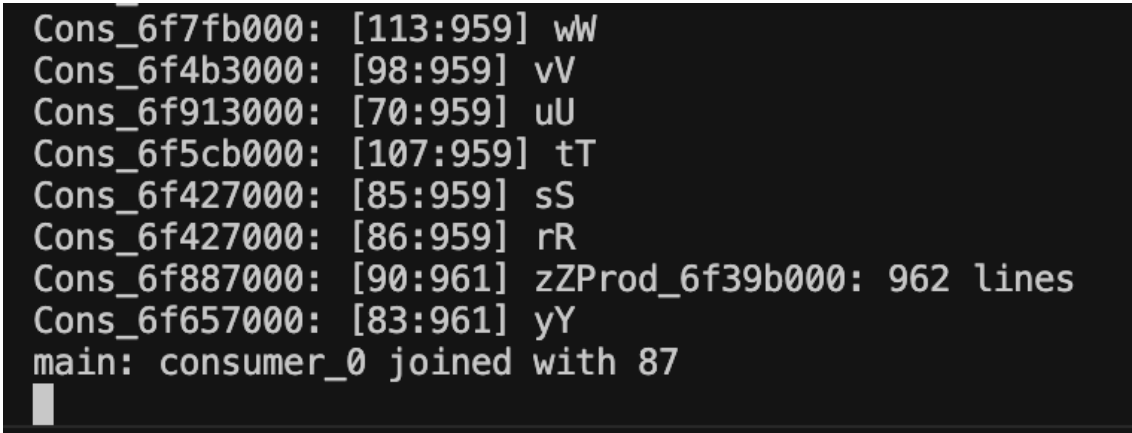
Goal 4) Implement 1:N from 1:1

Let's enhance the program so that it can execute with multiple consumers.

When handling 1 producer 1 consumer, the producer only reads a single line from the input file, and the shared object(buffer) can only hold a single line. However, when handling multiple consumers, the producer should read multiple lines from the input file and put them into the buffer.

So, I changed several things in the `shared_object` structure.

First, I changed a single pointer to a double pointer to store multiple strings. Second, I distinguished the condition variable like `thread_cond_t has_data` and `pthread_cond_t has_space` to better manage the buffer. Third, I used a `done` flag and a `count` variable instead of `full`, because the buffer contains multiple lines, not just a single line, and to clearly indicate whether the producer has finished reading the entire input file.



```
Cons_6f7fb000: [113:959] wW
Cons_6f4b3000: [98:959] vV
Cons_6f913000: [70:959] uU
Cons_6f5cb000: [107:959] tT
Cons_6f427000: [85:959] sS
Cons_6f427000: [86:959] rR
Cons_6f887000: [90:961] zZProd_6f39b000: 962 lines
Cons_6f657000: [83:961] yY
main: consumer_0 joined with 87
```

<Figure 9 - Deadlock issue with input file in 1 producer 10 consumer>

However, the deadlock issue occurred.

Some consumers might have been busy waiting because, before the producer finished, it only woke a single consumer using `pthread_cond_signal()`.

So I changed it to `pthread_cond_broadcast()` to wake up every consumer thread.

Since a single consumer changed to multiple consumers, the producer needs to take care of all the threads.

```

Cons_6b815000: [81:955] rK
Cons_6b97f000: [109:954] sS
Cons_6b7db000: [101:961] zZProd_6b51f000: 962 lines
Cons_6b7db000: [102:961] yY
Cons_6b5ab000: [117:961] xX
Cons_6b867000: [92:961] wW
Cons_6b867000: [93:961] vV
Cons_6b6c3000: [80:961] uU
Cons_6b74f000: [91:961] tT
main: consumer_0 joined with 118
main: consumer_1 joined with 70
main: consumer_2 joined with 81
main: consumer_3 joined with 92
main: consumer_4 joined with 103
main: consumer_5 joined with 94
main: consumer_6 joined with 82
main: consumer_7 joined with 110
main: consumer_8 joined with 106
main: consumer_9 joined with 106
main: producer_0 joined with 962
./a.out ./input 1 10 0.00s user 0.04s system 87% cpu 0.048 total

```

<Figure 10 - Race condition with input file in 1 producer 10 consumer>

Now, the multi-threaded execution works well, but another issue occurred.

The problem was that the character statistics code exists within the consumer function.

Since multiple consumers are running, the character statistics should be calculated after all of the consumer threads have completed.

Therefore, I moved the character statistics code into a separate function.

```

choyoonseo@junseo-MacBookAir: 2024-05-hw2 % ./a.out FreeBSD9-orig.tar 1 10
main continuing
Prod_6b8ef000: 21262910 Lines
Cons_6bb1f000: 2127988 Lines
Cons_6bc37000: 2111483 Lines
Cons_6bba000: 2148335 Lines
Cons_6bcc3000: 2118018 Lines
Cons_6b97000: 2064722 Lines
Cons_6bdc000: 2141987 Lines
Cons_6ba07000: 2141253 Lines
main: consumer_0 joined with 2064722
Cons_6b667000: 2155812 Lines
main: consumer_1 joined with 2141253
Cons_6b933000: 2136361 Lines
Cons_6bd4f000: 2133119 Lines
main: consumer_2 joined with 2136361
main: consumer_3 joined with 2127988
main: consumer_4 joined with 2148335
main: consumer_5 joined with 2111483
main: consumer_6 joined with 2118018
main: consumer_7 joined with 2133119
main: consumer_8 joined with 2141987
main: consumer_9 joined with 2155812
main: producer_0 joined with 21262910
*** print out distributions ***
#ch freq
[ 1]: 907507 *****
[ 2]: 7245746 *****
[ 3]: 5473711 *****
[ 4]: 5780209 *****
[ 5]: 3208375 ****
[ 6]: 7692340 *****
[ 7]: 2824541 ***
[ 8]: 2585468 ***
[ 9]: 1552322 **
[10]: 2170713 ***
[11]: 988332 *
[12]: 806850 *
[13]: 706381 *
[14]: 596018
[15]: 516884
[16]: 453856
[17]: 358912
[18]: 354407
[19]: 268867
[20]: 288863
[21]: 207442
[22]: 164614
[23]: 145650
[24]: 125262
[25]: 104175
[26]: 82590
[27]: 70950
[28]: 64521
[29]: 52127
[30]: 508582
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
14239598 4972835 11383608 9549671 22447966 7962974 3715946 4591418 14228229 344862 1483389 8158596 5628639 12358324 10792819 6635469 512006 12521644 13178317 16491358 6070339 2637484 1777558 7532456 2215212 510983

```

<Figure 11 - Success with FreeBSD9-orig.tar in 1 producer 10 consumer>

Finally, it works!

As I used multiple consumers and commented out the print statements for the consumer threads' work, it only took around a minute. It is faster than 1 producer and 1 consumer.

But, there was a shocking thing for me to figure out.

```
./a.out ./FreeBSD9-orig.tar 1 10 50.27s user 348.18s system 498% cpu 1:19.94 total
```

<Figure 12 - Execution time in 1 producer 10 consumer>

```
./a.out ./FreeBSD9-orig.tar 1 100 126.24s user 2085.46s system 608% cpu 6:03.46 total
```

<Figure 13 - Execution time in 1 producer 100 consumer>

How come 1:100 is much slower than 1:10? Since it uses more threads to process, I thought it would finish quicker than 1:10. But it was not.

So, I asked chatGPT about why it was so.

The main reason was the lock contention, which means that consumer threads have to wait to access the single buffer. As the number of consumers increases, parallel processing becomes possible, which is why 1:10 is much faster than 1:1. However, if the number of threads becomes too large, the consumer threads end up waiting for a long time.

Additionally, since I am using only one producer, it cannot provide sufficient data for multiple consumers, which could also be a reason.

Goal 5) Implement N:M with K shared objects from 1:N

Let's change a single producer to multiple producers to make it faster. I implemented multiple shared objects with a circular queue. In other words, I replaced the buffer with a circular queue.

Unlike 1:N, the shared object stores a single line from the input file at a time.

For example, with 3 producers and 5 consumers using 5 shared objects, each producer reads a single line from the input file and puts it in the shared object(so) in the following sequence.

The first producer: $so[0] \rightarrow so[1] \rightarrow so[2] \rightarrow so[3] \rightarrow so[4]$

The second producer: $so[1] \rightarrow so[2] \rightarrow so[3] \rightarrow so[4] \rightarrow so[0]$

The third producer: $so[2] \rightarrow so[3] \rightarrow so[4] \rightarrow so[0] \rightarrow so[1]$

Also, each consumer reads a single line from the shared object in the following sequence.

The first consumer: $so[0] \rightarrow so[1] \rightarrow so[2] \rightarrow so[3] \rightarrow so[4]$

The second consumer: $so[1] \rightarrow so[2] \rightarrow so[3] \rightarrow so[4] \rightarrow so[0]$

The third consumer: $so[2] \rightarrow so[3] \rightarrow so[4] \rightarrow so[0] \rightarrow so[1]$

The fourth consumer: $so[3] \rightarrow so[4] \rightarrow so[0] \rightarrow so[1] \rightarrow so[2]$

The fifth consumer: $so[4] \rightarrow so[0] \rightarrow so[1] \rightarrow so[2] \rightarrow so[3]$

Also, as the number of producers reading the input file increases, I used a `file_lock` to avoid race condition. I also used a `global_done` variable to notify all threads(both producers and consumers) when the producer finish reading the input file. Additionally, I used a `global_done_lock` to ensure that only the producer who reads the last line of the

input file can change the flag. Finally, I used a `so_index_lock` to make sure that each producer and consumer uses a different shared object in each thread.

```
main: producer_72 joined with 284140
main: producer_73 joined with 282848
main: producer_74 joined with 222362
main: producer_75 joined with 285771
main: producer_76 joined with 218187
main: producer_77 joined with 212973
main: producer_78 joined with 222618
main: producer_79 joined with 218920
main: producer_80 joined with 218114
main: producer_81 joined with 282572
main: producer_82 joined with 218896
main: producer_83 joined with 286581
main: producer_84 joined with 217545
main: producer_85 joined with 285243
main: producer_86 joined with 288628
main: producer_87 joined with 218989
main: producer_88 joined with 231982
main: producer_89 joined with 288291
main: producer_90 joined with 284210
main: producer_91 joined with 283687
main: producer_92 joined with 158823
main: producer_93 joined with 288587
main: producer_94 joined with 238882
main: producer_95 joined with 225852
main: producer_96 joined with 212338
main: producer_97 joined with 213168
main: producer_98 joined with 216531
main: producer_99 joined with 218856
*** print out distributions ***
#ch freq
[ 1]: 9877967 *****
[ 2]: 1245746 *****
[ 3]: 5473711 *****
[ 4]: 3768280 *****
[ 5]: 3288375 ***
[ 6]: 7652940 *****
[ 7]: 8244641 ***
[ 8]: 2585468 ***
[ 9]: 1552922 **
[10]: 2178713 ***
[11]: 988332 *
[12]: 886458 *
[13]: 786391 *
[14]: 596818
[15]: 268884
[16]: 433456
[17]: 358912
[18]: 354497
[19]: 268867
[20]: 288863
[21]: 281742
[22]: 164614
[23]: 145638
[24]: 125452
[25]: 184175
[26]: 82598
[27]: 78958
[28]: 64521
[29]: 52127
[30]: 586582
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
14239598 497283 11381808 9549171 22447966 7962974 3715946 4591418 14228229 344862 1483389 8158586 5628639 12358324 18792819 6635469 512886 12521644 13178317 16491358 6878339 2637484 1777558 7532456 2215212 518983
./a.out FreeBSD9-orig.tar 100 100 67.97s user 434.92s system 656% cpu 1:16.60 total
```

<Figure 14 - Success with FreeBSD9-orig.tar in 100 producer 100 consumer>

As you can see, it took a minute and 16.60 seconds to execute, which is shorter than 1:1 or 1:N.

The performance did increase, but not as much as I expected. I thought that the performance would increase dramatically due to having multiple producers and multiple shared objects, but it wasn't. The reason was that the number of mutex locks increased. Acquiring and releasing mutex locks has high performance impact caused by frequent context switching penalty and overhead.

Build

environment: **visual studio code**

programming language: **C**

compile: `gcc Nprod_Mcons.c`

execute: `./a.out FreeBSD9-orig.tar 100 100`

Lesson

It was interesting to implement the concept of thread.

However, it was quite challenging because I've never used the pthread library before. While using functions from the library, ensuring proper synchronization between threads to avoid race conditions and deadlocks was one of the most difficult parts.

While implementing, it was surprising that the performance didn't dramatically improve from 1:1 to 1:N and from 1:N to N:M was surprising. Especially, the fact that acquiring and releasing mutex locks takes some time was one of the main reasons for only a slight, rather than dramatic, performance improvement from 1:N to N:M. According to Amdahl's Law, there is a limit to performance improvement, regardless of how much the processors are parallelized.