Multithreaded Word Count

32190984 Isu Kim

October 29, 2022

Left Free Days : 5

1 Index

1 Building

There are 4 directories in this `git`.

- `prod_cons_v1`: For 1 producer 1 consumer.
- `prod_cons_v2`: For 1 producer N consumers.
- `project`: For N producers, M consumers and K shared objects.
- `its_time_to_let_go`: For Beating performance.

For `prod_cons_v1` and `prod_cons_v2`, you can compile in following steps.

```
$ cd project
$ make all
```

Output 1: Using of `Makefile`

This will compile and generate two executable files named:

- `prod_cons_v1`
- `prod_cons_v2`

The usage is same as the original `prod_cons.c` that professor gave us.
For `project`, you need `CMake` to compile the project. You can achieve by following commands
You can compile the project using `CMake` by

```
$ cd project
$ mkdir build && cd build
$ cmake .. && cmake --build .
```

Output 2: Compilation using `CMake`

The result will give you an executable file named project.

2 Analysis on Original Code

The original code `prod_con.c` that professor gave us had problems. Let's compile it and run it with a short text file.

```
$ ./a ../LICENSE 1 1
main continuing
Cons_7c4ea640: [00:14]
Cons: 1 lines
Prod_7cceb640: 21 lines
main: consumer_0 joined with 1
main: producer_0 joined with 21
```

Output 3: Execution of `prod_con.c`

This situation happened: (The number is the sequence)

1 Producer reads a line from file and `stored` line into shared object.

1 Consumer tries to `read` and `update` the line that was in shared object.

2 Producer reads a line from file and `stored` line into shared object.

2 Consumer stops loop because the line stored in shared object was `NULL`. So, it just joined. (Sequence 1 occurred at the same time. Unless the producer thread stored data into shared object before consumer thread read and update the shared object, consumer thread will see shared object's line as `NULL`)

3 Producer reads a line from file and `stored` line into shared object.

4 Producer reads a line from file and `stored` line into shared object.

5 ...

6 Producer reads a line from file and `stored` line into shared object.

7 Producer finishes reading line from file since it was last line. Producer now joins.

This makes following results:

1. Producer thread successfully reads all 21 lines from file.
2. Consumer thread could not print all 21 lines (might have printed out one or two if consumer thread was lucky).

We need to fix his code so that it can work properly!

3 1 Producer & 1 Consumer

In HW2, our first goal is to correct code for 1 producer and 1 consumer. Since we have one producer thread and one consumer thread which are racing for shared object, race condition is inevitable.

To solve "1 producer & 1 consumer" problem, we need to synchronize both threads. Meaning that producer and consumer has to work together, not separated from each other. The idea that I came up with is as it follows:

Algorithm 1 1:1 Producer

```
while File not finished do
    Lock Mutex
    if sharedObject.full = 1 then
        Unlock Mutex, wait for signal        ▷ Signal when shared object was consumed
    end if
    sharedObject.full ← 1
    sharedObject.line ← line from file
    Unlock Mutex and send signal             ▷ Signal to consumer thread
end while
```

In short, the producer thread will check if the shared object was empty. If it was empty, it will read a line from file and store data into the shared object.

Algorithm 2 1:1 Consumer

```
while True do
    Lock Mutex
    if sharedObject.full = 0 then
        Unlock Mutex, wait for signal        ▷ Signal when shared object was produced
    end if
    sharedObject.full ← 0
    if sharedObject.line = NULL then                    ▷ The file ended
        Unlock Mutex and send signal                    ▷ Signal to producer thread
        Break loop
    end if
    Process sharedObject.line
    Unlock Mutex and send signal                        ▷ Signal to producer thread
end while
```

In short, the consumer thread will check if the shared object was full, it will process (in our case it will be `printf()`) read line.

Producer thread and consumer thread iterates this process until the file that we are reading is over. Our key is to make each thread wait for the other thread if the condition for current thread is not satisfied (in this case if shared object is empty or not). The logic that I came up with can be implemented in `C` with features from `pthread.h`.

pthread.h offers us following functions that come in handy.

1. pthread_mutex_lock: For locking mutex while using shared object.
2. pthread_mutex_unlock: For unlocking mutex while using shared object.
3. pthread_cond_wait: For waiting for condition to be satisfied. According to manual, this function will internally call pthread_mutex_unlock. After the condition was set, it will automatically call pthread_mutex_lock as well.
4. pthread_cond_signal: For sending signal at condition.
5. pthread_cond_broadcast: For sending signal to all threads at condition.

The implementation of code can be found from prod_cons_v1. that is attached with this document.

Since I would like to keep this document as clean as posssible, I will not attach code nor screenshot of the code snippet. If you would like to take a look at how I implemented the code, please take a look at prod_cons_v1.c.

Let's now compile and execute the code with same file named LICENSE (The file is basically a MIT license which has 21 lines).

```
$ ./prod_cons_v1 ../../LICENSE 1 1
main continuing
Cons_5254640: [00:00] MIT License
Cons_5254640: [01:01]
Cons_5254640: [02:02] Copyright (c) 2019 mobile-os-dku-cis-mse
...
Cons_5254640: [20:20] SOFTWARE.
Prod_5a55640: 21 lines
Cons: 21 lines
main: consumer_0 joined with 21
main: producer_0 joined with 21
```

Output 4: Successful Execution of prod_cons_v1.c

As we all can see, the execution was successful. It can read the file successfully without any missing lines or duplicated lines. Also each consumer thread and producer thread both joined with 21 lines which means they did the job they were supposed to do. Yay!

Now, let's try with 1 producer and 5 consumers with the same code

```
$ ./prod_cons_v1 ../../LICENSE 1 5
Cons_fbe7a640: [00:00] MIT License
Cons_fb679640: [00:01]
main continuing
Cons_fbe7a640: [01:02] Copyright (c) 2019 mobile-os-dku-cis-mse
Cons_f3fff640: [00:03]
free(): double free detected in tcache 2
Aborted (core dumped)
```

Output 5: Failure Execution of `prod_cons_v1.c`

Oops! It failed. Not only did it just failed, it also printed out some weird characters. (Unfortunately, due to my latex editor's problem, it could not compile broken characters, but there are some broken characters if you execute the code yourself). Also, it is mentioned that `free()` was called twice. Let's take a guess on what caused the failure.

In short, for locking a single shared object, race condition occurs between consumer threads and producer thread.

When producer thread finished and broadcasted signal to all waiting consumers threads, all consumer threads will fight for locking a single shared object. No matter which consumer gets the privilege to lock shared object, that consumer thread has to unlock shared object and send signal to producer thread that is waiting for the signal. Right at this moment everything gets messed up. The 4 remaining consumers that failed to lock shared object and producer thread will all fight for locking a single shared object and at this moment race condition for locking shared object occurs.

In this case, there are two possible cases:

a) Producer thread wins and locks shared object.

b) One of consumer threads wins and locks shared object.

Case a), the producer thread will read file and put data into the shared object. However, the mutex order is messed up already. (Since 4 other threads are waiting for their turn to lock shared object).

Case b), one of the consumer threads wins and locks shared object. It tries to perform what it has to do: read data from shared object, print it out, and `free()`. Since it is trying to print out a `free()`ed string, it prints out broken strings.

Also, not only this thread will print out a broken string, but it will also try to `free()` a string that has been already `free()`ed. It is well known fact that if some variable that was `free()`ed twice, it has potential to crash process and it is an *Undefined Behavior* in C. (Of course, the mutex order is messed up as well). Even if you do not `free()` the line that was processed, due to mutex order getting messed up, following thing happens.

```
$ ./prod_cons_v1 ../../LICENSE 1 5
Cons_83fff640: [02:04] Permission is hereby granted, ...
Cons_892c4640: [00:04] Permission is hereby granted, ...
Cons_8a2c6640: [01:04] Permission is hereby granted, ...
Cons_8aac7640: [04:04] Permission is hereby granted, ...
Cons_89ac5640: [01:04] Permission is hereby granted, ...
main continuing
...
```

Output 6: Failure Execution of `prod_cons_v1.c`

Since mutex order is messed up, duplicate lines get printed out and sum of all consumer's processed line exceeds total lines in the original file. In some cases, it even deadlocks. Since threads read same lines multiple times, the concurrency and synchronization is broken.

Therefore, no matter it was Case a) or Case b), result is far from accuracy. (If Case a) gets super lucky, there is very minuscule chance that it might get it correctly. Which is almost a miracle.)

So, we have another goal to solve single producer and multiple consumer problem: "Avoiding race condition between mutex locking". To do that, I came up with an idea of using two mutexes and two conditions. This topic will be explained in "1 Producer with N Consumers" problem. Which we will be looking at in the next page.

4 1 Producer & N Consumers

Our second goal is to solve "1 Producer & N consumers" problem. In order for us to solve the problem we need to prevent race condition between consumer threads. This means that producer will work just like it did, however there should be one more synchronization between those consumer threads. Since the pseudo-code and algorithm for producer does not change, I will show you the consumer thread part only.

Algorithm 3 1:N Consumer

```
while True do
    Lock consumer Mutex                          ▷ Avoid race condition between consumers
    Lock global Mutex
    if sharedObject.full = 0 then
        Unlock Mutex, wait for signal            ▷ Signal when shared object was produced
    end if
    sharedObject.full ← 0
    if sharedObject.line = NULL then                                    ▷ The file ended
        Unlock consumer Mutex
        Unlock global Mutex and send signal
        Break loop
    end if
    Process sharedObject.line
    Unlock consumer Mutex
    Unlock global Mutex and send signal                  ▷ Signal to producer thread
end while
```

Producer works just as it did, for consumer thread we need to refactor the code a bit. In this problem, our key is to make a single consumer thread active at a time by using mutex. We do not need to take care about which consumer thread is working at the moment, but we shall have just one consumer thread at a time. The logic that I came up with can be implemented in `C` with functions from `pthread.h`. I am going to use the same functions that `pthread.h` offers, however will be replacing `pthread_cond_signal` into `pthread_cond_broadcast` since there are multiple threads waiting for the condition to be met.

Also for implementation, we will be having two `pthread_mutex_t`s and one `pthread_cond_t` stored in shared object like below:

- `globalLock`: This is for mutex between producer and consumer threads.
- `consumerLock`: This is for condition between producer and consumer threads.
- `globalCond`: This is for mutex among consumer threads. This will prevent consumer threads racing into the shared object and messing up the mutex order.

This will prevent producer thread and consumer threads racing for locking mutex for a shared object at once. Since consumer thread shares a single consumer mutex and races for locking consumer mutex, only one consumer thread will be alive at the moment. When a consumer thread locked consumer mutex, there might be two cases:

a) The shared object is filled.

b) The shared object is not filled.

Case a) This will mean that producer thread stored something into the shared object, unlocked global mutex and broadcasted signal to consumer thread. In this case, the consumer thread will lock the global mutex and will access data from shared object, print the store line, `free()` data. When the consumer thread is done, it will unlock global mutex and consumer mutex as well as broadcasting signal to producer thread.

Case b) This will mean that the producer thread did not store something into the shared object. Thus, the consumer thread who won the race will wait for producer thread to finish putting data into the shared object. After that, same thing goes like Case a).

In either cases, everything will run successfully since:

1. Only one consumer thread is active at a time.
2. Only one thread (including all consumer threads and producer threads) has access to shared object.

This will not violate concurrency and keep our data alive!

The implementation of code can be found from `prod_cons_v2`. that is attached with this document.

Now, let's execute the program using `LICENSE` text file.

```
$ ./prod_cons_v2 ../../LICENSE 1 5
[+] Producer 0 created !
[+] Consumer 0 created !
[+] Consumer 1 created !
[+] Consumer 2 created !
[+] Consumer 3 created !
[+] Cons_a569a640: [00:00] MIT License
...
[+] Cons_9f7fe640: [14:20] SOFTWARE.
[+] consumer_0 joined with 6
[+] consumer_1 joined with 0
[+] consumer_2 joined with 0
[+] consumer_3 joined with 15
[+] consumer_4 joined with 0
[+] producer_0 joined with 21
```

Output 7: Successful Execution of `prod_cons_v2.c`

The execution of 1 producer with 5 consumers ended without any failure. However, there is some strange thing! There was total 21 lines in text file, however 17 of them were consumed by consumer 1 thread. Also, the performance seemed not to have improved at all.

Let's take a look at this problem and see if this 1 producer with N consumer problem was solved properly and whether this is an efficient solution or not.

In short, this problem occurs since only 1 consumer thread is alive at a time. No matter how many consumer threads were generated, all other threads except the one which locked the

consumer mutex will be “blocked”.

According to official document for `pthread_mutex_lock`, it is mentioned that *“If the mutex is already locked, the calling thread shall block until the mutex becomes available.”*. “Block” means that the thread is put in a wait (sleeping) state, like the picture below:

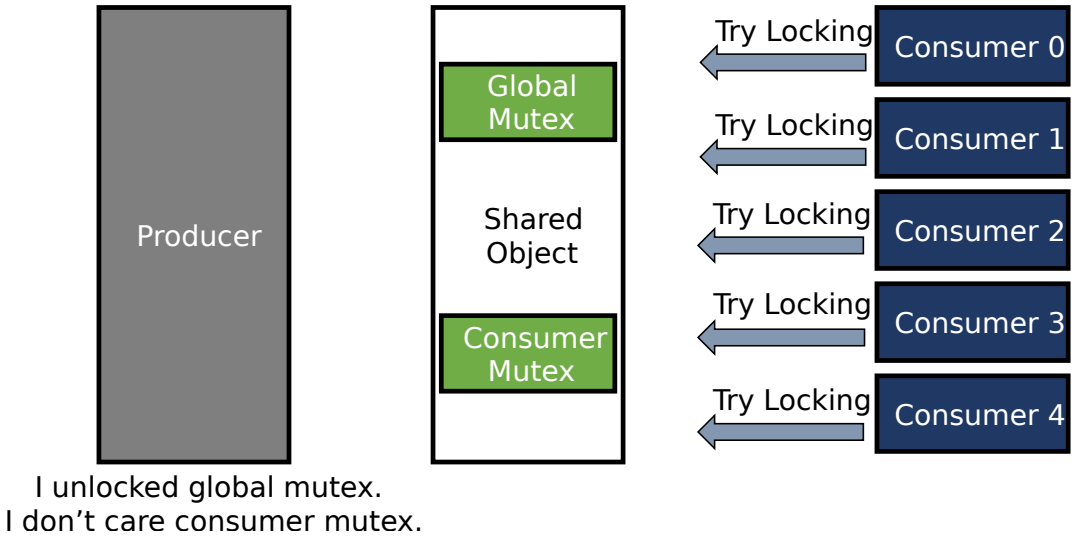

Figure 1: 1 Producer with N Consumers, 1 Shared Object

This will make only consumer thread active at a time when consumer thread has to be activated. This makes no difference between using a single consumer thread. Let’s assume that we implement multiple producers in the same as how we implemented, the following picture will describe this.

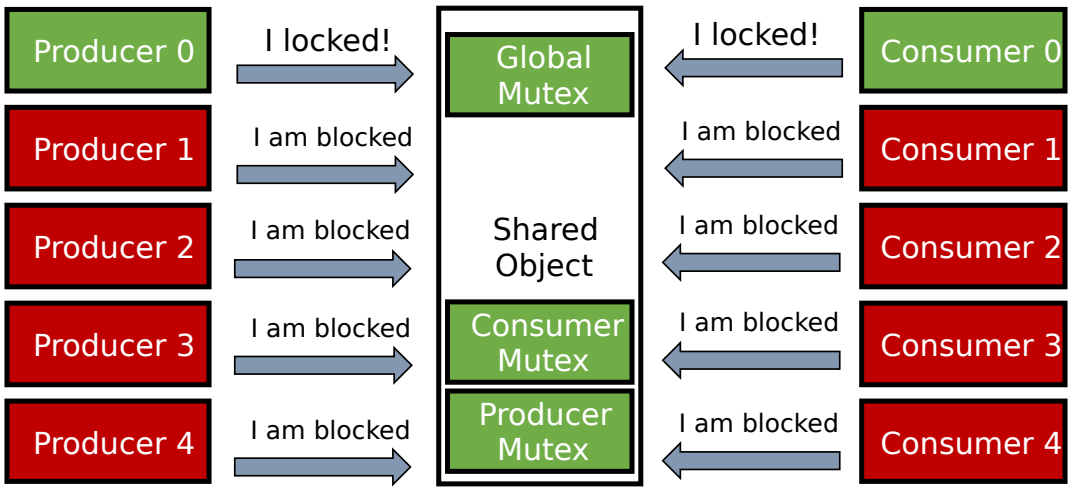

Figure 2: 1 Producer with N Consumers, 1 Shared Object

Therefore, even if we implement multiple producers and multiple consumers, with using just one single shared object, it makes no performance boost. Also, in theory this might be slower since the producer and consumers have their own mutexes to lock and unlock, while with single producer and single consumer there is only one mutex to lock and unlock.

So, implementing multiple producers in this way with single shared object seems pointless to me since this will make our multi-threaded program act as if it is just single threaded program. Therefore, I will not try to implement this at all.

In short, I can pretty much safely say that *"If we are using multiple producers and multiple consumers with one single shared object, this will make no performance difference from single producer and single consumer"*.

Then, how can we improve performance when using multiple consumers and multiple producers? I came up with the idea that we can improve performance with making multiple shared objects instead of single shared object.

Let's give it a shot. (Is continued in the next page)

5 N Producer & M Consumers with K Shared objects

Before we start this subject, I came across a fun picture in the web while implementing this project. I would like to share this picture since it explained lots of things that happened during implementing this project.

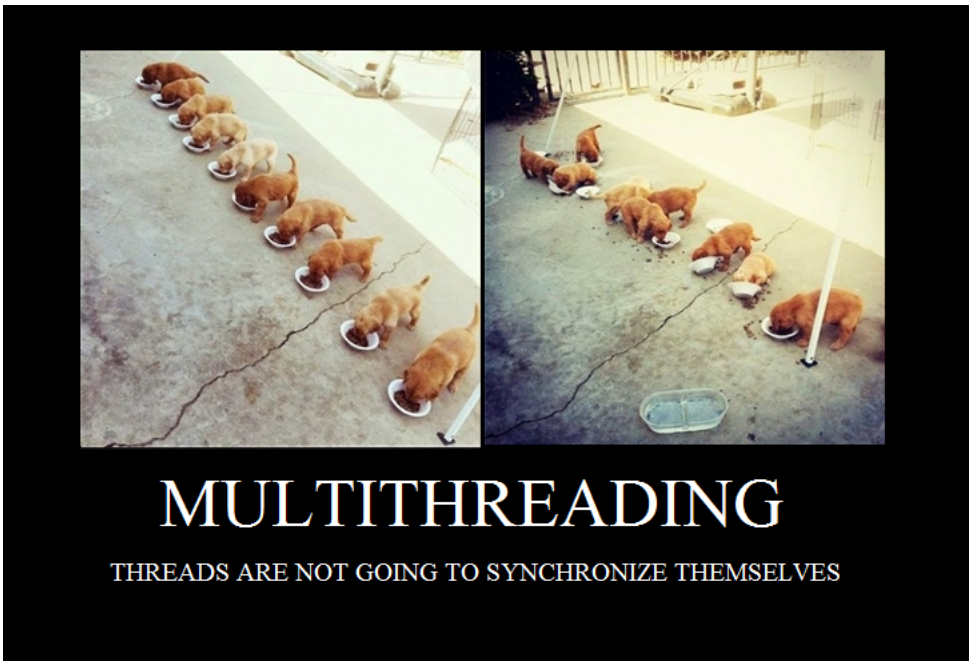

Figure 3: From https://kartikiyer.com/2019/06/16/synchronizing-multi-threaded-code-based-on-object-value/

In short, this implementation unfortunately failed. When I was implementing this project, I encountered countless:

- Dead locks
- Race conditions
- Messed up execution
- Messed up signals
- Desynchronization

Professor told us to fix a bit of part from the given source code `prod_cons.c`. However, in order for me to implement my idea into real `C` code, I had to reconstruct everything from scratch. Also, no semaphore was used. Everything was implemented with mutexes, signals, and conditions (which I came to regret a lot).

In short, the implementation kind of worked, but failed in aspect of reliability due to my lack of skills when it comes to concurrency. The implementation was painful and took almost two weeks, however it was super fun.

So, the problem that occurred from "1 Producer N consumer" was that if we had a single shared object, that would make no performance improvement at all since it will work as if it is a single thread process. Thus, I came up with methods of using multiple shared objects at once.

In order for me to achieve this goal, I came up with using handlers and worker threads. The basic idea is as it follows:

- Producer handler will tell producer worker to read line from file and put data in specific shared object.
- Consumer handler will tell consumer worker to retrieve data from specific shared object and process the string. (In our case it will be counting words)

To implement this, there should be 4 types of different threads.

- Producer handler ×1
- Consumer handler ×1
- Producer worker ×M
- Consumer worker ×N

In order for me to manage those threads, we need a precise control over all threads. Not to dead locks, no race conditions.

Following pseudo-code explains how each threads are designed and implemented.

Algorithm 4 Producer Handler

```
emptyIndex ← 0
curWorker ← 0
while File not finished do
    if All Shared objects full then
        Wait for signal                    ▷ Signal when shared object was consumed
        Continue                           ▷ Go back to the start of the loop
    end if
    emptyIndex ←Empty shared object        ▷ Assign job info
    curWorker ←Available worker
    Assign job to curWorker
end while
while Workers not over do
    Send signal to all worker threads      ▷ Wake sleeping workers
    Set empty shared objects as finished
end while
```

Algorithm 5 Producer Worker

```
readCount ← 0
line ← NULL
read ← 0
while True do
    Wait until handler's signal arrives
    line ←Read line from file
    read ←Read Result
    targetObject ←Assigned work from handler
    if Assigned empty job then
        Set state terminated
        Break loop
    end if
    if read = -1 then
        targetObject.line ← NULL                ▷ Let consumer know this was last
        targetObject.isFull ← −1
        Send signal targetObject.cond           ▷ Send signal to consumer worker
        Break loop
    end if
    targetObject.line ← line                    ▷ Store data
    targetObject.isFull ← 1
    Send signal targetObject.cond
    Set state available
    readCount ←readCount+1
end while
Return readCount
```

Algorithm 6 Consumer Handler

```
emptyIndex ← 0
curWorker ← 0
while Workers not over do
    if All Shared objects empty then
        Wait for signal          ▷ Signal when shared object was produced
        Continue                 ▷ Go back to the start of the loop
    end if
    emptyIndex ←Empty shared object          ▷ Assign job info
    curWorker ←Available worker
    Assign job to curWorker
end while
```

Algorithm 7 Consumer Worker

```
readCount ← 0
line ← NULL
read ← 0
while True do
    Wait until handler's signal arrives
    targetObject ←Assigned work from handler
    if targetObject.isFull = 0 then          ▷ Target object was not filled in yet
        Wait for signal                      ▷ Signal when shared object was filled
        Continue                             ▷ Go back at where it got signal
    end if
    if targetObject.line = NULL then         ▷ File was over
        Set worker state terminated
        Break loop
    end if
    line ← targetObject.line                 ▷ Retrieve data
    targetObject.isFull ← 0
    Process line
    Set state available
    readCount ←readCount+1
end while
Return readCount
```

Let's assume the following case:

- 1 Producer
- 1 Consumer
- 2 Shared Objects

The threads will work like this picture below:

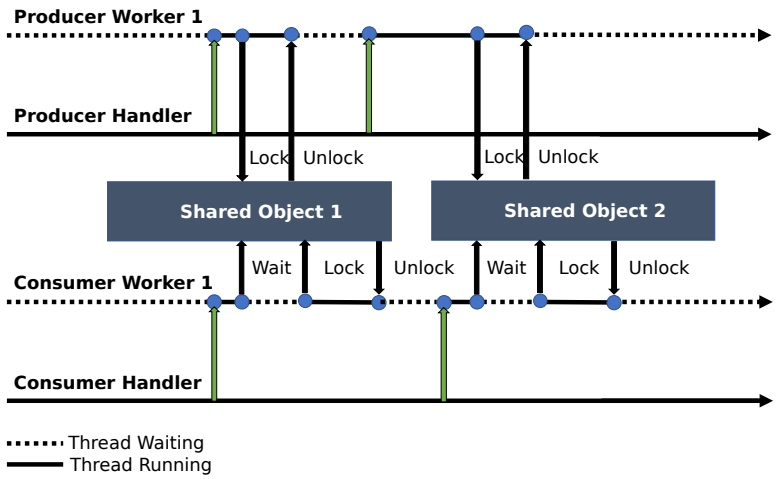

Figure 4: Thread and Time graph

In short, producer handler and consumer handler will assign workers for each shared object. Producer worker will lock shared object and write data into it. When it is done writing data into the shared object, it will unlock mutex. Consumer worker at the same time tried to lock shared object, however since it was not filled in yet, it will wait until producer worker finishes writing data. When producer worker finished writing data, consumer worker will lock mutex and access data. When it is done using shared object, it will release lock. This process goes on until the end of the file.

After the file was finished, each workers and handlers will join. Then the stats of the file retrieved by each consumer workers will be merged into one stat. Then print out the result of statistics of the file that was requested to analyze.

In order for us to achieve this, we need:

- A mutex between producer handler and consumer handler. Since they need to know shared object's status accurately, they need a mutex that avoids race condition on shared object's status
- Mutexes in shared objects. Since producer worker and consumer worker will use a same shared object, they need a mutex that avoids race condition on shared object's data. For example, consumer worker should not access shared object when producer worker was writing some data into it, vice versa.
- Conditions between each handlers and worker threads. Since handler thread needs to wake worker thread up and assign job to the worker thread, they need a condition that will wake worker thread up.
- Condition between producer handler and consumer workers, worker handler and producer workers. Since they need to know if the shared object was filled or not.

You can refer to the code in `project` directory. The code has following sources:

- `main.c`: Code for main function. This initializes all variables and starts threads.
- `common.c` & `common.h`: Code for some shared codes.
- `producer.c` & `producer.h`: Code for producer threads.
- `consumer.c` & `consumer.h`: Code for consumer threads.

You can execute project using following command.

```
$ ./project <readfile> #Producer #Consumer #SharedObjects
```

Output 8: Executing Project

Please be advised that *#SharedObjects* must be greater or equal to maximum between *#Producer* and *#Consumer*.

So, an example execution will be:

```
$ ./project ObsoleteFiles.incOld 10 10 20
```

Output 9: Example of Executing Project

This will execute 10 producer threads, 10 consumer threads with 20 shared objects for file named `ObsoleteFiles.incOld`. Let's execute with file named `ObsoleteFiles.incOld` from `FreeBSD9-orig.tar` (originally named `ObsoleteFiles.inc`)

```
$ ./project ObsoleteFiles.incOld 10 10 20
[+] Starting job...
[+] Reading file : ObsoleteFiles.incOld
[+] Job info : 10 Producers / 10 Consumers /  20 Shared objects
[+] If this process get stuck, kill it. ...
```

Output 10: Example of Executing Project

The process starts with 10 producers, 10 consumers and 10 shared objects. (If you executed the command and if it was stuck, please kill process using `SIGINT`. This has low chance of happening however it sometimes does. Unfortunately, I could not make the possibilities of my process getting into dead lock down to 0

```
[PH] Joined
[PW00] Read 614 lines
[PW01] Read 9 lines
[PW02] Read 619 lines
[PW03] Read 549 lines
...
[+] Producer read 5490 lines
...
[CW08] Processed 557 lines
[CW09] Processed 547 lines
[CH] Joined
[+] Consumer read 5490 lines
```

Output 11: Example of Executing Project

Consumer and producer handlers each has a mechanism that distributes workload to each worker thread. This might be inefficient since the handlers need to perform extra operation to select worker threads, however I just wanted to implement a kind of load balancing for each worker thread.

```
[+] Consumer read 5490 lines
*** print out distributions ***
  #ch  freq
[  1]:  539     *****
[  2]:  209     **
[  3]:  185     **
[  4]:  141     *
[  5]:  123     *
[  6]:  189     **
[  7]:  208     **
[  8]:  143     *
[  9]:  314     ***
...
[ 25]:   67
[ 26]:   75
[ 27]:   71
[ 28]:   93     *
[ 29]:  134     *
[ 30]: 4497     **************************************************
...
```

Output 12: Result of Executing Project

Then, this will print out the distribution of the content. The source code came from professor's `char_stat.c`. Modified a bit of that code and it worked like magic! The image was too tiny, so I will attach result file as `output.png`.

So, the question arises. Is this really faster than original `char_stat.c`? Unfortunately, the answer happened to be no. Let's see comparison of the result:

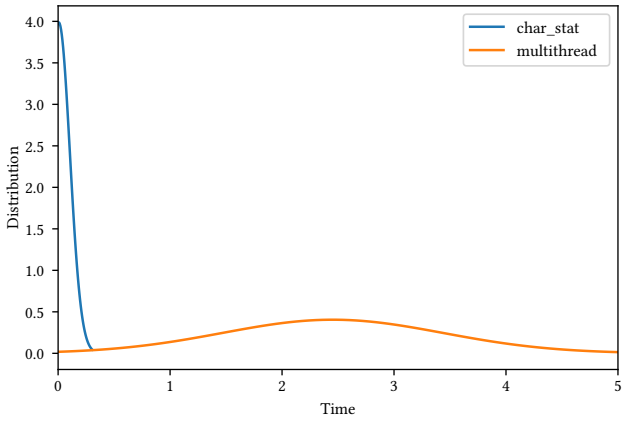

Figure 5: Standard Deviation of Each Results

Our project had average of 2.4522 seconds and standard deviation about 0.97. However, `char_stat.c` had average of 0.005 and had standard deviation about 0.01. So, sadly I have failed to beat the fastest execution time. Then, let's compare results in this process with different parameters.

The test was conducted with file named `ObsoleteFiles.incOld` from `FreeBSD9-orig.tar` (Originally named `ObsoleteFiles.inc`). Now, I am going to change some numbers of producers, consumers and shared objects. Now, let's see how it goes!

#Producers	#Consumers	#Shared Objects	Time (Sec)
10	10	10	3.051
10	10	20	1.048
10	20	20	1.045
20	10	20	1.041
20	20	20	1.047
30	20	30	1.050
30	30	30	1.048

Table 1: Execution Time by Arguments

Surprisingly, the results did not vary that much. Also, it had no consistency within those results as well. For example, the variance between each execution with same arguments varied so much:

Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Trial 10
3.043s	1.036s	DL	1.038s	5.043s	DL	1.038s	DL	2.040s	1.037s

Table 2: Execution Time for Each Trials

In the table, DL means dead lock. The process could not proceed since the execution got stuck.

In short, the implementation failed since there were too many dead locks and the result were not consistent: which were slower up to 5 times more than the fastest execution. However, I can definitely say one thing: “Multithreading might perform worse if programmed bad”. I was expecting “more producers, consumers and shared objects will be faster”. Which turned out to be not the case for this implementation.

While I was preparing for my mid-term exam, I kind of got smarter. So, I made a new way but simple way of solving this problem. Which I think can beat the maximum concurrency. If you are interested, please take a look at the next page.

6 Beating Max Performance (Yes it did)

So the basic idea of implementing this one was the following:

1. Use `fseek()` instead of using shared `getdelim()`.
2. Assign segments to the producer threads to concurrently read the file.
3. Merge all the word stats in the last moment.
4. Match a single producer thread and a consumer thread to use a single shared object.

Since I had so much things going on after the mid-term exam, I could not implement an N producers and M consumers method. This implementation is only limited to N producers and N consumers.

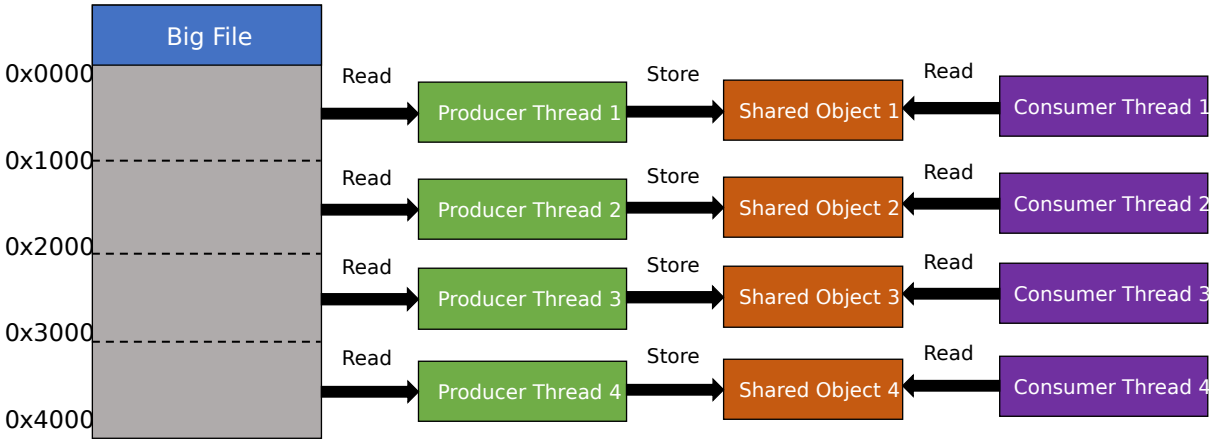

Figure 6: Implementation of Beating Performance

My idea works in this way:

1. When starting the program, main function will check how big the file is.
2. With that file size, it will try to make most evenly distributed segments as possible.
3. With those segments, each producer threads are assigned a segment to read from file.
4. When producer reads, it reads data by 4096 bytes. (Which is block size)
5. Producer stores the data into shared object with its corresponding consumer.
6. Consumer retrieves data from shared object and process word counter.

In this way, each threads just need one synchronization between producers and consumers. Meaning that the maximum concurrency can be met.

The source code can be found in `its_time_to_let_go` directory under git. You can try this one out by building project using following method:

```
$ mkdir build && cd build
$ cmake .. && cmake --build .
```

Output 13: Compiling Code

It will give you an executable file named `hw2_final`. You can execute the file by following command.

```
$ ./hw2_final ./FreeBSD09-orig.tar 100
```

Output 14: Example of executing code

This will execute word counter for file named `FreeBSD09-orig.tar` with 100 concurrent threads.

```
$ ./hw2_final ../FreeBSD9-orig.tar 100
[+] Total file size : 679818240 / Thread count : 100
[+] Producer 0 read 6818663 bytes
...
[+] Total : 679818240 / 682386432
*** print out distributions ***
  #ch  freq
[  1]: 381278   **************
[  2]: 321529   ************
[  3]: 245409   ********
[  4]: 265877   *********
[  5]: 142357   ****
...
```

Output 15: Example output

Let's check the execution time so that we can verify that this code is actually better than `char_stat.c` that professor gave us which runs in single thread.

```
$ time ./hw2_final ../FreeBSD9-orig.tar 100
real    0m0.371s
user    0m2.553s
sys     0m6.077s
...
```

Output 16: Execution Result

Meanwhile, `char_stat.c` which professor gave us had following result:

```
$ time ./char_stat ../FreeBSD9-orig.tar
...
real    0m2.980s
user    0m2.844s
sys     0m0.136s
```

Output 17: Execution Result

So, this program definitely beats the performance. By comparing `real` time, my program was roughly 8 times faster than `char_stat.c`. Let's compare how it works with different thread counts.

1 10	2 20	3 50	5 100
3.339 0.436	1.433 0.333	1.101 0.318	0.695 0.374

Table 3: Execution Time by Thread Count

Quite impressing, isn't it? Although, 100 threads performed worse than 50 threads, there were kind of a improvement with increasing thread count. The table can be visualized like this:

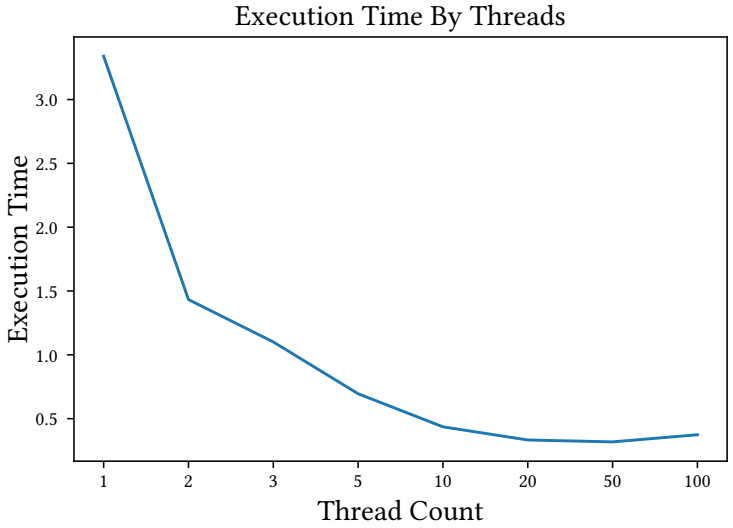

Figure 7: Performance Boost by Thread Count

However, with 100 threads, it performs worse than 50 threads. Besides my poor multithreading skills, I needed something to take blame on. Which turned out to be a law named *Amdhal's Law*. I will deal with this subject in the next page.

7 Conclusion & Analysis

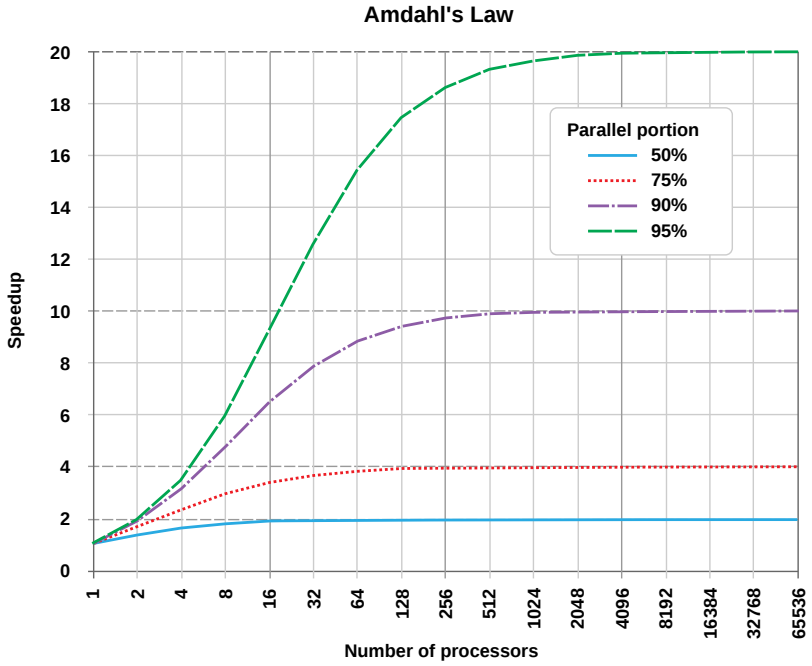

Figure 8: From https://en.wikipedia.org/wiki/Amdahl%27s_law/media/File:AmdahlsLaw.svg

So, let's assume that in a multiverse, I have successfully implemented this program flawlessly (no dead locks, no inconsistency, no whatsoever).

Let's assume the following:

- 75% of my program was able to be ran in parallel. Let's say 25% is not able to be ran in parallel.
- We had some threads that could accelerate my program.
- My program was perfect, no dead locks. 100% guaranteed working.

Let's apply *"Amdahl's law"*:

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

So, we have $p = 0.75$ (since our 75% of my program was able to be ran in parallel). Let's see how it goes when we had multiple threads that accelerate my program. For example, if we had 2 threads (this means $s = 2$), the performance would become:

$$S_{\text{latency}}(0.75) = \frac{1}{(1-0.75) + \frac{0.75}{2}} = 1.6$$

That is, compared to 1 thread, the performance would be 1.6 times faster than the original.

However, as Figure 6 shows us, with 75% of my program being able to run in parallel, at most the performance boost that we can get is limited to 4. Even if we had 65536 threads. This means that multithreading can give us performance boost, however that is limited. For example, even if we had our program's 95% being able to run in parallel, that makes maximum limit of 20 times faster than single thread.

In conclusion, I really enjoyed implementing the multithread project. Also, I could learn some big lessons with this homework:

- Multiple threads using single shared object will be just same as using a single thread. Or even might be worse. Using multiple consumers and producers with single shared object is same as single consumer and producer.
- Even if I manage to make it multithread, the speed up is limited by *"Amdahl's law"*.
- Fixing broken code with multithread is painful.
- Threads do not synchronize automatically. At least in `C`.

With those two lessons, I came to conclusion that *"Proper and precise usage of multithreading is beneficial, however improper and broken usage of multithreading will perform worse than single threaded"*. Also, let's not forget the fact that the time implementing and debugging broken code for multithreading was far beyond the time spent for single threaded programs.

Since I had used some other programming languages such as `Java` and `Python`, I just looked into their methods of implementing mutexes and synchronizations.

- `Java`: Has a keyword named `synchronized` and `atomic` which helps us resolve race conditions easily.
- `Python`: Has mutexes and semaphores.

In summary, the project became bigger than I expected and took more time as well. However, it was fun and enjoyed the journey of implementing a multithreaded program. Yes, it was painful but it was worth it. I will fix the project and resolve dead locks with the project when I have extra time (like in the winter break or some time that I can spare). I will be signing off this project and this document since I have to move forward for projects from the class.