Operating Systems - Homework 2

- Multi-Threaded Word Count Program-

ChangYoon Lee

Dankook University, Korea Republic
32183641@gmail.com

Abstract. Most modern operating systems have extended the concept of the process to allow having multiple threads of execution and thus to perform more than one task at a time (*Silberschatz et al., 2014*). While allowing this multi-threaded concept, there are some issues to consider in designing it: Synchronization (Concurrency Control). In this paper, we will discuss the concept of the multi-thread, synchronization examples, and their solutions, especially on producer and consumer problems. Also, we will implement the example case of producer and consumer problems with a multi-threaded word count program and evaluate it.

Keywords: Concurrency Control, Multi-Thread, Mutex, Producer and Consumer, Semaphore, Synchronization, Reader and Write

1 Introduction

Thread is a unit of execution. It has an execution context, which includes the registers, and stack. Denote that the address space in the memory is shared among the threads in the same process, so there is no clear separation and protection for accessing the memory space among the threads in the same process (Yoo, Mobile-os-DKU-cis-MSE). This single thread allows the process to perform only one task simultaneously. However, modern operating systems support the process of having multiple threads, so that they can execute multiple tasks parallelly at a time.

The concept of multi-threaded programming has some benefits, but there are some problems to be resolved to apply the following concepts, such as synchronization, mutual exclusion, deadlock, starvation, and optimization. To determine the following problem, we use several solutions such as queue, mutex, semaphore, and optimization methods, for the synchronization.

In this paper, we will first explain the concepts of thread, multi-thread, problems along the multi-threaded programming, and their solutions. By applying these concepts, we will explain how we implemented the multi-threaded word count program and its results for versions 1 through 3. Also, we will show the optimization with some methods to present the enhanced performance. At the end of the paper, we will present the execution time among the differences between the number of threads and implementation methods.

2 Requirements

Index	Requirement
1	Correct the values in the thread and keep the consistencies in the given producer and consumer program.
2	Correct the given code for producer and consumer program so that it works with single producer and single consumer.
3	Enhance the given producer and consumer program to support multiple consumers.
4	Make consumer threads to gather some statistics of the given text in the file. a. Count the number of each alphabet character in the line. b. At the end of the execution, the program should print out the statistic of the entire text. c. Reach to the fastest execution and maximize the concurrency.

Figure 1 - Requirement Specification

Figure 1 shows the requirements for a multi-threaded word count program. The implementations for these requirements will be described in detail afterwards.

3 Concepts

3.1 Thread

The normal process model implies that a process is a program that performs a single thread of execution. For example, when a process is running a word-processor program, a single thread of the instructions is being executed. This single thread of control allows the process to perform only one task at a time (*Silberschatz et al., 2014*). On the systems that supports thread, the process control block (PCB) is expanded to include the information for the thread. Other changes throughout the system are also needed to support the threads.

3.2 Multi-Thread

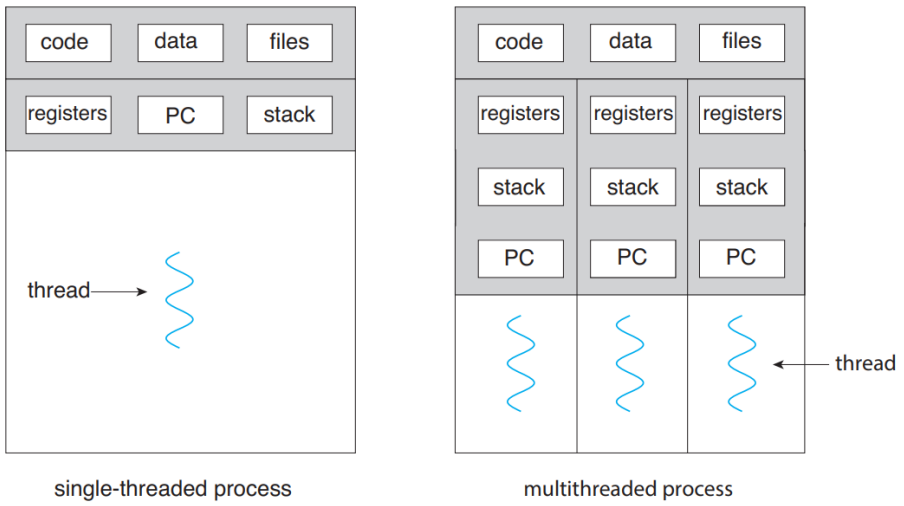


Figure 2 - Single-Threaded and Multi-Threaded Processes (*Silberschatz et al., 2014*)

As mentioned, a thread is a basic unit of CPU utilization, and most modern operating systems support the multi-thread for a single process. Also, modern software and applications run on multi-threaded devices. A single thread comprises a thread ID, a program counter (PC), a register set, and a stack. The concept of multi-thread uses multiple threads so that the program can execute multiple tasks parallelly at a time. Figure 2 shows the models of single-threaded and multi-threaded processes (*Silberschatz et al., 2014*).

The benefits of multi-threaded programming can be presented following categories:

- **Responsiveness**: Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.
- **Resource Sharing**: Processes can share resources only through techniques such as shared memory and message passing.
- **Economy**: Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads (*Silberschatz et al., 2014*).
- **Scalability**: The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores (*Silberschatz et al., 2014*).

Although multi-threaded programming has various advantages, there are also challenges in modifying multi-threaded programs.

The challenges that must be resolved in multi-threading are presented in the following categories:

- **Identifying Tasks**: This involves examining applications to find areas that can be divided into separate, concurrent tasks (*Silberschatz et al., 2014*).
- **Balancing**: While identifying tasks that can run in parallel, programmers must also ensure that the tasks perform equal work of equal value (*Silberschatz et al., 2014*)
- **Data Splitting**: Just as applications are divided into separate tasks, the data accessed and manipulated by the tasks must be divided to run on separate cores (*Silberschatz et al., 2014*).
- **Data Dependency**: The data accessed by the tasks must be examined for dependencies between two or more tasks. When one task depends on data from another, programmers must ensure that the execution of the tasks is synchronized to accommodate the data dependency (*Silberschatz et al., 2014*).
- **Testing and Debugging**: When a program run in parallel on multiple cores, many different execution paths are possible (*Silberschatz et al., 2014*)

In this paper, we will focus on applying the concept of multi-threaded programming, and the solutions for resolving the presented challenges in the word count program. These challenges will also be described in detail, in a section on the optimization of the implemented program.

3.3 Synchronization

According to the concept of multi-threaded programming, data dependency can occur. By the data dependency, we would arrive at the incorrect state when the outcome of the execution depends on the order in which the access takes place. This situation is called a race condition. To avoid the following situation, we need to ensure that only one process at a time can manipulate the variable count.

Such situations occur frequently in the operating systems as distinct parts of the system manipulate the resources as multiple threads. As mentioned before, resolving the data dependency of multi-thread programming is an important challenge. Resolving the following situations is called synchronization and coordination among cooperating threads.

Each process and threads have a segment of code that accesses or updates the data that is shared with at least other processes or threads. These segmentations of code are called critical sections. One of the main situations in synchronization is protecting the access to the following critical section while one other process or thread is executing the codes that refer to the critical section, and this is called the critical-section problem.

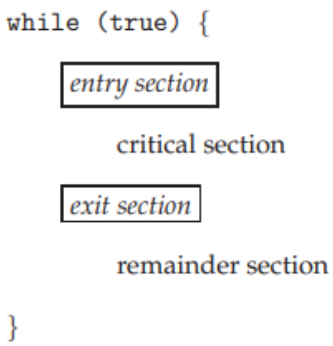


Figure 3 – General Structure of Typical Process (*Silberschatz et al., 2014*)

The critical-section problem is to design a protocol that the thread can use to synchronize their activity to cooperatively share the data. Figure 3 shows the general structure of the code in a process that is used to resolve the critical-section problem. Each thread must request permission to enter its critical section. The section of code that requests this permission is called the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section. A solution to resolve the critical-section problem must contain the following three requirements:

- **Mutual Exclusion**: If one thread is executing its critical section, no other threads can execute their critical sections.
- **Progress**: If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely (*Silberschatz et al., 2014*).

- **Bounded Wait**: There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted (*Silberschatz et al., 2014*).

There are several solutions for the following situation that satisfies the requirements above. In this paper, we will present two main solutions for resolving the critical-section problem, which is mutex and semaphore. The details of the mutex and semaphore will be presented in the later sections.

4 Multi-Threaded Word Count Program

4.1 Mutex

Operating system designers built higher-level software tools to solve the critical section problem. The simplest tool is the mutex lock. Mutex lock is used to protect the critical sections and prevent race conditions. This means that the thread must acquire the lock before entering a critical section and releases the lock when it exits the critical section.

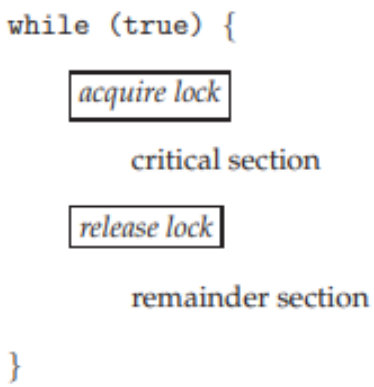


Figure 4 – Solving Critical-Section Problem by using Mutex Lock (*Silberschatz et al., 2014*)

Figure 4 shows the solution code that uses mutex lock to resolve the critical section problem. The acquire function acquires the lock, and the release function releases the lock. A mutex has a Boolean variable whose value indicates whether the lock is available or not. If the lock is available, the acquire function succeeds, and the lock is then considered to be unavailable. A thread that attempts to acquire an unavailable lock is blocked until the lock is released.
The main disadvantage of the implementation of the mutex lock is that it acquires busy waiting. While a process or thread is in its critical section, any other process that tries to enter its critical section must loop continuously in the call of acquiring function. This continual looping becomes a problem in the multi-programming system because it wastes the CPU cycle that some other processes and threads might be able to use productively. In the aspect of continuously looping for the busy wait, the mutex lock is also called a spinlock because the thread spins while waiting for the lock to become available.

4.2 Semaphore

A semaphore is an integer variable that is accessed only through two standard atomic operations, which are waiting and signal functions. When one thread modifies the semaphore value, no other thread can simultaneously modify that same semaphore value. Also, in the case of the wait function, the testing for value whether the semaphore is less than zero or not must be executed without interruption.

Operating systems often distinguish the semaphores between counting and binary semaphores. The value of a counting semaphore can range over an unlimited value. However, the value of a binary semaphore can range only between zero and one. Therefore, the binary semaphores act similarly to mutex locks.

Counting semaphores can be used to control the access to given resources that are consisted of a finite number of instances. This semaphore is initialized to the number of available resources. Each thread that needs to use a resource performs the wait function on the semaphore. When a thread releases a resource, it performs a signal function. If the count of the semaphore goes to zero, all resources are being used. After that, threads that wish to use a resource will block until the count of the semaphore becomes greater than zero.

By using the semaphore, we can resolve the various synchronization problems. One of the well-known problems in synchronization is the producer and consumer problem. The details and implemented solutions for the following problem will be discussed in the later sections.

4.3 Producer and Consumer Problem

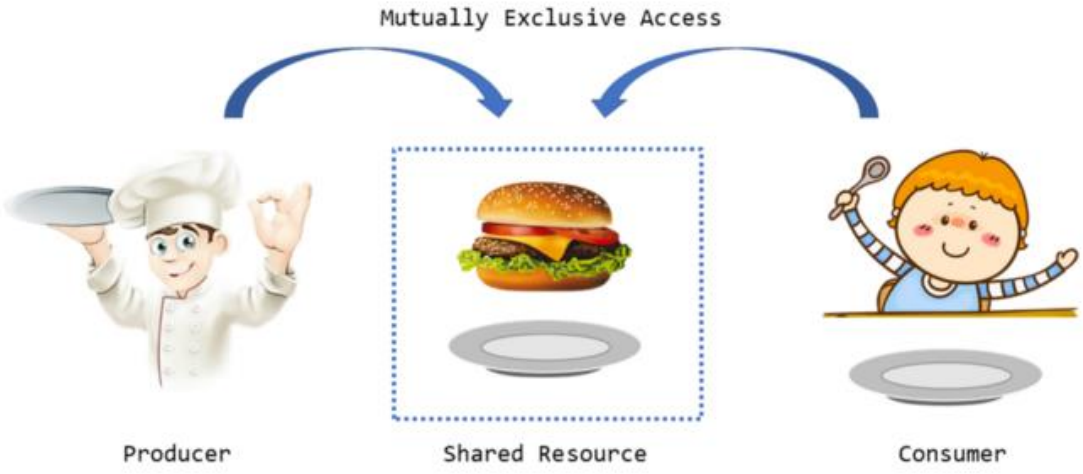


Figure 5 – Producer and Consumer (*Yan, 2020*)

The common concept of the cooperating processes or threads is the producer and consumer problem. A producer thread produces information that is consumed by a consumer thread. For example, a compiler may produce the assembly code that is consumed by an assembler.
One solution to the producer and consumer problem is using shared memory. To allow producer and consumer threads to run concurrently, the computer must have the available buffer of items that can be filled by the producer and emptied by the consumer. A producer can produce the item while the consumer is consuming another item. The producer and consumer threads must be synchronized so that the consumer does not try to consume an item that has not been produced (*Silberschatz et al., 2014*).

There can be two types of buffers, which are unbounded and bounded. The unbounded buffer places no practical limit on the size of the buffer. The consumer may have to wait for the latest item, but the producer can always produce the latest items. However, for the bounded buffer, because it assumes the fixed size buffer, the consumer must wait for the buffer is empty and the producer must wait if the buffer is full.
One issue in the bounded buffer producer and consumer situation concerns the situation that both the producer and consumer threads attempt to access the shared buffer concurrently. To reach the concurrency of the producer and consumer threads, we can use semaphore and mutex.

```
while (true) {
     . . .
   /* produce an item in next_produced */
     . . .
   wait(empty);
   wait(mutex);
     . . .
   /* add next_produced to the buffer */
     . . .
   signal(mutex);
   signal(full);
}
```

Figure 6 – The structure of the producer process (*Silberschatz et al., 2014*)

```
while (true) {
   wait(full);
   wait(mutex);
     . . .
   /* remove an item from buffer to next_consumed */
     . . .
   signal(mutex);
   signal(empty);
     . . .
   /* consume the item in next_consumed */
     . . .
}
```

Figure 7 – Solving Critical-Section Problem by using Mutex Lock (*Silberschatz et al., 2014*)

Figure 6 and 7 shows the code for the producer and consumer threads. Note the symmetry between the producer and the consumer. We can interpret the following code as the producer producing the full buffer for the consumer or as the consumer producing empty buffers for the producer.

The following problems will be also presented in the implantation of a multi-threaded word count program. By applying the code presented in Figures 6 and 7, we will state the problem that occurs while implanting the following program and explain how we solved the problem by using mutex and semaphore.

4.4 POSIX Synchronization - Mutex

The POSIX API allows the programmers at the user level to proceed with sections that pertain to synchronization. The following API is not part of any particular operating system kernel, so it can be used widely along any operating system.

```
#include <pthread.h>

pthread_mutex_t mutex;

/* create and initialize the mutex lock */
pthread_mutex_init(&mutex,NULL);
```

Figure 7 – pthread_mutex_init function (*Silberschatz et al., 2014*)

Mutex locks represent the fundamental synchronization technique used with the Pthreads. Pthreads uses the pthread_mutex_t data type for mutex lock. A mutex is created with the pthread_mutex_init function. The first parameter is a pointer to the mutex. By passing NULL as a second parameter, we can initialize the mutex to its default attributes. Figure 7 shows the code for the following data type and function.

```
/* acquire the mutex lock */
pthread_mutex_lock(&mutex);

/* critical section */

/* release the mutex lock */
pthread_mutex_unlock(&mutex);
```

Figure 8 – pthread_mutex_lock and pthread_mutex_unlock functions (*Silberschatz et al., 2014*)

The mutex is acquired and released with the pthread_mutex_lock and pthread_mutex_unklock functions. If the mutex lock is unavailable when the pthread_mutex_lock function is invoked, the calling thread is blocked until the owner invokes the pthread_mutex_unlock. Figure 8 shows the code for the following functions.

4.5 POSIX Synchronization - Semaphore

POSIX system also provides the semaphores, although semaphores are not part of the POSIX standard and instead belong to the POSIX SEM extension. POSIX specifies two types of semaphores, which are named and unnamed semaphores. In this paper, we will only explain the named semaphore.

```
#include <semaphore.h>
sem_t *sem;

/* Create the semaphore and initialize it to 1 */
sem = sem_open("SEM", O_CREAT, 0666, 1);
```

Figure 9 – sem_open function (*Silberschatz et al., 2014*)

The function sem_open is used to create and open a POSIX named semaphore. Figure 9 shows the code for the following function. The advantage of the named semaphore is that multiple unrelated threads can easily use a common semaphore as a synchronization mechanism by simply referring to the semaphore's name.

```
/* acquire the semaphore */
sem_wait(sem);

/* critical section */

/* release the semaphore */
sem_post(sem);
```

Figure 10 – sem_wait and sem_post functions (*Silberschatz et al., 2014*)

Figure 10 shows the code for sem_wait and sem_post functions. Which takes the role of the semaphore's signal and wait for operation which is presented in the previous section. Both LINUX and macOS systems provide the POSIX named semaphores.

4.6 POSIX Synchronization – Condition Variable

```
pthread_mutex_t mutex;
pthread_cond_t cond_var;

pthread_mutex_init(&mutex,NULL);
pthread_cond_init(&cond_var,NULL);
```

Figure 11 – ptheard_cond_t data type and pthread_cond_init function (*Silberschatz et al., 2014*)

Condition variables in Pthreads provide a locking mechanism to ensure data integrity. Condition variables in Pthreads use the pthread_cond_t data type and are initialized by the pthread_cond_init function. Figure 11 shows the code of the following data type and function.

```
pthread_mutex_lock(&mutex);
while (a != b)
     pthread_cond_wait(&cond_var, &mutex);

pthread_mutex_unlock(&mutex);
```

Figure 12 – pthread_cond_wait function (*Silberschatz et al., 2014*)

The pthread_cond_wait function is used for waiting on a condition variable. The code presented in Figure 12 shows how a thread can wait for the following condition to become true using a Pthreads condition variable. The mutex lock associated with the condition variable must be locked before the pthread_cond_wait function is called since it is used to protect the data in the conditional clause from a race condition (*Silberschatz et al., 2014*).

```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&cond_var);
pthread_mutex_unlock(&mutex);
```

Figure 12 – pthread_cond_signal function (*Silberschatz et al., 2014*)

A thread that modifies the sheared data can invoke the pthread_cond_signal function to return the conditional variable. Figure 13 shows the code of the following function. It is important to note that the call to the pthread_cond_signal does not release the mutex lock. Once the mutex lock is released, the signaled thread becomes the owner of the mutex lock and returns the control from the call to the pthread_cond_wait function.

4.7 Program Definition

Before implementing the multi-threaded word count program, we will state the additional program definition that will be used in the real implementation.

- Global Variables

Variables	Data Type	Definition
ASCII_SIZE	256	Size of the total number of ASCII characters
BUFFER_SIZE	100	The size of the shared buffer
MAX_STRING-LENFTG	30	Maximum length of the single word that the word count program will read

- Modules and Functions

Modules	Functions	Definition
prod_cons	producer	Reads the lines from a given file, and put the line string on the shared buffer
	consumer	Get string from the shared buffer, and print the line out on the console screen
	main	Main thread which performs the admin job

5 Implementation

5.1 Producer and Consumer Version 1

```
typedef struct sharedobject {
        FILE *rfile;
        int linenum;
        char *line;
        int buffsize;
        pthread_cond_t full;
        pthread_cond_t empty;
        pthread_mutex_t lock;
} so_t;
```

Figure 13 – Structure of the Shared Buffer of the Program 'Producer and Consumer Version 1'

```
void *producer(void *arg) {
        so_t *so = arg;
        int *ret = malloc(sizeof(int));
        FILE *rfile = so->rfile;
        int i = 0;
        char *line = NULL;
        size_t len = 0;
        ssize_t read = 0;

        while (1) {
                pthread_mutex_lock(&so->lock);
                while (so->buffsize == 1) pthread_cond_wait(&so->empty, &so->lock);

                read = getdelim(&line, &len, '\n', rfile);
                if (read == -1) {
                        so->line = NULL;
                        so->buffsize = 1;
                        pthread_cond_signal(&so->full);
                        pthread_mutex_unlock(&so->lock);
                        break;
                }

                so->linenum = i;
                so->line = strdup(line);      /* share the line */
                so->buffsize = 1;
                i++;

                pthread_cond_signal(&so->full);
                pthread_mutex_unlock(&so->lock);
        }

        free(line);
        printf("Prod_%x: %d lines\n", (unsigned int)pthread_self(), i);
        *ret = i;
        pthread_exit(ret);
}
```

Figure 14 – Code for Producer Thread of the Program 'Producer and Consumer Version 1'

```c
void *consumer(void *arg) {
        so_t *so = arg;
        int *ret = malloc(sizeof(int));
        int i = 0;
        int len;
        char *line;

        while (1) {
                pthread_mutex_lock(&so->lock);
                while (so->buffsize == 0) pthread_cond_wait(&so->full, &so->lock);

                line = so->line;
                if (line == NULL) {
                        break;
                }

                len = strlen(line);
                printf("Cons_%x: [%02d:%02d] %s", (unsigned int)pthread_self(), i, so->linenum, line);
                so->buffsize = 0;
                free(so->line);
                i++;

                pthread_cond_signal(&so->empty);
                pthread_mutex_unlock(&so->lock);
        }

        printf("Cons: %d lines\n", i);
        *ret = i;
        pthread_exit(ret);
}
```

Figure 15 – Code for Consumer Thread of the Program 'Producer and Consumer Version 1'

```c
// mutex initialization
pthread_cond_init(&share->full, NULL);
pthread_cond_init(&share->empty, NULL);
pthread_mutex_init(&share->lock, NULL);
```

Figure 16 – Initialization of the mutexes used in the Program 'Producer and Consumer Version 1'

The program 'producer and consumer version 1' is focused on solving the producer and consumer problem for a single producer and a single consumer. To reach the following problem, we used two pthread_cond_t variables, full and empty, and one pthread_mutex_t variable, which is a lock. Figure 13 to 16 shows the implemented code of the program 'producer and consumer version 1'.

Each code of the producer and consumer thread follows the operation flow which is presented in section 4.3. In the producer thread, it first acquires the mutex lock to get into its critical section. Then, after getting the buffer empty signal, it processes the critical section by filling the produced data into the shared buffer and unlocking the lock and full to signal that the consumer thread can get into its critical section. In the consume thread, it acquires the mutex lock to get into its critical section and consumes the data in the shared buffer. Then, it unlocks the lock and empties it to signal that the producer thread can get into its critical section. In short, the lock mutex limits the accessibility to the shared buffer, and full and empty mutex synchronizes the execution of the prouder thread and the consumer thread.

The following program executes successfully on the single producer and single consumer cases. However, it still has the limitation on executing single producers and multiple consumers, multiple producers, and single consumers, and multiple producers and multiple consumers.

5.2 Producer and Consumer Version 2.1

```
typedef struct sharedobject {
        int nextin;
        int nextout;
        FILE* rfile;
        int consumer;
        int buffsize;
        char* line[BuffSize];

        pthread_cond_t full;
        pthread_cond_t empty;
        pthread_mutex_t lock;
} so_t;
```

Figure 17 – Structure of the Shared Buffer of the Program 'Producer and Consumer Version 2'

```
share->consumer = 0;
share->rfile = rfile;
for (i = 0; i < BuffSize; i++) share->line[i] = NULL;

// mutex initialization
share->nextin = 0;
share->nextout = 0;
share->buffsize = 0;
pthread_cond_init(&share->full, NULL);
pthread_cond_init(&share->empty, NULL);
pthread_mutex_init(&share->lock, NULL);
```

Figure 18 – Initialization of the mutexes used in the Program 'Producer and Consumer Version 2'

```
void *producer(void *arg) {
        so_t *so = arg;
        int i = 0;
        int count = 0;
        size_t len = 0;
        ssize_t read = 0;
        char *line = NULL;
        FILE *rfile = so->rfile;
        int *ret = malloc(sizeof(int));

        while (1) {
                pthread_mutex_lock(&so->lock);
                while (so->buffsize > 0) pthread_cond_wait(&so->empty, &so->lock);

                read = getdelim(&line, &len, '\n', rfile);
                if (read == -1) {
                        so->line[so->nextin] = NULL;
                        so->buffsize++;
                        pthread_cond_broadcast(&so->full);
                        pthread_mutex_unlock(&so->lock);
                        break;
                }

                so->line[so->nextin] = strdup(line); // share the line
                so->nextin = (so->nextin + 1) % BuffSize;
                so->buffsize++;
                count++;

                pthread_cond_broadcast(&so->full);
                pthread_mutex_unlock(&so->lock);
                sleep(0.1);
        }

        free(line);
        printf("Prod_%x: %d lines\n", (unsigned int)pthread_self(), count);
        *ret = count;
        pthread_exit(ret);
}
```

Figure 19 – Code for Producer Thread of the Program 'Producer and Consumer Version 2'

```
void *consumer(void *arg) {
        int len;
        int i = 0;
        char *line;
        int count = 0;
        so_t *so = arg;
        int *ret = malloc(sizeof(int));

        while (1) {
                pthread_mutex_lock(&so->lock);
                while (so->buffsize == 0) pthread_cond_wait(&so->full, &so->lock);

                line = so->line[so->nextout];
                if (line == NULL) {
                        so->buffsize--;
                        pthread_cond_broadcast(&so->empty);
                        pthread_mutex_unlock(&so->lock);
                        break;
                }

                len = strlen(line);
                printf("Cons_%x: [%02d:%02d] %s", (unsigned int)pthread_self(), count, so->nextout, line);
                so->nextout = (so->nextout + 1) % BuffSize;
                so->buffsize--;
                count++;

                pthread_cond_broadcast(&so->empty);
                pthread_mutex_unlock(&so->lock);
                sleep(0.1);
        }

        printf("Cons: %d lines\n", count);
        *ret = count;
        pthread_exit(ret);
}
```

Figure 20 – Code for Consumer Thread of the Program 'Producer and Consumer Version 2'

The limitation of the program 'producer and consumer version 1' is that it can only be executed in a single producer and single consumer condition. The program 'producer and consumer version 2.1' is focused on solving the following limitations. To reach the following problem, we changed the previous buffer that can contain a single line into a buffer that can contain multiple lines. Also, we changed the string variable into the list of the string and added new variables, next in and next out, to point out the current index that the producer and consumer are producing and consuming the data in the array. After the condition check of the while loop, the buffer variable must also be added or subtracted due to the execution of the producer and consumer threads. Figure 17 to 20 show the implemented code of the program 'producer and consumer version 2.1'.

Since we use a buffer that can contain multiple lines, we must change the buffer size which is limited to one to limit the number of the lines that buffer can contain. Also, we must deal with the condition statement of the while loop for conditional mutex locks. One of the problems while implementing the following program was the termination operation of the consumer threads. After the termination of the first consumer thread, other consumer threads wait for the full semaphore to be filled, and as a result, the program cannot terminate its execution. To solve this problem, we fixed an operation at the termination of the producer threads and consumer threads. In Figure 20, we can check the orange box that denotes the broadcasting signal for the conditional mutex lock. By broadcasting the signal of the conditional mutex lock to every single consumer thread, we can cascade terminated consumer threads.

Also, initializing the conditional mutex locks in the proper value is important for work to be done. As we mentioned in section 4.6, we can initialize our conditional variables by using the pthread_cond_init function. The buffer size that is used to control the conditional mutex locks must also be set as zero for the initial state.

5.3 Producer and Consumer Version 2.2

```
typedef struct sharedobject {
        int nextin;
        int nextout;
        FILE* rfile;
        int consumer;
        char* line[BuffSize];

        sem_t full;
        sem_t empty;
        pthread_mutex_t lock;
} so_t;
```

Figure 21 – Structure of the Shared Buffer of the Program 'Producer and Consumer Version 2'

```
share->consumer = 0;
share->rfile = rfile;
for (i = 0; i < BuffSize; i++) share->line[i] = NULL;

// mutex initialization
share->nextin = 0;
share->nextout = 0;
sem_init(&share->full, 0, 0);
sem_init(&share->empty, 0, BuffSize);
pthread_mutex_init(&share->lock, NULL);
```

Figure 22 – Initialization of the mutexes used in the Program 'Producer and Consumer Version 2'

```
void *producer(void *arg) {
        so_t *so = arg;
        int *ret = malloc(sizeof(int));
        FILE *rfile = so->rfile;
        int i = 0;
        int count = 0;
        char *line = NULL;
        size_t len = 0;
        ssize_t read = 0;

        while (1) {
                sem_wait(&so->empty);
                pthread_mutex_lock(&so->lock);

                read = getdelim(&line, &len, '\n', rfile);
                if (read == -1) {
                        so->line[so->nextin] = NULL;
                        pthread_mutex_unlock(&so->lock);
                        sem_post(&so->full);
                        break;
                }

                so->line[so->nextin] = strdup(line);      /* share the line */
                so->nextin = (so->nextin + 1) % BuffSize;
                count++;

                pthread_mutex_unlock(&so->lock);
                sem_post(&so->full);
                sleep(0.1);
        }

        free(line);
        printf("Prod_%x: %d lines\n", (unsigned int)pthread_self(), count);
        *ret = count;
        pthread_exit(ret);
}
```

Figure 23 – Code for Producer Thread of the Program 'Producer and Consumer Version 2'

```
void *consumer(void *arg) {
        so_t *so = arg;
        int *ret = malloc(sizeof(int));
        int i = 0;
        int len;
        char *line;
        int count = 0;

        while (1) {
                sem_wait(&so->full);
                pthread_mutex_lock(&so->lock);

                line = so->line[so->nextout];
                if (line == NULL) {
                        pthread_mutex_unlock(&so->lock);
                        sem_post(&so->empty);
                        sem_post(&so->full);
                        break;
                }

                len = strlen(line);
                printf("Cons_%x: [%02d:%02d] %s", (unsigned int)pthread_self(), count, so->nextout, line);
                so->nextout = (so->nextout + 1) % BuffSize;
                count++;

                pthread_mutex_unlock(&so->lock);
                sem_post(&so->empty);
                sleep(0.1);
        }

        printf("Cons: %d lines\n", count);
        *ret = count;
        pthread_exit(ret);
}
```

Figure 24 – Code for Consumer Thread of the Program 'Producer and Consumer Version 2'

We can also resolve the limitation of the program 'producer and consumer version 1' by using the semaphore, not conditional mutex lock. The program 'producer and consumer version 2.2' is focused on solving the following limitations by using semaphores. To reach the following problem, we changed the previous pthread_cond_t variable into the semaphores, full and empty. Also, we changed the string variable into the list of the string and added new variables, next in and next out, to point out the current index that the producer and consumer are producing and consuming the data in the array. Figure 21 to 24 show the implemented code of the program 'producer and consumer version 2.2'.

Since we changed the conditional mutexes into semaphores, we no longer use the while loop for the conditional statement, but we use the wait function of the semaphores. One of the problems while implementing the following program was the termination operation of the consumer threads. After the termination of the first consumer thread, other consumer threads wait for the full semaphore to be filled, and as a result, the program cannot terminate its execution. To solve this problem, we added an operation at the termination of the consumer threads. In Figure 24 we can check the orange box that denotes the post signal of the full semaphore. By signaling the full semaphore for every single consumer thread, we can cascade terminated consumer threads.

Also, initializing the semaphores in the proper value is important for work to be done. As we mentioned in section 4.5, we can initialize our semaphores by using the sem_init function. The important thing is that the empty semaphore should be initialized so that it can be increased to the shared buffer size and the full semaphore is initialized to be the conditional semaphore. The reason is that the producer can produce the item until the shared buffer is fully charged, but the consumer can only consume the item when the shared buffer is not empty. Therefore, the full semaphore should be set as a conditional semaphore.

5.4 Producer and Consumer Version 2.3

```
#define BUFFERSIZE 4096

typedef struct threadobject {
        int fd;
        int index;
        long offset;
} to_t;

int terminate;
char** buffer;
long int fsize;
int Nprod, Ncons;
pthread_mutex_t* full;
pthread_mutex_t* empty;
```

Figure 25 – Structure of the Shared Buffer of the Program 'Producer and Consumer Version 2'

```
void* producer(void* arg) {
        int result;
        int count = 0;
        to_t* to = arg;
        int offset = 0;
        lseek(to->fd, to->offset, SEEK_SET);
        char* block = (char*)malloc(sizeof(char) * BUFFERSIZE);

        while (1) {
                pthread_mutex_lock(&empty[to->index]);

                if (count >= fsize / Nprod) break;
                if (fsize / Nprod - count >= BUFFERSIZE) offset = BUFFERSIZE;
                else offset =  fsize / Nprod - count;

                result = read(to->fd, block, offset);
                buffer[to->index] = block;
                count += offset;

                pthread_mutex_unlock(&full[to->index]);
        }

        // printf("P_%d: exit with %d\n", to->index, count);
        pthread_exit(0);
}
```

Figure 26 – Initialization of the mutexes used in the Program 'Producer and Consumer Version 2'

```
void* consumer(void* arg) {
        int index = 0;

        while (1) {
                if (pthread_mutex_trylock(&full[index]) == 0) {
                        if (buffer[index] != NULL) {
                                // printf("C_%d:\n%s\n\n", index, buffer[index]);
                                buffer[index] = NULL;
                                pthread_mutex_unlock(&empty[index]);
                        }
                }

                else {
                        if (terminate < Nprod) index = (index + 1) % Nprod;
                        else break;
                }
        }

        // printf("C_%d: exit\n", index);
        pthread_exit(0);
}
```

Figure 27 – Code for Producer Thread of the Program 'Producer and Consumer Version 2'

```
buffer = (char**)malloc(sizeof(char*) * Nprod);
if (buffer == NULL) {
        perror("malloc");
        exit(0);
}
memset(buffer, 0, sizeof(buffer));

fseek(rfile, 0, SEEK_END);
fsize = ftell(rfile);
rewind(rfile);

full = (pthread_mutex_t*)malloc(sizeof(pthread_mutex_t) * Nprod);
for (int i = 0; i < Nprod; i++) pthread_mutex_init(&full[i], NULL);
empty = (pthread_mutex_t*)malloc(sizeof(pthread_mutex_t) * Nprod);
for (int i = 0; i < Nprod; i++) pthread_mutex_init(&empty[i], NULL);

// thred initialization
printf("main continuing\n");
for (int i = 0 ; i < Nprod ; i++) {
        to[i].index = i;
        to[i].fd = open((char*) argv[1], O_RDONLY);
        to[i].offset = fsize  - i * fsize / Nprod <= 0 ? fsize : i * fsize / Nprod;

        pthread_create(&prod[i], NULL, producer, &to[i]);
}

for (int i = 0 ; i < Ncons ; i++) pthread_create(&cons[i], NULL, consumer, NULL);
```

Figure 28 – Code for Consumer Thread of the Program 'Producer and Consumer Version 2'

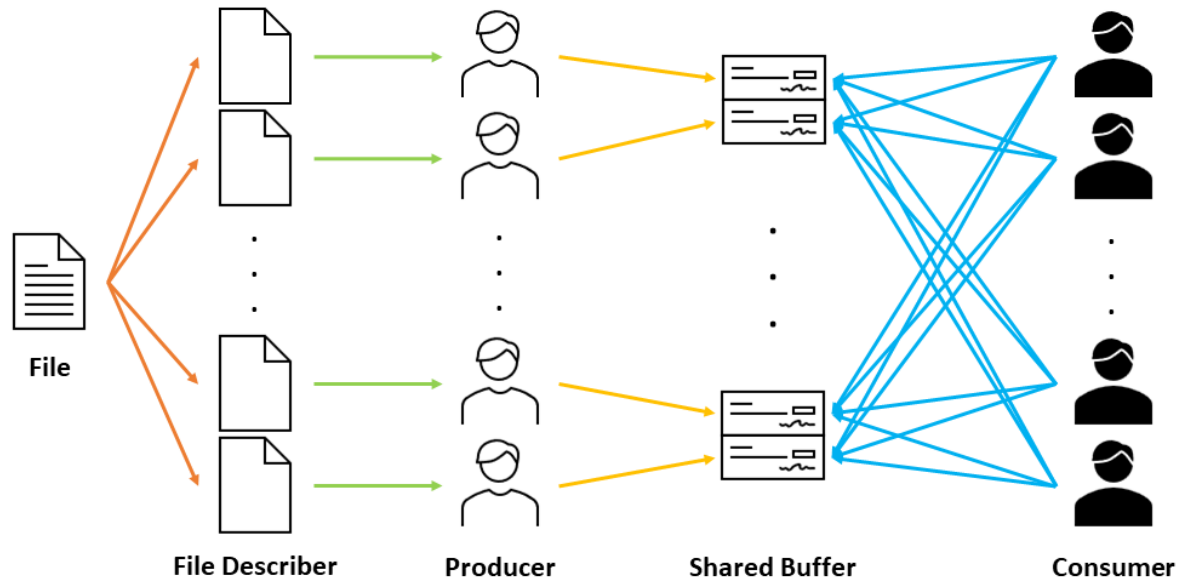


Figure 29 - Relationship Diagram between Producer and Consumer in Version 2.3

Even though we successfully implemented the code, the 'producer and consumer version 2.2,' that resolves the producer and consumer problem with the shared buffer, it does not show the ideal result which is the performance enhancement in terms of the increment for several threads. To achieve the performance increment in multithreading, we must consider the challenges to optimize the program version 2.2, which we mentioned in the previous section.

The 'producer and consumer version 2.3' follows the operation presented in Figure 29.
First, we generate multiple file describer that describes the target file that the program tries to read. Each file describer points at a different section, which is evenly distributed by the size of the file over some producer threads. Next, the producers start to read the file by using a file describer and store it in the local buffer. The file will be read in the size of the buffer and if the buffer overflows the region of the boundary which is allocated to each producer, it stops reading it. Third, every time the producer fills in the buffer, it waits for the shared buffer, which is shared with the consumers and owned by each producer, to be empty. If the following buffer is empty, the producer stores the red buffer into the shared buffer and notices the consumer that the shared buffer is full. While consumers are checking for the filled buffer and found one that is filled, it consumes the following buffer and notices the producer that the buffer is empty. If they reach empty, it searches for the next shared buffer. Since the consumer searches for the whole buffer, it terminates its operation.

To perform the following operation, we applied the methods of identifying the tasks, balancing, data splitting, data dependency, and testing and debugging, which are all about the challenges that we discussed. The following methods are implemented by the progress below:

- **Identifying Tasks**: In Figure 28, we can find out that the program generates multiple file describers with the offset that represents the region of the text file to each thread. Since each thread reads only the allocated region of the text file, we can say that the following program identifies the tasks for each thread.

- **Balancing**: The allocated region is the result of dividing the size of the file into the number of produce threads. As a result, all the same size for the region of text is allocated to each thread and performs a similar job of reading the text file.

- **Data Splitting**: As mentioned previously, each thread reads the separate region of the text file and performs each operation. This process is the same as processing the operation after splitting the data and allocating it to different threads.

- **Data Dependency**: Since each producer reads a different region of the text file and stores the read data into each allocated shared buffer, no data dependency occurs among the producer threads. Also, consumer threads first check the shared buffer and if it can consume the following buffer, it consumes. If not, update its index to point to the next shared buffer. In short, there is no data dependency among the consumer threads, either.

- **Test and Debugging**: As the multiple threads are running concurrently, many different execution paths are possible in the following program. We can see the results in the later section.

The first consideration of implementing the program is the change of the mutex locks. According to the following implementation, we no longer use the original mutex and semaphores, but we use two mutex locks for each shared buffer which notices that the following buffer is full or empty. These locks are used as signals that notify the consumer that the buffer is full or notify the producer that the buffer is empty. Also, on the consumer side, the operation that we wanted to implement was that it checks whether the shared buffer is full or not. Therefore, we used pthread_mutex_trylock, which checks the mutex lock and if it is available, it gets it, and if not, it returns the EBUSY. These operations show a small number of executions in terms of a mutex lock and unlock operation, and this occurs the better performance of the program in an aspect of execution time.

The second consideration in this program is using the file describer rather than the file pointer. If multiple threads own the file pointer that points to the same text file, one thread operating on the file pointer can ruin the other pointer's file pointer, because they share the same file stream. To avoid the following situation, we used the file describer and moved it with the separate offsets. As a result, we can divide the region of the text file and allocate them to each different producer thread.

The third consideration of the program is changing the size of the reading data of the produce threads. Previously, the producer threads read the text file line by line. When the text file gets longer and longer, the processing time of reading the text file in each line costs a lot of the execution time, which represents low performance. Therefore, we extended the size of the reading data into the local buffer of each producer thread, which has a size of 4KB. By reading the text file in a 4KB buffer for every reading operation, the program 'producer and consumer version 2.3' can achieve performance enhancement compared to the previous program.

Compared to the previous program, the 'producer and consumer version 2.3' shows a small number of executions on mutex lock and unlock operation, reads the text file in the size of the buffer, which is 4KB, and allocates each text file region to each different producer threads. These three main differences compared to 'producer and consumer version 2.2' occurs with a significant difference in performance between the following two programs. The implementation of the following program is presented in Figures 25 to 28.

5.5 Producer and Consumer Version 3

```
share->consumer = 0;
share->rfile = rfile;
memset(share->stat1, 0, sizeof(share->stat1));
memset(share->stat2, 0, sizeof(share->stat2));
for (i = 0; i < BUFFER_SIZE; i++) share->line[i] = NULL;
```

Figure 30 – Counting Word for its Length and the Starting Character in the Consumer Thread

```
// gather the char stat for the sinlge line
char* cptr = NULL;
char* brka = NULL;
char* substr = NULL;
char* sep = "{}()[],;\" \n\t^";

// for each line,
cptr = line;
for (substr = strtok_r(cptr, sep, &brka); substr; substr = strtok_r(NULL, sep, &brka)) {
        length = strlen(substr);
        if (length >= 30) length = 30;
        so->stat1[length-1]++;
        for (int i = 0; i < length; i++) {
                if (*cptr < 256 && *cptr >= 0) {
                        so->stat2[*cptr]++;
                }
                cptr++;
        }
        cptr++;
        if (*cptr == '\0') break;
}
```

Figure 31 – Counting Word for its Length and the Starting Character in the Consumer Thread

```
        // sum
        sum = 0;
        for (i = 0; i < 30; i++) sum += share->stat1[i];

        // print out the distributions
        printf("\n");
        printf("*** print out distributions *** \n");
        printf("  #ch  freq \n");
        for (i = 0; i < 30; i++) {
                int j = 0;
                int num_star = share->stat1[i] * 80 / (sum == 0 ? 1 : sum);
                printf("[%3d]: %4d \t", i + 1, share->stat1[i]);
                for (j = 0; j < num_star; j++) printf("*");
                printf("\n");
        }
        printf("\n");

        printf("       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\n");
        printf("%8d %8d %8d %8d %8d %8d %8d %8d %8d %8d %8d %8d %8d %8d %8d %8d %8d %8d %8d %8d %8d %8d %8d %8d %8d %8d\n",
                        share->stat2['A']+share->stat2['a'], share->stat2['B']+share->stat2['b'],  share->stat2['C']+share->stat2['c'],  share->stat2['D']+share->stat2['d'],  share->stat2['E']+share-
>stat2['e'],
                        share->stat2['F']+share->stat2['f'], share->stat2['G']+share->stat2['g'],  share->stat2['H']+share->stat2['h'],  share->stat2['I']+share->stat2['i'],  share->stat2['J']+share-
>stat2['j'],
                        share->stat2['K']+share->stat2['k'], share->stat2['L']+share->stat2['l'],  share->stat2['M']+share->stat2['m'],  share->stat2['N']+share->stat2['n'],  share->stat2['O']+share-
>stat2['o'],
                        share->stat2['P']+share->stat2['p'], share->stat2['Q']+share->stat2['q'],  share->stat2['R']+share->stat2['r'],  share->stat2['S']+share->stat2['s'],  share->stat2['T']+share-
>stat2['t'],
                        share->stat2['U']+share->stat2['u'], share->stat2['V']+share->stat2['v'],  share->stat2['W']+share->stat2['w'],  share->stat2['X']+share->stat2['x'],  share->stat2['Y']+share-
>stat2['y'],
                        share->stat2['Z']+share->stat2['z']);
```

Figure 32 – Printing Out the Counted Result in the Main Thread

The program ‘producer and consumer version 3’ is printing out the word count result from the read text file. To reach the following problem, add the code, which is implemented in the consumer thread, which counts the length of the line from the text file and the code counts the starting character of the word from the text file. Also, the code that prints out the counted results is implemented in the main thread. Figure 30 to 32 shows the implemented code of the program ‘producer and consumer version 3’.

6 Build Environment

- Build Environment:
  1. Linux Environment -> Vi editor, GCC Complier
  2. Program is built by using the Makefile.

- Build Command:
  1. $make prod_cons -> build the execution program for prod_cons from version 1 to 4.
  2. $make clean -> clean all the object files that consists of the prod_cons programs.

- Execution Command:
  1. . /Prod_cons_v1 {$readfile}
     -> Execute the producer and consumer version 1 program.
  2. . /Prod_cons_v2.1 {$readfile} #Producer #Consumer
     -> Execute the producer and consumer version 2.1 program.
  3. . /Prod_cons_v2.2 {$readfile} #Producer #Consumer
     -> Execute the producer and consumer version 2.2 program.
  4. . /Prod_cons_v2.3 {$readfile} #Producer #Consumer
     -> Execute the producer and consumer version 2.3 program.
  5. . /Prod_cons_v3 {$readfile} #Producer #Consumer
     -> Execute the producer and consumer version 3 program.
  6. . /Prod_cons_v4 {$readfile} #Producer #Consumer
     -> Execute the producer and consumer version 4 program.

7 Results

- Producer and Consumer Version 1 reading LICENSE file.

```
changyoon18@assam:~/Operating-Systems/hw2/prod_cons$ ./prod_cons_v1 LICENSE
main continuing
Cons_f8b20700: [00:00] MIT License
Cons_f8b20700: [01:01]
Cons_f8b20700: [02:02] Copyright (c) 2019 mobile-os-dku-cis-mse
Cons_f8b20700: [03:03]
Cons_f8b20700: [04:04] Permission is hereby granted, free of charge, to any person obtaining a copy
Cons_f8b20700: [05:05] of this software and associated documentation files (the "Software"), to deal
Cons_f8b20700: [06:06] in the Software without restriction, including without limitation the rights
Cons_f8b20700: [07:07] to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
Cons_f8b20700: [08:08] copies of the Software, and to permit persons to whom the Software is
Cons_f8b20700: [09:09] furnished to do so, subject to the following conditions:
Cons_f8b20700: [10:10]
Cons_f8b20700: [11:11] The above copyright notice and this permission notice shall be included in all
Cons_f8b20700: [12:12] copies or substantial portions of the Software.
Cons_f8b20700: [13:13]
Cons_f8b20700: [14:14] THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
Cons_f8b20700: [15:15] IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
Cons_f8b20700: [16:16] FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
Cons_f8b20700: [17:17] AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
Cons_f8b20700: [18:18] LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
Cons_f8b20700: [19:19] OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
Cons_f8b20700: [20:20] SOFTWARE.
Prod_f9321700: 21 lines
Cons: 21 lines
main: consumer_0 joined with 21
main: producer_0 joined with 21
```

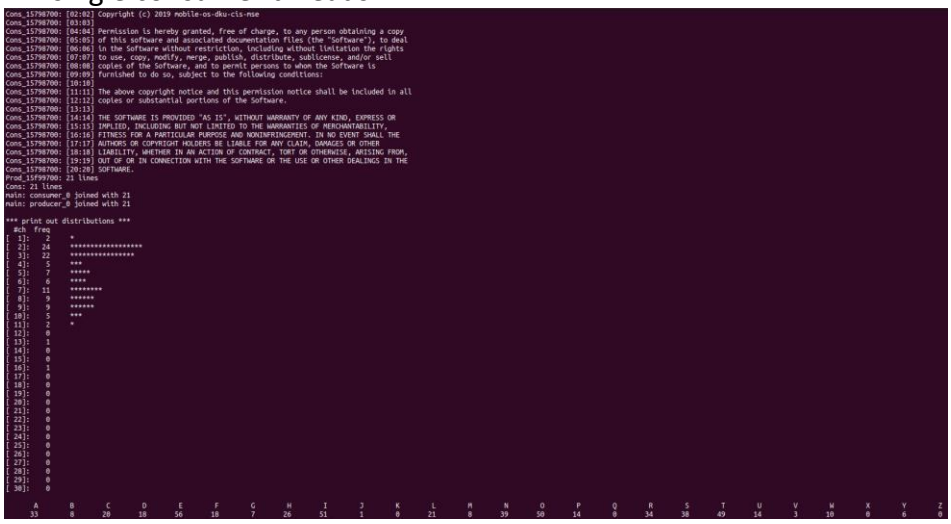
- Producer and Consumer Version 2.1 and 2.2 reading LICENSE file with two producer threads and one consumer thread.

```
changyoon18@assam:~/Operating-Systems/hw2/prod_cons$ ./prod_cons_v2 LICENSE 2 1
main continuing
Cons_fa3e3700: [00:00] MIT License
Cons_fa3e3700: [01:01]
Cons_fa3e3700: [02:02] Copyright (c) 2019 mobile-os-dku-cis-mse
Cons_fa3e3700: [03:03]
Cons_fa3e3700: [04:04] Permission is hereby granted, free of charge, to any person obtaining a copy
Cons_fa3e3700: [05:05] of this software and associated documentation files (the "Software"), to deal
Cons_fa3e3700: [06:06] in the Software without restriction, including without limitation the rights
Cons_fa3e3700: [07:07] to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
Prod_fabe4700: 10 lines
Cons_fa3e3700: [08:08] copies of the Software, and to permit persons to whom the Software is
Prod_fb3e5700: 11 lines
Cons_fa3e3700: [09:09] furnished to do so, subject to the following conditions:
Cons_fa3e3700: [10:10]
Cons_fa3e3700: [11:11] The above copyright notice and this permission notice shall be included in all
Cons_fa3e3700: [12:12] copies or substantial portions of the Software.
Cons_fa3e3700: [13:13]
Cons_fa3e3700: [14:14] THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
Cons_fa3e3700: [15:15] IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
Cons_fa3e3700: [16:16] FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
Cons_fa3e3700: [17:17] AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
Cons_fa3e3700: [18:18] LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
Cons_fa3e3700: [19:19] OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
Cons_fa3e3700: [20:20] SOFTWARE.
Cons: 21 lines
main: consumer_0 joined with 21
main: producer_0 joined with 11
main: producer_1 joined with 10
```

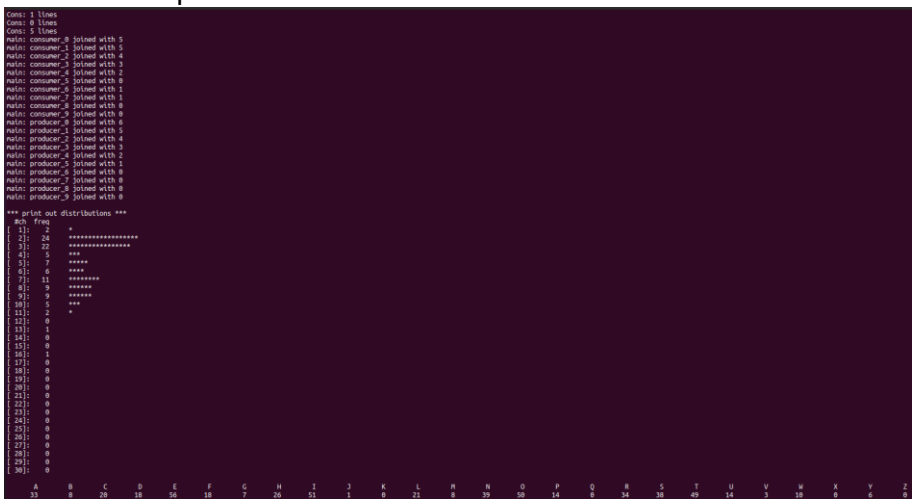
- Producer and Consumer Version 2.1 and 2.2 reading LICENSE file with one producer threads and two consumer thread.

```
changyoon18@assam:~/Operating-Systems/hw2/prod_cons$ ./prod_cons_v2 LICENSE 1 2
main continuing
Cons_7aa97700: [00:00] MIT License
Cons_7a296700: [00:01]
Cons_7aa97700: [01:02] Copyright (c) 2019 mobile-os-dku-cis-mse
Cons_7a296700: [01:03]
Cons_7aa97700: [02:04] Permission is hereby granted, free of charge, to any person obtaining a copy
Cons_7a296700: [02:05] of this software and associated documentation files (the "Software"), to deal
Cons_7aa97700: [03:06] in the Software without restriction, including without limitation the rights
Cons_7a296700: [03:07] to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
Cons_7aa97700: [04:08] copies of the Software, and to permit persons to whom the Software is
Cons_7a296700: [04:09] furnished to do so, subject to the following conditions:
Cons_7aa97700: [05:10]
Cons_7a296700: [05:11] The above copyright notice and this permission notice shall be included in all
Cons_7aa97700: [06:12] copies or substantial portions of the Software.
Cons_7a296700: [06:13]
Cons_7aa97700: [07:14] THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
Cons_7a296700: [07:15] IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
Cons_7aa97700: [08:16] FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
Cons_7a296700: [08:17] AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
Cons_7aa97700: [09:18] LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
Cons_7a296700: [09:19] OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
Cons_7aa97700: [10:20] SOFTWARE.
Prod_7b298700: 21 lines
Cons: 10 lines
Cons: 11 lines
main: consumer_0 joined with 11
main: consumer_1 joined with 10
main: producer_0 joined with 21
```

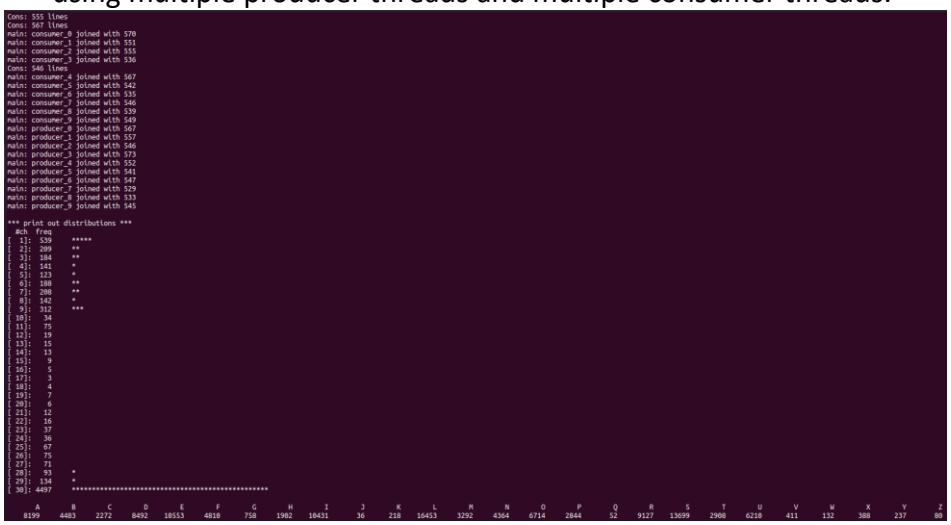
- Producer and Consumer Version 3 reading LICENSE file with single producer threads and single consumer threads.



- Producer and Consumer Version 3 reading LICENSE file with multiple producer threads and multiple consumer threads.



- Reading FreeBSD9-orig/ObsoleteFiles.inc with Producer and Consumer Version 3 by using multiple producer threads and multiple consumer threads.



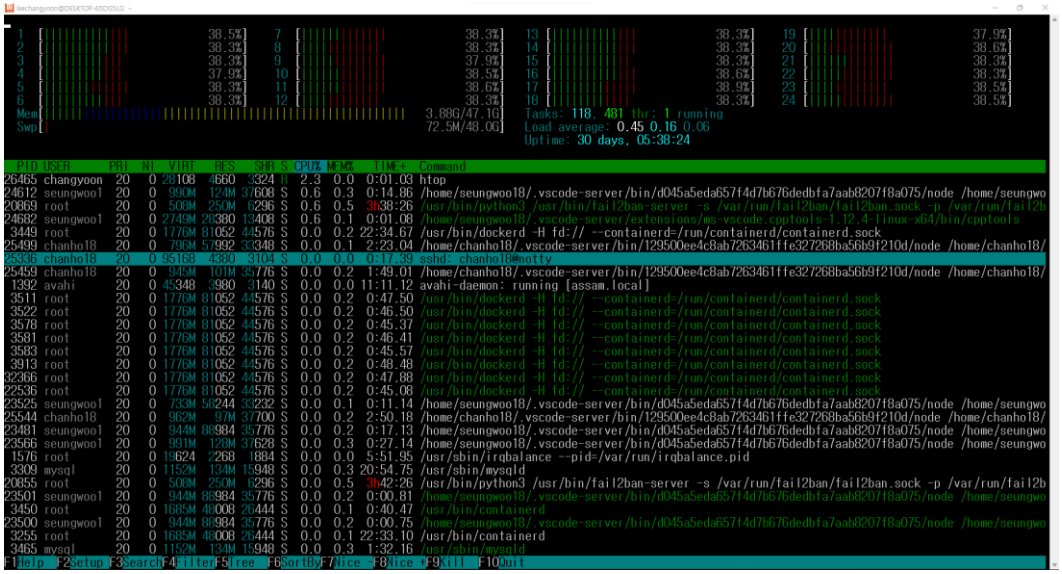
- Tabulated table of the execution time on ObsoleteFiles.inc among the program version 2.1 and 2.2.

	Conditional Mutex	Semaphore
10	0.079	0.017
20	0.087	0.016
30	0.087	0.022
40	0.093	0.023
50	0.096	0.024
60	0.096	0.025
70	0.098	0.026
80	0.092	0.024
90	0.092	0.026
100	0.108	0.027

- Tabulated table of the execution time on FreeBSD9-orig.tar among the program version 2.1 and 2.2.

	Semaphore	Optimization
10	33.513	0.166
20	41.395	0.139
30	41.811	0.125
40	43.717	0.123
50	43.642	0.112
60	43.815	0.111
70	43.794	0.105
80	45.207	0.106
90	44.274	0.106
100	44.461	0.093

- The resource allocation for the producer and consumer program version 2.3 for one hundred producer threads and one hundred consumer threads.



8 Evaluation

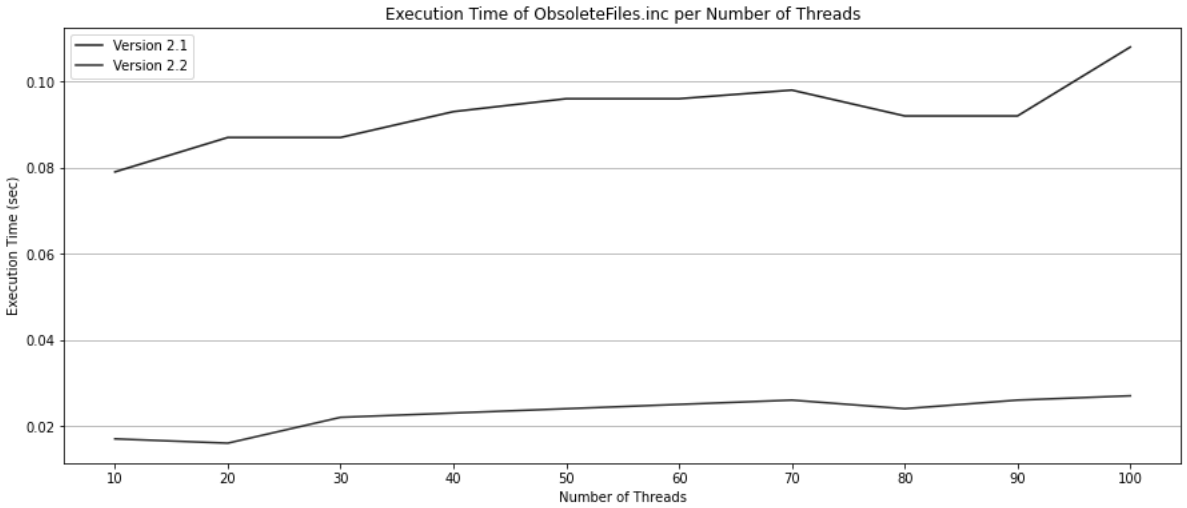


Figure 33 – Execution Time per Number of Threads on Programs of ObsoleteFiles.inc

Figure 33 shows the graph result of execution time per number of the threads for the producer and consumer versions 2.1 and 2.2 program that reads the file of ObsoleteFiles.inc. There are two graphs in the following figure: execution time of program version 2.1 that uses the conditional mutex lock, and execution time of program version 2.2 that uses the semaphores. According to the figure, we can see that there is significant difference of performance between the program version 2.1 and 2.2. As we mentioned, program version 2.1 allows only one thread to wait for the mutex lock to enter the critical section. However, program 2.2 allows multiple threads to wait for the mutex lock to enter the critical section. Due to this difference in operations, program version 2.2, which uses semaphores, shows better performance than the program version 2.1, which uses the conditional mutex locks.

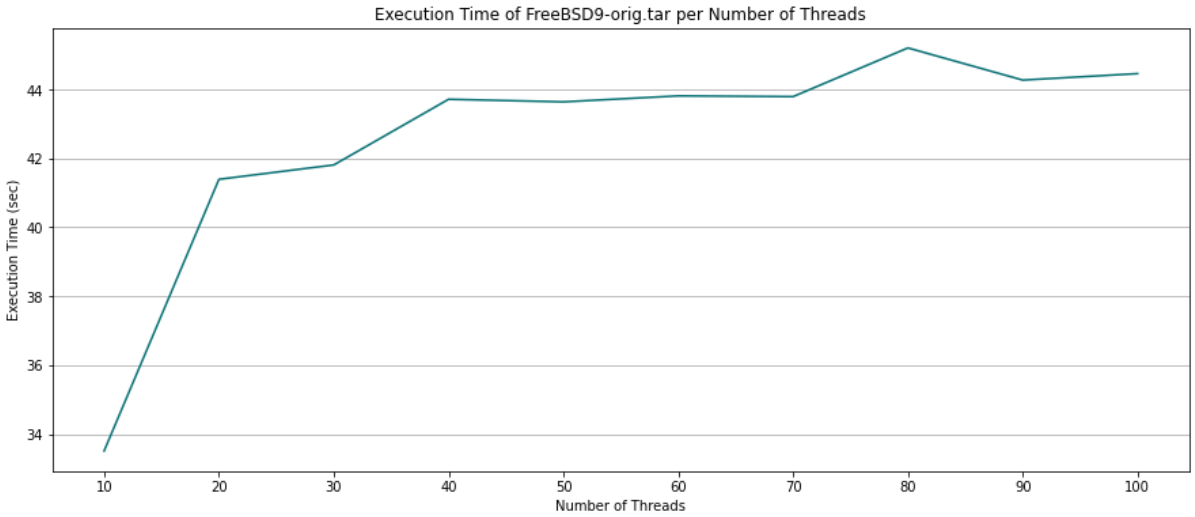


Figure 34 – Execution Time per Number of Threads on Program Version 2.2 of FreeBSD9-orrig.tar

Figure 34 shows the graph result of execution time per number of the threads for the producer and consumer program versions 2.2 that reads the file of FeeBSD9-Orig.tar. The increment of the execution time of the following program in terms of the number of threads is not the ideal result that we expected. The following result shows that the performance of the multithreaded program can get worse if we do not think about the consideration in the multithread program, which also means that the performance is not only enhanced due to the increased number of threads. It means that we must think about the challenges and optimize the following program to perform better. The problem with the producer and consumer program version 2.2 was too much execution of the semaphore post and signal, and mutex lock and unlock operation. To solve the following problem, we implemented the optimized version of the program by applying the following methods: decrease the number of the mutex locks and their number of executions, use file describer to parse the text file into several regions and allocate them to each thread, and extending the reading line into the 4KB sized thread-local buffer.

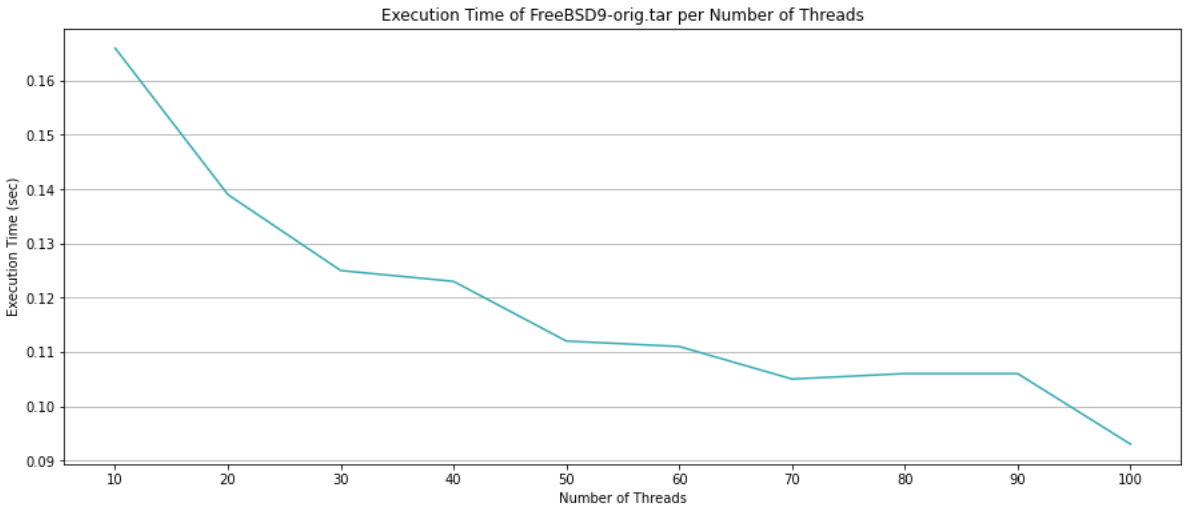


Figure 35 – Execution Time per Number of Threads on Program Version 2.3 of FreeBSD9-orrig.tar

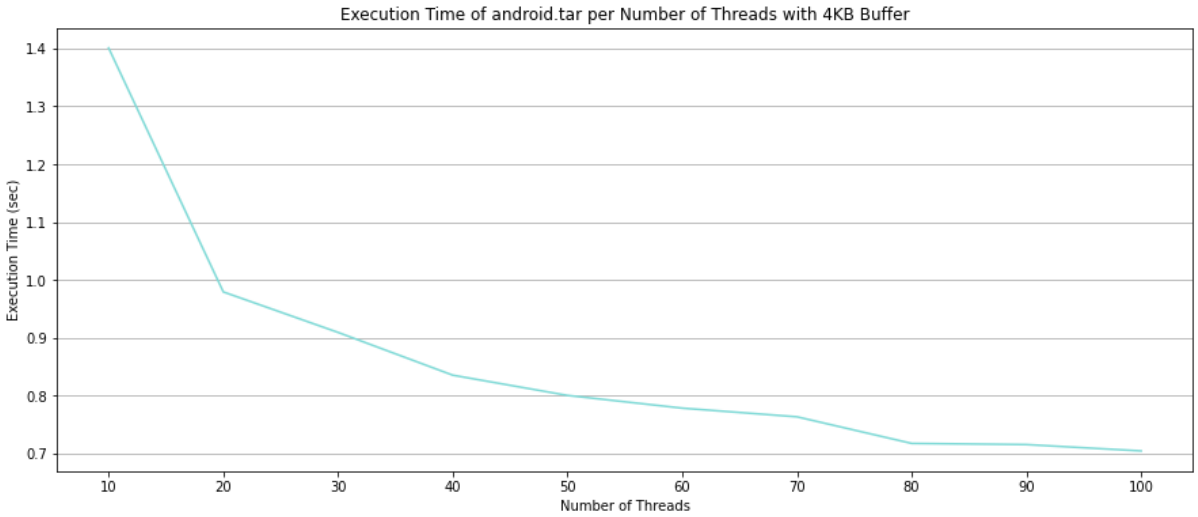
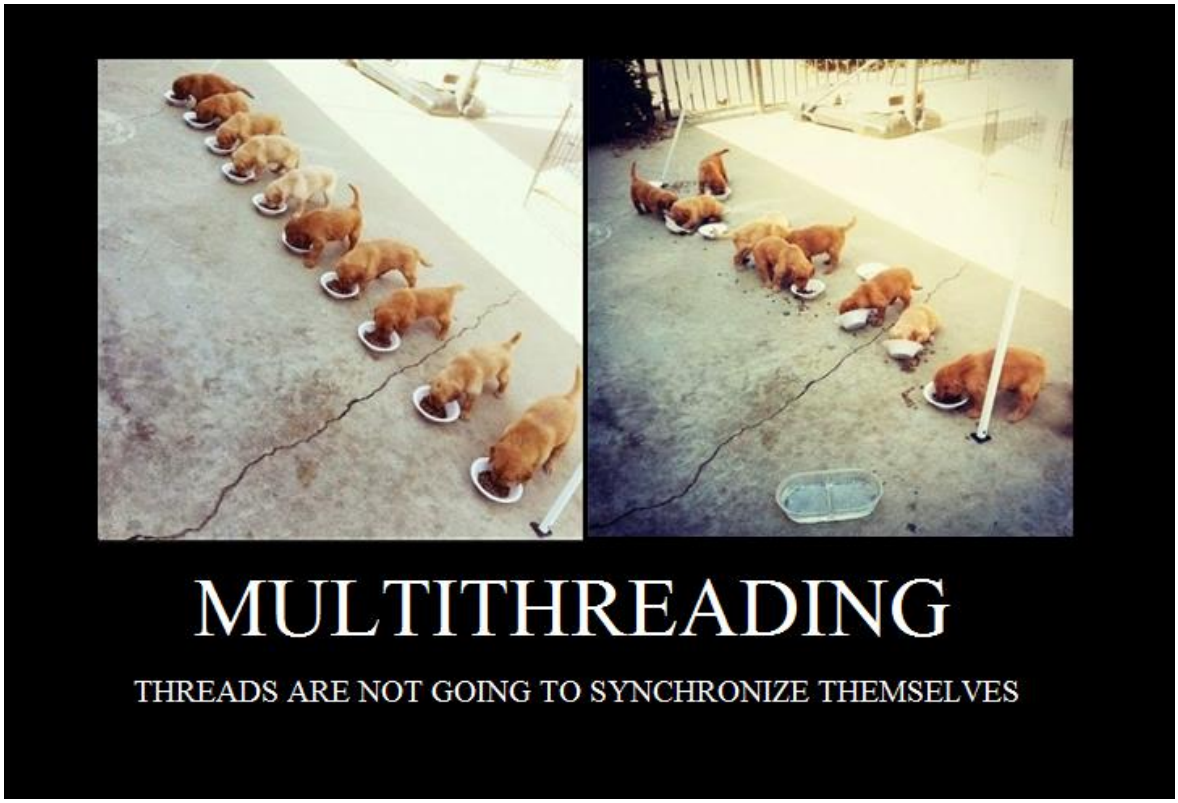


Figure 36 – Execution Time per Number of Threads on Program Version 2.2 of android.tar

Figures 35 and 36 show the graph result of execution time per number of the threads for the producer and consumer program versions 2.3 that reads the file of FeeBSD9-Orig.tar and the android.tar. The following program is implemented by applying the methods that are presented previously. As we can see from the figures, the execution time decreases as the number of threads increases. In short, the producer and consumer program version 2.3 show the ideal and faster result of execution time among the other programs.

9 Conclusion



In this paper, we will first explain the concepts of thread, multi-thread, problems along the multi-threaded programming, and their solutions. By applying these concepts, we will explain how we implemented the multi-threaded word count program and its results for versions 1 through 3. At the end of the paper, we will present the execution time among the differences between the number of threads.

Multi-thread programming is the ability of a program or an operating system to enable more than one user at a time without requiring multiple copies of the program to run on the computer. Multi-thread programming can also manage multiple requests from the same user (Kirvan, 2022). One of the main problems that should be considered in applying multi-thread programming is managing the data dependency among the threads. The data accessed by the tasks must be examined for data dependency on the shared data from other threads. Therefore, we must deal with this data dependency with synchronization along the different threads.

In this paper, we have explained the concept of the thread, multi-thread, and the synchronization of the threads. Then, we discussed the synchronization tools for multi-thread programming, which are applied to the multi-threaded word count program that we have implemented. We looked over the mutex, semaphore, and the most common cause of the synchronization problem, which is the producer and consumer problem. Due to the building environment that we have used, we explained the synchronization tools of the POSIX environment. From the first requirement to the third requirement, we implemented multiple versions of the multi-threaded word count program that fits each requirement. The first version was for single producers and single consumer cases. The second version was for multiple producers and multiple consumers by using different synchronization mechanisms, such as conditional mutex, semaphore, and the optimized version of the mutex. The third version was the complete program that has the character count operations in addition to the second version of the program. At the end of this paper, we evaluated the performance among the different numbers of threads and different implementations for the execution of a multi-threaded word count program. As a result, we found out that the performance of the multi-threaded program does not always increase when the number of threads for the following program increases but decreases when the number of locks and unlock operations of the mutex gets higher. To avoid the following result, we optimized the following implementation by fixing the current mutex synchronization mechanism, parsing the region of the text file, and extending the line to a 4KB thread-local buffer. In short, the optimized version of the program showed the ideal result and the best result among all the programs.

By understanding this paper, we can understand the basic concepts of thread and multi-threaded programming. Also, we can understand the problems and the solutions that occur by applying the following concept, which is about data dependency and synchronization.

Citations

[1] Kirvan, P. (2022, May 26). *What is multithreading?* WhatIs.com. Retrieved September 11, 2022, from https://www.techtarget.com/whatis/definition/multithreading

[2] Silberschatz, A., Galvin, P. B., & Gagne, G. (2014). 2.2.1 Command Interpreters. In Operating Systems Concepts. essay, Wiley.

[3] Yan, D. (2020, December 22). *Producer-consumer problem using mutex in C++*. Medium. Retrieved September 9, 2022, from https://levelup.gitconnected.com/producer-consumer-problem-using-mutex-in-c-764865c47483

[4] Yoo, S. H. (n.d.). Mobile-os-DKU-cis-MSE. GitHub. Retrieved September 8, 2022, from https://github.com/mobile-os-dku-cis-mse/