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 mainly 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 the access of the memory space among the threads which are in the same process (Yoo, Mobile-os-DKU-cis-MSE). This single thread allows the process to perform only one task at a time. However, modern operating systems support the process to have 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 and deadlock. To resolve the following problem, we use several solutions such as queue, mutex, semaphore, and monitors, for the synchronization. For deadlocks, we can avoid them, or detect and resolve them.

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 result for versions 1 through 3. At the end of the paper, we will present the execution time among the difference between the number of threads.

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

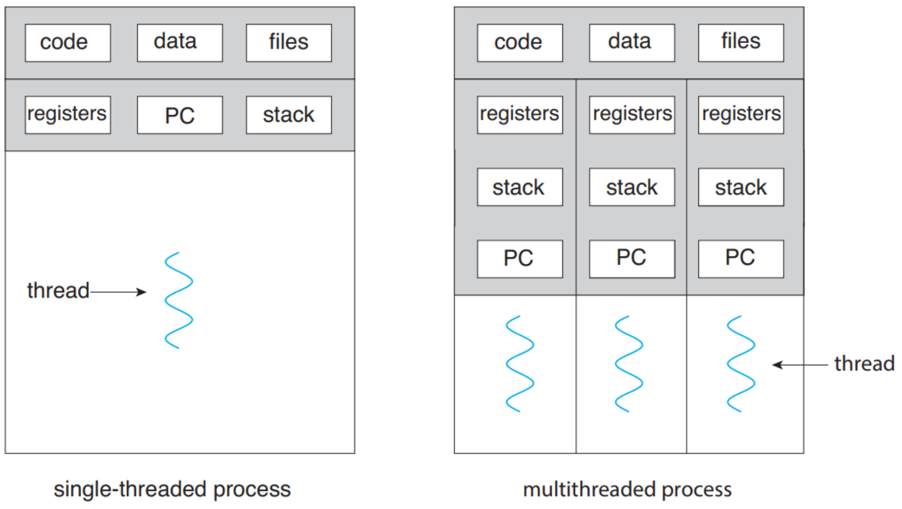


Figure 2 - Single-Threaded and Multi-Threaded Processes (*Silberschatz et al., 2014*)

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.

- Scalability: The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores.

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.

- Balancing: While identifying tasks that can run in parallel, programmers must also ensure that the tasks perform equal work of equal value.

- 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.

- 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

- Testing and Debugging: When a program is running in parallel on multiple cores, many different execution paths are possible

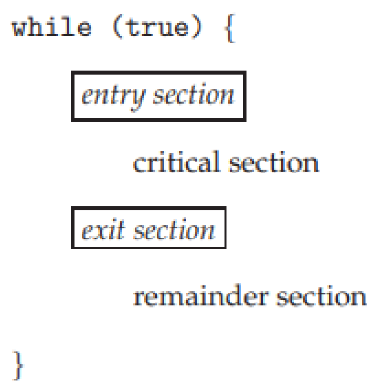
In this paper, we will focus on applying the concept of multi-threaded programming, and the solutions for resolving the presented challenges in the word count program.

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 particular 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 different 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.



```
while (true) {

    entry section

        critical section

    exit section

        remainder section

}
```

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

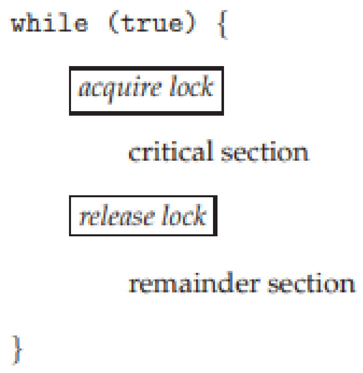
- Bounded Wait: There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted

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.



```
while (true) {
    acquire lock
        critical section
    release lock
        remainder 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 spin-lock 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

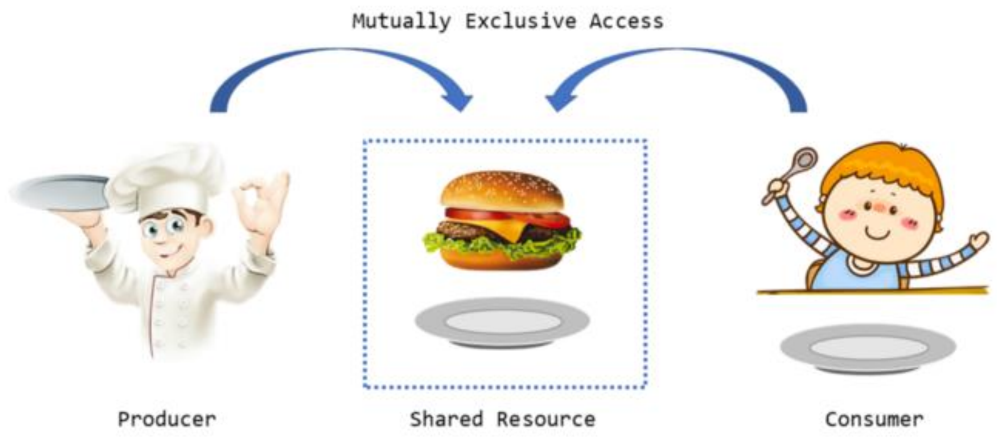
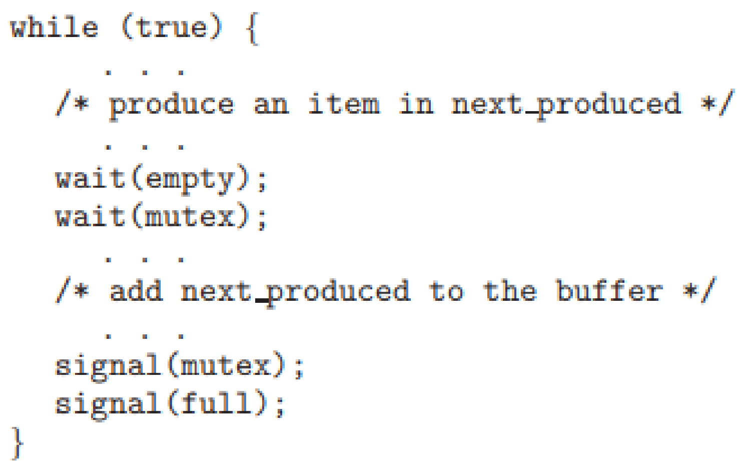


Figure 5 – Producer and Consumer (*Yan, 2020*)

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.

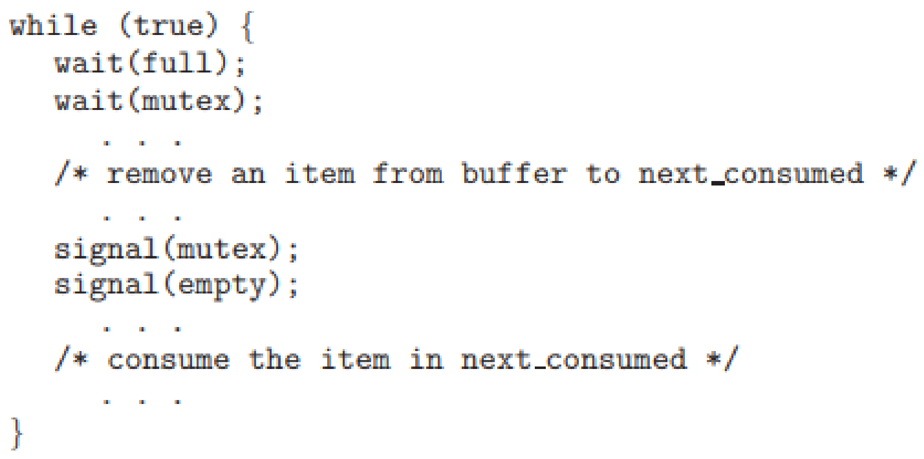
One solution to the producer and consumer problem is using shared memory. To allow producer and consumer threads to run concurrently, the computer must have the available buffer of items that can be filled by the producer and emptied by the consumer. A producer can produce the item while the consumer 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.

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 new item, but the producer can always produce the new 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 possible race condition.

```
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

Figure 10 shows the implemented code of the built-in functions for changing direction and quit commands. First is SiSH_cd function. This function first checks whether its second argument exists or not and prints an error message if there is no second argument. If there is a second argument, it calls the chdir function, checks for error, and returns. The second is the SiSH_quit function. The following function exits the shell.

5.2 Producer and Consumer Version 2

At the start of the following function, it begins tokenizing by calling the strtok function, which stands for string tokenize. This function returns a pointer to the first token. While executing while loop, the shell stores each pointer in an array of the character pointer. The shell reallocates if it is necessary to do it. The process repeats the following operation until there is no token left and returns a null pointer at the end of the operation.

5.3 Producer and Consumer Version 3

Figure 14 shows the Simple Shell Execution Function. The following function checks if the command is equal to the one in the built-in functions, and if it is, the shell runs it. If it does not match a built-in function, it calls the Simple Shell Launch function to launch the process. One consideration is that the argument might just contain NULL if the user inputs an empty string or just white space. Then, the shell must check for that case at the beginning.

6 Build Environment

Following build environments are required to execute the Simple Shell.

- Build Environment:

1. Linux Environment -> Vi editor, GCC Complier

2. Program is built by using Makefile.

- Make Command:

1. $make SiSH -> build the execution program of Simple Shell
2. $make clean -> clean all of the object files that consists of the main function

7 Results

8 Conclusion

Shell is a small program that allows the user to directly interact with the operating system. By typing the commands into the shell, we can create, delete, and copy the file, or run the program. After the input commands are processed by the operating system, the shell waits for the next input command. In this paper, we discussed the Shell that we implemented, the Simple Shell. We first introduced the concepts that are mainly used in the shell program: what is the shell, the basic lifetime of the shell, and the basic loop of the shell. Then, we presented the program definition before the real implementation of the Simple Shell. According to the concepts and program definition that we previously mentioned, we explained the code of the Simple Shell, its operation, and its result. By understanding this paper, we can understand the basic operation of the shell, also known as a command line interpreter, on various operation systems, such as LINUX, UNIX, and Windows.

Citations

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

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

[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