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

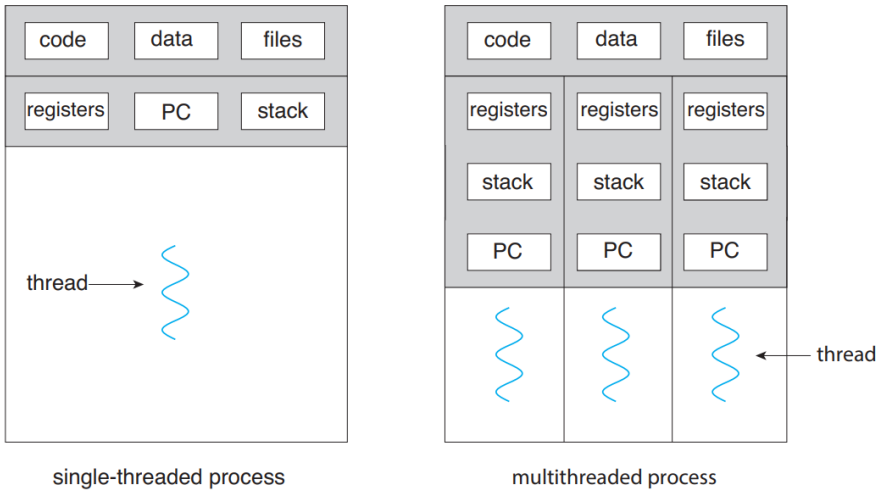

Figure 2 - Single-Threaded and Multi-Threaded Processes (*Silberschatz et al., 2014*)

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 to the incorrect state when the outcome of the execution depends on the particular order on which the access takes place. This situation is called race condition. To avoid the following situation, we need to ensure that only one process at a time can be manipulate the variable count.

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

Each processes and threads have a segment of code which access or update the data that are shared with at least other processes or threads. These segmentations of code are called critical section. 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 refers the critical section, and this is called critical-section problem.

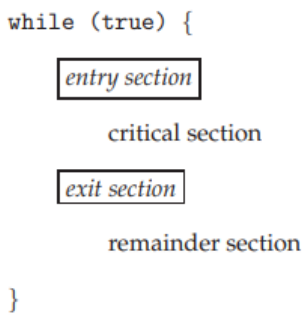

Figure 3 – General Structure of Typical Process (*Silberschatz et al., 2014*)

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 process that is used to resolve the critical-section problem. Each thread must request the permission to enter its critical section. The section of code that requests this permission is called entry section. The critical section may be followed be 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 and satisfies the requirements above. In this paper, we will present two main solution for resolving the critical-section problem, which are 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

Before implementing the Simple Shell (SiSH), we will state the additional program definition that will be used in the real implementation.

4.2 Semaphore

Before implementing the Simple Shell (SiSH), we will state the additional program definition that will be used in the real implementation.

4.3 Producer and Consumer Problem

Before implementing the Simple Shell (SiSH), we will state the additional program definition that will be used in the real implementation.

4.4 Program Definition

Before implementing the Simple Shell (SiSH), we will state the additional program definition that will be used in the real implementation.

- Global Variables

Variables	Data Type	Definition
QUIT	Integer	Variable that is used to check the terminate condition
SHELL_NAME	String	Specifies the name of the shell
builtin_cmd	Array of String	Array that stores the built-in commands that needs additional parameters to execute.

- Modules and Functions

Modules	Functions	Definition
Read	readLine	Function to read a line from the command into the buffer.
	splitLine	Function to split a line into the consistent commands.
Func	numBuiltin	Function that returns the number of implemented built-in commands.
	SiSH_Launch	Functions to create the child process and run the commands.
	SiSH_Exec	Function to execute the command from the terminal.
	readConfig	Function that read and parse the context from the configuration file.
	SiSH_Interact	Function that become active when the Simple Shell is called interactively.
	SiSH_Script	Function that become active when the Simple Shell is called with a script as an argument.
Built	SiSH_cd	Function to operate the change directory command.
	SiSH_exit	Function to operate the quit command.
Main	HU_Init	Main function of the Simple Shell program.

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., & 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/