CS3103 Project (Option A)

The goal of this project:

- Learn how to synchronize the execution of processes and coordinate the use of shared memory segment with POSIX Semaphores.
- Learn how to coordinate the execution of multiple threads in multi-threaded programming with POSIX Thread.
- Learn to understand the file system and related directory/file operations.

Introduction

In this project, you will implement two C programs to solve Problem 1 and Problem 2, respectively. For both problems, your implementation should comply with the proposed requirements. Since completing former problem will help you with the later one more smoothly, it is highly recommended that you implement the two programs for Problem 1 and Problem 2 in order.

To help you get started, some C program files in a compressed file named **project.zip** are available on the Gateway server, which is in */public/cs3103/project/*, including the following files/directories:

```
/project
|── helpers.c
|── helpers.h
|── Makefile          <- It can be used to compile source code.
|── problem1.c        <- The source code file for problem 1.
|── problem2.c        <- The source code file for problem 2.
|── test_case1        <- The test case directory for problem 2.
|── test_case2        <- The test case directory for problem 2.
|── test_case3        <- The test case directory for problem 2.
```

In *helper.h* file, there are some helper functions, which are implemented in *helper.c* file, that may be used for your programs. Please read the comments carefully to understand the functionalities of the functions before using them. The provided *Makefile* can be directly used to compile *problem1.c*. If you want to use it to compile other source files, please modify *Makefile* before using it. For Problem 1, we provide the sample code in *problem1.c* about how to create process and use shared memory segment and semaphores. Please implement your program for Problem 2 in the file named *problem2.c*, where we have provided the necessary coding logic.

Tips: You can also learn how the shared memory and POSIX semaphore work and how to use them in your program from online pages. Here are some useful links:

- Shared memory: https://man7.org/linux/man-pages/man7/shm_overview.7.html
- Semaphore: https://man7.org/linux/man-pages/man7/sem_overview.7.html

The following table lists some functions with brief description that may be used in this project:

shmget()	Allocate a shared memory segment
shmat()	Shared memory attach operation
shmdt()	Shared memory detach operation
shmctl()	Perform the control operation for a shared memory
sem_open()	Initialize and open a named semaphore
sem_init()	Initialize an unnamed semaphore
sem_wait()	Lock a semaphore
sem_post()	Unlock a semaphore (similar to *signal()* in lecture)
sem_close()	Close a named semaphore
sem_unlink()	Remove a named semaphore
sem_destroy()	Destroy an unnamed semaphore
pthread_create()	Create a new thread
pthread_join()	Join with a terminated thread
pthread_exit()	Terminate the calling thread

Problem 1

1. Introduction

This problem aims to help you understand how the shared memory and semaphores are used among processes (or threads). Please implement your program in *problem1.c*. In this program, we use three types of variables, i.e., the global variable *global_param* (line 22), the local variable *local_param* (line 34), and the shared memory variables *shared_param_p*, *shared_param_c* (line 35). The values of all variables are initialized to **0** (the shared memory is automatically initialized).

We use the *fork()* system call to create a new child process. After that, the parent process and child process will execute their own code independently and concurrently. The parent process catches an input parameter in the command line from terminal (see *2. How to run*). The input parameter should be **a nine-digit decimal number**. Then the parent process assigns the value of the input parameter to the three variables. The child process tries to get the input parameter by reading the values of these variables and print them in terminal. We use the semaphore *PARAM_ACCESS_SEMAPHORE* coordinate the operations on the variables between the processes. In this way, we can ensure that only one process can access the variables at any time, i.e., **mutual exclusion**.

2. How to run?

Compile *problem1.c* with the provided Makefile:

```
$ make
```

or by typing:

```
$ gcc problem1.c helper.c -o problem1 -I. -pthread
```

If everything goes well, run the executable file with command:

```
$ ./problem1 <input_param>
```

3. Questions

Please read code in *problem1.c* and learn how to use the shared memory and semaphore to coordinate the execution of processes. **Please answer the following two questions in our project report.**

3.1 Question 1

Recall the introduction about the process creation and how processes communicate with each other (i.e., share data) in the lecture. From line 104 to 106 in *problem1.c*, the child process prints values of the three variables, i.e., *global_param*, *local_param*, and *shared_param_c.* When executing the three lines of code, does the variable *global_param* have the same value as the input parameter? Why? How about the variables *local_param*, and *shared_param_c*? **Please explain your answers.** *(Hint: You can compile the problem1.c program and run the corresponding executable to verify your answer.)*

3.2 Question 2

After finishing Question 1, the child process should have obtained the value of the input parameter. For this question, you will implement a multi-threaded program based on *problem1.c*. All threads try to access the shared data (a **nine-digit decimal number**, in this case). You should try to ensure mutual exclusion when the threads access the shared data to handle the potential issues.

Your program should accept two command-line parameters as shown below:

```
$ ./problem1 <input_param> <num_of_operations>
```

where *input_param* is a nine-digit decimal number, and *num_of_operations* is the number of times executing the addition operation for each thread (more detail in **3.2.1 Requirements e) and f)**).

```
188  /**
189  * This function should be implemented by yourself. It must be invoked
190  * in the child process after the input parameter has been obtained.
191  * @parms: The input parameter from terminal.
192  */
193  void multi_threads_run(long int input_param)
194  {
195      // Add your code here
196  }
```

Please implement your code into the function *multi_threads_run()*, which can be found in line 193 as shown above. Remember to uncomment the function call in line 128, and pass the input parameter when calling *multi_threads_run()*. Your code in *multi_threads_run()* function **must** comply with the following requirements.

3.2.1 Requirements

a) This function can only be invoked in the child process. It should catch a nine-digit decimal number, i.e., the input parameter, which should be obtained from one of the three variables, i.e., *global_param*, *local_param*, and *shared_param_c* (Recall Question 1, you should know which variable(s) has the value of the input parameter).

b) This function (in the master thread of the child process) must create **nine threads** (i.e., child threads) by calling *pthread_create()*. Each thread **must** access (read and write) two adjacent digits among the nine digits of the nine-digit decimal number. See **3.2.2 Explanation** for more detail about the thread-to-digit mapping rule.

c) This function must create **nine semaphores**, each of which should be associated with one digit of the nine-digit decimal number. The semaphore is used to ensure that **only one thread** can access the corresponding digit at any time. When a thread tries to access one digit, it must **first** be able to lock the corresponding semaphore.

d) You should design a strategy to ensure **mutual exclusion** when the threads access the digits. Specifically, **only** when a thread can lock two semaphores at the same time, it can access the two corresponding digits and further proceed to execute operations. Please use the semaphore to achieve this. Any other mechanisms are **NOT allowed** and will be graded with zero points.

e) Once a thread can access the two digits, it can read their values and increase each by 1 respectively. If a digit's value reaches ten after adding 1, the thread should reassign modulo 10 of the value to it, since the number is decimal. **DON'T** do carry in operation. Once the thread finishes the above operation, we say it executes an *addition operation*.

f) Once a thread is created or it finishes an addition operation, it must try to lock the semaphores immediately until it finishes all the addition operations (depending on *num_of_operations*).

g) When all nine child threads finish their addition operations, the master thread should write the modified nine-digit decimal number into a text file by calling the *saveResult()* function, which is in *helper.h*. The function has two input parameters, i.e., the name of the text file (must be ***p1_result.txt*** for Problem 1) and a *long int* type number, such as the modified nine-digit decimal number for Problem 1.

⚠Warnings!

- Please define the name of a semaphore as "xxx_GROUP_NAME", where GROUP is your group number, NAME is your full name, and "xxx" can be any characters for your own usage.
- Please close and delete your semaphores in the parent process by calling *sem_close()* and *sem_unlink()* functions. Read the code in *problem1.c* to learn how to do this.

3.2.2 Explanation

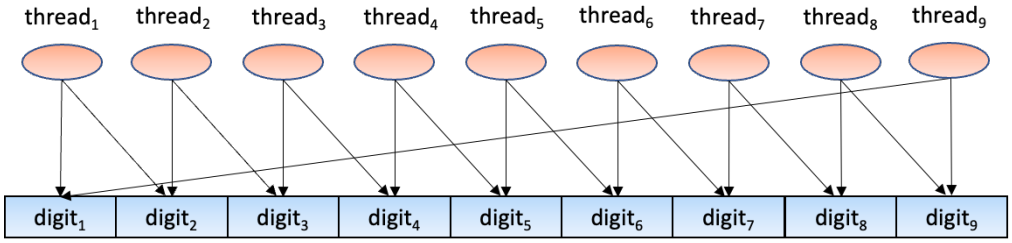
Here we introduce the detail about the thread-to-digit mapping rule (as stated in 3.2.1 **Requirements** b)). For example, the input decimal number is 123456789. We can assign an index to each of the nine digits from the most significant digit to the least significant digit (i.e., from left to right):

- $digit_1 = 1$
- $digit_2 = 2$
- $digit_3 = 3$
- $digit_4 = 4$
- $digit_5 = 5$
- $digit_6 = 6$

- $digit_7 = 7$
- $digit_8 = 8$
- $digit_9 = 9$

Suppose that each thread has an index corresponding to the order in which they were created. For example, the first thread created can be named $thread_1$, the second thread created can be named $thread_2$, and so on until the ninth thread created is named $thread_9$.

Then, the thread-to-digit mapping rule is shown in the following figure. The first $thread_1$ should access the $digit_1$ and $digit_2$, the second $thread_2$ should access the $digit_2$ and $digit_3$, and so on. Note that the ninth $thread_9$ should access the $digit_9$ and $digit_1$.



3.2.3 Hints

1) You can add any other code for your own implementation, such as defining a local variable (in the *main()* function) or a global variable, etc.
2) You can encapsulate any other C functions as auxiliary functions to be called in the function *multi_threads_run()*.
3) You can modify the input parameter of the *multi_threads_run()* function.
4) The master thread should wait for the child threads to finish by calling *pthread_join()*.
5) Your program should be able to handle any nine-digit decimal number as the input parameter. To test the correctness of your program, the grading team will evaluate it with random nine-digit decimal numbers and a random number of times executing the addition operation when grading.

4. Explain your design in project report

In project report, please answer the following two questions:

1) Could your program execute properly? If not, please give the possible reason.
2) How do you handle the thread deadlocks when using the semaphore?

Problem 2

1. Introduction

In this problem, you are required to implement a word counter to automatically count the number of words in text files (with suffix *.txt*). You should achieve this with a C program using two processes (named parent process and child process respectively). The parent process reads text files (with suffix *.txt*) under the given directory (which can be obtained from the input parameter of this program). Then the child process counts the number of words in those text files by calling the *wordCount()* function in *helpers.h (implemented in helpers.c)*. Two processes communicate with each other using a shared memory. You should write your code in the provided *problem2.c* file.

The following table lists some functions may be used for your program implementation:

opendir()	Open a directory
readdir()	Read a directory, return a *dirent* structure
stat(), *lstat()*	Get file/directory status, return a *stat* structure
fopen()	Open a file and associates a stream with it
fseek()	Reset the file position indicator for the stream
fread()	Read the file data
fclose()	Close a file and the stream

2. Requirements

Please carefully read the following requirements, with which your implementation should comply:

a) The parent should first find all the text files (**no more than 100**) under the given directory. Since the number of the text files and their locations in the directory are unknown, you should design your code to traverse the directory to find all the text files under it. Please directly implement this into the *traverseDir()* function. By invoking *traverseDir()* with a given directory, the program should find all the text files and their access paths under the directory.

b) The parent process should read the text files **one-by-one** and write the content of each text file into a shared memory with a **limited buffer size of 1MB**. **Only one** shared memory is allowed to be used in your program.

c) Once the parent process has written the content of a text file into the shared memory, the child process is then able to read the content from the shared memory and count the number of words by invoking the *wordCount()* function (**DON'T** modify this function). After that, the parent process can write data into the shared memory again.

d) The above process must alternate. Specifically, the parent process must first write data into the shared memory, followed by the child process reading the data. Subsequently, the parent process can write data again, and then the child process reads the data. This sequence can repeat multiple times. You should use semaphore to **synchronize** the writing and reading operations of the two processes on the shared memory.

e) Given that the parent process should not modify the shared memory while the child process is reading from it (and vice versa), you should use the semaphore to ensure **mutual exclusion** while the processes access (i.e., read or modify) the shared memory.

f) Finally, the child process should write the total number of words into a text file by calling the *saveResult()* function. The first parameter should be the name of the text file (must be ***"p2_result.txt"*** for Problem 2), and the second one should be the total number of words.

> ⚠**Warnings!**
> - Please detach your processes (including the parent process and child process) from the shared memory by calling *shmdt()* if it will not be used.
> - Please delete the shared memory in the parent process by calling *shmctl()* if it will not be used. Read the code in *problem1.c* carefully to learn how to do this.

3. Hints

1) When the parent process reads the text files, it doesn't matter in what order the files are read.
2) You should handle the case that the total size of a text file exceeds the buffer size and count the total number of words in such file.
3) In Problem 1, the semaphore is used for the purpose of **mutual exclusion**. Mutual exclusion means that at any time only one process can access a "critical section" (the decimal digits in this case) that you want only one process/thread to access at a time. In this Problem 2, you should also think about how to use the semaphore to achieve **synchronization** between the parent and child processes.

4. Test you program

We provide 3 test cases in the compressed file for you to test your program. We will use other test cases when grading your submission. The test cases are just to help you to evaluate your solution. Before submitting your program, please run it on the test cases and check whether it works correctly.

You can compile the files based on the provided *Makefile* (you may modify it before using it for this problem) following this command:

```
$ make
```

or you can directly use the following command:

```
$ gcc problem2.c helpers.c -o problem2 -I. -pthread
```

After compiling, run the executable to test your program by typing command:

```
$ ./problem2 <source_dir>
```

where *source_dir* is the name of a source directory. If your program works correctly, it should be able to save the result, i.e., the total number of words in a given directory, into ***p2_result.txt*** file.

5. Explain your design in project report

In project report, please answer the following four questions:

1) Could your program pass all the provided 3 test cases? If not, please give the possible reason.
2) How you find all the text files under a directory, i.e., the implementation of *traverseDir()* function.
3) How you achieve the synchronization between two processes using the semaphore.
4) How you handle the case that the total size of a text file exceeds the buffer size.

Grading

We will grade your work with the following three considerations:

- Design (Project Report) 30%
- Implementation (Code) 30%
- Evaluation (Test Result) 40%

Design (30%)

Your project report explaining your design details will account for 30% of your project grade. The project report should encompass the following content:

a) **Answer the proposed questions (15%).** Please explain how you address the proposed questions for both problems. Please provide a detailed explanation of the abstract idea behind your design. You can utilize various methods to present your design, such as textual descriptions, flow diagrams, pseudo-code, or any other suitable means.

b) **Implemented functions (15%).** List and explain all the functions you have implemented for both problems. Ensure to include the input and output arguments for each function, as well as a clear description of their functionality. It is recommended to provide comprehensive coverage of all the relevant functions.

You could also discuss any difficulties encountered and the lessons learned in solving them. In case your code cannot be compiled or does not give the correct result, you may still get part of the design scores according to the quality of your project report.

Implementation (30%)

The implementation of your two programs will account for 30% of your project grade. Your code must be written by you own. The code should be nicely formatted with sufficient comments. Each function should be preceded by a header comment that describes what the function does. The code should be easy to read, properly indented, employ good naming standards, and structure, and correctly implement the design. Your code should match your design.

In case your code cannot be compiled or does not give the correct result, you may still get part of the implementation scores according to the quality and degree of correctness of the codes.

Evaluation (40%)

The evaluation of your two programs will account for 40% of your project grade, with 20% for each Problem 1 and Problem 2. Your programs will be tested for correctness validation, ensuring that they work correctly. For each problem, the grading team will use 5 test cases to test your programs when grading. For both problems, each test case is worth 4%. For the sake of fairness, the same 5 test cases for each problem will be used for each group when grading.

Submission

1. Please submit the following files:

- Source code files (with suffixes "*.h*" and "*.c*"), (at least) including *problem1.c*, *problem2.c*, *helpers.h*, *helpers.c*, and Makefile (if any).
- The compiled executable files. The executable files should be named as "***problem1***" and "***problem2***" for two problems, respectively.
- Project report in PDF format based on report template for Project A. It should be named as "**XX-report.pdf**", where XX represents your group number. For example, if you are in group #12, then your submitted report should be named as "12-report.pdf". Please list the names, student numbers, and emails of all group members at the beginning of the project report and describe each team member's contribution.

Please organize and compress the above listed files in a **zip format file** (named as **"XX-project.zip"**, where XX is your group number):

```
/XX-project
|── helpers.c
|── helpers.h
|── ...              <- Any other source code files.
|── Makefile         <- If any.
|── problem1         <- The compiled executable for problem 1.
|── problem1.c       <- The source code file for problem 1.
|── problem2         <- The compiled executable for problem 2.
|── problem2.c       <- The source code file for problem 2.
|── XX-report.pdf    <- The project report file.
```

2. Submit to CANVAS

Self-sign-up is enabled for groups. Instead of all students having to submit a solution to the project, Canvas allows one person from each group to submit on behalf of their team. If you work with partner(s), both you and your partner(s) will receive the same grade for the project.

When you're ready to hand in your work, go to the course site in Canvas, choose the "Assignments" section > "Project" item > "Start Assignment" button, and upload your compressed zip file named as "**XX-project.zip**", where XX is your group number.

Academic Honesty

All work must be developed by each group independently. Please write your own code. **All submitted source code will be scanned by anti-plagiarism software.** We will both test your code and **carefully check your source code**. If your submitted code does not work, please indicate it in the report clearly.

Questions?

If you have questions, please first post them on Canvas so others can also get the benefit of the TA's answer. If the posts on Canvas do not resolve your issue, please contact the TA, Mr. LI Ruoxiang **<ruoxiang.li@my.cityu.edu.hk>.**