# CS Project Report

Student1:

Name: Ng Chi To
Sid: 55742115
Email: chitong2-c@my.cityu.edu.hk
Contributions: Q1, Q2, Q3 and report

Student2:

Name: Wong Hin Leung
Sid: 56245016
Email: hlwong335-c@my.cityu.edu.hk
Contributions: Q1, Q2, Q3 and report

Student3:

Name: Hon Shing Hei
Sid: 56627774
Email: shhon-3@my.cityu.edu.hk
Contributions: Q1 and report

## Table of contents

# Problem1

## *Problem1.1 design idea:*

The program takes an eight-digit decimal number as an input parameter from the command line. It then creates a semaphore using sem_open() for synchronization and a shared memory segment using shmget() for inter-process communication.

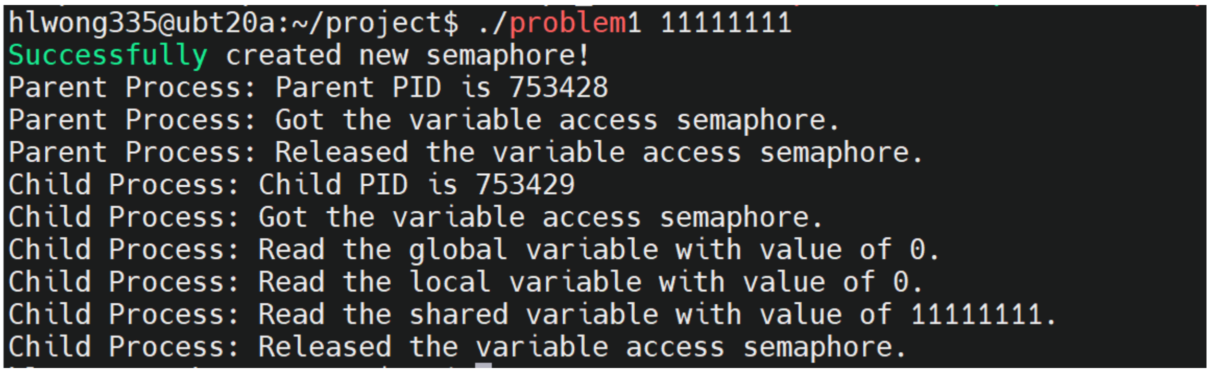
After creating the semaphore and shared memory, the program forks a child process. In the child process, the multi_threads_run() function is called with the input parameter as an argument. The multi_threads_run() function converts the input parameter to an array of integers, initializes an array of eight semaphores and eight threads, and starts the threads. Each thread represents a digit in the input parameter, and each thread updates its own digit and the next digit in the array in a loop. Once all threads finish, the final array is converted back to an integer, and the saveResult() function is called to save the result to a file.

In the parent process, the global_param, local_param, and shared_param_c variables are set to the input parameter, and the parent process waits for the child process to finish using the wait() function. Once the child process has finished, the parent process deletes the shared memory segment using shmctl() and the semaphore using sem_unlink().

The multi_threads_run() function uses sem_init() to initialize an array of eight semaphores. Each semaphore is initialized to 1, which means that it is initially available for use. It then creates an array of eight threads using pthread_create(). Each thread represents a digit in the input parameter, and each thread updates its own digit and the next digit in the array in a loop. The threads use sem_wait() and sem_post() to acquire and release the semaphores, respectively, in order to ensure that only one thread can update a digit at a time.

Once all threads have finished updating the digits, the final array is converted back to an integer, and the saveResult() function is called to save the result to a file. The saveResult() function opens a file using fopen() and writes the result to the file using fprintf(). It then closes the file using fclose().

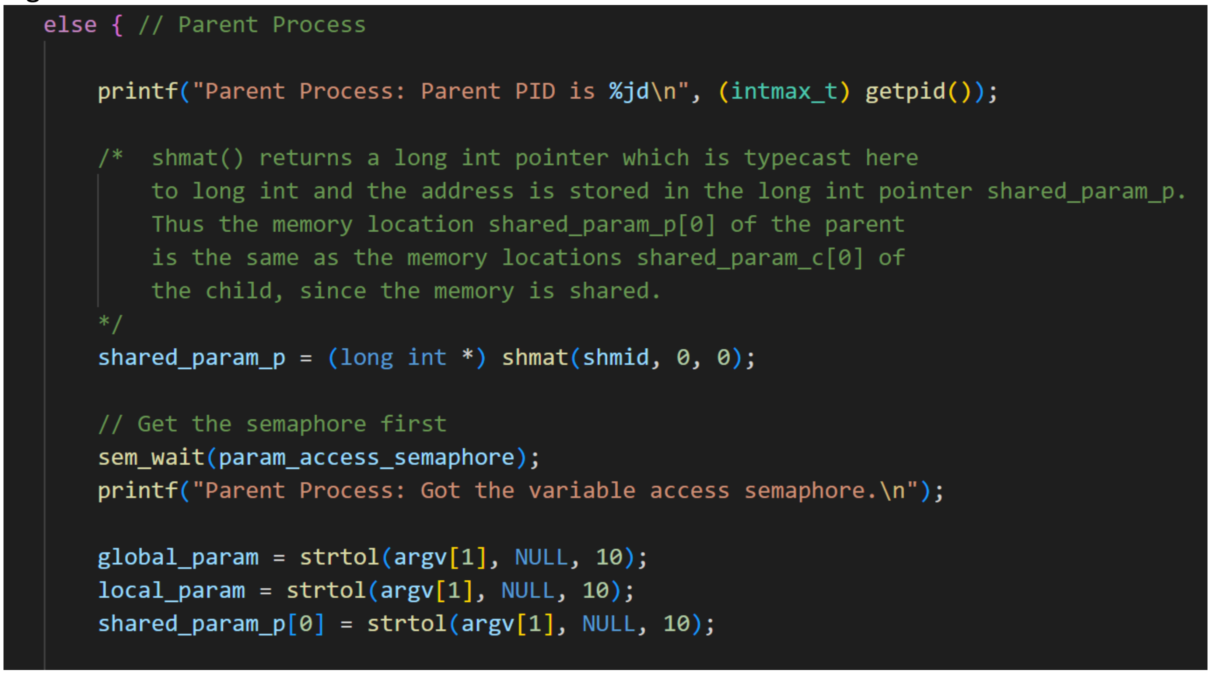
## *Question 1.3.1:*



```
hlwong335@ubt20a:~/project$ ./problem1 11111111
Successfully created new semaphore!
Parent Process: Parent PID is 753428
Parent Process: Got the variable access semaphore.
Parent Process: Released the variable access semaphore.
Child Process: Child PID is 753429
Child Process: Got the variable access semaphore.
Child Process: Read the global variable with value of 0.
Child Process: Read the local variable with value of 0.
Child Process: Read the shared variable with value of 11111111.
Child Process: Released the variable access semaphore.
```

The program will store the input parameter in three places: global_param, local_param, and shared_param_c. However, when the parent process forks into two processes, each process will have its own copy of the global_param and local_param variables. This means that any modifications made to these variables in one process will not affect the values of the same variables in the other process (shown in figure 1.1).

Figure1.1

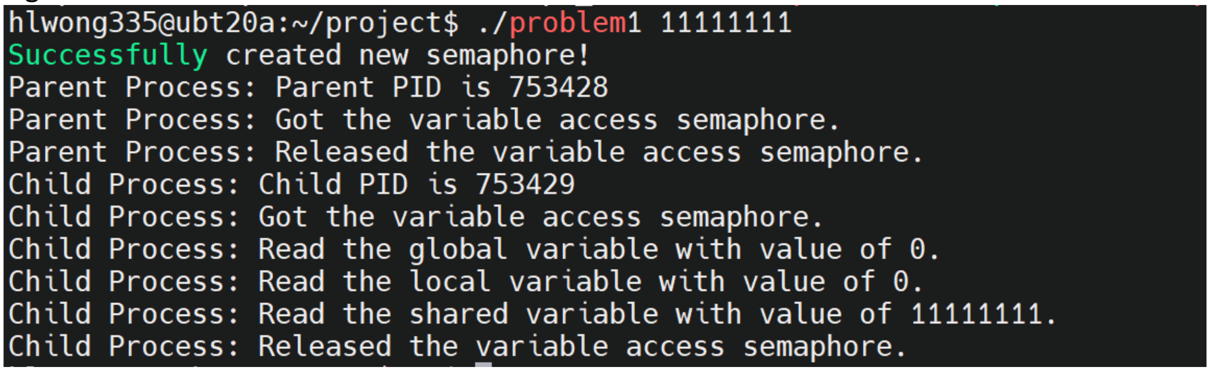


```
else { // Parent Process

    printf("Parent Process: Parent PID is %jd\n", (intmax_t) getpid());

    /*  shmat() returns a long int pointer which is typecast here
        to long int and the address is stored in the long int pointer shared_param_p.
        Thus the memory location shared_param_p[0] of the parent
        is the same as the memory locations shared_param_c[0] of
        the child, since the memory is shared.
    */
    shared_param_p = (long int *) shmat(shmid, 0, 0);

    // Get the semaphore first
    sem_wait(param_access_semaphore);
    printf("Parent Process: Got the variable access semaphore.\n");

    global_param = strtol(argv[1], NULL, 10);
    local_param = strtol(argv[1], NULL, 10);
    shared_param_p[0] = strtol(argv[1], NULL, 10);
```

On the other hand, the shared memory segment created using the shmget function is accessible by both processes, allowing them to communicate and share data. In this case, the input parameter from the console is stored in the shared memory segment, which both processes can access and read.

Therefore, when the program runs, both processes will have their own copies of the global_param and local_param variables, which will be initialized to 0. The shared memory segment will contain the input parameter from the console, which both processes can access and read. As a result, the value of global_param and local_param in each process will be 0, while the value of shared_param_c will be the input parameter from the console. Therefore, when the program is executed with an input parameter of 11111111, the resulting values will be as shown in figure 1.2.

Figure 1.2



```
hlwong335@ubt20a:~/project$ ./problem1 11111111
Successfully created new semaphore!
Parent Process: Parent PID is 753428
Parent Process: Got the variable access semaphore.
Parent Process: Released the variable access semaphore.
Child Process: Child PID is 753429
Child Process: Got the variable access semaphore.
Child Process: Read the global variable with value of 0.
Child Process: Read the local variable with value of 0.
Child Process: Read the shared variable with value of 11111111.
Child Process: Released the variable access semaphore.
```

## *Problem 1.2 Design idea:*

The multi_threads_run() function first converts the input parameter to an array to allow the threads to access one digit at a time. It then initializes an array of 8 semaphores to control access to each digit of the array.

The function creates 8 threads using the pthread_create function and passes each thread a pointer to the child function, along with the index of the digit in the array that the thread should operate on. After all threads have finished execution, the function converts the array of digits back to an integer value and saves it to a file using the saveResult() function. For debugging purposes, the function prints the final array of digits to the console.

The child() function is a function that is called by each of the 8 threads created in the multi_threads_run() function. The child() function takes a single argument, which is the index of the digit in the array that the thread should operate on.

Inside the child() function, a loop is executed times number of times. In each iteration of the loop, the thread first waits on a semaphore associated with its own digit, ensuring that only one thread can modify that digit at a time. The thread then performs an addition operation on its own digit by incrementing it by 1 and taking the result modulo 10 to ensure that the digit remains within the range of 0-9.

After modifying its own digit, the thread waits on a semaphore associated with the next digit in the array (i.e., (id+1)%8), ensuring that only one thread can modify that digit at a time. The thread then performs the same addition operation on the next digit, again taking the result modulo 10.

Once all iterations of the loop have completed, the thread calls pthread_exit to terminate.

## *Question 1.4.1:*

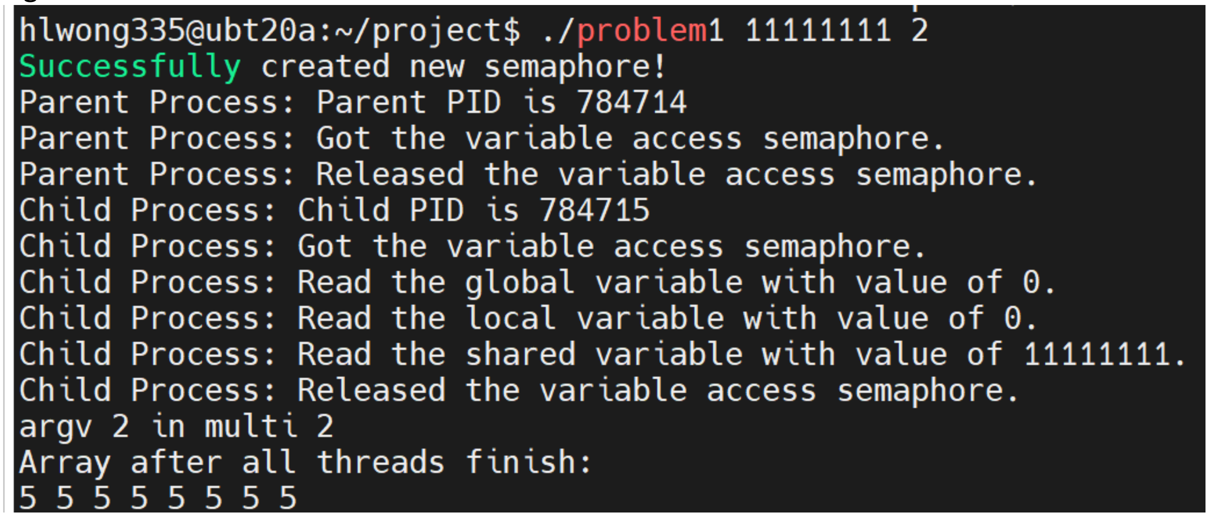
**Could your program execute properly? If not, please give the possible reason.**

The program function properly.

When the input parameters 11111111 and 2 are used, the expected output is 00000000, as shown in Figure 1.3.

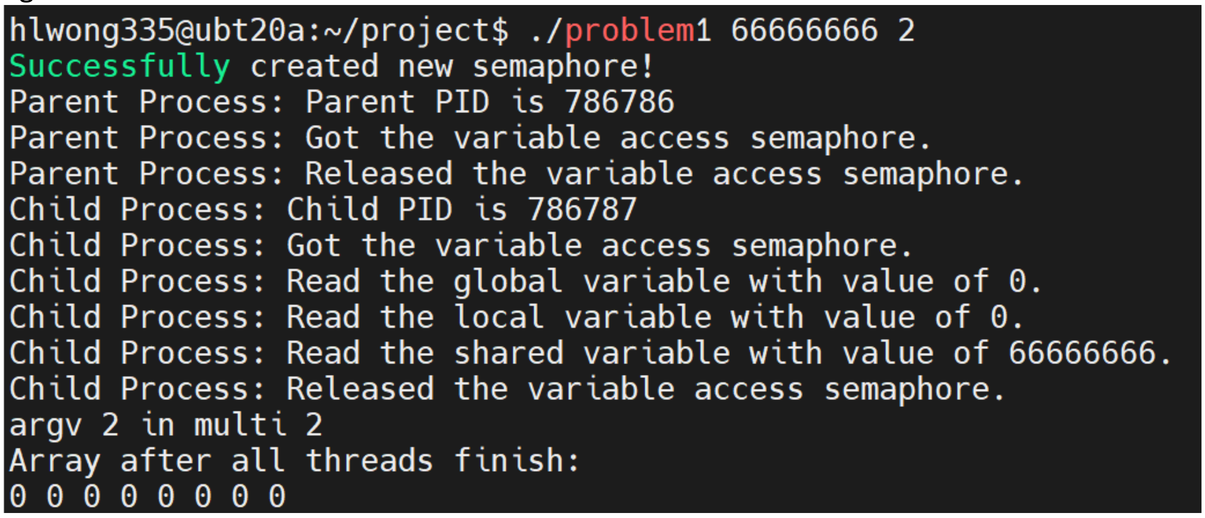
When the input parameters 66666666 and 2 are used, the expected output is 00000000, as shown in Figure 1.4. This test case demonstrates that the modulo 10 function is working correctly, as the output consists of eight zeros, which is the expected result when the input is divisible by 10.

Figure 1.3:



```
hlwong335@ubt20a:~/project$ ./problem1 11111111 2
Successfully created new semaphore!
Parent Process: Parent PID is 784714
Parent Process: Got the variable access semaphore.
Parent Process: Released the variable access semaphore.
Child Process: Child PID is 784715
Child Process: Got the variable access semaphore.
Child Process: Read the global variable with value of 0.
Child Process: Read the local variable with value of 0.
Child Process: Read the shared variable with value of 11111111.
Child Process: Released the variable access semaphore.
argv 2 in multi 2
Array after all threads finish:
5 5 5 5 5 5 5 5
```

Figure1.4



```
hlwong335@ubt20a:~/project$ ./problem1 66666666 2
Successfully created new semaphore!
Parent Process: Parent PID is 786786
Parent Process: Got the variable access semaphore.
Parent Process: Released the variable access semaphore.
Child Process: Child PID is 786787
Child Process: Got the variable access semaphore.
Child Process: Read the global variable with value of 0.
Child Process: Read the local variable with value of 0.
Child Process: Read the shared variable with value of 66666666.
Child Process: Released the variable access semaphore.
argv 2 in multi 2
Array after all threads finish:
0 0 0 0 0 0 0 0
```

*Question 1.4.2:*

**How do you handle the thread deadlocks when using the semaphore?**

In the child() function, each thread first waits on a semaphore associated with its own digit before modifying it. This ensures that only one thread can modify that digit at a time, preventing multiple threads from attempting to modify the same digit simultaneously.

After modifying its own digit, the thread then waits on a semaphore associated with the next digit in the array before modifying it. This ensures that only one thread can modify each digit in the array at a time, and that the threads operate on the digits in a specific order, preventing any circular wait that could lead to a deadlock.

# Problem2

*Problem2 design idea:*

The design idea of the program is to count the total number of words in all text files in a given directory and its subdirectories using shared memory and semaphores for inter-process communication and synchronization between the parent and child processes.

The program first traverses the source directory and its subdirectories to find all text files and record their paths. It then calculates the number of times the child process will read data from shared memory based on the size of the text files and the size of the shared memory segment.

The program then forks a child process, and the parent process and child process run concurrently. In the parent process, data is read from each text file and written to shared memory while using semaphores to synchronize access to the shared memory between the parent and child processes. In the child process, data is read from shared memory, and the total number of words in all text files is counted while also using semaphores to synchronize access to the shared memory between the parent and child processes.

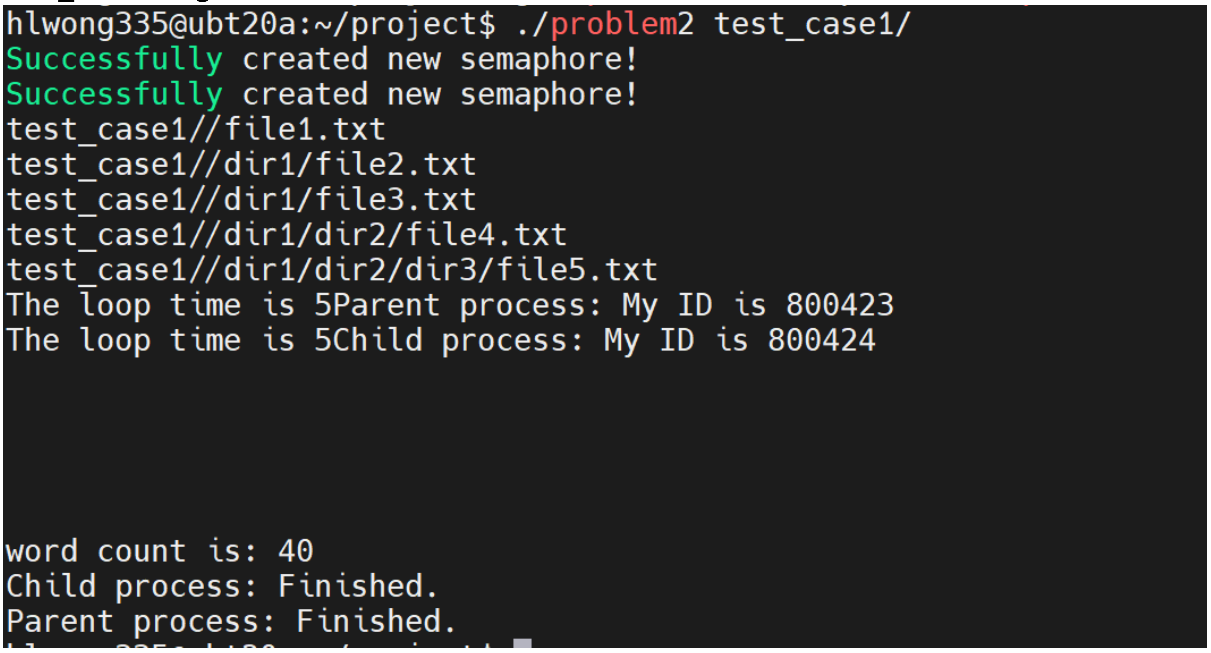
Once the child process has finished counting the number of words, it saves the result to a file, and the program cleans up by deleting the semaphores and freeing the memory allocated for the paths array.

*Question 2.4.1:*

**Could your program pass all the provided 3 test cases? If not, please give the possible reason.**

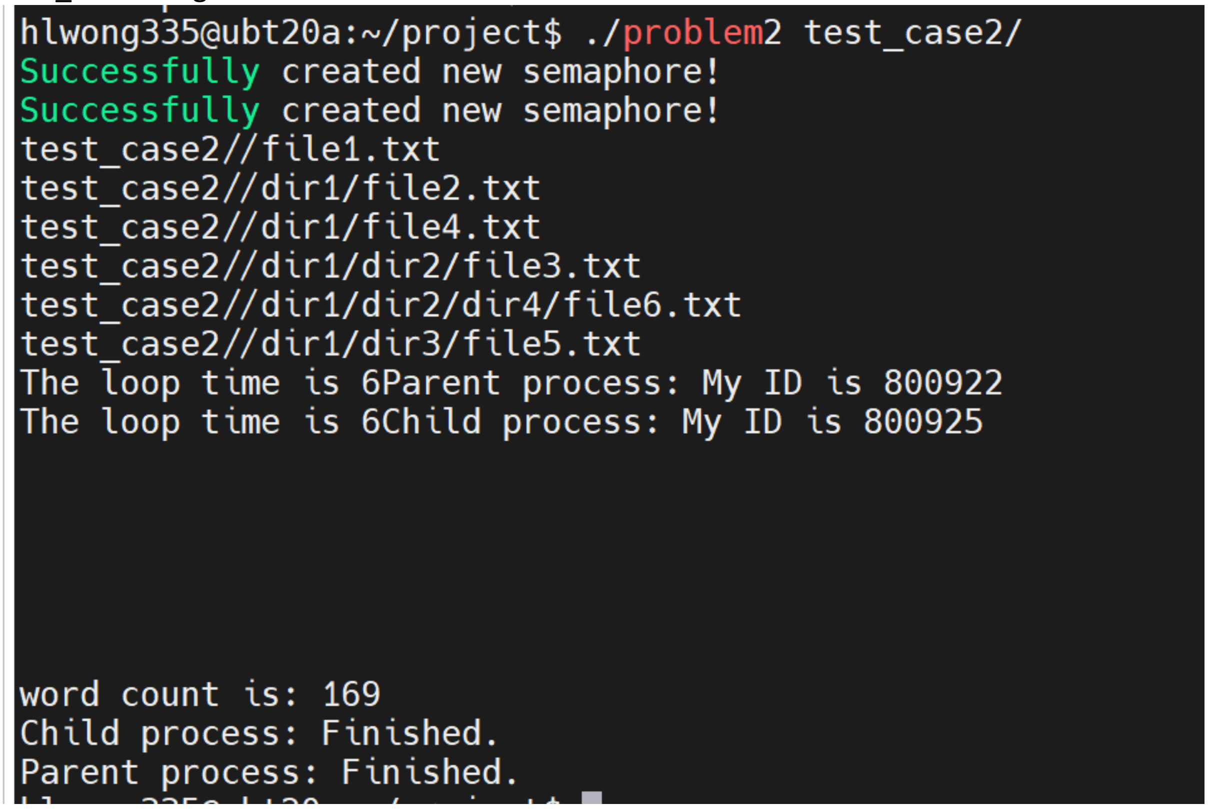
Our program has been thoroughly tested and has successfully passed all test cases.

Test_case1 – figure 2.1:



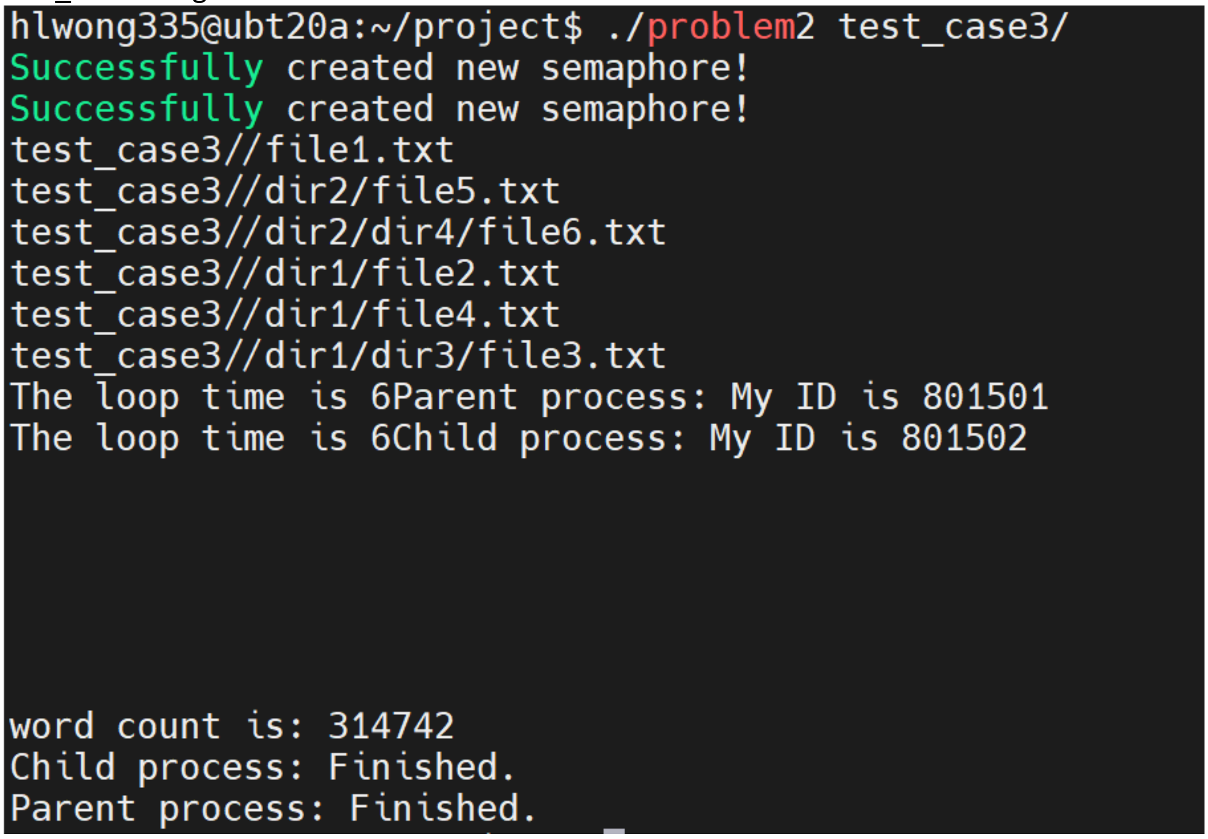
```
hlwong335@ubt20a:~/project$ ./problem2 test_case1/
Successfully created new semaphore!
Successfully created new semaphore!
test_case1//file1.txt
test_case1//dir1/file2.txt
test_case1//dir1/file3.txt
test_case1//dir1/dir2/file4.txt
test_case1//dir1/dir2/dir3/file5.txt
The loop time is 5Parent process: My ID is 800423
The loop time is 5Child process: My ID is 800424




word count is: 40
Child process: Finished.
Parent process: Finished.
```

Test_case2 – figure 2.2:



```
hlwong335@ubt20a:~/project$ ./problem2 test_case2/
Successfully created new semaphore!
Successfully created new semaphore!
test_case2//file1.txt
test_case2//dir1/file2.txt
test_case2//dir1/file4.txt
test_case2//dir1/dir2/file3.txt
test_case2//dir1/dir2/dir4/file6.txt
test_case2//dir1/dir3/file5.txt
The loop time is 6Parent process: My ID is 800922
The loop time is 6Child process: My ID is 800925






word count is: 169
Child process: Finished.
Parent process: Finished.
```

Test_case3 – figure2.3:



```
hlwong335@ubt20a:~/project$ ./problem2 test_case3/
Successfully created new semaphore!
Successfully created new semaphore!
test_case3//file1.txt
test_case3//dir2/file5.txt
test_case3//dir2/dir4/file6.txt
test_case3//dir1/file2.txt
test_case3//dir1/file4.txt
test_case3//dir1/dir3/file3.txt
The loop time is 6Parent process: My ID is 801501
The loop time is 6Child process: My ID is 801502






word count is: 314742
Child process: Finished.
Parent process: Finished.
```

*Question 2.4.2:*

**How you find all the text files under a directory, i.e., the implementation of traverseDir() function.**

We create a function named traverseDir to do the job.

First, the function opens the directory using the opendir() function and checks if the directory exists. If the directory does not exist, the function returns.

Next, the function uses the readdir() function to read all the objects in the directory. The function reads each object one by one in a loop and skips over the . and .. directories.

For each object that is not . or .., the function concatenates the current directory path with the object's name to get the full path of the object.

If the object is a directory, the function calls itself recursively with the new directory path. This allows the function to traverse all the subdirectories of the input directory.

If the object is a text file with the extension .txt, the function prints its full path to the console and adds its path to the paths array. The paths array is a dynamically allocated array of strings that stores the paths of all the text files found in the input directory and its subdirectories.

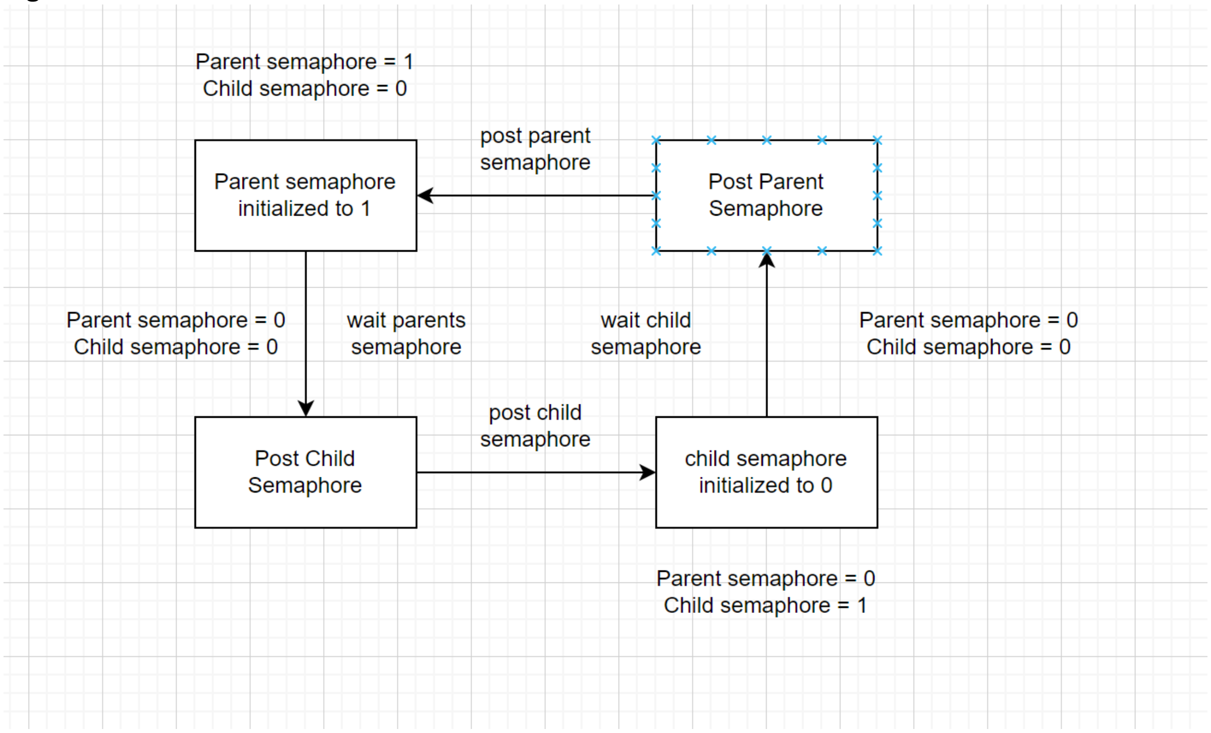
*Question 2.4.3:*

**How you achieve the synchronization between two processes using the semaphore.**
To prevent race conditions, we use two semaphores to synchronize the access of the shared memory between two processes. The parent semaphore is initialized to 1 and the child semaphore is initialized to 0. This ensures that the parent process runs first and puts the word into the shared memory, while the child process waits for the parent semaphore to be released.

Once the parent process has finished writing to the shared memory, it posts the child semaphore, allowing the child process to access the shared memory. When the child process has finished its task, it wakes up the parent semaphore, which allows the parent process to run again. This process continues in a loop until all the words have been written to the shared memory.

By using semaphores, we ensure that the child process does not run before the parent process and that there are no race conditions when accessing the shared memory. This allows us to safely share data between processes in a concurrent environment, the flow chart can be referred to figure 2.4.

Figure 2.4

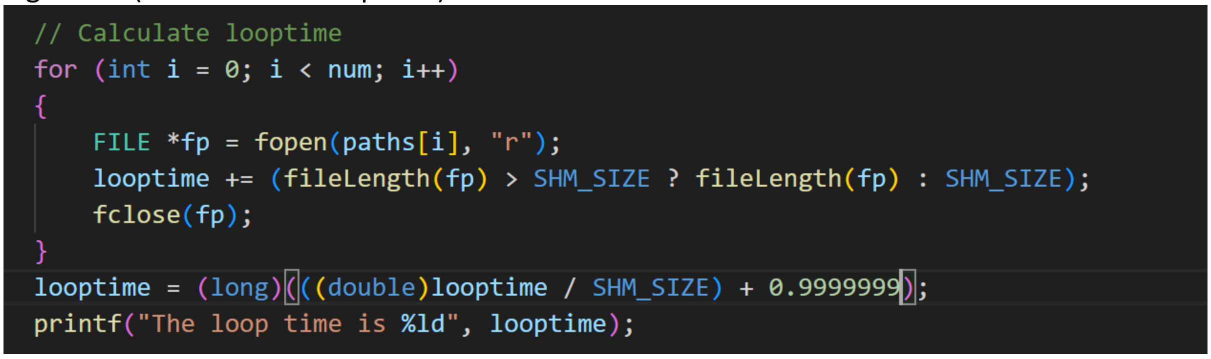


*Question 2.4.4*

**How you handle the case that the total size of a text file exceeds the buffer size.**

To handle situations where the file size exceeds the buffer size of 1MB, we use the filelength function provided by helper.c to calculate the total size of all the files. We then divide this total size by 1MB and take the ceiling of the result to determine the number of times the child process needs to run.

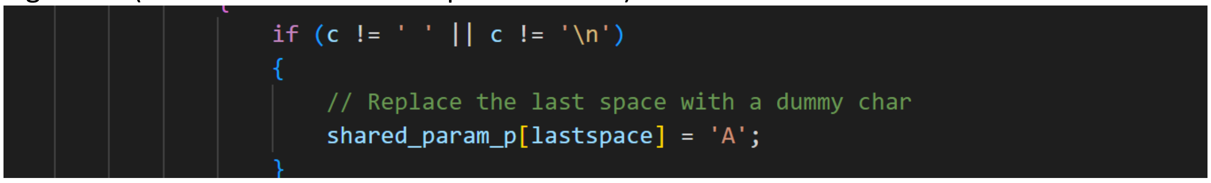
Figure 2.5(Calculate the looptime)



```
// Calculate looptime
for (int i = 0; i < num; i++)
{
    FILE *fp = fopen(paths[i], "r");
    looptime += (fileLength(fp) > SHM_SIZE ? fileLength(fp) : SHM_SIZE);
    fclose(fp);
}
looptime = (long)(((double)looptime / SHM_SIZE) + 0.9999999);
printf("The loop time is %ld", looptime);
```

Also, we check if the character is part of a word or not. If it is part of a word, we replace the previous space character with a dummy character to ensure that the word is not split in two, the word count will be incremented twice instead of once. We then wake up the child process to count the shared memory so that we can calculate the correct number of words.

Figure 2.6(Determine character is part of a verb)



```
if (c != ' ' || c != '\n')
{
    // Replace the last space with a dummy char
    shared_param_p[lastspace] = 'A';
}
```

This approach ensures that all the words in the files are counted correctly and that no words are split in two due to exceeding the buffer size. By handling these cases properly, we ensure that the program runs correctly and produces accurate results.

# Problem3

*Problem3 design idea:*
The whole program is similar to problem2, where we only change the child process.

The child process first attaches to the shared memory segment using the shmat() function and initializes some variables. It then creates an array of thread_data structures, each of which contains a pointer to a segment of the shared memory, a thread ID, and a result variable.

The child process waits for all the threads to finish using pthread_join(), and then retrieves the word count from each thread and adds them up to get the total word count for the current iteration of the loop. It then clears the shared memory by setting all its values to '\0'. The reason why we set all the values in the shared memory to '\0' is to clear the memory and ensure that the data from the previous file does not affect the current file. By setting all the values to '\0', we ensure that the memory is empty and ready to receive the next file's data. This is important because different files may have different lengths and contents, and we want to ensure that each file's word count is calculated accurately and independently of the others. Therefore, clearing the shared memory before each file is processed is an important step in ensuring the correctness of the program's output.

In threadFunc(), which is executed by each thread, the segment of shared memory is first copied into a local buffer and its length is determined. If the length is equal to the segment length (262144 bytes), the function checks the last character to see if it is part of a word or not. If it is part of a word, the word count will be incremented twice instead of once, leading to an incorrect result. To prevent this, the function sets the result variable to -1, which subtracts 1 from the incorrect value and ensures that the final result is correct.

Also, in threadFunc(), it is important to check if the shared memory segment is empty or not before calling the wordCount() function. This is because the wordCount() function from helper.c always returns a value of at least 1, even if the input string is empty. Therefore, if the entire shared memory segment is empty (i.e., all the characters are '\0'), the wordCount() function will still return 1, which is incorrect.

To prevent this, the threadFunc() checks if the segment is empty by inspecting the first character of the buffer. If the first character is '\0', then the entire segment is empty, and the function sets the result variable to 0. This ensures that the correct word count is obtained, even when the shared memory segment is empty.

The function then calls the wordCount() function to count the words in the buffer and stores the result in the result variable of the thread_data structure. Finally, it prints the thread ID and the word count to the console and frees the local buffer.

*Question 3.3.1:*

**How you parallelize the word counting operation with multiple threads in the child process.**
To parallelize the processing of the file, we first create a thread_data struct, which is a composite data type that contains the process ID, a result variable to store the return value of the wordCount() function, and a segment to store the portion of shared memory that each thread will process.

To divide the shared memory into four equivalence size arrays or segments, we calculate the segment length as 256KB, which is 1MB divided by 4. The child process creates four threads, and each thread is attached to a unique thread_data struct that contains the thread's segment to process. The thread then executes the threadFunc() function, which copies the segment into a local buffer, determines the length, checks if the segment is empty, and calculates the word count.

After all four threads have completed their processing, the child process waits for all the threads to finish and adds up the results of each thread to obtain the word count of 4 threads for the file. It then clears the value of the result variable in each thread_data struct to 0 to ensure that the value of the previous file does not affect the processing of the next file.

## Difficulties

1. **Managing shared memory**: Working with shared memory can be challenging, as multiple processes or threads may access the same data simultaneously. It is important to ensure that the shared memory is properly synchronized to prevent race conditions and other synchronization issues.

2. **Implementing multi-threading**: Implementing multi-threading can be complex, especially when dealing with shared resources. It is important to carefully synchronize access to shared resources to ensure that data is not corrupted or lost.

3. **Handling large files**: The program is designed to handle large files, which can be challenging to read and process efficiently. Careful attention must be paid to memory management and buffer sizes to ensure that the program does not run out of memory or become inefficient.

4. **Debugging**: Debugging multi-process or multi-threaded programs can be challenging, as issues may occur in unexpected ways and at unpredictable times. It is important to use debugging tools and techniques to identify and diagnose issues quickly.