CS3103 Project A Report

Group Information

Member	Name	SID	Email
#1 (contact person)	LIU Hengche	57854329	hengchliu2-c@my.cityu.edu.hk
#2	WANG Ying	57854262	ywang3843-c@my.cityu.edu.hk
#3	WAN Zimeng	57222904	zimengwan2-c@my.cityu.edu.hk

Note: Please specify each team member's contribution if not all members make significant contributions to this project.

Everyone in the group contributed equally to the project. We have also finished Project B.

Problem 1

Question 1: When executing the three lines (104-106) 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.**

No, the global_param does not have the same value as the input parameter. This is because when a new process is created using fork(), the child process does not share the same memory despite it's a global variable, each of them will have its own separate copy. So the child's global_param = 0, as parent's global_param initial value is 0.

Same reason for the local_param, as the processes have their own memory space, they do not share the same value, child's local_param would be its initial value which is also 0.

Shared_param_c would have the same value as the input parameter. This is because shared_param_c and shared_param_p are long int pointers that point to the same array in the share memory space after the shmat() operation. So when shared_param_p is updated in the parent process, shared_param_c would also be updated with the value of shared_param_p. This way, shared_param_c would have the same value as the input parameter.

Output is as follows:

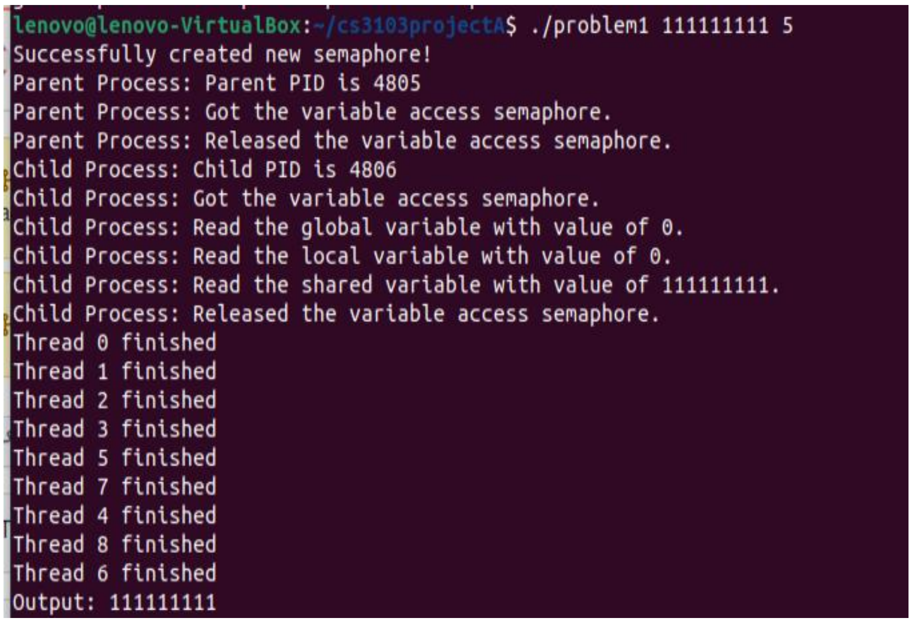
```
hengchliu2@ubt20a:~/project$ ./problem1 123456789
Successfully created new semaphore!
Parent Process: Parent PID is 687288
Parent Process: Got the variable access semaphore.
Parent Process: Released the variable access semaphore.
Child Process: Child PID is 687289
Child Process: Got the variable 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 123456789.
Child Process: Released the variable access semaphore.

Question 2:

1) Have you successfully implemented the program? If not, please provide potential reasons for the issues.

Yes, we have.



```
lenovo@lenovo-VirtualBox:~/cs3103projectA$ ./problem1 111111111 5
Successfully created new semaphore!
Parent Process: Parent PID is 4805
Parent Process: Got the variable access semaphore.
Parent Process: Released the variable access semaphore.
Child Process: Child PID is 4806
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 111111111.
Child Process: Released the variable access semaphore.
Thread 0 finished
Thread 1 finished
Thread 2 finished
Thread 3 finished
Thread 5 finished
Thread 7 finished
Thread 4 finished
Thread 8 finished
Thread 6 finished
Output: 111111111
```

2) How do you handle the thread deadlocks when using the semaphore?

Problem1.c is similar to the eating-philosophers covered in the lecture, where each thread picks up the adjacent "chopsticks". First we tried to make certain semaphores always obtained before other semaphores, but this is not realistic since the number of digits is not even. Then we tried to use the try-wait method. Each thread tries to obtain the locks simultaneously, if it fails to do so, i.e. obtaining only one lock or no lock at all, it would release the lock it's currently holding and sleep. The thread would keep trying until it finally obtains both locks, which means that it could leave the while(1) loop. Corresponding code is as follows:

```
while(1) {
    int getSemResult1 = sem_trywait(semaphores_group_8[index1]);
    int getSemResult2 = sem_trywait(semaphores_group_8[index2]);
    if (getSemResult1 == 0 && getSemResult2 == 0) {
        break;
    }
    else {
        if (getSemResult1 == 0) {
            sem_post(semaphores_group_8[index1]);
        }
        if (getSemResult2 == 0) {
            sem_post(semaphores_group_8[index2]);
        }
        usleep(100);
    }
}
```

3) Implemented functions.

The functions implemented are the thread_function which is a pointer, and the multi_threads_run function.

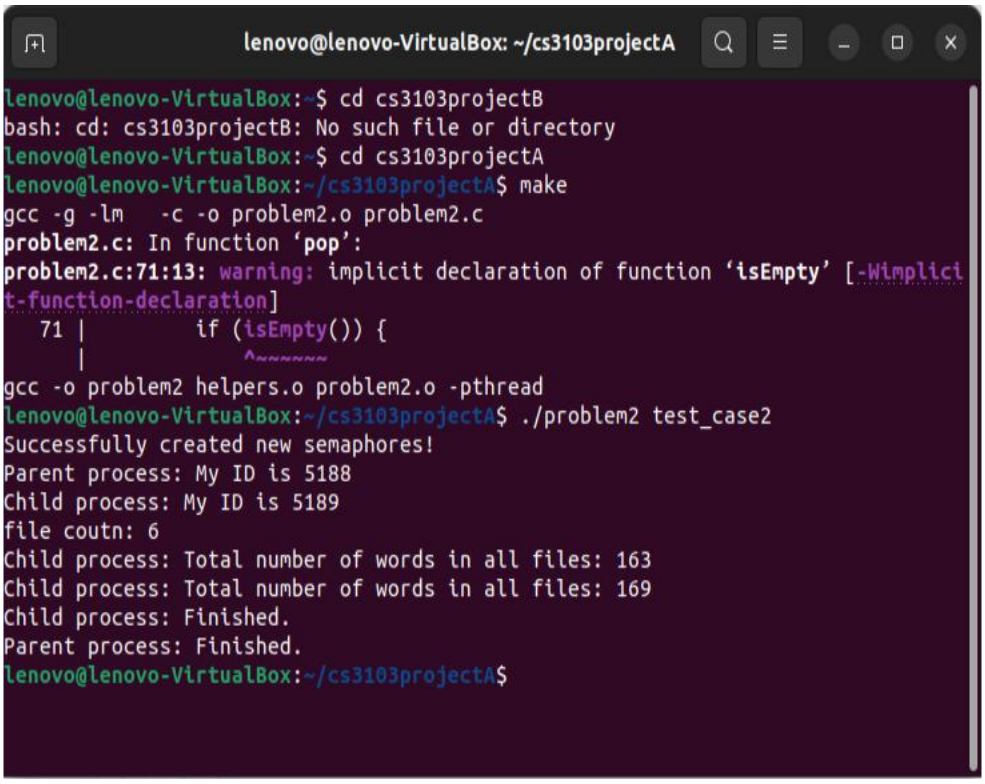
At the start of thread_function, two pointers, digit and traverse, are initialized to point to the same location as para. digit will be used to modify the digits in the array, while traverse will be used to traverse the array. The function then enters a while loop that continues until it encounters -1 in the array. For each integer in the array, it increments pos_to_exit and moves traverse to the next integer. This loop is used to find the position of -1 in the array, which indicates the end of the digits. After the loop, loop_time is set to the integer following -1 in the array, and thread_id is calculated as NUM_THREADS - pos_to_exit. index1 and index2 are then set to thread_id and thread_id + 1 respectively, with index2 wrapping around to 0 if it equals NUM_THREADS. The function then enters another while loop where it tries to acquire two semaphores using the sem_trywait function. If it successfully acquires both semaphores, it breaks out of the loop. If it only acquires one semaphore, it releases it using the sem_post function and then sleeps for 100 microseconds before trying again. This loop ensures that the thread has acquired both semaphores before it proceeds. Once it has acquired both semaphores, the function enters a for loop that runs for loop_time iterations. In each iteration, it increments the digit pointed to by digit and the digit following it, wrapping around to 0 if the digit is 9. If the digit following digit is -1, it instead increments the digit 8 places before digit. After the for loop, the function releases the two semaphores using the sem_post function, prints a message indicating that the thread has finished, and then returns NULL. This function is designed to be run as a separate thread, so it returns NULL to indicate that the thread has finished executing.

Multi_threads_run unlinks any existing semaphores with the names DIGIT_ONE through DIGIT_NINE at first. This is done to ensure that the semaphores are in a clean state before they are created again. Next, it creates nine semaphores using the sem_open function, with the same names DIGIT_ONE through DIGIT_NINE. These semaphores are stored in the semaphores_group_8 array. The sem_open function is called with the O_CREAT and O_EXCL flags, which means that a new semaphore will be created, and if a semaphore with the same name already exists, the call will fail. The S_IRUSR and S_IWUSR flags are used to set the permissions of the semaphore so that the user can read and write to it. The last argument to sem_open is the initial value of the semaphore, which is set to 1 in this case. An array of pthread_t type, threads_group_8, is declared to hold the thread identifiers for the threads that will be created later. The function then extracts the digits from input_param and stores them in the digits_group_8 array. This is done by repeatedly dividing input_param by 10 and storing the remainder in digits_group_8. After storing the digits, the function creates a number of threads equal to NUM_THREADS using the pthread_create function. Each thread is created to execute the thread_function with a pointer to an element of digits_group_8 as the argument. Once all threads have been created, the function waits for all of them to finish executing using the pthread_join function. After all threads have finished executing, the function closes all the semaphores using the sem_close function and then unlinks them using the sem_unlink function. This is done to clean up the semaphores after they are no longer needed. Finally, the function calculates the output by adding up all the digits in digits_group_8 and then multiplying the sum by 10 for each digit except the last one. The output is then printed to the console and saved to a file named "p1_result.txt" using the saveResult function.

Problem 2

1) Have you successfully implemented the program? Has your program passed all the provided 3 test cases? If not, please provide potential reasons for the issues.

Yes, we have.



```
lenovo@lenovo-VirtualBox:~/cs3103projectA
lenovo@lenovo-VirtualBox:~$ cd cs3103projectB
bash: cd: cs3103projectB: No such file or directory
lenovo@lenovo-VirtualBox:~$ cd cs3103projectA
lenovo@lenovo-VirtualBox:~/cs3103projectA$ make
gcc -g -lm   -c -o problem2.o problem2.c
problem2.c: In function 'pop':
problem2.c:71:13: warning: implicit declaration of function 'isEmpty' [-Wimplici
t-function-declaration]
   71 |         if (isEmpty()) {
      |             ^~~~~~~
gcc -o problem2 helpers.o problem2.o -pthread
lenovo@lenovo-VirtualBox:~/cs3103projectA$ ./problem2 test_case2
Successfully created new semaphores!
Parent process: My ID is 5188
Child process: My ID is 5189
file coutn: 6
Child process: Total number of words in all files: 163
Child process: Total number of words in all files: 169
Child process: Finished.
Parent process: Finished.
lenovo@lenovo-VirtualBox:~/cs3103projectA$
```

2) How you find all the text files under a directory, i.e., the implementation of *traverseDir()* function. (*Hint: You may describe the system calls, the data structures, the header files, and any other functions used in your code.*)

First we implemented filestack data structure and tried to do a BFS search to traverse the directory, since the file system is basically a tree structure. But later we figured DFS is much more intuitive so we went for that. At the start of the traverseDir(), it opens the directory specified by dir_name using the opendir function. If the directory cannot be opened, the function returns immediately. The function then enters a while loop that continues until all entries in the directory have been read. In each iteration of the loop, it reads the next directory entry using the readdir function and stores the result in entry. For each directory entry, it constructs the full path to the entry by concatenating dir_name and entry->d_name and stores the result in current_file_group_8. It then gets the status of the entry using the stat function and stores the result in statbuf. If stat returns -1, it prints an error message and exits the program. If the directory entry is a directory (as determined by the S_ISDIR macro), and its name is not "." or "..", it calls traverseDir recursively to traverse the subdirectory. If the directory entry is not a directory and its name is not "." or "..", it checks if the entry is a text file using the *validateTextFile function. If validateTextFile returns 1, it adds the path to the entry to filelist and increments count. After all entries in the directory have been read, it closes the directory using the closedir function. This function is designed to be used in a recursive manner to traverse a directory and its subdirectories to find all text files.*

3) How you achieve the synchronization between two processes using the semaphore.

The read_lock and write_lock are semaphores, which are used to control access to a shared resource, in this case, the shared memory. Semaphores are integer values that are used to signal between processes or threads. They have two main operations: wait (or P), and signal (or V).

In the parent process:
The parent process waits for the write_lock semaphore before it starts writing to the shared memory. This is done using the sem_wait(write_lock) function call. If the write_lock semaphore is 0, this means that the child process is currently reading from the shared memory, so the parent process will block (i.e., stop executing) until the child process has finished reading. After writing to the shared memory, the parent process signals the child process that it can start reading from the shared memory. This is done using the sem_post(read_lock) function call, which increments the read_lock semaphore.

In the child process:
The child process waits for the read_lock semaphore before it starts reading from the shared memory. This is done using the sem_wait(read_lock) function call. If the read_lock semaphore is 0, this means that the parent process is currently writing to the shared memory, so the child process will block until the parent process has finished writing. After reading from the shared memory, the child process signals the parent process that it can start writing to the shared memory again. This is done using the sem_post(write_lock) function call, which increments the write_lock semaphore.

This way, the read_lock and write_lock semaphores ensure that the parent and child processes do not read from or write to the shared memory at the same time, preventing race conditions and ensuring that the data in the shared memory remains consistent.

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

Problem2.c is a producer-consumer scenario where the parent process reads data from a file and writes it to a shared memory buffer, and the child process reads from this buffer. The shared memory buffer has a size of MEM_SIZE.

To avoid the overloading of the buffer, the size of the file is first determined using the fileLength(file) function, and the result is stored in file_size. The parent process then enters a while loop that continues as long as the size of the file exceeds MEM_SIZE - 1. In each iteration of the loop, it reads MEM_SIZE - 1 bytes from the file and writes them to the shared memory buffer using the fread function. It then null-terminates the string in the shared memory buffer by setting the last byte to \0. After writing to the shared memory buffer, it reduces file_size by MEM_SIZE - 1 to reflect the remaining size of the file. It then signals the child process that it can start reading from the shared memory buffer by posting to the read_lock semaphore. It then waits for the write_lock semaphore before it can write to the shared memory buffer again. This is done to ensure that the child process has finished reading from the shared memory buffer before the parent process writes to it again. Once the size of the file is less than MEM_SIZE - 1, it exits the loop and reads the remaining bytes from the file and writes them to the shared memory buffer. It then null-terminates the string in the shared memory buffer, closes the file, and signals the child process that it can start reading from the shared memory buffer. This way, even if the total size of the text files exceeds the size of the shared memory buffer, the parent process can still read the files and write their contents to the shared memory buffer in chunks of MEM_SIZE - 1 bytes. The child process can then read these chunks one at a time, ensuring that the shared memory buffer does not overflow.

5) Implemented functions.

*createNode(char *file_name): This function creates a new node for a stack. It takes a file name as an argument, allocates memory for a new node, assigns the file name to the node, and returns the node.*

initStack(): This function initializes the stack by setting the top of the stack to NULL.

*push(filenode *node): This function pushes a node onto the stack. It takes a node as an argument, sets the next pointer of the node to the current top of the stack, and then sets the top of the stack to the node.*

pop(): This function pops a node from the stack. It checks if the stack is empty, and if it is not, it removes the top node from the stack, saves the file name from the node, frees the memory of the node, and returns the file name.

isEmpty(): This function checks if the stack is empty. It returns 1 if the top of the stack is NULL, and 0 otherwise.

*traverseDir(char *dir_name, char files[][100], int* count): This function recursively traverses a directory and its subdirectories to find all text files. It takes a directory name, a 2D array to store the paths of the text files, and a pointer to an integer to keep track of the number of text files. It opens the directory, reads each entry, checks if the entry is a directory or a text file, and if it is a text file, it adds the path to the file to the 2D array and increments the count.*

*main(int argc, char **argv): This is the main function of the program. It takes the command-line arguments, checks if a source directory name was provided, initializes the stack, calls traverseDir to find all text files in the source directory, creates shared memory segments and semaphores, forks a new process, and then depending on whether it is the parent process or the child process, it reads from the text files and writes to the shared memory, or reads from the shared memory and counts the number of words. After all text files have been read and all words have been counted, it cleans up the shared memory segments and semaphores, and exits.*