CS3103 Group Project Report

Group Information

Group#	Name	SID	Email
	Yancheng Long	40159630	yanchlong3-c@my.cityu.edu.hk

Test Cases

Problem 1

Input 1	Input 2	Output
1234 56789	3	150381296
1521 31413	4	116375231
5701 86532	6	518920712
9837 18385	23	317989894
2182 91883	11	348091861

Problem 2

Test Case	Output(Contents in ***"p2_result.txt"***)	Pass(Yes/No)
test_case1	44	Yes
test_case2	181	Yes
test_case3	314746	Yes

Problem 3

Output	Result
Total Cooking Times (s)	90.00
Total Running Time (s)	210.03

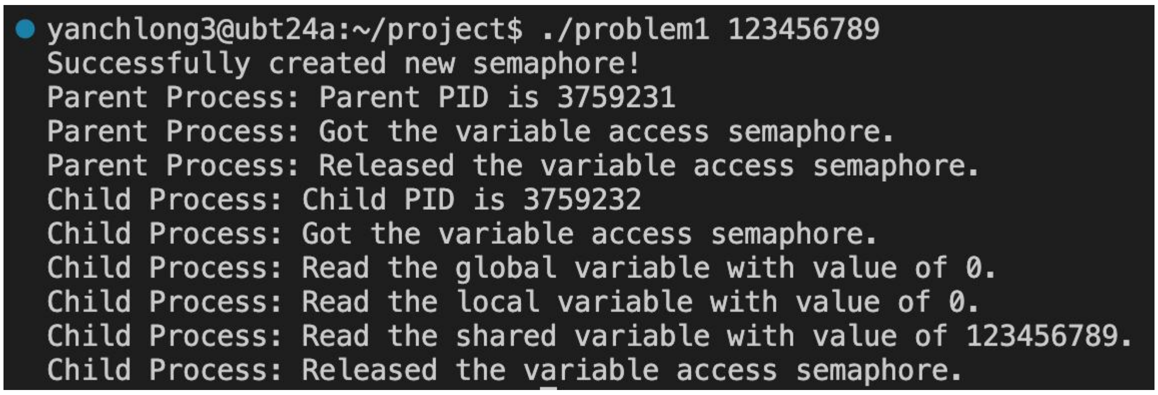
Note: All time data should be the average of 10 times running tested on CSLab server.

Please refer to https://canvas.cityu.edu.hk/courses/59872/assignments/270675 for Submission due.

Problem 1

Explanation Question 1): Could your program execute properly? If not, please give the possible reason.

Yes. Here is my output.



```
● yanchlong3@ubt24a:~/project$ ./problem1 123456789
  Successfully created new semaphore!
  Parent Process: Parent PID is 3759231
  Parent Process: Got the variable access semaphore.
  Parent Process: Released the variable access semaphore.
  Child Process: Child PID is 3759232
  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.
```

Explanation: global_var and local_var are stored separately in the parent and child processes' memory spaces. When the parent process updates these variables, the changes are not reflected in the child process because each process has its own independent copy of these variables.

shared_var_c[0] is different because it resides in shared memory. By using the special system calls shmget() and shmat(), both processes can access the same memory segment. When the parent writes the input value to shared_var_c[0], this value is directly accessible to the child process, allowing data to be shared between the two processes.

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

To handle potential deadlocks in this code when using semaphores, I've implemented the following strategies:

- Consistent Lock Ordering: Each thread locks the two required semaphores (identified as first_digit and second_digit) in a fixed order. This approach avoids circular wait conditions, one of the primary causes of deadlocks. By always locking semaphores[first_digit] first and semaphores[second_digit] second, I ensure that each thread follows the same order, preventing the classic deadlock scenario where two threads wait indefinitely on each other’ s locks.

- Fine-Grained Locking: Each semaphore controls access to a specific "digit" or index in the shared_var array. This design minimizes the duration for which a thread holds a lock, as it only locks the semaphores for two indices, performs the modification quickly, and then releases both locks. This reduces the chances of deadlock due to long-held locks.

- Semaphore Unlinking: Before creating the semaphores, I call sem_unlink() to ensure that any existing semaphore with the same name is removed, avoiding unexpected behavior across different program runs.

Function explanation:

1. Interaction between Parent and Child Processes

- The parent process and child process communicate through shared memory. In the code, shmget creates a shared memory segment, and shmat attaches the shared memory to each process’ s address space.
- The parent process writes a nine-digit number, taken from the command line arguments, into shared memory, while the child process reads this number and performs further operations.
- Semaphores are used to control access to the shared memory, ensuring mutual exclusion. Both the parent and child processes use semaphores when writing to and reading from the shared variable, which prevents data races.

2. multi_threads_run Function for Multithreading Operations

In the child process, the multi_threads_run function is invoked to create nine threads, each performing concurrent operations on specific parts of the shared variable.

- Multithreading Synchronization: Each thread uses two semaphores to control access to specific portions of the shared variable. Before accessing shared data, the thread locks the two relevant semaphores and releases them after completing its operations. This mechanism ensures data consistency and prevents race conditions between threads.
- Modifying Shared Variables: During its task, each thread performs custom addition and modulo operations on specific positions of the shared variable, based on the thread ID. All threads work concurrently on these tasks, ultimately updating the shared variable.

3. thread_function - Executing Concurrent Tasks

Within multi_threads_run, nine threads are created, each running thread_function to handle specific computation tasks.

- Execution Logic:
  Each thread first locks two semaphores to prevent other threads from accessing the specific parts of the shared variable it needs.
  The thread then reads two digits of the shared variable, performs its computation, and writes the result back.
  Finally, it releases the semaphores, allowing other threads to proceed with their tasks.
- Semaphore Control: Semaphores ensure exclusive access in this multithreading environment, preventing race conditions and maintaining data integrity.

4. Termination and Resource Cleanup between Parent and Child Processes

- Shared Memory Cleanup: After the parent process detects that the child process has completed, it detaches from the shared memory and deletes it.
- Semaphore Cleanup: Both the parent and child processes release and delete semaphores upon completion, ensuring proper management of system resources.

Problem 2

Explanation Question 1): Could your program pass all the provided 3 test cases? If not, please give the possible reason.

My program successfully passes all the provided test cases. This includes handling both regular files and files containing special keywords like "math." The synchronization mechanism using semaphores effectively prevents race conditions, and the handling of large files with buffered reads ensures all file data is processed without errors.

Explanation Question 2): How you find all the text files under a directory, i.e., the implementation of *traverseDir()* function.

The traverseDir() function performs a recursive directory traversal using the opendir and readdir functions. It checks each entry within the directory, skipping current and parent directories (. and ..). If a directory entry is found, the function recursively calls itself to continue traversing the subdirectory. For regular files, it identifies text files by checking if their names contain the .txt extension and stores the file paths in the file_paths array up to a maximum of MAX_FILES (100 files).

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

The synchronization between the parent and child processes is achieved using two named semaphores, write_semaphore and read_semaphore. The parent process, after writing a file segment into the shared memory, releases the read_semaphore to allow the child process to read. The child process then performs the word count and releases the write_semaphore, allowing the parent to write the next segment. This alternating mechanism ensures that only one process accesses the shared memory at a time, achieving mutual exclusion and preventing race conditions.

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

To handle files larger than the 1MB buffer size (SHM_SIZE), I implemented a segmented reading approach in the parent process. The parent reads the file in chunks of up to 1MB and writes each chunk sequentially to the shared memory. After each chunk is written, the parent signals the child using read_semaphore to start processing. The child processes each chunk in turn, allowing for word counts across multiple chunks. Once the end of the file is reached, a special "EOF" marker is placed in the shared memory to signal the child that the file is fully processed, ensuring an accurate total count for large files.

Function explanation:

1. traverseDir(char *dir_name)

- Input Parameter: dir_name, the path of the directory to be traversed.
- Output: No direct output, but it stores the paths of .txt files encountered.
- Functionality:

  This function recursively traverses the specified directory dir_name and all its subdirectories to locate files.
  When a regular file ending with .txt is found, it stores the file's path in the global array file_paths for further processing.
  It uses recursion to handle multi-level directory structures, ensuring that all .txt files are identified.
  It utilizes opendir to open the directory, readdir to read directory entries, and strstr to check for .txt file extensions.

2. countLongWords(char *text)

- Input Parameter: text, the text content to be analyzed.
- Output: Returns an integer representing the count of words longer than 5 characters.
- Functionality:

  This function counts the number of words in text that are longer than 5 characters.
  It uses the strtok function to tokenize words using spaces and newline characters as delimiters.
  For each tokenized word, it checks the length, and if the length exceeds 5 characters, it increments the count.
  This function is primarily used for specific files (e.g., filenames containing "math") to count long words and returns this count to the main program for tracking.

3. Alternating Roles of Parent and Child Processes

- Parent Process:

  Reads each file in sequence and writes its content to shared memory.
  Signals the child when the shared memory is ready by releasing a semaphore.
  Waits for the child to finish processing before moving to the next file.
- Child Process:

  Waits for the semaphore from the parent to know the shared memory is ready.
  Reads the file content from shared memory, counts the words, and if applicable, counts long words.
  Signals the parent when it's done so the next file can be loaded.

This alternating sequence ensures each file is processed fully before the next one is written to shared memory.

Problem 3

Explanation Question 1): How did you use semaphores to synchronize the activities of the chefs and the provider?

To synchronize the activities between the provider and the chefs, semaphores were used in the following way:

- Each chef has a semaphore (in an array semaphoreChefs[]), which the provider uses to signal a specific chef that their required ingredients are available. When the provider prepares ingredients, it posts to the respective chef's semaphore, allowing that chef to start cooking.
- semaphoreFinish is a semaphore used by chefs to signal back to the provider once they have finished cooking. This allows the provider to know that it can move on to prepare ingredients for the next chef.
- providerReady is a semaphore initialized to allow the provider to start, setting up synchronization before the chefs start cooking. This semaphore was set up to manage the initial state and ensure the correct sequence of events at the beginning.

Explanation Question 2): How did you handle the termination condition of the program?

The termination condition is based on each chef completing a set number of dishes (DISHES_PER_CHEF). In the provider’ s loop, there is a check to determine if all chefs have completed cooking their required number of dishes. This is achieved by locking a mutex to check the chefCookCount[] array, which stores the number of completed dishes for each chef. Once all three chefs have reached DISHES_PER_CHEF, the provider sets a flag (allCooked = 1) to break out of its loop, allowing the program to terminate gracefully.

Explanation Question 3): How did you use mutex locks to protect shared resources?

Mutex locks are used to protect shared resources, ensuring that only one thread can access or modify them at a time:
A mutex (mutex) protects the shared chefCookCount[] array. This ensures that when the provider or chefs update their count, no other thread interferes, preventing race conditions.
The mutex is also used to check the termination condition and update the total cooking time (totalCookingTime), ensuring that the data remains consistent across threads.

Explanation Question 4): How does your program avoid deadlocks?

The program avoids deadlocks by following a strict order of semaphore operations and maintaining only essential locks:

- Semaphores control the order of operations between the provider and chefs, so that chefs only start cooking after the provider has posted ingredients.

- By using mutex only when necessary (for accessing shared resources like chefCookCount[] and totalCookingTime), the program minimizes the time threads hold onto locks, reducing the risk of deadlocks.
- All semaphores and mutexes are released immediately after use, following a predictable pattern, ensuring no thread waits indefinitely for a resource.

Explanation Question 5): How did you calculate and print the total running time?

The total running time was calculated by recording the start and end times of the program. clock_gettime(CLOCK_MONOTONIC, &startTime); was used at the beginning of the main function, and clock_gettime(CLOCK_MONOTONIC, &endTime); was used after all threads finished execution. The total time was then computed as the difference between endTime and startTime, which was printed as the total running time in seconds. Additionally, totalCookingTime was accumulated across chefs to represent the total cooking time including ingredient wait time, and printed at the end.

Function explanation:

1. Semaphore Usage for Synchronization

- Chef Semaphores: Each chef has a dedicated semaphore (semaphoreChefs[i]). The provider signals these semaphores to let chefs know when ingredients are ready. Chefs wait on these signals to ensure they only proceed when ready.
- Finish Semaphore: Once a chef completes cooking, it signals semaphoreFinish, letting the provider know it can prepare the next ingredients.

2. Program Termination Handling

- Completion Check: The provider checks if each chef has cooked the required number of dishes (DISHES_PER_CHEF). When all chefs are done, it exits the loop, ending the program.

3. Mutex Locks for Shared Resources

- Protected Updates: Shared variables, such as chefCookCount and totalCookingTime, are updated by chefs under a mutex lock to prevent conflicts and ensure accurate data.

4. Deadlock Avoidance

- Sequential Flow: The provider cycles through chefs in order, signaling only one at a time. This organized sequence minimizes waiting and prevents deadlock.
- Efficient Locking: Mutexes are held briefly, reducing blocking and allowing threads to operate without long waits.

5. Timing Calculation

- Total Running Time: Using clock_gettime(), we measure total runtime from start to end.
- Cumulative Cooking Time: Each chef’ s active cooking time is added to totalCookingTime, providing separate outputs for actual cooking time and overall program duration.