**CS2106 Introduction to Operating Systems**

**Lab 3**

**Answer Book**

Please read the instructions in the main lab sheet before completing this document. Submission deadline is **1 pm, Sunday 31 March 2024**. The folder will stay open slightly after this, but once the folder closes, **absolutely no submissions will be allowed.**

**Submission checklist:** A ZIP file called AxxxxxxY.zip, where AxxxxxxY is the student ID of the student submitting. The ZIP file should contain:

* Your answer book, properly renamed.
* Your barrier.c and barrier.h
* Your sum-par.c

|  |  |
| --- | --- |
| **Student 1** | |
| Name: Eunice Koh Shu Ning |  |
| Student ID (A0256986E): |  |
| Group (B08): |  |
| **Student 2** | |
| Name: |  |
| Student ID (AxxxxxxY): |  |
| Group (Bxx): |  |

**Part 1**

Question 1.1 (1 mark)

This pattern indicates that the time quantum for each child process is such that it allows each process to complete its entire execution before the next one gets a chance to run. The time quantum is long enough for a process to complete its task before being preempted.

Question 1.2 (1 mark)

The final counter value is 0 because each child process operates independently of the others, and they each have their own copy of the counter variable. In the parent process, where the final counter value is printed, the counter variable has not been modified by any of the child processes.

Question 1.3 (1 mark)

The output suggests that the time quantum allows each process to execute once before the time quantum expires, preempting the next child process in a round-robin fashion.

Question 1.4 (1 mark)

// Create shared memory segment

if ((shmid = shmget(key, sizeof(int), IPC\_CREAT | 0666)) < 0) {

perror("shmget");

exit(EXIT\_FAILURE);

}

The function shmget() is used to create a shared memory segment. This function allocates a shared memory segment with a specified size (sizeof(int) in this case) and returns a unique identifier for that segment

// Attach shared memory

if ((counter = shmat(shmid, NULL, 0)) == (int \*) -1) {

perror("shmat");

exit(EXIT\_FAILURE);

}

Once the shared memory segment is created, it needs to be attached to the address space of the calling process so that it can be accessed. This is done using the function shmat(), which takes the shared memory identifier (shmid) returned by shmget() and attaches the shared memory to the calling process. It returns a pointer to the attached shared memory segment. In this code, the pointer counter is used to access the shared memory.

\*counter = 0; // Initialize counter

Once the shared memory is attached, the counter pointer provides access to the shared integer variable. Each process (both parent and child processes) can increment this integer variable to maintain a shared counter.

// Detach shared memory

if (shmdt(counter) == -1) {

perror("shmdt");

exit(EXIT\_FAILURE);

}

After the shared memory is no longer needed, it should be detached from the process's address space to free up system resources. This is done using the function shmdt(), which takes the pointer to the shared memory segment (counter) and detaches it from the calling process.

// Remove shared memory segment

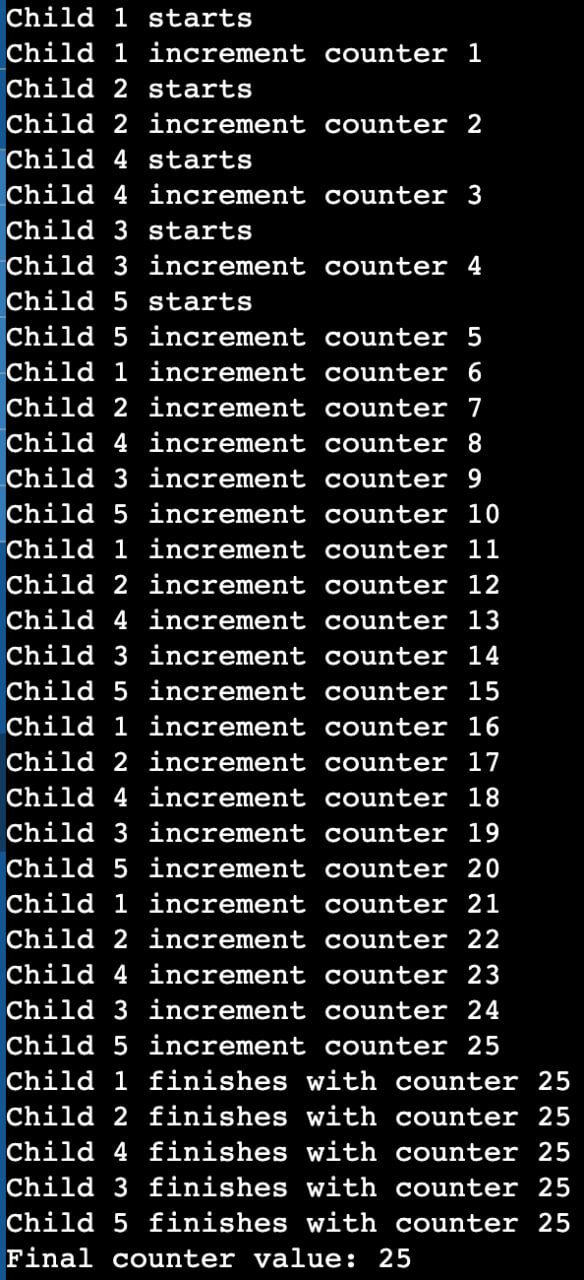
if (shmctl(shmid, IPC\_RMID, NULL) == -1) {

perror("shmctl");

exit(EXIT\_FAILURE);

}

Finally, when the shared memory segment is no longer needed by any process, it can be removed using shmctl(). This function takes the shared memory identifier (shmid) and the command IPC\_RMID to remove the shared memory segment from the system. This step ensures proper cleanup and frees up system resources.



Question 1.5 (1 mark)

With more child processes trying to access and modify the shared counter variable, there's increased competition for this resource. This can lead to race conditions, where the final value of the counter may not be as expected due to unpredictable interleavings of increment operations by different processes.

Question 1.6 (1 mark)

Deadlocks can occur even in scenarios where locks are employed due to their lack of priority control. For instance, consider a situation where Child 3 requires a resource held by Child 2, but due to the absence of priority control, Child 2 is scheduled to acquire the lock last in the sequence, after Child 3. Child 3 may find itself unable to proceed because the resource it needs is held by Child 2, which is yet to acquire the lock. This situation can lead to a deadlock, where Child 3 is blocked, waiting for a resource held by Child 2.

Question 1.7 (1 mark)

// Shared turn variable

int turn = 0;

// Wait for turn

while (turn != i)

; // Do nothing while it's not our turn

// Pass turn to the next process

turn = (turn + 1) % NUM\_CHILDREN;

Shared turn variable: Declared an integer variable turn outside the loop, which serves as the shared turn variable among processes. This variable will be used to control the order of execution for child processes.

Wait for turn: Added a loop inside the child processes to wait until it's their turn to execute. This loop keeps checking if the current value of turn matches the index i of the current child process. If not, it continues looping until it becomes their turn.

Pass turn to the next process: After executing the child process's code block, updated the value of turn to pass the turn to the next process. The turn is passed by incrementing turn and taking the modulo with NUM\_CHILDREN to ensure it cycles back to 0 after reaching the last process index. This ensures fair turn-taking among processes.

Question 1.8 (1 mark)

sem\_init(&sem, 1, 0):

&sem: This is a pointer to the semaphore object that you want to initialize.

1: This is the parameter specifying whether the semaphore should be shared among threads or processes. A value of 1 indicates the semaphore is shared among processes, while 0 indicates it's shared among threads (within a process).

0: This is the initial value of the semaphore. In this case, it's initialized to 0, meaning it starts in a locked state.

sem\_wait(&sem):

&sem: This is again a pointer to the semaphore object.

This function decrements (locks) the semaphore. If the semaphore value is greater than 0, it decrements it by 1 and continues execution. If the semaphore value is 0, it blocks the calling thread or process until the semaphore value becomes greater than 0.

sem\_post(&sem):

&sem: Once again, a pointer to the semaphore object.

This function increments (unlocks) the semaphore. It increases the value of the semaphore by 1. If there are any threads or processes blocked on this semaphore, one of them is unblocked and allowed to proceed.

Question 1.9 (1 mark)

In the provided code, there is no shared memory mechanism established between the parent and child processes. As a result, the communication between them relies solely on the semaphore sem. However, since the semaphore is initialized with an initial value of 0 and there's no mechanism to increment it within the child process before the parent process calls sem\_wait, the parent process is indefinitely blocked at sem\_wait(&sem) waiting for the semaphore to be posted (incremented) by the child process. Since the semaphore is never incremented in this scenario, the parent process remains blocked, causing the program to hang indefinitely, as there's no way for it to proceed without the child process incrementing the semaphore. Therefore, without shared memory or another means of synchronization, such as signaling or inter-process communication, the program will hang due to the lack of progress in the synchronization mechanism.

Question 1.10 (1 mark)

The program initializes an array of semaphores, one for each child process, and forks multiple child processes. Each child process waits for its corresponding semaphore to be signaled by the parent before printing a message indicating it has waited for the parent's signal. Meanwhile, the parent process signals each child process one at a time after waiting for 1 second using the semaphores, ensuring each child executes in order. This synchronization mechanism orchestrates the desired output where the parent's message precedes the messages from each child process, demonstrating effective inter-process communication and synchronization using semaphores and shared memory.

**Part 2**

Question 2.1 (1 mark)

Question 2.2 (1 mark)

**Part 3**

Question 3.1 (1 mark)

(For grader only)

**Report: \_\_\_\_\_\_\_\_\_\_\_\_ / 13**

**Demo: \_\_\_\_\_\_\_\_\_\_\_\_\_ /7**

**TOTAL: \_\_\_\_\_\_\_\_\_\_\_\_\_/20**