**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: | Kenneth Seet |
| Student ID (AxxxxxxY): | A0258173Y |
| Group (Bxx): | B18 |
| **Student 2** | |
| Name: | Daniel Wang |
| Student ID (AxxxxxxY): | A0255689H |
| Group (Bxx): | B11 |

**Part 1**

Question 1.1 (1 mark)

The output suggests that the time quantum used by the operating system scheduler is longer than the time it takes for a child process to complete.

This was indicated by the run displaying that each child process starts, increments the counter 5 times, and finishes with counter 5 with no interleaving between itself and other child outputs, suggesting that the time quantum is longer than the time it takes for the child process to complete.

Question 1.2 (1 mark)

This discrepancy occurs because each child process has its own copy of the counter variable due to the nature of process forking. When a child process modifies its own copy of the counter, it does not affect the counter in the parent process or any other child process.

Question 1.3 (1 mark)

The output suggests that the time quantum is less than 250000 milliseconds. We can observe from the output of the program that control gets handed over from child to child after the counter is incremented, and in the code, usleep(250000) is called after the counter is incremented.

Question 1.4 (1 mark)

We added lines to include the <sys/ipc.h> and <sys/shm.h> header files.

#include <sys/ipc.h>  
#include <sys/shm.h>

Next, we removed the counter variable, declaring an int\* shm instead.

*int* i;  
*int* \*shm;

Next, we created and attached the shared memory region, adding code which handled errors in getting and attaching the shared memory region. We also initialised \*shm, which replaced our counter, as 0.

// create Shared Memory Region  
*int* shmid = *shmget*(IPC\_PRIVATE, *sizeof*(*int*), IPC\_CREAT | *0600*);  
*if* (shmid == -*1*)  
{  
 *perror*("shmget");  
 *exit*(EXIT\_FAILURE);  
}  
// attach the shared memory region to this process  
shm = *shmat*(shmid, NULL, *0*);  
*if* (shm == (*int*\*) -*1*)  
{  
 *perror*("shmat");  
 *exit*(EXIT\_FAILURE);  
}  
\*shm = *0*;

In the else if block of the code which ran for child processes, we replaced the counter variable with \*shm instead, and incremented it accordingly for every iteration of the loop. We also replaced the counter variable in the printf statement with \*shm instead.

*else if* (pid == *0*)  
{  
 // Child process  
 *printf*("Child %d starts\n", i + *1*);  
 // Simulate some work  
 *for* (*int* j = *0*; j < *5*; j++)  
 {  
 (\*shm)++;  
 *printf*("Child %d increment counter %d\n", i + *1*, \*shm);  
 *fflush*(stdout);  
 *usleep*(*250000*);  
 }  
 *printf*("Child %d finishes with counter %d\n", i + *1*, \*shm);  
 *exit*(EXIT\_SUCCESS);  
}

Finally, in the final printf statement which prints the final counter value, we replaced counter with \*shm. After that, we added code which detached and removed the shared memory region, adding code which handled errors in doing so as well.

// Print the final value of the counter  
*printf*("Final counter value: %d\n", \*shm);  
// Detach the shared memory segment  
*if* (*shmdt*(shm) == -*1*) {  
 *perror*("shmdt");  
 *exit*(EXIT\_FAILURE);  
}  
  
// Remove the shared memory segment  
*if* (*shmctl*(shmid, IPC\_RMID, NULL) == -*1*) {  
 *perror*("shmctl");  
 *exit*(EXIT\_FAILURE);  
}

Output:

A screenshot of a computer program

Description automatically generated

Question 1.5 (1 mark)

Increasing NUM\_CHILDREN and the loop variable leads to the final counter variable being smaller than (NUM\_CHILDREN \* loop variable), and this is because of the increased potential for race conditions arising from the number of increments being increased and the number of children being increased, leading to the final counter variable being inaccurate and the written value of the counter variable being overwritten by other children.

Question 1.6 (1 mark)

The lock variable may fail to coordinate the processes as processes can still get preempted while they are using the lock before they get to set the lock back to 1, as the busy wait loop and the writing of 0 to lock[0] are not atomic.

This might lead to multiple processes acquiring control to the lock at the same time, allowing them to access the critical section and leading to race conditions, causing synchronization issues to persist.

Question 1.7 (1 mark)

From question 1.4, we changed the second argument of the shmget function to 2\*sizeof(int) to accommodate the array we will be using.

// create Shared Memory Region  
*int* shmid = *shmget*(IPC\_PRIVATE, 2\**sizeof*(*int*), IPC\_CREAT | *0600*);

We used a shared integer array instead of a shared integer variable, with the first integer in the array representing the counter, and the second integer in the array representing the turn variable. We initialized both as 0.

// shm[0] is the counter, shm[1] is the turn  
shm[*0*] = *0*;  
shm[*1*] = *0*;

We introduced a busy-wait loop in the else-if block which made the processes wait for their turn while it was not theirs. We also had to update the variable in the printf statement to reflect the changes we made to the shared memory.

After the process exits the for loop, we also incremented the turn variable by 1 to allow the next process to start and increment the counter.

*else if* (pid == *0*)  
{  
 // wait if it is not the process' turn  
 *while* (shm[*1*] != i);  
 // Child process  
 *printf*("Child %d starts\n", i + *1*);  
 // Simulate some work  
 *for* (*int* j = *0*; j < *5*; j++)  
 {  
 shm[*0*]++;  
 *printf*("Child %d increment counter %d\n", i + *1*, shm[*0*]);  
 *fflush*(stdout);  
 *usleep*(*250000*);  
 }  
 *printf*("Child %d finishes with counter %d\n", i + *1*, shm[*0*]);  
 // release the turn  
 shm[*1*]++;  
 *exit*(EXIT\_SUCCESS);  
}

Like earlier, we also carried on to detach and free the shared memory.

// Print the final value of the counter  
*printf*("Final counter value: %d\n", shm[*0*]);  
// Detach the shared memory segment  
*if* (*shmdt*(shm) == -*1*) {  
 *perror*("shmdt");  
 *exit*(EXIT\_FAILURE);  
}  
  
// Remove the shared memory segment  
*if* (*shmctl*(shmid, IPC\_RMID, NULL) == -*1*) {  
 *perror*("shmctl");  
 *exit*(EXIT\_FAILURE);  
}  
  
*return 0*;

Question 1.8 (1 mark)

Parameters of sem\_init:

sem: A pointer to the semaphore object that you want to initialize.

pshared: If the pshared argument has a non-zero value, then the semaphore is shared between processes; in this case, any process that can access the semaphore sem can use sem for performing sem\_wait(), sem\_trywait(), sem\_post(), and sem\_destroy() operations.

value: Specifies the initial value of the semaphore. This value represents the number of resources that the semaphore controls access to.

Function of sem\_wait:

sem\_wait decrements (or waits on) the value of the semaphore by 1. If the value of the semaphore is greater than 0, indicating that resources are available, sem\_wait decrements the value and proceeds. If the value is 0, indicating that no resources are available, sem\_wait blocks the calling process or thread until the semaphore's value becomes greater than 0 (i.e., until resources become available).

Function of sem\_post:

sem\_post increments (or posts to) the value of the semaphore by 1. It indicates that a resource previously controlled by the semaphore is now available. If there are any processes or threads blocked in sem\_wait waiting for resources, sem\_post unblocks one of them, allowing it to proceed.

Question 1.9 (1 mark)

After the parent forks the child process, a copy of the semaphore is created in the child process. Even though sem\_post() is called in the parent, the semaphore in the child is not updated as it is not in shared memory, leaving its value stuck at 0, forcing it to wait indefinitely.

Question 1.10 (1 mark)

Our program creates two shared memory regions: one for the counter variable and one for an array of semaphores.

Each child process initialises its corresponding semaphore in the array of semaphores, ensuring that each process has access to its semaphore.

Each child process waits by using sem\_wait() to wait on the semaphore corresponding to its index in the array. After the child process completes its work, it posts its semaphore using sem\_post() to signal the next child process to proceed.

Each child process increments the shared counter variable while ensuring mutual exclusion using the semaphore, ensuring that only one child process can read and write the counter variable at a time, preventing race conditions.

Once all the child processes have exited, the parent process prints the final value of the shared counter variable.

The shared memory regions are all detached and destroyed afterwards.

**Part 2**

Question 2.1 (1 mark)

*void* init\_barrier(*int* numproc) {  
 nproc = numproc;  
 count\_shmid = *shmget*(IPC\_PRIVATE, *sizeof*(*int*), IPC\_CREAT | *0600*);  
 count = (*int*\*) *shmat*(count\_shmid, NULL, *0*);  
 \*count = *0*;  
  
 bar\_shmid = *shmget*(IPC\_PRIVATE, *sizeof*(sem\_t), IPC\_CREAT | *0600*);  
 barrier = (sem\_t\*) *shmat*(bar\_shmid, NULL, *0*);  
 *sem\_init*(barrier, *1*, *0*);  
  
 mutex\_shmid = *shmget*(IPC\_PRIVATE, *sizeof*(sem\_t), IPC\_CREAT | *0600*);  
 mutex = (sem\_t\*) *shmat*(mutex\_shmid, NULL, *0*);  
 *sem\_init*(mutex, *1*, *1*);  
}

We created and attached shared memory regions for the mutex, barrier semaphore, and count variables, and initialised count to 0, barrier with initial value 0, and mutex with initial value 1. Both semaphores were also shared, so the second argument in sem\_init was set to 1.

Question 2.2 (1 mark)

*void* reach\_barrier() {  
 *sem\_wait*(mutex);  
 (\*count)++;  
 *sem\_post*(mutex);  
 *if* ((\*count) == nproc) {  
 *sem\_post*(barrier);  
 } *else* {  
 *sem\_wait*(barrier);  
 *sem\_post*(barrier);  
 }  
}

We protected the count variable with the mutex which was created earlier, and if the count was already at the desired number of processes, the barrier was signalled to allow all processes to proceed.

Otherwise, it waits for the barrier semaphore to be signalled by another process, ensuring that all processes have reached the barrier before allowing any process to proceed.

The use of semaphores and mutex ensures that processes synchronize correctly at the barrier, preventing any process from proceeding until all processes have reached the barrier.

**Part 3**

Question 3.1 (1 mark)

This way of measuring timing may not be fair because the parallel code’s measure of timing does not account for the time the parallel code takes to fork before the timer starts, as well as the fact that the timing does not account for the fact that the start timing might not start from the earliest process to run, leading to an unfair measurement of the timing.

The overhead of forking all the processes and initializing the barrier is also not accounted for.

(For grader only)

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

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

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