**CS2106 Introduction to Operating Systems**

**Lab 3**

**Answer Book**

|  |  |
| --- | --- |
| **Name: Ling Guan Yu** | **Name: NIL** |
| **Student ID: A0308292A** | **Student ID: NIL** |
| **Lab Group: B20** | **Lab Group: NIL** |

**Part 1**

Question 1.1 (1 mark)

Each child process blocks and gives up the CPU when it sleeps which allows a child process created later to print its “<some number> I am child X”. Also, each child process calls “fflush(stdout);” before sleeping which causes the 1 number from “printf("%d ", j);” stored in the output buffer of stdout (stream) to be flushed (need since printing without ‘\n’) so all the “<some number> I am child X” appears before most instead of all of the numbers.

This suggests that the time quantum is longer than the total time needed to call “printf("I am child %d\n", i);”, “printf("%d ", j);” and “fflush(stdout);” in succession.

The interleaving happens as “printf("%d ", j);”, “fflush(stdout);” and “usleep(250000);” called in succession causes each child process to print a number to the screen before it blocks and gives up the CPU due to usleep, after which it is replaced by another child process as decided by the operating system’s scheduler.

The operating system’s scheduling affects the interleaved order of output.

Question 1.2 (1 mark)

The lock variable is not atomic, meaning any operation involving the lock variable can be interrupted by the operating system’s scheduler through preemption. Hence trying to use it as a synchronization mechanism will fail eventually as a race condition can still occur.

Question 1.3 (1 mark)

Step 1:

I created the shared variable “turn” and initialized it to 0 at the start of main().

Step 2.1:

I added “while(turn[0] != i);” above “break;” in the loop used to create processes.

Step 2.3:

I added “turn[0] = turn[0] + 1;” as the last statement to execute in the if-block for “pid == 0”.

Extra Step:

I added code to detach and free the shared variable “turn” as the last statements to execute in the else-block for “pid == 0”.

Turn variables might be less efficient of multi-core systems due to busy waiting occurring on each core from checking the turn variable every frame, wasting CPU cycles per core which could have been used to make progress in other computations.

Question 1.4 (1 mark)

sem\_init params (from left to right):

1st param (pass in address of the semaphore to initialize)

2nd param (pass in 0 for semaphore shared between threads of a process or non-0 for semaphore shared between processes)

3rd param (pass in value to initialize semaphore with)

sem\_wait param:

1st and only param (pass in address of semaphore, value of semaphore is decremented if it is non-0 and call blocks if value of semaphore is 0 until it becomes non-0 [decrement proceeds after] or a signal handler interrupts the call)

sem\_post param:

1st and only param (pass in address of semaphore, value of semaphore is incremented, resulting value is non-0 if no threads or processes were blocked waiting for semaphore to be unlocked or 0 if a thread or process was unblocked and allowed to return from sem\_wait)

Question 1.5 (1 mark)

Since the semaphore is not part of shared memory, when the child process calls sem\_wait, it will block indefinitely as the semaphore has the value of 0 on initialization. Since the parent process calls wait and the child process never exits, the parent process will also be blocked indefinitely, causing the program to hang.

Question 1.6 (1 mark)

Code to declare and initialize an array of semaphores (value of 0 for all) and an array of shmIds was added at the start of main() before the loop where the child processes were created.

In the if-block for “(pid = fork()) == 0” in the for loop, the line “sem\_wait(semaphoreArr[i]);” was added before “break;” for all child processes created to block (happens as all the semaphores have a value of 0).

In the else-block for “pid == 0”, the line “sem\_post(semaphoreArr[i]);” was added before “wait(NULL);” in the for loop to unlock the semaphore for 1 child process sequentially per loop iteration then the “wait(NULL);” is left there for the parent process to wait for the child process that is currently running to finish executing before unlocking the semaphore for the next child in the sequence. This ensures the desired output is produced.

Also in the else-block for “pid == 0”, code for destroying all semaphores, detaching all shared memories and freeing all shared memories was added after the aforementioned for loop.

More processes will mean more shared memory usage with more semaphores since it is currently 1 semaphore per process, leading to a large overhead and huge inefficiencies. Also, a child process must run to completion before another child process can run due to the line “wait(NULL);” in the for loop, further contributing to the problem. Therefore, the current implementation has poor overall scalability.

To rectify this, I would probably modify the code such that all processes will use 1 shared semaphore instead. To maintain the order of printing, I would probably introduce a turn variable that is shared among the processes through shared memory. This turn variable would require its own semaphore to act as a mutex to prevent race conditions.

**Part 2**

Question 2.1 (1 mark)

static int processTotalCount; //nproc

static int processCurrCountShmId; static int\* processCurrCountPtr; //count

static int mutexShmId; static sem\_t\* mutexPtr;

static int turnstile0ShmId; static sem\_t\* turnstile0Ptr;

static int turnstile1ShmId; static sem\_t\* turnstile1Ptr;

void init\_barrier(int num\_proc) {

processTotalCount = num\_proc; //Init processTotalCount (not updated by processes so not shm)

///Init processCurrCount (shm as shared between processes)

processCurrCountShmId = shmget(IPC\_PRIVATE, sizeof(int), IPC\_CREAT | 0600);

processCurrCountPtr = shmat(processCurrCountShmId, NULL, 0);

\*processCurrCountPtr = 0; //Init to 0

///Init mutex (for operations involving processCurrCountPtr)

mutexShmId = shmget(IPC\_PRIVATE, sizeof(sem\_t), IPC\_CREAT | 0600);

mutexPtr = (sem\_t\*)shmat(mutexShmId, NULL, 0);

sem\_init(mutexPtr, 1, 1); //Init to 1 (need to let 1 process through for operations involving processCurrCountPtr so init to 1)

///Init turnstile0 (for reusable barrier)

turnstile0ShmId = shmget(IPC\_PRIVATE, sizeof(sem\_t), IPC\_CREAT | 0600);

turnstile0Ptr = (sem\_t\*)shmat(turnstile0ShmId, NULL, 0);

sem\_init(turnstile0Ptr, 1, 0); //Init to 0 (need all processes to block on this semaphore before last process arrives so init to 0)

///Init turnstile1 (for reusable barrier)

turnstile1ShmId = shmget(IPC\_PRIVATE, sizeof(sem\_t), IPC\_CREAT | 0600);

turnstile1Ptr = (sem\_t\*)shmat(turnstile1ShmId, NULL, 0);

sem\_init(turnstile1Ptr, 1, 1); //Init to 1 (last process will wait on this semaphore so must init to 1 so it becomes 0 when waited then all other processes will block on it)

}

Question 2.2 (1 mark)

void reach\_barrier() { //2 stages to allow for reusability

//\* Stage 0 (Ensures all processes enter “reach\_barrier()” before proceeding)

sem\_wait(mutexPtr); //Lock mutexPtr

++(\*processCurrCountPtr); //Increment \*processCurrCountPtr by 1

if(\*processCurrCountPtr == processTotalCount){ //Last process

sem\_wait(turnstile1Ptr); //Lock turnstile1Ptr

sem\_post(turnstile0Ptr); //Unlock turnstile0Ptr

}

sem\_post(mutexPtr); //Unlock mutexPtr

sem\_wait(turnstile0Ptr); //All non-last processes will wait here until last process unlocks turnstile0Ptr

sem\_post(turnstile0Ptr); //Starts chain of unlocking turnstile0Ptr

//\*/

//\* Stage 1 (Ensures reusability by making val of \*processCurrCountPtr be 0 before all processes exit “reach\_barrier()”)

sem\_wait(mutexPtr); //Lock mutexPtr

--(\*processCurrCountPtr); //Decrement \*processCurrCountPtr by 1

if(\*processCurrCountPtr == 0){ //Another last process (might be diff from Stage 0)

sem\_wait(turnstile0Ptr); //Lock turnstile0Ptr

sem\_post(turnstile1Ptr); //Unlock turnstile1Ptr

}

sem\_post(mutexPtr); //Unlock mutexPtr

sem\_wait(turnstile1Ptr); //All non-last processes will wait here until last process (might be diff from Stage 0) unlocks turnstile1Ptr

sem\_post(turnstile1Ptr); //Starts chain of unlocking turnstile1Ptr

//\*/

}

**Part 3**

Question 3.1 (1 mark)

In the original, the “largest” array and “smallest” array were not in shared memory and hence child processes were updating their own copy of both arrays while the parent process was checking its own copy of both arrays. Also, there was no synchronization done to ensure all the child processes were done updating both arrays before the parent process searches for the largest element in the “largest” array and the smallest element in the “smallest” array.

Question 3.2 (1 mark)

Changed the type of “largest” and “smallest” from an int array to a pointer to an int array. This is to allow for shared memory allocation and attachment. All access to the array elements also were updated to reflect this (e.g. (\*largest)[j] instead of largest[j]).

Added the line “reach\_barrier();” to the end of the if-block for “pid == 0” and also after the line “start = clock();” in the else-block for “pid == 0”. This is to allow for coordination between the parent process and child processes through the use of barriers.

Other code not specified here includes any initialization and deinitialization code required for the aforementioned to function.

Question 3.3 (1 mark)

Steps that were not in the sequential version like overhead from creating multiple processes and that from setting up shared memory were not included when measuring time taken.

**TOTAL:** \_\_\_\_\_\_ / 11