**CS2106 Introduction to Operating Systems**

**Lab 3**

**Answer Book**

|  |  |
| --- | --- |
| **Name: Wu Xiao Yun** | **Name: Ang Koon Hwee** |
| **Student ID: A0221772J** | **Student ID: A0221986W** |
| **Lab Group: B06** | **Lab Group: B06** |

**Part 1**

Question 1.1 (1 mark)

The child processes got switched before it prints the numbers after the `usleep(250000)` in the loop. This suggests that the time quantum is smaller than 250000 microseconds.

Question 1.2 (1 mark)

A child process could read the value of the lock (1) and context switch happens before it is able to write the value of 0 back to the lock. As such, multiple child processes could claim the lock (before the new value of lock is written back to shared memory) and run simultaneously to print the lines “<some number> I am child X”.

Question 1.3 (1 mark)

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <sys/shm.h>

#include <sys/wait.h>

#define NUM\_PROCESSES 5

int main() {

int i, j, pid;

// create a new shared variable for turn

int \*turn;

int shmid;

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

// attach the shared memory region to this process

turn = shmat(shmid, NULL, 0);

turn[0] = 0; // intialize turn to 0

for(i=0; i<NUM\_PROCESSES; i++)

{

if((pid = fork()) == 0) {

break;

}

}

// busy waiting for my turn

while (turn[0] != i);

// it is currently my turn, execute my code

if(pid == 0) {

printf("I am child %d\n", i);

for(j = i\*10; j<i\*10 + 10; j++){

printf("%d ", j);

fflush(stdout);

usleep(250000);

}

printf("\n\n");

turn[0]++; // pass turn to next process

}

else {

for(i=0; i<NUM\_PROCESSES; i++)

wait(NULL);

// detach and destroy shared memory

shmdt(turn);

shmctl(shmid, IPC\_RMID, 0);

}

}

int \*turn;

int shmid; // these two lines create a shared variable for turn

shmid = shmget(IPC\_PRIVATE, sizeof(int), IPC\_CREAT | 0600); // creates a shared memory region with size of one integer

turn = shmat(shmid, NULL, 0); // attach the shared memory region to this process

turn[0] = 0; // intialize turn to 0

while (turn[0] != i); // busy waiting and do nothing when it is not current process’s turn, if executed pass this line, it means that it is current process’s turn and the process will execute the print lines.

turn[0]++; // pass turn to next process after current process has finished executing

// detach and destroy shared memory after all child processes have been executed

shmdt(turn);

shmctl(shmid, IPC\_RMID, 0);

|  |
| --- |
| Question 1.4 (1 mark)  sem\_init(sem\_t \*sem, int pshared, unsigned int value) initializes the semaphore variable pointed by sem to value amount   * sem points to a semaphore variable to be initialized * If pshared is non-zero, then the semaphore can be shared between processes; if pshared is zero, it can be only shared between threads instead of processes. * value is initial value of semaphore   sem\_wait(sem\_t \*sem)   * decrements semaphore value atomically * if the semaphore value <= 0, the calling process blocks * sem points to the semaphore variable to decrement the value   sem\_post(sem\_t \*sem)   * increments semaphore value atomically * if the semaphore value > 0, wakes up a blocked process waiting on the semaphore if any * sem points to the semaphore variable to increment the value |

Question 1.5 (1 mark)

The semaphore is not shared between the parent and child processes so the child process has another semaphore variable on its stack with an initial value of 0.

When the parent process increments its semaphore with sem\_post(&sem), the child process is unaware of the increment as this semaphore is not shared and the semaphore in the child process still has a value of 0 (initial value). Hence, the child process does not wake up and the program hung.

Question 1.6 (1 mark)

A semaphore pointer is created that points to an array of semaphores in the shared memory region. Each semaphore is initialized to a value of 0. The first semaphore is incremented so that the first child process can begin, while other child processes will be blocked and wait for their semaphore to be incremented. Just before the first child process finishes execution, it signals the semaphore of the next child process and this cycle repeats until the last child which do not need to signal any more semaphores.

**Part 2**

|  |
| --- |
| Question 2.1 (1 mark)  void init\_barrier(int *numproc*) {  nproc = *numproc*;  shmid\_count = shmget(IPC\_PRIVATE, sizeof(int), IPC\_CREAT | 0600);  count = (int \*) shmat(shmid\_count, NULL, 0);  count[0] = 0; // initialize count to 0  shmid\_sem = shmget(IPC\_PRIVATE, 2 \* sizeof(*sem\_t*), IPC\_CREAT | 0600);  sem = (*sem\_t* \*) shmat(shmid\_sem, NULL, 0);  sem\_init(&sem[0], 1, 1); // sem[0] = count\_mutex  sem\_init(&sem[1], 1, 0); // sem[1] = barrier  }   * nproc is assigned to the number of processes that will call the barrier. * shmid\_count is the allocated shared memory identifier from shmget for creating the shared variable count to count the number of processes that have reached the barrier, which is to be initialized to 0. * shmid\_sem is the allocated shared memory identifier from shmget for creating the two semaphores to protect count and barrier. * The first semaphore is for protecting count increment and is initialized to 1 so that one process can increment count at a time. * The second semaphore is for protecting barrier and is initialized to 0 so that all the child processes will wait until the last child process has arrived. |

|  |
| --- |
| Question 2.2 (1 mark)  void reach\_barrier() {  sem\_wait(&sem[0]);  count[0]++;  sem\_post(&sem[0]);  if (count[0] == nproc) {  // release one process  sem\_post(&sem[1]);  } else {  // we are not the last process,  // wait at the barrier semaphore until we are freed.  sem\_wait(&sem[1]);  // now that we are freed.  sem\_post(&sem[1]);  }  }   * When a process arrives, it will block if there is another process incrementing the shared variable count, else it will decrement the semaphore for count (sem[0]) and increment count variable. After incrementing the count variable, it will signal to allow another process to increment count, if any. * If the process is the last process, it will signal to barrier semaphore to release a process, which in turn releases another process and the cycle repeats until all processes have been released. |

**Part 3**

Question 3.1 (1 mark)

The parent process might execute and compare the values in the “largest” and “smallest” arrays before the child processes finishing computing the largest and smallest integer for each of their parts, resulting in the wrong results produced.

|  |
| --- |
| Question 3.2 (1 mark)  int pid, shmid\_largest, shmid\_smallest;  int \*largest;  int \*smallest;  shmid\_largest = shmget(IPC\_PRIVATE, NUM\_PROCESSES \* sizeof(int), IPC\_CREAT | 0600);  largest = (int \*) shmat(shmid\_largest, NULL, 0);  shmid\_smallest = shmget(IPC\_PRIVATE, NUM\_PROCESSES \* sizeof(int), IPC\_CREAT | 0600);  smallest = (int \*) shmat(shmid\_smallest, NULL, 0);   * Original “largest” and “smallest” arrays are local variables which means computation and modification of the array values by the child processes are not updated in the parent process. So these arrays have to be changed to shared variables for the parent process to be able to access the updated values. * “init\_barrier(NUM\_PROCESSES+1)” is called before the for loop to fork processes to set up the barrier * In the child process, “reach\_barrier()” is called after the child process has finish computing the largest and smallest values among their 250,000 integers to signal that their part of computation has been done. * In the parent process, “reach\_barrier()” is called before “clock()” to wait for all the child processes to complete their execution before the timer starts for the parent process to execute to find the smallest and largest values. |

Question 3.3 (1 mark)

The timer for parallel code only starts after the values of the “largest” and “smallest” arrays have been computed. As such, the time taken for the child processes for find the largest and smallest integers in each of their 250,000 integers are not taken into account. Hence, this way of measuring timing may not be fair.

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