CS69001 Computing Laboratory – I

Practice Exercises: Set 5

1. Let x be a shared integer variable. Write a program to create the variable and initialize it to zero. Write another program in which n processes $P_1, P_2, P_3, \ldots, P_n$ add random integers in the range $[1,100]$ to x. Here, P_1 is the process that you launch by running the executable of your code. This process reads n from the user. Subsequently, for $i = 1,2,3,\ldots,n-1$ (in that sequence), Process P_i forks Process P_{i+1}. *Before* the fork, each process adds its random contribution to x. After all of the n processes add their contributions, each process prints the final sum stored in x. Since the processes are not trying to write to x at the same time, their is no need to guard this access to ensure mutual exclusion. However, each process must print the final sum, and so must wait until all of the n processes have added their contributions to x. Achieve this synchronization using the **`wait()`** system call.

2. Repeat Exercise 1 in the case that 2^d processes are adding personal contributions to x. The processes should be created in the form of a d-dimensional hypercube (see Assignment C1), where d is supplied by the user to the initial process. You need to make two changes. First, multiple processes now may try to add to x simultaneously, so mutual exclusion is to be ensured for accessing x. Second, waiting only for the terminations of the child processes does not guarantee that every process sees the final sum. Use semaphores to ensure the correct working of your program under this changed scenario.

3. Write a program to add the elements of an array A of n integers as follows. The initial process reads n and p from the user, and creates and populates A in the shared memory by random integers in the range $[-100,100]$. It then creates p child processes. Each of the child processes computes the partial sum of a chunk of A of size n/p (assume that n is a multiple of p), and writes the partial sum into shared memory. When all of the p child processes finish, the initial process adds the p partial sums, and prints the final result. The initial process must know when all child processes have computed their respective sums. This synchronization is to be achieved as follows. Each child process, after writing its partial sum to the shared memory, sets a flag in the shared memory to indicate that it is done. The initial process keeps on checking the flags until all (p) of these have been set (*busy wait*). The partial sums and the end-of-work flags should reside in two shared arrays, each of size p.

4. Repeat Exercise 3 with the change that the synchronization is to be achieved by semaphores. The initial process, after populating A and creating the child processes, waits on a semaphore w. The child processes use a shared variable **`count`** created and initialized to p by the initial process. After each child process writes its partial sum to the shared memory, it decrements **`count`**. Since **`count`** is a shared variable, its access is to be guarded (mutual exclusion) by another semaphore s. The last child process to finish its task reduces **`count`** to zero. It then wakes up the initial process by signaling the semaphore w.

5. Write two programs **`listen.c`** and **`talk.c`** that work as follows. Let x be a shared integer variable. Each talk process keeps on writing random integers to x. After each write, the listen process reads and prints the value. Run one instance of the listen process, and multiple talk processes from different terminals. No two instances of the talk process should be able to write to x simultaneously. Moreover, once a value is written to x, it must first be read by the listen process before another talk process can overwrite this value. Implement the synchronization of the processes by semaphores. Allow each talk process to write 1000 times to x. After each write, a talk process sleeps for 1ms, whereas after each read, the listen process sleeps for 2ms.

6. Repeat Exercise 3 using pthreads. The master thread creates p worker threads, each of which computes the partial sum of n/p elements of A, and writes the partial sum and sets an end-of-work flag in shared memory locations reserved for that worker thread. The master thread waits busily until all of the p end-of-work flags are set. It then adds the p partial sums, and prints the final result.

7. Repeat Exercise 6 with the difference that the wait of the master thread will be not a busy wait but instead on a condition variable. The last worker thread to finish wakes the master thread up. Maintain a shared **`count`** of how many worker threads finish. Its access is to be guarded by a mutex.

8. Repeat Exercise 5 using pthreads.