Operating Systems Laboratory (CS39002)

Spring Semester 2021-2022

Assignment 3: Hands-on experience for using shared-memory

Assignment given on: February 08, 2022

Assignment deadline: February 14, 2022, 11:55 pm

In this assignment you will learn how to use shared memory to communicate between processes and in turn make your computation faster by dividing compuations. There are two tasks in this assignment.

Task 1: [15 marks]

a) For this assignment, you will be writing a program for matrix-multiplication. Let A and B be two input matrices of sizes $r_1 * c_1$ and $r_2 * c_2$ respectively, where $c_1 = r_2$. You will be implementing the naïve matrix multiplication algorithm, where each element of the result matrix C of size $r_1 * c_2$ can be computed independently from one another. The overall scheme for implementation is:

- Create a function void *mult(void *arg): which multiplies a row of matrix A with a column of matrix B, and stores the result in the appropriate cell of preallocated matrix C, **all in shared memory.**
- The argument arg is a structure of type ProcessData, which stores the input and output parameters for the function shown below. Note that the actual matrix data is stored in the shared memory and is accessible to all.
- In the main program, create child processes to compute all the elements of the matrix C.

The structure for passing parameters can be:

```
//    compute A[i,:] * B[:,j] and store in
C[i,j] typedef struct _process_data {

    double **A;
    double **B;
    double **C;
    int veclen, i, j;
} ProcessData;

//    veclen: c1/r2
//    i, j: cell to compute
```

In this part you should simply create $r_1 * c_2$ child processes each for computing each element. Note that no mutual exclusion is needed as all

the output entries are created independently and all the inputs are already available.

b) The problem with the above implementation is that you have to allocate memory for $r_1 * c_2$ processes and input arguments. However, in most practical systems you will not have enough cores to execute all the processes concurrently. Keeping this in mind, what is the maximum size of matrix that you can multiply(in terms of $r_1 * c_2$). Show your reasoning/calculation. Include this in a report (See **submission guidelines**).

Task 2: [35 marks](30+5)

Implement a producer / worker system using shared memory, which can multiply a sequence of matrices block-wise:

a) Your program will first read the values of NP (number of producers) and NW (number of workers), and create the required number of producer and worker processes. It will also take the total number of matrices to multiply as an input.

b) Create a shared memory segment SHM, which is shared among all the producer and consumer processes spawned by your code. The shared memory segment will contain a standard queue of finite size (say, can hold 8 elements). It will also create a *job_created* counter (integer) in the shared memory, which will store the number of matrices already generated.

c) Each producer process should generate a computing job, wait for a random interval of time between 0 and 3 seconds, and insert the computing job in the shared memory queue SHM. After insertion, the producer will repeat the process. Each computing job is represented by a structure which contains the following elements at a minimum:

   i) producer number

   ii) status of the matrix multiplication

   iii) a matrix of size NxN; N=1000

   iv) a matrix id

   The matrix id is a random integer between 1 and 100000, elements of the matrix should be randomly between -9 and +9. The producer generates each of these three numbers randomly while creating a job. The status can be set according to your needs.

d) While insertion, if the queue has no empty space, the producer waits until more space becomes available. It will also print the producer number, pid

and details of the job generated. Then the producer will increase the *jobs_created* counter by 1.

e) Each worker process waits for a random interval of time between 0 and 3 seconds, retrieves two blocks of the first two matrices in the queue, and updates the status as required. The block-wise multiplication is described below:

Let M = N/2. Break the matrices into M × M blocks. We want to multiply C = A * B

$$A = \begin{vmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{vmatrix}$$

$$B = \begin{vmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{vmatrix}$$

$$C = \begin{vmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{vmatrix}$$

For each I, J, K $\in \{0,1\}$, define the block product $D_{IJK} = A_{IK} * B_{KJ}$

For each I, J $\in \{0,1\}$, we then have $C_{IJ} = D_{IJ0} + D_{IJ1}$

Each worker process will calculate a D_{IJK} in its local memory, and then copy/add it to a new element in the SHM queue to store C_{IJ}. Thus to multiply two matrices you need 8 worker processes working on it.

When the first worker starts to access the blocks, add a new memory segment to store the resultant matrix at the back of the queue.
After the 8 workers have accessed all the blocks, remove the first two memory segments from the SHM queue.

If a worker finds that all 8 blocks of the first two memory segments have been accessed, it does nothing and waits for the next job.

The process which first accesses the block C_{IJ} copies its local block product to C_{IJ}. The second process to access C_{IJ} adds its local block product to C_{IJ}. The status variable can be used to map out which blocks of matrices have been processed already, and which are remaining.

Whenever a worker fetches or writes a block, print an appropriate status message stating: worker number, producer numbers, Matrix IDs, block numbers and the work done (reading/copying/adding the blocks).

f) The parent process (i.e., your code) will wait till *job_created* counter reaches a specified number of jobs (e.g., 10), and only 1 element remains in the queue. Output the total time taken to run those specified number of

jobs, along with the sum of the elements in the principal diagonal of the final matrix and then kill the process.

g) [BONUS]: use mutex locks to prevent concurrent updation to shared variables leading to possible race conditions. **[5 marks]**

Submission Guideline:

- Create two programs for the two parts of the assignment in C or CPP
- Include a short report as asked in Part 1, as a pdf file.
- Zip all the three files into Ass3_<groupno>_<roll no. 1>_< roll no. 2>.zip (replace <groupno> and <roll no.> by your group number and roll numbers), and upload it on Moodle.
- You must show the running version of the program(s) to your assigned TA during the lab hours.
- **[IMPORTANT]: Please make only one submission per group.**

Evaluation Guidelines:

Total marks for this assignment: 50.

We will check features of your code (including but not limited to) if the computation is correct, you are creating and using the shared memory correctly, correct number of producer/workers are created, if the jobs are correctly created, and the consumers are executing those jobs correctly (i.e., sleeping for the right time)