CSE 3100 Systems Programming	Fall 2021
Final exam	Due: 12/10/2021 at 11:59pm

Welcome to the final exam! Read the questions carefully before beginning work. **The format of the output must be adhered to strictly to pass the test cases.** A `Makefile` is provided. *Do not forget to submit your work before the deadline.* Please keep in mind that this is an exam and you are therefore expected to do this alone. Any academic misconduct will be pursued and punished with an 'F' in the course. Stay safe!

All the best!

1 (20 points) Shared virtual memory

Answer the following question: Suppose d is a real number and $0 \leq d \leq 1$, and x and y are two real numbers randomly picked in the interval $[0, 1)$. What is the probability that $|x - y| < d$?

One strategy is to perform many trials of the following experiment: randomly pick two numbers x and y in the interval $[0, 1)$, and determine if $|x - y| < d$. If this is true, then the trial is called an event. The probability of $|x - y| < d$ can be estimated by dividing the number of events by the number of trials.

You are required to complete the code provided in `rpair.c`. The `rpair.c` program is the parent process. It performs the above experiment using multiple worker (child) processes and virtual shared memory. The program takes three arguments from the command line: the value of d, the total number of trials s, and the number of worker processes nbW. Clearly, the parent must spread the work among the nbW child processes and aggregate the result to report the final answer. Specially, you are required to complete two functions: the first is main to setup the computation involving multiple processes and shared map. The second is `createSamples` function that is used by all the child processes to carry out their task. Note that the standard library function `rand_r` will be handy to generate a stream of random numbers between 0 to RAND_MAX (defined in stdlib.h). We provide the first line of the function to initialize the seed of that random stream. Check the man page of `rand_r` with: `man -S2 rand`.

Some sample sessions of running the program are shown below. As can be seen, the probability of occurrence of an event increases with the value of d. Also, as shown using the time command, the computation can be sped up by increasing number of processes from 1 to 30 processes for larger number of trials. Note that the exact values may vary for each machine.

```
kriti@VM$ ./rpair 0.1 3000000000 30
Total trials = 3000000000        Total events = 569997954        Probability = 0.189999

kriti@VM$ ./rpair 0.5 3000000000 30
Total trials = 3000000000        Total events = 2249994107       Probability = 0.749998

kriti@VM$ time ./rpair 0.9 3000000000 1
Total trials = 3000000000        Total events = 2969999148       Probability = 0.990000

real    0m23.862s
user    0m23.858s
sys     0m0.000s

kriti@VM$  time ./rpair 0.9 3000000000 30
Total trials = 3000000000        Total events = 2969998650       Probability = 0.990000

real    0m5.748s
user    0m45.843s
sys     0m0.010s
```

2 (40 points) Determinant of a matrix

The determinant of a matrix is a scalar value that is a function of the entries of a *square matrix*. It helps us find the inverse of a matrix, tells us things about the matrix that are useful in systems of linear equations, calculus and more. The determinant of a matrix A is denoted by det(A) or $|A|$.

In the case of a 0x0 matrix, the determinant is $|A| = \left| \right| = 1$

In the case of a 1x1 matrix, the determinant is $|A| = \left|a\right| = a$

In the case of a 2x2 matrix, the determinant is $|A| = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$

In the case of a 3x3 matrix, the determinant is $|A| = \begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = a \begin{vmatrix} e & f \\ h & i \end{vmatrix} - b \begin{vmatrix} d & f \\ g & i \end{vmatrix} + c \begin{vmatrix} d & e \\ g & h \end{vmatrix}$

That is, to work out the determinant of a 3×3 matrix:

- Multiply a by the determinant of the 2×2 matrix that does not include a's row or column.
- Likewise for b, and for c.
- Sum them up, but remember the minus in front of the b.

This procedure can be extended to give a recursive definition for the determinant of an n × n matrix, known as Laplace expansion, as given below.

$|A| = \begin{vmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ . & . & . & . \\ . & . & . & . \\ a_{n,1} & a_{n,2} & \dots & a_{n,n} \end{vmatrix} = \sum_{j=1}^{n} a_{1,j} * (-1)^{1+j} |A'|$

where $|A'| = \begin{vmatrix} \cancel{a_{1,1}} & \cancel{a_{1,2}} & \cancel{\dots} & \cancel{a_{1,j}} & \cancel{\dots} & \cancel{a_{1,n}} \\ a_{2,1} & a_{2,2} & \dots & a_{2,j} & \dots & a_{2,n} \\ . & . & . & . & . & . \\ . & . & . & . & . & . \\ a_{n,1} & a_{n,2} & \dots & a_{n,j} & \dots & a_{n,n} \end{vmatrix}$

(determinant of the matrix A excluding row 1 and col j)

2.1 (10 points) Sequential implementation

You are required to complete the code provided in `matrix.c` to calculate the determinant of a matrix sequentially (single thread). Specifically, you are required to complete the function detMatrix. The matrix data structure and APIs are provided in `matrix.h`. The main program is provided in `detSeq.c`. The program takes one argument on the command line: the file containing the number of rows, columns and a matrix. *Hint: you may create an auxiliary function to implement the recursive definition.*

2.2 (30 points) Parallel implementation using threads

You are required to complete the code provided in `matrix.c` to calculate the determinant of a matrix using multiple threads in parallel. Specifically, you are required to complete the function detMatrixPar. The matrix data structure and APIs are provided in `matrix.h`. The main program is provided in `detMT.c`. The program takes two arguments on the command line: the file containing the number of rows, columns and a matrix, and the number of threads (*nbW*) to be used. *Hint: notice that the summation contains n terms (as many as rows or columns). You may split the computation of n terms fairly among the nbW worker threads.*

Some sample sessions of running the programs are shown below. As can be seen, the output of sequential implementation matches the output of parallel implementation. Also, as shown using the time command, the computation can be sped up by increasing number of threads from 1 to 4 threads for larger matrices.

```
kriti@VM$ cat m0.txt
0 0
kriti@VM$ ./detSeq m0.txt
Determinant = 1
kriti@VM$ ./detMT m0.txt 4
Determinant = 1

kriti@VM$ cat m1.txt
1 1
9
kriti@VM$ ./detSeq m1.txt
Determinant = 9
kriti@VM$ ./detMT m1.txt 4
Determinant = 9

kriti@VM$ cat m2.txt
2 2
2 3
4 5
kriti@VM$ ./detSeq m2.txt
Determinant = -2
kriti@VM$ ./detMT m2.txt 4
Determinant = -2

kriti@VM$ cat A.txt
12 12
  3   1   2   0   3   0   1   2   4   1   2   2
  0   4   3   1   0   1   2   1   1   3   2   4
  2   0   2   3   2   0   4   2   2   3   4   2
  3   1   1   2   4   3   1   4   4   2   3   4
  0   0   3   1   1   0   1   3   2   0   1   1
  0   0   4   2   1   0   1   4   3   2   4   0
  2   0   4   2   4   4   3   0   2   3   1   3
  3   4   3   1   4   4   2   0   1   3   4   2
  1   1   4   4   0   0   4   3   2   1   2   2
  2   2   2   1   1   0   1   1   0   4   4   4
  0   4   1   2   2   1   1   0   4   2   2   1
  2   3   1   0   1   3   4   4   3   1   2   4
kriti@VM$ time ./detSeq A.txt
Determinant = -355536

real    0m6.274s
user    0m6.271s
sys     0m0.000s

kriti@VM$ time ./detMT A.txt 4
Determinant = -355536

real    0m1.890s
user    0m7.280s
sys     0m0.000s
```

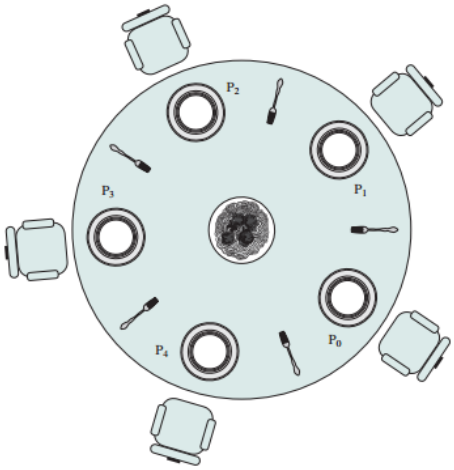

Figure 1: Dining Philosophers' problem with 5 philosophers

3 (40 points) Dining philosophers' problem

In this problem, n silent philosophers sit at a round table, each with a bowl of spaghetti, and n forks are placed between each pair of adjacent philosophers as shown in fig. 1. Each philosopher must alternately *think* and *eat*. However, a philosopher can only eat spaghetti when they have **both** left and right forks. Each fork can be held by only one philosopher at a time and so a philosopher can use the fork only if it is not being used by another philosopher. After an individual philosopher finishes eating, they need to put down both forks so that the forks become available to others. A philosopher can only take the fork on their right or the one on their left as they become available and they cannot start eating before getting both forks.

The problem is how to design a discipline of behavior (a concurrent algorithm) such that no philosopher will starve; i.e., each can continue to alternate between eating and thinking for c cycles.

One strategy would be to use mutexes and condition variables to synchronize the behaviour of philosophers at the table. In each cycle, a philosopher must wait for its immediate neighbours on the left and right to finish eating in order to pick up the left and right forks to eat. Upon eating, they must inform their immediate left and right neighbours of availability of left and right forks respectively. Multiple philosophers may eat concurrently if they aren't adjacent to each other and can pick up forks on their left and right.

You are required to complete the code in `philo.c` to implement the above strategy. The program takes two arguments on the command line: the number of philosophers n and the number of cycles c of *think* and *eat* that each philosopher performs. Specifically, you are required to implement the functions makeTable to initialize the shared buffer, clearTable to destroy the shared buffer, muse to perform think and eat cycles for each philosopher and main to setup multi-threaded computation of the dining philosopher's problem. You are provided with the ADT *PhiloTag* as shown below. The ADT consists of information for a philosopher including a unique ID for the philosopher (pid), current activity state (state), number of times eaten (ate), number of cycles left (cycle) and the shared ADT *Table*. Initially, the philosopher has c cycles left and is in the *THINK* state. You need to complete the shared ADT *Table*. *Hint: all philosophers must be able to access the table in a synchronized manner.*

```
typedef struct PhiloTag {
   int            pid;
   int            state;
   int            ate;
   int            cycle;
   Table*         t;
} Philosopher;
```

A sample session of running the program is shown below. As can be seen, each philosopher completes the prescribed number of cycles of thinking and eating (in this case 2), and hence, eats a total of two times. Each philosopher alternates between thinking and eating. The order in which the activities are performed by different philosophers may vary each time you execute depending on the sequence of thread execution.

```
kriti@VM$ ./philo 2 2
philo [1] is eating...
philo [1] is thinking...
philo [0] is eating...
philo [0] is thinking...
philo [1] is eating...
philo [1] is thinking...
philo [0] is eating...
philo [0] is thinking...
0 ate 2 times
1 ate 2 times
```