CSE 3100 Systems Programming	Spring 2024
Homework #7	

A Makefile is provided. Submit your work to gradescope.

We will be working on `mmul.c` in the first problem and `printing.c` in the second problem.

Problem 1. Matrix multiplication (50 points)

In this exercise we will continue to work on the matrix ADT.

An implementation of matrix ADT (abstract data type) is given in `matrix.c`. The API (application programming interface) functions that operates on the matrices are listed in `matrix.h`. One of the functions, `mulMatrix()`, performs matrix multiplication, which is implemented in `matrix.c`.

In this assignment, we implement `mulMatrix_thread()` in `mmul.c`, which has the same interface as `mulMatrix()`, but performs matrix multiplication with two threads. We only need to change `mmul.c`.

`test-mmul.c` is provided to test our implementation. The program takes the following arguments from the command line: the number of rows in the first matrix, the number of columns in the first matrix, and the number of columns in the second matrix. Then it fills two matrices with random numbers, and compares the result of `mulMatrix()` and `mulMatrix_thread()`. If no argument is specified, the program works on two matrices of size 6×6. In addition, if a command line option `-t<n>` is present, `test-mmul` prints the time (in seconds) spent on matrix multiplications, using the average of `<n>` calls.

Here are some sample sessions running `test-mmul`. The first command multiplies a matrix of 1000 by 500 with a matrix of 500 by 800. The resulting matrix is 1000 by 800. The second command also shows the timing information, the average of calling each multiplication function 3 times. `time1` is the average time on `mulMatrix()` and `time2` is the average time on `mulMatrix_thread()`. The numbers are likely to change in different runs.

```
$./test-mmul 1000 500 800
Good work!
$./test-mmul 100 500 300 -t3
Good work!
num_runs=3 time1=0.0652 time2=0.0341 speedup=1.9132
```

Problem 2. Printing (50 points)

Suppose p printers need to get j print jobs done. The print jobs are already placed in a queue. The starter code `printing.c` defines a type `job_queue_t` for the queue and provides functions to operate on the queue.

A printer performs the following operations in a loop.

1. Call `q_num_jobs()` to get the number of remaining jobs in the queue.
2. If no job is pending, exit from the loop.
3. Call `q_fetch_job()` to get a job from the queue. The function returns an integer indicating how long the job takes.
4. Use macro `print_job()` to print. The macro simulates the fact that different print jobs take different amounts of time to complete.

5. Keep track the number of jobs the printer has done.

The function `printer_single()` in the starter code `printing.c` shows how a single printer completes all the jobs.

The tasks in this problem are to use threads to simulate the process of multiple printers completing the print jobs. Each thread is a printer and performs similar operations as `printer_single()`. Apparently, threads need to coordinate their operations on the queue, which is shared by all printers. A mutex is defined in the `job_queue_t` structure for this purpose.

The program `printing` takes optional arguments from the command line. An argument can be one of the following.

- `-p <n>`. Specify the number of printers. The default value is 2.
- `-j <n>`. Specify the number of jobs. The default value is 20.
- `-d`. Call the demo function showing the operations of a single printer and exit.

Checking results. A script `check-printing.py` is provided to check the output of `printing`. Below is an example of how to use `check-printing.py`.

```
$./printing -p 5 -j 1000 | python3 ./check-printing.py
```

If you have made `check-printing.py` executable by command `"chmod +x ./check-printing.py"`, you can run it directly.

```
$./printing -p 5 -j 1000 | ./check-printing.py
```

Note that it is not guaranteed that a program that passes the check is correct. We should also examine the output manually sometimes. Some synchronization errors may manifest themselves only for some values of the parameters. And even for the same parameters, errors may happen non-deterministically due to different timing and scheduling orders of the threads. You may run the program multiple times even with the same parameters. For example, the following bash command runs the above example for 10 times. (Yes, it looks like a loop in C! and it may not work in other shells.)

Debugging. gdb supports multithreading. Run your code in gdb until it stops at a breakpoint or appears to stop making progress. If threads are not making progress, interrupt the execution with `Ctrl-C` to get to the gdb prompt. Here are some commonly used thread commands.

- `info threads` See what threads are running.
- `thread n` Switch to thread `n`, where `n` is a thread number.
- `thread apply [threadno] [all] args` Apply commands to one or more threads.