Question 1: (8 pts) Assume the following four code statements:

S1 : x = x * y;
S2 : y = x - y;
S3 : y = x + y - z;
S4 : z = x + y

Assume that x is initially set to 2, y is initially set to 4, and z is initially set to 6. Further assume that S1 through S4 are executed atomically. For each of the following Programs A-D, what are the possible final values of x and y and z for any program that runs to completion. Provide ALL possible final values. Be sure to explain your answer. If a program will not complete (stalls indefinitely), explain why.

A. S4; S2; S1; S3
B. co S2; // S1; // S3; S4 oc
C. co await (x > y) S1; S2; // S3; S4; oc
D. co await (x > y) S2; S1; // S4; S3; oc

Question 2: (12 pts). We’ve discussed in lecture different topologies, and how some are better suited than others for execution of concurrent programs. Three topologies are shown below, where N is network and P is processor. P includes registers, ALUs, AND cache. M is main access (off chip) memory.

For each of the topologies i-iii, enumerate and discuss briefly (a few sentences for each should suffice):

A. What is a disadvantage / possible shortcoming of a system with such a topology
B. What advantage(s) does the topology afford?

Keep in mind that one topology might have a shortcoming in terms of memory access, while another topology’s major shortcoming might be concerned with network latency. The Network is intended to be interpreted in the general sense; for example if i’s 2 Ps, N and M are all on chip, then consider the network...
as a bus, but if you consider that i’s 2 Ps are physically separated from the shared memory by hundreds of miles, then the Network should be thought of as a network with packet switching capabilities. Regardless of how you think of the Network it shouldn’t change your analysis much, because it is the number of entry points and exit points in the Network, and the number of dedicated paths and distances between different components that are the crucial factors. Answer the two questions for each topology in the general sense, but be specific. Don’t say, “Memory access is bad.” Instead, you could say, “Memory access is problematic because ______.”

For each topology i, ii, and iii, also provide

C. Pseudocode of a concurrent program that would be well suited for that topology, and explain WHY the code that you are providing is well suited for use on that topology. 5-10 lines of pseudocode for each answer should suffice. If A and B for each topology is well reasoned, then writing pseudocode should be straight-forward.

Question 3 : (3 points)

Optimizing code can be (is) a “creative” process, but at times it can be accomplished using established “algorithmic” techniques. For this question, provide code (pseudocode is fine) that is an optimized version of the below snippets of code, and explain WHY your optimized version is “optimized” versus the original one. Your optimized version need not be suitable for running concurrently. If the code cannot be optimized, explain why. Your explanation of why the optimized version is better than the original, does not need to be lengthy. A few sentences should suffice.

```cpp
int i = 0;
while( i != 100) {
    for(int j=2;j<i;j++) {
        if(i%j==0) break;
        if(j==i-1) cout << i << endl;
    }
    i++;
}
```

Question 4 : (3 points). Same instructions as Question 3.

```cpp
// already declared 2D arrays x and y
// already declared 1D array z
int i, j;
for (i=0; i<50; i++){
    for (j=0; j<50; j++){
        x[i,j] = y[j,i] + z[i];
    }
}
```
**Question 5**: (4 pts) In lecture you’ve seen Test and Set (TS), and Test, Test and Set (TSS). Assume that your computer has neither TS nor TSS. Instead, it has an atomic `Flip` instruction (shown below left), where `temp` refers to an internal register accessible to ONLY `Flip`:

<table>
<thead>
<tr>
<th>Atomic Flip instruction</th>
<th>Process (Worker) code</th>
</tr>
</thead>
</table>
| `Flip(var1, var2) : <  
  temp = var1;  
  var1 = var2;  
  var2 = temp; >` | `Process CS[i = 1 to n]{  
  while (true){  
    // critical section;  
    // noncritical section;}}` |

Use `Flip` to engineer a solution to the critical section problem for \( n \) processes. As we’ve done several times in lecture, modify the Process (Worker) code (shown above right) so that it uses `Flip`. If done correctly, only one of the \( n \) processes may be executing its critical section at any one time. You’ll need to decide on an entry protocol and/or an exit protocol. Your submission must include psuedocode that is a revised Process (Worker) code which relies on `Flip`. You CANNOT modify `Flip`. For each global variable and/or array that you declare and use, describe clearly its role. Also explain how your solution which uses `Flip` is correct. Clearly discuss how your solution ensures 1) mutual exclusion, 2) absence of deadlock, 3) absence of unnecessary delay, 4) and Eventual Entry for \( n \) processes.

**Hints**: You may pass as arguments to the `Flip` instruction individual indices of an array, and `Flip` will swap those values atomically.
Programming Task (18 points)

In lecture and in previous labs and homework we’ve discussed threads, as well as finding the maximum of an array. For this homework assignment you’ll implement a variant of that problem -- finding the maximum entry of a large 2-D array. Note that this task is different than what you did previously, because the threads MUST communicate with each other (return their “largest” found entry to the main method that invoked the threads, which then should compare each thread’s maximum entry found to determine the overall largest entry).

Threading

As you’ve now experienced, just because you create a thread, does not mean that the scheduler will run that thread as soon as that portion of your code where the thread is declared is executed. A scheduler on an $n$ processor computer might indeed even decide to run in series $n$ threads on a single of the $n$ processors if the load on all available processors is high. But there’s some hope. Use what you’ve learned to force the scheduler to run a thread as soon as it is created.

Coding Details -- Non-Threaded version

- Retrieve from the course website the `maxNonThreaded.cpp` file. That file contains a complete non-threaded program in which a 2D array of a large dimension is created and populated with random numbers. The main routine then inspects each entry of the 2D array in search for the largest entry.

- Compile `maxNonThreaded.cpp`.

- Run the program. The provided value $\text{dim}=50$ is used to create a small array. It should take only a second (or less) to run.

- Modify the code to include a wall time, and calculate the wall time for finding the largest entry (exclude the time needed to create the array).

- Run the program for $40,000 \times 40,000$, $20,000 \times 20,000$, $10,000 \times 10,000$, and $5,000 \times 5,000$ matrix sizes ($\text{dim}$). Run it twice for each size of matrix, and record the times as well as indices of the largest entry, as well as the largest entry. Consider modifying the program to accept command-line arguments for easy invocations.

Due to the inherent difficulty of generating a truly random number (using `rand()` draws from a small range), a very large 2D array (in your case $40,000 \times 40,000$) will have many duplicates, so the largest entry in one run of the program is often very similar to another run, or the largest entry may not be unique. Therefore, to confirm that the program correctly identifies the largest entry, your program MUST output the indices $i$ and $j$ of the/a largest entry as a simple check that the program works as intended. On subsequent runs, $i$ and $j$ should be different.
Coding Details -- Threaded version

- Create a new version of the program, `maxThreaded.cpp`, that employs threading. Be sure to launch each thread right away. How you break up the 2D array, and into how many threads, is up to you.

- Your goal is to utilize threading so that the run time of finding the largest entry is significantly reduced when compared to the non-threaded version of the program.

- As you did for the non-threaded program, run the threaded program twice for 40,000 x 40,000, 20,000 x 20,000, 10,000 x 10,000, and 5,000 x 5,000 matrix sizes. Record the times as well as indices of the largest entry, as well as the largest entry.

Run time Analysis

- Tally the run-times of your programs, and present in tabular form. For example, the following table has run-time values for the threaded and non-threaded versions (entries are for demonstration purposes; your tallies might differ significantly than those shown), includes the indices (x,y coordinates) for the/a largest entry, along with the largest entry. $entry_m$ designates maximum entry.

<table>
<thead>
<tr>
<th>Run Size</th>
<th>Threaded</th>
<th>Non-Threaded</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Run 1</td>
<td>Run 1</td>
</tr>
<tr>
<td>5000</td>
<td>time / indices / $entry_m$</td>
<td>time / indices / $entry_m$</td>
</tr>
<tr>
<td>10000</td>
<td>2.4 / (673,32) / 6732</td>
<td>2.2 / (872,781) / 3212</td>
</tr>
<tr>
<td>20000</td>
<td>4.9 / (9932,62) / 2233</td>
<td>5.4 / (256,8868) / 7782</td>
</tr>
<tr>
<td>40000</td>
<td>12.7 / (18982,3) / 7433</td>
<td>16.7 / (178,7652) / 782</td>
</tr>
</tbody>
</table>

- Generate a plot of your run-times to clearly demonstrate the utility of multithreading. Plot both Threaded and Non-Threaded data on the SAME plot. Be sure to label your axes, label your series, and provide a title.

Submission and Rubric. Submit via Canvas:

- A document (.doc, .docx, .pdf) with your answers to the “book” questions.
- A document (.doc, .docx, .pdf) with you programming task writeup (tabular results and plot). If you encountered unexpected run times, discuss possible reasons for what you observed
- Source code for your program `maxThreaded.cpp`
- A README file with compile instructions and/or comments.

<table>
<thead>
<tr>
<th>Question/Component</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free response questions (1-5)</td>
<td>30</td>
</tr>
<tr>
<td>Programming task – programs written, commented, and README included</td>
<td>3</td>
</tr>
<tr>
<td>Programming task – program compiles and runs as intended</td>
<td>10</td>
</tr>
<tr>
<td>Programming task writeup – table of run times</td>
<td>2</td>
</tr>
<tr>
<td>Programming task writeup – Plot of threaded and non-threaded run times</td>
<td>4</td>
</tr>
<tr>
<td>Total</td>
<td>49</td>
</tr>
</tbody>
</table>