Question 1: (15 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>
<tbody>
<tr>
<td>Flip(var1, var2) : &lt;</td>
<td>Process CS[i = 1 to n]{</td>
</tr>
<tr>
<td>temp = var1;</td>
<td>while (true){</td>
</tr>
</tbody>
</table>
|   var1 = var2;          |       // critical section;
|   var2 = temp; >        |       // 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: Refer to the lecture slides which explained how a coordinator process in combination with Workers, along with arrive and continue arrays, were used to solve the global variable contention problem when only a single flag variable was used. Consider using multiple arrays. You may pass as arguments to the Flip instruction individual indices of an array, and Flip will swap those values atomically.

Question 2: (15 pts). We’ve seen in lecture a tie breaker algorithm for ensuring that CS1 and CS2 processes have fair “turns” at accessing their critical sections. The approach shown in lecture included the Boolean variables in1 and in2, and the integer variable last. Unfortunately, that implementation was not suited for n>2 processes. There are many other tie-breaker algorithms available, one of which is shown on the right. The one shown is suitable for n processes.

```c
int in = 0, last[1:n];
Process CS[i = 1 to n]{
    int stage;
    while (true){
        < in = in + 1;>; stage = 1; last[stage] = i;
        <await (last[stage] != i or in <= stage);>
        while (last[stage] != i){
            stage = stage + 1;
            last[stage] = i;
            <await (last[stage] != i or in <= stage);>
        }
    // critical section;
    <in = in - 1;>
    // noncritical section;
}
```

Explain how the above program ensures mutual exclusion, avoids deadlock, and ensures eventual entry. Explain clearly the purpose of in and the array last, and explain clearly the entry and exit protocols.
Question 3: (15 pts) In lecture we’ve discussed semaphores and monitors. Although both are used to achieve the same objective (impose mutual exclusion and condition synchronization), monitors provide more structure than do semaphores and are often easier to use.

Write a monitor declaration (a class `semaphoreEquivalent`) that implements the same functionality as a semaphore. Recall that a semaphore has two methods, decrement and increment, whose roles are to self-stall a calling process and to release (if there exists one) a stalling process, respectively. Thus at a minimum your monitor class should have these equivalent capabilities. Use the following skeleton of code for `semaphoreEquivalent`:

```java
monitor semaphoreEquivalent {
}
```

Declare specific variables and methods, arrays, etc. as needed. Although you do not have to adhere to any specific language and its syntax, you must write more than pseudocode. Note that when a thread increments or decrements a semaphore, it is not aware of any other threads, and whether they are stalled or not. Monitors however are able to reason about other threads/processes that have made calls to a monitor, so your `semaphoreEquivalent` class should provide that feature (see lecture slides).

Question 4: (15 pts). We’ve discussed the butterfly and dissemination barriers. Your task is to assess which of the two is better suited for a problem with 8 workers that must be synchronized. Each worker performs 2 calculations, but all 8 workers cannot proceed to their second calculation until all 8 workers have completed their first round. Hence a “single” barrier must be imposed. For both the butterfly and dissemination barriers, and for this specific example (8 workers):

- **Butterfly**: specify all of the variables that are needed (to implement the barrier)
- **Butterfly**: specify that code that each process (worker) must execute to implement the barrier
- **Dissemination**: specify all of the variables that are needed (to implement the barrier)
- **Dissemination**: specify that code that each process (worker) must execute to implement the barrier

For the following units of time for assignment and `await`, specify the total time needed for barrier synchronization for both butterfly and dissemination barriers

- **Assignment**: 1 unit of time; **await**: 1 unit of time
- **Assignment**: 3 units of time; **await**: 1 unit of time
- **Assignment**: 1 unit of time; **await**: 3 units of time

Submission. Submit a single file to Canvas that includes your answers to the two questions.

Rubric

<table>
<thead>
<tr>
<th>Question/Component</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qs 1-4</td>
<td>15 points each</td>
</tr>
<tr>
<td>Total</td>
<td>60 points</td>
</tr>
</tbody>
</table>