Question 2. Synchronization [10 marks]

Mark has created a simple function called add that adds 1 to each element of a global array called Array. All the elements of Array are initialized to 0. Assume size is the size of the array (a constant). Also, assume that the only way that Array is updated is using the add function.

An invariant (constraint) that should hold when the add function is not executing is that all elements in the array have the same value. Mark is a meticulous programmer and uses the assert function in the add function to check the invariant, i.e., that all the elements of the array have the same value:

```c
int Array[size];

add() {
  int i;
  for (int i = 0; i < size; i++) {
    if (i < size - 1) { // no check needed for last element
      // check that current element is the same as the next element
      assert(Array[i] == Array[i + 1]);
    }
    // then update current element
    Array[i] += 1;
  }
}
```

Part (a) [2 marks] When Mark runs the add function within a single thread, it works as expected. Mark has recently learned how threads can be used to speed up programs and decides to create two threads, each of which run the add function. Mark finds that his multi-threaded program crashes randomly. Provide a scenario that could lead to a crash.

There are many possible scenarios. The reason for the crash is that the assert will fail since different invocations of the add function can update different elements in any order.

2: The scenario is described clearly and leads to an assertion failure.
1: The scenario is described in such a way that it is unclear whether an assertion failure will occur.
0: No answer or the description will definitely not lead to assertion failure.
Part (b) [2 MARKS] Mark has learned that locks can fix random crashes and decides to fix the problem by using locks as follows:

```c
int Array[size];
lock 1;

add() {
    int i;
    lock(1);
    for (int i = 0; i < size; i++) {
        if (i < size - 1) { // no check needed for last element
            // check that current element is the same as the next element
            assert(Array[i] == Array[i + 1]);
        }
        // then update current element
        Array[i] += 1;
    }
    unlock(1);
}
```

Will his solution fix the random crashes? If so, briefly explain why. If not, provide a scenario that could lead to a crash.

This code will fix the random crashes. The locking ensures that each invocation of add will run to completion before another add runs, ensuring that the assertion will remain true.

| 2: Answer is clearly explained. |
| 1: Answer says random crashes are fixed, but the explanation is not clear. |
| 0: No answer, or answer says that random crashes are still possible. |
Part (C) [3 Marks] Mark has recently learned that using fine-grained locks can provide better parallelism. He decides to create a lock for each element of the array and lock this element when updating it, as shown below.

```c
int Array[size];
lock l[size];

add() {
    int i;
    for (int i = 0; i < size; i++) {
        // Lock current element
        lock(l[i]);
        if (i < size - 1) { // no check needed for last element
            // check that current element is the same as the next element
            assert(Array[i] == Array[i + 1]);
        }
        // then update current element
        Array[i] += 1;
        // unlock current element
        unlock(l[i]);
    }
}
```

Does this fine-grained locking solution work? If so, briefly explain why. If not, provide a scenario that could lead to unexpected behavior.

This code will not work and cause crashes again. The problem is that while each element of the array is updated atomically, the assertion is checking the current and the next value. This assertion may not hold during concurrent add operations. For example, say Thread 1 updates Array[0]. Before it updates Array[1], say Thread 2 starts running. When it checks Array[0] and Array[1], they will not be the same, thus the assertion will fail.

3: The scenario is described clearly and leads to an assertion failure.
1: The scenario is described in such a way that it is unclear whether an assertion failure will occur.
0: No answer or the description will definitely not lead to assertion failure.
Part (D) [3 marks] As Mark learns more about fine-grained locking, he decides to use the strange locking scheme shown below.

```c
int Array[size];
lock l[size];

add() {
    int i;
    // lock first element
    lock(l[0]);
    for (int i = 0; i < size; i++) {
        if (i < size - 1) { // no check needed for last element
            // lock next element
            lock(l[i+1]);
            // check that current element is the same as the next element
            assert(Array[i] == Array[i + 1]);
        }
        // then update current element
        Array[i] += 1;
        // unlock current element
        unlock(l[i]);
    }
}
```

Does this fine-grained locking solution work? If so, briefly explain why. If not, provide a scenario that could lead to unexpected behavior.

This code works correctly. The reason is that it solves the problem in the previous version. In this case, the code locks the current and the next element before checking the assertion, which guarantees that these two values will be the same. Then the code unlocks the current element, allowing any concurrent later `add()` to proceed to update earlier elements of the array. In effect, the threads are pipelining updates to the array.

It may appear that the locking is asymmetric (more lock calls than unlock calls) but that is not the case.

Iteration i = 0: lock(0), lock(1), unlock(0)
Iteration i = 1: lock(2), unlock(1)
...
Iteration i = size - 2: lock(size - 1), unlock(size - 2)
Iteration i = size - 1: unlock(size - 1) (last iteration doesn't perform a lock)

---

3: Answer is clearly explained.
2: Answer says that solution will work, but the explanation is not clear.
1: Answer says solution will not work, and the explanation makes some sense.
0: No answer, or answer says that solution will not work and explanation is meaningless.