Brian Rodriguez
U0853593

Quiz 2

Memory Models, and the Java Memory Model:

1. The memory model specifies special synchronization barriers for threads when dealing with shared variables. The threads only need to be aware of the changes to the shared variables, when a synchronization barrier is reached. These rules give compilers freedom to optimize the code and also stay true to the unoptimzed code, when the compiler reaches a critical section.

2. A platform model is essentially an uncrutched way of defining when critical sections that are being accessed. In the JVM, it allows for Java developers to not worry about these critical sections with the use of barriers, such as locks. Instead the JVM only demands for the use of proper synchronization, where as the platform model demands for the use of special barriers and fences.

3. **Execution scenario**: In the execution scenario, both threads are seemingly ran at the same exact time. With reordering, the JMM can make both threads run in different orders, since both threads do not have a data dependency with each other they can produce abstract results.

   Thread A is executed in the opposite order and Thread B has no idea about Thread A being executed in the opposite order, thus producing these strange results.
   Both threads are able to see (0,0) returned because of how the reordering is executed. In either case threads A and B can be reordered to see the results (0,0). The developer has no control over the type of optimizations that the JVM/JMM can produce, unless other wised specified by a synchronization flag.

   In a sequentially consistent memory model, there is a contract between the programmer and the system. If the programmer follows the contract, then the system guarantees that the output of the program will be predictable. In the case of the PossibleOrdering program, if the sequential memory rules were followed then the system will guarantee that the program output will be as expected.

4. *Happens before* is a partial ordering rule that is defined by the JMM over all actions of the program being executed. *Happens before* is essentially a global order of operations table for all threads, situation A must happen before situation B.

   Happens before is essentially an all encompassing rule that ensures that a program's execution happens in the way that it was intended to when dealing with multiple threads.

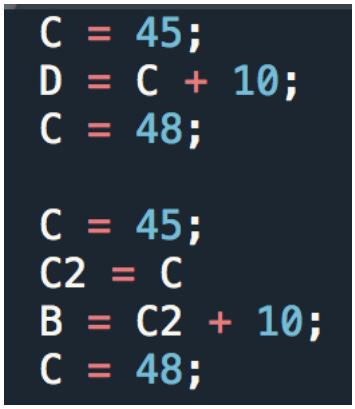
   If one thread acquires a mutex, then the other thread cannot execute the operations that it is presented with until, the mutex is released by the first thread. Once the first thread has released the mutex, then all the changes made to the variables operated on by thread one are then made globally visible to all the other threads working.

Data Dependency Analysis:

1. **True dependency**: A true dependency or a flow dependency is a type of dependency that is a read-after-write dependency. This occurs when an instruction depends on the output of a previously executed result.
    a. **1**. A = 50;
    b. **2**. Z = A;
    c. **3**. C = Z – 20;

    Anti-dependency: Anti-dependency is a type of write-after-read dependency, this type of dependency happens when a line of instruction requires a value that is later changed/updated.
    d. **1.** C = 45;
    e. **2.** D = C + 10;
    f. **3.** C = 48;

    Output dependency: Output dependency is a type of write-after-write dependency. Output dependencies occur when the output is dependent on the ordering of the program's execution.
    g. **1.** E = 30;
    h. **2.** F = E + 20;
    i. **3.** E = 2;
2. Output and anti dependent programs can be easily removed with simple naming conventions. Meaning that the dependencies of these programs can be removed by changing variable names. This happens because the compiler may not be able to tell whether all of the different variables have a common middle-ground.



OpenMP, Amdahl's Law:

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

1. If a CPU spends 20% of its time doing serial work, meaning that if a program ran for an hour then it would spend 12 minutes doing serial work. This leaves us with 48 minutes to parallelize the rest of the program. If we are subject to 80% of parallel work then we can use p = .8, where p is equal to the portion of the program that can be parallelized. Using this formula, we can get that the highest amount of speed-up is roughly ~ 5.0 speed-up or an 80% speed-up.

2. Pre-transformation, the author illustrates that A(i) is dependent on itself. Meaning that if another thread attempts to read while another thread is writing to A(i), then the output will almost certainly be different every time the program is ran.

The author suggests to rename the variables and use an OpenMP to do some parallelization under the hood. A(i) is now stored in another variable T(i) then T(i) is stored in A(i), this allows for the elimination of a data dependency that may cause the read after write issue posed in the previous pre-transformation.