Tyler Thomas

Prof. Goldberg

CISC 3320

The project in question is a Java class named BasicDeckSim.java that creates a Data Structure representation of a deck of cards. This class has two core functions:

- A shuffle function that randomizes the position of each “card” in the deck
- A draw function that simulates the opening setup of a deck by taking the first 20 cards of the deck and evenly distributing them among 4 “hand” arrays.

For the sake of completion, the class also has the following functions:

- A reset function that restores the class to its state immediately after it is initialized via constructor.
- An overridden toString function that returns a string representation of the deck in its current state.
- A printAll function that prints the deck and the 4 “hands” in their current state.

The shuffle and draw functions both use threading in order to perform their operations. Java’s version of threading requires you to use the ForkJoinTask library and create a subclass of the RecursiveAction class whenever you define a child process. For this reason, the drawchild and shufflechild functions have been created in order to allow the core functions to fork their own children.

The extra code required to implement a multithreaded implementation of these functions is among the smaller problems with this approach. Both of these functions are implemented using data parallelism, where each child of the process is operating on the same deck at the same time.

In both cases, doing this requires reckoning with the risk of race conditions, and ensuring that two or more children aren't operating on the same point at the same time. The steps that the shuffle function takes to perform its operation are as follows:

1. Take in your arguments and establish which section of the deck you are operating with.
2. Generate a random number within the domain of position numbers in the deck. For example, the deck used in the main method has a domain of all positive integers between 0 and 51.
3. Check the index called by the random number generator to see if an item has already been assigned to it.
4. If there's no item assigned to that index, indicate that the random number generated in step 2 is being assigned. Otherwise, loop back to step 2.
5. Assign an item in the deck to the index generated in step 2.
6. Repeat Steps 2 through 5 for each item in your section of the deck.

In a single-threaded solution, steps 4 and 5 are interchangeable. However, with multithreading, it is important to make sure that two threads don't try to put an item at the same position in the deck. This is possible if two or more threads manage to simultaneously come to the conclusion that any one index is unoccupied. Flagging the index before occupying it is a step in the right direction, because it increases the chance that one thread is successfully preempted by another. However, there is still a potential issue; if two threads *check* an index before either can flag it to begin with, there will still be an error. Solving the problem requires reducing the time between the execution of steps 3 and 4. Luckily, Java's Set.add() function actually does two things; not only does it add the argument passed to it, but it also returns a boolean indicating the

success of the operation. This knowledge allows us to compress the execution of steps 3 and 4 to a single function call, reducing the chances of conflict to a negligible amount.

Avoiding race conditions for the draw function is more difficult. The process that each thread needs to follow is as follows:

1. Keep a record of the position of the deck that you're looking at.
2. Check to make sure that the card in that position has not been drawn yet. The card has been drawn if the string in that position is equal to the empty string.
3. If the card in that position has been drawn, look at the next spot in the deck and loop back to step 1.
4. Make a note of the identity of the card in that position, and then change that card to the empty string.
5. Add the card to the "hand" that the thread is operating on.
6. Repeat steps 1-5 until the thread has drawn 5 cards.

Just like in the shuffle function, steps 4 and 5 would be interchangeable in a single-threaded solution, but need to specifically be in that order in a multithreaded solution. This is because, just like in the shuffle function, there is a chance that two threads get to the same undrawn card and check for its existence before either thread officially "draws" the card. This is exacerbated by the fact that all four threads start at the same index, and because of that, it's significantly more likely that two or more threads check the same item at the same time.

There are solutions to this problem that are much simpler than the solution I ended up settling on. You can resolve the problem by having each thread start at different positions in the list and simply have each thread operate on different subsections of the deck. You can either have

the starting indexes spread 5 spaces apart and increment the index variable by 1 with each iteration, or have the starting indexes 1 space apart and increment the index variable by 5 with each iteration. I didn't implement either of these solutions because the resulting hands would be predictable, and that isn't the result I was personally going for.

The solution is to create an array of flags, with each flag representing a corresponding position in the deck. When a thread begins its loop, it will check an index by first looking at its flag. If that index has been or is being operated on, its flag will be false and the thread will check the next index until it finds an available index. Once it finds one, it will set its flag to false and begin its operations. Unfortunately, this is still not enough to guarantee that two or more threads don't flag the same index at the same time. We need to add extra redundancy to our solution through strategic use of random wait times, followed by a secondary check for the use of the resource. If two threads that flag the same index are forced to wait at random times, the thread that spends less time waiting can proceed as normal while the other can know not to. There are pros and cons to different wait times. A longer wait time will extend the execution time of the program, but it'll reduce the chance of an error. Shorter wait times will have the opposite effect.

The multi-threaded solution has benefits and drawbacks for both functions. For the shuffle function, the single-threaded solution has less space complexity, as the multi-threaded solution in Java requires the creation of one extra copy of the deck. The time complexity is also greater in the multi-threaded solution for the exact same reason, but this is deceptive. While the multi-threaded solution needs to spend time making an extra copy of the deck, the amount of actions needed to reassign each item is identical. This is important to note, because the multi-threaded solution will perform these actions in, at best, ¼ of the time the single-threaded solution will take. The efficiency of this solution depends on how many concurrent threads your

computer can handle. If your computer can only run one or two threads at once, you will prefer a single-threaded solution, but if it can handle more, then you may need to weigh factors such as available memory in order to determine whether you place more value on time or space complexity.

The solution for the draw function is more difficult to analyze. Recall that the solution presented is more complex than necessary in order to get a less neatly organized set of hands. That extra randomness forces us to spend more time worrying about races, and the solution will require extra time and space. It is worth noting that a single-threaded solution aiming to achieve the same randomness will have to try to achieve it through less organic means, which will require complex algorithms and a use of one or more data structures. A single-threaded solution would need to incorporate a method similar to that of the shuffle function, and the HashSet required for that will occupy the same amount of space as the array of flags in the multi-threaded solution. The need to modify and search that HashSet will also increase the time complexity of such a solution. The multi-threaded solution will have comparable time complexity split between 2 or more concurrent threads, but the addition of random waiting makes it much harder to predict exactly how long the solution will take. The actual runtime of this solution when compared to the single-threaded solution depends on a multitude of factors, including the level of parallelism of the process and the domain of wait times possible.

As a final note regarding the draw function, the multi-threaded solution would always be better if we were aiming for a more structured result(i.e [1,2,3,4,5]). The removal of arbitrary factors eliminates the chance for race conditions, meaning that the multi-threaded solution will use the exact same memory and perform the exact same actions as the single-threaded solution, but in a fraction of the time depending on the level of parallelism of your process.