Scalable Lock Free Stack - Sequential Implementation

William Pearigen
COP 4520
Programming Assignment 2
February 15, 2016

***Abstract*—In this paper, I will discuss the use of a stack, ways the stack can be represented as a data structure (specifically how the original research paper uses them), and how to operate on the stack in a multi-threaded approach. I will also discuss the efficiency and execution time of a sequential approach to multi-threading with this stack.**

I. Classes

```
/**
* Cells make up the stack, with the "next"
* cell being the one under the current cell,
* so the bottom cell in the stack will
* have a null next cell
*/
public static class Cell{
  Cell next;
  char data;

  public Cell(Cell next, char data){
    this.next = next;
    this.data = data;
  }
}

/**
* Shared stack object. Very simple,
* consists only of the top cell,
* which points down the stack,
* similar to a linked list.
*/
public static class SimpleStack{
 Cell top;

 public SimpleStack(Cell top){
  this.top = top;
 }
}

/** This class is not used in the sequential
implementation, and will be set to null for
    now **/
public static class AdaptParams{
 int count;
 double factor;

 public AdaptParams(int count, float factor){
 this.count = count;
 this.factor = factor;
 }
}
```

```
/**
 * This class extends Thread class in order
     to add an id,
 * stack operation, current cell to use and
     adapt parameters
 * (not used in sequential implementation).
 * The run method will create an array of
 * instructions for each thread, then go
     through
 * each instruction on the stack until the
 * entire array has been performed
 */
public static class ThreadInfo extends Thread{
 int id;
 char op;
 Cell cell;
 AdaptParams adaptParams;

 public ThreadInfo(int id, char op, Cell
     cell, AdaptParams adaptParams){
  this.id = id;
  this.cell = cell == null ? new Cell(null,
      'T') : cell;
  this.op = op;
  this.adaptParams = adaptParams;
  }

 public void run(){
  List<Character> instructions =
      initInstructions();
  long startTime = System.currentTimeMillis();
  for(char instruction : instructions){
   this.op = instruction;
   stackOp(this);
  }
  System.out.println("Finished thread " + id +
      " in " + ((System.currentTimeMillis() -
      startTime) / 1000.0) + " seconds");
}
}
```

II. Introduction

A. Class Structures

First, the cell class is very similar to an index in a linked list. Each cell has some data, in this case only a character, and a pointer to the next cell in the list, or in this case, a stack. The cell does not have an index so you do not know where you are in the stack by the cell unless you keep a counter variable, but this information is not needed since all

operations will be performed on the top cell only, so the top cell is the only one that needs to be kept track of.

Next, the SimpleStack object is what is used to represent the stack. There will only be one SimpleStack instantiated and it will be shared between all the threads. The SimpleStack object only contains a pointer to the top cell of the stack, and this value can be null if the stack is empty. Since each Cell keeps track of where the next Cell down in the list is, we don't need to keep track of the index. In this implementation, the threads will perform operations to the stack sequentially, using a synchronized block.

Finally, the ThreadInfo class is the one that does all the work. Each ThreadInfo extends the Java Thread class, so they are used very similarly. Some of the additional attributes to a ThreadInfo includes an ID, a Cell which will be pushed onto the stack in the case of a 'push' operation, an operation to be performed on the stack in the form of a character, and a set of AdaptParameters. The ThreadInfo class has an override run method that performs all the operations and will be discussed later.

B. Background

1) Original Stack: The original, single threaded, most basic implementation of a stack works as such: figure out what operation is next in line to be performed (PUSH or POP), and then go to the top cell of the stack. If the operation is a POP, we will reset the top cell of the stack to be the cell the current top cell is pointing to as it's 'next' cell. Then the old top cell should be cleaned up. If the top cell is null when first checked, then no operation is performed because the stack is empty and remains unchanged. If the operation is a PUSH, then a new Cell is created and the new Cell's 'next' attribute is set to point at the stack's old top Cell. Then the stack's top Cell is reset to the new cell that was just created. These operations are performed one after another until there is no more operations to be performed or the program is terminated.

2) Scalable Lock Free Stack: The original research paper "'A Scalable Lock-free Stack Algorithm"' discusses a away to approach a stack similar to before but with multiple threads. The threads would not wait for one another to finish before operating on the stack, rather they will use a technique to run concurrently, using a 'back-off elimination' array. By using this array, multiple threads can attempt to perform an operation on the stack and adjust how they work based on each circumstance. This is not the technique used in this implementation, but it is worth mentioning for future use.

3) Sequential Implementation of SimpleStack: In this paper, we will focus on how a sequential version of a stack is implemented with multiple threads. This is done using simple Java Thread library methods. Since the ThreadInfo class extends the Thread class, they will be used virtually the same way. We will spawn multiple threads, each of which will perform 500,000 stack operations, distributed in different ways, on the same stack object. Since the operations take place inside a synchronized block, this makes them sequential, or working one after another when their turn comes around. This is not as efficient as the original paper, but is completely Thread safe. These findings will be discussed more.

III. Sequential Implementation

In this implementation, I will use the same data structures presented in "'A Scalable Lock-free Stack Algorithm"' but instead of using the elimination back off array and thread collision techniques, a thread safe sequential version has been employed. This is done using an extension of Java Threads and a synchronized block. The process can spawn as many threads as the user desires, each of them performing 500,000 operations on the same `SimpleStack S` object sequentially. This can be seen by the code in the override run method in the ThreadInfo class declaration:

```
public void run(){
    List<Character> instructions =
        initInstructions();
    long startTime =
        System.currentTimeMillis();
    for(char instruction : instructions){
     this.op = instruction;
     stackOp(this);
    }
```

Each thread will start by first initializing a list of character instructions, either '+' or '-' for PUSH or POP, respectively. The ratio of the list of operations can be adjusted since it is generated with a random double generator. by default, it is 50% push and 50% pop. These distributions are approximate of course, due to the nature of the random number generator. Then a time stamp is taken right as the thread begins traversing through it's unique list of operations. This time stamp will be used once the thread is finished in order to see how long it took to complete. for each instruction, the ThreadInfo object will set the current operation in the list as its 'op' attribute and then it calls the `stackOp(this)` method, using itself as the parameter to pass in. This method is where the `SimpleStack S` is operated on in a thread safe manor and works as such:

```
public static void stackOp(ThreadInfo p){
   Cell newCell;
   synchronized(lock){
    if(p.op == '+'){
     newCell = p.cell;
     newCell.next = S.top;
     S.top = newCell;
     p.cell = new Cell(null, ++currentData);
    } else{
     if (S.top != null) {
      S.top = S.top.next;
     }
    }
   }
  }
```

The thread bedin by declaring a new Cell object in case it needs to add it to the thread if the current one is pushed onto the stack. Then it enters the `synchronized(lock)` block, which is what makes this implementation sequential. When a thread hits this line of code, if the lock object is not available, it must mean another thread is accessing the stack and performing an operation on it currently, so the current thread will wait there until the other thread is finished and returns the lock to an available state. Once it becomes available, the first thread in line will acquire it and perform its operation on the stack. The PUSH operation happens by setting `newCell` to the current ThreadInfo's cell. The `newCell`'s next attribute is then set to the stacks current top cell. After that happens, the stack's top cell can be reset to `newCell` and the ThreadInfo can create a new `Cell` for itself for future use. In the case of a POP operation, all that needs to be done is the top of the stack should be reset to the top's next cell, and the old top of the stack should be cleaned up.

1) Deadlock and Starvation Freedom: One of the biggest concerns when it comes to milti-threaded approaches is *Deadlocking*. As stated by [2, wiki/Deadlock], Deadlocking can be described as such: "In concurrent programming, a deadlock is a situation in which two or more competing actions are each waiting for the other to finish, and thus neither ever does." For example, if a Thread was to obtain a lock for a synchronized block of code, but within that code block tries to access an object that is protected by a lock, then there is a chance another thread will already have that lock but is also waiting to enter the first synchronized block as well. This means the first thread is waiting for the second one to release the object's lock and the second thread is waiting for the first thread to exit the synchronized block. It can be seen, in the circumstance there is no way for either thread to progress at this point.

Within this implementation of a stack, it can be seen that this solution is deadlock free. By using the synchronized block, the stack is only being accessed by one thread at a time, and within that synchronized block, there are no calls to any other methods or objects protected by mutex locks. Since this is the case, it can be seen that there will never be a case where two different threads will be waiting on each other to complete in order to progress.

Another huge concern of multi-threading is *Starvation*. From the Oracle website, [3, concurrency/starvelive], "Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads. For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked." In the case of this implementation, the synchronized block does not consist of any loops or method calls that could potentially get hung up for long periods of time. It simply performs its single operation, then releases the lock for the next thread. Also, each thread has equal priority, so no thread will be stuck at the back of a queue, unable to perform its operation for a long time. These facts prove this implementation is starvation free.

A. Execution Time and Efficiency

In this section I will discuss different test modules used and the resulting execution times, and explain what the results mean in terms of efficiency. Due to the nature of this implementation, different parameters, such as the number of threads used, and the distribution of the operations, will affect the execution time of each thread and the program as a whole. This is proven in two test modules with different operation distributions, each of them running with 1, 2, 4, and 8 threads.

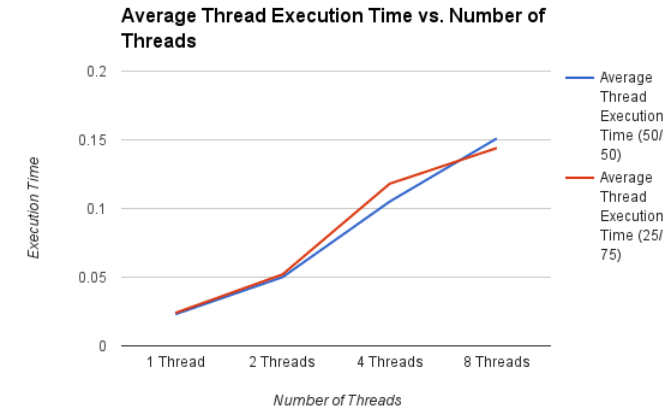
For the first test module, when each thread initializes its unique stack, it used a random number generator to generate a double between 0.0 and 1.0. If the number is less than 0.5, then a PUSH will be used, otherwise, a POP will be used, as can be seen in the code below:

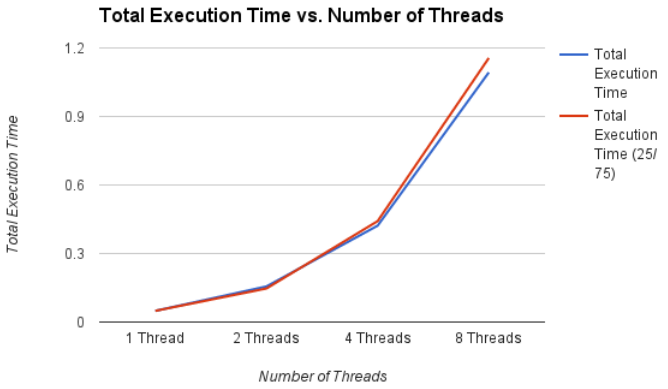
```
private static List<Character>
    initInstructions(){
   List<Character> instructions = new
       ArrayList<Character>();
   for(int i = 0; i < NUM_OPERATIONS; i++){
     if(random.nextDouble() < 0.50){
       instructions.add('+');
     }else {
       instructions.add('-');
     }
   }
   return instructions;
 }
```

Using this convention, the distribution will not be exactly equal, but between all the threads they will be approximately the same. This holds true for all the threads beings used, even though each set of operations is spawned independently. This test was performed using 1, 2, 4, and 8 threads as it is shown that as the number of threads increases, the average execution time for each thread increases, along with the total execution time. The total execution time is expected to increase because with each thread being created, another 500,000 operations will be performed on the stack. more operations should mean more time. The reason each thread's execution time increases is due to the synchronized block. When there is only one thread running, it will never be in contention of the lock with another thread, so it will always be able to acquire it and move on. This results in the thread performing its 500,000 operations very quickly. When another thread is added to the process, there will be an occasional conflicts over the lock, but not every time one of the threads needs the lock. And when one thread needs to wait for the lock, it is guaranteed that it will be a very short amount of time until the lock is released. Once more threads are added, the rate of conflict is increased, and threads will

start to have to wait longer and more often for the lock to operate on the stack. This guarantees that no two threads will try to perform an operation on the stack at the same time, returning inaccurate results. The trade off is each thread will be waiting longer, thus not doing anything, for more time.

The second Test module is the same as the first except for the distribution of the list of operations. In this case the `initInstructions()` method will produce lists that are roughly 25%POP/75%PUSH so the stack will become much larger. it is shown in the data that even though the distribution changes and the size of the stack is increased so much, there is negligible difference between the execution times. This is mainly due to how the `SimpleStack S` object is constructed. No matter how large the stack gets, the object is still only a pointer to a single cell, the top of the stack, so the actual object stays the same says and the access time remains constant once the thread has been acquired. All of these results can be seen in the graphs below.





IV. Conclusion

To summarize the results, in a sequential implementation of the `SimpleStack`, the execution time is directly related to the number of threads. There is very little change when the distribution of the operations is adjusted, namely because of the nature of the stack itself, only pointing to one cell at a time. So the more threads you add, no matter how the operations are distributed, the longer the total execution time will take, along with the average execution time of each thread. This is due to more threads competing for the same lock causing them to wait more often and for longer based on how many threads are running. Each thread will perform the same amount of operations, but the context switches for the lock take a lot of clock cycles, making this approach less than or equally as efficient as a single threaded, non synchronized approach.

References

[1] D. Hendler and N. Shavit and L. Yerushalmi, *A Scalable Lock-free Stack Algorithm*, 3rd ed. 16th Annual ACM Symposium on Parallelism in Algorithms, Barcelona, Spain, 2004, pages 206-215.

[2] https://en.wikipedia.org/wiki/Deadlock

[3] https://docs.oracle.com/javase/tutorial/essential/concurrency/starvelive.html