Slide 1: What is the project?

For my project I created a music recommendation system. This system takes a song and artist as an input and finds a similar song to recommend, specifically by finding which other song appears most often on the same playlists. First, this system searches a dataset from Spotify for all playlists this song appears on. This list of playlists is then used to find all *songs* that appear on these playlists. The number of occurrences of each song is counted and duplicates are removed. Whichever song has the highest count is returned as a recommendation. The intent of this system is to be implemented in a music streaming service as an added value for their user base. Its design intent is to be able to handle a large number of recommendations while working with a large dataset.

Slide 2: What is the problem?

Querying a database of over a million songs can be a very computationally heavy task, especially when the system is trying to make many recommendations at once. This can result in significant lag or the system crashing. A considerable portion of the processing power is tied up in creating and destroying threads for each task as they arise, creating a need for some way to manage the collection of threads being used.

Slide 3: Solution

The solution to this problem is the thread pool pattern. This pattern solves the previously mentioned problem by creating a pool of threads that can be reused to avoid the latency created by continuously creating and destroying threads. My implementation of this uses an abstract class called Task that has 4 subclasses of the specific tasks. A worker class which implements runnable gets assigned a task subclass object and calls a method called DoTask from the object. The last step of each task is to save the generated data to a CSV file for the next task to pick up where it left off. After each task is complete, it deletes the CSV file from the previous task.

Slide 4: Important parts of code

One of the main essential areas of implementation is the DoTask method in the Task class. This method handles most of the process of executing the task. It does this by creating a ProcessBuilder object which effectively performs the same function as submitting the given arguments to the command line. In this case it’s calling a Python script called pyTasks which contains each task. I originally started implementing this all in Java but quickly found Python to be much more efficient at executing a given task. Specifically, analyzing the original dataset took at best 9 minutes for the Java script, and 4 seconds for the python script. Additionally the Python script is part of a larger project of mine focusing on music recommendation through machine learning. The results from the process builder object are printed to the command line from the python script and returned by the get input stream method. They are then passed to the process results method before getting returned. The process results method was created as abstract to force the task subclasses to implement their own way of processing the results to deal with whatever exceptions that may occur as a result of their python task. This is notably important as an exception in the python script does not get passed back to the process builder, resulting in either an exception at an unexpected time, or a blank recommendation being returned. So extra caution must be taken for how each task processes the results.

Slide 5: Important parts of code

This is from a JUnit test to test the function of passing multiple inputs at once. This snippet of code shows the executor service object and how it allows for easy creation and management of the thread pool. The submit method on the executor object is one of the ways the class offers to execute the task. The shutdown method closes all threads after they have completed while blocking the submission of any new threads. Additionally, another class in the concurrent library, the thread pool executor class, allows access to information about the thread pool itself such as number of currently running threads, tasks completed, set thread timeout limit, and more.

Slide 6: Important parts of code

The last areas of code to highlight is the AtomicInteger object. Managing variables between threads can create race conditions resulting in unexpected and hard to trace bugs. The solution to this is the atomic integer class. Admittedly, it is also possible to create a class to generate ids and use the synchronized keyword to ensure thread safety, but this introduces the use of locks. Locks have a couple of disadvantages being that when multiple threads try to get a lock, only one can win and the rest must wait. Additionally the process of suspending and resuming a thread can be very expensive. This is where atomic operations are useful. Atomic operations exploit low-level machine instructions to ensure data integrity by attempting to find ways to do common high-level tasks in a single low-level step. Take for example an integer called myInt using two plus signs to increment its value. While this may look like one step, this process entails obtaining the value, incrementing it, and writing the updated value. In contrast a similar atomic operation would be compare and swap. This operation is a low-level machine instruction that atomically updates the value of a given memory location to the value that needs to be set, as long as that memory location matches the expected input before update, otherwise nothing is done. If multiple threads were to attempt to perform this process at once, still only one would be successful, but none would be locked.

Slide 7: Results and conclusion

While the system itself could absolutely be expanded to incorporate other parts of the machine learning program, the thread pool pattern did seem to effectively allow for faster processing of large amounts of inputs and a large dataset. Between the executor service and the atomic integers, managing and keeping track of a thread pool with a decently high granularity is a very easy task. And the design pattern of the tasks allows for new tasks to be easily added in whenever needed by simply extending the task class. While it can speed the process up, implementing multiple threads inevitably makes any system more complex and consequently results in more potential areas for bugs to arise. Additionally, atomic operations are possible due to the ability to implement efficient machine level atomic instructions that are available on contemporary processors. However, for some platforms support for these instructions may entail some internal locking, meaning locks may be implemented anyways.