4.1 Lecture Summary

4.1 Combining Distribution and Multithreading

Lecture Summary: In this lecture, we introduced *processes* and *threads*, which serve as the fundamental building blocks of distributed computing software. Though the concepts of processes and threads have been around for decades, the ability to best map them on to hardware has changed in the context of data centers with multicore nodes. In the case of Java applications, a process corresponds to a single Java Virtual Machine (JVM) instance, and threads are created within a JVM instance. The advantages of creating multiple threads in a process include increased sharing of memory and per-process resources by threads, improved responsiveness due to multithreading, and improved performance since threads in the same process can communicate with each other through a shared address space. The advantages of creating multiple processes in a node include improved responsiveness (also) due to multiprocessing (e.g., when a JVM is paused during garbage collection), improved scalability (going past the scalability limitations of multithreading), and improved resilience to JVM failures within a node. However, processes can only communicate with each other through message-passing and other communication patterns for distributed computing.

In summary, processes are the basic units of distribution in a cluster of nodes - we can distribute processes across multiple nodes in a data center, and even create multiple processes within a node. Threads are the basic unit of parallelism and concurrency -- we can create multiple threads in a process that can share resources like memory, and contribute to improved performance. However, it is not possible for two threads belonging to the same process to be scheduled on different nodes.

Optional Reading:

1. Wikipedia articles on processes and threads .

4.2 Lecture Summary

4.2 Multithreaded Servers

Lecture Summary: In this lecture, we learned about *multithreaded servers* as an extension to the servers that we studied in client-server programming. As a motivating example, we studied the timeline for a single request sent to a standard sequential file server, which typically consists of four steps: A) accept the request, B) extract the necessary information from the request, C) read the file, and D) send the file. In practice, step C) is usually the most time-consuming step in this sequence. However, threads can be used to reduce this bottleneck. In particular, after step A) is performed, a new thread can be created to process steps B), C) and D) for a given request. In this way, it is possible to process multiple requests simultaneously because they are executing in different threads.

One challenge of following this approach literally is that there is a significant overhead in creating and starting a Java thread. However, since there is usually an upper bound on the number of threads that can be efficiently utilized within a node (often limited by the number of cores or hardware context), it is wasteful to create more threads than that number. There are two approaches that are commonly taken to address this challenge in Java applications. One is to use a *thread pool*, so that threads can be reused across multiple requests instead of creating a new thread for each request. Another is to use lightweight tasking (e.g., as in Java's ForkJoin framework) which execute on a thread pool with a bounded number of threads, and offer the advantage that the overhead of task creation is significantly smaller than that of thread creation. You can learn more about tasks and threads in the companion courses on "Parallel Programming in Java" and "Concurrent Programming in Java" in this specialization.

Optional Readings:

1. Wikipedia article on thread pools.

2. Tutorial on Java's fork/join framework (also covered in Parallelism course).

4.3 MPI and Multithreading

Lecture Summary: In this lecture, we learned how to extend the *Message Passing Interface (MPI)* with threads. As we learned earlier in the lecture on Processes and Threads, it can be inefficient to create one process per processor core in a multicore node since there is a lot of unnecessary duplication of memory, resources, and overheads when doing so. This same issue arises for MPI programs in which each rank corresponds to a single-threaded process by default. Thus, there are many motivations for creating multiple threads in an MPI process, including the fact that threads can communicate with each other much more efficiently using shared memory, compared with the message-passing that is used to communicate among processes.

One approach to enable multithreading in MPI applications is to create one MPI process (rank) per node, which starts execution in a single thread that is referred to as a *master thread*. This thread calls MPI_Init() and MPI_Finalize() for its rank, and creates a number of *worker threads* to assist in the computation to be performed within its MPI process. Further, all MPI calls are performed only by the master thread. This approach is referred to as the MPI_THREAD_FUNNELED mode, since, even though there are multiple threads, all MPI calls are "funneled" through the master thread. A second more general mode for MPI and multithreading is referred to as MPI_THREAD_SERIALIZED; in this mode, multiple threads may make MPI calls but must do so one at a time using appropriate concurrency constructs so that the calls are "serialized". The most general mode is called MPI_THREAD_MULTIPLE because it allows multiple threads to make MPI calls in parallel; though this mode offers more flexibility than the other modes, it puts an additional burden on the MPI implementation which usually gets reflected in larger overheads for MPI calls relative to the more restrictive modes. Further, even the MPI_THREAD_MULTIPLE mode has some notable restrictions, e.g., it is not permitted in this mode for two threads in the same process to wait on the same MPI request related to a nonblocking communication.

Optional Reading:

1. Lecture by Prof. William Gropp on MPI, Hybrid Programming, and Shared Memory.

4.4 Lecture Summary

4.4 Distributed Actors

Lecture Summary: In this lecture, we studied *distributed actors* as another example of combining distribution and multi-threading. An actor is an object that has a mailbox, local state, a set of methods and an active (logical) thread of control that can receive one message at a time from the mailbox, invoke a method to perform the work needed by that message, and read and update the local state as needed. Message-passing in the actor model is *nonblocking* since the sender and receiver do not need to wait for each other when transmitting messages. We also studied a simple algorithm for generating prime numbers, called the Sieve of Eratosthenes, that can be conveniently implemented using actors. The actor paradigm is well suited to both multicore and distributed parallelism, since its message-passing model can be implemented efficiently via shared memory within a single process or in a more distributed manner across multiple processes.

Most actor implementations that support distributed execution require you to perform the following steps. First, you will need to use some kind of *configuration file* to specify the host process on which each actor will execute as well as the port that can be used to receive messages from actors on other processes. Second, you will need the ability to create actors on remote processes. Third, you will need to provide some kind of logical name to refer to a remote actor (since a reference to the actor object can only be used within the process containing that actor). Finally, messages transmitted among actors that reside in different processes need to be serialized, as in client-server programming. Besides these steps, the essential approach to writing actor programs is the same whether the programs execute within a single process, or across multiple processes.

Optional Readings:

1. Wikipedia article on the Actor Model (also covered in the Concurrency course).

2. Example of distributed actor configurations in article on Location Transparency in Akka actors.

4.5 Lecture Summary

4.5 Distributed Reactive Programming

Lecture Summary: In this lecture, we studied the r*eactive programming* model and its suitability for implementing distributed service oriented architectures using *asynchronous events*. A key idea behind this model is to balance the “push" and “pull" modes found in different distributed programming models. For example, actors can execute in push mode, since the receiver has no control on how many messages it receives. Likewise, Java streams and Spark RDDs operate in pull mode, since their implementations are demand-driven (lazy). The adoption of distributed reactive programming is on a recent upswing, fueled in part by the availability of the Reactive Streams specification which includes support for multiple programming languages. In the case of Java, the specification consists of four interfaces: Flow.Publisher, Flow.Subscriber, Flow.Process, and Flow.Subscription.

Conveniently, there is a standard Java implementation of the Flow.Publisher interface in the form of the SubmissionPublisher class. If we create an instance of this class called pub, a publisher can submit information by calling pub.submit(). Likewise, a subscriber can be registered by calling pub.subscribe(). Each subscriber has to implement two key methods, onSubscribe() and onNext(). Both methods allow the subscriber to specify how many elements to request at a time. Thus, a key benefit of reactive programming is that the programmer can control the "batching" of information between the publisher and the subscriber to achieve a desired balance between the "push" and "pull" modes.

Optional Reading

1. Article on Reactive Programming with JDK 9 Flow API.

2. Wikipedia article on reactive programming.

Mini Project 4: Multi-Threaded File Server

---

miniproject_4.zip

Project Goals and Outcomes

In general, most production JVM web servers are multi-threaded. Using multiple threads allows a single JVM server instance to handle many concurrent requests at once.

In Mini Project 2 of this course, you developed a simple file server that responded to HTTP GET requests with the contents of files. In this mini-project, we will make a simple extension to that file server by using multiple Java Threads to handle file requests.

If you need a refresher on HTTP, file servers, or the problem statement you are free to refer back to the Mini Project 2 description in Coursera and to your own Mini Project 2 solution.

Note that for this mini-project, the expected performance is highly dependent on the execution platform you use. The provided performance tests are calibrated to work well on the Coursera autograder, and may fail in your local testing environment even when they would pass on the autograder. Once you see speedup using multiple threads on your local machine, we encourage you to also test your submission on the autograder even if local testing is failing.

Project Setup

Please refer to Mini-Project 0 for a description of the build and testing process used in this course.

Once you have downloaded and unzipped the project files using the gray button labeled miniproject_4.zip at the top of this description, you should see the project source code file at

miniproject_4/src/main/java/edu/coursera/distributed/FileServer.java

and the project tests in

miniproject_4/src/test/java/edu/coursera/distributed/FileServerTest.java

It is recommended that you review the demo video for this week before starting this assignment, in which Professor Sarkar works through a similar example.

Project Instructions

Your modifications should be made entirely inside of FileServer.java. You should not change the signatures of any public or protected methods inside of FileServer.java, but you can edit the method bodies and add any new methods that you choose. We will use our copy of FileServerTest.java in the final grading process, so do not change that file or any other file except FileServer.java.

Your main goals for this assignment are as follows:

1. Inside FileServer.java, implement the run method to use the provided ServerSocket object and root directory to implement a multi-threaded HTTP file server by handling each request received on the ServerSocket in a new thread. There are several helpful TODOs in FileServer.java to help guide your implementation.

Like in Mini Project 2, you are provided with a proxy filesystem class called PCDPFilesystem which you will use to read files requested by file server clients. The only PCDPFilesystem API you need to be aware of is called readFile, which accepts a PCDPPath object representing the path to a file you would like the contents of. readFile returns the contents of the specified file if it exists. Otherwise, it returns null.

The PCDPPath object has a single constructor which accepts the absolute path to a file as a String. Hence, you can construct a PCDPPath object after parsing the requested file from an HTTP GET request.

Project Evaluation

Your assignment submission should consist of only the FileServer.java file that you modified to implement this mini-project. As before, you can upload this file through the assignment page for this mini-project. After that, the Coursera autograder will take over and assess your submission, which includes building your code and running it on one or more tests. Your submission will be evaluated on Coursera’s auto-grading system using 2 and 4 CPU cores. Note that the performance observed for tests on your local machine may differ from that on Coursera's auto-grading system, but that you will only be evaluated on the measured performance on Coursera. Also note that for all assignments in this course you are free to resubmit as many times as you like. See the Common Pitfalls page under Resources for more details. Please give it a few minutes to complete the grading. Once it has completed, you should see a score appear in the “Score” column of the “My submission” tab based on the following rubric:

- 5% - correctness of testTermination
- 5% - correctness of testFileGet
- 5% - correctness of testFileGets
- 10% - correctness of testNestedFileGet
- 10% - correctness of testDoublyNestedFileGet
- 10% - correctness of testLargeFileGet
- 10% - correctness of testMissingFileGet
- 10% - correctness of testMissingNestedFileGet
- 15% - performance of testPerformance on 2 cores
- 20% - performance of testPerformance on 4 cores