User Id: wjz224

CSE 262: Quiz #5

Due Nov 20th, 2022 at 11:59 PM

The quiz has TWO questions. Please submit your answer by adding a file named `<<your username>>_q5.pdf` to the quizzes folder of your Bitbucket account (e.g., mfs409_q5.pdf), and then committing and pushing. You should use as much space as you want for each answer. Please be detailed in your answers. Remember: this quiz is worth 9% of your grade.

Question 1: While we discussed the importance of language-level support for *concurrency*, it is not as clear that language support for *parallelism* is necessary. Investigate the Intel Threading Building Blocks library (Intel TBB, now part of Intel OneAPI). Is it satisfactory for enabling easy access to multiple cores? Be sure to justify your answer. You might want to look into how TBB was affected by the changes in C++11, especially C++11's lambdas.

Concurrency is the operation of making progress on more than one task seemingly at the same time. Since a single CPU processor can only run one task a time, concurrency operates on one task for a short period of time, then it switches to another task for a short period of time, and switches back and forth between the threads (each separate task); this happens so fast that the user thinks each of the tasks are processing concurrently, hence the name "concurrency". On the other hand, there is parallel execution. Parallel execution is the operation of making progress on more than one task at the exact same time. This requires multiple CPU processors (or cores) since one CPU core can only run one task at a time. Knowing these different types of operations, they can be used conjointly to perform parallelism.

Parallelism is the operation of splitting a single task into subtasks which can be executed in parallel that may also use concurrency. These subtasks are often parts of the single task that can be executed independently (meaning each task is a complete action) to avoid errors. Afterwards, we leave it up to the operating system to allocate the subtasks to available CPU cores to work on. Though, often there may not be enough CPU cores available for each subtask. This may be because other tasks are currently using some CPU cores or there are simply not enough CPU cores. When this happens, concurrency is used for the subtasks along with parallel execution (if there are enough CPUs for parallel execution) to put each subtask into their own thread that may be in parallel with another subtask in a different CPU core's thread, while also being in concurrency with another subtask.

Language-level support is very important for concurrency for a lot of obvious reasons. One of the reasons is I/O intensive applications. Concurrency can help speed up input and output between disk writing, DB, network, and, etc. by offloading writing an output to a thread while the main program can be reading an input in another thread. Another reason is that concurrency can speed up large computational problems by reducing CPU core idle times (single thread would spend a lot of time in idle) and using it to perform other tasks. There are many more reasons why concurrency is important, especially its contribution to processing speed, but I will only mention two.

Evidently, concurrency is very important, but what about language-level support for parallelism? Well, parallelism is also very important because it makes processing even faster. Modern day processors have reached maximum clock speed, therefore the only way to further speed up our processing speed is parallelism. It creates multiple concurrent threads that allows communication with each other if necessary. This allows information to be fed to another task that may need the information before it executes. Consequently, parallelism improves scalability because the ability to interchange information between tasks allows the system to adapt to the changes in the application and system processing demands better.

In the modern day, language-level support for parallelism has become a standard. To further prove the importance of parallelism, I would like to give an example. One example would be C++ and the Threading Build Blocks (TBB) C++ library developed by Intel for making use of parallelism across multicore processors. C++ is a language that supports parallelism, and the TBB library developed by Intel is specifically designed to take advantage of that support. One example of C++ providing language-level support for parallelism is lambda expressions. The C++11 lambda expressions made the TBB's parallel for much easier to use by letting the compiler do the tedious work of creating a function object. It avoids to the need to introduce extra classes to encapsulate code as functions. This is just one of many language-level supports that C++ has provided for parallelism.

Developers of TBB took advantage of the language-level support for parallelism by creating operations that take advantages of multiple cores and dynamically distributes the work of the application across the various cores. They were able to do this by having experienced programmers code parallel solutions to common programming functions and tasks, such as random number generation, multiplying matrices, and much more, and included it in the library for users with multiple cores to take advantage of parallelism. Many tests have shown that the TBB and their usage of parallelism has consistently had the quickest execution time and the best scalability. The ability to run multiple tasks at the exact time obviously saves time, which saves money and allows us to solve larger and more complex problems. Therefore, it is obvious that language support for parallelism is important.

Question 2: The Node.js framework for writing server-side JavaScript relies heavily on nested callbacks. This has led to something often called "callback hell". How has JavaScript changed to remedy this problem? What are the strengths and weaknesses of these changes?

In an application, sometimes not all the code will be perfectly executed and finished from top to bottom. Sometimes there may be a delay that causes a line of code to finish later than the line of code succeeding it. This may cause issues when the line succeeding it relies on the previous line of code's information. One example to illustrate this is downloading information from a database and printing it. The database that we are requesting information from may be located in a very far place, causing the download to take a very long time. This download time may take a longer time than it takes to execute the next line of code, which is printing the information. But since the information has not been downloaded yet, there is no information to print, which leads to an error. To fix this, JavaScript uses something called callbacks.

Callbacks are functions that are passed to other functions as a parameter which can be used to enforce the order of operations that we want. For example, we may have a function that originally downloads the information from a database, and succeeding it was the function that printed the information. Now with callbacks, we can pass the printing information function as a parameter to the downloading function and enforce that the print statement happens after the information is downloaded.

Now, callbacks are nice, but sometimes we need to call a callback multiple times in a specific order. This causes us to nest callbacks within each other to ensure the order. Nesting callbacks can appear very ugly and hard to read because of all the "})" syntax. This situation is often referred to as "callback hell". Callback hell often occurs from poor coding practices, and one of the most common ones is writing JavaScript where execution visually happens from top to bottom. To fix this issue, JavaScript introduced "Promises" to remedy this problem.

JavaScript promises operate on the idea of an actual "promise" in real life. A promise has two results. Either a promise was completed and resolved, or a promise failed and was rejected. The syntax of a promise consists of two possible returns, a resolve and a reject. The resolve and the reject returns can be anything that is passed in. The main idea of a promise is to use the ".then" method and "catch" method. The ".then" and "catch" method waits for the lines of code within the promise to finish and return either a resolve or reject. If the promise returns a resolve, then the code within the ".then" method block would execute with access to the returned values passed into the resolve. If the promise returns a reject, then the code within the "catch" method block would execute with access to the returned values passed into the reject. This is nice because the promise waits for a response before it does a specific action that is specified within the ".then" and "catch" method which removes the need for callbacks. Consequently, this allows us to execute the rest of our code while allowing the parts that need to wait for a response to execute a specific task to wait independently. With the removal of callbacks and with the ".then" and catch methods, this fixes the problem of callback hell by removing the need of nested callbacks. For example, look at snippet of code below for a comparison of one with callback and another with a promise (I used geeksforgeeks code as reference):

```
// function that adds two numbers using callbacks

function foo(a,b, callback) {

            setTimeout(() => {

                        callback(a +b);

            }, 2000 )

};

// using nested call backs to calculate the sums of 4 numbers

foo(4, 6, (c1) => {
  foo(7, c1, (c2) => {
    foo(8, c2, (c3) => {
      console.log(`Sum of 4 numbers is ${c3}`);
    });
  });
})
```

```
// function that adds two numbers using  a promise

function fooPromise(){

             return new Promise((resolve, reject) => {

                        setTimeout(() => {

                            resolve(a+b);
                        }, 2000);

                    });
            };

// using nested promise to calculate the sums of 4 numbers

fooPromise(4, 6)
  .then((c1) => {
   return fooPromise(7, c1);
  })
  .then((c2) => {
    return fooPromise(8, c2);
  })
  .then((c3) => {
         console.log(`Sum of 4 numbers is ${c3}`);
    );
  });
```

As you can see, using the promises ".then" and "catch" method, it removes the need to nest callbacks within each other and it provides much better readability than with callbacks.

User Id: wjz224

The main advantages seen so far with promises over callbacks are better readability, easy error handling (with catch blocks), and more specific and defined code logic. Evidently, there are many advantages that promises has over callbacks, so why do callbacks exist? Well, this is because there are weaknesses to promises that callbacks don't have. One of the major weakness of promises is that it kills the purpose of asynchronous non-blocking I/O. A nonblocking I/O is a thread that is not stuck waiting for a request to finish. Therefore, there is no need to use something like the "setTimeOut" method. Another weakness of promises is that they can only return one object and we cannot return multiple arguments. This could be a downside where a method needs to return multiple arguments. Overall, promises do resolve the issue of "callback hell" and even provides better structured code, but its weaknesses may lead some programmers to use callbacks still.