User Id: ___zhw323_____

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.

Intel TBB is a ++ library developed by Intel for making use of multicore processors. Developers express operations that are to take advantage of multiple cores and their dependencies through high level algorithms. The runtime environment then dynamically distributes the work across the various cores. TBB had been a good choice of adding parallelism and concurrency to codes. However, since C++ 11 published, there had been a debate between Standard C++ and TBB. C++ 11 made a big change by adding concurrent feature to standard library (thread class) which simplified the usage of concurrency. During investigation, I found out that developers at that time were spitted into those who preferred new functionality of C++ 11 and those who sticked with TBB. From the perspectives of whom supported C++ 11, their claim includes "The built-in allocators that come with the C++11 compilers are designed to handle concurrent memory allocations and generally are both faster and more reliable. The TBB allocators are now obsolete", "TBB can't work with the new lambda feature"(Based on the official Intel advisor user guide, even now, it is still complicated to make many intel® oneAPI Threading Building Blocks constructs easier to program because of the need to introduce extra classes to encapsulate code as functions, and we need a complier such like Intel® C++ Compiler Classic or Intel® oneAPI DPC++/C++ Compiler to enable lambda expression support with TBB.), etc..... On the contrary, TBB do included many powerful features that can really simplify access to multiple cores. TBB operates with tasks, not threads. TBB's task-based-scheduler utilizes all cores by allocating a pool of threads and letting it dynamically select which tasks to run, and it also implements concurrent queue , with which developers need to map available work to threads and construct concurrent queue manually. And then TBB offers high-level constructs such as parallel_for, parallel_pipeline, etc. that can be used to express most common parallel patterns, and hide all manipulation with tasks. TBB will automatically distribute all loop iterations over available cores and dynamically balance the load so that if some thread has more work to do, other threads don't just wait for it but help, maximizing CPU utilization. Additionally, TBB supports nested parallelism. This means that you can write parallel code that itself calls functions that involve parallel code, with you being safe in the knowledge that multiple levels of parallelism will not overload your computer. Lastly, in the paper, *Benchmarking Usability and Performance of Multicore Languages*, Sebastian Nanz, Scott West, Kaue Soares da Silveira, and Bertrand Meyer compared several different popular approaches to multicore programming. They concluded that "the library [TBB] provides is the most comprehensive of the four languages, containing algorithmic skeletons, task groups, synchronization and message passing facilities. The high-level parallel algorithms were sufficient to implement every task in the benchmark set without dropping down to lower level primitives such as manual task creation and synchronization. Being a library for a well-known language, it also has the fastest coding times". (retrieved from https://www.computerworld.com/article/2845683/why-threading-building-blocks-are-the-best-multicore-programming-solution.html, November, 17th ,2022).Therefore, although TBB had a couple of problem with C++ 11(Probably fixed right now), it is still a good choice for programs that access the multi-cores.

User Id: ___zhw323_____

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 JavaScript, callbacks are functions that take time to produce a result(eg. Accessing values from database). Other lines of code can not be executed before callback returns for the reason that it might throw an error or other functions relies on the result of callback. Therefore callback hell is an issue when we have code with complex nested callbacks. It is hard to read and maintain such code, and if there is an error in one function, all other functions get affected.

One way JavaScript has changed to help with callback hell is the introduction of the async/await keywords. These keywords allow developers to write asynchronous code that looks and feels like synchronous code. This makes code much easier to read and write, and can help avoid callback hell altogether.

Another important change is the Promise object and event queue. A promise is a returned object from any asynchronous function, to which callback methods can be added based on the previous function’s result. .then() method can chain as many callbacks as we want and the order is also strictly maintained. .fetch() method to fetch an object from the network ,and .catch() method to catch any exception when any block fails. Promises are put in event queue so that they don’t block subsequent code. Also once the results are returned, the event queue finishes its operations. Therefore, It allows the async call to return a value just like the synchronous function and doesn’t need to run extra checks or try/catch for the error handling. Therefore, speaking of its strength, Promise enhances readability, reduces coupling, provides better exception and error handling, provides a well-defined control flow, and provides functional programming semantics

There are some drawbacks to these changes, however. For async/await, one is that async/await can make code more difficult to understand for beginners. Additionally, comparing to Promise, its error handling is done using .try() and .catch() methods manually. For Promise objects, it can be harder to work with than traditional callback functions. It also kills the purpose of asynchronous non-blocking I/O as well.