#### 1) **does nodejs spawns new threads or process for every new HTTP request?**

Node.js is built on an event-driven, non-blocking I/O model that uses a single-threaded event loop to handle asynchronous operations. This means that by default, Node.js runs in a single thread and does not spawn new threads or processes for every new HTTP request. Instead, it uses an event loop to efficiently handle multiple concurrent requests.

When an HTTP request is received, Node.js processes it asynchronously without blocking the main thread. If there are I/O operations (such as reading from files, making network requests, etc.), Node.js delegates those operations to its underlying asynchronous APIs, allowing the main thread to continue handling other tasks while waiting for the I/O to complete. Once the I/O operation is finished, a callback is t

.riggered, and the event loop processes it.

This architecture allows Node.js to handle a large number of concurrent connections efficiently without the overhead of creating new threads or processes for each request. It is particularly suited for applications that involve I/O-heavy operations, such as web servers, because it can handle many connections simultaneously while keeping resource usage relatively low.

It's worth noting that while Node.js itself uses a single thread for event handling, some operations in Node.js, like certain native modules or worker threads, can utilize additional threads or processes when explicitly programmed to do so. However, the core event loop and HTTP request handling remain single-threaded.

**2)**

*http.get("/something/all",(req, res)=>{*

*// heavly I/O call or db call*

*db.query("SELECT \* ....", function(result){*

*....*

*for(i=0;i<result.length;i++){*

*// loop through and build result*

*}*

*response.end(modifiedResult)*

*})*

*})*

**Take this example.**

**So basically when 50 HTTP requests are made to this API, how does node handle this**

In the scenario you've described, where 50 HTTP requests are made to an API that involves heavy I/O calls or database calls, Node.js uses its event-driven, non-blocking architecture to handle these requests efficiently. Here's how it would work:

1. \*\***Request Reception**\*\*: When a request is received, the event loop in Node.js triggers the corresponding callback (the function you provided in `http.get`). This callback is executed asynchronously, without blocking the main thread, allowing Node.js to continue processing other events.

2. \*\***I/O Operations**\*\*: Within the callback, you have an I/O-intensive operation (database query). Node.js delegates this operation to its asynchronous I/O APIs. While waiting for the database query to complete, Node.js doesn't just sit idle; it continues processing other events and incoming requests.

3. \*\***Callback Execution**\*\*: When the database query is finished, its callback function is triggered. This callback function processes the results of the query.

4. \*\***Response Building**\*\*: Within the database query's callback, you're building a response by looping through the results and modifying them. This is also done asynchronously, and Node.js continues processing other events during this time.

5. \*\***Response Sending**\*\*: Once the response is built, you use the `response.end()` method to send the modified result to the client.

Since Node.js's event loop allows it to handle multiple events concurrently without blocking the main thread, it can effectively manage these 50 HTTP requests without creating a separate thread or process for each request. While some requests might be waiting for I/O operations to complete, Node.js can still process other requests and events, making efficient use of the available resources.

It's important to note that if the I/O operations or database queries are slow, they can potentially become a bottleneck, impacting the overall throughput of your server. To mitigate this, you might consider strategies like connection pooling, optimizing database queries, and making use of asynchronous programming patterns to ensure that your application remains responsive and efficient even under heavy load.

**3) how it processes 50 concurrent connection if it doesnot spawn new threads or process?**

Node.js is designed to handle a large number of concurrent connections without the need to spawn new threads or processes. It achieves this through its event-driven, non-blocking architecture. Here's how it processes 50 concurrent connections without creating new threads or processes:

1. \*\***Single Thread**\*\*: Node.js operates in a single thread. This thread is responsible for managing the event loop and executing JavaScript code.

2. \*\***Event Loop**\*\*: When multiple HTTP requests come in, Node.js's event loop efficiently manages these requests. Each incoming request is treated as an event.

3. \*\***Non-Blocking I/O**\*\*: When a request involves I/O operations (such as reading from a file or making a database query), Node.js delegates these operations to its underlying asynchronous APIs. While waiting for I/O to complete, the main thread is not blocked and can continue processing other events.

4. \*\***Callbacks and Asynchronous Code**\*\*: The asynchronous nature of Node.js allows it to efficiently switch between different tasks without waiting for one to complete before moving on to the next. Callback functions are used to handle the results of I/O operations once they're finished.

5. \*\***Concurrency**\*\*: While one request's I/O operation is in progress, Node.js can continue processing other requests that are either awaiting I/O or performing non-blocking operations. This concurrency is managed by the event loop, which prioritizes tasks based on their readiness.

6. \*\***Scaling**\*\*: Node.js leverages the operating system's underlying asynchronous I/O capabilities, which are often implemented using mechanisms like epoll on Linux. This allows Node.js to handle a high number of connections concurrently.

7. \*\***Resource Efficiency**\*\*: By avoiding the overhead of creating new threads or processes for each request, Node.js maintains a low memory footprint. This makes it suitable for applications that require high scalability and efficient use of resources.

Overall, Node.js's ability to handle multiple concurrent connections is based on its event-driven, non-blocking architecture, which allows it to efficiently manage I/O operations and callbacks without the need to spawn new threads or processes. This approach can result in high throughput and responsiveness for applications, especially those that involve I/O-intensive tasks like web servers.

**4) explain me the scenarion where two req comes in at the same time.**

**Assume there is no I/o operations under that req callback**

**u are saying it'll not spawn anything then how on earth it'll handle multiple request paralley.. doesn't the other requests have to wait**

Certainly, let's go through a scenario where two requests come in at the same time, assuming there are no I/O operations directly within the request callback. In this scenario, we'll focus on the basic event loop behavior of Node.js.

1. \*\***Two Requests Simultaneously**\*\*:

Let's say two HTTP requests, Request A and Request B, arrive at the server at the same time.

2. \*\***Event Loop**\*\*:

Node.js processes these requests within its single-threaded event loop. The event loop manages the execution of various tasks, and it maintains a queue of pending tasks to be executed.

3. \*\***Callbacks Execution**\*\*:

Both Request A and Request B have corresponding callbacks associated with them. These callbacks are executed in the order they were received.

4. \*\***Non-Blocking Nature**\*\*:

If there are no blocking operations or significant computational tasks within the callbacks, Node.js can quickly execute both callbacks without having to wait for one to finish before starting the other.

5. \*\***Concurrency**\*\*:

Even though Node.js operates in a single thread, it can switch between executing tasks rapidly. When Request A's callback is executing, and it doesn't block the event loop, Node.js can quickly move on to execute Request B's callback.

6. \*\***Time Sharing**\*\*:

Node.js's fast context switching allows it to time-share between the callbacks, making it seem like both requests are being processed concurrently. While they are being processed one after the other in the same thread, the rapid context switching creates the illusion of parallel processing.

7. \*\***Response**\*\*:

Once both callbacks are executed, the respective responses for Request A and Request B are sent back to their corresponding clients.

In this scenario, Node.js leverages its ability to quickly switch between tasks and execute callbacks in a non-blocking manner. This allows it to provide concurrency and handle multiple requests seemingly in parallel, even though it's happening within a single thread. The absence of blocking operations or significant computational tasks within the callbacks ensures that the event loop can keep moving forward without getting stalled by any single request's processing.

It's important to note that while this approach provides concurrency and efficient handling of multiple requests, it might not be suitable for all scenarios. Node.js's strength lies in its ability to handle I/O-intensive operations concurrently. If there were CPU-bound tasks within the callbacks, they could still block the event loop and impact the overall responsiveness of the server.

**5) now explain this same scenarion in case of a GoLang HTTP server**

In a GoLang HTTP server, the concurrency model is different from Node.js's event-driven architecture. GoLang's concurrency is built around lightweight threads called "goroutines" and a synchronization mechanism called channels. Let's walk through the scenario where two requests come in at the same time in a GoLang HTTP server:

1. \*\***Two Requests Simultaneously**\*\*:

Similar to the Node.js scenario, two HTTP requests, Request A and Request B, arrive at the GoLang server at the same time.

2. \*\***Goroutines**\*\*:

In GoLang, each incoming request is typically handled within its own goroutine. Goroutines are lightweight threads that are managed by the Go runtime. Unlike traditional threads, goroutines have a much smaller memory footprint and are more efficient to create and manage.

3. \*\***Goroutine Creation**\*\*:

When Request A comes in, a new goroutine is spawned to handle its processing. Simultaneously, when Request B arrives, another goroutine is created to handle its processing. These two goroutines can be created quickly and efficiently due to their lightweight nature.

4. \*\***Concurrency**\*\*:

GoLang's runtime scheduler manages the execution of these goroutines across the available CPU cores. The Go runtime employs a technique called "multiplexing" to efficiently switch between active goroutines, allowing them to run concurrently.

5. \*\***Non-Blocking I/O**\*\*:

If there are I/O operations involved in the processing of the requests (like reading from files or making network calls), the goroutines can yield control to the scheduler while waiting for I/O to complete. This non-blocking behavior allows other goroutines to execute while one is waiting.

6. \*\***Synchronization**\*\*:

If there are shared resources that need coordination, GoLang provides channels for synchronization. Goroutines can communicate with each other using channels, ensuring safe concurrent access to data.

7. \*\***Response**\*\*:

Once the processing for Request A and Request B is complete, and if there are response actions to be taken, the respective responses are sent back to the clients.

In this scenario, GoLang's goroutines provide a more traditional multithreading-like concurrency model. Each incoming request can be handled within its own goroutine, which allows the GoLang server to efficiently utilize available CPU cores. The lightweight nature of goroutines, combined with the runtime's scheduling and synchronization mechanisms, enables effective parallelism without creating heavy overhead.

It's important to note that both Node.js and GoLang have their strengths and trade-offs when it comes to handling concurrent connections. Node.js's event-driven model is well-suited for I/O-intensive operations, while GoLang's goroutines are designed to efficiently manage concurrency and parallelism across multiple CPU cores. The choice between them depends on the specific requirements of your application.

**6) means in GoLang if i have two incoming request (assuming I've a dual core cpu). then it means it'll spawn two go routines, and if i have a db call defined. Then those two routiones will parallely call the DB then it means those individual goroutine will be blocked in their own space right? but since they are being handled parallely so its non-io blocking right?**

Yes, your understanding is correct. In a GoLang server running on a dual-core CPU, if you have two incoming requests, GoLang will indeed spawn two goroutines to handle these requests. If each of these requests involves a database call, the goroutines can run the database calls concurrently and in parallel, effectively utilizing the available CPU cores. Here's how this works:

1. \*\***Goroutine Creation**\*\*: When two incoming requests arrive simultaneously, GoLang's server will create two separate goroutines to handle them.

2. \*\***Concurrency and Parallelism**\*\*: Since your CPU has two cores, these two goroutines can run in parallel on the two available cores. This allows the server to efficiently utilize the processing power of the CPU.

3. \*\***Database Call**\*\*: If the requests involve a database call, each goroutine can initiate its own database query. These database queries can be executed concurrently, thanks to the parallelism provided by the multiple CPU cores.

4. \*\***Non-Blocking I/O**\*\*: While the database queries are in progress, the goroutines are not blocked waiting for the I/O to complete. Instead, they yield control to the Go runtime scheduler, allowing other goroutines to run. This ensures that the server remains responsive and can continue processing other requests or performing other tasks while waiting for I/O operations to finish.

5. \*\***Synchronization and Response**\*\*: If there's shared data that needs coordination between the goroutines, GoLang provides synchronization mechanisms such as channels. Once the database queries complete and the responses are ready, the goroutines can handle the responses and send them back to the respective clients.

In this scenario, even though the individual goroutines may wait for the database I/O to complete, their non-blocking behavior ensures that they don't monopolize CPU resources. Other goroutines can continue running in the meantime, providing concurrency and parallelism.

GoLang's lightweight goroutines and efficient concurrency model make it well-suited for applications that require high parallelism and concurrent I/O operations, like web servers that handle multiple requests simultaneously.

**7) on point number 4,**

**clear me out.. in OS theories**

1. *if thread is blocked then entire process if blocked right?*
2. *by yeild u mean the goroutine here is not blocked but authority is given to goruntime (or go's scheduler), its kind of illusion that the thread is still up (or online), other wise the process will be blocked*

Certainly, let's clarify your questions:

1. \*\***Thread Blocking and Process Blocking**\*\*:

In traditional operating systems, if a thread within a process is blocked (for example, waiting for I/O to complete), it can potentially block the entire process, preventing other threads within the same process from executing. This is because threads within the same process share the same memory space and resources, and blocking one thread can have a cascading effect on the entire process.

2. \*\***Yielding and Goroutine Execution**\*\*:

In GoLang, when we talk about yielding, we mean that a goroutine voluntarily gives up control to the Go runtime scheduler. The Go runtime scheduler efficiently schedules and switches between active goroutines, allowing them to run concurrently. When a goroutine encounters a point where it needs to wait for something (e.g., an I/O operation like a database query), it can yield control to the scheduler.

While a goroutine is yielding, the Go runtime scheduler can schedule other runnable goroutines to run on the available CPU cores. This is what allows for true parallelism and concurrency within a single GoLang process. It's not an illusion of a thread being "online" while blocked; instead, it's a way to ensure that the CPU is effectively utilized by allowing other non-blocking goroutines to execute in parallel.

Importantly, while a goroutine is yielding due to an I/O operation or other blocking event, the Go runtime scheduler can continue executing other runnable goroutines, ensuring that the entire process is not blocked. This concurrency and parallelism are key features of GoLang's goroutines and scheduler.

In summary, GoLang's concurrency model with goroutines and the scheduler is designed to efficiently manage blocked and non-blocked goroutines, allowing for high parallelism and responsiveness. This is different from traditional thread-based approaches, where blocking one thread can have more significant consequences on the entire process.

1. **now explain this same scenarion in case of a Java HTTP server**

**two incoming request at the same time with some I/O operations after 10lines and at 11th line there is some I/O call.**

Sure, let's go through the scenario of handling two incoming requests simultaneously in a Java HTTP server with some I/O operations.

1. \*\*Two Requests Simultaneously\*\*:

Two HTTP requests, Request A and Request B, arrive at the Java HTTP server at the same time.

2. \*\*Thread-Per-Request Model\*\*:

In many traditional Java HTTP server implementations, a common approach is to use a thread-per-request model. This means that a new thread is created for each incoming request to handle its processing.

3. \*\*Thread Creation\*\*:

When Request A comes in, a new thread is created to handle it. Simultaneously, when Request B arrives, another thread is created to handle its processing. Each thread operates independently.

4. \*\*I/O Operations\*\*:

As you mentioned, after 10 lines in the code of each request, there are I/O operations. These I/O operations could be reading from files, making network requests, or database queries.

5. \*\*Blocking Behavior\*\*:

When an I/O operation occurs, the thread that is currently executing the request is blocked while waiting for the I/O operation to complete. This blocking behavior means that if an I/O operation is slow, the thread will be held up, unable to handle other requests during that time.

6. \*\*Concurrency and Blocking Impact\*\*:

If the I/O operations after the 10th line are quick, then the server can handle both requests efficiently. However, if the I/O operations are slow or if they involve waiting for external resources, it can impact the server's ability to handle other incoming requests concurrently.

7. \*\*Additional I/O at 11th Line\*\*:

If there's another I/O operation after the 11th line, this operation will further block the thread currently handling the request.

8. \*\*Resource Management\*\*:

In a thread-per-request model, it's important to manage the number of threads carefully. Creating too many threads can lead to excessive memory usage and resource contention. Some server implementations use thread pools to manage and reuse threads, which can help mitigate resource-related issues.

9. \*\*Response Handling\*\*:

Once the I/O operations are complete and the responses are ready, the respective threads can send the responses back to the clients.

In this scenario, Java's thread-per-request model provides concurrency by using multiple threads to handle multiple requests concurrently. However, the blocking nature of I/O operations can impact the overall efficiency and responsiveness of the server. Proper thread management, including using thread pools, can help mitigate some of these issues, but the potential for resource contention and blocking remains a consideration.

It's worth noting that modern Java frameworks and libraries, such as those built on top of the Reactor or Netty libraries, offer more advanced non-blocking and event-driven approaches to handle I/O operations efficiently, similar to what Node.js provides. These frameworks can help mitigate some of the challenges associated with traditional thread-per-request models.

**9)i still dont understand one thing here...**

**since both the request came at the same time and two threads are created. say T1 and T2**

**assume we have multi-core CPU, so these threads will be processed parallely.**

**now suppose T1 hits some blocking I/o operation, if it waits means T1 is in blocked state right?**

**coming to Operating System Concept.. if a thread spawned from some process is blocked then all the other threads are blocked right?**

**so in this case since T1 is blocked/or waiting for some I/o operation to complete... does it mean T2 is also blocked because of this.. assuming T2's code dont contain any blocking I/o.**

**after T1 gets into readystate again, t1 and t2 again starts excuting parallely. Am i correct?**

You're almost correct, but there's a key distinction to be made between threads blocking at the application level and threads blocking at the operating system level.

In a multi-threaded environment like Java's thread-per-request model:

1. \*\*Application-Level Blocking\*\*:

When a thread (T1) encounters a blocking I/O operation, it becomes blocked at the application level. This means that within the application's code, that specific thread is waiting for the I/O operation to complete. During this time, other threads (like T2) can continue executing their code, even if they're part of the same process.

2. \*\*Operating System-Level Blocking\*\*:

At the operating system level, threads are typically managed individually. If one thread (T1) within a process becomes blocked at the application level, it does not necessarily block all other threads (like T2) within the same process. The operating system's thread scheduler can continue allocating CPU time to other runnable threads in the same process.

So, in your scenario:

- When T1 hits a blocking I/O operation, it will be waiting for that operation to complete at the application level. However, this does not automatically block other threads, such as T2, at the operating system level.

- T2 can continue to execute as long as it's not waiting for any blocking I/O operations or synchronization mechanisms that could cause it to block.

- Once the blocking I/O operation in T1 completes, T1 becomes runnable again, and the thread scheduler can allocate CPU time to it.

In summary, application-level blocking doesn't necessarily imply operating system-level blocking for all threads within the same process. Other threads can continue running, even if one thread is waiting for a blocking operation to complete at the application level. This separation allows for concurrency and parallelism within the same process.