***Question 2: Asynchronous I/O***

Libuv is a C library that provides support for asynchronous I/O through the use of event loops and is used in Node.js to abstract non-blocking I/O operations of the operating system allowing for efficient and user-friendly handling of asynchronous I/O operations through event-driven architecture.

When a Node.js application requires an asynchronous operation such as reading a file or making a network request, libuv uses the epoll system call to monitor the file descriptor associated with that operation. When this asynchronous operation completes libuv generates an event and invokes the appropriate event handler registered by the application. With this event-driven architecture events are created when I/O operations are complete rather than a blocking I/O model where the application stops and must wait for I/O operations to complete. One of the major benefits of this model is that it can handle multiple concurrent connections without blocking the event loop, allowing for Node.js applications to handle many concurrent connections efficiently and gives applications high performance and scalability. A more in depth look into this system is explored below.

When an application using the Libuv library starts, libuv initializes an event loop that manages all asynchronous I/O operations, when an asynchronous I/O operation is initiated libuv registers the corresponding file descriptor with epoll using system call epoll\_ctl(). Libuv then waits for events using the epoll\_wait() system call, which blocks the event loop until an event occurs on the registered file descriptors. When an event such as data becoming available to read happens, epoll notifies libuv by returning from the epoll\_wait() system call. Libuv then generates an event associated with the file descriptor indicating the type of event that has occurred. Libuv then calls the appropriate event handler registered by the application that is responsible for the event for handling and executing the appropriate logic in the application. After the event handler completes the event loop continues to wait for more events by going back to the epoll\_wait() system call and the process cycle is repeated.

Libuv uses a single-threaded model where all event call backs are executed in the same thread that originally initiated the event loop. The event loop in libuv is designed to handle multiple events and call backs concurrently, making it highly efficient. Libuv also provides a built-in thread pool feature that allows applications to offload CPU-bound or blocking tasks to separate threads, preventing blocking of the event loop and ensuring the responsiveness of the application. The thread pool in libuv is implemented as a separate pool of worker threads that can be used to perform tasks in parallel with the main event loop, this allows for concurrent execution of CPU-bound or blocking tasks while the event loop continues to process other events and call backs in the same thread of execution. This design helps prevent performance bottlenecks and ensures that the application remains responsive even whilst dealing with time consuming tasks.