1. This version of **getenv** achieve **thread-safe** through the use of thread-specific data and synchronization mechanisms.

   (1) The **pthread_once()** function is used to ensure that the thread_init function is executed only once during the entire lifetime of the program. This initialization is essential for creating the thread-specific data key (**pthread_key_t key**) that will be used to store thread-specific environment variable data.
   (2) The **pthread_key_t** key is a key for the thread-specific data. It is created once for the entire program using **pthread_once**() and is associated with a cleanup function (**free** in this case) that will be called when a thread terminates. The thread-specific data is used to store a pointer to the thread's environment variable buffer.
   (3) The use of **pthread_mutex_t env_mutex** provides a mutex (lock) to protect access to shared data. The mutex ensures that only one thread can access or modify the environment variable data at a time, preventing race conditions and ensuring thread safety.
   (4) When a thread calls **getenv** and needs to allocate memory for its environment variable buffer, it first checks if the thread-specific data (**envbuf**) is already set. If not, it allocates memory for the buffer using **malloc** and associates it with the thread-specific key. The mutex (**env_mutex**) is used to protect the entire process of checking and allocating memory. This ensures that only one thread at a time can perform these operations.
   (5) The thread uses the thread-specific data (**envbuf**) to store its own copy of the environment variable value. The mutex is used to protect the iteration over the global **environ** array, preventing other threads from accessing or modifying the environment variables simultaneously.

   The code achieves thread safety by using thread-specific data (**pthread_key_t**) to maintain separate environment variable buffers for each thread. Synchronization is achieved through the use of a mutex (**pthread_mutex_t**) to protect access to shared data structures, ensuring that only one thread at a time can perform critical operations, such as memory allocation and accessing/modifying global environment variables. The use of **pthread_once** ensures that the thread-specific data key is initialized only once, providing a reliable and thread-safe implementation of **getenv** in a multi-threaded environment.

2. No, through adding temporarily blocking signals at the beginning of the function and then restoring the previous signal mask before the function returns will make this getenv function async-signal safety, we cannot use **malloc(3)** in an **async-signal safety** function since two situation will occur because **malloc(3)** may not be implemented in signal-safe fashion.
   (1) The process will deadlock inside the signal handler, because `malloc` will be unable to acquire the heap lock.

   (2) The process will corrupt its heap, because malloc(3) *does* acquire the lock (or doesn't think it needs it), then proceeds to render the heap inconsistent, leading to a later crash.

3. In **thread_init()**, the code uses **pthread_key_create()** , this function will enroll a cleanup function, in this case, which is **free()**. This will free the **envbuf** after the thread terminates.

   However, in **getenv**, if a thread-specific data is NULL, it will call **malloc(3)** to allocate the memory. This means that the thread-specific data will point to some characters in **environ** (strncpy(envbuf, &environ[i][len + 1], MAXSTRINGSZ -1)), and these characters will be free in the termination period of each threads.

   In the end of a thread, the function **free()** which is enrolled by **pthread_key_create()** will

be called, and try to free the memory which is allocated by **malloc(3)**. This will cause a segmentation fault because the memory points to some characters in **environ**, but these characters in **environ** is not allocated by **malloc(3)** but a static memory provided by system in the beginning of program.