In this project, we implement a memory management API. We create a shared memory, in order to manage memory requests coming from multiple threads and grant or decline requests based on free space available. We manage all mutexes and semaphores for shared memory access.

- In the main, firstly, thread ID array is created. Threads are numbered starting from 0 to NUM_THREADS.
- Then we call init() function to initialize threads semaphores and memory array (that initially all cells are filled with zeros), and we start the server, that will use server_function().
- After that, we create threads with using thread ID array. Threads are created in a loop that iterate number of threads time and we create threads, that will use thread_function, for each unique thread ID.
- Thread_function(void * id) run as a thread. Firstly we lock mutex, and we generate random number, that refers to requested memory size. Then we create integer pointer, that points to id, to be able to reach which id we are using. Next, we call my_malloc(int id, int size) to allocate new node to the shared queue with unique thread id. Threads block on their semaphore until the memory request is handled by the server thread. Once thread is unblocked by server, if thread_message[id] is -1, this means that there is not enough memory for this thread, program displays appropriate message, otherwise, we fill corresponding memory space with unique thread ids, and after that process, we increase usedMemory by initially requestedSize. Finally, we unlock the mutex, and return 0. (Additionally, usedMemory will become start location of another thread actually).
- This thread will run in a loop until all threads are done. It will check the queue for the requests and depending on the memory size it either grants or declines them (from assignment pdf). While queue is not empty, firstly we assign a node pointer that points to front node of the queue, therefore, even if we pop the front node, we will not loose any information because we keep it with pointer. Then if taken node's (that is reached by pointer) size exceeds the available size (MEMORY_SIZE - usedMemory), we put -1 to thread_message array at index idth. Otherwise, we put start location of the memory that will be allocated to the requesting thread. When we finish the queue, we check that all thread requests are handled or not, if handled, we break loops and return 0.

- After these, we will print out the all memory with dump_memory(). This function basically does, go over each element of the memory array and print all elements. Since we fill the memory array with the thread ids, we are able to see which part is allocated with which thread.

- Finally, we print whole thread_message array. This array was filled by server_function, and it indicates that starting location of the each threads by order.
  ([starting position 0th thread] [starting position 1st thread]…..[starting position nth thread]). If enough space was not found as it was said before, we put [-1] instead of "start position".