Locking Mechanism: I used mutex locks to ensure mutual exclusion in this assignment.

Implementation: Each HeapManager object has a mutex which is used to lock critical sections. The mutex is locked at the beginning of each critical operation (myMalloc, myFree, and print) and unlocked immediately after the operation is completed. This ensures that no two threads can perform conflicting operations on the heap simultaneously.

Pseudocode:

```
class HeapManager:
   mutex: Mutex

   function myMalloc(id, size):
      mutex.lock()
      // Allocation logic
      mutex.unlock()

   function myFree(id, start):
      mutex.lock()
      // Deallocation logic
      mutex.unlock()

   function print():
      mutex.lock()
      // Print logic
      mutex.unlock()
```

Mutual Exclusion: When a thread locks the mutex, no other thread can enter a critical section until the mutex is unlocked. This guarantees that operations like allocation & deallocation are not interleaved.

Consistency: By ensuring that the heap is only modified in a locked state, the its consistency is maintained, even when multiple threads are making simultaneous requests.

Deadlock Avoidance: Placement of lock and unlock calls prevents deadlock. Each function acquires the lock at the beginning and releases it at the end, ensuring that locks are not held indefinitely.

Ceren Dinç – 28220
CS307 PA4 – Heap Management

Overview: The implemented HeapManager class is responsible for managing a heap represented as a linked list of HeapBlock structures. Each HeapBlock contains an id to identify the allocating thread (or -1 if the block is free), the size of the block, and the index representing the start address. The next pointer links to the subsequent block in the heap.

myMalloc: This function searches for a suitable free block to allocate. If a block of the exact size is found, it is marked as allocated. Otherwise, a larger block is split into two, allocating the required size and leaving the remainder as a free block. The heap is traversed to find a suitable block for allocation. During this process, no other thread can modify the heap, ensuring that the allocation logic operates on a consistent view of the heap.

myFree: This function frees an allocated block. It also performs coalescing, merging adjacent free blocks into a single larger free block to avoid fragmentation. The deallocation and coalescing logic require accurate knowledge of the neighboring blocks. The mutex lock ensures that the heap remains unchanged by other threads, allowing reliable coalescing of free blocks.

print: This function iteratively traverses the linked list and prints the status of each block, showing whether it's allocated or free, along with its size and start index.