## Overview

### *Mechanism*

For the thread_safe version of malloc function. The mechanism for my implementation is keep a global free list shared by all threads. I use two global mutex locks as my synchronization solution to prevent the inconsistent updates to my block list or conflicting malloc or free to my block list. I identify different categories of critical sections as following.

### *Possible critical sections*

(1) First, thread_safe use Best-Fit strategy to find the proper block for malloc. However, here has the possiblity of race condition. Because previous threads are very likely to malloc the same blcok and this block is no longer proper any more. So the order threads call malloc can result in different malloc results here. To prevent this race conditon, I use a mutex lock here. When I find the proper block, I lock. Then I check if this block is still "free", if so. I mark it malloced and unlock the mutex lock. If this block is not "free", means other threads have malloced it before. Then I unlock the block to avoid dead lock and then call my ts_malloc once again to figure out what the current thread should do.

(2) Second critical section is that when I search the blcok without finding the proper block and I need to extend the heap to get the block. This is the second critical section I identify. Because previous threads are likely to extend the heap and current thread is no longer at the tail of my list. So I add a lock at the beginning when I need to extend the heap. Then I check if current thread is at the tail of list. If so, just extend the heap. If not, I do not need to start again, because I have checked that existing blocks are not suitable and new malloc blocks has been used by other threads. I just move my current pointer forward to the current tail of free list then extend the heap. After extending the heap, I unlock my mutex lock.

(3) Third critical section I identify is that when my head is NULL, meaning that my block list is empty. I need to extend the heap to get the head block of my list. However, here has a serious critical section. When multiple threads arrive concurrently, they all check that the list is empty and create new block and assign the head pointer to the "head" they think it is. Without synchronization, our list can have chaos head block and pointer. So, I add the mutex lock at the beginning when I need to create the head block for my list. Then I check the if the head exists, if it does not. I just create new head and assign the head pointer to this block then unlock my mutex lock. If I have head in my block list, I just unlock the mutex lock and then call my ts_malloc function once again.

(4) The last critical section I identify is more sophisticated, I consider the possiblity that multiple frees and mallocs can happen at the same. What I identify is that mallloc and free should have clear execution order. A malloc needs to finish free and combining nearby blocks before it is avaliable for malloc calls. So I add the lock at the beginning of free, I unlock the mutex lock after I combine the nearby free blocks. Here I use the same lock as extending heap and creating head because these conditions need to have a clear order in order to avoid race conditions.

### *Summary*

In my implementation, I create two global mutex_lock to prevent race conditions. First mutex_lock is used for locking the section when I find the proper block and want to return this block for one threads. Second mutex_lock is used for extend_heap, creating_head and free functions. These mechanism need to have clear execution order, so I use the global mutex to identify their execution orders.