Design strategy

There are two versions of *malloc* policies implemented in this project: Lock and NoLock. The reason that we have this project, which is same in terms of functionality compared to project1, is that we enable multiple threads at the same time so that the overall running time would be reduced compared to a single thread which is default.

Thus, problems are raised: inter-thread synchronization. And we have two solutions.

For the locked version, the strategy is to use *pthread_mutex_lock* to ensure that only one thread at a time could modify the free list, which is the data structure we use. To be more detailed, everytime there is a modification to the free list, we put that chunk of code inside two *lock()* and *unlock()* functions, known as a critical section. By doing so, we can ensure only one thread at a time could gain the lock and others are kept waiting. The two critical sections I have in my code is *malloc()* and *free()*.

For the No-locked version, I utilize a strategy called *ThreadLocalStorage*, known as TLS, which makes *static* and *global* variables local to each thread. By doing this, each thread would then have its own free list and it is safe to modify it.

Policy comparisons

The locked version is easy to implement but relatively useless in reality. As only one thread each time could gain the lock, we basically make the process single-threaded or serialized. Thus we lost the meaning and benefits of allocating multiple threads.

The No-locked version, on the other hand, utilizes the benefits of multiple threads and is simple to implement, too: making static free list *head* element as *__thread head.* However, this method is inefficient in space efficiency: each thread keeps its own free list. This imposes heavier load to scheduler. Taking pointer lookups and modifications as examples, a scheduler would need to do so during task switches. And therefore at the same time, slows down the process.