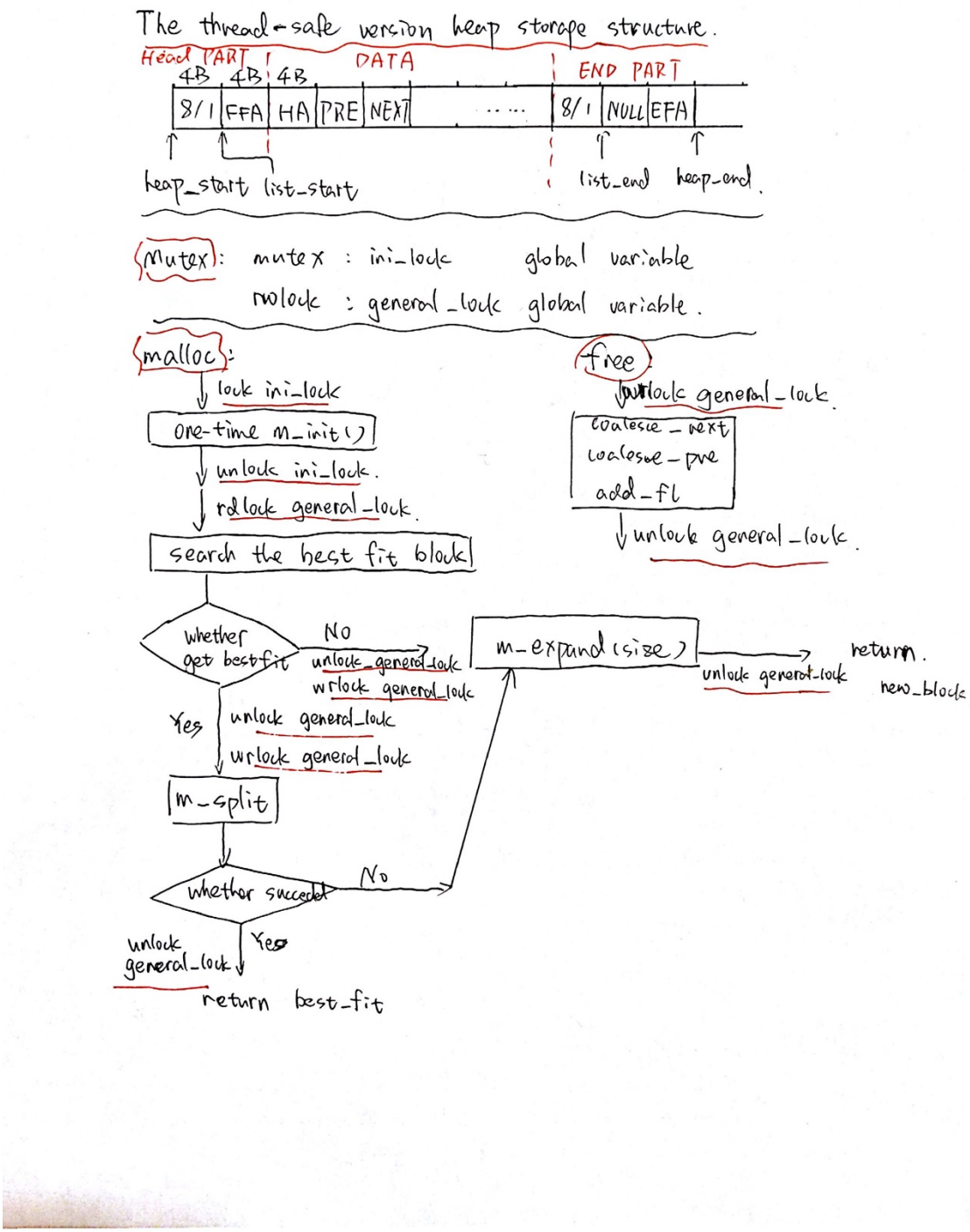
Thread-safe dynamic memory allocator report

SHENGXIN QIAN

1. **Compile requirement**
   The test case makefile must add –m32 at CFLAGS part

2. **The general heap model**



The structure of the thread safe dynamic memory allocator is not multiple ascending free linked list but single free linked list. The single linked list is very easy to be thread safe. It does not need multiple synchronization primitives which are necessary when we want to make multiple linked lists thread safe.

When I want to use multiple primitives, the most difficult part is avoiding dead lock and the performance lost when thread locks and unlocks multiple synchronization primitives. When one thread wants to change the free block size and move it to another free list, this thread should lock more than one linked list. When one thread tries to coalesce the front and the back free block, it may need to lock three different linked lists. Even through we know lock part of free list is better than lock the whole linked list, we should be careful about this kind of synchronization strategy. Multiple primitives lock and unlock operation would cause severe drop of performance and make code much more complicated.

The most often operation of allocator is searching the best fit free block. The modification to single block would not cost too much time. That is why we need to use read lock to make the search operate concurrently in different threads and block modification in other threads. If one thread finishes its search, it should unlock the read lock and acquire the write lock to make sure each modification is exclusive. This synchronization strategy would make each thread has less chance to be blocked.

The heap structure is shown in the previous picture. It only contains one free linked list. The free or used block structure is the same as single thread version. The ini_lock is a mutex which is just for the program to initialize the head part and end part of the whole dynamic memory block. When the initialization is over, it will not need to require this mutex and call m_init() again. The general_lock is a read_write lock for the whole free linked list. When we want to search the linked list, we need to required read lock which would accept search operation and block any modification to this list. When we get the required block, we should release the read lock and try to acquire write lock if we need to use any sub function which may modify the free linked list. Of course, the free block we choose may be modified when we just release read lock because this is not atomic. So, we need to recheck the condition and just expand the whole memory if the condition is not meet previous requirement. The clear structure of how program work is showed in previous picture.

3. result

```
parallels@ubuntu:~/ece650/Malloc/Paralleltest/test$ ls
Makefile  thread_test  thread_test.c  thread_test_malloc_free  thread_test_malloc_free.c
parallels@ubuntu:~/ece650/Malloc/Paralleltest/test$ ./thread_test
No overlapping allocated regions found!
Test passed
parallels@ubuntu:~/ece650/Malloc/Paralleltest/test$ ./thread_test_malloc_free
No overlapping allocated regions found!
Test passed
parallels@ubuntu:~/ece650/Malloc/Paralleltest/test$
```

The thread_test could run with 10 threads and 5000 items in several seconds. My program could also pass thread_test_malloc_free test with 4 threads and 200 items in several seconds.