HW1(b) Thread Safe

Wanning Jiang (wj43) (wjiang@cs.duke.edu)

How to test

I include a Makefile in the my_thread folder. The Makefile will build and load my_malloc.c source code to a shared dynamic library as libmymalloc.so, and then build the executable file with the test codes.

Noted that to test the code, you need to change the path macro in Makefile to be the actual path of my_thread folder on your machine.

Objective

Based on HW1(a), this part of the homework allows multiple threads making *malloc()* and *free()* requests simultaneously. The goal is to ensure correctness and minimise the average waiting time of the threads, which means we hope to achieve the smallest granularity.

Implementation

1. Mutex Data Structure
   For every piece of data that will be shared between threads, we set a separate lock. Specifically, we have the following mutexes:
   - For each doubly linked list in the array, we set a mutex for it, so we have the equal numbers of mutexes to the array size. The variables are *arr_mutex[ARRAY_SIZE]*.
   - At a certain time, only one thread should be allowed to have access to the *malloc_init()* subprocess. Thus we set a lock for the initialisation process called *init_mutex*.
   - At a certain time, only one thread should be allowed to call the *sbrk()* library function in the extend_heap() sub routine. Thus we set a lock for the sub process called *sbrk_mutex*.

2. Initialise the heap
   Multiple threads may enter the heap initialisation routine, but only one thread will acquire the *init_mutex* so the others will wait outside. The very thread in the *init_mutex* sets the *is_init* bool variable *true*. Therefore, after the thread releases the mutex and another thread enters, the latter thread will check the *is_init* and find it true. As a result, it releases the lock *init_mutex* and returns immediately. This mechanism ensures only the initialisation process is conducted only one time during all the conditions, and the whole process is atomic.

3. Memory allocation
   - Best fit

   To find the most probable block to alloc we need to traverse the whole free list array. During the process we will lock the corresponding list. After deleting the block from the list, we unlock it.

   - Extend heap

   The issue is that if we cannot find a suitable block, we will extend heap. Similar with initialising the heap, we set a mutex to ensure only one thread can extend the heap.

   - Split

After finding the best fit (or extending the heap), we need to allocate the part to the block of necessary size and restore the redundant part to the block back to the free list. When add the block, we will lock the destination mutex in the mutex array.

4. Memory free
The function *free()* is composed of two steps: coalesce and add to free list.

- Coalesce

To coalesce, we will inspect the neighbour blocks of the block to be free. During inspection we will lock the corresponding free lists. The free blocks must be deleted from the free list before coalescing. Thus we avoid other threads to access the same blocks.

- Add to free list

We then add the coalesced free block to the proper entry of the free list array. The *add_node()* process will lock the corresponding list. Noted here before adding to the free list, we set the alloc bit of the blocks to be zero, which means it is allocated. Therefore no other threads can access the blocks. (A block can access the block in two ways. One is to access the free list, and the other is to access the neighbour blocks. If we set the alloc bit to be true, then the second way is disabled. In addition, the block is now not in the free list, so we successfully isolate the block from access for any other threads.)

Important principles

1. Before entering *add_node()* function, which will add the block into the free list, the block must be set as allocated, therefore the block won't be accessed during the coalesce process.
2. Before entering *delete_node()* function, the block should be free. Otherwise, the block might be modified by other threads due to contact switch. We use assertion to detect this bug.
3. Whenever changing the alloc bit in whatever direction, the thread must lock the corresponding *arr_mutex*.
4. Only blocks with free *alloc* bit might be modified.

Result & Analysis

Testing environment: 64-bit ubuntu-12.04
Test codes:

- the original thread_test.c
- the updated thread_test_malloc_free.c with NUM_THREADS = 1000, NUM_ITEM = 1000
- the test written by ourselves