ECE 650 Project 2: Thread-Safe Malloc Report

By Junqi Sun js895

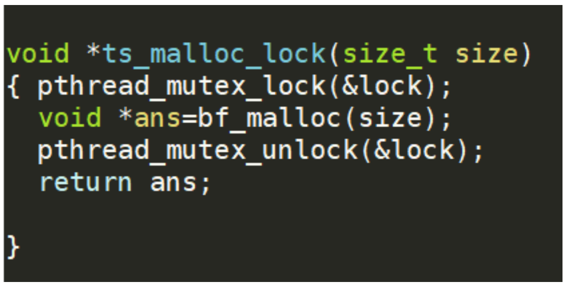
1. Design of my Thread-Safe Malloc and Free

The functions of this project could be listed as below:

1. First find function
2. Best find function (which need First find function to initialize)
3. Alloc_new function (implement sbrk)
4. Alloc_new_nolock function (TLS version)
5. Merge function
6. Addfreelist function (called in free function)
7. Addfreelist_nolock function (TLS version)
8. Split function
9. Best find malloc function (prj1 version)
10. Best find free function (prj1 version)
11. Thread safe malloc function (lock)
12. Thread safe free function (lock)
13. Thread safe malloc no lock (TLS version)
14. Thread safe free no lock (TLS version)

In order to simplify the operation to memory space, I re-design the data structure from a full linked list (link every block of memory no matter it is used or freed) to a single free list with sequence of block address's value. I use a global variable to behave as head pointer to the free list, which should be adjust to __thread type (Thread local storage, TLS) to perform a no lock version.

For the Thread safe malloc function with lock version, I just keep everything inside best find malloc function (prj1 version) as well as every function it calls unchanged. Then I write a new function ts_malloc_lock, in which declare and initialize a new variable with the return value from bf_malloc. The pthread_mutex_lock and unlock are located in front and behind this initialize process, which means the whole bf_malloc function is the critical section, while the final return value allow concurrency. The picture below shows the layout:



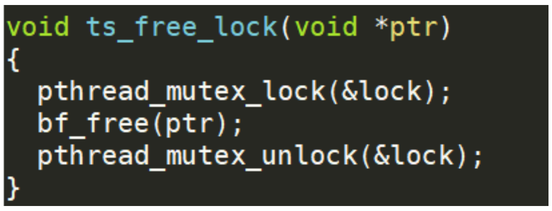
```
void *ts_malloc_lock(size_t size)
{ pthread_mutex_lock(&lock);
  void *ans=bf_malloc(size);
  pthread_mutex_unlock(&lock);
  return ans;

}
```

Figure 1.1 layout of ts_malloc_lock

In this method, since any code that deals with global variable (threads share the same global variable plus they could change its value) could have race condition, so I just block the whole malloc function. By doing this, each thread could only access the critical section sequentially, and unlock it by reaching the pthread_mutex_unlock that allows other thread to access it. Except for malloc function itself, the function it calls such as best find, alloc_new will also be included in critical section.

For ts_free_lock function, my method is almost the same — I encapsulate it by a pthread_mutex_lock and unlock, every implement inside just keep the same. Its layout is shown as figure below:

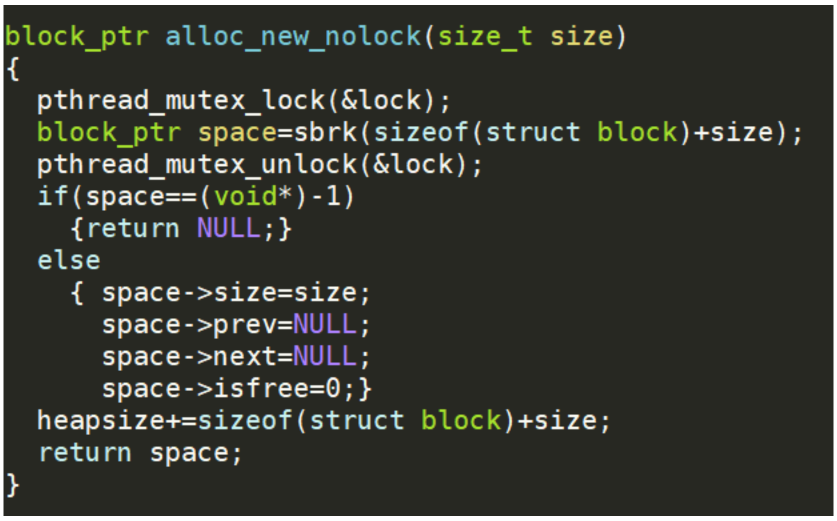


```
void ts_free_lock(void *ptr)
{
  pthread_mutex_lock(&lock);
  bf_free(ptr);
  pthread_mutex_unlock(&lock);
}
```

Figure 1.2 layout of ts_free_lock

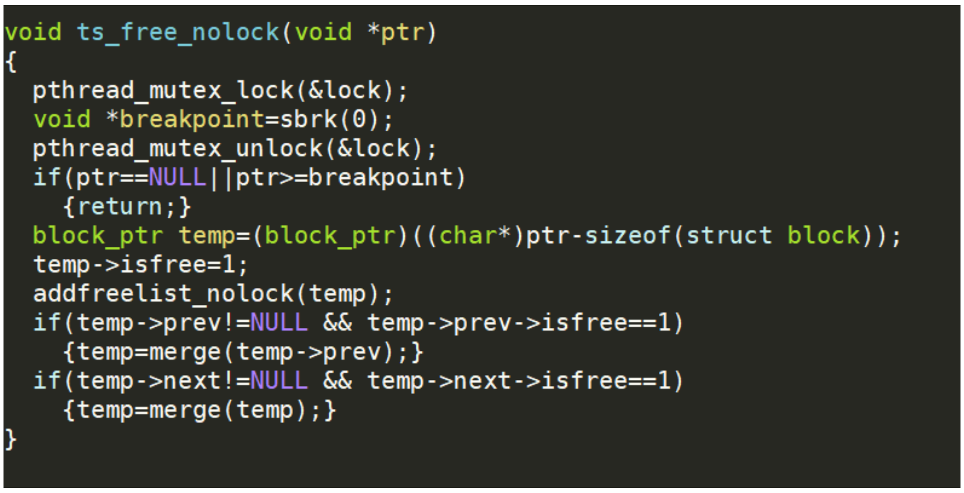
For the Thread safe malloc function with no lock version, however, I use the method called thread local storage which allows every thread has its own process buffer thus there will be no address conflict or space overlapping. In order to implement it, I declare the head pointer to __thread + typeof block head_nolock and exchange every function with parameter or accessing to original head pointer such as Alloc_new function, Addfreelist function and bf_malloc itself into new version with head_nolock. Except for this, I also use lock to encapsulate all sbrk call to ensure the memory allocation won't overlap.

In no lock version of malloc, every function besides the sbrk part (because it's not a thread safe function, behave as critical section in this case) could run concurrently, since the TLS guarantee that the changing value of head_nolock pointer won't go wrong. The method is the same when implement the ts_free_no lock function. Code of alloc_new_nolock function and ts_free_nolock are shows as figures below:



```
block_ptr alloc_new_nolock(size_t size)
{
  pthread_mutex_lock(&lock);
  block_ptr space=sbrk(sizeof(struct block)+size);
  pthread_mutex_unlock(&lock);
  if(space==(void*)-1)
    {return NULL;}
  else
    { space->size=size;
      space->prev=NULL;
      space->next=NULL;
      space->isfree=0;}
  heapsize+=sizeof(struct block)+size;
  return space;
}
```

Figure 1.3 layout of alloc_new_nolock

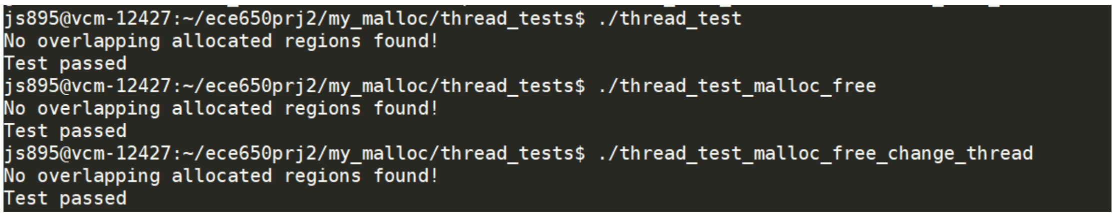


```c
void ts_free_nolock(void *ptr)
{
  pthread_mutex_lock(&lock);
  void *breakpoint=sbrk(0);
  pthread_mutex_unlock(&lock);
  if(ptr==NULL||ptr>=breakpoint)
    {return;}
  block_ptr temp=(block_ptr)((char*)ptr-sizeof(struct block));
  temp->isfree=1;
  addfreelist_nolock(temp);
  if(temp->prev!=NULL && temp->prev->isfree==1)
    {temp=merge(temp->prev);}
  if(temp->next!=NULL && temp->next->isfree==1)
    {temp=merge(temp);}
}
```

Figure 1.4 layout of ts_free_nolock

2. Experiment Result

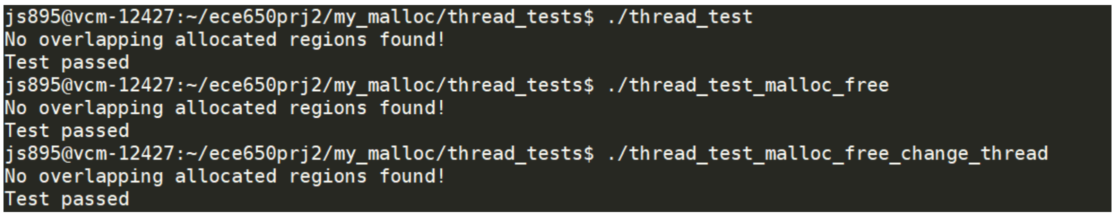
The first three tests' outputs of ts_malloc_lock and ts_free_lock are shown below:



```
js895@vcm-12427:~/ece650prj2/my_malloc/thread_tests$ ./thread_test
No overlapping allocated regions found!
Test passed
js895@vcm-12427:~/ece650prj2/my_malloc/thread_tests$ ./thread_test_malloc_free
No overlapping allocated regions found!
Test passed
js895@vcm-12427:~/ece650prj2/my_malloc/thread_tests$ ./thread_test_malloc_free_change_thread
No overlapping allocated regions found!
Test passed
```

Figure 2.1 result of the first three tests for ts_malloc_lock and ts_free_lock

The first three test's outputs of ts_malloc_nolock and ts_free_nolock are shown below:



```
js895@vcm-12427:~/ece650prj2/my_malloc/thread_tests$ ./thread_test
No overlapping allocated regions found!
Test passed
js895@vcm-12427:~/ece650prj2/my_malloc/thread_tests$ ./thread_test_malloc_free
No overlapping allocated regions found!
Test passed
js895@vcm-12427:~/ece650prj2/my_malloc/thread_tests$ ./thread_test_malloc_free_change_thread
No overlapping allocated regions found!
Test passed
```

Figure 2.2 result of the first three tests for ts_malloc_nolock and ts_free_nolock

For measurement tests:

	Thread safe lock version		Thread safe non-lock version	
Testing number	Execution time /second	Data segment size /byte	Execution time /second	Data segment size /byte
1	1.5604	43673088	0.2728	44470304
2	1.4746	44340608	0.2439	44529632
3	1.9482	43297568	0.2756	44190432
4	1.6108	44123872	0.3038	43501952
5	1.0287	45397664	0.2194	44979232
6	1.8570	43642560	0.2476	43811520
7	1.4719	43887776	0.3060	43691200
8	1.8461	43307264	0.3398	44344256
9	1.5794	43292736	0.2391	44740096
10	1.4393	44321344	0.2995	43505344

3. Result Analysis

Based on data shown above, It is clear to draw a conclusion that thread safe lock version will definitely cost more time than no lock version. The reason for its performance declination is that I almost regard the entire malloc and free function as critical section that could only process thread by thread, while the entire no lock version's code could operate concurrently (except for sbrk).

When referring to data segment size, I could say that two methods' data segment usages are almost the same. To be precise, the no lock version will use slightly more data space than lock version (In my test, six out of ten cases obey this rule). The differences could be attributed to the additional allocated space for __thread variables (thread local storage for each thread) and each thread will have their memory space for head_nolock pointer.

To draw a conclusion, the no lock version will run faster than lock version, while lock version will use less memory space than no lock version.