ECE_650 Project2: Thread-Safe Malloc and Free

2025/1/30-31

Sienna Zheng (NetID: sz318)

Environment:

- C
- Linux VM

Introduction:

- Based on codes of project1.
- Implement two different thread-safe versions of `malloc()` and `free()` using the **best fit allocation policy**.

  Version 1: Use **lock-based synchronization** to prevent race conditions. Functions:

  - `void *ts_malloc_lock(size_t size);`
  - `void ts_free_lock(void *ptr);`

  Version 2: Implement without using **locks or semaphores**, except for acquiring a lock **only** when calling `sbrk()`. Functions:

  - `void *ts_malloc_nolock(size_t size);`
  - `void ts_free_nolock(void *ptr);`

Thread-Safe Model

- **Locking Version (`ts_malloc_lock` / `ts_free_lock`)**
  - Uses a global mutex (`pthread_mutex_t`) to ensure only one thread accesses the memory allocator at a time.
  - Prevents race conditions but may introduce contention in multi-threaded environments.
- **Non-Locking Version (`ts_malloc_nolock` / `ts_free_nolock`)**
  - Utilizes **thread-local storage (`__thread`)** to maintain separate free lists for each thread.
  - Avoids locks except when using `sbrk()`, which is inherently **not thread-safe**.
  - Reduces contention but may lead to memory fragmentation since memory cannot be shared across threads.

Performance Comparison

1. Experimental Results

Metric	Non-Locking Version (`ts_malloc_nolock`)	Locking Version (`ts_malloc_lock`)
Execution Time	**0.098750 seconds**	**0.108753 seconds**
Data Segment Size	**42,281,160 bytes**	**43,265,248 bytes**

2. Observations and Analysis

1. **Execution Time**
   - The **non-locking version** runs **~9.2% faster** than the locking version.
   - This confirms that avoiding locks reduces contention and improves performance in multi-threaded scenarios.
2. **Memory Usage**
   - The **locking version has slightly higher memory usage (~2.3% more)**, likely due to global synchronization and less frequent thread-local fragmentation.

3. Conclusion

- **Performance**: The **non-locking version** is faster due to reduced synchronization overhead.
- **Memory Efficiency**: The **locking version** uses slightly more memory but may exhibit better memory reuse in the long run.
- Trade-offs:
  - **Use non-locking** for better performance in high-thread-count applications.
  - **Use locking** if memory efficiency and reuse are a priority.

Appendix: Code

my_malloc.h:

```c
#include <unistd.h>  // for sbrk()
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define META_SIZE sizeof(block_meta)

typedef struct _block_meta {
  size_t size;
  struct _block_meta *next, *prev;
} block_meta;

/**
  * Thread-safe by using lock around malloc and free
  */
```

```c
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

// head and tail ptr of free_list
block_meta *head = NULL;
block_meta *tail = NULL;

// for test
void print_free_list() {
  block_meta *curr = head;
  printf("Free list: ");
  while (curr) {
    printf("[%p: size=%zu] -> ", curr, curr->size);
    curr = curr->next;
  }
  printf("NULL\n");
}

// allocate the whole free block by deleting it from the free_list
void allocate_block(block_meta *block) {
  // 1. only one block in free list
  if (head == block && tail == block) {
    head = NULL;
    tail = NULL;
  }
  // 2. block is head
  else if (head == block) {
    head = block->next;
    block->next->prev = NULL;
  }
  // 3. block is tail
  else if (tail == block) {
    tail = block->prev;
    block->prev->next = NULL;
  }
  // 4. block is in the mid part
  else {
    block->next->prev = block->prev;
    block->prev->next = block->next;
  }
    block->prev = NULL;
    block->next = NULL;
}

// split a free block into an allocated block followed by a free block
void split_block(block_meta *block, size_t size) {
  block_meta *new_block = (block_meta *)((char *)block + META_SIZE + size);
  // char offset
  new_block->size = block->size - size - META_SIZE;

  // 1. only one block in the free list
  if (head == block && tail == block) {
    head = new_block;
    tail = new_block;
    new_block->prev = NULL;
    new_block->next = NULL;
  }
```

```
  // 2. block is head
  else if (head == block) {
    if (block->next) block->next->prev = new_block;
    new_block->next = block->next;
    new_block->prev = NULL;
    head = new_block;
  }
  // 3. block is tail
  else if (tail == block) {
    if (block->prev) block->prev->next = new_block;
    new_block->prev = block->prev;
    new_block->next = NULL;
    tail = new_block;
  }
  // 4. block is in the mid part
  else {
    new_block->prev = block->prev;
    new_block->next = block->next;
    if (block->prev) block->prev->next = new_block;
    if (block->next) block->next->prev = new_block;
  }

  // allocated block
  block->size = size;
  block->prev = NULL;
  block->next = NULL;
}

// allocate space on heap. An allocated block always needs a meta
block_meta *request_space(size_t size) {
  void *requested = sbrk(size + META_SIZE);
  // allocation failure
  if (requested == (void *) -1) {
    fprintf(stderr, "Error: allocation failure");
    exit(EXIT_FAILURE);
  };

  block_meta *block = (block_meta *)requested;
  block->size = size;
  block->next = NULL;
  block->prev = NULL;

  //data_segment_size += size + META_SIZE;

  return block;
}

// only need to merge with direct prev and next free blocks
void merge_blocks(block_meta *block) {
  if (!head || !block) return;

  if (block->next && (char *)block->next == (char *)block + META_SIZE +
block->size) {
    block_meta * next_block = block->next;
    block->size += next_block->size + META_SIZE;
    // renew tail
```

```
126     if (next_block == tail) {
127       tail = block;
128     }
129     block->next = next_block->next;
130     // here block->next might be NULL
131     if (next_block->next) {
132       next_block->next->prev = block;
133     }
134     next_block->prev = NULL;
135     next_block->next = NULL;
136   }
137 
138   if (block->prev && (char *)block == (char *)(block->prev) + META_SIZE +
    block->prev->size) {
139     block_meta * prev_block = block->prev;
140     prev_block->size += block->size + META_SIZE;
141     if (block == tail) {
142       tail = prev_block;
143     }
144     prev_block->next = block->next;
145     if (block->next) block->next->prev = prev_block;
146     block->prev = NULL;
147     block->next = NULL;
148   }
149 }
150 
151 
152 void bf_free(void *ptr) {
153     if (!ptr) return;
154 
155     block_meta *block = (block_meta*)ptr - 1;   // meta address
156     // 1. head = null, empty free list
157     if (head == NULL) {
158       head = block;
159       tail = block;
160       block->prev = NULL;
161       block->next = NULL;
162     }
163     // 2. freed block before head, block becomes the new head
164     else if (block < head) {
165       block->next = head;
166       head->prev = block;
167       block->prev = NULL;
168       head = block;
169     }
170     // 3. freed block after tail
171     else if (block > tail) {
172       block->prev = tail;
173       tail->next = block;
174       block->next = NULL;
175       tail = block;
176     }
177     // 4. somewhere between head and tail
178     else {
179       block_meta *curr = head;
180       // find two free blocks right before and right after block
```

```
      block_meta *front = NULL, *back = NULL;
      while (curr) {
        if (curr < block) front = curr;
        if (curr > block) {
          back = curr;
          break;
        }
        curr = curr->next;
      }
      if (front -> next != back) {
        printf("front -> next != back\n");
      }
      block->prev = front;
      block->next = back;
      front->next = block;
      back->prev = block;
    }

    merge_blocks(block);
}


// find a sutable free block accoridng to bf strategy
block_meta * find_bf_block(size_t size) {
  block_meta * curr = head;
  block_meta * res = NULL;
  while (curr) {
    // a new meta is always needed for allocation
    if (curr->size >= size) {
      // find the smallest space bigger equal than size
      if (res == NULL || curr->size < res->size) {
        res = curr;
        // already found the best one
        if (curr->size == size) break;
      }
    }
    curr = curr->next;
  }
  return res;
}

// Best Fit malloc/free
void *bf_malloc(size_t size) {
  if (size == 0) return NULL;

  block_meta *block = find_bf_block(size);
  if (block == NULL) {   // no adequate free block, expand the heap
    block = request_space(size);
    // if (!block) return NULL; // heap expansion failure
  } else {  // adequate free block founded
    // 1. no need to split
    if (block->size <= size + META_SIZE) {
      allocate_block(block);
    }
    // 2. split the block
    else {
```

```
      split_block(block, size);
    }
  }
  return (void *)(block + 1);  // return the actual data's address by skipping block_meta
  // ptr addition: 1 refers to 1 ptr's size, aka 1 block_meta size.
}

//Thread Safe malloc/free: locking version
void *ts_malloc_lock(size_t size) {
  pthread_mutex_lock(&lock);
  void * ptr = bf_malloc(size);
  pthread_mutex_unlock(&lock);
  return ptr;
}

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

/**
  * Thread-safe using no lock:
  * __thread for free_list
  * lock only for sbrk()
  */
pthread_mutex_t sbrk_lock = PTHREAD_MUTEX_INITIALIZER;

// head and tail ptr of free_list
__thread block_meta *head_nolock = NULL;
__thread block_meta *tail_nolock = NULL;

// allocate the whole free block by deleting it from the free_list
void allocate_block_nolock(block_meta *block) {
  // 1. only one block in free list
  if (head_nolock == block && tail_nolock == block) {
    head_nolock = NULL;
    tail_nolock = NULL;
  }
  // 2. block is head
  else if (head_nolock == block) {
    head_nolock = block->next;
    block->next->prev = NULL;
  }
  // 3. block is tail
  else if (tail_nolock == block) {
    tail_nolock = block->prev;
    block->prev->next = NULL;
  }
  // 4. block is in the mid part
  else {
    block->next->prev = block->prev;
    block->prev->next = block->next;
  }
    block->prev = NULL;
```

```
292     block->next = NULL;
293 }
294 
295 // split a free block into an allocated block followed by a free block
296 void split_block_nolock(block_meta *block, size_t size) {
297   block_meta *new_block = (block_meta *)((char *)block + META_SIZE + size);
    // char offset
298   new_block->size = block->size - size - META_SIZE;
299 
300   // 1. only one block in the free list
301   if (head_nolock == block && tail_nolock == block) {
302     head_nolock = new_block;
303     tail_nolock = new_block;
304     new_block->prev = NULL;
305     new_block->next = NULL;
306   }
307   // 2. block is head
308   else if (head_nolock == block) {
309     if (block->next) block->next->prev = new_block;
310     new_block->next = block->next;
311     new_block->prev = NULL;
312     head_nolock = new_block;
313   }
314   // 3. block is tail
315   else if (tail_nolock == block) {
316     if (block->prev) block->prev->next = new_block;
317     new_block->prev = block->prev;
318     new_block->next = NULL;
319     tail_nolock = new_block;
320   }
321   // 4. block is in the mid part
322   else {
323     new_block->prev = block->prev;
324     new_block->next = block->next;
325     if (block->prev) block->prev->next = new_block;
326     if (block->next) block->next->prev = new_block;
327   }
328 
329   // allocated block
330   block->size = size;
331   block->prev = NULL;
332   block->next = NULL;
333 }
334 
335 // allocate space on heap. An allocated block always needs a meta
336 block_meta *request_space_nolock(size_t size) {
337   pthread_mutex_lock(&sbrk_lock);
338   void *requested = sbrk(size + META_SIZE);
339   pthread_mutex_unlock(&sbrk_lock);
340 
341   // allocation failure
342   if (requested == (void *) -1) {
343     fprintf(stderr, "Error: allocation failure");
344     exit(EXIT_FAILURE);
345   };
346 
```

```
  block_meta *block = (block_meta *)requested;
  block->size = size;
  block->next = NULL;
  block->prev = NULL;

  //data_segment_size += size + META_SIZE;

  return block;
}

// only need to merge with direct prev and next free blocks
void merge_blocks_nolock(block_meta *block) {
  if (!head_nolock || !block) return;

  if (block->next && (char *)block->next == (char *)block + META_SIZE + block->size) {
    block_meta * next_block = block->next;
    block->size += next_block->size + META_SIZE;
    // renew tail
    if (next_block == tail_nolock) {
      tail_nolock = block;
    }
    block->next = next_block->next;
    // here block->next might be NULL
    if (next_block->next) {
      next_block->next->prev = block;
    }
    next_block->prev = NULL;
    next_block->next = NULL;
  }

  if (block->prev && (char *)block == (char *)(block->prev) + META_SIZE + block->prev->size) {
    block_meta * prev_block = block->prev;
    prev_block->size += block->size + META_SIZE;
    if (block == tail_nolock) {
      tail_nolock = prev_block;
    }
    prev_block->next = block->next;
    if (block->next) block->next->prev = prev_block;
    block->prev = NULL;
    block->next = NULL;
  }
}


void bf_free_nolock(void *ptr) {
    if (!ptr) return;

    block_meta *block = (block_meta*)ptr - 1;   // meta address
    // 1. head = null, empty free list
    if (head_nolock == NULL) {
      head_nolock = block;
      tail_nolock = block;
      block->prev = NULL;
      block->next = NULL;
```

```
401     }
402     // 2. freed block before head, block becomes the new head
403     else if (block < head_nolock) {
404       block->next = head_nolock;
405       head_nolock->prev = block;
406       block->prev = NULL;
407       head_nolock = block;
408     }
409     // 3. freed block after tail
410     else if (block > tail_nolock) {
411       block->prev = tail_nolock;
412       tail_nolock->next = block;
413       block->next = NULL;
414       tail_nolock = block;
415     }
416     // 4. somewhere between head and tail
417     else {
418       block_meta *curr = head_nolock;
419       // find two free blocks right before and right after block
420       block_meta *front = NULL, *back = NULL;
421       while (curr) {
422         if (curr < block) front = curr;
423         if (curr > block) {
424           back = curr;
425           break;
426         }
427         curr = curr->next;
428       }
429       if (front -> next != back) {
430         printf("front -> next != back\n");
431       }
432       block->prev = front;
433       block->next = back;
434       front->next = block;
435       back->prev = block;
436     }
437
438     merge_blocks_nolock(block);
439 }
440
441
442 // find a sutable free block accoridng to bf strategy
443 block_meta * find_bf_block_nolock(size_t size) {
444   block_meta * curr = head_nolock;
445   block_meta * res = NULL;
446   while (curr) {
447     // a new meta is always needed for allocation
448     if (curr->size >= size) {
449       // find the smallest space bigger equal than size
450       if (res == NULL || curr->size < res->size) {
451         res = curr;
452         // already found the best one
453         if (curr->size == size) break;
454       }
455     }
456     curr = curr->next;
```

```
  }
  return res;
}

// Best Fit malloc/free
void *bf_malloc_nolock(size_t size) {
  if (size == 0) return NULL;

  block_meta *block = find_bf_block_nolock(size);
  if (block == NULL) {   // no adequate free block, expand the heap
    block = request_space_nolock(size);
    // if (!block) return NULL; // heap expansion failure
  } else {  // adequate free block founded
    // 1. no need to split
    if (block->size <= size + META_SIZE) {
      allocate_block_nolock(block);
    }
    // 2. split the block
    else {
      split_block_nolock(block, size);
    }
  }
  return (void *)(block + 1);  // return the actual data's address by
skipping block_meta
  // ptr addition: 1 refers to 1 ptr's size, aka 1 block_meta size.
}

//Thread Safe malloc/free: non-locking version
void *ts_malloc_nolock(size_t size) {
  return bf_malloc_nolock(size);
}
void ts_free_nolock(void *ptr) {
  bf_free_nolock(ptr);
}
```