ECE_650 Project1: Malloc and Free

2025/1/19-23

Sienna Zheng (NetID: sz318)

Environment:

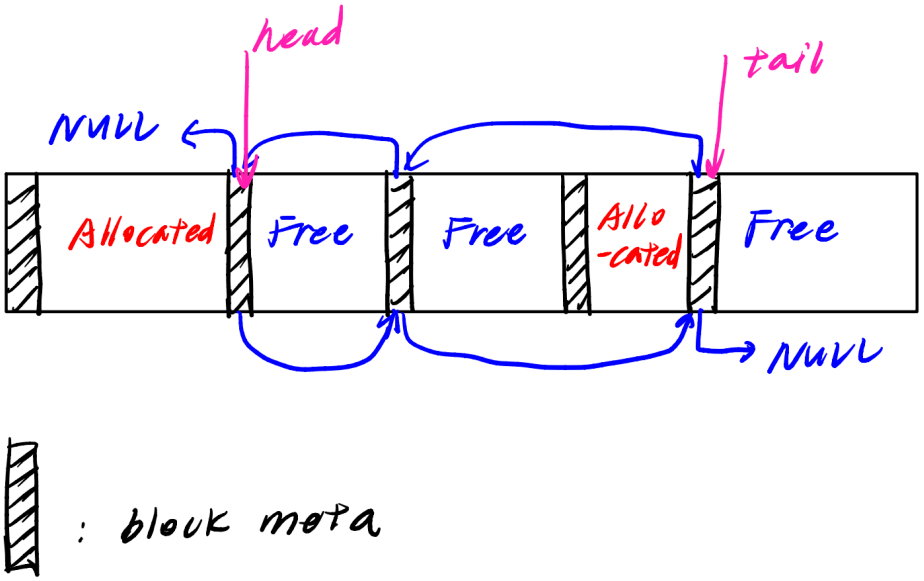
- C
- Linux VM

Introduction:

- Use `sbrk(size)` to implement `malloc` and `free` in c.
- 2 different strategies of malloc:
  - First-fit: Allocates the **first** free block that is large enough.
  - Best-fit: Allocates the **smallest** free block that is large enough.
- May need to **split** a free block into a allocated block of size and the rest of it remained free when `malloc`.
- May need to **merge** continuous free blocks into one free block when `free`.

Data Structure:

Free_List:



- Doubly-Linked List Implementation
  - `head` and `tail` are maintained.
  - Better performance than singly-linked list especially when freeing. (In a singly linked list, freeing a node requires traversing the list twice.)
- Only need to track **free** blocks for better performance.

Block:

- block_meta:
  - size of block data
  - prev (type: block_meta *)
  - next
- block_data

Test Results:

Data Segment Size:

Strategy	FF	BF
Data Segment Size / Bytes	3705640	3529960
Data Segment Free Space / Bytes	273784	95160

Execution Time and Fragmentation:

Testcases	FF Exc. Time (s)	BF Exc. Time (s)	FF Fragmentation	BF Fragmentation
Small_Range	11.275213	4.624651	0.073883	0.026958
Large_Range	36.338664	49.396084	0.093421	0.041318
Equal_Range	14.508256	14.636678	0.450000	0.450000

Analysis of the Test Results:

The table compares **First-Fit (FF)** and **Best-Fit (BF)** memory allocation strategies based on different parameters.

1. Data Segment Size & Free Space

- FF uses more memory compared to BF, suggesting FF tends to leave more unused space.
- FF also has more free space left than BF, suggesting FF results in more fragmentation.

2. Execution Time Comparison

- **Small_Range**: BF is faster.
- **Large_Range**: FF is faster.
- **Equal_Range**: FF is almost the same as BF.

3. Fragmentation Comparison

- **BF has lower fragmentation** than FF in **Small_Range** and **Large_Range**, meaning it better optimizes space utilization.
- **Equal_Range** fragmentation is **equal for both (0.45)**, implying both strategies struggle under this condition.

Summary:

- **Small_Range → BF is better** (less fragmentation, faster allocation)
- **Large_Range → FF is better** (faster execution)
- **Equal_Range → FF and BF perform similarly, so either can be chosen based on specific needs**.

Problems and Solution:

- In bf_malloc case, if we always traverse the entire free list to find the suitable block, it might lead to timeout.
- Improvement: when finding a block's size equal to `size`, we can return that block directly. It's already the best solution.

Appendix: Code

my_malloc.h:

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

#define META_SIZE sizeof(block_meta)

unsigned long data_segment_size = 0;

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

// 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");
}

// find a sutable free block accoridng to ff strategy
block_meta * find_ff_block(size_t size) {
  block_meta * curr = head;
  while (curr) {
    // a new meta is always needed for allocation
    if (curr->size >= size) {
      return curr;
    }
```

```
    curr = curr->next;
  }
  return NULL;
}

// 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;
}


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

  block_meta *block = find_ff_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 *)((char*)block + META_SIZE);  // return the actual data's
address by skipping block_meta
```

```
}

// 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
    if (next_block == tail) {
      tail = 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) {
      tail = prev_block;
    }
    prev_block->next = block->next;
    if (block->next) block->next->prev = prev_block;
    block->prev = NULL;
    block->next = NULL;
  }
}


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

    block_meta *block = (block_meta*)ptr - 1;   // meta address
    // 1. head = null, empty free list
    if (head == NULL) {
      head = block;
      tail = block;
      block->prev = NULL;
      block->next = NULL;
    }
    // 2. freed block before head, block becomes the new head
    else if (block < head) {
      block->next = head;
      head->prev = block;
      block->prev = NULL;
      head = block;
    }
```

```
201     // 3. freed block after tail
202     else if (block > tail) {
203       block->prev = tail;
204       tail->next = block;
205       block->next = NULL;
206       tail = block;
207     }
208     // 4. somewhere between head and tail
209     else {
210       block_meta *curr = head;
211       // find two free blocks right before and right after block
212       block_meta *front = NULL, *back = NULL;
213       while (curr) {
214         if (curr < block) front = curr;
215         if (curr > block) {
216           back = curr;
217           break;
218         }
219         curr = curr->next;
220       }
221       if (front -> next != back) {
222         printf("front -> next != back\n");
223       }
224       block->prev = front;
225       block->next = back;
226       front->next = block;
227       back->prev = block;
228     }
229 
230     merge_blocks(block);
231 }
232 
233 
234 // find a sutable free block accoridng to bf strategy
235 block_meta * find_bf_block(size_t size) {
236   block_meta * curr = head;
237   block_meta * res = NULL;
238   while (curr) {
239     // a new meta is always needed for allocation
240     if (curr->size >= size) {
241       // find the smallest space bigger equal than size
242       if (res == NULL || curr->size < res->size) {
243         res = curr;
244         // already found the best one
245         if (curr->size == size) break;
246       }
247     }
248     curr = curr->next;
249   }
250   return res;
251 }
252 
253 // Best Fit malloc/free
254 void *bf_malloc(size_t size) {
255   if (size == 0) return NULL;
256 
```

```
  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.
}

// same as ff_free
void bf_free(void *ptr) {
  ff_free(ptr);
}

// In bytes
// heap size
unsigned long get_data_segment_size() {
  return data_segment_size;
}
unsigned long get_data_segment_free_space_size() {
  unsigned long free_size = 0;
  block_meta *curr = head;

  while (curr) {
    free_size += curr->size + META_SIZE;  // free block's block_meta is included!
    curr = curr->next;
  }

  return free_size;
}
```