LIBMYMALLOC

Section 1: Overview

From what we've learned in `ECE551`, if a user requests for `32 bytes` size of space in heap, `malloc` function won't just allocate exactly `32 bytes` size for the user. Instead, the space that is allocated by `malloc` will be larger than `32 bytes` , because we will need meta information about this allocated space area. For convenience, from now on, let's define a name **BG(short for block group)** to represent an allocated space segment consisting of **meta information area(MIA)**and the **actual data space(ADS)** which will return to the user.

We will need to keep some kind of data structure to store **MIA** for each **BG**. The easiest way to do this is by creating a Linked List. Let's say we have several **BG's** inside the heap already. We let the head of the Linked List points at the first **BG's MIA**, and **next** field of **MIA** will point at the **MIA** inside next consecutive **BG**, so on and so forth. Each time when we want to `malloc` certain size of place, we search through the whole linked list and make sure:

- This **BG** is free
- This **BG's ADS** is at least larger than or equal to the size that the user requests for

  The structure of such Linked List may look like following:

```
typedef struct _Node_t Node_t;
struct _Node_t{
  Node_t * next;
  Node_t * prev;
  size_t blk_num;
  int isFree;
}
extern Node_t * head;
```

If we don't find such a **BG** after we search through the whole linked list, we increment the heap size by system call `sbrk()` . This implementation seems feasible, but each time we need to `malloc` , we need to search through the whole linked list(in the worst case) to find an available **BG**, which will result in poor performance in `alloc_policy` testing.

To improve our performance, we can think of keeping a linked list which only includes current free **BGs**, in which case wo do not need to search the whole data segment to find an available linked list. You may argue that we still need to search through the whole free linked list to find the **BG**. but in asymptotic analysis, the free space size will be much less than the whole data segment size(heap size). So this will improve our execution time dramatically for sure(This is actually the difference between the implementation I used before I went to prof. Tyler's OH and after that).

But how do we actually achieve this in implementation? Well, we can kill a bird with two stones. That is a `node` **(a.k.a MIA)** can be a member of both **normal linked list**[1] and **free linked list**. That forces us to make two head pointers, one called `head` and access it can make us search through the whole data segment while the other called `free_head` and access it can make us only search through the free segment.

We can also add a tail pointer to point at the last **BG** of the current data segment, which makes it easier to increment the heap size. We just need to modify the above `struct`:

```
typedef struct _Node_t Node_t;
struct _Node_t{
  Node_t * next; //next free node
  Node_t * prev; //prev free node
  size_t blk_num;
  Node_t * log_next; //next physcial node
  Node_t * log_prev; //prev physcial node
}
extern Node_t * head;  //head of normal linked list
extern Node_t * free_head; //head of free linked list
extern Node_t * tail; //point at last node of the current data segment
```

At this point, you may ask free linked list strategy seems better, but why do you still keep the normal linked list? This is the issue we need to deal with when merging adjacent free regions.

However, we have one more thing to tackle with in `malloc` implementation. What if the size of **ADS** inside a certain `free` **BG** is larger than the size that the user asks for? In such case, we may need to split the **ADS**. However, that depends on whether the remaining `free` segment is enough to at least fill in a **MIA** size. If not, we don't split the **ADS**, just returning the whole **ADS** to user. This implementation can be wrapped inside a function which dedicates to this job for us:

```
void * malloc(size_t size){
  if(size == 0){return NULL;}
  else{
    if(head == NULL){return add_new_segment_head(size);} //first malloc, we increment the heap
    else{
      Node_t * curr = free_head;
      while (curr != NULL){
        if(curr->blk_num >= size &&
           curr->blk_num < size + sizeof(Node_t)){
          //situation when don't need split
            deleteNode(curr);
            return (void *) (curr + 1);
        }
        else if(curr->blk_num >= size + sizeof(Node_t)){
          //situation when split is needed
          return split_and_insert(curr,size);
        }
        else{
          curr = curr->next;
          continue;
        }
      }
      return add_new_segment(size) //no free segment available, we increment the heap
    }
  }
}

void * split_and_insert(Node_t * curr, size_t size){
  //find the split point
  Node_t * new_meta_info = (Node_t *)( (int8_t *)curr + size + sizeof(Node_t));
  Node_t temp = {.next = NULL, .prev = NULL, .blk_num = curr->blk_num - size - sizeof(Node_t),
.log_prev = curr, .log_next = curr->log_next};
  *new_meta_info = temp;
  deleteNode(curr); //delete original segment from the free linked list
  addToFreeList(new_meta_info); //add the remaining new segment into the free linked list
  curr->log_next = new_meta_info;
  if(new_meta_info->log_next != NULL){
    new_meta_info->log_next->log_prev = new_meta_info;
  }
  else{
    tail = new_meta_info; //change tail
  }
  curr->blk_num = size;
  return (void *) (curr + 1);
}
```

In summary, when a user requests for space, our `malloc` searches through the free linked list to see if there is any **BG** that meets the requirements. If so, we allocate this **BG** by deleting it from the free linked list(setting both `next` and `prev` to `NULL`). If not, we allocate **BG** by incrementing the heap size.

Finally, we come to implement the `free` functionality. At first glance, it should be easy, because if we need to `free` a **BG**, we first check if it is already `free`. If not, simply remove it from the free list, then we are done! However, to avoid poor region selection when `malloc`, our `free` should be able to the newly freed region with any currently free adjacent regions. That is the reason why I keep my normal linked list. In doing so, I can easily trace to the currently being freed **BG** physically next and previous **BG** to check if they are also `free` so that I can merge them into one single **BG** if necessary.

Note: there are some tricks I use to reduce performance time:

- I can tell whether a **BG** is free or not in O(1) time complexity
- I can add a **BG** into free linked list in O(1) time complexity

Section 2: Performance Study

I . equal_size_alloc

The execution result of my first-fit and best-fit implementation are showed as follows:

> first_fit_implementation:
>
> Execution Time = 1.763137 seconds Fragmentation = 0.342857
>
> best_fit_implementation:
>
> Execution Time = 2.052996 seconds Fragmentation = 0.342857

First we need an overview about what this testing program does. This testing program basically allocate **2*NUM_ITEMS** pieces of segment, each of which is exactly *128 bytes*. The allocated pieces are divided into `array` and `spacing_array`. `spacing_array` is just like bulkhead that separates two consecutive `array` segments. Then it frees all `array` segments, but leaving `spacing_array` unchanged. After that it mallocs **INNER_ITERS** pieces of segments and keeps this amount of allocated `array` by mallocing repeatedly new piece and freeing the previous piece.

We can see that fragmentation for two algorithms are exactly the same. This is reasonable because all of the segments that involved in `malloc` and `free` have the same size. We have the same size of heap size for sure, for both programs initially allocate **2*NUM_ITEMS** pieces of segment. Because we have `spacing_array` always allocated to keep two free **BG** merging with each other, All **BGs** in the free linked list should have same size of **ADS**, and also they have same number of `Nodes`. So the free_data_segment size should be the same, thus leading to the same result of fragmentation. As for the `execution time`, because each time the program allocate and free with same `bytes`, each node in the linked list in the has enough **ADS** for next allocation. If we use `first-fit algorithm`, the first node in the free linked list will meet the requirement while for `best-fit algorithm`, we need to search through the whole free linked list to find the best **BG** to allocate, even if there is no need to do such redundancy. So best-fit algorithm's execution time will be longer.

II. small_range_rand_allocs

First as always, given the execution result:

> first_fit_implementation:

> data_segment_size = 3973552, data_segment_free_space = 240696 Execution Time = 3.555274 seconds Fragmentation = 0.060575
>
> best_fit_implementation:
>
> data_segment_size = 3713888, data_segment_free_space = 67328 Execution Time = 1.434206 seconds Fragmentation = 0.018129

As said in `README` file, this program allocate a large number (default 10000) of segments ranging from 128 to 512 bytes. And it alternates freeing a random selection of 50 of those, and mallocing 50 more regions with a random size from 128 - 512 bytes.

When allocating blocks, ***best_fit*** always finds a **BG** with the exact same available **ADS** as user requests for (in this case) so that there won't be split happening. However, if you use ***first_fit*** algorithm, `i.e:` a `128 bytes` request may occupy a **BG** with 512 bytes available **ADS**'s **BG**, which makes the next `512 bytes` request not able to find an available place to sit in current data segment(because the `512 bytes` block group has already been taken), thus resulting in increase of heap size. Another example will be easy to understand why `data_segment_free_space` for ***best_fit*** will be less than ***first_fit***. Let's say we have the following heap organization:

- 1000K free | 100K allocated | 30K free

The user requests for 30K space and 1000K space respectively. If we use ***first_fit*** algorithm, that may result in the following (for simplicity, we don't count the meta information size):

- 30K allocated | **970K free** | 100K allocated | 30K free | 1000K allocated

But the ***best_fit*** algorithm will try to find the block group with the most suitable size:

- 1000K allocated | 100K allocated | 30K allocated

best_fit algorithm improves the utilization efficiency, so it will has fewer useless free block group.

Now let's take a look at the execution time of the allocation policies. At first glance, you may seem very surprise that ***best_bit*** actually have a better performance than ***first_fit***. You may think that in order to find the **BG** with the best size, should ***best_fit*** go through the whole free list each time while ***first_fit*** only need to find the first available **BG**? What the heck is going on here? Well, this is not always the case. Firstly, if the current **BG's** available size is exactly the same as the user requests for, then we don't need to continue searching since this is definitely the best block group. Secondly, remember what we mentioned when we talked about the data_segment_size:

- ***first_fit*** will result in more split of block group, since there are more

  pieces and bits free memory **BGs** that are sparse through the whole data segment area. This will make the length of free linked list longer than the one in ***best_fit*** algorithm and this will inevitably increase the search time.
- Due to more fragmentation, ***first_fit*** will call more times of `sbrk()` to malloc space for users, which also increment the execution time.

III. large_range_rand_allocs

> first_fit_implementation:
>
> Execution Time = 28.642433 seconds Fragmentation = 0.111409

> best_fit_implementation:
>
> Execution Time = 106.584593 seconds Fragmentation = 0.040612

The basic pattern of this program is the same as the `small range` one except that the allocated block size is ranging from `128 B` to `64 KB`.

The execution time of both implementations become longer than before. This may because of the following reasons:

- since the range of block group's size is larger, it will take longer time to finish system call `sbrk()`
- Since the range is larger, there will be more `malloc` operation on space with larger discrepancy, which will result in more split operation when `mallocing`. This will make more number of block group inside free linked list. So it will increase the time to search available free block group to `malloc` next time. (O(N), N is increasing)

WAIT! At this time, you may feel extremely confused? What? Yes, it is the exact same pattern as that in the **small_range_allocs** policy, so the trends should be the same. But the ***best_fit*** becomes the slower one this time. It's chaos! If you see this more carefully, you will notice that the range of this policy changes from `128 B` to `64 KB`. It is actually quite different from the **small_range** one. Statistically, size of every block group will be much more different from other ones. So it will be much harder for ***best_fit*** to find a **BG** with the exact same size. In this case, split possibility won't differ very much in two allocation policy. So ***best_fit*** becomes the slower one again!

Section 3: Summary

first_fit algorithm have less execution time(better performance), but with poor heap utilization efficiency. On the other hand, ***best_fit*** algorithm have more execution time(poor performance), but with pretty good heap utilization efficiency (fragmentation is lower). If you are using `malloc` in the situation when performance is not the issue but the computer's memory is limited, then ***best_fit*** will be a better choice. Otherwise, ***first_fit*** will reduce time complexity of your program. However, if you are frequently allocating and freeing blocks in a relatively small range, you can achieve both benefits with ***best_fit*** algorithm.

---

1. A linked list in which the next field of one node points at the **BG** that is physically next to it.↩