590-01 software reliability

Aohua Zhang az147

```
#include "my_malloc.h"
#include <unistd.h>

//#define heapstart sbrk(0);

struct block {
  size_t size;
  struct block* next;
  struct block* prev;
};
typedef struct block block;

/* struct merge_bound { */
/*   block* left; */
/*   block* right; */
/* }; */
/* typedef struct merge_bound mergebound; */

block head = {0, NULL, NULL};

//helper functin that splits free space for use
void splitspace(block** cur, size_t size) {
  if((*cur)->size >= size && (*cur)->size <= size + 24) {
    block *toremove = *cur;
    if(toremove->next != NULL) {
      toremove->next->prev = toremove->prev;
    }
    (*cur) = (*cur)->next;
  } else if((*cur)->size > size + 24) {
    block* newblock = (*cur) + size + 24;
    newblock->prev = (*cur)->next->prev;
    newblock->size = (*cur)->size - size - 24;
    newblock->next = (*cur)->next;
    (*cur)->next = newblock;
    (*cur)->size = size;
  }
}

//first fit malloc/free
void *ff_malloc(size_t size) {
  //first iterate through the heap find a suitable space
  block** cur = &(head.next);
  while(*cur != NULL) {
    if((*cur)->size >= size) {
      //allocate the memory and spilt the region, modify the list accordingly
      splitspace(cur, size);
      return (void*)(*cur) + sizeof(block);
    }
    cur = &((*cur)->next);
```

```
}
  //if not find the avalable free space, move the program brk of the heap for more space
  void *oldbrk = sbrk(size + 24);
  block* node = oldbrk;
  node->size = size;
  node->next = NULL;
  node->prev = *cur;
  return oldbrk + sizeof(block);
}
void ff_free(void *ptr) {
  //mergebound mb = {NULL, NULL};
  block* left = NULL;
  block* start = ptr - 24;
  block* end = ptr + start->size;
  block** cur = &(head.next);
  if(!head.next) {
    head.next = start;
    start->prev = NULL;
    start->next = NULL;
    return;
  }
  //???这里没有显式转换指针类型
  if(head.next >= end) {
    //the newly freed space is leftmost
    if(head.next == end) {
      start->size += (head.next)->size + 24;
      start->next = (head.next)->next;
      start->prev = NULL;
      if((head.next)->next != NULL) {
    (head.next)->prev = start;
      }
      head.next = start;
    } else {
      start->next = head.next;
      start->prev = NULL;
      if(head.next != NULL) {
    head.next->prev = start;
      }
      head.next = start;
    }
  }
  //if not, find the position of this newly freed space
  while(*cur != NULL) {
    if(*cur <= ptr && ptr <= (*cur)->next || *cur <= ptr && (*cur)->next == NULL){
      left = *cur;
      break;
    }
  }
  if(left->next == NULL) {
    if(left + left->size == start) {
      left->size += (24 + start->size);
    } else {
      left->next = start;
      start->next = NULL;
      start->prev = left;
    }
    //newly freed block in between two adjacnet block
  } else {
```

```
    //three adjacent free blocks
    if(start == left + 24 + left->size && start + 24 + start->size == left->next) {
      left->size = left->size + 24 + start->size + 24 + left->next->size;
      left->next = left->next->next;
      if(left->next->next != NULL) {
    left->next->next->prev = left;
      }
    } else if(start == left + 24 + left->size) {
      left->size = left->size + 24 + start->size;
    } else if(start + 24 + start->size == left->next) {
      start->size = start->size + 24 + left->next->size;
      start->next = left->next->next;
      start->prev = left;
      if(start->next != NULL) {
    start->next->prev = start;
      }
    } else {
      start->next = left->next;
      start->prev = left->next->prev;
      start->next->prev = start;
      start->prev->next = start;
    }
  }

  return;
}

//Best fit malloc/free
void *bf_malloc(size_t size) {
block* cur = head;
  block* min = NULL;
  while(cur != NULL) {
    if(cur->size >= size) {
      min = cur;
      break;
    }
    cur = cur->next;
  }
  while(cur != NULL && min != NULL) {
    if(cur->size >= size && cur->size < min->size) {
      min = cur;
    }
    cur = cur->next;
  }
splitspace(cur, size);
      return (void*)(*cur) + sizeof(block);
}
void bf_free(void *ptr) {
  return;
}

//performance test
unsigned long get_data_segment_size() {
  return 0;
}
unsigned long get_data_segment_free_space_size() {
  return 0;
}
```

this is an implementation of malloc in c

1.the red highlighted code is duplicated code, the iteration to find a proper address to allocate the block for data should be abstracted out and form a new independent function.

duplicated code often comes from a copy-and-paste style, if we find there is some place needs to be changed for better algorithm or efficiency, we need to find and change every duplication of this piece of code, which could cause a large waster of time. And if there is something wrong and the later programmer can not manage to find all places, then the system could go wrong and crash.

2.the yellow highlighted region is a very long code segment, with a lot of if, else, while and embedded logics, this piece of code becomes very hard to understand, even for myself.

very long code and complicated code logics makes it very difficult for prorammer to analyze and understand, which makes it hard to debug. This could cause a large waster of time when the system needs to be maintained and upgraded.

3.the blue highlighted line is god line(which means that it is a very long line of code)

similar like 2, long line of code includes a lot of information, which make it hard to read and understand. Also, it is not a good looking when others reading the code.

4.Shotgun Surgery: there is a number 24 that appears in this programm a lot, which is the byte size of the block class, use the value of current class is very bad. Which means that if I want to apply a small modification to block class, I will have to also change other tens of spots. If I just omit some places, then the program could give the wrong result and still is difficult to debug.

develop/design principles to avoiding code smells

> *E.g. There is a design principle “DRY”: Don’t Repeat Yourself. It aims to avoid the*
> *code smell “Duplicate Code”. Duplicate codes make maintenance more difficult*
> *because if the developer wants to make one change, (s)he needs to change in several*

> *places. Faults easily arise if the developer forgets to change in one place. Therefore,*
> *avoiding the code smell “Duplicate Code” by the “DRY” principle improves*
> *software reliability.*

1. Do not use confusing naming

   an instance name being very long or too simple would cause difficulties in understanding and slow down the process of developing, debugging, and maintaining. Use clear name can save a lot of trouble later. Also, good method naming can speed up code reading and using a lot.

2. write comment properly

   comments in the right place can make the code more readable and helps other to understand, debug, import. But unclear and too much comments could ruin the flow of the code.

3. do not write very long function

   short and simple function means good abstraction. This kind of function is more readable, less possible to go wrong.

4. do not use very long argument list

   Similarly, very long argument list makes the code less readable and more easily to go wrong. When we really needs to deal with a lot of arguments in a function. We’d better wrap them into a single parameter object.