动态内存分配实验报告

Malloc Lab

(Dynamic Storage Allocators)

07300720035 电子信息科学与技术 王泮渠

(Department of Electrical Engineering, Chris Wang)

2010.01.02

INTRODUCTION

In this lab you will be writing a dynamic storage allocator for C program,i.e.,your own version of the malloc, free and realloc routines. You are encouraged to explore the design space creatively and implement an allocator that is correct, efficient and fast.

PREPARATIONS

Read the introduction of malloc_lab carefully and think about it in depth. It shows us how to on the program, how to check our syntax and grammars, and how to evaluate our performance and find how to improve it.

The text book CSAPP is always your best instructor and helper. It provide with an original version of Implicit Free List. Although its performance is relatively low, it shows the basic program structure and algorithm about dynamic memory allocators. Also, we can get familiar with all functions, libraries, pointers and variables used in the gigantic program.

We should cautiously select our algorithm and data structure used in the program to meet with the request and evaluation system of malloc_lab. The most important, our program must be run on the server successfully, smoothly and efficiently without any errors, bugs or leaks.

ABOUT MY PROGRAM

DATA STRUCTURE

As mentioned above, the original version of implicit free list introduced in the text book is a good gateway program, but it can be only the gateway, cannot be the summit. As we all know, although implicit free list is simple, a significant disadvantage is that the cost of any operation, such as placing allocated blocks, that requires a search of the free list will be linear in the total number of allocated free blocks in the heap. As a result, implicit free list is not appropriate for a general-purpose allocator.

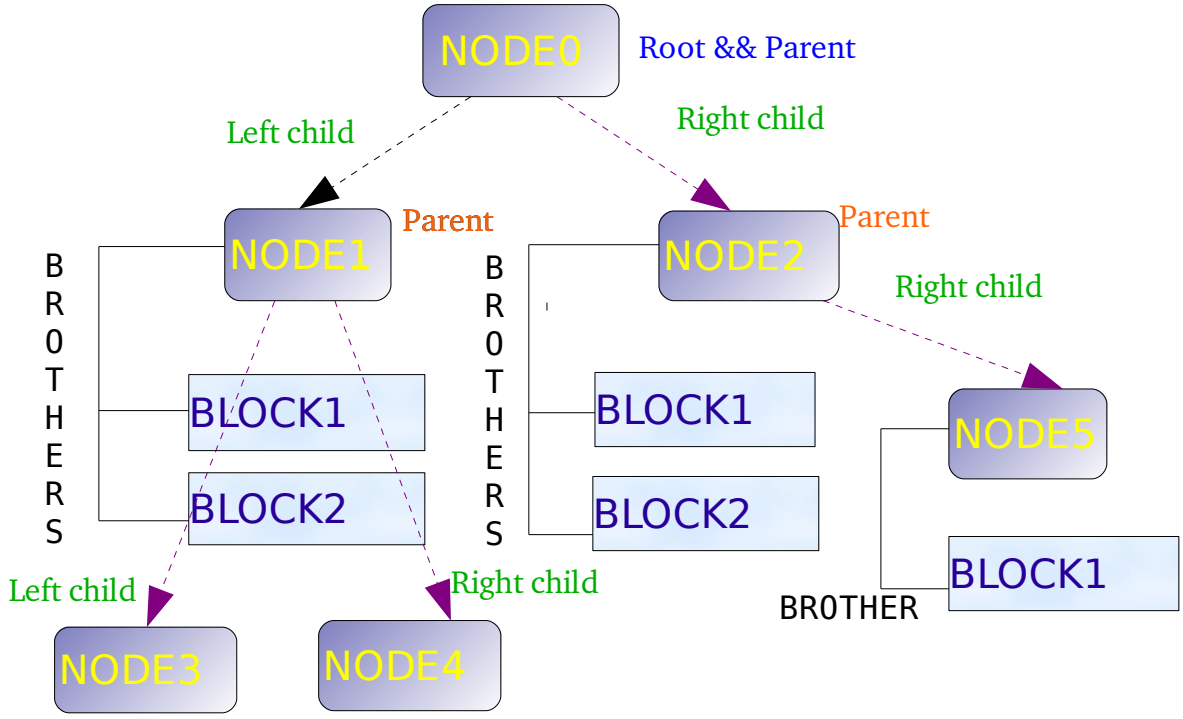
So I turned to the Explicit Free List. Since by definition the body of a free block is not needed by the program, the pointers that implement the data structure can be stored within the bodies of the free blocks. Using a doubly-linked list instead of an implicit free list reduces the first fit allocation time from linear in the total number of blocks to linear in the number of free blocks. However, the time to free a block can be either linear or constant, depending on the policy we choose for ordering the blocks in the free list.

In my program, the essence is to find the policy(data structure) for ordering the blocks in the free list. I select the data structure named **BINARY TREE**.

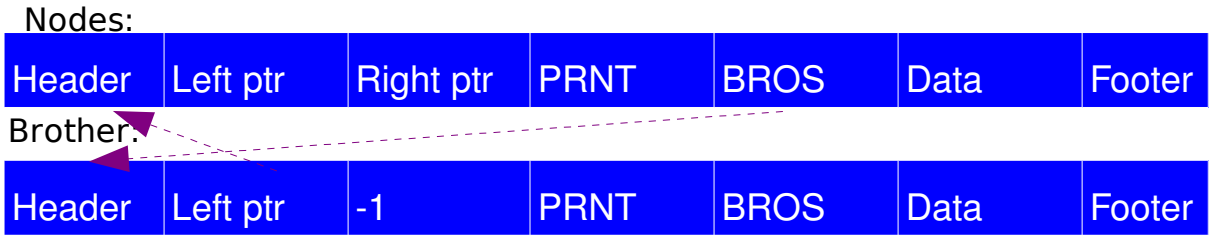
The name of BINARY TREE is not unfamiliar to us, since we all meet with it in the secret phase our our bomb lab! In my program, we use BINARY TREE to store our free list. That is, from the root, nodes and the leaves, each one represents one **KIND** of free list. Every node who have children are called **PARTENTS**, and if one node has one or more blocks that has the same size of it, the node, together with the blocks, are called **BROTHERS**. The parent node typically have 2 children, the **LEFT** child and the **RIGHT** child.

We can use the graph below to describe the relationship of all characters

mentioned above:



When I put my structure onto one list, it will appear as below:



As can be seen from above, nodes definitely have parents,brothers and its left and right child. However, the node's brothers DONOT have its right child. Their left children are the clustering lines to show their paralleled but not privileged status to the nodes. If one node is deleted, one of their brothers will replace it to be parents or children. So, all data structure of my BINARY TREE will require at least 24 bytes. (one block represent 4 bytes, 1 word)

Fit Strategy(Algorithm)

We will choose BEST FIT strategy for my program. It examines every free block and selects the free block with the smallest size that fits. If we combine the best fit strategy with the BINARY TREE, the original disadvantage for best fit, that is the cost of time for utter search, will be eliminated.

We can predict that our BINARY TREE will possess following personalities:
1.With relatively high speed in searching and allocating, the throughput per time will be enormous.
2.As we adopt BEST FIT strategy, the memory utilization will be quite good. However, even the smallest block in the binary tree can be 24 bytes, it may cause relatively high amount fragmentations. How to balance them must be an interesting but crucial issue.

Now, let's start the throughout observation of my code!

Firstly, we have to define some constants and macros:

```
/*Constants*/
#define SSIZE 4                 //Single word, 4 bytes
#define DSIZE 8                 //Double words, 8 bytes
#define TSIZE 12                //Triple words, 12 bytes
#define QSIZE 16                //Quadri words, 16 bytes
#define OVERHEAD 8              //Header and footer sign, 8 bytes
#define ALIGNMENT 8             //Alignment request, 8 bytes
#define BLKSIZE 24              //Single word, 24 bytes
#define CHUNKSIZE (1<<12)       //Initial heap size
#define INISIZE 1016            //Heap extended size

/*Macros*/
/*Max and min value of 2 values*/
#define MAX(x, y) ( (x)>(y)? (x): (y) )
#define MIN(x, y) ( (x)<(y)? (x): (y) )

/*Read and write a word at address p*/
#define GET(p) (*(size_t *)(p))
#define PUT(p, val) (*(size_t *)(p)=(val))
```

```
/*Read the size and allocated fields from address p*/
#define SIZE(p) (GET(p)&~0x7)
#define PACK(size, alloc) ((size)|(alloc))
#define ALLOC(p) (GET(p)&0x1)

/*Given pointer p at the second word of the data structure, compute addresses of its HEAD,LEFT,RIGHT,PRNT,BROS and FOOT pointer*/
#define HEAD(p) ((void *)(p)-SSIZE)
#define LEFT(p) ((void *)(p))
#define RIGHT(p) ((void *)(p)+SSIZE)
#define PRNT(p) ((void *)(p)+DSIZE)
#define BROS(p) ((void *)(p)+TSIZE)
#define FOOT(p) ((void *)(p)+SIZE(HEAD(p))-DSIZE)

/*Make the block to meet with the standard alignment requirements*/
#define ALIGN_SIZE(size) (((size) + (ALIGNMENT-1)) & ~0x7)

/*Given block pointer bp, get the POINTER of its directions*/
#define GET_SIZE(bp) ((GET(HEAD(bp)))&~0x7)
#define GET_PREV(bp) ((void *)(bp)-SIZE(((void *)(bp)-DSIZE)))
#define GET_NEXT(bp) ((void *)(bp)+SIZE(HEAD(bp)))
#define GET_ALLOC(bp) (GET(HEAD(bp))&0x1)

/*Get the LEFT,RIGHT,PRNT,BROS and FOOT pointer of the block to which bp points*/
#define GET_LEFT(bp) (GET(LEFT(bp)))
#define GET_RIGHT(bp) (GET(RIGHT(bp)))
#define GET_PRNT(bp) (GET(PRNT(bp)))
#define GET_BROS(bp) (GET(BROS(bp)))
#define GET_FOOT(bp) (GET(FOOT(bp)))
```

```
/*Define value to each character in the block bp points to.*/
#define PUT_HEAD(bp, val) (PUT(HEAD(bp), (int)val))
#define PUT_FOOT(bp, val) (PUT(FOOT(bp), (int)val))
#define PUT_LEFT(bp, val) (PUT(LEFT(bp), (int)val))
#define PUT_RIGHT(bp, val) (PUT(RIGHT(bp), (int)val))
#define PUT_PRNT(bp, val) (PUT(PRNT(bp), (int)val))
#define PUT_BROS(bp, val) (PUT(BROS(bp), (int)val))
```

All functions and global variables used in the program:

```
/* non-static functions */
int mm_init ( void );
void *mm_malloc ( size_t size );
void mm_free ( void *bp );
void *mm_realloc ( void *bp, size_t size );
/* static functions */
static void *coalesce ( void *bp );
static void *extend_heap ( size_t size );
static void place ( void *ptr, size_t asize );
static void insert_node ( void *bp );
static void delete_node ( void *bp );
static void *find_fit ( size_t asize );
/* Global variables */
static void *heap_list_ptr;
static void *free_tree_rt;
```

Now we will analyze the functions step by step.

MM_INIT

mm_init is to initialize the malloc package. It gets four words from the memory system and initializes them to create empty free list. Then it calls extend_heap function to extends the heap by CHUNKSIZE bytes and creates the initial free block. At this point, the allocator is initialized and ready to accept allocate and

free requests from the application.

```
int mm_init(void)
{
        /* create the initial empty heap */
        if( (heap_list_ptr = mem_sbrk(QSIZE)) == NULL )
                return -1;
        PUT( heap_list_ptr, 0 ); /* alignment padding */
        PUT( heap_list_ptr+SSIZE, PACK(OVERHEAD,1) ); /* prologue header */
        PUT( heap_list_ptr+DSIZE, PACK(OVERHEAD,1) ); /* prologue footer */
        PUT( heap_list_ptr+TSIZE, PACK(0,1) ); /* epilogue header */
        heap_list_ptr += QSIZE;
        free_tree_rt = NULL;

        /* Extend the empty heap with a free block of CHUNKSIZE bytes */
        if( extend_heap(ALIGN_SIZE(INISIZE))==NULL )
                return -1;
        return 0;
}
```

EXTEND_HEAP

extend_heap extends the heap with a new free block. It is invoked when the heap is initialized or mm_malloc is unable to find a suitable fit. The function must meet with the required size standard and then request the additional heap space from the memory system. At the end of the function, we call the coalesce function to merge the two free blocks and return the block pointer to the merged blocks.

```
void *extend_heap(size_t size)
{
        void *bp;
        if( (unsigned int)(bp=mem_sbrk(size)) ==(unsigned)(-1))
```

```
//if( (int)(bp=mem_sbrk(size)) <0 )//new
        return NULL;

    /* Initialize free block header/footer and the epilogue header */
    PUT_HEAD( bp, PACK(size,0) ); /* free block header */
    PUT_FOOT( bp, PACK(size,0) ); /* free block footer */
    PUT_HEAD( GET_NEXT(bp), PACK(0,1) ); /* new epilogue header */
    insert_node(coalesce(bp));
    return (void *)bp;
}
```

MM_MALLOC

After initializing, we use mm_malloc to allocate a block by incrementing the brk pointer. We always allocate a block whose size is a multiple of the alignment. The function below is very likely to the function described on the textbook. The behaviors of the function are:

1. Checking the spurious requests.
2. Adjust block size to include overhead and alignment requirements.
3. Search the free list for a fit.
4. Place the block into its fit.

At the end of the function, I looked in the traces and found the best strategy to meet the performance evaluation principle of the project, so I added the if sentence after it. If not, the performance of No.7&8 will decrease significantly.

```
void *mm_malloc(size_t size)
{
    size_t asize; /* adjusted block size */
    size_t extendsize; /* amount to extend heap if no fit */
    void *bp;
    /* Ignore spurious requests */
    if( size <= 0 )
        return NULL;
```

```
        /* Adjust block size to include overhead and alignment requirements. */
        if( size <= BLKSIZE-OVERHEAD)
       /*size=required size; block size = all size excluded head&foot(overhead)*/
                asize = BLKSIZE;
        else
               /*asize=ajusted size*/
                asize = ALIGN_SIZE(size+(OVERHEAD));
        /* Search the free list for a fit */
               if( (bp=find_fit(asize)) == NULL ){
                extendsize = MAX( asize + 32, INISIZE );
                extend_heap(ALIGN_SIZE(extendsize));
                if( (bp=find_fit(asize)) == NULL )
                        return NULL;
        }
        /* place the block into its fit */
        if( size==448 && GET_SIZE(bp) > asize+64 )
                asize += 64;
        else if( size==112 && GET_SIZE(bp) > asize+16 )
                asize += 16;
        place(bp, asize);
        return bp;
}
```

MM_FREE

mm_free is to free a block to do nothing. The coalesce function will be explained later.

```
void mm_free(void *bp)
{
        size_t size = GET_SIZE(bp);
        PUT_HEAD( bp, PACK(size,0) );
        PUT_FOOT( bp, PACK(size,0) );
```

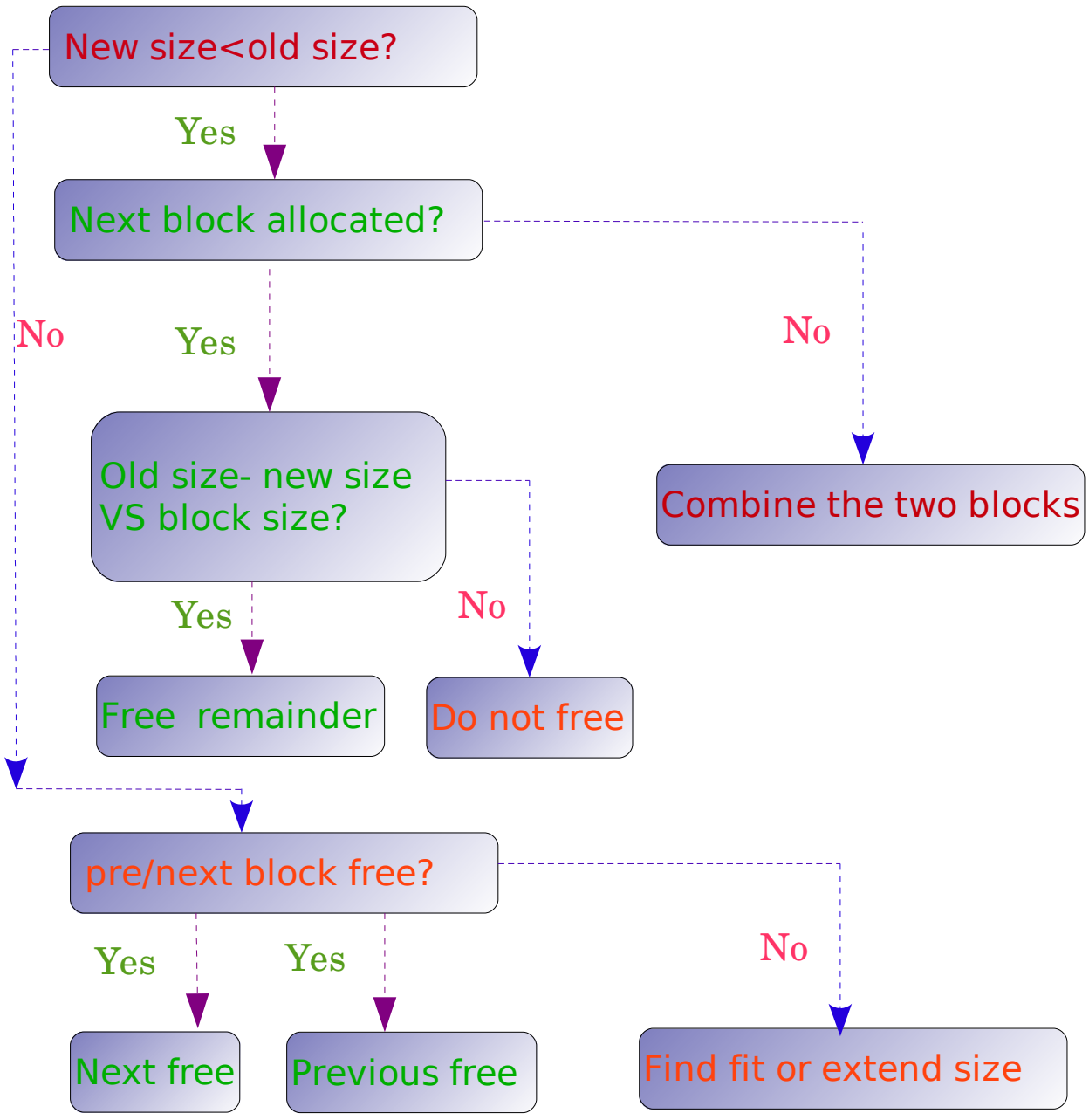
```
    insert_node(coalesce(bp));
}
```

MM_REALLOC

mm_realloc is implemented simply in terms of mm_malloc and mm_free .
The behavior of the function is listed below:

1.When ptr==NULL or size==0, free ptr.

2.When size>0, compare new size and old size and then adopt relative strategies. I will show all the possibilities below:



```
void *mm_realloc(void *ptr, size_t size)
{
        if( ptr==NULL || size==0 ){
                mm_free(ptr);
                return NULL;
        }
        if( size > 0 ){
                size_t oldsize = GET_SIZE( ptr );
                size_t newsize = ALIGN_SIZE( size+OVERHEAD );

                if( newsize < oldsize ){ /* newsize is less than oldsize */
                        if( GET_ALLOC( GET_NEXT(ptr) ) ){
 /* the next block is allocated */
                                if( (oldsize-newsize) >= BLKSIZE ){
 /* the remainder is greater than BLKSIZE */
                                        PUT_HEAD( ptr, PACK(newsize,1) );
                                        PUT_FOOT( ptr, PACK(newsize,1) );//newsize
                                        void *temp = GET_NEXT(ptr);
//this pointer points to extra space
                                        PUT_HEAD( temp, PACK(oldsize-newsize,0) );
                                        PUT_FOOT( temp, PACK(oldsize-newsize,0) );
                                        insert_node(temp);
                                }
                                else{ /* the remainder is less than BLKSIZE */
                                        PUT_HEAD( ptr, PACK(oldsize,1) );
//oldsize still occupies all spaces.
                                        PUT_FOOT( ptr, PACK(oldsize,1) );
                                }
                                return ptr;
                        }
                        else{ /* the next block is free */
                                size_t csize = oldsize + GET_SIZE( GET_NEXT(ptr) );
```

```
                        delete_node( GET_NEXT(ptr) );
                        PUT_HEAD( ptr, PACK(newsize,1) );
                        PUT_FOOT( ptr, PACK(newsize,1) );
                        void *temp = GET_NEXT(ptr);
                        PUT_HEAD( temp, PACK(csize-newsize,0) );
                        PUT_FOOT( temp, PACK(csize-newsize,0) );
                        insert_node(temp);
                        return ptr;
                }
        }
        else{ /* newsize is greater than oldsize */
                size_t prev_alloc = GET_ALLOC(GET_PREV(ptr));
                size_t next_alloc = GET_ALLOC(GET_NEXT(ptr));
                size_t csize;
/* the next block is free and the addition of the two blocks no less than the new
size    */
                if( !next_alloc &&
( (csize=oldsize+GET_SIZE(GET_NEXT(ptr))) >= newsize) ){
                        delete_node(GET_NEXT(ptr));
                        if((csize-newsize)>=BLKSIZE){
                                PUT_HEAD( ptr, PACK(newsize,1) );
                                PUT_FOOT( ptr, PACK(newsize,1) );
                                void *temp=GET_NEXT(ptr);
                                PUT_HEAD( temp, PACK(csize-newsize,0) );
                                PUT_FOOT( temp, PACK(csize-newsize,0) );
                                insert_node(temp);
                        }else{
                                PUT_HEAD( ptr,PACK(csize,1) );
                                PUT_FOOT( ptr,PACK(csize,1) );
                        }
                        return ptr;
                }
```

```
/ * the previous block is free and the addition of the two blocks no less than the new size */
                    else if( !prev_alloc &&
( (csize=oldsize+GET_SIZE(GET_PREV(ptr))) >= newsize) ){
                            delete_node(GET_PREV(ptr));
                            void *newptr=GET_PREV(ptr);
                            memcpy( newptr, ptr, oldsize-OVERHEAD );
                            if((csize-newsize)>=BLKSIZE){
                                    PUT_HEAD( newptr,PACK(newsize,1) );
                                    PUT_FOOT( newptr,PACK(newsize,1) );
                                    void *temp=GET_NEXT(newptr);
                            PUT_HEAD( temp,PACK(csize-newsize,0) );
                            PUT_FOOT( temp,PACK(csize-newsize,0) );
                                    insert_node(temp);
                            }else{
                                    PUT_HEAD( newptr,PACK(csize,1) );
                                    PUT_FOOT( newptr,PACK(csize,1) );
                            }
                            return newptr;
                    }
                    else{
/* the next and previous block is free and the addition of the two blocks less than the new size */
                            size_t asize=ALIGN_SIZE(size+(OVERHEAD));
                            size_t extendsize;
                            void *newptr;
                            if((newptr=find_fit(asize))==NULL){
                                    extendsize=MAX(asize,CHUNKSIZE);
                                    extend_heap(extendsize);
                                    if((newptr=find_fit(asize))==NULL)
                                            return NULL;
                            }
```

```
                        place( newptr, asize );
                        /*copy content from memory*/
                        memcpy( newptr, ptr,oldsize-OVERHEAD);
                        mm_free(ptr);
                        return newptr;
                    }
                }
            }
            else
                return NULL;
}
```

Above all, the mm_realloc must follow the principle that make full use of current space as much as possible and try to use adjacent space of the changed blocks to meet with new requests.

COALSCE

coalesce is to merge one free block with any adjacent free blocks and to update binary tree's structure in time.

There are 4 possibilities:

Blocks	Previous block	Next block	Results (coalescing)
Block condition: Allocated=1, Freed=0	0	0	previous+current+next
	0	1	previous+current
	1	0	current+next
	1	1	current

Each possibility is listed below as case 0 to 3.

After coalescing, pointer returns to the big freed block.

```
static void *coalesce(void *bp)
{
        size_t prev_alloc = GET_ALLOC(GET_PREV(bp));
        size_t next_alloc = GET_ALLOC(GET_NEXT(bp));
        size_t size = GET_SIZE(bp);

        if ( prev_alloc && next_alloc ) /* Case 0 */
                return bp;

        else if ( !prev_alloc && next_alloc ) { /* Case 1*/
                delete_node(GET_PREV(bp));
                size += GET_SIZE( GET_PREV(bp) );
                PUT_HEAD( GET_PREV(bp), PACK(size, 0) );
                PUT_FOOT( bp, PACK(size,0) );
                return GET_PREV(bp);
        }

        else if ( prev_alloc && !next_alloc ) { /* Case 2 */
                delete_node( GET_NEXT(bp) );
                size += GET_SIZE( GET_NEXT(bp) );
                PUT_HEAD( bp, PACK(size,0) );
                PUT_FOOT( bp, PACK(size,0) );
                return bp;
        }

        else { /* Case 3 */
                delete_node(GET_NEXT(bp));
                delete_node(GET_PREV(bp));
                size += GET_SIZE( GET_PREV(bp) ) +
GET_SIZE( GET_NEXT(bp) );
                PUT_HEAD( GET_PREV(bp), PACK(size,0) );
                PUT_FOOT( GET_NEXT(bp), PACK(size,0) );
```

```
            return GET_PREV(bp);
        }
}
```

PLACE

place is to place the requested block.
If the remainder of the block after slitting would be greater than or equal to the minimum block size, then we go ahead and split the block. We should realize that we need to place the new allocated block before moving to the next block. It is very likely to the operation on mm_realloc.

```
static void place(void *bp,size_t asize)
{
        size_t csize = GET_SIZE(bp);
        delete_node( bp );

        if((csize-asize)>=BLKSIZE){
            PUT_HEAD( bp,PACK(asize,1) );
            PUT_FOOT( bp,PACK(asize,1) );
            bp=GET_NEXT(bp);
            PUT_HEAD( bp,PACK(csize-asize,0) );
            PUT_FOOT( bp,PACK(csize-asize,0) );
            insert_node( coalesce(bp) );
        }

        else{
            PUT_HEAD( bp,PACK(csize,1) );
            PUT_FOOT( bp,PACK(csize,1) );
        }
}
```

FIND_FIT

find_fit performs a fit search. Our basic principles for BEST FIT strategies in BINARY TREE are :

1.We must eventually find a fit block after searching the binary free tree.

2.We must choose the least size free block compared with the requested size. So we should initially move toward left and go on. When the block in the left is not big enough to support the block, move right.

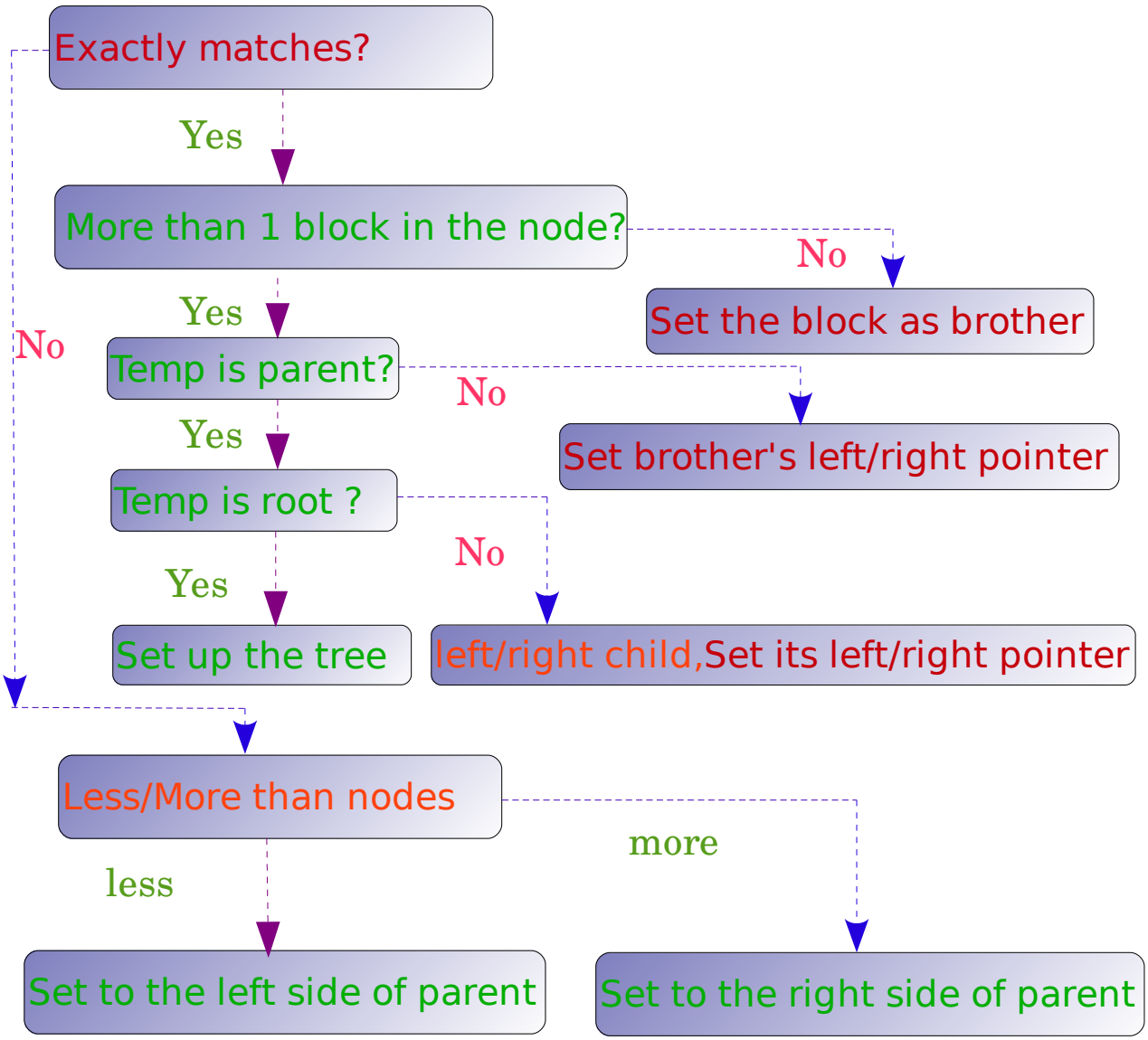
3.If the block is so big that every node cannot fit it (till the rightmost), extend the heap and put the block in the rightmost of the tree.

```
static void* find_fit( size_t asize )
{
        /* the most fit block */
        void *fit = NULL;
        /* temporary location of the search */
        void *temp = free_tree_rt;
        /* use tree to implement a comparative best fit search */
        for(;temp!=NULL;){
/* The following node in the search may be worse, so we need to record the
most fit so far. */
                if( asize <= GET_SIZE(temp) ){
                        fit = temp;
                        temp = (void *)GET_LEFT(temp);
                }
                else
                        temp = (void *)GET_RIGHT(temp);
        }
        return fit;
}
```

INSERT_NODE

insert_node is to insert a free block into the free-block binary tree. It requires deep understand of the tree structure. It also have multiple conditions.

At first, the insert_node will help us to build a new free tree.
And then, we should divide the circumstance into catalogs to treat.
The method is to draw the graph again. You may see the principle maybe similar to the mm_realloc function. However, insert node is more complicated considering whether the temp is the root, parent, child, block or node.



```
static void insert_node( void *bp )
{
        /* root is NULL */
        if( free_tree_rt == NULL ){
                free_tree_rt = bp;
                PUT_LEFT( bp, NULL );
                PUT_RIGHT(bp, NULL );
                PUT_PRNT( bp, NULL );
                PUT_BROS( bp, NULL );
                return;
        }

        /* treat temp as the start */
        void *temp = free_tree_rt;
        /* loop to locate the position */
        while( 1 ){
                /* Case 1: size of the block exactly matches the node. */
                if( GET_SIZE(bp)==GET_SIZE(temp) ){
                        if( (void *)GET_BROS(temp) != NULL )
                 {/* more than one block in the node */
                                if( temp == free_tree_rt ){/* temp is parent,and the root*/
                                        free_tree_rt = bp;
                                        PUT_PRNT( bp, NULL );
                                }
                                else{/* temp is parent, and not the root */
                                        if( (void *)GET_LEFT(GET_PRNT(temp)) == temp
)
                                /* temp is left child(temp's parent's left child is temp) */
                                                PUT_LEFT( GET_PRNT(temp), bp );
//put temp's parent's left point as bp.
                                        else /*  temp is right child */
                                                PUT_RIGHT( GET_PRNT(temp), bp );
```

```
                        PUT_PRNT( bp, GET_PRNT(temp) );
//put bp's parent as temp's parent.
                    }
                    PUT_LEFT( bp, GET_LEFT(temp) );
                    PUT_RIGHT( bp, GET_RIGHT(temp) );
                    PUT_BROS( bp, temp );
                //if temp is not parent(only siblings)
                    if( (void *)GET_LEFT(temp) != NULL )
                        PUT_PRNT( GET_LEFT(temp), bp );
                    if( (void *)GET_RIGHT(temp) != NULL )
                        PUT_PRNT( GET_RIGHT(temp), bp );
                    PUT_LEFT( temp, bp );
                    PUT_RIGHT( temp, -1 );
                    break;
                }

                else{/* no more than one block in the node */
                    PUT_BROS( bp, NULL );
                    PUT_LEFT( bp, temp );
                    PUT_RIGHT( bp, -1 );
                    PUT_BROS( temp, bp );
                    if( (void *)GET_BROS(bp) != NULL )
                        PUT_LEFT( GET_BROS(bp), bp );
                    break;
                }
            }
            /* Case 2: size of the block is less than that of the node. */
            else if( GET_SIZE(bp) < GET_SIZE(temp) ){
                if( (void *)GET_LEFT(temp) != NULL ){
                    temp = (void *)GET_LEFT( temp );
                }else{
                    PUT_LEFT( temp, bp );
```

```
                        PUT_PRNT( bp, temp );
                        PUT_LEFT( bp, NULL );
                        PUT_RIGHT( bp, NULL );
                        PUT_BROS( bp, NULL );
                        break;
                }
        }
        /* Case 3 size of the block is greater than that of the node. */
        else{
                if( (void *)GET_RIGHT(temp) != NULL ){
                        temp = (void *)GET_RIGHT(temp);
                }else{
                        PUT_RIGHT( temp, bp );
                        PUT_PRNT( bp, temp );
                        PUT_LEFT( bp, NULL );
                        PUT_RIGHT( bp, NULL );
                        PUT_BROS( bp, NULL );
                        break;
                }
        }
    }
}
```

DELETE_NODE

delete_node is to delete a free block from the free-block binary tree. It also has many possibilities. As they are more or less similar to the mm_realloc and insert_node, I will not draw the graph this time. We can see the logic structure from the sentence below.

```
static void delete_node(void *bp)
{
    /* Case that the block is the only one in the node */
```

```
if( (void *)GET_BROS(bp) == NULL && GET_RIGHT(bp) != -1 ){
        if( bp == free_tree_rt ){/* the node is the root */
                if( (void *)GET_RIGHT(bp) == NULL ){/* no right child */
                        free_tree_rt=(void *)GET_LEFT(bp);
                        if( free_tree_rt != NULL )
                                PUT_PRNT( free_tree_rt, NULL );
                }
                else{/* it has a right child */
                        void *temp = (void *)GET_RIGHT(bp);
                        while( (void *)GET_LEFT(temp) != NULL )
                                temp = (void *)GET_LEFT(temp);
                        void *tempL = (void *)GET_LEFT(bp);
                        void *tempR = (void *)GET_RIGHT(temp);
                        void *tempP = (void *)GET_PRNT(temp);
                        free_tree_rt = temp;
                        if( free_tree_rt != NULL )
                                PUT_PRNT( free_tree_rt, NULL );
                        PUT_LEFT( temp,GET_LEFT(bp) );
                        if( temp != (void *)GET_RIGHT(bp) ){
                                PUT_RIGHT( temp,GET_RIGHT(bp) );
                                PUT_LEFT( tempP, tempR );
                                if( tempR != NULL)
                                        PUT_PRNT( tempR, tempP );
                                PUT_PRNT( GET_RIGHT(bp),temp );
                        }
                        if( tempL != NULL )
                                PUT_PRNT( tempL, temp );
                }
        }
        else{/* the node is not the root */
                if( (void *)GET_RIGHT(bp) == NULL ){/* no right child */
                        if( (void *)GET_LEFT( GET_PRNT( bp ) ) == bp )
```

```
                    PUT_LEFT( GET_PRNT(bp), GET_LEFT(bp) );
                else
                    PUT_RIGHT( GET_PRNT(bp), GET_LEFT(bp) );

                if( (void *)GET_LEFT(bp) != NULL)
                    PUT_PRNT( GET_LEFT(bp), GET_PRNT(bp) );
            }else{/* it has a right child */
                void *temp = (void *)GET_RIGHT(bp);
                while( (void *)GET_LEFT(temp) != NULL )
                    temp = (void *)GET_LEFT(temp);
                void *tempL = (void *)GET_LEFT(bp);
                void *tempR = (void *)GET_RIGHT(temp);
                void *tempP = (void *)GET_PRNT(temp);
                if( (void *)GET_LEFT(GET_PRNT(bp)) == bp )
                    PUT_LEFT( GET_PRNT(bp), temp );
                else
                    PUT_RIGHT( GET_PRNT(bp), temp );

                PUT_PRNT( temp, GET_PRNT(bp) );
                PUT_LEFT( temp, GET_LEFT(bp) );
                if( temp != (void *)GET_RIGHT(bp)){
                    PUT_RIGHT( temp, GET_RIGHT(bp) );
                    PUT_LEFT( tempP, tempR );
                    if( tempR != NULL )
                        PUT_PRNT( tempR,tempP );
                    PUT_PRNT( GET_RIGHT(bp), temp );
                }
                if( tempL != NULL )
                    PUT_PRNT( tempL, temp );
            }
        }
    }else{/* Other case */
```

```
            if( bp == free_tree_rt ){/* the node is the root */
                free_tree_rt = (void *)GET_BROS(bp);
                PUT_PRNT( free_tree_rt, NULL );
                PUT_LEFT( free_tree_rt, GET_LEFT(bp) );
                PUT_RIGHT( free_tree_rt, GET_RIGHT(bp) );
                if( (void *)GET_LEFT(bp) != NULL )
                    PUT_PRNT( GET_LEFT(bp), free_tree_rt );
                if( (void *)GET_RIGHT(bp) != NULL )
                    PUT_PRNT( GET_RIGHT(bp), free_tree_rt );
            }else{/* the node is not the root */
                if( GET_RIGHT(bp) == -1 ){/* not the first block in the node */
                    PUT_BROS( GET_LEFT(bp),GET_BROS(bp) );
                    if( (void *)GET_BROS(bp) != NULL )
                        PUT_LEFT( GET_BROS(bp),GET_LEFT(bp) );
                }else{/* the first block in the node */
                    if( (void *)GET_LEFT(GET_PRNT(bp)) == bp )
                        PUT_LEFT( GET_PRNT(bp), GET_BROS(bp) );
                    else
                        PUT_RIGHT( GET_PRNT(bp), GET_BROS(bp) );
                    PUT_PRNT( GET_BROS(bp), GET_PRNT(bp) );
                    PUT_LEFT( GET_BROS(bp), GET_LEFT(bp) );
                    PUT_RIGHT( GET_BROS(bp), GET_RIGHT(bp) );
                    if( (void *)GET_LEFT(bp) != NULL )
                        PUT_PRNT(GET_LEFT(bp), GET_BROS(bp) );
                    if( (void *)GET_RIGHT(bp) != NULL)
                        PUT_PRNT(GET_RIGHT(bp), GET_BROS(bp) );
                }
            }
        }
}
```

Performance test result:

```
Team Name: stu007300720035
Member 1 : Wang Panqu: wangpanqumanu@sina.com
Using default tracefiles in /home/traces/
Measuring performance with gettimeofday().

Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.000541 10535
 1       yes   99%    5848  0.000599  9756
 2       yes  100%    6648  0.000691  9624
 3       yes  100%    5380  0.000571  9420
 4       yes   88%   14400  0.000753 19134
 5       yes   96%    4800  0.001498  3204
 6       yes   95%    4800  0.001576  3045
 7       yes   89%   12000  0.001818  6601
 8       yes   90%   24000  0.002461  9752
 9       yes   58%   14401  0.003057  4711
10       yes   89%   14401  0.000972 14819
Total          91%  112372  0.014537  7730

Perf index = 55 (util) + 40 (thru) = 95/100
correct: 11
perfidx: 95
```

Summarization

We can compare the results with my predictions above, do you remember?

“1.With relatively high speed in searching and allocating, the throughput per time will be enormous.

2.As we adopt BEST FIT strategy, the memory utilization will be quite good. However, even the smallest block in the binary tree can be 24 bytes, it may cause relatively high amount fragmentations. How to balance them must be an interesting but crucial issue.”

As can be seen from the result, we can see the throughput is really amazing. More than 7500 key throughput per second is far over the full score

requirements. That is because my design takes full consideration of all kinds of circumstances that will happen in mm_realloc, insert_node and delete_node. This will help the allocator find the right and most appropriate free list in time with high efficiency.

Secondly, the **BINARY TREE** is really a save of time. It separates all circumstances into 2 different parts, and then divide again and again. That is to say, we will save at least ½ time in searching for the best fit point. As a result, our method get over the shortcoming of best fit——time-consuming full search. It is really a magnificent combination of data structure and algorithm.

Unfortunately, however, our binary tree is really something divergent and gigantic. The MINIMUM of the block should be 24 bytes. In another word, even an one-byte requests to the allocator will cost 24 bytes consuming, which is really waste of spaces. Another disadvantage is the spatial utilization: The minimum 24 bytes will let every space smaller than 24 bytes become zombie(I would like to say that) because they can neither be allocated nor be freed as long as one free block follows it. So the spatial utilization will not be optimal, and the result turns out to prove it: The last 5 is not stable, and NO.9 is only 58%, which must be in consistency with the original data in it.

To sum up, when we deal with an allocation problem, two crucial characters need to be considered and balanced: **Fit strategy and Program Structure**. We would like to say the algorithm and data structure radically. Different fit strategy, structure selected, or server operated, even expected outcomes will influence our dynamic memory allocating strategy. Anyway, one sentence to fit every allocators: The best fit, the best used!