实验报告(动态内存分配器的实现)

PB17000289 于佳睿

实验目的

- 使用显示空闲链表实现一个32位系统堆内存分配器。

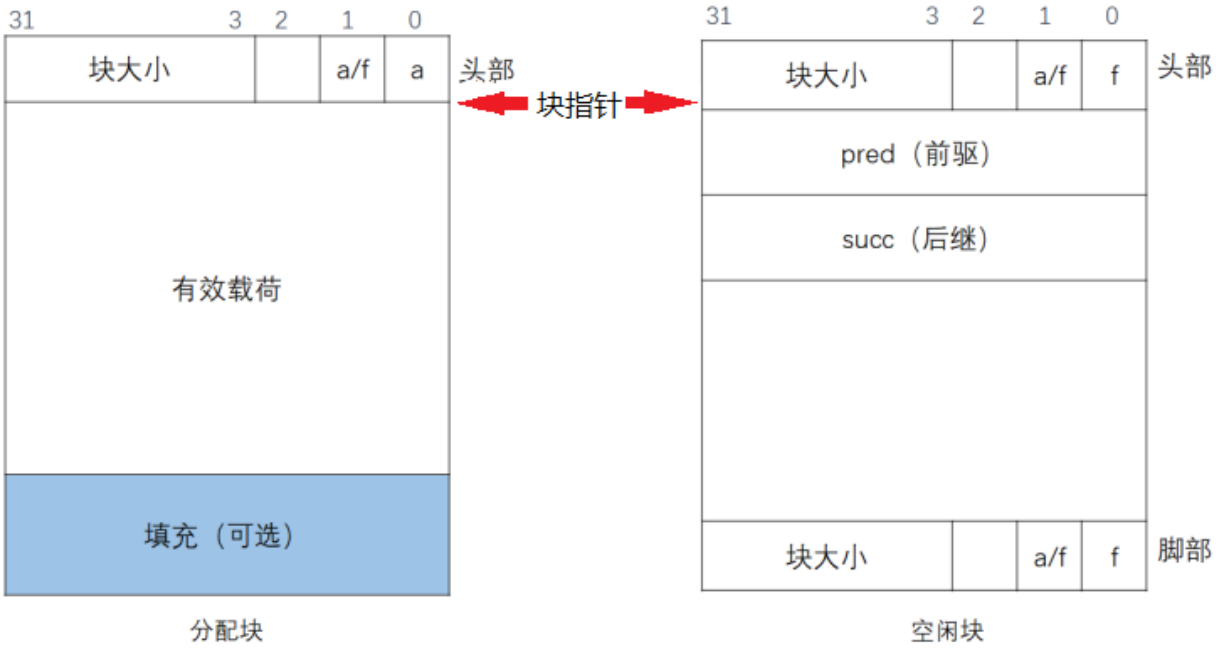
实验环境

- OS：Ubuntu 14.04 i386（32位）
- Linux内核版本：Kernel 2.6.26

实验步骤

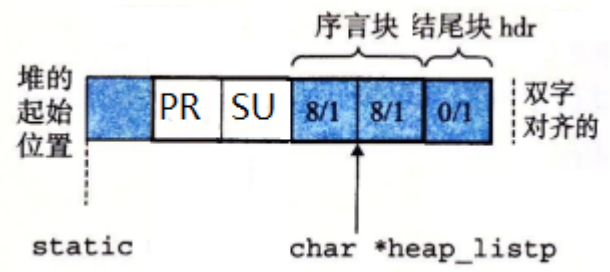
一、确定存储数据结构

1. 块结构



表示块地址的块指针指向有效载荷(分配快)和前驱地址(空闲块)

2. 空闲块显式链表结构



除图中所示的数据外，结尾块的前块空闲标志位应该设置为1，表示已分配，PR和SU是链表头，PR永远为NULL，SU是首个空闲块的指针，如果没有空闲块，SU = NULL.

3. 代码思路

- mm_init是存储空间的初始化，申请堆和初始化链表.
- mm_malloc是在链表中找到合适大小的存储空间，如果找得到，就返回该块的指针，否则申请堆，返回堆的块地址.同时对剩余的存储空间分割合并在空闲链表中.
- mm_free是释放指针指向的空间，释放后与周围的空闲空间合并，串到链表头部.
- mm_realloc是对某块数据重新分配存储空间，首先观察前后有没有可以合并的块，如果没有，则寻找新的合适的存储空间，将数据copy到新的空闲块，如果有，观察合并后是否够大，如果够大，这调整指针位置将数据放入该块.

4. 宏定义说明(新增的7个宏)

```
#define GET_PREV_ALLOC(p) (GET(HDRP(p))& 0x2)    //获取前块分配信息
#define SET_ALLOC(size, p)         ((GET_PREV_ALLOC(p))? PACK(size,
0x3):PACK(size, 0x1))   //用于当前块已被申请时，设置当前申请块的块大小，  前块信息
#define SET_UNALLOC(size, p)       ((GET_PREV_ALLOC(p))? PACK(size,
0x2):PACK(size, 0x0))   //用于当前块未被申请时，设置当前申请块的块大小，  前块信息
#define GET_PREV(p)      (GET(p))          //获得块的前驱地址
#define PUT_PREV(p, val) (PUT(p, (size_t)val))  //写入块的前驱地址
#define GET_SUCC(p)      (*((size_t *)p + 1))        //获得快的后继
#define PUT_SUCC(p, val) (*((size_t *)p + 1) = (size_t)(val)) //写入块的后继
```

5. 基本函数

- link2root: 将空闲块头插到链表

```
void link2root(void* p)   //将p连到root后
{
    size_t succ = GET_SUCC(list_root);   //获得list_root的后继
    PUT_PREV(p, list_root);                //将list_root的地址值放入p中
    PUT_SUCC(list_root, p);              //将p的地址值放入listroot中
    PUT_SUCC(p, succ);                  //将后继的地址放在p中
    if(succ) PUT_PREV(succ, p);         //如果后继不是NULL，将p放在后继中
}
```

- linkjump: 将P块从链表中删除

```
void linkjump(void *p)    //将p块从显式链表中删除
{
    size_t succ, prev;
    succ = GET_SUCC(p);                 //p的后继
    prev = GET_PREV(p);                 //p的前驱
```

```
    PUT_SUCC(prev, succ);               //将后继地址放在前驱中
    if(succ) PUT_PREV(succ, prev);       //如果后继不是NULL，将前驱地址放在后继中
}
```

- coalesce函数: 合并空闲块，并维护链表

```
static void *coalesce(void * p){
    size_t prev_alloc = GET_PREV_ALLOC(p);            //前面块是否被分配 1分配0空闲
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(p)));   //后面块是否被分配 1分配0空闲
    size_t size = GET_SIZE(HDRP(p));                  //p块的大小
    if(prev_alloc && next_alloc){      //前后都被分配
        link2root(p);                  //将p块连在头后面
        return p;
    }
    else if(prev_alloc && !next_alloc){    //前分后闲
        size += GET_SIZE(HDRP(NEXT_BLKP(p))); //获取下一块的大小，加在一起作为合并块的大小
        linkjump(NEXT_BLKP(p));                //将后块分离
        PUT(HDRP(p), SET_UNALLOC(size, p));   //本块空闲，大小size
        PUT(FTRP(p), PACK(size, 0));
        link2root(p);                          //合并后放在链表头后
    }
    else if(!prev_alloc && next_alloc){      //前闲后分
        size += GET_SIZE(HDRP(PREV_BLKP(p)));     //获取前块的大小，加在一起
        linkjump(PREV_BLKP(p));             //先断开，合并后放到前面
        PUT(FTRP(p), PACK(size, 0));               //本块空闲，大小size
        PUT(HDRP(PREV_BLKP(p)), SET_UNALLOC(size, PREV_BLKP(p)));  //前面块的信息更新，大小size未分配
        p = PREV_BLKP(p);                  //合并
        link2root(p);                      //放入链表
    }
    else{                                      //前后都闲
        size += GET_SIZE(HDRP(PREV_BLKP(p))) + GET_SIZE(FTRP(NEXT_BLKP(p)));  //算总的大小
        linkjump(PREV_BLKP(p));                                                //前后析离
        linkjump(NEXT_BLKP(p));
        PUT(HDRP(PREV_BLKP(p)), SET_UNALLOC(size, PREV_BLKP(p)));              //更新前面后面的信息
        PUT(FTRP(NEXT_BLKP(p)), PACK(size, 0));
        p = PREV_BLKP(p);                                                       //合并
        link2root(p);                  //放入链表， 由于后面本来就是空闲块，所以后面的状态信息没变
    }
    return p;                          //返回合并块指针
}
```

二、修改**mm.c**

- 修改mm_init

  按照上面设计的空间初始状态，设计初始状态.

```
int mm_init(void)
{
    if((heap_listp = (char *)mem_sbrk(6*WSIZE)) == (void *)-1)//静态指针向堆申请6*4的内存，分别是填充字，链表头，序言快和结尾块
        return -1;
    PUT(heap_listp, 0);     //填充字置零
    PUT(heap_listp + 3*WSIZE, PACK(DSIZE, 1));   //序言块
    PUT(heap_listp + 4*WSIZE, PACK(DSIZE, 1));
    PUT(heap_listp + 5*WSIZE, PACK(0, 0x3));   //结尾块前块是序言快，所以不是空闲
    heap_listp += 4*WSIZE;     //指向序言块
    list_root = (size_t *)(heap_listp - 3*WSIZE);  //指向链表头
    PUT_PREV(list_root, 0);     //前驱后继置为null
    PUT_SUCC(list_root, 0);     //后继设置为null
    if(extend_heap(CHUNKSIZE) == NULL)     //申请堆
        return -1;
    return 0;
}
```

- 修改mm_malloc

  mm_malloc函数与书上版本大致一致，改动的是求newsize的方法，由于分配块不留尾部，所以只要加WSIZE取双字对齐即可

```
void *mm_malloc(size_t size)
{
    char *p;
    size_t extendsize;
    if(size <= 0)
        return NULL;
    size_t newsize = ALIGN(size + WSIZE);  //预留头的位置，也保证双字对齐
    if((p = find_fit(newsize)) != NULL){   //找到了足够的空闲块

        place(p, newsize);                 //把p后面的newsize置为分配状态
        return p;                          //返回指针p
    }
    extendsize = MAX(newsize, CHUNKSIZE);   //扩展堆
    if((p = extend_heap(extendsize)) == NULL){   //无法分配时，返回NULL
        return NULL;
    }
    place(p, newsize);                       //把p后面的newsize大小置为分配
    return p;
}
```

在find_fit函数中采用小范围最佳适配策略(PS:上面注释掉的是首次适配)

```
static void *find_fit(size_t newsize){          //找大小合适的空闲块
   /* size_t t = GET_SUCC(list_root);            //t是listroot的后继地址值
    while(t != 0){                              //如果t不是NULL，则尝试该地址
        if(newsize <= GET_SIZE(HDRP(t)))        //注意newsize已经是加上头部和双字对齐
后的，所以只和size比较
            return (void *)t;                   //返回void型的指针
        else
            t = GET_SUCC(t);                    //t变成t块的后继
    }
    return NULL;                                 //找不到返回NULL*/
        size_t t = GET_SUCC(list_root);
        int count = 150;
        size_t minsize = 0xffffffff;
        void * minp = NULL;
        while(t != 0 && count != 0){
                count --;
                if(newsize <= GET_SIZE(HDRP(t))){
                        if(GET_SIZE(HDRP(t))<minsize){
                                minsize = GET_SIZE(HDRP(t));
                                minp = (void *)t;
                        }
                }
                t = GET_SUCC(t);
        }
        return minp;
}
```

place函数与原来大体相同，要额外增加状态信息(下面块的头部信息)，以及链表维护

```
static void place(void *p, size_t newsize){
    size_t total_size;
    total_size = GET_SIZE(HDRP(p));                          //总块大小
    linkjump(p);                                              //将P从链表中剔除
    if(total_size-newsize < 4*WSIZE){                        //<=16，  保证双字对齐，
分不出来了
        //printf("no need to cut, only %d left\n", total_size-newsize);
        PUT(HDRP(p), PACK(GET(HDRP(p)), 0x1));               //更新本块信息，本块已经
不空闲了
        PUT(HDRP(NEXT_BLKP(p)), PACK(GET(HDRP(NEXT_BLKP(p))), 0x2)); //告诉下一块，
上一块已经不空闲了
    }
    else{                                                    //可以分割
        PUT(HDRP(p), SET_ALLOC(newsize, p));                 //将信息写入当前块头部，指明其
大小，是否被申请，保留上一块信息
        void *pnew = NEXT_BLKP(p);                           //分割得到的空闲块的块指针
        PUT(HDRP(pnew), PACK((total_size-newsize), 0x2));    //将信息写入得到的空闲块
头部，尾部，头部指明前块已分配，本快未分配
        PUT(FTRP(pnew), PACK((total_size-newsize), 0x0));
        //printf("cut, %d left  before coalese\n", total_size-newsize);
        coalesce(pnew);
```

```
    }
}
```

- 修改mm_free

  mm_free也和原版本大致相同，区别在于更新头部的存储的上一块信息

```
void mm_free(void *ptr)
{
    size_t size = GET_SIZE(HDRP(ptr));          //获取本快大小
    PUT(HDRP(ptr), SET_UNALLOC(size, ptr));     //将该块的是否空闲的信息更新
    PUT(FTRP(ptr), PACK(size, 0));              //由于该块空闲所以更新尾部信息
    coalesce(ptr);                              //合并空闲块两边的块，并维护链表
}
```

- 定制mm_realloc解决方案

  由于realloc的mem_cpy时间复杂度未知，所以应该尽量减少数据块移动的次数，所以应该先检查前后空闲块的合并是否足够再去free，malloc. 这里没有考虑ptr指向空闲块的情况，也通过了，所以不考虑这种情况，为了维护链表方便，定制了

```
void *mm_realloc(void *ptr, size_t size)
{
    void *oldptr = ptr;                                 //旧指针
    void *newptr;                                       //新指针
    size_t newsize = ALIGN(size + WSIZE);               //需要申请的块的大小
    size_t oldsize = GET_SIZE(ptr);                     //旧指针块的大小
    int free_enough = 0;                                //相邻的下一块够不够
    if(size == 0){                                      //如果size是0，free掉空间返
回NULL
        mm_free(oldptr);
        return NULL;
    }
    if(ptr == NULL){                                     //如果ptr是空指针
        return mm_malloc(size);                         //申请一块返回块指针
    }
    if(newsize == oldsize){                              //显然不需要分割，直接返回
ptr
        return ptr;
    }
    else if(newsize < oldsize){                          //newsize更小，所以要考虑
是不是有分割
        realloc_place(ptr, newsize);                    //放入newsize中
        return ptr;
    }
    else{                                                //oldsize更小，需要扩展
空间
        newptr = realloc_coalesce(ptr, newsize, &free_enough);
        size_t copySize = GET_SIZE(HDRP(ptr));
```

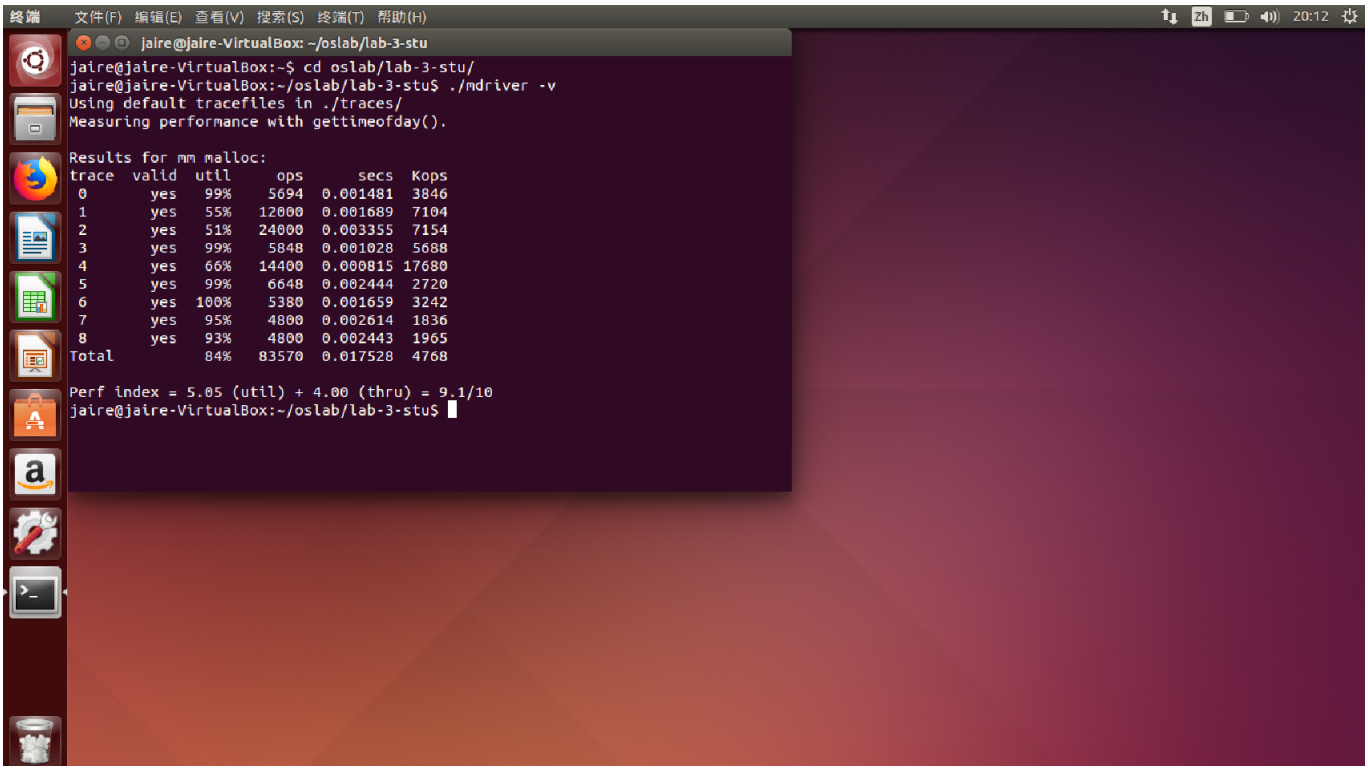
```
        if(free_enough == 1){   //case 2
            realloc_place(ptr, newsize);
            return ptr;
        }
        else if(newptr != ptr){   //case 3 4
            memcpy(newptr, oldptr, copySize);
            realloc_place(newptr, newsize);
            return newptr;
        }
        else{
            newptr = mm_malloc(newsize);
            memcpy(newptr, oldptr, copySize);
            mm_free(oldptr);
            return newptr;
        }
    }
}
```

由于合并后，马上会使用，所以在realloc时，如果要合并前后块，无需在realloc_coalesce中将空闲块插入，因为马上要使用该空闲块，然后在realloc_place中使用该块，并维护其分割.

```
static void *realloc_coalesce(void *p, size_t newsize, int *next_free_enough)
//p又可能是分配快
{
    size_t prev_alloc = GET_PREV_ALLOC(p);
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(p)));
    size_t size = GET_SIZE(HDRP(p));
    *next_free_enough = 0;

    if (prev_alloc && !next_alloc) {      /* Case 2 */  //上分下闲
        size += GET_SIZE(HDRP(NEXT_BLKP(p)));           //和下面的大小加在一起获取总
的大小
        if(size >= newsize)                             //如果newsize小就合并
        {
                linkjump(NEXT_BLKP(p));                     //将该块从链表中分离出来
            PUT(HDRP(p), SET_UNALLOC(size, p));         //设置头部信息
            PUT(FTRP(p), PACK(size, 0));                //设置尾部信息
            *next_free_enough = 1;                      //标志位，块指针不用动
        }
    }

    else if (!prev_alloc && next_alloc) {      /* Case 3 */
        size += GET_SIZE(HDRP(PREV_BLKP(p)));
        if(size >= newsize)
        {
         linkjump(PREV_BLKP(p));
            PUT(FTRP(p), PACK(size, 0));
            PUT(HDRP(PREV_BLKP(p)), SET_UNALLOC(size, PREV_BLKP(p)));
            p = PREV_BLKP(p);
        }
    }
```

```
    else {                                          /* Case 4 */
        size += GET_SIZE(HDRP(PREV_BLKP(p))) + GET_SIZE(FTRP(NEXT_BLKP(p)));
        if(size >= newsize)
        {
        linkjump(PREV_BLKP(p));
            linkjump(NEXT_BLKP(p));
            PUT(HDRP(PREV_BLKP(p)), SET_UNALLOC(size, PREV_BLKP(p)));
            PUT(FTRP(NEXT_BLKP(p)), PACK(size, 0));
            p = PREV_BLKP(p);
        }
    }
    return p;
}

static void realloc_place(void *p, size_t newsize){
    size_t total_size;
    total_size = GET_SIZE(HDRP(p));
    if(total_size- newsize < 4*WSIZE){
        PUT(HDRP(p), PACK(GET(HDRP(p)), 0x1));                          //不分
割，本快已分配
        PUT(HDRP(NEXT_BLKP(p)), PACK(GET(HDRP(NEXT_BLKP(p))), 0x2));     //下一块
的前块信息更新为已分配
    }
    else{
        PUT(HDRP(p), SET_ALLOC(newsize, p));
        void *pnew = NEXT_BLKP(p);
        PUT(HDRP(pnew), PACK((total_size-newsize), 0x2));   //将信息写入得到的空闲块
头部，尾部，头部指明前块已分配
        PUT(FTRP(pnew), PACK((total_size-newsize), 0x0));
        coalesce(pnew);                                          //因为
realloc_coalesce已将p块剔除，这里将分割出来的块放到链表头
    }
}
```

实验结果



代码获得了9.1分.

实验总结

1. 遇到的BUG和解决方法

- int型不能作为membrk的返回值

  > 刚开始我参考的是CSAPP的旧版本，它以int型作为membrk返回值，导致数据溢出，返回一个负值，进而导致了mm_init failed，查阅资料后我使用size_t作为返回值，因为size_t类型是存放size的数据类型，一定可以存储地址值.

- 注意位运算的信息丢失

  > 宏定义中很多是位运算，位运算要注意逻辑和数据位数，否者容易信息丢失.

- 删除链表节点要在更新size之前

  > 刚开始写coalesce函数时，我先更新size然后将合并前的空闲块与链表分离，导致了Segmentation Fault错误，GDB单步调试后发现，是因为更新size在分离之前，而寻找下一个指针需要使用NEXT_BLKP(p)宏，而该宏依赖于size的原来大小，size的不同导致了越界和错误.

- 不同类型的指针+1结果不同

  > size_t类型的指针+1是加一个字，char类型的指针+1是加一个字节

- GCC的编译等级-O2

  > 修改或删除该参数可以控制GCC优化的级别，使调试更明晰，否则GDB调试时乱七八糟.....

2. trace文件与代码优化

由运行结果可以看出，trace 1 2的得分很低，分析数据可知，trace 1 2都是大块小块相间申请，然后free小块，造成碎片化，显式空闲链表无法合并不相邻的块，所以无法解决这一问题，在调整了适配策略之后，才上升为9.1分，但是trace 1 2的得分没有提高...

3. 实验收获

通过本次实验，我阅读了CSAPP整个第九章(旧版第十章)，对OS课程中讲的碎片化等概念有了进一步的了解，通过本次实验，我复习了宏定义，位运算，指针等C语言概念，熟练了在LINUX系统下的GDB_DEBUG，了解了一些malloc函数的底层知识，收获很大.