

Lab5 实验报告

本次实验为五次实验最难的一次实验，本实验报告仅按照书上的版本进行了实现(find_fit 和 place 函数是我自己实现的)，不过我会将提供书上代码的十分完整的注释版，且进行了一些小小的优化，可使分数提高。

无论是隐式空闲链表还是显式空闲链表(或者分离式空闲链表)，根据 malloclabPDF 的要求，任何复合的数据结构都是不允许的(数组，树，结构体等)，只能使用普通的数据类型，因此在书上，mm.c 中使用了 static char*heap_listp 作为隐式空闲链表的头节点，建立在此基础上，开始以下代码讲解：

首先是书上的宏：

```
#define WSIZE      4           //书上自定义的字、脚部或头部的大小(字节)，intel中2个字节是一个字，4个字节是双字
#define DSIZE      8           //书上自定义的双字大小(字节)
#define CHUNKSIZE (1<<12)     //扩展堆时的默认大小(4096/WSIZE)
#define MINBLOCK (DSIZE + 2*WSIZE) //设置最小块的大小(块内容(有效荷载+填充)+头部(一个字)+尾部(一个字))，尽管整个书上的代码并没有用到

#define MAX(x, y) ((x) > (y) ? (x) : (y)) //比较x和y的大小，返回较大的那个，相等的话返回y

#define PACK(size, alloc) ((size) | (alloc)) //将 size 和有效位合并为一个字

#define GET(p) (*(unsigned int *) (p)) //读地址p处的一个字
#define PUT(p, val) (*(unsigned int *) (p) = (val)) //写地址p处写一个字，写为val

#define GET_SIZE(p) (GET(p) & ~0x07) //得到地址p处的 size (GET得到该处的字，接着与上~0x07，即0xfffffff8，将size取出)
#define GET_ALLOC(p) (GET(p) & 0x1) //得到地址p处的有效位(是否分配，取最低位)
//bp=block point(指向有效荷载)
#define HDRP(bp) ((char*) (bp) - WSIZE) //获得头部的地址
#define FTRP(bp) ((char*) (bp) + GET_SIZE(HDRP(bp)) - DSIZE) //获得脚部的地址，利用GET_SIZE取得块大小，加上这个size后减去头部和脚部的大小(总共为双字)

#define NEXT_BLK(p) ((char*) (bp) + GET_SIZE(((char*) (bp) - WSIZE))) //计算后块的地址，利用头部较近的特点，算出size，此时直接加上这个size就行了，因为中间跨过了所在块的脚部和下一个块的头部
#define PREV_BLK(p) ((char*) (bp) - GET_SIZE(((char*) (bp) - DSIZE))) //计算前块的地址
```

我们跳过头文件：

```
#define ALIGNMENT 8 //这条宏将该分配器设置为8字对齐
```

```
#define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~0x7)//将size伸展成对齐的大小
```

```
#define SIZE_T_SIZE (ALIGN(sizeof(size_t)))//将size_t的大小调成和对齐标准一样
```

```
static char* heap_listp;//指向序言块
```

mm_init 函数:

```
int mm_init(void)//初始化隐式空闲链表
```

```
{
    if ((heap_listp = mem_sbrk(4 * WSIZE)) == (void*)-1)//注意，mem_sbrk为一个外部函数，位于memlib.c中，其作用为申请额外的堆内存，返回申请位置的首地址，此处是为了得到堆的起始位置，同时如果初始化失败，返回-1。
        return -1;
    PUT(heap_listp, 0);//将初始位置置零
    PUT(heap_listp + (1 * WSIZE), PACK(DSIZE, 1));//将初始位置偏移量为一个字的位置设为双字大小，有效位为1
    PUT(heap_listp + (2 * WSIZE), PACK(DSIZE, 1)); //将初始位置偏移量为两个字的位置设为双字大小，有效位为1
    PUT(heap_listp + (3 * WSIZE), PACK(0, 1)); //将初始位置偏移量为三个字的位置设为0，有效位为1
    heap_listp += (2 * WSIZE);//将heap_listp偏移两个字
    if (extend_heap(CHUNKSIZE / WSIZE) == NULL)//extend_heap函数见下面
        return -1;//extend_heap返回异常，初始化失败
    return 0;//正常返回
}
```

Extend_heap 函数:

```
static void* extend_heap(size_t words)//扩充堆，成功的话执行合并操作，并返回申请空间指向有效荷载的指针，否则返回NULL
```

```
{
    char* bp;
    size_t size;
    size = (words % 2) ? (words + 1) * WSIZE : words * WSIZE;//将words对齐，并算出该字数对应的字节数
    if ((long)(bp = mem_sbrk(size)) == -1)//调用mem_sbrk函数提高堆空间
        return NULL;//失败返回NULL

    PUT(HDRP(bp), PACK(size, 0));//对新分配的块的头部进行初始化
    PUT(FTRP(bp), PACK(size, 0)); //对新分配的块的脚部进行初始化
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1));//处理结尾块

    return coalesce(bp);//调用合并函数（由于链表中最后一个块有可能为空闲块，故分配后有可能出现两个空闲块，故还需判断是否需要合并）
}
```

Coalesce 函数：

`static void* coalesce(void* bp)` //将bp指向的块与前后空闲的块合并，并返回合并后的“有效荷载的”首地址

```
{
    size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKPTR(bp))); //得到前一个块的分配情况
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKPTR(bp))); //得到后一个块的分配情况
    size_t size = GET_SIZE(HDRP(bp)); //得到本块的大小

    if (prev_alloc && next_alloc) //前后两块都已分配
    {
        //跟原书的做法相比，该处我进行了一个小小的优化，能使分数高一分
        PUT(HDRP(bp), PACK(size, 0)); //更新该块头部的状态，这一步和接下来的这一步是从
        mm_free函数那儿贴过来的，当时分析这一步时有个疑问，其他的if else条件里都改了对应块的状态，就这个没改，后面分析到mm_free函数时发现可以将这两个语句放到这里来，就试了一下，发现分数高了一分
        PUT(FTRP(bp), PACK(size, 0)); //更新该块脚部的状态
        return bp; //一定要保留这句话，后面太多if else了，浪费时间
    }
    else if (prev_alloc && !next_alloc) //只有后块未分配
    {
        size += GET_SIZE(HDRP(NEXT_BLKPTR(bp))); //将两个块的大小之和拿到
        PUT(HDRP(bp), PACK(size, 0)); //更新bp的头部
        PUT(FTRP(bp), PACK(size, 0)); //再更新脚部（该处设计较为精妙，FTRP本身“调用”了HDRP，改完头部就可以直接改脚部了
    }
    else if (!prev_alloc && next_alloc) //只有前块未分配
    {
        size += GET_SIZE(HDRP(PREV_BLKPTR(bp))); //将两块大小之和拿到
        PUT(FTRP(bp), PACK(size, 0)); //先改脚部，因为bp相比于前块在后面
        PUT(HDRP(PREV_BLKPTR(bp)), PACK(size, 0)); //此时不得不“调用”PREV_BLKPTR得到前块地址，再来改前块的size大小
        bp = PREV_BLKPTR(bp); //将bp更新为前块
        //注意，此处并没有处理bp原始的头部和前块的脚部（上面一种情况和下面一种情况同理），是因为合并后的块会直接跳到下一个块，该部分不再会被这个链表访问，即外面看是删除了，里面的数据实际还在，只不过无法到达
    }
    else //都未分配，全部要合并
    {
        size += GET_SIZE(HDRP(PREV_BLKPTR(bp))) +
            GET_SIZE(FTRP(NEXT_BLKPTR(bp))); //得到三块大小之和
        PUT(HDRP(PREV_BLKPTR(bp)), PACK(size, 0)); //改前块的头部的大小
        PUT(FTRP(NEXT_BLKPTR(bp)), PACK(size, 0)); //改后块的大小
    }
}
```

```

        bp = PREV_BLKp(bp); //更新bp为前块
    }
    return bp;
}

```

mm_alloc 函数:

```
void* mm_malloc(size_t size) //分配size大小的空间
```

```

{
    size_t asize; //实际要分配的空间(对齐要求)
    size_t extendsize; //堆要增长的大小
    char* bp;
    if (size == 0)
        return NULL; //分配空间为零, 不分配

```

//下面这个if else是为了确定实际分配的大小

```

    if (size <= DSIZE)
        asize = 2 * DSIZE;
    else

```

asize = DSIZE * ((size + (DSIZE)+(DSIZE - 1)) / DSIZE); //size加上双字(头部和脚部)再加上补偿(类似于表示浮点数的那个补偿)除以双字, 得到最少要多少个双字, 最后乘以双字得到总的字节数

if ((bp = find_fit(asize)) != NULL) //如果堆中有适配的空闲块, 代码分析见place函数的下面

```

{
    place(bp, asize); //该函数将大小为asize的块设为已占用, 代码分析见下面
    return bp; //返回分配好的指针
}

```

extendsize = MAX(asize, CHUNKSIZE); //未找到适配块, 判断堆需要增长的大小

```

if ((bp = extend_heap(extendsize / WSIZE)) == NULL) //如果增长堆失败
    return NULL;

```

```

place(bp, asize); //成功增长后, 将大小为asize的块设为已占用
return bp; //返回分配好的指针
}

```

Place 函数: //自己实现的版本

```
static void place(void* bp, size_t asize)
```

```

{
    size_t whole_size = GET_SIZE(HDRp(bp)); //whole_size表示bp这个块的实际大小
    if ((whole_size - asize) >= (2 * DSIZE)) //判断是否要对该空闲块进行分割(asize中不包含头部和脚部, 所以这里是2*DSIZE, 一个DSIZE是头部加脚部, 另一个DSIZE是填充)
    {
        //要分割

```

```

        PUT(HDRP(bp), PACK(asize, 0x1)); //更新头部
        PUT(FTRP(bp), PACK(asize, 0x1)); //更新脚部
//该处的设计和合并函数coalesce中只有后块未分配的情况一样，利用FTRP“调用”HDRP的特性
        bp = FTRP(bp) + DSIZE; //脚部加上一个双字(bp的脚部大小和下一个块的头部大小)，得到下一个块的bp
        PUT(HDRP(bp), PACK(whole_size - asize, 0x0)); //更新头部
        PUT(FTRP(bp), PACK(whole_size - asize, 0x0)); //更新脚部
    }
    else
    {
        //不用分割，直接更新即可
        PUT(HDRP(bp), PACK(whole_size, 0x1)); //更新头部
        PUT(FTRP(bp), PACK(whole_size, 0x1)); //更新脚部
    }
}

```

Find_fit函数: //自己实现的版本

```

static void* find_fit(size_t asize) //该分配函数采用首次适配
{
    void* bp = heap_listp; //从链表的头开始遍历
    while (GET_SIZE(HDRP(bp)) > 0) //当游走指针尚未到达结尾块时继续遍历(结尾块的意义!)
    {
        if (!GET_ALLOC(HDRP(bp)) && (GET_SIZE(HDRP(bp)) >= asize)) //找到了适配的块(两个判断条件，是否已分配和大小是否足够)
        {
            return bp;
            bp = NEXT_BLK(bp); //更新bp为下一个块
        }
    }
    return NULL; //没有找到合适的，返回空指针
}

```

mm_free函数:

```

void mm_free(void* ptr)
{
    size_t size = GET_SIZE(HDRP(ptr)); //得到该块的大小

//下面这两句在我的优化版本里是没有的，可以将这两句放coalesce里处理
    PUT(HDRP(ptr), PACK(size, 0)); //更新该块头部的状态
    PUT(FTRP(ptr), PACK(size, 0)); //更新该块脚部的状态

    coalesce(ptr); //判断是否合并
}

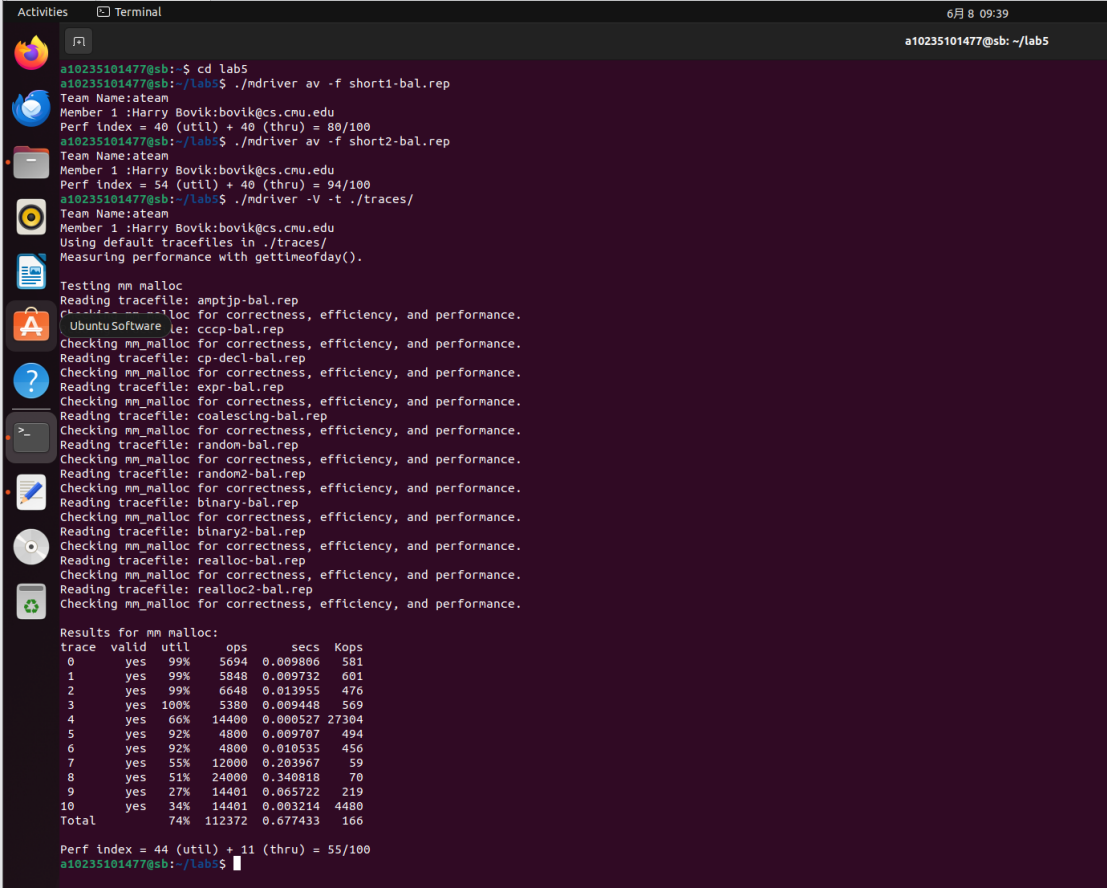
```

mm_realloc 函数:

```
void* mm_realloc(void* ptr, size_t size)
{
    void* oldptr = ptr; //旧指针的位置
    void* newptr; //创建一个新指针
    size_t copySize; //复制内存的大小

    newptr = mm_malloc(size); //先开辟新区域
    if (newptr == NULL) //无法分配
        return NULL;
    copySize = *(size_t*)((char*)oldptr - SIZE_T_SIZE); //拿到除头部和脚部的大小(有效荷载加填充)
    if (size < copySize) //重新分配的位置更小
        copySize = size;
    memcpy(newptr, oldptr, copySize); //将内容拷贝进去
    mm_free(oldptr); //释放旧指针
    return newptr; //返回新指针
}
```

原书版本的跑分:

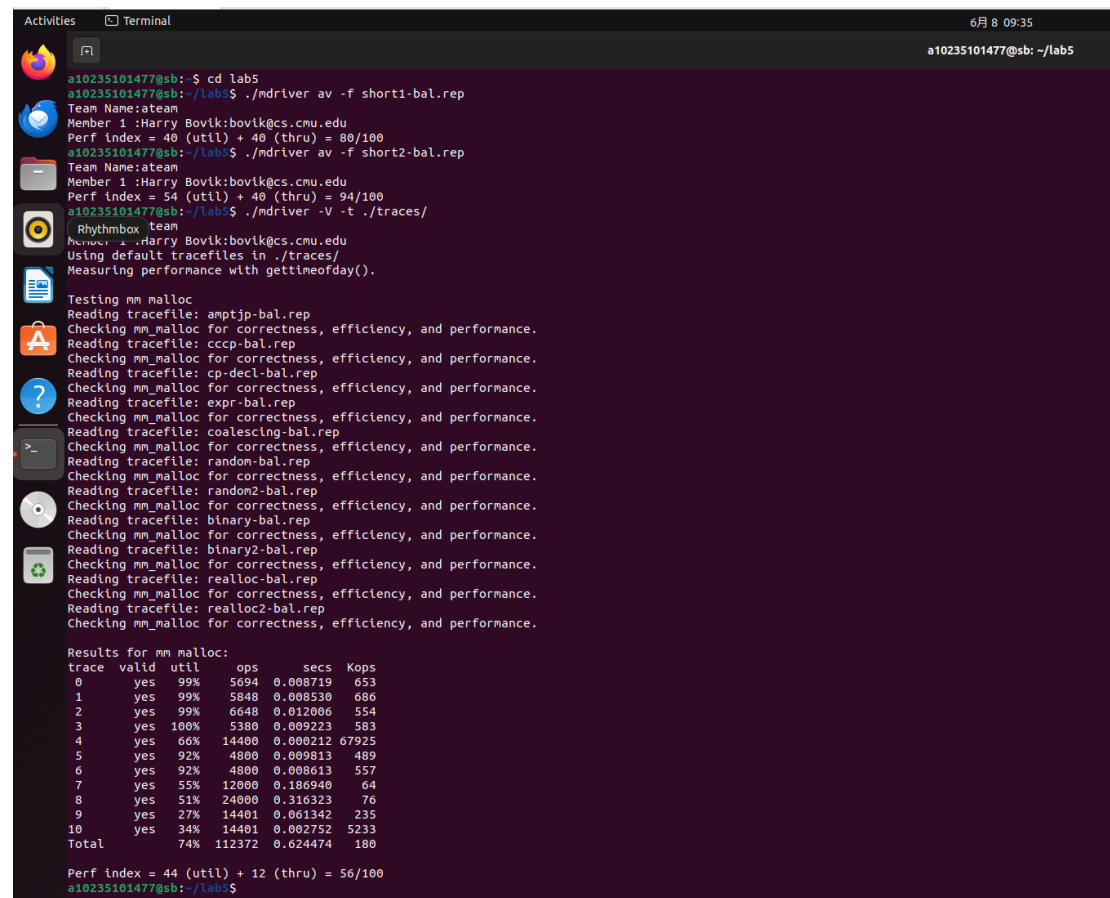


The terminal window shows the execution of various benchmarks for mm_malloc. The tests include amtpj-bal.rep, cccp-bal.rep, cp-decl-bal.rep, expr-bal.rep, coalescing-bal.rep, random-bal.rep, random2-bal.rep, binary-bal.rep, binary2-bal.rep, and realloc-bal.rep. Each test checks for correctness, efficiency, and performance. The results for mm_malloc are summarized in the following table:

| trace | valid | util | ops | secs | Kops |
|-------|-------|------|--------|----------|-------|
| 0 | yes | 99% | 5694 | 0.009806 | 581 |
| 1 | yes | 99% | 5848 | 0.009732 | 601 |
| 2 | yes | 99% | 6648 | 0.013955 | 476 |
| 3 | yes | 100% | 5380 | 0.009448 | 569 |
| 4 | yes | 66% | 14400 | 0.006527 | 27304 |
| 5 | yes | 92% | 4800 | 0.009707 | 494 |
| 6 | yes | 92% | 4800 | 0.010535 | 456 |
| 7 | yes | 55% | 12000 | 0.203967 | 59 |
| 8 | yes | 51% | 24000 | 0.340818 | 70 |
| 9 | yes | 27% | 14401 | 0.065722 | 219 |
| 10 | yes | 34% | 14401 | 0.003214 | 4480 |
| Total | | 74% | 112372 | 0.677433 | 166 |

Perf index = 44 (util) + 11 (thru) = 55/100

我优化版本的跑分：



```
Activities Terminal 6月 8 09:35 a10235101477@sb: ~/lab5
a10235101477@sb:~$ cd lab5
a10235101477@sb:~/lab5$ ./ndriver av -f short1-bal.rep
Team Name:ateam
Member 1 :Harry Bovik:bovik@cs.cmu.edu
Perf Index = 40 (util) + 40 (thru) = 80/100
a10235101477@sb:~/lab5$ ./ndriver av -f short2-bal.rep
Team Name:ateam
Member 1 :Harry Bovik:bovik@cs.cmu.edu
Perf Index = 54 (util) + 40 (thru) = 94/100
a10235101477@sb:~/lab5$ ./ndriver -V -t ./traces/
Rhythmbox team
Member 1 :Harry Bovik:bovik@cs.cmu.edu
Using default tracefiles in ./traces/
Measuring performance with gettimeofday().

Testing mm_malloc
Reading tracefile: amptjp-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: cccp-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: cp-decl-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: expr-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: coalescing-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: random-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: random2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: binary-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: binary2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: realloc-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.
Reading tracefile: realloc2-bal.rep
Checking mm_malloc for correctness, efficiency, and performance.

Results for mm_malloc:
trace valid util ops secs Kops
0 yes 99% 5694 0.008719 653
1 yes 99% 5848 0.008530 686
2 yes 99% 6648 0.012006 554
3 yes 100% 5380 0.009223 583
4 yes 66% 14400 0.000212 67925
5 yes 92% 4800 0.009813 489
6 yes 92% 4800 0.008613 557
7 yes 55% 12000 0.186940 64
8 yes 51% 24000 0.316323 76
9 yes 27% 14401 0.061342 235
10 yes 34% 14401 0.002752 5233
Total 74% 112372 0.624474 180

Perf index = 44 (util) + 12 (thru) = 56/100
a10235101477@sb:~/lab5$
```

后记：

本次实验涉及大量的指针操作，当时我想不用书上的宏做(相当于要接受他的思路)，但是当我发现在不能定义任何结构体时(不像实现 cache 那样)，我才感受到这个实验空前的难度，只有几个静态变量，还不能是数组(网上的有些高分做法违规了，使用了链表的数组实现分离适配)，后面结合时间的问题，便还是没有自己想出方案来，只是对原书的做法进行了自己的优化，虽然没有自己实现 allocator，其中隐式空闲链表的实现方式是彻底理解了，包括设置序言块和结尾块的意义(当时我觉得设置这两块完全是多此一举，不过分析完代码后成小丑了)，此外，还增加了卡常的能力？那一分的增加是我完全没想到的，就试了一下，没想到确实快了。