

# 操作系统实验-动态内存分配器

PB18151866 龚小航

## 一、实验名称

实验三：动态内存分配器 malloc 的实现

## 二、实验目的

- 使用隐式空闲链表实现一个 32 位堆内存分配器
- 掌握 makefile 的基本写法
- 使用显式空闲链表实现一个 32 位堆内存分配器

## 三、实验平台

Ubuntu 18.04.4 LTS  
LINUX 0.11

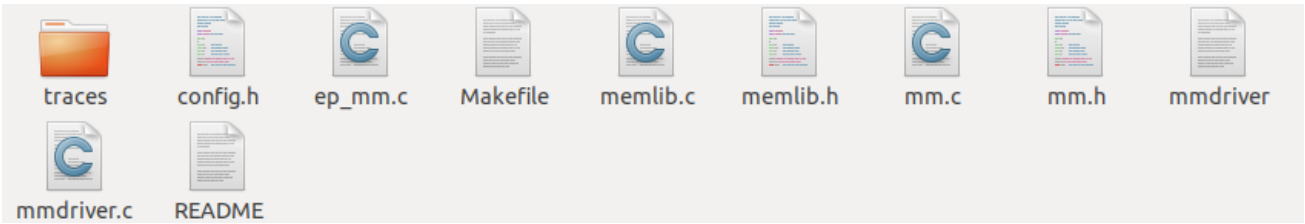
## 四、实验过程与结果

### 1. 实现隐式空闲链表

这一部分需要补全给出的代码，有三个函数需要补全。

- o static void \*find\_fit(size\_t asize)
  - 针对某个内存分配请求，该函数在隐式空闲链表中执行首次适配搜索。
  - 参数asize表示请求块的大小。返回值为满足要求的空闲块的地址。
  - 若为NULL，表示当前堆块中没有满足要求的空闲块。
- o static void place (void \*bp, size\_t asize)
  - 该函数将请求块放置在空闲块的起始位置。
  - 只有当剩余部分大于等于最小块的大小时，才进行块分割。
  - 参数bp表示空闲块的地址。参数asize表示请求块的大小。
- o static void \*coalesce(void \*bp)
  - 释放空闲块时，判断相邻块是否空闲，合并空闲块
  - 根据相邻块的的分配状态，有如下四种不同情况（具体参见合并步骤这一节）
  - 需补充第四种情况：前后块都空闲

这些待补全函数位于 mm.c 文件中。本次实验通过软件模拟操作系统的内存分配，支持文件如下所示：



其中 traces 文件夹包含了两组测试数据；.h 文件和 memlib.c 是支持文件，模拟系统内存空间申请和堆栈的管理；mm.c 和 ep\_mm.c 是隐式空闲链表和显式空闲链表的实现；mmdriver.c 是测试代码。

进入文件 mm.c, 先实现函数 find\_fit，实现首次适配：

```
static void *find_fit(size_t asize)
{

    /* 待补全
     * @return 第一个大小合适的空闲内存块堆栈地址
     */

    char *p = heap_listp;          /* p 指向链表头*/
    size_t size = GET_SIZE(HDRP(p)); /*size 表示 p 指向的块的大小*/
    p = NEXT_BLK(p);               /*第一块*/
    size = GET_SIZE(HDRP(p));

    while(size){                   /*尾块大小是 0，如果到了尾块就退出 while 循环*/
        if(GET_ALLOC(HDRP(p))){ /*如果已经分配，就需要向后继续找*/
            p = NEXT_BLK(p); /*p 指向下一块，即判断下一块是否够用*/
            size = GET_SIZE(HDRP(p));
        }
        else{                      /*没有被分配，下一步判断大小是否符合要求*/
            if(size >= asize){
                return p;          /*找到合适的块，首次适应匹配完成*/
            }
            else{                  /*块未被使用，但不够大*/
                p = NEXT_BLK(p);
                size = GET_SIZE(HDRP(p));
            }
        }
    }
    return NULL;                  /*匹配失败条件：到尾块时，都没有找到合适的块*/
}
```

先需要声明一个指向链表头（序言块）的指针 p，通过 p 的移动来确定可以插入请求块的位置。p 的移动通过 while 循环内的 p = NEXT\_BLK(p) 实现, 在开始循环前先将 p 指向第一块非序言块的块，从这里开始循环判断即可。

由于尾块 size=0 因此若循环执行至尾块，则执行 return NULL，说明没有可以分配的块。具体的注释标于代码之中。

再是实现 place 函数，将需要存入内存的块按首次适配得到的地址存入堆中，若有剩余空间，则进行块的分割：

```
static void place(void *bp, size_t asize)
{
    const size_t total_size = GET_SIZE(HDRP(bp));
    size_t rest = total_size - asize;

    if (rest >= MIN_BLK_SIZE)    /*剩余部分过大时，执行的操作*/
        /* need split */
        /* 待补全 */

        PUT(HDRP(bp),PACK(asize,1));          /*把块的大小调整至 asize 并标记为已分配*/
        PUT(FTRP(bp),PACK(asize,1));

        PUT(HDRP(NEXT_BLK_P(bp)),PACK(rest,0)); /*剩下的部分标记为未分配*/
        PUT(FTRP(NEXT_BLK_P(bp)),PACK(rest,0));

    }
    else
        /* 待补全 */

        PUT(HDRP(bp),PACK(total_size,1)); /*不需要分块，直接标记已分配*/
        PUT(FTRP(bp),PACK(total_size,1));

    }
}
```

需要补全的部分是 if 语句。这个 if 语句是判断是否有剩余块，即是否需要分割。if 成立时，需要进行分块：即将首次适配传过来的地址代表的块大小置为需要的大小 asize，并标记已分配，再将剩余部分大小标为 rest，标记为未分配即可；if 不成立时，不需要进行分割，直接将 total 标记为已分配即可。

再是函数 coalesce 的实现，该函数合并空闲块。在释放空间时，某块被释放为空闲块，可能出现四种情况，即这一块的前后块是否为空闲块。因此只需对这四种情况分别实现即可：

```
static void *coalesce(void *bp)
{
    size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLK_P(bp)));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLK_P(bp)));
    size_t size = GET_SIZE(HDRP(bp));
    if (prev_alloc && next_alloc)
    {
        return bp;
    }
    else if (prev_alloc && !next_alloc)    /*前两种情况都不需要改变bp*/
    {
        size += GET_SIZE(HDRP(NEXT_BLK_P(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));
    }
    else if (!prev_alloc && next_alloc)
    {
        size += GET_SIZE(FTRP(PREV_BLK_P(bp)));
        PUT(HDRP(PREV_BLK_P(bp)), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));
        bp = PREV_BLK_P(bp);
    }
    else
    { /* 待补全 */    /*第四种情况，前后两个块都空闲，合并空闲块*/
        size += GET_SIZE(FTRP(PREV_BLK_P(bp)))+GET_SIZE(HDRP(NEXT_BLK_P(bp)));
        PUT(HDRP(PREV_BLK_P(bp)), PACK(size, 0));
        PUT(FTRP(NEXT_BLK_P(bp)), PACK(size, 0));
        bp = PREV_BLK_P(bp);
    }
    return bp;
}
```

需要补充的情况就是第四种，即前后两块均为空闲块的情况。合并后的新块大小为三块之和，因此 size=三块大小之和。再将合并的新块头部和脚部都标注为未分配，最后再返回新块的起始地址PREV\_BLK\_P(bp)即可。

## 2. 实现显式空闲链表

显示空闲链表需要补充两个函数的实现：

- static void \*find\_fit(size\_t asize)
- static void place(void \*bp, size\_t asize)

这两个函数均处于ep\_mm.c文件中。这两个函数的功能与隐式空闲链表相同，均为首次适配结果以及将需求块存入堆中并进行分割。

首先是实现函数 find\_fit，进行首次适配：

```
static void *find_fit(size_t asize)
{
    char *bp = free_listp;
    if (free_listp == NULL)
        return NULL;

    while (bp != NULL) /*not end block;*/
    {
        /* ??? */
        size_t size = GET_SIZE(HDRP(bp));
        if (size >= asize) /*找到了合适的块，返回*/
            break;
        else /*这一块不合适，继续找下一块*/
            bp = GET_SUCC(bp);
    }
    return (bp != NULL ? ((void *)bp) : NULL);
}
```

需要补充的部分就是 while 内的部分。通过分析，可知要补充的就是寻找合适块的过程。因此只需要一个判断语句，合适则返回，不合适则找下一块。

最后是实现函数 place：

```
static void place(void *bp, size_t asize)
{
    size_t total_size = 0;
    size_t rest = 0;
    delete_from_free_list(bp);
    /*remember notify next_blk, i am allocated*/
    total_size = GET_SIZE(HDRP(bp));
    rest = total_size - asize;

    if (rest >= MIN_BLK_SIZE) /*need split*/ /*这种情况需要分割*/
    {
        /* ??? */
        size_t prev_alloc = GET_PREV_ALLOC(HDRP(bp)); /*表示上一块的分配情况*/
        PUT(HDRP(bp), PACK(asize, prev_alloc, 1)); /*将asize大小的块标记为可分配*/
        PUT(HDRP(NEXT_BLKP(bp)), PACK(rest, 1, 0)); /*修改空闲块的信息*/
        PUT(FTRP(NEXT_BLKP(bp)), PACK(rest, 1, 0));
        add_to_free_list(NEXT_BLKP(bp)); /*把空闲块加入显式空闲链表*/
    }
    else /*不需要分块的情况*/
    {
        /* ??? */
        size_t prev_alloc = GET_PREV_ALLOC(HDRP(bp));
        size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
        size_t next_size = GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(NEXT_BLKP(bp)), PACK(next_size, 1, next_alloc)); /*修改下一块的上块分配信息*/
        if (!next_alloc) /*如果下一块是空块，需要修改尾部*/
            PUT(FTRP(NEXT_BLKP(bp)), PACK(next_size, 1, next_alloc));
        PUT(HDRP(bp), PACK(total_size, prev_alloc, 1)); /*直接将当前块大小定义为total_size*/
    }
}
```

这个函数需要补充的就是 if 语句内的内容。

if 成立即需要分割：将 asize 大小的块声明为已分配，将 rest 部分的多余块加入显式空闲链表。分割就需要分出新的空闲块信息，需要知道上一块是否已经分配。

if 不成立时，不需要分割：直接将当前块分配为 total\_size，还需修改下一块的上块分配信息。

### 3. 编辑 makefile 文件，生成可执行文件并执行

补完实现代码后，再通过 makefile 文件声明所有的.c 源文件和.h 头文件之间的链接关系。完成 makefile 文件后，只需在该目录下执行 make 命令，就能生成目标二进制可执行文件。

```
CC = gcc -g
CFLAGS = -Wall

.c.o:
    @$(CC) $(CFLAGS) -c -o $*.o $<

# 待补充
OBJ1 = memlib.o mm.o mmdriver.o
OBJ2 = memlib.o ep_mm.o mmdriver.o

all:mmdriver epmmdriver

mmdriver: $(OBJ1)
    $(CC) $(CFLAGS) -o $@ $(OBJ1)

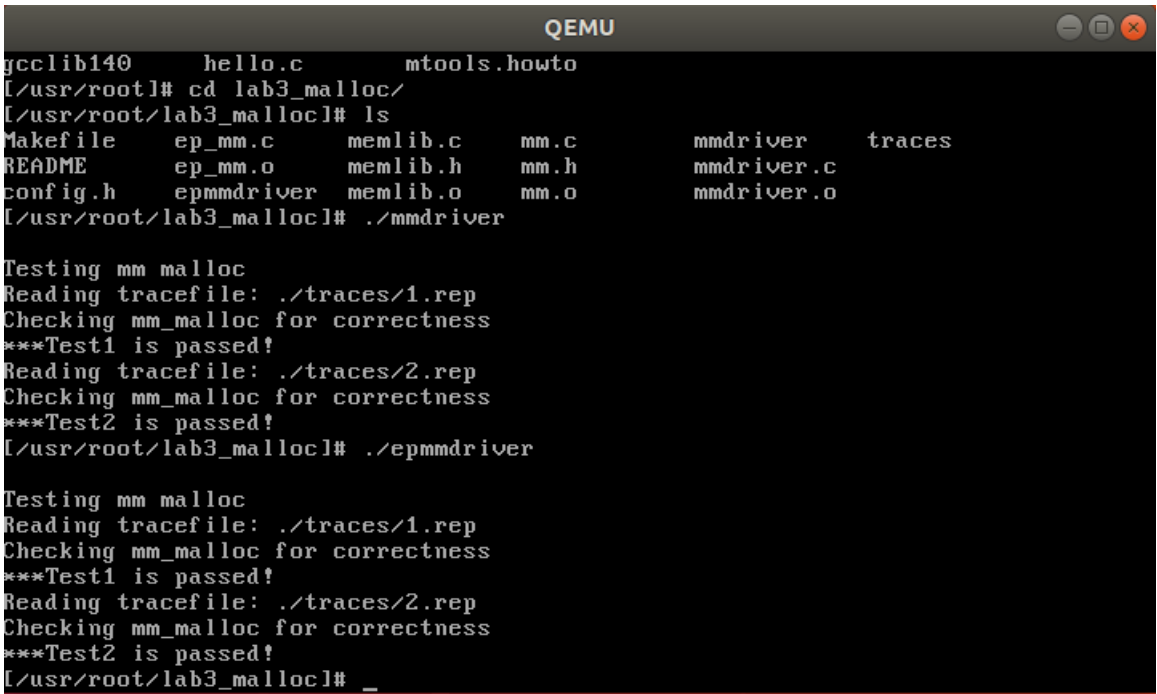
epmmdriver:$(OBJ2)
    $(CC) $(CFLAGS) -o $@ $(OBJ2)

#待补充
mmdriver.o: mmdriver.c mm.h memlib.h
memlib.o: memlib.c memlib.h config.h
mm.o: mm.c mm.h memlib.h
ep_mm.o: ep_mm.c mm.h memlib.h

clean:
    rm -f *~ *.o mmdrive epmmdriver
```

声明的 gcc 变量表示.o 文件严格按照.c 扩展。补充部分需要说明目标文件用到哪些 c 源文件，然后在下方声明它们与哪些文件相关联。

将整个源码文件夹复制到挂载目录的 hdc/usr/root 目录下，再卸载挂载分区，启动 Linux-0.11 的 qemu 模拟器，进入源码文件夹，执行 make 指令生成可执行文件，再运行它们，观察运行结果，如下所示：



```
QEMU
gcc lib140 hello.c mtools.howto
[/usr/root]# cd lab3_malloc/
[/usr/root/lab3_malloc]# ls
Makefile ep_mm.c memlib.c mm.c mmdriver traces
README ep_mm.o memlib.h mm.h mmdriver.c
config.h epmmdriver memlib.o mm.o mmdriver.o
[/usr/root/lab3_malloc]# ./mmdriver

Testing mm malloc
Reading tracefile: ./traces/1.rep
Checking mm_malloc for correctness
***Test1 is passed!
Reading tracefile: ./traces/2.rep
Checking mm_malloc for correctness
***Test2 is passed!
[/usr/root/lab3_malloc]# ./epmmdriver

Testing mm malloc
Reading tracefile: ./traces/1.rep
Checking mm_malloc for correctness
***Test1 is passed!
Reading tracefile: ./traces/2.rep
Checking mm_malloc for correctness
***Test2 is passed!
[/usr/root/lab3_malloc]# _
```

可见结果符合预期。