实验三

动态内存分配器malloc的实现

一、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(进阶)

二、实验环境

- OS：Ubuntu 18.04LTS
- Linux内核版本：Kernel 0.11
- 需要工具：gcc qemu （本次实现以实验一为基础，如有不熟悉的步骤，请浏览实验一文档）

  注意： 1、在Linux 0.11 环境下编程，注释 **仅支持 /* 注释内容 */**，**不支持 // 注释内容**

  2、Linux 0.11 局部变量必须在**函数最开始**就声明，否则会报错

三、实验要求

1、实验任务

任务一：补全隐式空闲链表的实现

- 在下面文档讲解隐式空闲链表的过程中，已经提供并分析了隐式链表的代码实现，所以你只需要在mm.c 文件中**补全部分函数**即可，并在录屏时展示相关代码并简单解说。待补全的函数如下：
  - 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。
  - 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。
  - 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
- 编写Makefile文件编译运行
  - 在mm.c 中补全代码，trace目录下为测试用例
  - 修改Makefile文件
  - 使用Makefile文件编译项目，main函数在mmdriver.c文件中，主要用于测试
  - 使用以下命令编译运行程序

```
#进入源代码目录
#编译，执行后可以看到生成了一个名为mdriver的可执行文件
$ make mmdriver
$ ./mmdriver
```

任务二：显式空闲链表的实现 （进阶）

- 需要在ep_mm.c 文件中**补全部分函数**，并在录屏时展示相关代码并简单解说，待补全的函数如下：
  - static void *find_fit(size_t asize)
  - static void place (void *bp, size_t asize)
- 编写Makefile文件编译运行
  - 需要在ep_mm.c 中填入自己的代码，trace目录下为测试用例
  - 修改Makefile文件
  - 使用Makefile文件编译项目，main函数在mmdriver.c文件中，主要用于测试
  - 使用以下命令编译运行程序

```
#进入源代码目录
#编译，执行后可以看到生成了一个名为mdriver的可执行文件
$ make epmmdriver
$ ./epmmdriver
```

2、提交要求

- 本次实验要求提交**实验录屏、实验报告和代码**
- 按照下面描述的方式组织相关文件（具体的实验报告和录屏的要求见实验内容部分）

```
- 顶层目录（命名为lab3）
    - mm.c
    - ep_mm.c
    - Makefile
    - 实验录屏（5分钟之内）
    - 实验报告（学号+姓名+lab3,提交PDF版本）
```

- 将上述文件压缩
  - 格式为.7z/rar/zip
  - 命名格式为 学号_姓名_实验3，如果上传后需要修改，由于ftp服务器关闭了覆盖写入功能，需要将文件重新命名为学号_姓名_实验3_修改n (n为修改版本)，以后修改版本为准

3、上传视频至ftp服务器

- 服务器地址：ftp://OS2020:OperatingSystem2020@nas.colins110.cn:2001/
- 上传至文件夹: **第三次实验**
- 实验截止日期：2020-05-24 23:59

4、评分标准

- **得分由三部分构成**：
  1. 隐式链表实验代码运行结果（录屏和代码源文件）；
  2. 显示链表实验代码运行结果（录屏和代码源文件）；

3. malloc待补充代码解析实验报告(隐式和显示实现)。

- 隐式链表实现代码运行结果正确和实验代码解析报告完整，**最高可得7分**
- 显示链表实现代码运行结果正确和实验代码解析报告完整，**最高可得10分（进阶）**

四、实验内容

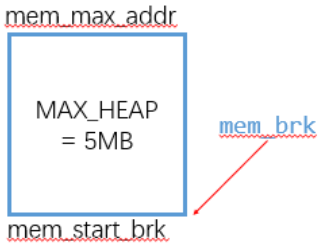
下载实验源码，解压后通过实验一的挂载硬盘镜像的方法，将代码文件夹复制到Linux 0.11中:

```
$ wget https://github.com/ZacharyLiu-CS/USTC_OS/raw/master/Lab3-Memory-Alloc/lab3_malloc.tar.gz
$ tar zxf lab3_malloc.tar.gz
```

1. API简介

(1) 本次实验中的相关代码简介

- memlib.c - 模拟内存系统的模块，主要通过libc中malloc分配一段内存空间，用于模拟系统内存空间申请空间并实现堆栈的管理
  - 主要变量如下所示：
  - ```
    /* private variables */
    static char *mem_start_brk;  /* points to first byte of heap */
    static char *mem_brk;        /* points to last byte of heap */
    static char *mem_max_addr;   /* largest legal heap address */
    ```
  - mem_init函数介绍：
    - 通过libc中malloc分配一段内存空间，主要用于模拟系统空间，空间大小为5MB，
    - 自己实现的malloc函数主要在模拟系统内存空间申请空间并实现堆栈的管理
    - 其中mem_brk指向自己实现的malloc空间的堆顶




```
void mem_init(void)
{
 /* allocate the storage we will use to model the available VM */
 if ((mem_start_brk = (char *)malloc(MAX_HEAP)) == NULL) {
     fprintf(stderr, "mem_init_vm: malloc error\n");
     exit(1);
 }

 mem_max_addr = mem_start_brk + MAX_HEAP;  /* max legal heap address */
 mem_brk = mem_start_brk;                  /* heap is empty initially */
 }
```

    - mem_brk函数介绍：模拟brk系统调用，用于扩展堆空间，具体用法见下节
```

```
void *mem_sbrk(int incr)
{
    char *old_brk = mem_brk;

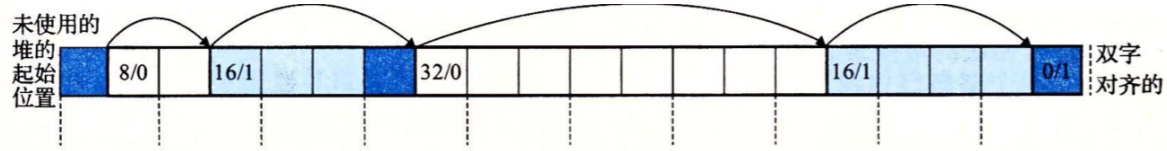
    if ( (incr < 0) || ((mem_brk + incr) > mem_max_addr)) {
        errno = ENOMEM;
        fprintf(stderr, "ERROR: mem_sbrk failed. Ran out of
memory...\n");
        return (void *)-1;
    }
    mem_brk += incr;
    return (void *)old_brk;
}
```

- mm.c，ep_mm.c：隐式和显式空闲链表的实现代码，在下文中展开介绍
- mmdriver.c: 测试相关代码

2. 隐式空闲链表管理

这里通过一种简单的堆内存管理方式，隐式空闲链表，为例来讲解内存分配器的实现。

隐式空闲链表将堆中的内存块按地址顺序串成一个链表，接受到内存分配请求时，分配器遍历该链表来找到合适的空 闲内存块并返回。当找不到合适的空闲内存块时（如：堆内存不足，或没有大小足够的空闲内存块），调用sbrk向堆顶扩展更多的内存。隐式空闲链表如下图所示：



图中淡蓝色部分为已分配块，深蓝色为填充块（为了内存双字对齐），数字为块头部，见下节。（图中链表指针指向块的起始处，实际实现时是指向块头部后面一个字，即有效载荷地址，见下节图）

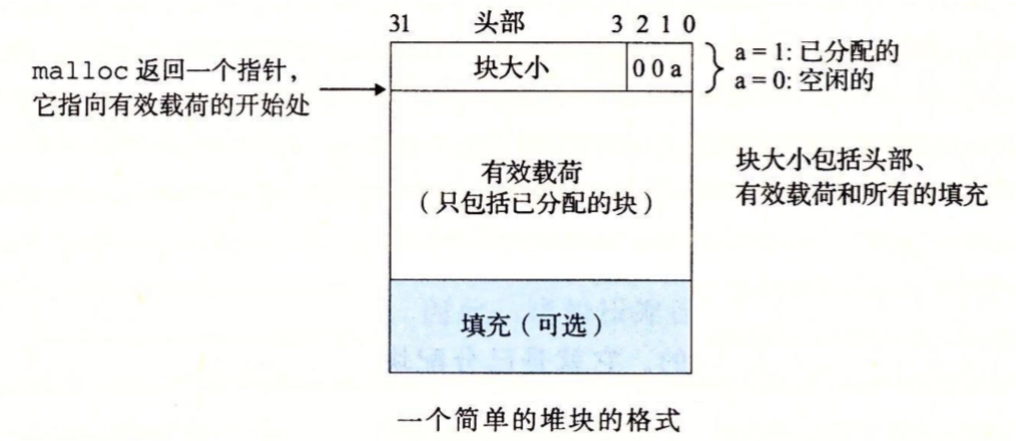
下面分析如何实现隐式空闲链表管理

2.1 块头部

堆中的各内存块需要某种标志来区分块的边界，记录块的大小，以及标记该内存块是否已被使用。因此为每个内存块保留一个字（4字节）的头部记录这些数据。

块头部记录了该内存块的大小。由于内存块以8字节对齐，块大小二进制的最低3位一定为0，因此可以用最后一位来标记该块是否已被分配。

综上，引入头部后一个内存块的格式如下图所示：



一个简单的堆块的格式

块大小包括头部在内，因此对于一个32字节的内存分配请求，考虑块头部以及内存对齐后，需要为其分配至少40字节的内存块。

其中分配器实现会涉及大量的指针操作，包括从地址取值/赋值，查找块头/块脚地址等。为了方便操作以及保障操作性能，定义如下宏操作。

```
/* Basic constants and macros */
#define WSIZE 4             /* Word and header/fonter size(bytes) */
#define DSIZE 8             /* Double word size(bytes) */
#define CHUNKSIZE (1 << 12)  /* Extend heap by this amount (bytes) */
#define MAX(x, y) ((x) > (y) ? (x) : (y))

/* Pack a size and allocated bit into a word */
#define PACK(size, alloc) ((size) | (alloc))

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

/* Read the size and allocated fields from address p */
#define GET_SIZE(p) (GET(p) & ~0x7)
#define GET_ALLOC(p) (GET(p) & 0x1)

/* Given block ptr bp, compute address of it's header and footer */
#define HDRP(bp) ((char *)(bp)-WSIZE)
#define FTRP(bp) ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)

/* Given block ptr bp, compute address of next and previous blocks */
#define NEXT_BLKP(bp) ((char *)(bp) + GET_SIZE(((char *)(bp)-WSIZE)))
#define PREV_BLKP(bp) ((char *)(bp)-GET_SIZE(((char *)(bp)-DSIZE)))
/* Used for the sp place() function */
#define MIN_BLK_SIZE (2 * DSIZE)
```

- 针对32位系统，定义字长位4byte（WSIZE）
- CHUNKSIZE为内存分配器扩充堆内存的最小单元
- PACK将块大小和分配位结合返回一个值（即将size的最低位赋值为分配位）
- GET/PUT分别对指针p指向的位置取值/赋值
- GET_SIZE/GET_ALLOC分别从p指向位置获取块大小和分配位。注意：p应该指向头/脚部
- HDRP/FTRP返回bp指向块的头/脚部
- NEXT_BLKP/PREV_BLKP返回与bp相邻的下一/上一块

2.2 初始化分配器

在开始调用malloc分配内存前，需要先调用mm_init函数初始化分配器，其主要工作是分配初始堆内存，分配序言块和尾块，以及初始化空闲链表，如下代码所示。

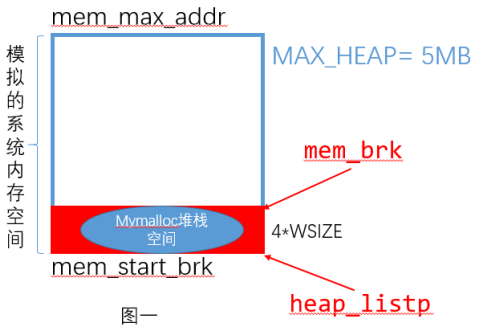
```
/*
 * mm_init - initialize the malloc package.
 */
int mm_init(void)
{
    /* 首先通过mem_sbrk请求4个字的内存(模拟sbrk) */
    if ((heap_listp = mem_sbrk(4 * WSIZE)) == (void *)-1)
        return -1;
    /* 调用mem_sbrk模拟系统空间和mymalloc堆栈变化见下面图一*/
    /*  这四个字分别作为填充块（为了对齐）
```

```
     *   序言块头/脚部，尾块
     *   并将heap_listp指针指向序言块使其作为链表的第一个节点
     */
    PUT(heap_listp, 0);
    PUT(heap_listp + (1 * WSIZE), PACK(DSIZE, 1));
    PUT(heap_listp + (2 * WSIZE), PACK(DSIZE, 1));
    PUT(heap_listp + (3 * WSIZE), PACK(0, 1));
    heap_listp += (2 * WSIZE);
    /* mymalloc堆栈指针heap_listp变化见下面图二*/
    /* Extend the empty head with a free block of CHUNKSIZE bytes (4Kb)
     * 调用extend_heap函数向系统申请一个CHUNKSIZE的内存作为堆的初始内存
     */
    if (extend_heap(CHUNKSIZE / WSIZE) == NULL)
        return -1;
    /* 调用extend_heap()模拟系统空间和mymalloc堆栈变化见下面图三*/
    return 0;
}
```

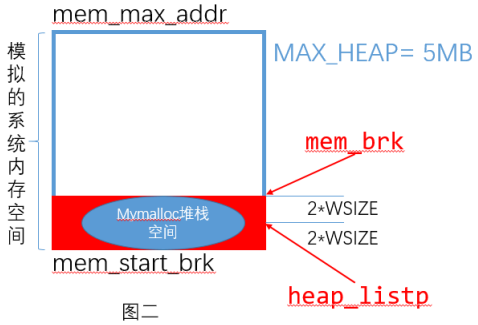
```
/* extend_heap函数是对mem_sbrk的一层封装，接收的参数是要分配的字数，
 * 在堆初始化以及malloc找不到合适内存块时使用。
 * 它首先对请求大小进行地址对齐，然后调用mem_sbrk获取空间
 */
static void *extend_heap(size_t words)
{
    char *bp;
    size_t size;
    /* Allocate an even number of words to maintain alignment */
    size = (words % 2) ? (words + 1) * WSIZE : words * WSIZE;
    if ((long)(bp = mem_sbrk(size)) == -1)
        return NULL;
    /* Initialize free block header/footer and the epilogue header */
    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1));

    /* Coalesce if the previous block was free */
    return coalesce(bp);
}
```

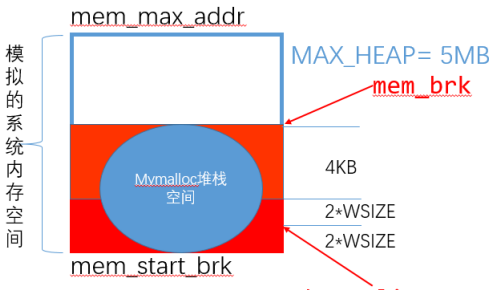
> 注意：这里实际操作是将扩展前的尾块作为了新空闲块的头块，然后在新的堆末尾分配一个新的尾块。



图一

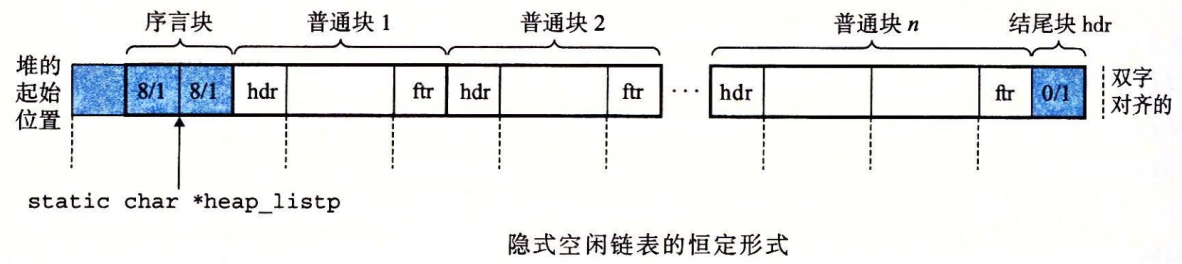


图二



图三

其中Mymalloc堆内存内部组织格式如下图：



隐式空闲链表的恒定形式

堆内存中第一个字是一个为了内存对齐的填充字。填充字后面紧跟一个特殊的序言块，它是一个8字节的已分配块， 只由一个头部和一个脚部组成。序言块是在初始化时创建的，并且永不释放。序言块后面是普通块。堆的最后一个字 是一个特殊的结尾块，它是一个有效大小为0的已分配块，只由一个头部组成。
序言块和结尾块的作用是消除空闲块合并时的边界检查，在后续代码中可以看到。
分配器使用一个私有全局变量heap_listp表示链表头，它总是指向序言块。

2.3 链表管理

在隐式链表管理方案下，分配器维护一个指针heap_listp指向堆中的第一个内存块，也即链表中的第一个块。

根据该内存块头部中记录的块大小信息便可计算出下一块的位置(heap_listp+size)，依此类推。具体代码参见上一节mm_init(void)函数

```
int mm_init(void)
{
    // 省略
    heap_listp += (2 * WSIZE);
    // 省略
}
```

2.4 放置策略

当请求一个k字节的内存块时，分配器需要搜索堆中的内存块找到一个足够大的空闲块并返回。具体选择哪一个内存块由放置策略决定。主要有两种：

- 首次适配。从头开始搜索链表，找到第一个大小合适的空闲内存块便返回。
- 最佳适配。搜索整个链表，返回满足需求的最小的空闲块。

两者相比较，首次适配速度较快，最佳适配内存利用率更高。后面的实现采用首次适配方法。

本次实验需要补充该函数的实现，并需要在实验报告分析代码思路

```
static void *find_fit(size_t asize)
{
    // 需补充
    // @return    第一个大小合适的空闲内存块堆栈地址
}
```

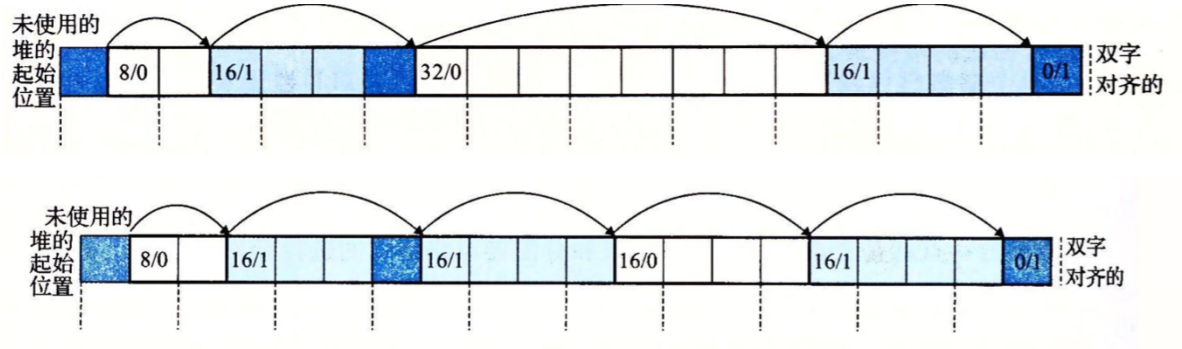
2.5 分割空闲块

当分配器找到一个合适的空闲块后，如果空闲块大小大于请求的内存大小，则需要分割该空闲块，避免内存浪费。

具体步骤为：

- 修改空闲块头部，将大小改为分配的大小，并标记该块为已分配。
- 为多余的内存添加一个块头部，记录其大小并标记为未分配，使其成为一个新的空闲内存块。
- 返回分配的块指针。

例如在如图堆中分配一个16字节的块会将中间32字节的块分割成两个



本次实验需要补充该函数的实现，并需要在实验报告分析代码思路

```
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)
    /*
    //need split
    */
    {
        /*  待补全 */
    }
    else
    {
        /*  待补全 */
    }
}
```

2.6 释放和合并块

当调用mm_free释放某个块后，如果该块相邻有其他的空闲块，则需要将这些块合并成一个大的空闲块，避免出现“假碎片”现象（多个小空闲块相邻，无法满足大块内存分配请求）。

```
/*
 * mm_free - Freeing a block does nothing.
 */
void mm_free(void *ptr)
{
    size_t size = GET_SIZE(HDRP(ptr));
    /*
     * mm_free首先将其该内存标记为未分配
     */
    PUT(HDRP(ptr), PACK(size, 0));
    PUT(FTRP(ptr), PACK(size, 0));
    /* 调用coalesce函数尝试合并相邻空闲块，具体见下节 判断相邻块是否空闲 分析 */
    coalesce(ptr);
}
```

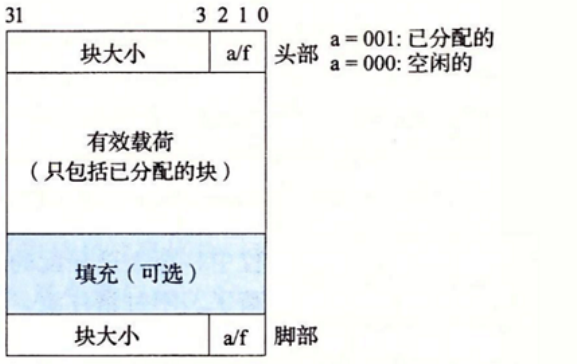
2.7 判断相邻块是否空闲

判断相邻的下一个块是否空闲很简单：根据当前块的大小即可计算出下一块的头部位置。

但是，对于相邻的前一个块，由于不知道其头部位置，只能从头开始遍历链表，这样性能很差。

解决这个问题的办法是，为每个块再维护一个脚部，内容为头部的复制。有了脚部以后，当前块头部地址向前4个字节便是前一个块的脚部，因此就可以快速地获取前一个块的元数据了。

添加脚部以后，块的格式如图所示：

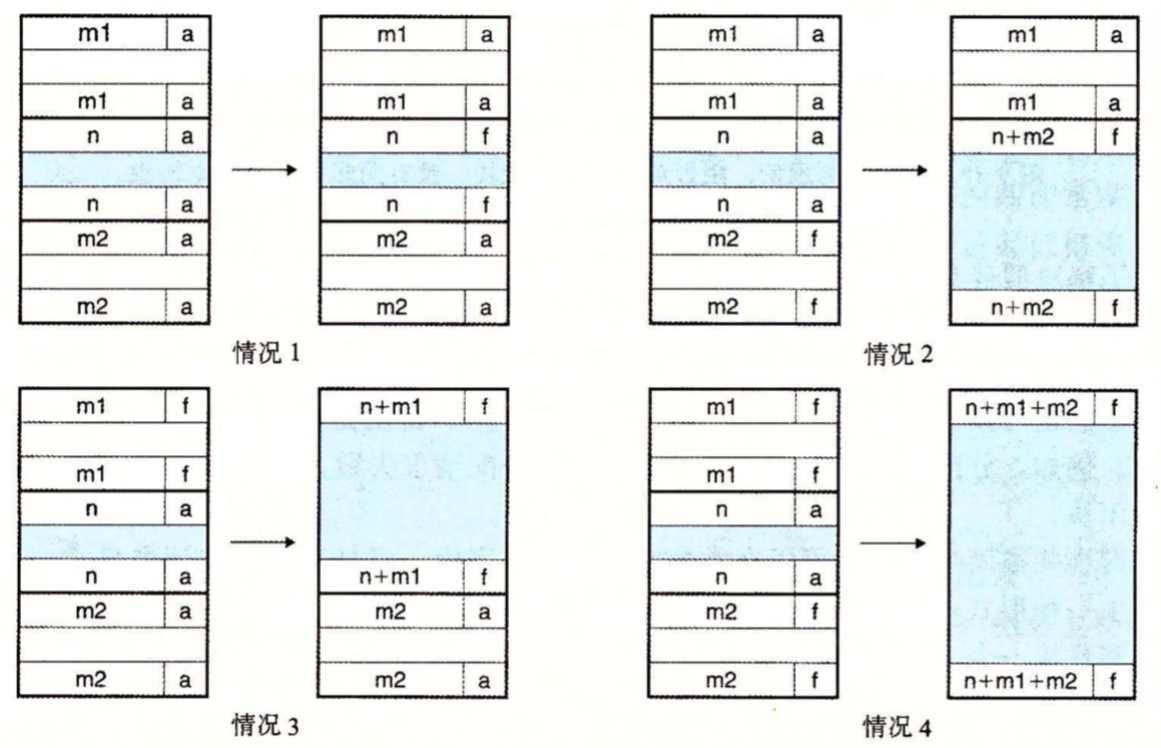


2.8 合并步骤

释放当前内存块时，根据相邻块的的分配状态，有如下四种不同情况：

1. 前面的块和后面的块都已分配
2. 前面的块已分配，后面的块空闲
3. 前面的块空闲，后面的块已分配
4. 前后块都空闲（本次实验需要补充该情况的实现，并需要在实验报告分析代码思路）

以下为这四种情况的合并前后示意图



图中m/n表示块大小，a表示已分配，f表示未分配。即根据合并结果修改当前块的头/脚部元数据。

coalesce函数首先从前一块的脚部后后一块的头部获取它们的分配状态，然后根据前文所述的4种不同情况作相应处理，最后返回合并后的指针。

由于序言块和尾块的存在，不需要考虑边界条件。

```
static void *coalesce(void *bp)
{
    size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    size_t size = GET_SIZE(HDRP(bp));
    /* 第一种情况 */
    if (prev_alloc && next_alloc)
    {
        return bp;
    }
    /* 第二种情况 */
    else if (prev_alloc && !next_alloc)
    {
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));
    }
    /* 第三种情况 */
    else if (!prev_alloc && next_alloc)
    {
        size += GET_SIZE(FTRP(PREV_BLKP(bp)));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));
        bp = PREV_BLKP(bp);
    }
    /* 第四种情况 */
    else
    {
        /*  待补充 */
```

```
    }
    return bp;
}
```

2.9 分配块

最后介绍mm_malloc函数，向堆申请size大小的内存并返回指针。

```
/*
 * mm_malloc - Allocate a block by incrementing the brk pointer.
 *     Always allocate a block whose size is a multiple of the alignment.
 */
void *mm_malloc(size_t size)
{
    size_t newsize;         /* Adjusted block size */
    size_t extend_size;     /* Amount to extend head if not fit */
    char *bp;
    /* Ignore spurious requesets */
    if (size == 0)
        return NULL;

    /* Adjust block size to include overhead and alignment reqs. */
    newsize = ALIGN(size) + DSIZE;

    /* Search the free list for a fit */
    if ((bp = find_fit(newsize)) != NULL)
    {
        place(bp, newsize);
        return bp;
    }
    /* no fit found. Get more memory and place the block */
    extend_size = MAX(newsize, CHUNKSIZE);
    if ((bp = extend_heap(extend_size / WSIZE)) == NULL)
    {
        return NULL;
    }
    place(bp, newsize);
    return bp;
}
```

首先将申请内存大小加上块头/尾部大小并进行对齐，然后调用find_fit函数(想想怎么实现)从内存块链表中找到合适的块，如果成功找到则调用place函数判断是否需要对该块作分割操作。

如果查找失败则向系统请求分配更多堆内存。为了避免频繁请求，一次最少申请CHUNKSIZE的内存

至此一个隐式链表管理方式的堆内存分配器实现完成。

2.10 测试

测试代码实现在mmdriver.c中，通过读取trace文件，生成一系列分配释放操作，验证所实现代码的正确性。

2.11 编写makefile文件编译并运行（需要录屏）

```
#
# Students' Makefile for the Malloc Lab
#
```

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

# 待补充
OBJS = xxxx.o

mmdriver: $(OBJS)
# 待补充gcc命令（使用变量）
    xxx

#待补充
mmdriver.o: xxxx.h
memlib.o: xxxx.h
mm.o: xxxx.h

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

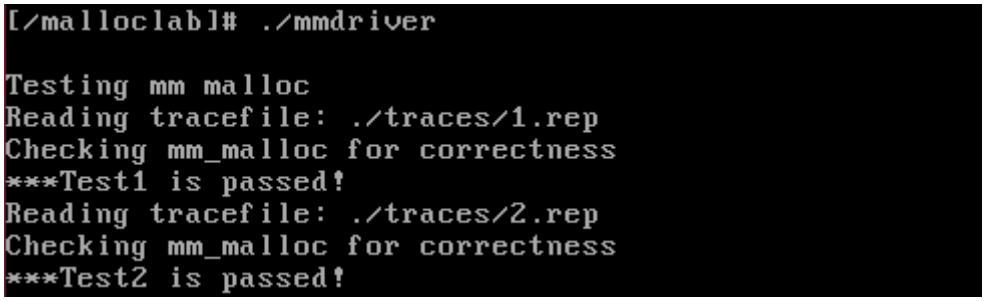
补充上述Makefile文件(参考下述学习资料)

- Makefile入门学习

编译项目并运行

```
$  make
$  ./mmdriver
```

见到如下结果表示运行成功：



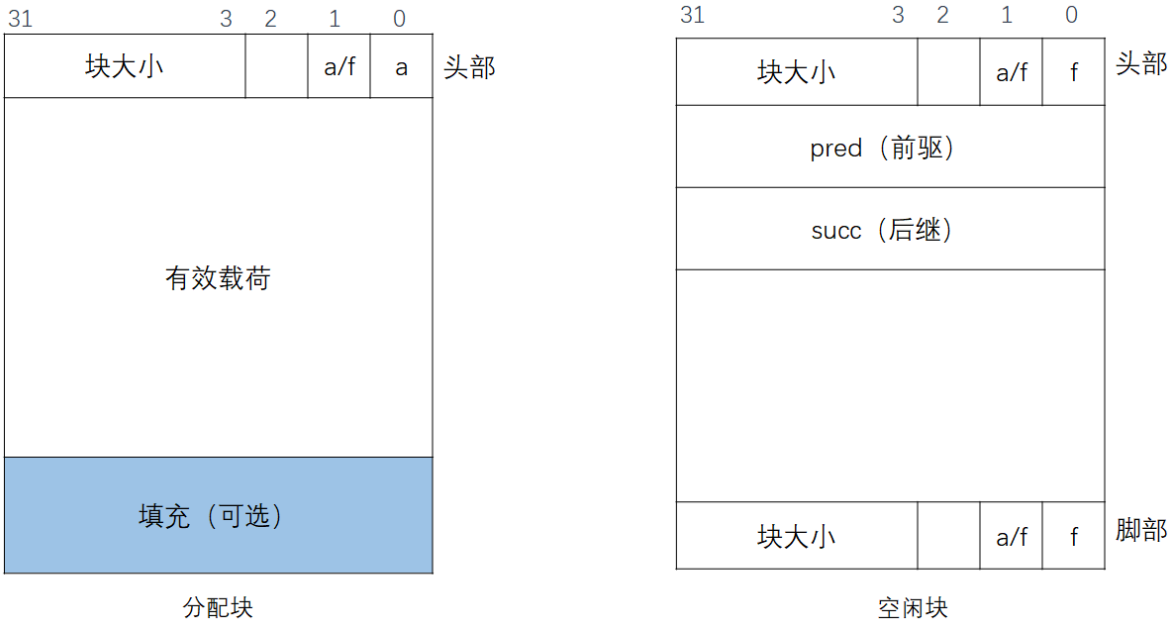
3. 显式空闲链表

- 隐式空闲链表存在的问题
    - 隐式空闲链表为我们提供了一种简单的分配方式。但是，在隐式空闲链表方案中：块分配时间复杂度与堆中块的总数呈线性关系。这在实际中是不能接受的。

下面介绍一种由双向链表组织的显式空闲链表方案。

3.1 块的格式

实际实现中通常将空闲块组织成某种形式的显式数据结构（如，链表）。由于空闲块的空间是不用的，所以实现链表的指针可以存放在空闲块的主体里。例如，将堆组织成一个双向的空闲链表，在每个空闲块中，都包含一个pred（前驱）和succ（后继）指针，如下图所示：



对比隐式空闲链表，双向空闲链表的方式使首次适配的分配时间由块总数的线性时间减少到空闲块数量的线性时间，因为它不需要搜索整个堆，而只是需要搜索空闲链表即可。

由上图可以看到，与隐式空闲链表相比，分配块和空闲块的格式都有变化。

- 首先体现在分配块没有了脚部，这可以优化空间利用率。回想前面的介绍，当进行块合并时，只有当前块的前面邻居块是空闲的情况下，才会使用到前邻居块的脚部。如果我们把前面邻居块的已分配/空闲信息位保存在当前块头部中未使用的低位中（比如第1位中），那么已分配的块就不需要脚部了。但是，一定注意：空闲块仍然需要脚部，因为脚部需要在合并时用到。
- 其次，空闲块中多了pred（前驱）和succ（后继）指针。正是由于空闲块中多了这两个指针，再加上头部、脚部的大小，所以最小的块大小为4字。

下面详细说明一下分配块和空闲块的格式

- 分配块：
  - 由头部、有效载荷部分、可选的填充部分组成。其中最重要的是头部的信息：
    - 头部大小为一个字(32 bits)，
    - 其中第3-31位存储该块大小的高位。（因为双字对齐，所以低三位都0）
    - 第0位的值表示该块是否已分配，0表示未分配（空闲块），1表示已分配（分配块）。
    - 第1位的值表示该块前面的邻居块是否已分配，0表示前邻居未分配，1表示前邻居已分配。
- 空闲块：
  - 由头部、前驱、后继、其余空闲部分、脚部组成。
    - 头部、脚部的信息与分配块的头部信息格式一样。
    - 前驱表示在空闲链表中前一个空闲块的地址。后继表示在空闲链表中后一个空闲块的地址。前驱和后继是组成空闲链表的关键。

3.2 空闲块的合并与分割

- 合并分割的思想与隐式空闲链表时的分析一致，只是在代码实现方式上不同。
- 注意：分割合并块时，一定要保证操作前和操作后所有块在空闲链表中不会互相覆盖。

3.3 显示链表实验要求（进阶: 3分）

- 补全find_fit()和place()函数代码,并在实验报告分析函数实现代码

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

```
     char *bp = free_listp;
    if (free_listp == NULL)
        return NULL;

    while (bp != NULL) /*not end block;*/
    {
        /* 待补全 */
    }
    return (bp != NULL ? ((void *)bp) : NULL);
}


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 alloced*/
    total_size = GET_SIZE(HDRP(bp));
    rest = total_size - asize;

    if (rest >= MIN_BLK_SIZE) /*need split*/
    {
        /* 待补全 */
    }
    else
    {
        /* 待补全 */
    }
}
```

- 在隐式空闲链表的基础上补全Makefile文件，编译运行（录屏）

```
#
# Students' Makefile for the Malloc Lab
#

CC = gcc -g
CFLAGS = -Wall

# 待补充
OBJS1 = xxxx.o
OBJS2 = xxxx.o

all: mmdriver epmmdriver

mmdriver: $(OBJS1)
# 待补充gcc命令（使用变量）
    xxx

epmmdriver:$(OBJS2)
# 待补充gcc命令（使用变量）
    xxx

#待补充
mmdriver.o: xxxx.h
memlib.o: xxxx.h
```

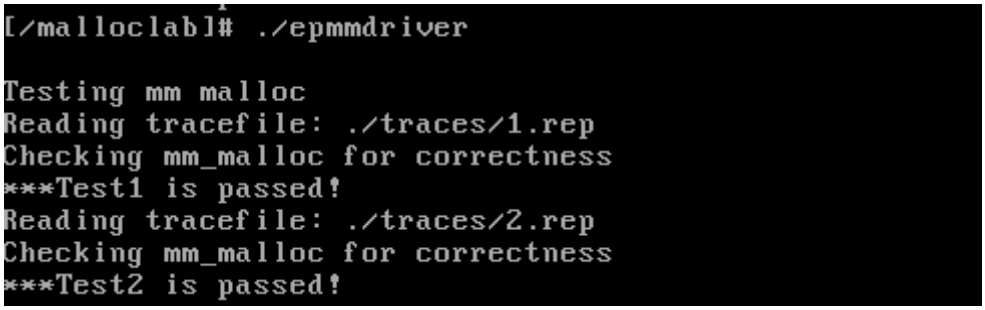
```
mm.o: xxxx.h
ep_mm.o: xxxx.h

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

编译项目并运行

```
$  make
$  ./epmmdriver
```

见到如下结果表示运行成功：



参考资料

- 《Computer Systems: A Programmer's Perspective 3rd》