OS 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

三、实验内容

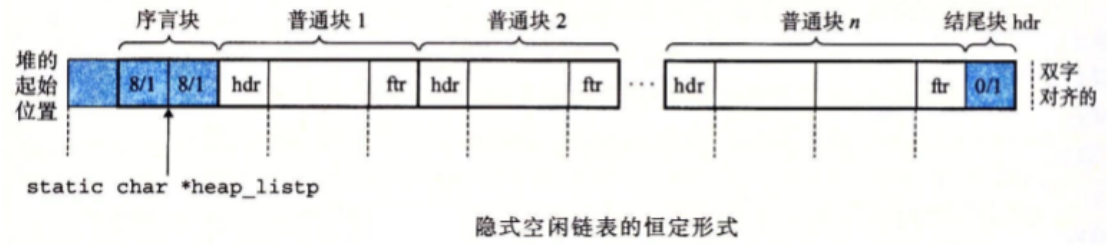
TASK1. 隐式空闲链表的实现

实验原理

隐式空闲链表将堆中的内存块按照地址顺序串成一个链表，接收到内存分配的请求是，分配器遍历链表来找到合适的空闲内存块并返回。

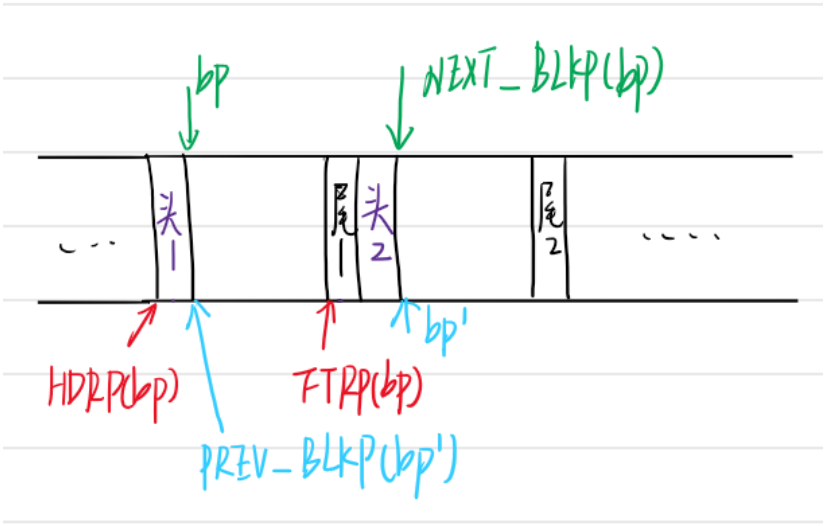
当找不到合适的空闲内存块时，比如堆内存不足，或者没有大小足够的空闲内存块，调用 sbrk 向堆顶扩展更多的内存。

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

宏定义操作的解释如下图：



关键函数代码如下：

a. 放置策略 -- 首次适配

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。

`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 *find_fit(size_t asize)
{
  /* @return 第一个大小合适的空闲内存块堆栈地址*/
  char *bp = heap_listp + DSIZE;
  while(GET_SIZE(HDRP(bp)) != 0) /*尾块边界条件*/
  {
    size_t alloc = GET_ALLOC(HDRP(bp));
    if(alloc == 0)
    {
      size_t size = GET_SIZE(HDRP(bp));
      if(size > asize || size == asize)
      {
        return bp;
      }
    }
    bp = NEXT_BLKP(bp);
  }
  return NULL;
}
```

b. 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：

1. 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。
2. 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。

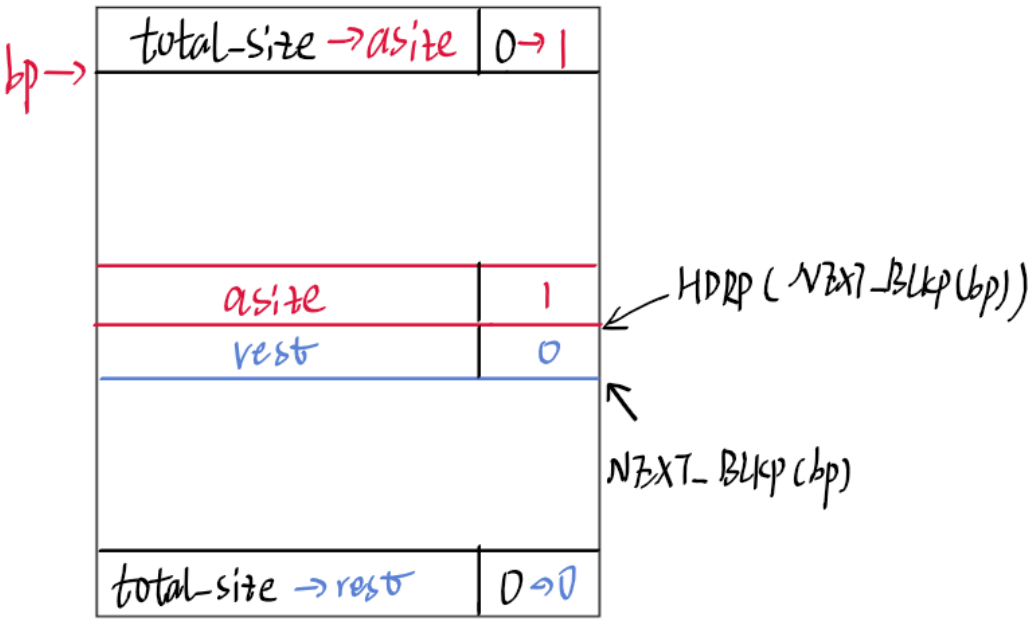
`static void place (void *bp, size_t asize)`

- 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 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 */
     {
       PUT(HDRP(bp),PACK(asize,1));
       PUT(FTRP(bp),PACK(asize,1));
       PUT(HDRP(NEXT_BLKP(bp)),PACK(rest,0));
       PUT(FTRP(NEXT_BLKP(bp)),PACK(rest,0));
     }
     else
     {
       PUT(HDRP(bp),PACK(total_size,1));
       PUT(FTRP(bp),PACK(total_size,1));
     }
     return bp;
 }
```

分割空闲块图示：



c. 判断相邻块是否空闲，若是，则合并空闲块

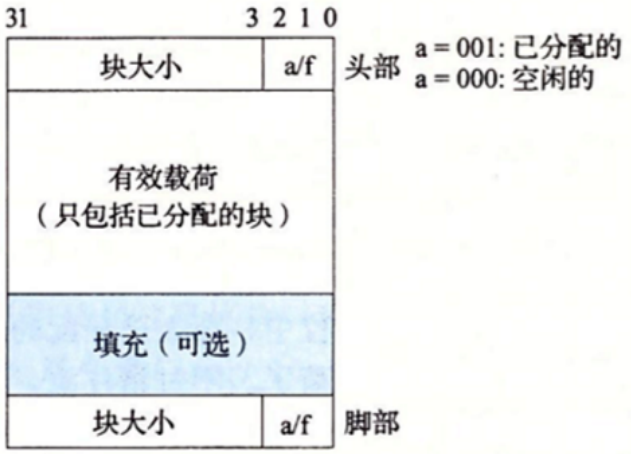
当调用 mm_free 释放某个块后，如果该块相邻有其他的空闲块，则需要将这些块合并成一个大的空闲块，避免出现"假碎片"现象（多个小空闲块相邻，无法满足大块内存分配请求）。

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：



`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

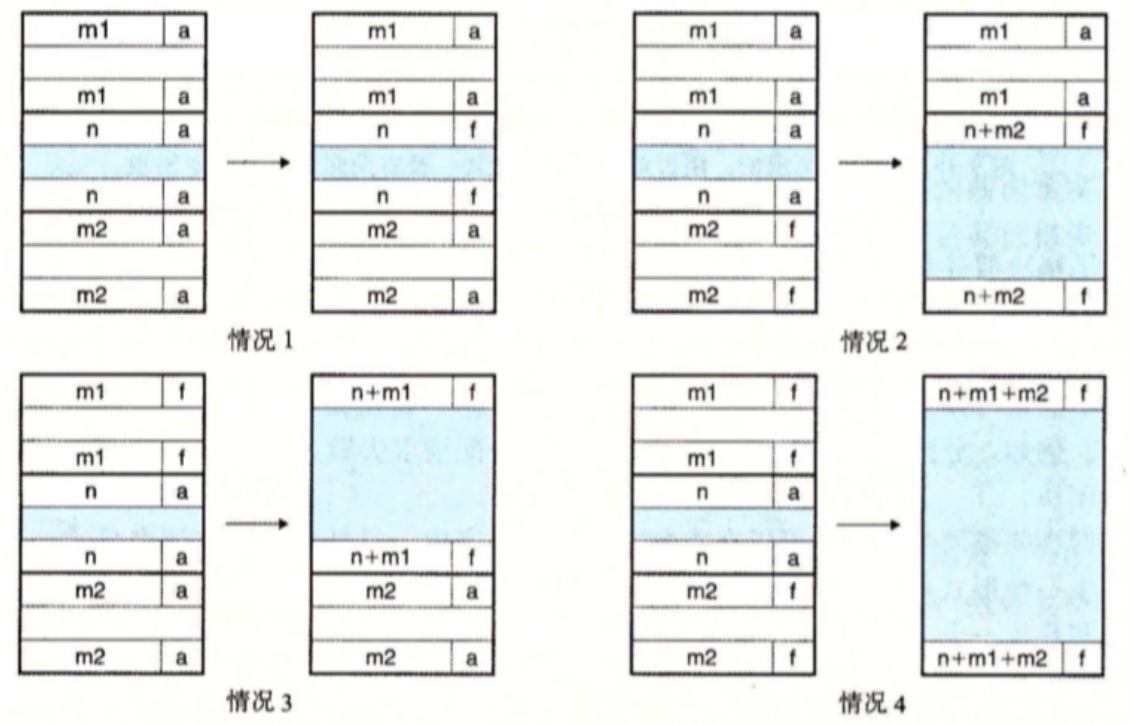
> 1. 前面的块和后面的块都已分配
> 2. 前面的块已分配，后面的块空闲
> 3. 前面的块空闲，后面的块已分配
> 4. 前后块都空闲：在实现了 2,3 步骤之后，第四种情况只需要顺序执行一遍 2,3 或者 3,2 即可正确合并。

```
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));
    if (prev_alloc && next_alloc)
    {
        return bp;
    }
    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));
    }
    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);
    }
```

```
    else
    {
      /*coalesce previous block*/
        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);
      /*coalesce next block*/
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));

    }
    return bp;
}
```

合并空闲块的四种情况图示：



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。

> 该部分的 Makefile 内容和下一部分一起说明。

测试程序在 Linux-0.11 内的运行结果：

```
QEMU
mmdriver.c:403: warning: implicit declaration of function `app_error'
mmdriver.c: In function app_error:
mmdriver.c:415: warning: `app_error' was previously implicitly declared to retur
n `int'
mmdriver.c: In function unix_error:
mmdriver.c:424: warning: `unix_error' was previously implicitly declared to retu
rn `int'
mmdriver.c: In function malloc_error:
mmdriver.c:433: warning: `malloc_error' was previously implicitly declared to re
turn `int'
gcc -g -Wall -o mmdriver memlib.o mm.o mmdriver.o
[/lab3_malloc]# ls
Makefile    ep_mm.c     memlib.o    mm.o        mmdriver.o
README      memlib.c    mm.c        mmdriver    traces
config.h    memlib.h    mm.h        mmdriver.c
[/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!
[/lab3_malloc]#
```

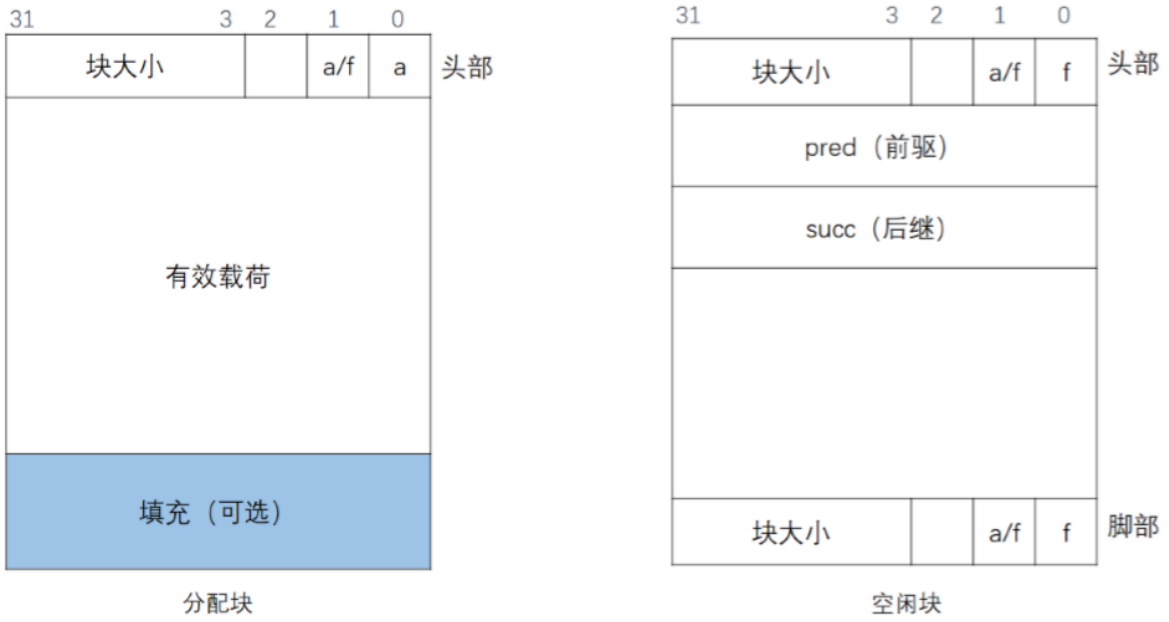
TASK2. 显式空闲链表的实现

实验原理

- 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。

下面就是显式链表的介绍：

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，显式空闲链表是一个双向链表，且只把空闲块链接起来。

所以在首次适配时，它不需要搜索整个堆，这让首次适配的时间从块总数线性时间缩短为空闲块个数的线性时间。

通过上图可以看到，显式链表的块格式和隐式空闲链表有所不同：

- 分配块没有脚部，优化空间利用率；
- 空闲块多了 prev（前驱）和 succ（后继）指针，存放相邻空闲块的位置。

> 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。

关键函数实现：

a. 放置策略 -- 首次适配

`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 *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 || size == asize)
        {
            break;
        }
        bp = GET_SUCC(bp); /*获得后继空闲块的位置 (GET(bp + WSIZE))*/
    }
    return (bp != NULL ? ((void *)bp) : NULL);
}
```

b. 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：

1. 调用`delete_from_free_list`，将该块移出空闲链表
2. 设置分配块的大小和头部信息，注意不设置尾部
3. 设置新空闲块的大小和头部信息
4. 调用`add_to_free_list`，将新空闲块添加到空闲链表中

> 若不需要分割空闲块，则只需将目标块移出空闲链表并设置对应头部信息后，修改目标块的相邻后块的头部信息即可。

`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 place(void *bp, size_t asize)
{
    size_t total_size = 0;
    size_t rest = 0;
    delete_from_free_list(bp); /*移出空闲链表*/
    /*remember notify next_blk, i am alloced*/
    total_size = GET_SIZE(HDRP(bp));
    rest = total_size - asize;

    if (rest >= MIN_BLK_SIZE) /*need split*/
    {
        PUT(HDRP(bp),PACK(asize,1,1)); /*设置分配块大小和前块已分配的信息，GET_PREV_ALLOC(HDRP(bp)) == 1 若为 0 则空闲块合并*/
        bp = NEXT_BLKP(bp);
        PUT(HDRP(bp),PACK(rest,1,0)); /*设置新的空闲块的大小和前块已分配的信息*/
        PUT(FTRP(bp),PACK(rest,1,0));
        add_to_free_list(bp); /*将新的空闲块加入空闲链表 - 头插法*/
    }
    else
    {
        PUT(HDRP(bp),PACK(total_size,1,1));
        bp = NEXT_BLKP(bp);
        PUT(HDRP(bp),PACK(GET_SIZE(HDRP(bp)),1,GET_ALLOC(HDRP(bp)))); /*设置下一块的前块已分配信息*/
    }
}
```

Makefile 文件

参考资料

make 是如何工作的

在默认的方式下，也就是我们只输入`make`命令。那么，

1. make 会在当前目录下找名字叫“Makefile”或“makefile”的文件。

2. 如果找到，它会找文件中的第一个目标文件（target），在下面的例子中，他会找到“all == mmdriver epmmdriver ”这两个文件，并把这个文件作为最终的目标文件。
3. 如果文件不存在，或是 mmdriver， epmmdriver 所依赖的后面的 `.o` 文件的文件修改时间要比 mmdriver， epmmdriver 文件新，那么，他就会执行后面所定义的命令来生成这两个文件。
4. 如果他们所依赖的 `.o` 文件也不存在，那么 make 会在当前文件中找目标为 `.o` 文件的依赖性，如果找到则再根据那一个规则生成 `.o` 文件。（这有点像一个堆栈的过程）
5. 于是 make 会生成 `.o` 文件，然后再用 `.o` 文件生成 make 的终极任务，也就是执行文件 `mmdriver`, `epmmdriver`了。

> `make clean` ，以此来清除所有的目标文件，以便重新编译。

让 make 自动推导

GNU 的 make 很强大，它可以自动推导文件以及文件依赖关系后面的命令，于是我们就没必要去在每一个 .o 文件后都写上类似的命令，因为，我们的make会自动识别，并自己推导命令。

只要make看到一个 .o 文件，它就会自动的把 .c 文件加在依赖关系中，如果 make 找到一个 mmdriver.o ，那么 mmdriver.c 就会是 mmdriver.o 的依赖文件。并且 cc -c whatever.c 也会被推导出来。于是，我们的makefile 可以写的比较简单。

make 中的变量

理解起来类似 C 语言中的 `#define` ： 提前用一个固定的字符串代替重复需要的内容，方便书写和修改。

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

CC = gcc -g
CFLAGS = -Wall

# 补充
OBJS1 = memlib.o mm.o mmdriver.o
OBJS2 = memlib.o ep_mm.o mmdriver.o

all: mmdriver epmmdriver

mmdriver: $(OBJS1)
# 补充gcc命令（使用变量）
    $(CC) $(CFLAGS) -o mmdriver $(OBJS1)

epmmdriver: $(OBJS2)
# 补充gcc命令（使用变量）
    $(CC) $(CFLAGS) -o epmmdriver $(OBJS2)


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

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

Linux-0.11 下测试程序运行结果