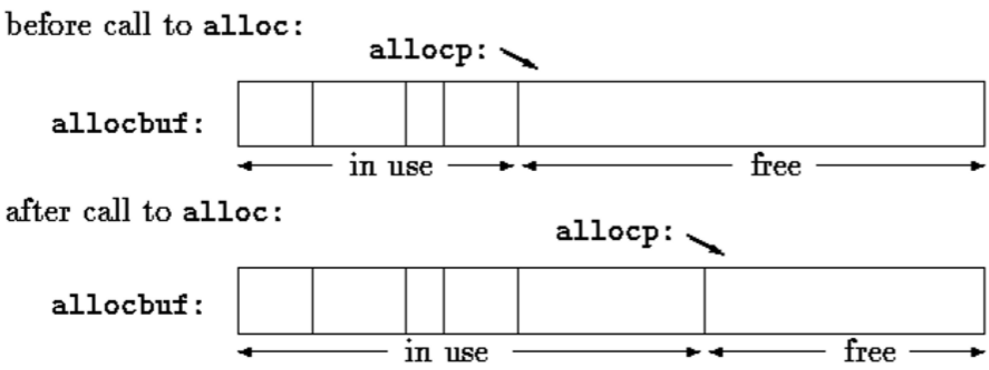
Malloc实现及改进

简单实现

```
/*
k&r 5.4
malloc simple implement
2010.4.10
*/

#define ALLOCSIZE 10000

static char allocbuf[ALLOCSIZE];
static char* allocp = allocbuf;//指向allocbuf中的下一个空闲单元

char* alloc(int n);//返回指向n个连续字符存储单元的指针
void afree(char* p);//释放已经分配的存储单元


char* alloc(int n) {
    if (allocbuf + ALLOCSIZE - allocp >= n) {
        allocp += n;
        return allocp-n;
    }
    else
        return 0;
}

void afree(char* p) {
    if (p >= allocbuf && p < allocbuf + ALLOCSIZE)
        allocp = p;
}



/*
这种简单实现的缺点:

        1.作为代表内存资源的allocbuf,其实是预先分配好的,可能存在浪费。

        2.分配和释放的顺序类似于栈,即"后进先出",释放时如果不按顺序会造成异
常。

    这个实现虽然比较简陋,但是依然提供了一个思路。如果能把这两个缺点消除,就能
够实现比较理想的malloc/free。
*/
```



改进

```
#include <unistd.h>  // sbrk

#define NALLOC 1024  // Number of block sizes to allocate on call to
sbrk
#ifdef NULL
#undef NULL
#endif
#define NULL 0


// 用于对齐
typedef long Align;

union header {
    struct {
        union header* next; //指向下一个空闲块
        unsigned size;
    } s;

    Align x; //仅用于对齐，用不到
};
typedef union header Header;


static Header base = {0};  //空链表的初始成员
static Header* freep = NULL;  /*指向空闲块链表的当前节点*/


static Header* morecore(unsigned nblocks);//获取更多空间
void kandr_free(void* ptr);




void* kandr_malloc(unsigned nbytes) {

    Header* currp;
    Header* prevp;
```

```
unsigned nunits;

/*
+1是包括了header
 */
nunits = ((nbytes + sizeof(Header) - 1) / sizeof(Header)) + 1;

// 创建一个退化的空闲链表，它只包含一个大小为0的块，且该块指向自己
if (freep == NULL) {
    base.s.next = &base;
    base.s.size = 0;
    freep = &base;
}


prevp = freep;
currp = prevp->s.next;

/*
遍历空链表找足够大的空闲区间,如果没有则申请
 */
for (; ; prevp = currp, currp = currp->s.next) {

    /*
    找到足够大的空间区间，连续的空闲区域>所需区域则分割处理
     */
    if (currp->s.size >= nunits) {

        /*
        大小正好相等
         */
        if (currp->s.size == nunits) {
            //注意next含义
            prevp->s.next = currp->s.next;
        }

        else {
            // 分配之后currp size - n
            currp->s.size -= nunits;
           //  分割
            currp += currp->s.size;
            // 后半部分用于分配
            currp->s.size = nunits;
        }

        freep = prevp;

        /*
        void*类型的指针可以不强制转型地赋给所有的指针类型变量
         */
        return (void*)(currp + 1);

    }

    /*
```

```
            遍历了一遍没有合适区间
             */
            if (currp == freep) {
                /*
                morecore()从系统申请更多的可用空间，并加入
                 */
                if ((currp = morecore(nunits)) == NULL) {
                    return NULL;
                }
            }
        }
    }
```

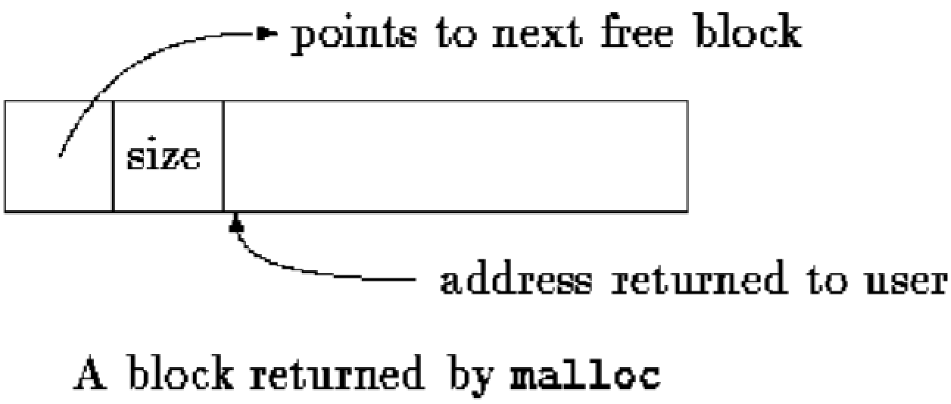


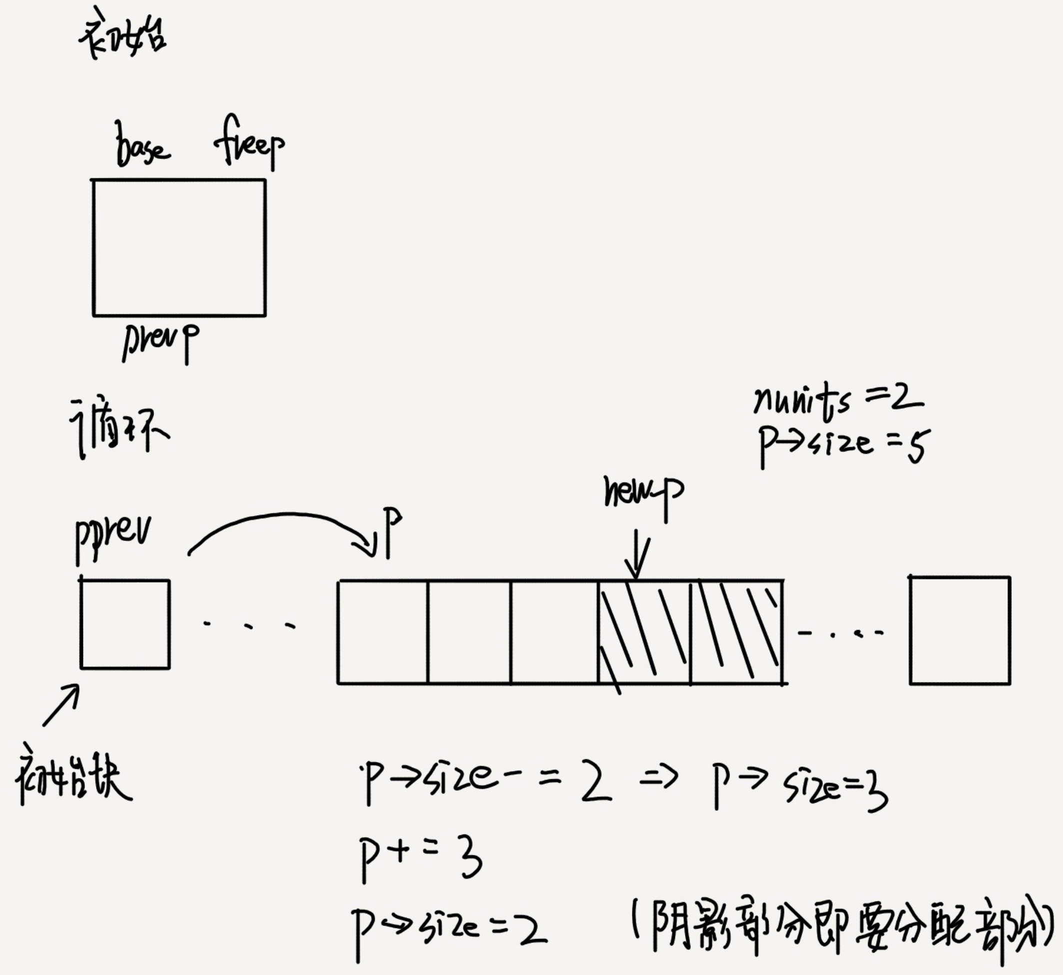
图 8-2　malloc 返回的块

几个疑问

1.nunits = (nbytes+sizeof(Header)-1)/sizeof(Header) + 1 ?

实际分配的空间是Header大小的整数倍，并且多出一个Header大小的空间用于放置Header。但是直观来看这并不是nunits = (nbytes+sizeof(Header)-1)/sizeof(Header) + 1啊？如果用(nbytes+sizeof(Header))/sizeof(Header)+1岂不是刚好？其实不是这样，如果使用后者，(nbytes+sizeof(Header))%sizeof(Header) == 0时，又多分配了一个Header大小的空间了，因此还要在小括号里减去1，这时才能符合要求。

2.看博客有人对于size>nunit下面的三行不理解，这里画出来了



为何最后 return p+1，因为头部是控制信息，不算空区
这里为方便理解设画。

```
static Header* morecore(unsigned nunits) {

    void* freemem;     // 新建的内存地址
    Header* insertp;  // Header ptr for integer arithmatic and
constructing header

    /*
    从os申请空间相当耗时，所以一次申请至少NALLOC
     */
    if (nunits < NALLOC) {
        nunits = NALLOC;
    }

    /*
    sbrk(n)返回一个指针，该指针指向n个字节的存储空间
     */
    freemem = sbrk(nunits * sizeof(Header));
    // sbrk分配失败return -1
    if (freemem == (void*)-1) {
```

```
        return NULL;
    }

    // 构建新块
    insertp = (Header*)freemem;
    insertp->s.size = nunits;

    /*
    将新块添入空闲链表
     */
    kandr_free((void*)(insertp + 1));

    return freep;
}


void kandr_free(void* ptr) {

    Header* insertp, * currp;

    // 需要插入的数据的header
    insertp = ((Header*)ptr) - 1;

    /*
    从freep指向的地址开始，逐个扫描空闲块链表，寻找可以插入空闲块的地方。该位置可能在两个空闲块之间，也可能在链表的末尾。在任何一种情况下，如果被释放的块与另一空闲块相邻，则将这两个块合并起来。合并两个块的操作很简单，只需要设置指针指向正确的位置，并设置正确的块大小就可以了
     */
    for (currp = freep; !((currp < insertp) && (insertp < currp->s.next)); currp = currp->s.next) {

        /*当currp>=currp->s.next时，currp位于链表的尾部，currp->s.next位于链表的头部
         */
        if ((currp >= currp->s.next) && ((currp < insertp) || (insertp < currp->s.next))) {
            break;
        }
    }
    /*
    区间合并
    */
    /*如果与后面的块相邻就合并两个块*/
    if ((insertp + insertp->s.size) == currp->s.next) {
        insertp->s.size += currp->s.next->s.size;
        insertp->s.next = currp->s.next->s.next;
    }
    //插入currp后面
    else {
        insertp->s.next = currp->s.next;
    }

    /*如果与前面的空闲块相邻就合并两个块*/
    if ((currp + currp->s.size) == insertp) {
```

```
        currp->s.size += insertp->s.size;
        currp->s.next = insertp->s.next;
    }

    else {
        currp->s.next = insertp;
    }

    freep = currp;
}
```

ucore first fit malloc实现

page结构定义

```
//memlayout.h
struct Page {
    int ref;                        // 页引用计数
    uint32_t flags;                 //
    unsigned int property;          // 记录某连续内存空闲块的大小
    list_entry_t page_link;
};

//flags有两种场景
#define PG_reserved                 0       // 置1保留不能用于空闲链表
#define PG_property                 1       // 置1可被分配
```

free_area_t结构为了有效地管理这些小连续内存空闲块

```
//memlayout.h
typedef struct {
    list_entry_t free_list;         // the list header
    unsigned int nr_free;           // 记录当前空闲页的个数
} free_area_t;
```

初始化

```
free_area_t free_area;  //新建一个空链表

#define free_list (free_area.free_list)
#define nr_free (free_area.nr_free)

static void
default_init(void) {
    list_init(&free_list);
    nr_free = 0;
}

//初始化最初的空闲物理页
static void
default_init_memmap(struct Page *base, size_t n) {
```

```
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p ++) {
        assert(PageReserved(p));//确认此页为内核保留
        p->flags = p->property = 0;
        set_page_ref(p, 0);
    }
    base->property = n;
    SetPageProperty(base);
    nr_free += n;
    list_add_before(&free_list, &(base->page_link));//插入到队头之前，即
队尾
}
```

malloc

```
static struct Page *
default_alloc_pages(size_t n) {
    assert(n > 0);
    if (n > nr_free) {
        return NULL;
    }
    struct Page *page = NULL;
    list_entry_t *le = &free_list;

    while ((le = list_next(le)) != &free_list) {//遍历freelist
        struct Page *p = le2page(le, page_link);//le2page即
container_of
        if (p->property >= n) {
            page = p;
            break;
        }
    }
    //剩余的部分作为新的空闲空间插入到原空间位置
    if (page != NULL) {
        if (page->property > n) {
            struct Page *p = page + n;
            p->property = page->property - n;
            SetPageProperty(p);
            list_add_after(&(page->page_link), &(p->page_link));
        }
        list_del(&(page->page_link));
        nr_free -= n;
        ClearPageProperty(page);
    }
    return page;
}
```

free

```
static void
default_free_pages(struct Page *base, size_t n) {
```

```c
    assert(n > 0);
    struct Page *p = base;
    for (; p != base + n; p ++) {
        assert(!PageReserved(p) && !PageProperty(p));
        p->flags = 0;
        set_page_ref(p, 0);
    }
    base->property = n;
    SetPageProperty(base);
    list_entry_t *le = list_next(&free_list);
    //查找可供合并的块
    while (le != &free_list) {
        p = le2page(le, page_link);
        le = list_next(le);
        //相邻块左侧合并，假设左侧为低地址
        if (base + base->property == p) {
            base->property += p->property;
            ClearPageProperty(p);
            list_del(&(p->page_link));
        }
        //右侧合并
        else if (p + p->property == base) {
            p->property += base->property;
            ClearPageProperty(base);
            base = p;
            list_del(&(p->page_link));
        }
    }
    nr_free += n;
    le = list_next(&free_list);
    while (le != &free_list) {// 将合并好的合适的页块添加回空闲页块链表
        p = le2page(le, page_link);
        if (base + base->property <= p) {
            assert(base + base->property != p);
            break;
        }
        le = list_next(le);
    }
    list_add_before(le, &(base->page_link));
}
```

参考

https://www.ituring.com.cn/book/miniarticle/55542

https://www.cnblogs.com/wuyuegb2312/archive/2013/05/03/3056309.html

https://my.oschina.net/u/4000302/blog/3215820/print

https://www.cnblogs.com/zijintime/p/7516635.html

http://blog.gqylpy.com/gqy/24219/

https://yuerer.com/%E6%93%8D%E4%BD%9C%E7%B3%BB%E7%BB%9F-uCore-Lab-2/

https://qinggniq.com/2019/10/10/uCore-lab2%E6%8A%A5%E5%91%8A-%E7%89%A9%E7%90%86%E5%86%85%E5%AD%98%E7%AE%A1%E7%90%86%EF%BC%88%E5%90%ABchallenge%EF%BC%89/

https://twinkle0331.github.io/ucore-lab2.html

https://github.com/zhenghaoz/ucore/tree/master/lab2?tdsourcetag=s_pctim_aiomsg

如有纰漏敬请指正！