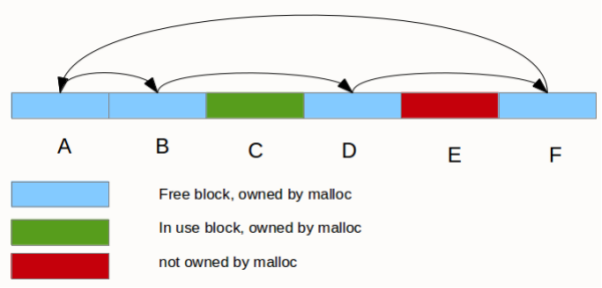
Memory Allocator for embedded system (K & R Ritchie book)

February 26, 2013 General embedded, linkedlist, malloc, memoryallocator

Memory allocator given in K & R book

This memory allocator use single linked list, the following picture represent the type of linked list it will create.



Block A, B, C, D, F are owned by the memory allocator and block E is the block not owned by memory allocator. A block with free memory are linked together. Basically memory allocator keep a track of blocks having free memory once the block is used, it will remove that block from this linked list.

So a block is a structure which keep size and pointer to next free block

```
typedef double Align;//for alignment to doubl boundary
union header{//block header
struct{
union header* ptr;///next block if on free list
unsigned int size;///size of this block
}s;
Align x;///Force alignment of blocks
};
```

Align x is never used , it just forces each header to be aligned to worst case boundary.

We will go ahead to malloc function and block by block I will explain the conditions.

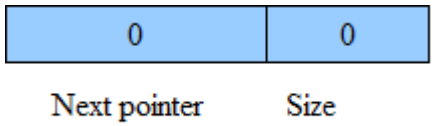
```
typedef union header Header;
static Header base;/* empty list to get started */
static Header* freep = NULL;/* start of free list */
```


We declared a static member of type Header because we always need a header (base).

Initially base will not point to anything, it wont have any memory to return and size will be zero.

We need a freep pointer to header type so that we keep track of all blocks.

So before we start malloc we have the following situation :



On my compute size of Header was 8 bytes.

Let say we have asked for 8 bytes.

```
/* malloc: general-purpose storage allocator */
void* myMalloc(unsigned nbytes)
{
Header* p , *prevp;
unsigned nunits;
nunits = (nbytes+sizeof(Header)-1)/sizeof(header)+1;
```

nunits we are using to calculate the required no of units of Header.

Let say we need a single byte even then we will need two blocks of Header either 16 Bytes. One block to keep size and other block that we will return to user by malloc request. Formula used above is a standard formula . Let say we want 8 bytes then it will ask for 2 units of Header either 16 bytes.

```
if((prevp=freep)==NULL)/* no free list yet */
{
base.s.ptr=freep=prevp=&base;
base.s.size=0;
}
```

for the first time when there is no free list we set the size of base header to zero and base pointer to base itself either base will point to base. And we initialize prev and freep with base address.

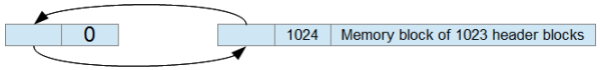
```
for(p=prevp->s.ptr;;prevp=p,p=p->s.ptr)
{
if(p->s.size>=nunits)
{
/* big enough */
if(p->s.size==nunits)/* exactly */
prevp->s.ptr=p->s.ptr;
else
{
/* allocate tail end */
p->s.size-=nunits;
```

```
p+=p->s.size;
p->s.size=nunits;
}
freep=prevp;
return (void*)(p+1);
}
if(p==freep)/* wrapped around free list */
{
if((p=morecore(nunits))==NULL)
return NULL;
}
/* none left */
}
}
```

Initially we will iterate through the base block and find that its size is less than units required. So malloc ask morecore function and get a new block of memory defined in moreCore function. According to book its 1024. So now we will have 2 blocks, base block and a new block with size of 1024.

```
#define NALLOC 1024/* minimum #units to request */
/* morecore: ask system for more memory */
static Header* morecore(unsigned nu)
{
char*cp;
Header*up;
if(nu<NALLOC)
nu=NALLOC;
cp=(char*)sbrk(nu*sizeof(Header));
if(cp==(char*)-1)
/* no space at all */
return NULL;
up=(Header*)cp;
up->s.size=nu;
myFree((void*)(up+1));
return freep;
}
```

in the end we use myFree((void*) (up + 1)), so that new block of 1024 size get linked with existing list.So that structure will be like this.



Now base ptr will point to new block we have just made and new blocks ptr will point to base.

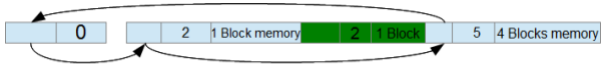
Now the loop inside malloc will go through the new block when its finds that size of new block is greater than the units asked. It will create a node at end, free it and update the size of block.

Green color show that it is owned by malloc but it in not in free list. It return 1 Block's address. We keep doing the samething until free block have enough memory.

There will be two situation

1st the free block have the same no of free blocks as nuntis.

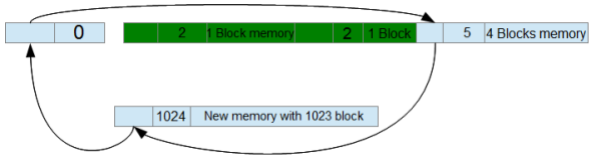
In the case below , lets we have asked for 2 blocks, the size of 2nd free block is exactly 2.



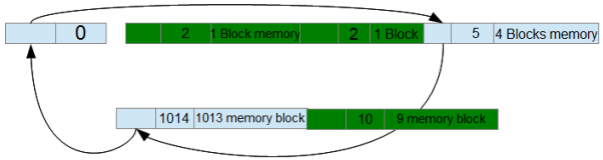
malloc will use 2nd block and free it and will link the 1st block to third. The situation will be like this

If it does not find the block with enough size it will again call morecore and get a block of 1024 size and will link it properly .

Let say we have asked for 10 nunits of header than it will do following



Than it will give memory from the tail of new block.The situation will be like this.



Now we will discuss the free function. One thing we have noticed that we do not point to header, we point to memory block of that header.

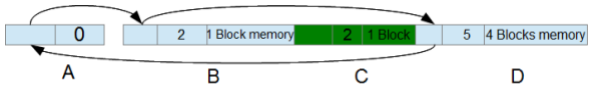
```
void myFree(void* ap)
{
Header*bp,*p;
bp=(Header*)ap-1;
for(p=freep;!(bp>p&&bp<p->s.ptr);p=p->s.ptr)
{
if(p>=p->s.ptr&&(bp>p||bp<p->s.ptr))
{
break;/* freed block at start or end of arena */
}
}
if(bp+bp->s.size==p->s.ptr)/* join to upper nbr */
{
bp->s.size+=p->s.ptr->s.size;
bp->s.ptr=p->s.ptr->s.ptr;
}
else
bp->s.ptr=p->s.ptr;
if(p+p->s.size==bp)/* join to lower nbr */
{
p->s.size+=bp->s.size;
p->s.ptr=bp->s.ptr;
}
```

```
else
p->s.ptr=bp;
freep=p;
}
```

The for lopp in the free function just ask find that whether the block to be freed is with in

block p and block pointed by p. if its find that situation . If that situation is there it exis the loop.

The following situation let say we have to delete the block C.



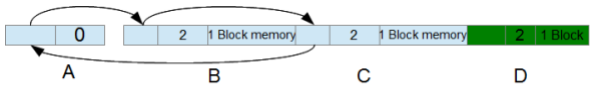
when p pointer will come to b the condition (bp > p && bp < p->s.ptr) will become true. So the loop will break there.

But let say we started at block D than it will iterate through the complete and come to block C and than it exit the lopp. For this situation if condition works (p >= p->s.ptr && (bp > p || bp < p->s.ptr)

because when p is at D block. P > p->s.ptr is true and bp < p->s.ptr is true so it will exit.

This condition just say the address of the block to be removed is with in the two free blocks.

If condtion with in the for loop check the following conditions



Lets assume that blocks in the right have higher memory address.

In this above condition will fail so we have used extra condition in if , when p will be at C block condition (p >= p->s.ptr && (bp > p || bp < p->s.ptr) will exit loop.

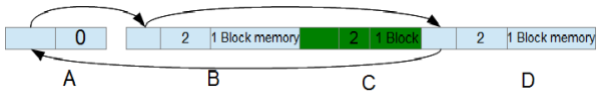
P > p->s.ptr and bp > p.

When there is only one free block that is base and it point to itself

p = p->s.ptr and bp > p suffice the condition.

Next two part are easy.

In the figure below let say we have to free C block

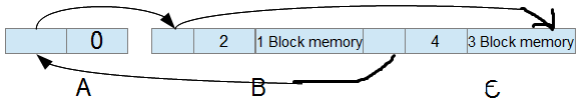


if it is adjacent to right that means

bp + bp->size == p->s.ptr

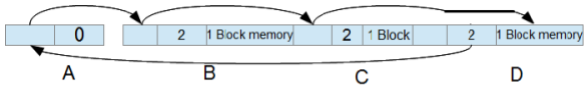
merge block C & D

following figure represent that.



Block c merged with D and size of c is incremented with size of D and block C will point to block whom block D was pointing.

If its not adjacent the situation will be the following



It just add a block in list and C will point to D .

In this if else structure only the block to be freed just point to next free element **but no block point to freed block that is done in next if else**

If it is adjacent to left block it merge the block with left block and update the left block pointer and size.

In case its not adjacent the left block , left block will point to freed block.

In the end it just update the freep pointer to just freed block.