**That will be the day,...**

**Johan Montelius**

**HT2018**

# 1 Introduction

This is an assignment where you will implement your own malloc using the buddy algorithm to manage the coalescing of free blocks. You should be familiar with the role of the allocator and how to implement and benchmark a simple version of it. In this presentation you will not be given all details on how to do the implementation, but we will go through a general strategy.

# 2 The buddy algorithm

The idea of the buddy algorithm is that given a block to free, we could quickly find its sibling and determine if we can combine the two into one larger block. The benefit of the buddy algorithm is that the amortized cost of the allocate and free operations are constant. The slight down side is that we will have some internal fragmentation since blocks only come in powers of two.

We will keep the implementation as simple as possible, and therefore we will not do the uttermost to save space or reduce the number of operations. The idea is that you should have a working implementation and do some benchmarking.

## 2.1 the implementation

Your implementation does not need to follow these guide lines but you can take them as a starting point. Feel free to improve or implement a different strategy (if it is equally good, better or have some advantage).

### 2.1.1 the setting

In this implementation the largest block that we will be able to handle is 4 Ki byte. We can then split it into sub-blocks and the smallest block will be 32 bytes ($2^5$). This means that we will have 8 different sizes of blocks: 32, 64, 128 , .. 4 Ki. We will refer to these as level 0 block, level 1 block etc. We encode these as constants in our system: *MIN* the smallest block is $2^{MIN}$, *LEVELS* the number of different levels and *PAGE* the size of the largest block ($2^{12} = 4Ki$).

```
#define MIN   5
#define LEVELS  8
```

```
#define PAGE 4096
```

When we allocate a new block of 4 Ki byte, it needs to be aligned on a 4 Ki boundary i.e. the lower 12 bits are zero. This is important and we will get this for free if we choose the system page sizes when allocating a new block.

## 2.2 operations on a block

A block consists of a *head* and a byte sequence. It is the byte sequence that we will hand out to the requester. We need to be able to determine the size of a block, if it is taken or free and, if it is free, the links to the next and previous blocks in the list.

```
enum flag {Free, Taken};

struct head {
  enum   flag  status;
  short  int   level;
  struct head  *next;
  struct head  *prev;
};
```

The size of the block is represented by the *level* and is encoded as $0, 1, 2$ etc. This is the index in the set of free lists that we will keep. Index 0 is for the smallest size blocks i.e. 32 byte.

We will define a set of functions that do the magic of the buddy algorithm. Given these functions we should be able to implement the algorithm independent of how we choose to represent the blocks.

### a new block

To begin with we have to create new block. We do this using the *mmap()* system call. This procedure will allocate new memory for our process and here we allocate a full page i.e. a 4 Ki byte segment. Here we take for granted that the segment returned by the *mmap()* function is in fact aligned on a 4 Ki byte boundary.

```
struct head *new() {
  struct head *new =  (struct head *) mmap(NULL,
                                           PAGE,
                                           PROT_READ | PROT_WRITE,
                                           MAP_PRIVATE | MAP_ANONYMOUS,
                                           -1,
                                           0);
  if(new == MAP_FAILED) {
    return NULL;
```

```
  }
  assert(((long int)new & 0xfff) == 0);  // 12 last bits should be zero
  new->status = Free;
  new->level = LEVELS -1;
  return new;
}
```

Look-up the man pages for *mmap()* and see what the arguments mean. When you extend the implementation to handle larger blocks you will have to change these parameters.

### who's your buddy

Given a block we want to find the buddy. We do this by toggling the bit that differentiate the address of the block from its buddy. For the 32 byte blocks, that are on level 0, this means that we should toggle the sixth bit. If we shift a one five positions (*MIN*) to the left we have created a mask that we can xor against the pointer to the block.

```
struct head *buddy(struct head* block) {
  int index = block->level;
  long int mask =  0x1 << (index + MIN);
  return (struct head*)((long int)block ^ mask);
}
```

### split a block

When we don't have a block of the right size we need to divide a larger block in two. To do this we set the bit that will separate two budies at the requested level. In the code below we do almost the opposite of the buddy operation, we find the level of the block, subtract one and create a mask that is or:ed with the original address. This will now give us a pointer to the second part of the block.

```
struct head *split(struct head *block) {
  int index = block->level - 1;
  int mask =  0x1 << (index + MIN);
  return (struct head *)((long int)block | mask );
}
```

### merge two blocks

We also need a function that merges two buddies. When we are to perform a merge, we first need to determine which block is the *primary* block i.e. the first block in a pair of budies. The primary block is the block that should be the head of the merge block so it should have all the lower bits set to zero.

We acheive this by masking away the lower bits up to and including the bit that differentiate the block from its buddy. Note that it does not matter which buddy we choose, the function will always return the pointer to the primary block.

```
struct head *primary(struct head* block) {
  int index = block->level;
  long int mask =  0xffffffffffffffff << (1 + index + MIN);
  return (struct head*)((long int)block & mask);
}
```

### the magic

We use the regular magic trick to hide the secret head when we return a pointer to the application. We hide the head by jumping forward one position. If the block is in total 128 bytes and the head structure is 24 bytes we will return a pointer to the 25'th byte, leaving room for 104 bytes.

```
void *hide(struct head* block) {
  return (void*)(block + 1);
}
```

The trick is performed when we convert a memory reference to a head structure, by jumping back the same distance.

```
struct head *magic(void *memory) {
  return ((struct head*)memory - 1);
}
```

### level

The only thing we have left to do is to determine which block we should allocate. If the user request a certain amount of bytes we need a slightly larger block to hide the head structure. We find the level by shifting a size right, bit by bit. When we have found a size that is large enough to hold the total number of bytes we know the level.

```
int level(int req) {
  int total = req  + sizeof(struct head);

  int i = 0;
  int size = 1 << MIN;
  while(total > size) {
    size <<= 1;
    i += 1;
  }
```

```
  return i;
}
```

### 2.2.1 before you continue

Implement the above functions in a file called *buddy.c*. Also implement a function *test()* that runs simple tests that ensures you that the operations works as expected. You can for example create a new block, divide it in two, divide in two and then hide its header, find the header using the magic function and then merge the blocks. Add a file *buddy.h* that holds a declaration of the test procedure.

In a file *test.c* write a *main()* procedure that calls the test procedure. Compile, link and run. Do extensive testing, if anything is wrong in these primitive operations it will be a nightmare to debug later.

## 2.3 the algorithm

So given that the primitive operations work as intended, it is time to implement the algorithm. Your task is now to implement two procedures *balloc()* and *bfree()* that will be the API of the memory allocator. We do not replace the regular *malloc()* and *free()* since we want to use them when we write our benchmark program.

We will use one global structure that holds the double linked free lists of each layer.

```
struct head *flists[LEVELS] = {NULL};
```

The implementation can be broken down into two phases. The first phase will hide or strip the magic head structure, and the second phase can work on regular block structures. This could be one way to go:

```
void *balloc(size_t size) {
  if( size == 0 ){
    return NULL;
  }
  int index = level(size);
  struct head *taken = find(index);
  return hide(taken);
}
```

```
void bfree(void *memory) {
  if(memory != NULL) {
    struct head *block = magic(memory);
    insert(block);
  }
  return;
```

```
}
```

It's now time for you to implement the procedures *find()* and *insert()*. First write down the pseudo code, what needs to be done in each step. Draw pictures that describes how the double linked list is manipulated. Do small test as you proceed, make sure that simple things work before you continue - happy hacking.

# 3 Benchmarking

When your buddy allocator works as expected it's time to do some benchmarks. The things we want to know is:

- the cost of a bfree and balloc operation
- does it vary with the size of the memory
- what is the memory utilization, free vs taken
- ...

You will need a benchmark program that can vary the requests given a min and max size. You should also be able to change the number of blocks requested and the number of blocks the application has allocated at a given time. The sequence of operations should preferably match a real program i.e. probably not be a sequence of identical requests.

Set up some benchmarks and try to describe the properties of your implementation. Try to pinpoint situations where it does not perform as well.

# 4 Improving the performance

When you have things up and running you might want to improve the system. Before you do this, you should however get an understanding of the costs and if it's worth improving upon. Here are some ideas that you could consider.

## 4.1 the size of the head

The size of the head structure is one thing you can take a look at. If you have implemented it as above it is 24 bytes (4+4+8+8) which we might improve. Since we want to hand out segments aligned by 8 bytes it's not possible to improve it by a byte or two, we have to do something radical!

One idea is that the head structure only needs to hold its two pointers if it is a free block. Only then are the pointers needed for linking. A taken block only need the status flag and the level counter. When we hide the head we do not have to jump 24 bytes forward but only as far as beyond the status and level fields. Try the following:

```
struct peggysue {
  enum flag status;
  short int level;
};
```

Now alter the code section that performs the magic trick, as well for the calculation of how big a block needs to be and use Peggy Sue. Since this structure is only 8 bytes we will allow more user information in the block that we use. This might not be of significance if the block is 128 bytes to start with but it means a lot if the block is 32 bytes.

Note, the smallest possible block is still 32 bytes since when converted to a free block it needs to be able to hold the 24 bytes representing the head structure.

### 4.2 tagged pointers

One way to shrink the size of the head structure is to let it hold two pointers only. The information that tells us if it is free or not and what level it belongs to could be encoded in bits not used by the pointers. Our pointers will always point to a block and if these are sixteen byte aligned that means that the four least significant bits could be zero.

We could use the four bits of the next pointer to tell us if the block is free or not and the four bits of the previous pointer to tell us the level of the block. If we do this then the smallest block could be 16 bytes instead of 32.

If you implement this you must of course mask the pointers before using them but it could be quite easily hidden in a macro. Note however that whenever you play around with bits, relying on that they will not be used, will make you code less portable (but that shouldn't stop you).

### 4.3 return pages

If we run large benchmarks you will have more and more 4 Ki byte pages being allocated. This is normal since the chance of requesting many large segments at the same time increase as we randomly choose the size of the segments. We will never return an allocated page but you could of course do this when a level 7 page is created from two level 6 blocks. We might want to keep one or two but why have ten lying around?

Do a check when a full page is created and see if you already have four pages in your free list. If that is the case you should be able to return the page to the operating system.

### 4.4 bit maps

Handling a double linked lists is expensive; you will do several operations and they might be reading and writing to memory locations that are not

held by the cache. A completely different approach is to keep track of free blocks in a bit map.

Let's say that we only have one 4 Ki byte block to think about. This could be broken down into 128 blocks of 32 bytes each. If each block is encoded as one bit this would mean that we would only need 16 bytes to encode all the information. We would still need to have the size information in each block so each block needs a head structure but we could do with out the pointers.

When we're looking for a free block of size 64 bytes, we then scan the bit map and look for two consecutive bits that are not set. If we find the two bits we can also determine the address of this block.

The head structure would still need 8 bytes but now we might squeeze in some more information there. If the level is from 0 to 7, this will only require three bits. The status flag is of course only one bit, so that would leave us with 7 bytes to encode something else. Could we use these extra bits to encode which bitmaps the block belongs to if we have several bitmaps?

### 4.5 a larger heap

In the implementation above we use a page as the largest possible block. We could however extend this and use the *huge page* option for *mmap()*. Look up the man pages and see if you can change the code to work with pages of 2 Mi byte. It might not work out of the box but with some configurations it should work; this command might come in handy:

```
> sudo bash -c "echo 10 > /proc/sys/vm/nr_hugepages"
```

## 5 Summary

If you have completed the buddy allocator and done some benchmarking you should have a better understanding of how *malloc()* and *free()* works - or rather could work; the regular malloc used by Linux uses another protocol to keep track of free blocks.

The buddy algorithms have some advantages and some disadvantages. Benchmarking the system will give you a good understanding of its performance. The limitation is that we will have some, predictable, level of internal fragmentation and that the smallest available block is, in our implementation, quite large.