UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG
SCHOOL OF COMPUTER SCIENCE & APPLIED MATHEMATICS

COMS3010A
Operating Systems

Project 1 - My Alloc

October 10, 2022

Instructors

COMS3010A Lecturer:
Branden Ingram branden.ingram@wits.ac.za
COMS3010A Tutors:
William Hill 2115261
Andrew Boyley 2090244
Oluwatimileyin Obagbuwa 2134111
Sheslin Naidoo 2094701
Derrin Naidoo 2127039
Phindulo Makhado 1832463
Mohammed Gathoo 2089236
Sedzani Ramathaga 2083519
Ryan Alexander 1827474
Talion Naidoo 1448771
Jonathan Nunes 2087190
Phoenix Krinsky 2233063
Christopher Walley 1846871
Alice Govender 1847313

Consultation Times

Questions should be firstly posted on the discord channel, for the Lecturer and the Tutors to answer. If further explanation is required consultation times can be organised.

1 Introduction

This is an assignment where you will implement your own malloc using the buddy algorithm to manage the splitting and coalescing of free blocks. You should be familiar with the role of the allocator and the theory behind the operation of the buddy algorithm from the Lectures. In this project you will not be given all details on how to do the implementation, but the general strategy will be outlined. You will be provided with some helper functions which you will be required to use in order to implement the buddy allocator. A Project Template has been provided on moodle to use. The functioning of these helper functions is explained below.

2 The buddy algorithm

The buddy memory allocation technique is a memory allocation algorithm that divides memory into partitions to try to satisfy a memory request as suitably as possible. This system makes use of splitting memory into halves to try to give a best fit. 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.

2.1 Implementation Guidelines

You have been provided with much of the algorithm already implemented and are only required to implement the "balloc" function which will be discussed below.

Your implementation should follow the guidelines outlined in this document. In this implementation the largest block of memory that we will be able to handle is 4KB. 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: 32bytes, 64, 128 , .. , 4 KB.

We will refer to these as a level-0 block, a level-1 block etc. We encode these as constants in our system in the "buddy.h" file:

- MIN the smallest block is (2^{MIN})
- LEVELS the number of different levels
- PAGE the size of the largest block $(2^{12} = 4\text{KB})$

2.2 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. You will find a "struct" defined to represent the head. The size of the block is represented by a variable called "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. a variable called "status" to indicate if the memory region is free or allocated as well as variables called "next" and "prev" for our block pointers.

```
struct head {
        bool status;
        int level;
        struct head *next;
        struct head *prev;
}
```

2.3 The freelist

Instead of having a single freelist which keeps track of blocks we utilise a list of freelists which each keeps track of a specific set of blocks dependant on their size.

```
struct head * flists[LEVELS];
```

2.4 Helper Functions

The following subsections are explanations to the functioning of the helping functions which have been provided to you. You must not change these in any way.

A set of functions are required to implement the functionality of the buddy algorithm. Some of these have been provided in the project template. Given these functions you should be able to implement the algorithm independent of how we choose to represent the blocks. **Ultimately you will be required to implement the following function**:

- balloc - Section 2.6

while making use of the following functions:

- new - Section 2.4.1
- level - Section 2.4.2
- buddy - Section 2.4.3
- split - Section 2.4.4
- primary - Section 2.4.5
- hide - Section 2.4.6
- bfree - Section 2.4.7

2.4.1 a new block

To begin with you have to create new block. you can do this using the mmap() system call. This procedure will allocate new memory for the process and here we allocate a full page i.e. a 4KB segment. This is the function which you need to call whenever there is not any space available in your freelist.

```
struct head *new() {
    void *ans = mmap(NULL, PAGE, PROT_WRITE | PROT_READ, MAP_PRIVATE
         | MAP_ANONYMOUS, -1, 0);
    if (ans == MAP_FAILED) {
        // error, return NULL.
        return NULL;
    }
    // cast it to a head pointer.
    head *headPtr = (head *)ans;

    headPtr->level = MAX_LEVEL;
    headPtr->status = STATUS_FREE;  // intially the block is free.
    headPtr->next = NULL;
    headPtr->prev = NULL;
    return headPtr;
}
```

Figure 1: struct head* block = new();

This function as demonstrated above returns a pointer to the start of a struct head* object

2.4.2 level

If the user request a certain amount of bytes we need a slightly larger block to hide the head structure. When we have found a size that is large enough to hold the total number of bytes we know the level. For example if a user wants to request 1000 bytes we need to find space for that plus the size of the header.

```
int level(int req) {
    int l = MAX_LEVEL;
    while ((1 << (l + MIN)) >= req + sizeof(head)) {
        --l;
    }
    return ++l;
}
```

2.4.3 who's your buddy

Due to the tree like nature that emerges during the application of the buddy allocator we may want to find the partner or buddy of a particular block in our freelist. Therefore given a block we want to find the buddy which is the other pair of that block.

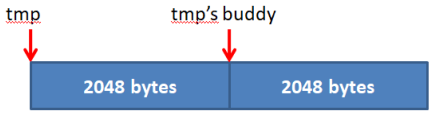

Figure 2: struct head* tmp's buddy = buddy(tmp);

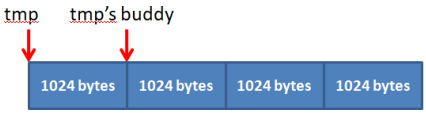

Figure 3: struct head* tmp's buddy = buddy(tmp);

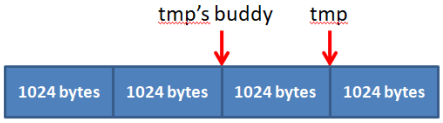

Figure 4: struct head* tmp's buddy = buddy(tmp);

The figures above demonstrate the desired behaviour of this function. One way of doing this is 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 1 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);
}
```

2.4.4 split a block

When we don't have a block of the right size we need to divide a larger block in two.

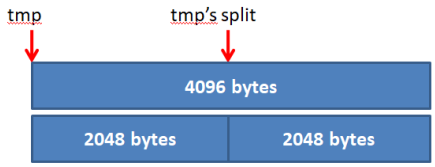

Figure 5: struct head* tmp's split = split(tmp);

This can be achieved by first finding 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);
}
```

2.4.5 the primary of 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 buddies. 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.

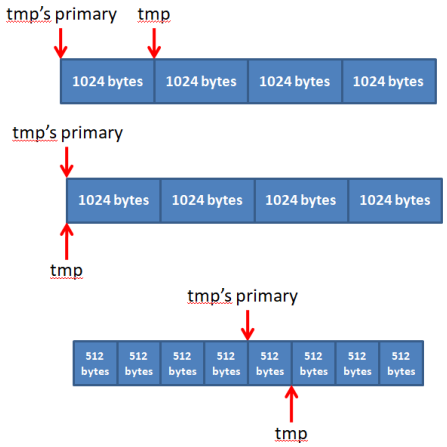

Figure 6: struct head* tmp's primary = primary(tmp);

We achieve 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);
}
```

2.4.6 hiding and un-hiding the header

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.

Figure 7: struct head* tmp's hide = hide(tmp);

```
struct head *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);
}
```

2.4.7 bfree

In conjunction with allocating free space, one needs to be able to free said memory as well. However, this has already been done for you. Below is the function structure as well as some psuedocode.

```
void bfree(void *memory) {
//Check memory is a null pointer
//Get the header of the pointer
//Update flists and free the block which the pointer points to
//Perform merging up the list if possible
}
```

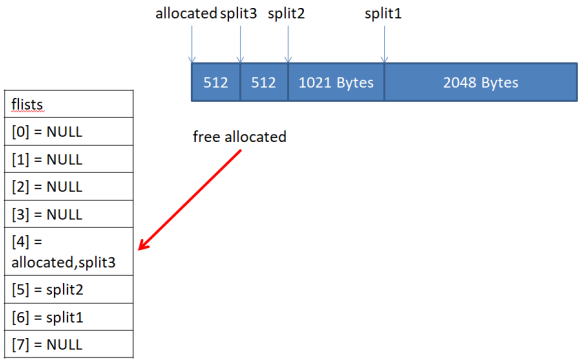

Figure 8: Free Example

To free some memory we need to identify which part of flists it belongs to.

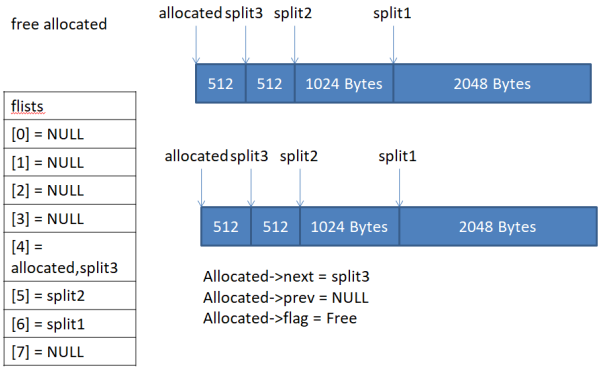

Figure 9: Free Example

We then have to update that block in terms of its status.

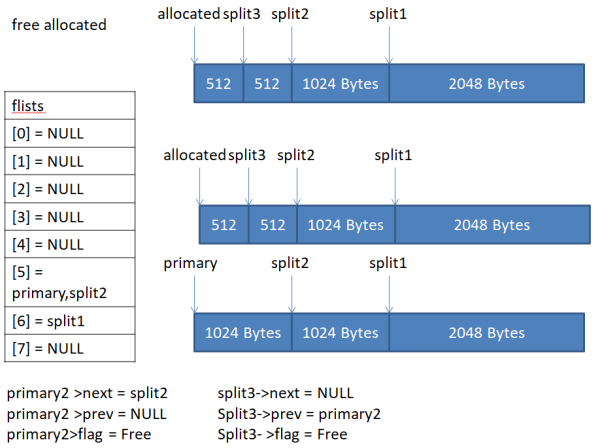

Figure 10: Free Example

lastly if merging is possible, i.e. if its buddy is also free, we need to combine these two blocks and update flists accordingly. If this results in the possibility to merge further up the tree you need to do so.

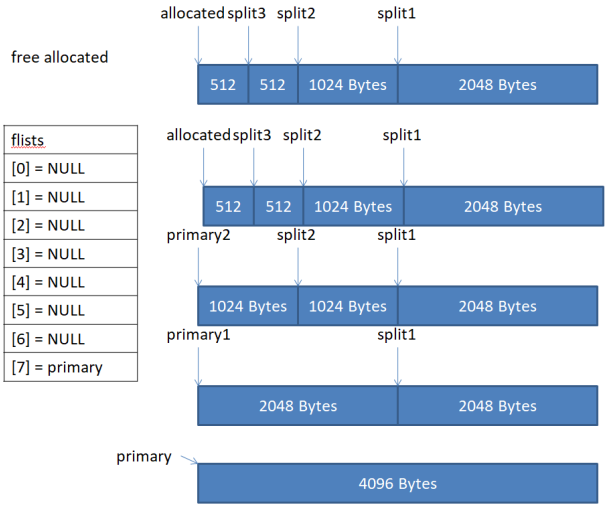

Figure 11: Free Example

2.5 Additional Help

Along with these fundamental functions required to be able to implement the buddy algorithm you have also been provided with functions which will help allow you manipulate the freelist:

- findSmallestFree
- addToLinkedListFront
- deleteFromLinkedList
- splitNodesForLevel
- possiblyMergeAsFarUpAsPossibleRecursive
- freeAll

Brief explanations within "buddy.c" for these functions has also been provided.

2.5.1 before you continue

have a look at the implemented functions provided in a file called "buddy.c". In a file test.c their is a main() procedure that calls the test procedure. You will find some basic testing functions however, these are nowhere near sufficient and only serve as an example. Do extensive testing, to get a better understanding of what the functions are doing. 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. Additionally you can try add elements to the freelist and use then print out the addresses.

2.6 the balloc algorithm

So given that the primitive operations work as intended, it is time to implement the algorithm. Your task is now to implement the balloc() procedure that will be the API of the memory allocator. We do not replace the regular malloc() and free(). We will use one global structure that holds the double linked free lists of each layer.

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

Our flists is basically an array of linked lists where each track the blocks of a specific size. The first time a user requests memory or when we have run out of space in our flists, then your program should utilise the "new" method to instantiate a new initial block of memory. Here we can see the results of our new block where we have one entry in the list at index 7.

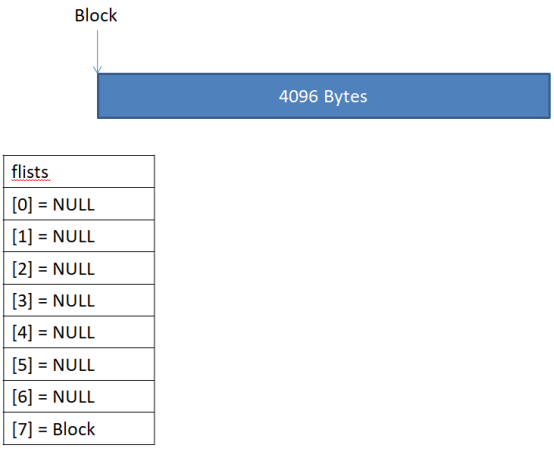

Figure 12: Flists Example

Below is the function structure as well as some psuedocode

```
void *balloc(size_t size) {
//Check if size is 0
//Determine required level based upon size
//Either find a free block at the appropriate level in flists, or
//split the necessary blocks down to that level.
//   if required remembering to update flists and
//   return a pointer to the allocated memory block
//Finally hide the header and return the pointer
}
```

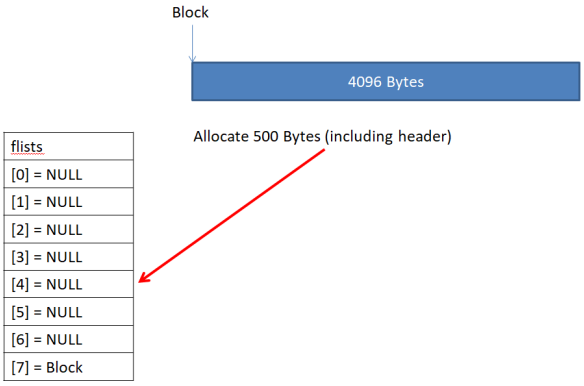

Figure 13: Allocation Example

If we were to allocate 500 Bytes we would need to first determine the index corresponding to the smallest block required to store this amount of data. In this case it would be 4. Since flists[4] is Null we need to split in order to divide our memory into the required block size. If there was already a free block in this list we would only need to return a pointer to it. **IMPORTANT : we keep both allocated blocks i.e. those with a status of STATUS_USED as well as those which are free i.e. those with a status of STATUS_FREE**

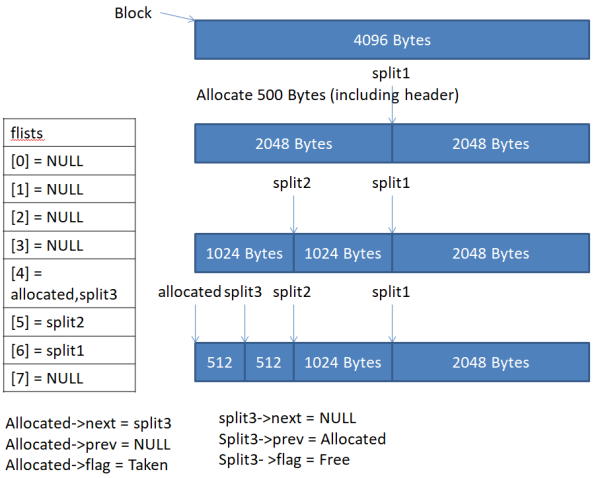

Figure 14: Allocation Example

Here you can see some information demonstrating the process of splitting that would be required to find a pointer to a block of the right size. Additionally you can see the state of flists. Note how it no longer contains an item in flists[7] since that block is no longer available. 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.

3 Submission

There will be a submission link on moodle for you to submit code. In the meantime you should run extensive testing locally to determining if your functions have been implemented correctly. You will be tested on performance of your code in a similar fashion as those tests found in "test.c".**Remember all you need to complete is the balloc function**.

Academic Integrity

There is a zero-tolerance policy regarding plagiarism in the School. Refer to the General Undergraduate Course Outline for Computer Science for more information. Failure to adhere to this policy will have severe repercussions.

During assessments:

- You may not use any materials that aren't explicitly allowed, including the Internet and your own/other people's source code.
- You may not access anyone else's Sakai, Moodle or MSL account.

Offenders will receive 0 for that component, may receive FCM for the course, and/or may be taken to the legal office.