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

COMS3010A
Operating Systems

Project 1A - Buddy

Instructors

COMS3010A Lecturer:
Mr. Branden Ingram branden.ingram@wits.ac.za
COMS3010A Tutors:
Mr. Joshua Bruton 1699140@students.wits.ac.za
Mr. Mark Alence 1676701@students.wits.ac.za
Mr. Tshepo Nkambule 1611821@students.wits.ac.za
Mr. Dylan Brandt 1352906@students.wits.ac.za
Mr. Skhumbuzo Dube 1608194@students.wits.ac.za

Consultation Times

Questions should be firstly posted on the moodle question forum, 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 how to implement and benchmark a simple version of it like we did last week. In this project you will not be given all details on how to do the implementation, but the general strategy will be outlined. In PartA of the project you will be implementing the functions that allow us to perform operations on the blocks and in PartB you will implement the actual Buddy algorithm. A Project Template has been provided on moodle to use. You will be required to add this these files by finishing the implementation of the functions discussed further down.

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 which you can later optimise upon.

2.1 the implementation 2 marks

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).

In this implementation the largest block that we will be able to handle is 4 KB 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: 32bytes, 64, 128 , .. 4 KB. 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 ($2^{12} = 4$KB). When we allocate a new block of 4 KB byte, it needs to be aligned on a 4 KB 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 5 marks

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. Define a "struct" to represent the head. The size of the block should be 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. variables called "next" and "prev" for our block pointers.

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

```
struct head {
//Fill in Here
//
//
//
}
```

2.2.1 a new block 7 marks

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 4 KB byte segment. Here you can take for granted that the segment returned by the mmap() function is in fact aligned on a 4 KB byte boundary. Make sure to initialise the status and level variables as well include a failure check for mmap using MAP_FAILED.

```
struct head *new() {
//Fill in Here
//
//
//
}
```

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.

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

This function should as demostrated above return a pointer to the start of a struct head* object

2.2.2 who's your buddy 4 marks

Given a block we want to find the buddy. The buddy of the block is the other pair of that block.

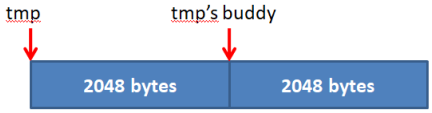

Figure 2: struct head* tmp's buddy = buddy(tmp);

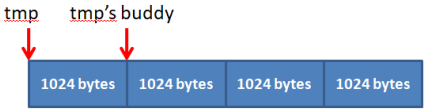

Figure 3: struct head* tmp's buddy = buddy(tmp);

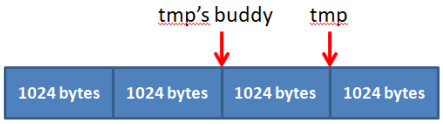

Figure 4: struct head* tmp's buddy = buddy(tmp);

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 one five positions (MIN) to the left we have created a mask that we can xor against the pointer to the block. Alternatively you can make use of the size of the block to shift the pointer either left or right.

```
struct head *buddy(struct head* block) {
//Fill in Here
//
//
//
}
```

2.2.3 split a block 4 marks

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

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. Alternatively you can make use of half the size of the block to shift the pointer.

```
struct head *split(struct head* block) {
//Fill in Here
//
//
//
}
```

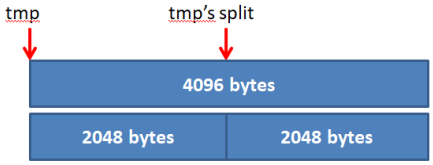

Figure 5: struct head* tmp's split = split(tmp);

2.2.4 merge two blocks 5 marks

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.

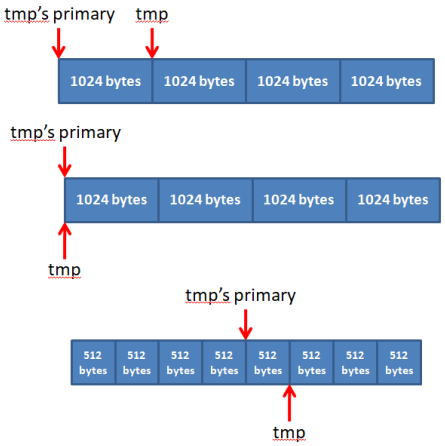

Figure 6: struct head* tmp's primary = primary(tmp);

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. Once again an alternative can be achieved through the manipulation of the pointer with respect to the block size.

```
struct head *primary(struct head* block) {
//Fill in Here
//
//
//
}
```

2.2.5 the magic 4 marks

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.

```
struct head *hide(struct head* block) {
//Fill in Here
//
//
```

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

```
//
}
```

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

Figure 8: struct head* tmp's magic = magic(tmp);

```
struct head *magic(void *memory) {
//Fill in Here
//
//
//
}
```

2.2.6 level 4 marks

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.

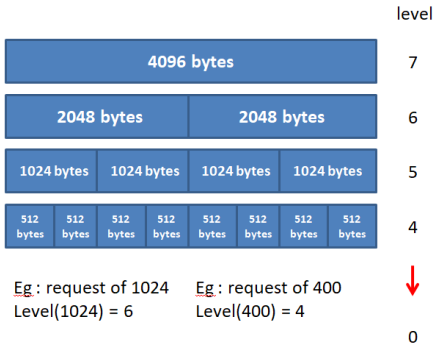

Figure 9: int requiredlevel = level(value);

```
int level(int req) {
//Fill in Here
```

```
//
//
//
}
```

2.2.7 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. You will find some primitive testing already implemented in the test.c file provided.

3 submission

Please upload a zip file containing all files to the moodle submission.

- initial implementation of constants : **2 marks**
- implementation of head structure : **5 marks**
- implementation of new : **7 marks**
- implementation of buddy : **4 marks**
- implementation of split : **4 marks**
- implementation of merge : **5 marks**
- implementation of magic and hide : **4 marks**
- implementation of level : **4 marks**
- **total 35**

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.