Machine Problem 3: Simple Memory Allocator

CSCE-313-501

By: Andrew Schlotzhauer and Jaiden Gerig

March 13, 2015

Introduction

In an ideal system, physical memory should be one contiguous segment from which a memory allocator can take portions of the memory and return them. However this is not so for many systems, and such a system is Linux which partitions the memory into *zones* and are treated differently for separate allocation purposes. Linux also uses a *buddy-system* allocator to allocate and free physical memory. In this machine problem, we are supposed to create a memory allocator that implements this *buddy-system*. What this system does is split memory blocks up into powers of 2 and if a smaller memory block is requested and no block that small exists, a larger block must first be split and keeps splitting until it is the size requested and the unused split memory gets put back together and put in a higher free list, which is the list that contains all of these memory blocks. We are to write a program that will take in user input of basic block size and memory size and initialize the memory and then calls the Ackermann function, which was given to us, and repeatedly call this function with larger numbers for n and m and measure the time it takes. Finally we have to de-allocate the memory before the program exits or aborts.

Procedures

First thing that we did was add the getopt() commands to our main again because we had the code already from Machine Problem 1, it just needed a few changes to variable names. We then set up a few functions to help make it easier on us, such as the next_power_2(), borrowed from a website that was cited in our code, and the Log2() function.

```
//taken from http://graphics.stanford.edu/~seander/bithacks.html
unsigned int next_power_2(unsigned int v){
        v--;
        v |= v >> 1;
        v |= v >> 2;
        v |= v >> 4;
        v |= v >> 8;
        v |= v >> 16;
        v++;
        return v;
}
```

```
// Calculates log2 of number.
long double Log2( double n )
{
    // log(n)/log(2) is log2.
    return log( n ) / log( 2 );
}
```

Next we did the init_allocator() function, which we started setting the memory size to the next power of 2 above what was given, to make it easier to use the blocks. It then checks to make sure if the memory size is double the total length input, if so then it just divides the memory size because it was already a power of 2. Then we set the main_block_addr using malloc() and set the block size to the next power of 2 if the input was not a power of 2. We then add a counter variable and add to the counter every time the total size is divided by two, the reason for this was because it is to set the free_list_size. Then the free_list is initialized using malloc() and the size

and is_free and next variables are set accordingly. Then each index in the free_list is initialized to NULL. Finally the total memory size is returned.

```
unsigned int init_allocator(unsigned int _basic_block_size,unsigned int _length){
        mem_size = next_power_2(_length);
        if(mem_size == _length*2)
                mem_size = mem_size/2;
        main_block_addr = malloc(mem_size + 100000);
        block_size = next_power_2(_basic_block_size);
        int counter = 1;
        unsigned int size = mem_size;
        while(size > block_size){
                counter++;
                size = size/2;
        }
        free_list_size = counter;
        free_list = malloc(free_list_size*sizeof(struct Header *));
        free_list[0] = (struct Header *)main_block_addr;
        free_list[0]->size = mem_size;
        free_list[0]->is_free = true;
        free_list[0]->next = NULL;
        int i;
        for(i=1;i<free_list_size;++i)
                free_list[i] = NULL;

        return mem_size;
}
```

After the init_allocator was finished, we moved onto the release_allocator() and the print_free_lists() in order to properly start testing and try out the program. The release_allocator() just calls free() on the main block of memory and the free_list. The print function just prints out each List and each corresponding block and Address associated with it.

```
int release_allocator(){
        free(main_block_addr);
        free(free_list);
        return 0;
}
```

```
void print_free_lists(){
        int i;
        for(i=0;i<free_list_size;++i){
                printf("List : %i \n",i);
                struct Header* temp = free_list[i];
                while(temp != NULL){
                        printf("Size: %u Free: %i Addr:%p \n",temp->size,temp->is_free,temp);
                        temp = temp->next;
                }
        }
        printf("\n");
}
```

Next we implemented the split function which we were going to make this to be called by our my_malloc() function. This is the function that splits the bigger blocks into the smaller ones in order for the buddy-system to work. We started this function by initializing a temp and prev pointer. Then if the node is NULL we just insert the appropriate sized block to the left or to the right on the current free_list node. Then set the next pointers accordingly, and do kind of the same process but a little changes.

```
struct Header* free = free_list[order+1];
if(free == NULL){
        //insert left
        free_list[order+1] = temp;
        free_list[order+1]->size = (mem_size/pow(2,order+1));
        free_list[order+1]->is_free = true;
        //insert right
        free_list[order+1]->next = (Addr)temp + free_list[order+1]->size;
        free_list[order+1]->next->is_free = true;
        free_list[order+1]->next->size = free_list[order+1]->size;
        free_list[order+1]->next->next = NULL;
        return 0;
}
while(free < temp -(int)(mem_size/pow(2,order+1)) && free->next != NULL){
        free = free->next;
}
```

Next we made the consolidate() function which is what binds two blocks together that aren't doing anything, as described in the instructions. This was done by saying for each free_list node, while the current list index is not NULL, then it checks if the two next lists are free and if they are then it binds them together through different ways after a few checks. If both the next and the next->next nodes weren't free, then it sets the current node to the next->next node. And it keeps iterating like this until the current list index is NULL, and that happens for each free_list.

```
if(temp->is_free && temp->next->is_free){
        if(prev == NULL)
                free_list[i] = temp->next->next;
        else
                prev->next = temp->next->next;
        struct Header* free = free_list[i-1];
        if(free == NULL){
                //insert
                free_list[i-1] = temp;
                free_list[i-1]->size = (mem_size/pow(2,i-1));
                free_list[i-1]->is_free = true;
                free_list[i-1]->next = NULL;
        }
        else{
                while(free < temp && free->next != NULL)
                        free = free->next;
                struct Header* next = free->next;
                //insert
                free->next = temp;
                free->next->size = (mem_size/pow(2,i-1));
                free->next->is_free = true;
                free->next->next = next;
        }
```

Finally, since the functionality was finished, we started implementing the Addr my_malloc(), and extern my_free(), and also added a find_free_node() in order to help the my_malloc() function. So what my_free does, is it takes in an Address of what needs to be removed from memory, and frees that memory and then sets the is_free to true. Find_free_node() takes in an integer for the index inside the free_list which is used to find a free node in that free_list, if it finds one it returns the address of that free node, if it doesn't then it returns NULL.

```
Addr find_free_node(int index){
    struct Header* temp = free_list[index];
    while(temp != NULL && !temp->is_free){
        temp = temp->next;
    }
    if(temp != NULL){
        temp->is_free = false;
        return(Addr)temp+sizeof(*temp);
    }
    else
        return NULL;
}
```

```
extern int my_free(Addr _a) {
  struct Header* temp = (_a - sizeof(struct Header));
  temp->is_free = true;
  return 0;
}
```

So finally my_malloc() can be successfully implemented using these functions, but my_free() isn't called from my_malloc() that is called from the main when the program is ran. So my_malloc() is started by getting the size of the memory and if it isn't a power of two it makes it a power of 2 by calling the next_power_2() function. The first if statement is just checking if it was already a power of 2, in which case it just makes it the original by dividing it by 2. Then it adds the size of the struct Header and then calls our function consolidate() which combines the lists if split. Then the while loop iterates while it doesn't have a return address, and calls the find_free_node() which will return the address of a free node in the current free_list. If it doesn't find a free node then it splits that memory block and keeps iterating in this way until it gets an address.

```
extern Addr my_malloc(unsigned int _length) {
    int alloc_size = next_power_2(_length);
    if(alloc_size  == _length*2)
        alloc_size = mem_size/2;
    alloc_size = next_power_2(alloc_size + sizeof(struct Header));
    consolidate();
    int index = Log2(mem_size/alloc_size);
    int offset = 0;
    Addr return_addr = NULL;
    while(return_addr == NULL){
        return_addr = find_free_node(index);
        if(return_addr != NULL){
            return return_addr;
        }
        split(offset);
        offset++;
        if(offset >= free_list_size){
            break;
        }
    }
    return NULL;
}
```

Results

When the program is ran, it first prints out each list corresponding to the given memory in the init function. In the picture below, the lists actually go on until List 17 because the next power of 2 of that given size can be split 17 times until it reaches our basic block size which is 8.

```
[schlotzh]@linux2 ~/313_MP3> (13:22:48 03/14/15)
:: ./memtest
List : 0
Size: 1048576 Free: 1 Addr:0x7f6f04615010
```

Then Ackerman is called on this and this is the output.

```
     n = 1, m = 1
Result of ackerman(1, 1): 3
Time taken for computation :  [sec = 0, musec = 186]
Number of allocate/free cycles: 4
```

```
     n = 1, m = 2
Result of ackerman(1, 2): 4
Time taken for computation :  [sec = 0, musec = 208]
Number of allocate/free cycles: 6
```

```
     n = 1, m = 3
Result of ackerman(1, 3): 5
Time taken for computation :  [sec = 0, musec = 611]
Number of allocate/free cycles: 8
```

```
     n = 1, m = 6
Result of ackerman(1, 6): 6
Time taken for computation :  [sec = 0, musec = 455]
Number of allocate/free cycles: 14
```

```
     n = 1, m = 8
Result of ackerman(1, 8): 10
Time taken for computation :  [sec = 0, musec = 835]
Number of allocate/free cycles: 18
```

```
     n = 2, m = 1
Result of ackerman(2, 1): 5
Time taken for computation :  [sec = 0, musec = 451]
Number of allocate/free cycles: 14
```

We ran this more times than shown, but those are just some picked at random. From what we can tell there is no memory checking error! Message popup so it seems to work, and the results of the Ackermann function is correct but there was no way to know for sure if the other things are correct considering we weren't given an example of output.

Conclusion

Our memory allocator works well for the Ackermann function for the smaller values of n and m but takes a while up on the higher values, as expected. And after split is called and consolidate is called it puts the blocks back together each time in the same memory spot as the original block which is how it is supposed to work. Overall our program worked as expected, except in earlier versions our memory got corrupted and messed some things up but that was fixed and works for the specifications required. Although with this type of system a lot of wastage occurs, such as if the user wants a memory size of (power of 2 + 1) and gives you a block size and if it is a large number like 2^20 + 1 then we take a memory size of 2^21 and split it which wastes a whole lot of memory. An analysis of this in the other submitted pdf, called Analysis.pdf as instructed.