Seminar 2. Buddy Algorithm

Albert Asratyan

November 29th, 2018

1 Introduction

The point of this assignment is to implement our own *malloc()* function using the buddy algorithm for finding and merging free blocks. The buddy memory allocation method is a memory allocation algorithm that divides memory into smaller parts to try to fit a requested amount in the memory in the most suitable way. This is done by splitting the memory in half at the each step.

2 The Buddy Algorithm

When doing this assignment, most of the functions have been given to us, with the exception of two functions, find() and insert(). The whole infrastructure of the algorithm consists of a doubly linked list that stores buddy blocks. The list contains information such as the level of the block, status of the block, next and previous blocks, if available. The page size chosen for the assignment is 4096 bytes and the list can have up to 8 levels, ranging from 32 bytes to 4096 bytes.

When allocating memory, *balloc(size_t size)* function is called. It finds a corresponding level for the required amount of memory and allocates memory using the find() function. Also, it hides the header of the block, so it would not be overwritten accidentally.

3 Find Function

find(int index) takes in one argument, index, that tells the method on what level the block should be allocated. For example, calling *balloc(5)* invokes *find(0)*, which means that there should be a block allocated at level 0. From here, find takes care of the rest of the procedure, which can be described by the following pseudo code:

```
function find(int index) {
    if (index < LEVELS) {
        if (flists[index] exists) {
            if (next block exists) {
                flists[index] = next block
```

```
            } else {
                flists[index] = NULL
            }
            // Take the block from the list:
            change status to Taken
            change prev to NULL
            change next to NULL
            return the block
        }
    } else {
        findBlock(index + 1)
        return find(index)
    }
    return NULL
}
```

findBlock(int index) is a function that takes care of finding a suitable block in the levels above. It could be substituted by having a for loop in the *find()* function, but was decided to be split into a separate function for simplicity. *findBlock()* can be described by the following pseudo code:

```
function findBlock(int index) {
    if (index < LEVELS) {
        if (flists[index] exists) {
            block1 = flists[index]
            block2 = split(block1)
            if (block1 -> next exists) {
                block1 = block1 -> next
            } else {
                block1 = NULL
            }
            //Free block1 and move it one level down
            block1-> status = Free
            block1->level = index - 1
            block1->next = block2
            block1->prev = NULL
            //Free block2 and move it one level down
            block1-> status = Free
            block1->level = index - 1
            block1->next = NULL
            block1->prev = block1

            flists[index - 1] = block1
            return NULL
        } else {
            findBlock(index + 1)
            //Go a level up, if no block is available
```

```
        }
    } else {
        block = new()
        //Create a new 4 Kb block if no memory is available
        flists[LEVELS - 1] = block
    }
}
```

As it can be seen from the code above, find checks whether there is an empty block on the required level. If there is, it allocates it. If there are no available blocks, it goes a level above and checks again and again, until it finds one. If there are no blocks whatsoever, it creates a new page, and then checks for an empty block again. *find()* utilizes *split()* that splits a block by adding another header, relative to the first header.

Personally, the most difficult part of writing this function was connecting the blocks correctly after splitting them into smaller parts. If done wrong, the sequence of the blocks will be mixed up and there is no way of checking how close the solution is to being right, it is either right or wrong.

4 Insert Function

bfree(struct head *block) is an implementation of malloc's free() function. It reveales the header of the block, and then calls insert(struct head *block) to return the block's memory back to the list. insert() can be described by the following pseudo code:

```
function insert(block) {
    block->level = Free
    if (block->level < LEVELS - 1) {
        buddyBlock = buddy(block)
        if (buddyBlock == Free and buddyBlock->level == block->level) {
            level = block->level
            mergedBlock = primary(block)
            mergedBlock->level = level + 1
            mergedBlock->status = Free
            mergedBlock->prev = NULL
            mergedBlock->next = NULL
            flists[level] = NULL
            //check if can be merged again
            insert(mergedBlock)
        } else {
            //otherwise put it in the list without merging
            block->next = NULL
            block->prev = NULL
            flists[block->level] = block
        }
```

```
    } else {
        flists[LEVELS-1] = block
    }
}
```

insert() checks whether the buddy of the block is available for merging. If it is, then the blocks are merged, and the procedure is repeated until no further blocks can be merged. If no blocks can be merged from the start, the function just returns the block back to its spot. Two functions are used here, buddy() and primary(). Buddy() find a corresponding buddy, whereas primary() merges two buddies.

After completing find() and familiarizing ourselves with the concept of buddy algorithm, this function was relatively easy to implement.

5 Benchmarks

5.1 Plotting Memory

Two major benchmarks have been completed in order to check whether the implemented algorithm works as expected. For the first benchmark, 100 blocks were randomly allocated between 0 and 4Kb, and the graph below shows the distribution with three lines, representing allocated data, used data, and available data. Available data is the maximum data that could be used, represented in amount of 4Kb blocks. Allocated data is the memory used for storing data from the random variables. Used data is the memory that is taken by the actual variables. It can be seen that allocated data is pretty close to the available data, which means good utilization. However, used data is quite different from the allocated data. This is a limitation of how buddy algorithm works. It increases the chunks of data by a factor of two, and sometimes it can waste quite a bit of memory, which can be observed in the later stage of the graph, where big chunks of memory are allocated, that are slightly above the limit of the previous size. Because they exceed this limit, they have to take the next level, which requires twice the memory.

5.2 Speed and Comparison with Malloc()

To check how fast the algorithm is, a small piece of code was written, that counts the amount of clock ticks of the CPU. It has been run for both balloc() and malloc(). The code executes balloc() or malloc() for 100 times and records the difference in ticks. This gives a good relative picture to how balloc() performs in relation to malloc(). The code for the benchmark is below:

```
void bench3(int iterations, int memory) {
  double total_t;
  clock_t start_balloc = clock();
  for(int i = 0; i < iterations; i++) {
    int *ballocate = balloc(memory);
```

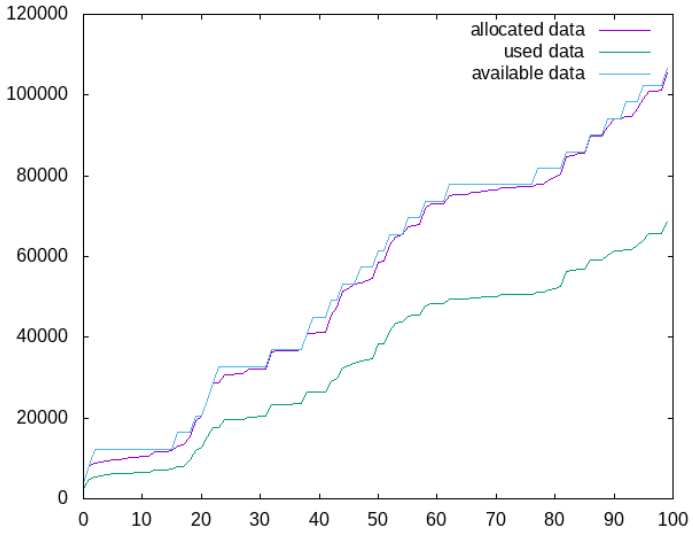

Figure 1: 100 rounds of balloc

```
    bfree(ballocate);
  }
  clock_t end_balloc = clock();
  clock_t balloc_time = (end_balloc - start_balloc);

  total_t = (double)(balloc_time);
  printf("Total time taken by balloc: %f\n", total_t   );

  clock_t start_malloc = clock();
  for(int i = 0; i < iterations; i++) {
    int *mallocate = malloc(memory);
    free(mallocate);
  }
  clock_t end_malloc = clock();
  clock_t malloc_time = (end_malloc - start_malloc);

  total_t = (double)(malloc_time);
  printf("Total time taken by malloc: %f\n", total_t   );
}
```

The results are presented in the table below and are quite interesting. For the smaller block size, balloc() is much slower than malloc(). However, the closer the block size to the maximum size of 4Kb, the faster balloc() becomes, even outperforming malloc() at the biggest sizes. For example, they even out at the level 6 (2000 bytes input) and at the level 7 (last level) balloc() allocates

size	balloc	malloc
32	84	8
64	57	7
100	59	7
220	45	7
480	41	8
1000	26	7
2000	16	16
4000	9	12

Table 1: Processor Tick Comparison

faster than malloc().

6 Conclusion

To sum up, I would consider that the implementation of the buddy algorithm was successful, because the allocated memory utilization is almost at the maximum and the algorithm performs very fast when little splitting has to be done. However, where multiple blocks need to be split up, the algorithm shows its weakness, but this is rather a conceptual problem than problem of the given implementation.