University of Arizona, Department of Computer Science

CSc 340 Foundations of Computer Systems — Assignment 6

Christian Collberg

Due Wednesday, May 2, at 11:59pm

1 Overview

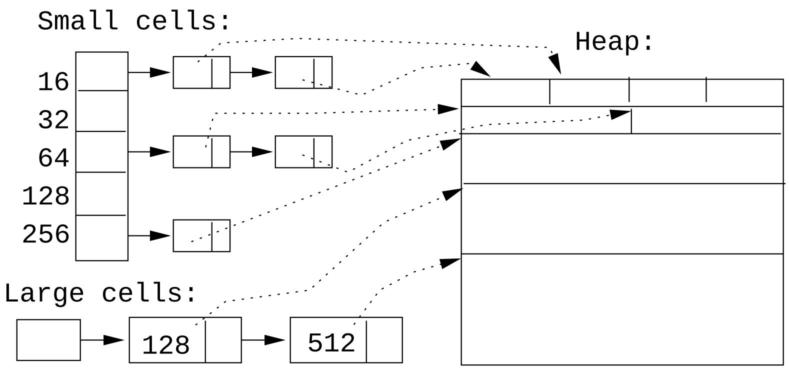
For this programming assignment you will write a heap manager in C. The heap manager is first initialized by calling the "real" malloc to allocate a region of memory of a particular size. The application then makes requests of the manager to allocate memory from the region and to free memory when it is no longer needed. The heap manager can display information about the regions in the free lists.

2 heap.h

The following is the contents of `alloc.h`, the header file for the heap manager. Applications that use the heap manager must include this file.

```
/* Create a heap of size 'heapSize'. Use 'malloc' from stdlib. */
extern void alloc_init(unsigned int heapSize);

/* Destroy the heap and return the memory to stdlib's 'free'.*/
extern void alloc_done();

/* Allocate and return a block of data of size 'size'.
 * Return NULL if no memory is available. Print an error
 * message to stderr if a magic number has been corrupted.
 */
extern void* alloc_malloc(unsigned int size);

/* Return a block of memory previously allocated by alloc_malloc.
 * Print an error message to stderr if a magic number has been
 * corrupted.*/
extern void alloc_free(void* m);

/* Print the elements of the free lists. */
extern void alloc_print();

/* Return the amount of memory available on the free lists. */
extern int alloc_available();

/* Print an error message if the free lists have been corrupted.*/
extern void alloc_check();
```

The module should be implemented using *binning* as shown in lecture. I.e. you should have one free list containing large free elements, and one array of lists of small free element:[1]



You should only allocate memory in multiples of 16-byte chunks, and you should never allocate anything smaller than 16 bytes. For example, ⌜`alloc_malloc(100)`⌝ should return a 112-byte memory block.

You should use a first-fit strategy for allocating from the large list. If the free block is larger than the amount of memory requested, the remainder of the block should be put back on the list. If, however, the remainder is less than 16 bytes long it is returned together with the allocated object.

The array of small free elements should have 8 buckets:

```
typedef struct alloc_smallFreeNode {
  struct alloc_smallFreeNode* next;
  int magic;
} alloc_SMALLFREENODE;

typedef struct alloc_largeFreeNode {
  struct alloc_largeFreeNode* next;
  int magic;
  int size;
} alloc_LARGEFREENODE;

#define BUCKETS 8
int bucketSizes[] = {16,32,48,64,80,96,112,128};
#define MIN_NODE_SIZE 16
#define MAGIC 0xfeedface
```

To allocate from the small bins you should move free elements from larger bins to smaller bins. For example, if during a call to ⌜`alloc_malloc(16)`⌝ you realize that the size-16 bin is empty you should

1. call ⌜`alloc_malloc(32)`⌝ to allocate a 32-byte chunk,
2. split this chunk in two 16-byte sub-chunks,

[1] Note that this picture is deceiving. The linked lists of small cells are not allocated *outside* the heap as is shown here, but are rather part of the free heap elements themselves, as shown in lecture.

3. call ⌜alloc_free()⌝ on one of the 16-byte sub-chunks to return it to a free list,
4. return the other 16-byte sub-chunk to the user.

The idea is that if the user is calling ⌜alloc_malloc(16)⌝ once, he'll probably do it again, so we might as well allocate two 16-byte objects when we're at it.[2]

⌜alloc_free(p)⌝ frees memory previously allocated by alloc_malloc() to one of the free lists. Note that the size of the block is not provided. Instead, each allocated node has an 8-byte hidden header consisting of the size and magic number of the object:

```
typedef struct alloc_allocNode {
  int size;
  int magic;
} alloc_ALLOCNODE;
```

⌜alloc_print()⌝ prints the heap in the following format:

```
16:
32: 0x22718 0x22796
48:
64:
80:
96:
112:
128: 0x227e0
LARGE: 0x227b8/144
AVAILABLE: 144
```

I.e., there is one line per small bucket and one line for the large bucket. On the small bucket lines we write the addresses of the free elements. On the large bucket line we also give the size of the free element. AVAILABLE gives the size in bytes of free memory on the heap, as returned by ⌜alloc_available()⌝.

3 Implementation

As an implementation strategy I would first make sure that the following program works:

```
alloc_init(256);
alloc_print();
```

[2]In a real implementation we might keep statistics over how many objects of each type has been allocated and pre-allocate objects of popular sizes. So, for example, every time the user calls ⌜alloc_malloc(16)⌝ and there are no more objects of this size available, we pre-allocate twice as many 16-byte objects as we did last time.

Then I'd try to allocate something off the large free list:

```
char *p;
alloc_init(256);
p = alloc_malloc(128);
alloc_print();
```

When that works, I'd try to allocate some small objects, with only two buckets:

```
int *p,*q;
#define BUCKETS 2
int bucketSizes[] = {16,32};
alloc_init(256);
p = alloc_malloc(sizeof(int));
q = alloc_malloc(sizeof(int));
alloc_print();
```

As always, you should work in small, incremental, steps, making sure that every simple task works before you move on to more complex tasks.

Remember that pointer arithmetic in C is type-specific. If `ptr` is 100, then `ptr+1` is 101 if `ptr` is of type ⌜`char *`⌝, 102 if it is of type ⌜`short *`⌝, and 104 if it is of type `int *`, etc. You'll have to cast pointers to the appropriate type, e.g. if you have a pointer `ptr` of type ⌜`void *`⌝ and you want it to be of type ⌜`Node *`⌝, you have to cast it: ⌜`(Node *) ptr`⌝.

A test file `/home/cs340/spring01/prog6/test1.c` will be provided for you. You should, however, create your own tests as well.

4 Extension 1 [10% extra credit]

Implement coalescing. You should do this both on the large free list and on the bins. For example, two adjacent 16-byte blocks can be merged and moved to the list of 32-byte blocks. You may find that keeping the free lists sorted by address will make your work easier.

5 Extension 2 [10% extra credit]

Implement a next-fit allocation policy for the large free list.

6 Extension 3 [10% extra credit]

Write a Java program that implements a database of names and telephone numbers. This program should make use of `hash.c` from the previous assignment which, in turn, should use your own `alloc.c` from this assignment. Good luck.

7 Honors Section

The honors students should sit back and relax.

8 Turnin

You should turn in your own `makefile`, and two files `alloc.h` and `alloc.c`. `alloc.c` should compile "out-of-the-box" when we type `''make''`.

If you have implemented any of the extensions you should describe them in the README file, so that the graders know what to test for.

This assignment can be done in teams of 1-2 students. In your README file, state clearly who are the members of your team, and what is the contribution of each team member. You could, for example, write:

> This assignment was completed by **Bob** (`bob1@cs.arizona.edu`) and **Alice** (`alice2@cs.arizona.edu`). We both agree that Alice did 60% of the work and Bob the remaining 40%. We accept the fact that the credits for this work will be distributed accordingly.

This assignment can be done individually or in teams of two (2) students. Regardless, don't show your code to anyone outside your team, don't read anyone's code, don't discuss the details of the code with anyone. If you need help with the assignment see the TA or the instructor.