CS-3411 Program I : Husky Malloc

Fall 2020
Due: September 24, Midnight

In this project, we will develop a relatively simple implementation of malloc. Although it is significantly simplified, it will still provide the necessary functionality to support memory allocation in any C program. We refer to our implementation of malloc *Husky malloc* (hmalloc). Hmalloc will use the *sbrk* system call to request memory from the operating system. An abbreviated manual page is given below:

```
void *sbrk(intptr_t increment);
```

sbrk() changes the location of the program break*, which defines the end of the process's data segment (i.e., the program break is the first location after the end of the uninitialized data segment). Increasing the program break has the effect of allocating memory to the process; decreasing the break deallocates memory.*

sbrk() increments the program's data space by increment bytes. Calling sbrk() with an increment of 0 can be used to find the current location of the program break.

Return value: *On success, sbrk() returns the previous program break. (If the break was increased, then this value is a pointer to the start of the newly allocated memory).*

As it can be seen from the description above, one could drop-in replace malloc references in a given C program with sbrk kernel calls, and it should work in terms of allocating memory. Unfortunately, freeing using sbrk is only possible if the last allocated memory segment is freed first. This would be a severe limitation for C programmers, as the allocation and freeing of memory is done freely in any given C program. Hence, memory allocation functions implemented in C library, malloc and its friends, calloc, realloc, and free provide the necessary convenience to the programmer.

There are many implementations of malloc and friends with varying capabilities. Most of these implementations act by maintaining a *free memory area pool* typically organized in the form of *bins*. During the initialization of malloc routines, several pages of memory are requested from the kernel, divided and placed into the bins in anticipation of requests to come. Whenever a malloc request is received, a segment from the nearest size bin is allocated and returned to the user. When there is nothing suitable, the program break is extended by several pages, part of the obtained memory is returned the user and the remaining available space is placed into bins. Most implementations keep track of the usage within a given page. When free is called to release the last used space in a given page and that page is the last page before the program break, the program break is reduced by the size of a page, essentially returning the memory to the kernel. If not, freed memory area is placed into an appropriate sized bin. In other words, production quality memory allocators typically work with page sized objects. In addition to this smart management, most state of the art memory allocators function correctly when they are called from multi-threaded programs, which means their internal structures are protected with appropriate locks.

While it is not very difficult to write a full-blown dynamic memory allocation mechanism for single-threaded programs, it still is a significant amount of time committment. We will therefore simplify the task as below.

1. **hmalloc** will not allocate an initial pool from the system and will not maintain a bin structure.
2. **hmalloc** is permitted to extend the program break by as many as user requested bytes (plus length and link information).
3. **hmalloc** keeps a single *free list* which will be a linked list of previously allocated memory areas that have been freed.
4. **hmalloc** traverses the free list to see if there is a previously freed memory area that is at least as big as the requested size. If there is one, it is removed from the free list and returned to the user.
5. **hfree** simply appends the returned area to the beginning of the single free-list.

Approaching the Problem

It is strongly encouraged that you follow a systematic approach in developing the necessary software. The following steps should help you to have a functioning implementation at all times:

1. Write a simple program on your own, similar to the program built during the discussion of the memory allocation mechanisms in Unix. This program should call only malloc and do not free any memory.

2. Replace malloc with a skeleton function hmalloc, which simply calls sbrk passing along its argument and returning the result. Verify that your program correctly works after this change.

3. Modify the program so that it allocates **two more words than the requested size**, similar to what was done in the class example. Store the length of the area in the first extra word, and set the next one will serve as a link to zero. Verify the program still works.

4. Modify your program by introducing hfree as a skeleton function which does nothing. Modify the main program so that it performs a number of mallocs/frees. Verify again that everything works.

5. Begin implementing hfree by introducing a free-list head pointer at the global which is initialized to NULL. Each time hfree is called, save the free-list head into a temporary, set the free-list head to the returned address, and set the link word of the returned area to the distance between the area pointed to by the temporary and the new area. Verify that everything still works.

6. Write a procedure called *traverse* which can start at the free-list head, visit and print the length of each area in the free pool. After a number of free calls from the main, call this procedure to see that you can correctly traverse the free list.

7. Modify hmalloc so that if the free-list pointer is null it just calls sbrk. If it is not null and an entry large enough to service the request is found, it returns the area pointer at the free list entry, and modifies the free list links to the next one by using pointer arithmetic (i.e., add the old free list pointer the link word value) and store it in the free list.

8. Modify your main program so that it performs a number of hmalloc followed by a number of hfree calls followed by a number of hmalloc calls. Before and after each hfree and hmalloc call, call *traverse* to see your free list growing and shrinking. If you made it this far, you are almost done.

9. Implement hcalloc by first calling hmalloc and zeroing the returned area before returning the pointer to the user.

10. For bonus points, implement hrealloc by obtaining the requested size memory using hmalloc, then copying over from the old area and finally calling hfree to free the old area.

Pragmatics

Source Template

You are provided a source template that includes:

- Makefile with the following commands:
  - all - compile project into *hmalloc*.
  - debug - compile project into *hmalloc* with debugging info for use with GDB/Valgrind.
  - clean - removes executable and object files.
  - submission - generates prog1.tgz with all source files needed for submission. Upload this file to Canvas.
- hmalloc.h - header with prototypes for hmalloc functions
- hmalloc.c - C source where you will implement *hmalloc*, *hfree*, *hcalloc* and *traverse*.
- main.c - C souce where you will implement tests that call the functions defined in hmalloc.c.

Submission Requirements

Your submission must be written in C.
Use Canvas to submit a tar file named `prog1.tgz` generated by running "make submission". Your souce modifications must include comments.

Grading

Your submission will be graded on completeness and correctness of *hmalloc*, *hfree*, *hcalloc* and *traverse*. Correct execution of these functions will result in full marks. If not fully functional, partial credit may be awarded. The main.c provided will be replaced when testing your submission.

Example 1
This example shows the process of allocating, freeing then reusing a chunk of memory.

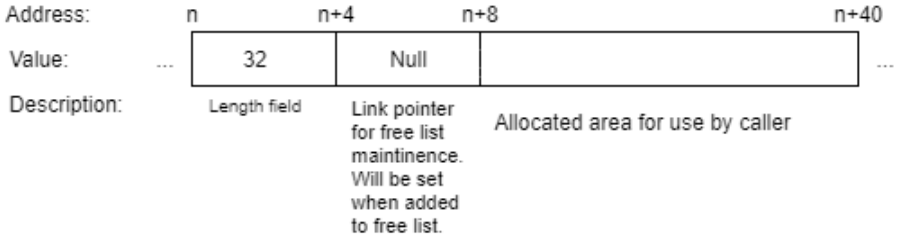
Free List: NULL

Alloc

call hmalloc(32 bytes)

//free list is empty so we will call sbrk

sbrk(40 bytes) //32 bytes plus an extra 2 words (4 bytes per word) for metadata



return pointer to n+8 //beginning of allocated area.
//Prior bytes are for use in hmalloc and hfree

Free

call hfree(p) //p is n+8 returned by hmalloc
//since the free list is null, we dont have to manage any links,
//just set the head to this free area. If free list is not null,
//update the link of this entry to point to the old free list head.

Free list = p-8 //set the free list head to point to metadata about this entry

---

Re-use

call hmalloc(8 bytes)

//iterate over free list starting with the head
//if the length field will fit the requested allocation, we will re-use this entry
//if it doesn't fit, check the next list entry by following the link pointer
//if no entry will fit this request, call sbrk()

//in this example, the first free list entry will fit the request
//remove this entry from the free list by updating links as necessary

temp = Free List
Free List = NULL

return temp+8

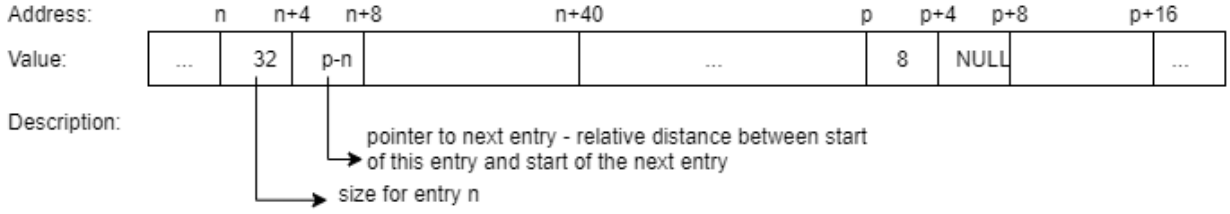
//When reusing an area from the free list, you do not change the length field.
//If it fits use it, if not move on. You do not need to split or combine free areas.

Example 2

This example shows the organization of the free list in memory. Note that the link field is not a pointer in the canonical sense (8 byte memory address). Rather, the link is a relative offset to the next list entry from the current address.

Let us assume for this example there are 2 entries in the free list. Below shows the organization and use of the "link"

Free List = n



The free list head pointer contains an absolute address as the pointer to the first entry in the list.
Each subsequent entry is reachable via the "link" which is the relative offset in bytes from the current entry.
Thus to traverse the link from n to p, we take the address of n and add the offset stored in n's "link" field.
Since p is the last entry, its "link" is null signifying the end of the list.