Computer Science Department

MHC Mount Holyoke College

CS 322
Operating Systems
Programming Assignment 4
Writing a memory manager
Due: April 11, 11:30 PM

Goals

- To understand the nuances of building a memory allocator.
- To create a shared library.

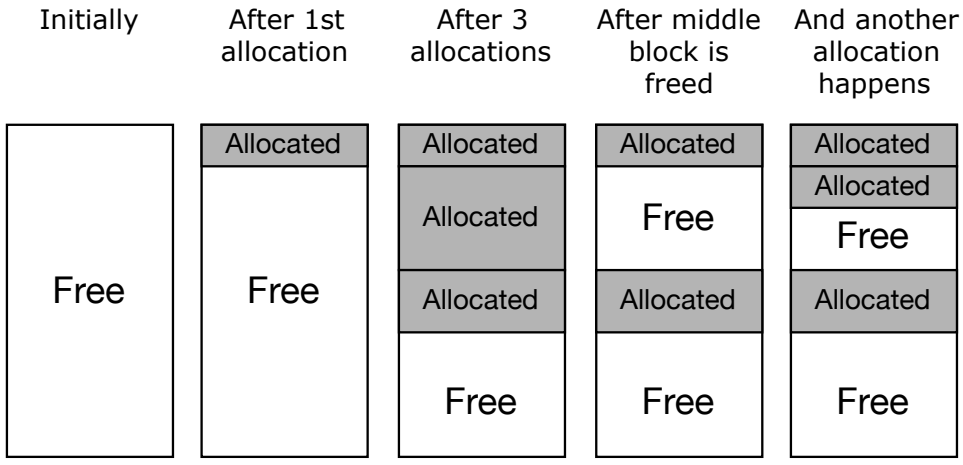
Background

A memory allocator has three distinct tasks:

1. The memory allocator asks the operating system to expand the heap portion of the process's address space by calling `mmap`. In a paged memory system, this request would be for some multiple of pages. (This is the easy part.)
2. The memory allocator doles out this memory to the calling process when the process calls `malloc`.
3. When the process later calls `free`, the freed chunk is added back to the pool of memory that can be given out again on a later `malloc` call.

To do this, the memory allocator needs to know where the free chunks of memory are. Initially, there is just one large chunk as it is all free. When memory is allocated out of this single large chunk, we still have one chunk, but it is smaller.

When a chunk is freed, we may now end up with multiple free blocks. The memory allocator will keep the free chunks in a list. When `malloc` is called, it searches the free list to find a contiguous chunk of memory that is large enough for the process's request. When the process calls `free` to free the memory, the memory allocator adds the freed chunk back to the free list.



The memory allocator operates entirely within the virtual address space of a single process and knows nothing about which physical pages have been allocated to this process or the mapping from virtual addresses to physical addresses, or about any other processes that are running. Process A's memory allocator manages a different chunk of memory than process B's memory allocator.

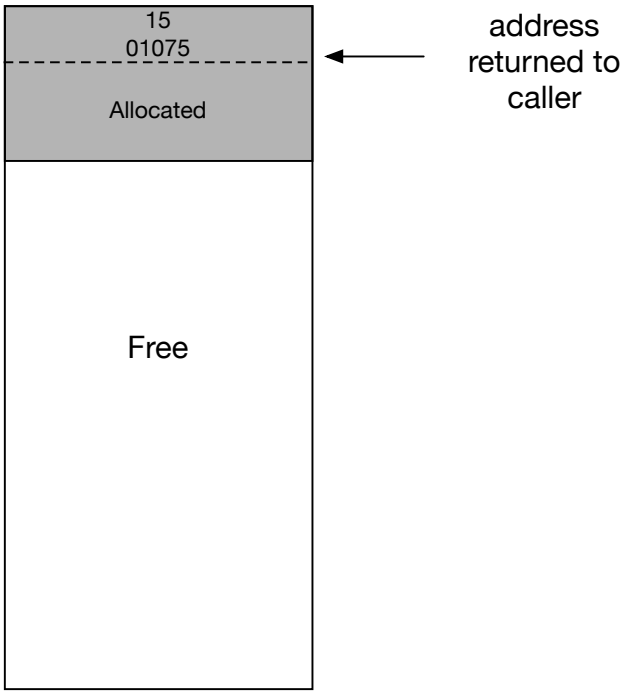
C's `malloc`() and `free`() are defined as follows:

- `void *malloc(size_t size)`: `malloc`() allocates `size` bytes and returns a pointer to the allocated memory. The memory is not cleared.
- `void free(void *ptr)`: `free`() frees the memory space pointed to by `ptr`, which must have been returned by a previous call to `malloc`(). Otherwise, or if `free(ptr)` has already been called before, undefined behavior occurs. If `ptr` is NULL, no operation is performed.

Keeping track of the size of malloc'ed memory

One interesting thing that you should notice is that while we tell the memory allocator how much memory we want when we call malloc, we do not tell it how much memory to free when we call free. The memory allocator keeps track of this information itself in the following way.

Every time that the memory allocator allocates memory it actually uses a little more memory than requested. In particular, it creates a small header that holds two pieces of information: the size of the chunk allocated and a "magic number" that helps it ensure that this really is a header for an allocated chunk (kind of like a password but it's not secret). If we zoom in on the memory of an allocated chunk, we would see something like the figure at the right if the user requested 15 bytes.



The header first contains the size of the allocated block, which is 15 bytes. Then, there is a magic number, 01075, in this case. Following that is the block of size 15 The address that is returned to the caller is where the 15 bytes start, not where the header is.

The reason the memory allocator needs the header is so that free can work correctly. When free is called, we pass in a pointer. If this address is an address that was returned by malloc, then it knows that immediately before the address should be the magic number. It checks for this. If it is not the magic number, free will fail because it knows this is some random address, not an address corresponding to memory that was malloc'ed.

If it finds the magic number, then it looks at the preceding int to see how big the block is so it knows how much to free.

Keeping track of the free list

The memory allocator's second challenge is keeping track of where all the free blocks are. To do this, it reuses the same chunk of memory that was used for the malloc header. Now, however, it contains the size of the free block. Instead of the magic number, it contains the address of the next free block.

Figure 17.3 from the text, shown here, is what would be in the heap immediately after it is initialized. Initially, we requested 4K from the OS for the heap. The size that is actually available is 4K - 8 since 8 bytes are used up by the header. The first word of the header shows this size, while the second word has a value that indicates that there is no other free block.

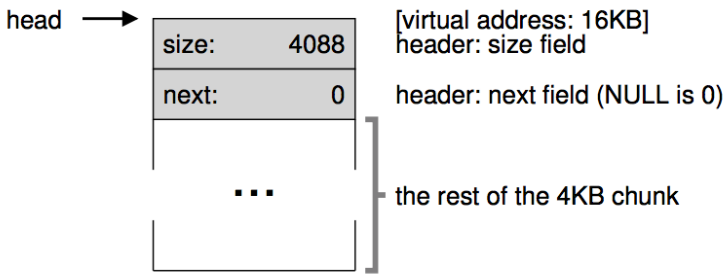


Figure 17.3: **A Heap With One Free Chunk**

Figure 17.4 shows what happens when 100 bytes is allocated. The header turns into an allocated block header, with a size of 100 followed by the magic number. Then there is the 100 blocks that is given to the process to use. The free chunk now follows this allocated memory. The header on the free chunk shows its size (4088 - 100 - 8 = 3980) and again it is the only free block so next is 0.

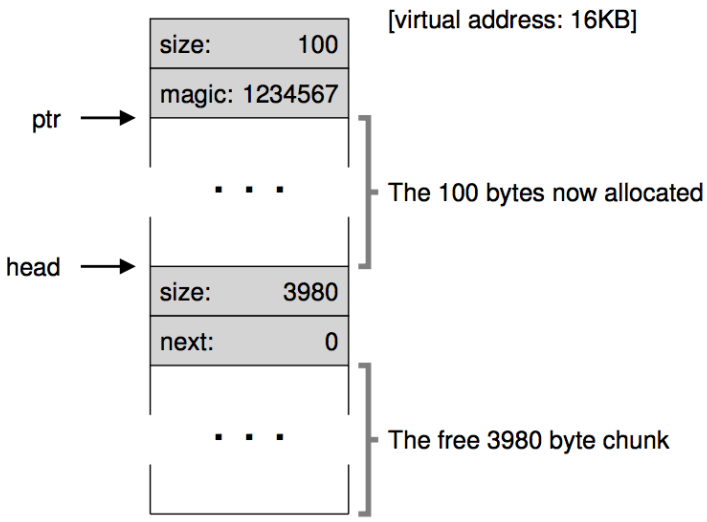


Figure 17.4: **A Heap: After One Allocation**

After 3 allocations, we may have something like what is shown in 17.5. Here there are 3 chunks of allocated memory (preceded by their headers) and then the free chunk (preceded by its header).

Now when the chunk pointed to by sptr is freed, we will have 2 free chunks. Figure 17.6 shows how these free chunks are chained together.

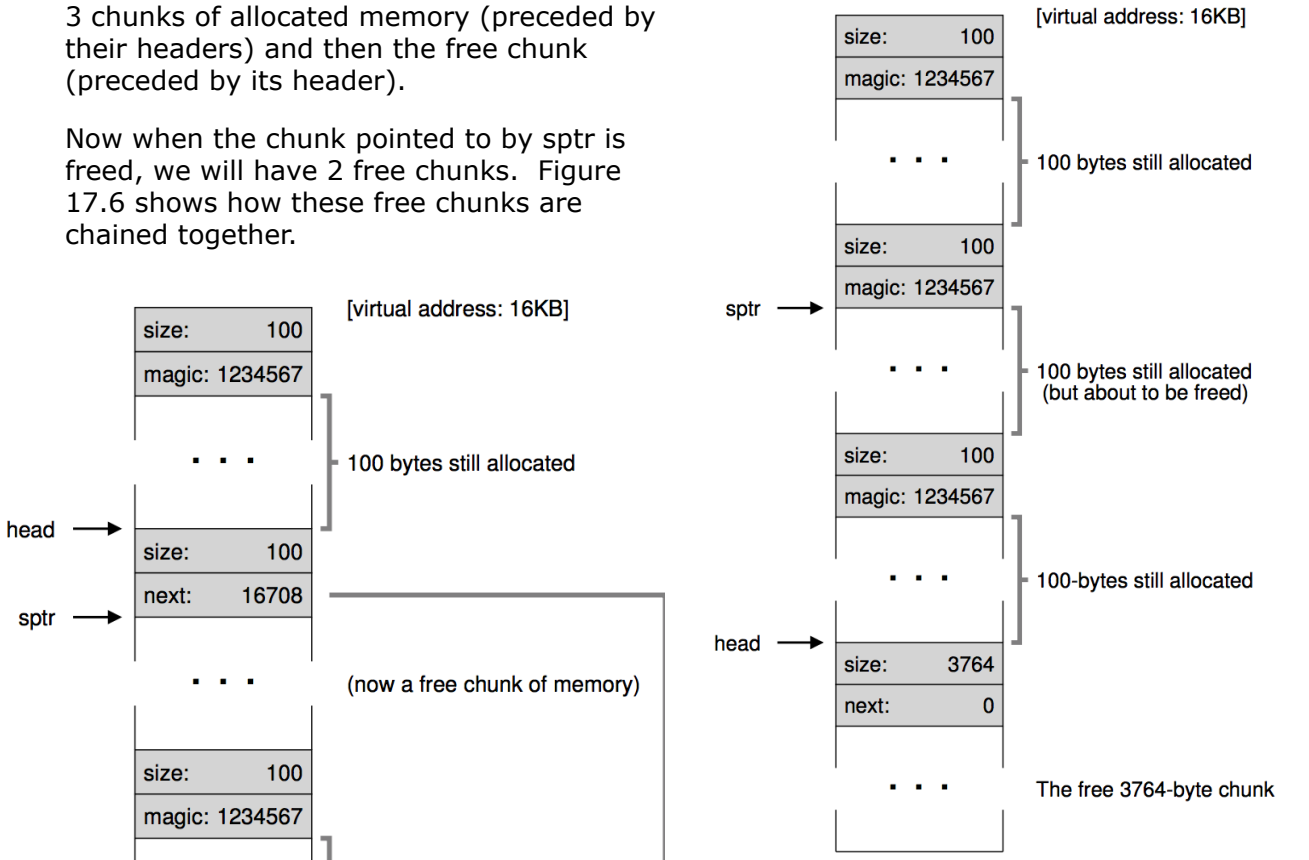


Figure 17.5: **Free Space With Three Chunks Allocated**

Figure 17.6: **Free Space With Two Chunks Allocated**

Confused? Read chapter 17. It provides all the details of how a memory manager works.

Assignment

In this project, you will be implementing a memory allocator for the heap of a user-level process that behaves like described above. Your memory allocator should provide these functions that can be called from user processes:

- `void *initMem (int sizeOfRegion)` - This function should call `mmap` to request `sizeOfRegion` bytes of memory to manage. Note that you may need to round up sizeOfRegion so that you request memory in units of the page size (see the man pages for `getpagesize`()). `initMem` should return the address of the region returned by `mmap`. (This will be used by my tests.) If mmap fails, return NULL.

- `void *allocMem (int size)` - This function should behave the same as malloc. It takes as input the size in bytes of the object to be allocated and returns a pointer to the start of that object. It should return NULL if there is not enough contiguous free space to satisfy the request.

- `int freeMem (void *ptr)` - This function should behave the same as free. It frees the memory object that `ptr` points to. Just like with the standard `free`(), if `ptr` is `NULL`, then no operation is performed and 0 is returned. The function returns 0 on success, and -1 otherwise.

- `void dumpMem()` - Use this for debugging. You can have it print out whatever you find helpful, such as the headers for all the blocks in the free list.

These functions will be part of a shared library. As a result, you will not turn in a `main`() routine for the code that you hand in (but you might use one for your own testing).

I have provided the prototypes for these functions in the file `mem.h` (which is available from the course website). You should include this header file in your code to ensure that you are adhering to the specification exactly. **You should not change `mem.h` in any way!**

Here are some more details about your implementation:

- `initMem` should be called exactly once by a process using your routines.

- To call `mmap`, use code like this:

```
// open the /dev/zero device
int fd = open("/dev/zero", O_RDWR);

// requestSize (in bytes) needs to be evenly divisible by the page size
void *ptr =
   mmap(NULL, requestSize, PROT_READ | PROT_WRITE, MAP_PRIVATE, fd, 0);
if (ptr == MAP_FAILED) { perror("mmap"); exit(1); }

// close the device (don't worry, mapping should be unaffected)
close(fd);
```

- You may have **only one global variable** in your code. This is very important and is what makes writing the memory manager tricky. You will lose a lot of points if you have more than one variable. This means you need to use the allocated memory region returned by mmap for your own data structures as well; that is, your infrastructure for tracking the mapping from addresses to memory objects has to be placed in this region as well.

- You are **not** allowed to use `malloc`(), or any other related function, in any of your routines!

- You should not allocate global arrays!

- You are not required to coalesce memory. However, free'd regions should be reusable for future allocations that are equal or smaller in size.
- You will need to use a header with each allocated or free block. The maximum size of such a header is 32 bytes.

Shared Library

You must provide these routines in a shared library named `libmem.so`. Placing the routines in a shared library instead of a simple object file makes it easier for other programmers to link with your code. There are further advantages to shared (dynamic) libraries over static libraries. When you link with a static library, the code for the entire library is merged with your object code to create your executable; if you link to many static libraries, your executable will be enormous. However, when you link to a shared library, the library's code is not merged with your program's object code; instead, a small amount of stub code is inserted into your object code and the stub code finds and invokes the library code when you execute the program. Therefore, shared libraries have two advantages: they lead to smaller executables and they enable users to use the most recent version of the library at run-time. To create a shared library named `libmem.so`, use the following commands (assuming your library code is in a single file `mem.c`):

```
gcc -c -fpic mem.c -Wall -Werror
gcc -shared -o libmem.so mem.o
```

To link with this library, you specify the base name of the library with `-lmem` and the path to the library "`-L.`". It's important to put `-lmem` last. For example, if your program is called `memtest` you would say:

```
gcc -L. -o memtest memtest.c -Wall -Werror -lmem
```

Of course, these commands should be placed in a `Makefile`. The Makefile should have one rule to build libmem.so and a separate rule to build the test program.

Before you run `memtest`, you will need to set the environment variable, `LD_LIBRARY_PATH`, so that the system can find your library at run-time. Assuming you always run `memtest` from the directory where memtest is, you can use the command:

```
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

To avoid needing to type this export command every time that you login or start a new bash shell, you can put the line inside a file called .bashrc in your home directory. This file is automatically executed every time that a new Terminal window running bash is opened. (Since the filename begins with . , it does not show up when you say "ls", but if you say "ls -a", you will see that it is there.)

Sample test code

Here is some sample test code and possible output.

```
#include "mem.h"
#include <stdio.h>
#include <string.h>

int main (int argc, char **argv) {
  // Get a big chunk of memory from the OS for the memory
```

```
// allocator to manage.
void *memory = initMem (4096);
printf ("initMem returned %p\n\n", memory);

// We should see one big chunk.
dumpMem();

// Allocate 4 bytes of memory and store "abc" in that address
void *ptr = allocMem (4);
printf ("Allocated 4 bytes at %p\n", ptr);
strcpy(ptr, "abc");

// Still one chunk of free memory but it is smaller
dumpMem();

// Free the block just allocated.
printf ("Freeing the allocated chunk.\n");
if (freeMem(ptr) == -1) {
  printf ("ERROR:  freeMem failed!\n");
}


// Should have 2 chunks of free memory:  one at the block from
// the previous list and one at the start of the memory.
dumpMem();

// Try to free the pointer again.  This should fail.
freeMem(ptr);
if (freeMem(ptr) != -1) {
  printf ("ERROR:  freeMem of same pointer should have failed!\n");
}

// Allocate 2 chunks of memory
printf ("Allocating 2 chunks of memory.\n");
char *ptr2 = allocMem(4);
strcpy (ptr2, "mhc");
char *ptr3 = allocMem(4);
strcpy (ptr3, "bos");

// Should see 1 chunk
dumpMem();

// Freeing the first chunk and asking for memory that should come from
// the second free chunk.
printf ("Freeing first chunk and allocating a 3rd bigger chunk.\n");
if (freeMem(ptr2) == -1) {
  printf ("ERROR:  freeMem failed!\n");
}
char *ptr4 = allocMem(11);
strcpy (ptr4, "0123456789");

// Should see 2 chunks
dumpMem();
```

```
    // Allocate a chunk that should fit in the first free block
    printf ("Reallocating from first chunk.\n");
    char *ptr5 = allocMem(4);
    strcpy (ptr5, "csc");

    // Should see 1 chunk
    dumpMem();

    // Verify that memory that was set and not freed has not changed.
    if (strcmp (ptr3, "bos")) {
      printf ("ERROR: ptr3 changed to %s\n", ptr3);
    }

    if (strcmp (ptr4, "0123456789")) {
      printf ("ERROR: ptr4 changed to %s\n", ptr4);
    }

    if (strcmp (ptr5, "csc")) {
      printf ("ERROR: ptr5 changed to %s\n", ptr5);
    }

    // Allocate 4000 bytes
    printf ("Allocate a big blocks\n");
    void *ptr6 = allocMem (4000);
    printf ("Allocated 4000 bytes at %p\n", ptr2);

    // Still one chunk of free memory but much smaller
    dumpMem();

    // This allocation should fail
    void *ptr7 = allocMem (1000);
    printf ("Tried to allocate 1000 bytes.  allocMem returned %p\n", ptr7);
    dumpMem();

    printf ("Freeing a random address; it should fail\n");
    if (freeMem((void *) ptr3 + 4) != -1) {
      printf ("ERROR:  freeMem should have failed!\n");
    }

    dumpMem();
  }
```

And here is the output. Note that your output does not need to exactly match this. The addresses might be different and how you describe the free blocks may vary. The important thing is that we see the right numbers of free blocks of the right size, and that allocMem and freeMem fail at appropriate times.

```
initMem returned 0x7f3f5c81e000


Free memory:
address = 0x7f3f5c81e000
size = 4080

Allocated 4 bytes at 0x7f3f5c81e010

Free memory:
address = 0x7f3f5c81e014
size = 4060

Freeing the allocated chunk.

Free memory:
address = 0x7f3f5c81e000
size = 4

address = 0x7f3f5c81e014
size = 4060

Allocating 2 chunks of memory.

Free memory:
address = 0x7f3f5c81e028
size = 4040

Freeing first chunk and allocating a 3rd bigger chunk.

Free memory:
address = 0x7f3f5c81e000
size = 4

address = 0x7f3f5c81e043
size = 4013

Reallocating from first chunk.

Free memory:
address = 0x7f3f5c81e043
size = 4013

Allocate a big blocks
Allocated 4000 bytes at 0x7f3f5c81e010

Free memory:
Tried to allocate 1000 bytes.  allocMem returned (nil)

Free memory:
Freeing a random address; it should fail

Free memory:
```

But does it really work???

The testing suggested above is like JUnit testing, testing each individual function for correct behavior. You might be left wondering whether it really works like a memory manager should. If so, try this:

- Modify your textsort program by adding a call to initMem at the start. Then replace all the malloc calls with allocMem and the free calls with freeMem. (Be careful to only replace free calls with freeMem calls for those memory allocations that use allocMem. For example, if you call strdup, it will continue to use C's malloc function, so you should continue to use C's free function to free that chunk of memory.) If your memory allocator works correctly, textsort should continue to work!

If that works, you could test all your programs together:

- Rewrite mohosh to use initMem, allocMem and freeMem
- Start mohosh.
- Run your modified textsort from within mohosh.

All these programs should work together!

Teams

TBA.

Honor Code Reminder

I would like to remind everyone of the honor code. Doing Google searches to find information is very tempting. However, many times it can lead you to code that solves a very similar problem to what I am asking you to do, which can lead to honor code violations. Here is a quick reminder about some activities you should avoid.

- Do not look at another student's solution
- Do not use solutions to same or similar problems found online or elsewhere.
- Do not search for homework solutions online.
- Do not look at GitHub repositories that belong to other users.
- Do not turn in any part of someone else's work as your own (with or without their knowledge).
- Do not share your code or written solutions with another student.
- Do not share your code or snippets of your own code online.
- Do not store your solutions in a public place, like a public github repository

You can find the full honor code statement for this course at http://www.mtholyoke.edu/~blerner/cs322/HonorCode.html.

Grading

Code that is turned in that does not compile by saying "make all" will be returned ungraded. I will grade them if there are compiler warnings, but not if it is unable to create an executable program.

In addition to mem.c, you should also turn in a makefile that has a rule to build the shared library, and a separate rule to build memtest.

Grading will be based on correctness, documentation, and coding style in these proportions:

80% Correctness (broken down as follows)

- 5% Makefile
- 15% Using exactly one global variable
- 10% initMem
- 25% allocMem
- 20% freeMem
- 5% Miscellaneous

10% Comments

10% Coding style

Turning in your solution

Place your C file(s) and makefile into a single tar.gz file, using your name and your partner's name rather than mine in the name of the file. Be sure to include both of your names. (First names are enough.)

tar -cvzf Barbara_Lerner_Assign4.tar.gz *.c makefile

Upload your tar.gz file to Moodle. If working with a partner, only one of you should submit this.