Computer Science Department — Mount Holyoke College

CS 322
Operating Systems
Programming Assignment 4
Writing a memory manager
Due: November 16, 11:30 PM

Goals

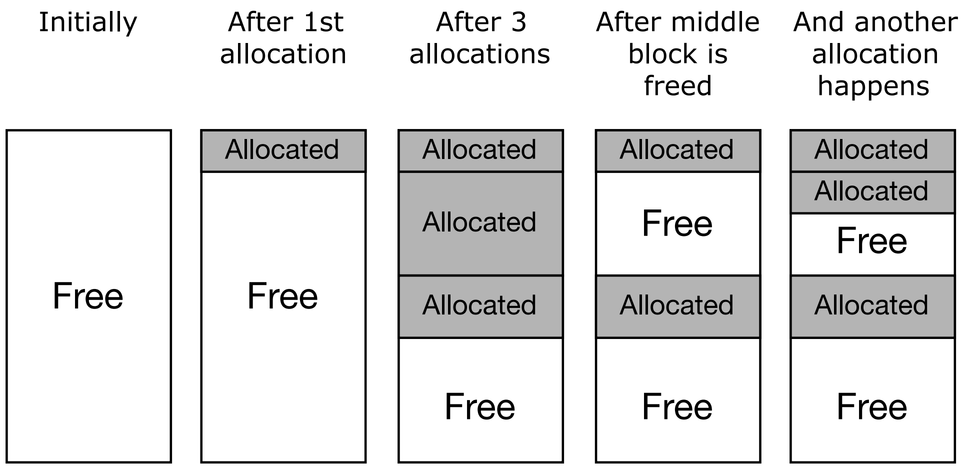
- To understand the nuances of building a memory allocator.
- To create a shared library.

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.

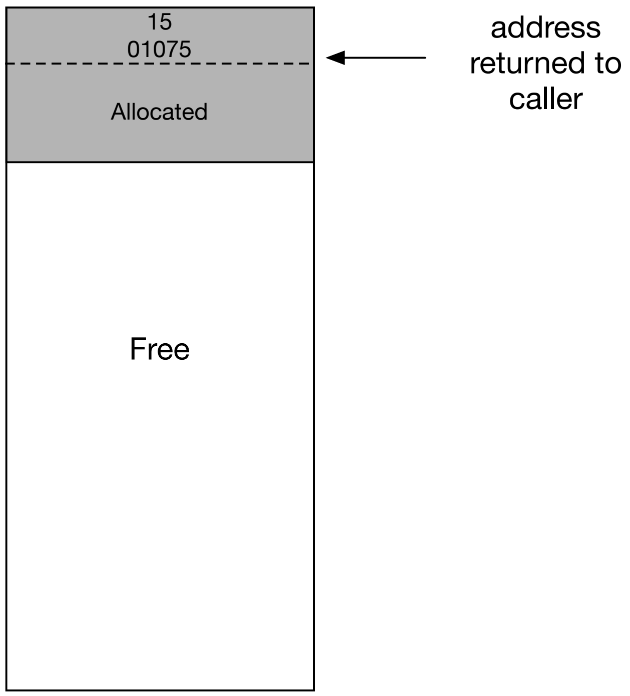
C's `malloc`() and `free`() are defined as follows:

- `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 whose address is returned to the caller.

The reason the memory allocator needs the header is so that free can work correctly. When free is called, it will pass the pointer that malloc returned. 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.

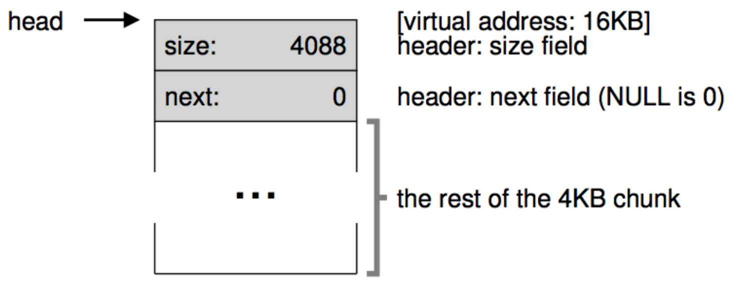


Figure 17.3: **A Heap With One Free Chunk**

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.

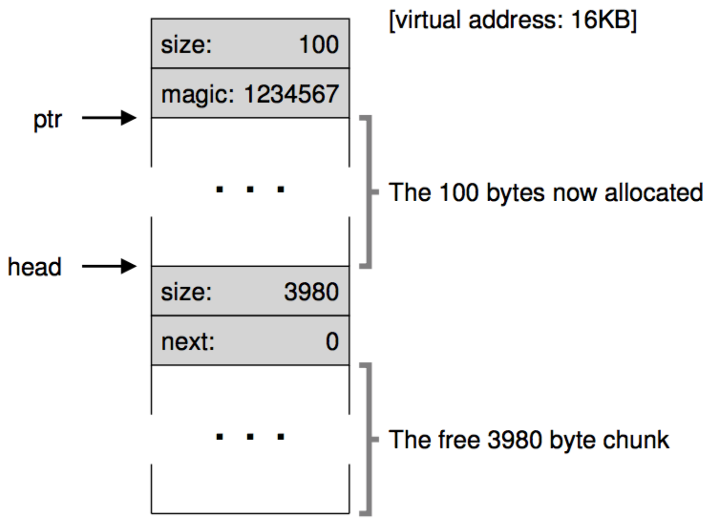


Figure 17.4: **A Heap: After One Allocation**

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.

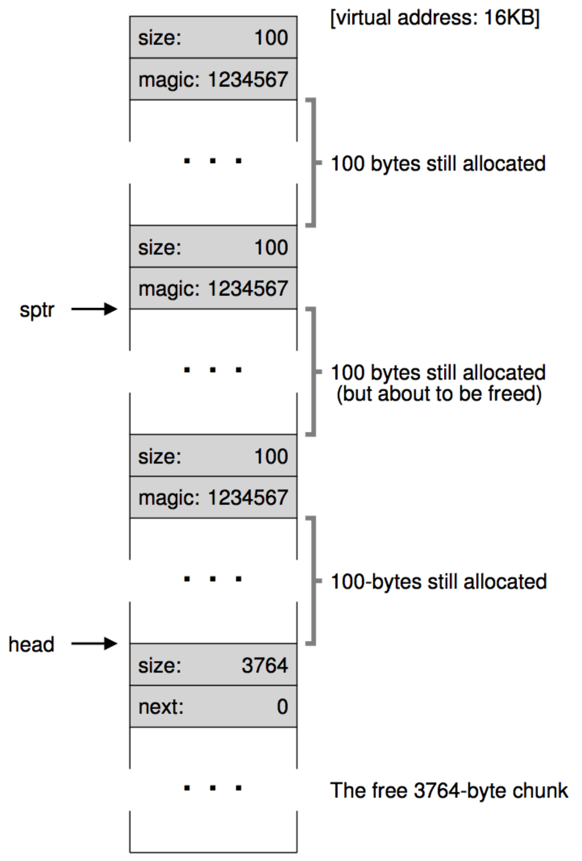


Figure 17.5: **Free Space With Three Chunks Allocated**

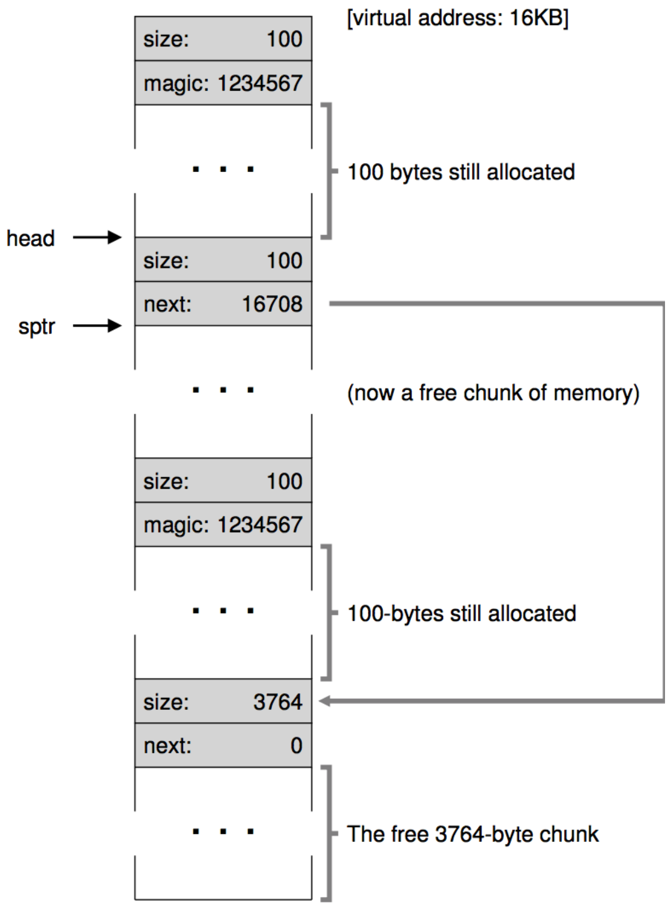


Figure 17.6: **Free Space With Two Chunks Allocated**

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 *Mem_Init (int sizeOfRegion)` - This function should call `mmap` to request `sizeOf-Region` 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`()). `Mem_Init` should return the address of the region returned by `mmap`. (This will be used by my tests.) If mmap fails, return NULL.
- `void *Mem_Alloc (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 Mem_Free (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 Mem_Dump()` - This function should print out the free list, which is useful for debugging purposes.

These functions will be part of a shared library. As a result, you will not be writing a `main`() routine for the code that you handi n (but you should implement 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:

- `Mem_Init` should only 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 variable should remember the value that `mmap` returned. 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.
- You will need to use a header with each allocated 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 `myprogram` you would say:

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

Of course, these commands should be placed in a `Makefile`. Before you run `myprogram`, 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 `myprogram` from this same directory, you can use the command:

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

Note that if you put this command in your makefile, you need to put parentheses around the variable name when you use it:

```
export LD_LIBRARY_PATH=.:$(LD_LIBRARY_PATH)
```

Sample test code

Here is some sample test code and output.

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

int main (int argc, char **argv) {
  // Get a big chunk of memory from the OS for the memory
  // allocator to manage.
  void *memory = Mem_Init (4096);
  printf ("Mem_Init returned %p\n\n", memory);
```

```
      // We should see one big chunk.
      Mem_Dump();

      // Allocate 1 byte of memory and store 16 in that address
      void *ptr = Mem_Alloc (1);
      printf ("Allocated 1 bytes at %p\n", ptr);
      *((int *) ptr) = 16;

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

      // Allocate 4000 bytes
      void *ptr2 = Mem_Alloc (4000);
      printf ("Allocated 4000 bytes at %p\n", ptr2);

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

      // This allocation should fail
      void *ptr3 = Mem_Alloc (1000);
      printf ("Tried to allocate 1000 bytes.  Mem_Alloc returned %p\n", ptr3);
      Mem_Dump();

      // Free the first block that was allocated
      printf ("Freeing first block\n");
      if (Mem_Free(ptr) == -1) {
        printf("Freeing ptr DID NOT WORK\n");
      }

      // Now there should be 2 free blocks
      Mem_Dump();

      // Try freeing with an address that malloc did not return.
      // This free should fail.
      printf ("Trying to free bad address\n");
      if (Mem_Free(ptr2 + 10) == -1) {
        printf("Trying to free a non-malloc'ed address.  Recognized bad ad-
dress!\n");
      }
      else {
        printf("Oh no!  Free did not complain!\n");
      }

      Mem_Dump();
   }
```

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 Mem_Alloc and Mem_Free fail at appropriate times.

```
Mem_Init returned 0x7f98c23a6000


Free memory:
block offset = 2
address = 0x7f98c23a6008
size = 4080

Allocated 1 bytes at 0x7f98c23a6010

Free memory:
block offset = 5
address = 0x7f98c23a6014
size = 4068

Allocated 4000 bytes at 0x7f98c23a601c

Free memory:
block offset = 1007
address = 0x7f98c23a6fbc
size = 60

Tried to allocate 1000 bytes.  Mem_Alloc returned (nil)

Free memory:
block offset = 1007
address = 0x7f98c23a6fbc
size = 60

Freeing first block

Free memory:
block offset = 2
address = 0x7f98c23a6008
size = 4

block offset = 1007
address = 0x7f98c23a6fbc
size = 60

Trying to free bad address
ptr not allocated
Trying to free a non-malloc'ed address.  Recognized bad address!

Free memory:
block offset = 2
address = 0x7f98c23a6008
size = 4

block offset = 1007
address = 0x7f98c23a6fbc
size = 60
```

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 Mem_Init at the start. Then replace all the malloc calls with Mem_Alloc and all the free calls with Mem_Free. Does textsort work?

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

- Rewrite whoosh to use Mem_Init, Mem_Alloc and Mem_Free
- Start whoosh.
- Run your modified textsort from within whoosh.

All these programs should work together!

Teams

You may work alone or with a partner on this assignment. You can choose your partner and the partner does not necessarily need to be in the same section of the course as you. The only rule is that you cannot pick someone you worked with on an earlier assignment.

If you are working with a partner and would like to have a git account set up for you, let me know!

Grading

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

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

80%	Correctness (broken down as follows)
	5% Compiling
	5% Makefile
	15% Using exactly one global variable
	10% Mem_Init
	25% Mem_Alloc
	15% Mem_Free
	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. (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.