A2 Implementing Malloc

Assignment 2: Implementing `malloc`

Due: Electronically, by 10:00 PM on February 26

Introduction

The library functions and system calls that implement dynamic memory allocation operate on a contiguous region of memory called the heap. Your task is to implement simple versions of `malloc` and `free` called `smalloc` and `sfree`. These two functions will operate nearly the same way as `malloc` and `free`: `smalloc` makes memory available ("allocates memory") and `sfree` frees that memory. One difference between `free` and your `sfree` is that `sfree` will return a -1 on error and a 0 on success. (The library `free` call doesn't return any value.)

Details

You are given some starter code for this assignment. You can find the starter code by checking out your repository. It will compile, but will not operate correctly until you implement the following four functions:

- `mem_init()`: (5%) Initializes the data structures that manage the dynamic memory allocation. Part of this function is given to you. A large region of memory is reserved using`mmap`. The comments above `mem_init` in the starter code explain how `mmap` is called. Using `mmap` in this way means that you are free to use `malloc` as normal to create your linked lists. (This is the only place in the code where `mmap` is used.)
- `void *smalloc(unsigned int size)`: (30%) Reserves `size` **bytes** of space from the memory region created by `mem_init`. If the memory is reserved (allocated) successfully, Returns a pointer to the reserved memory. If the memory cannot be reserved (i.e. there is no block that is large enough to hold `size` bytes), returns `NULL`.
- `int sfree(void *addr)`: (30%) Returns memory allocated by `smalloc` to the list of free blocks so that it might be reused later.
- `mem_clean()`: (10%) Uses `free` (that's the C function `free`, not your `sfree`!) to free **all** the dynamically allocated memory (`allocated_list` and `freelist`) used by the program before exiting. (The `valgrind` program must show `all heap memory freed`, otherwise you have a memory leak. See below.)

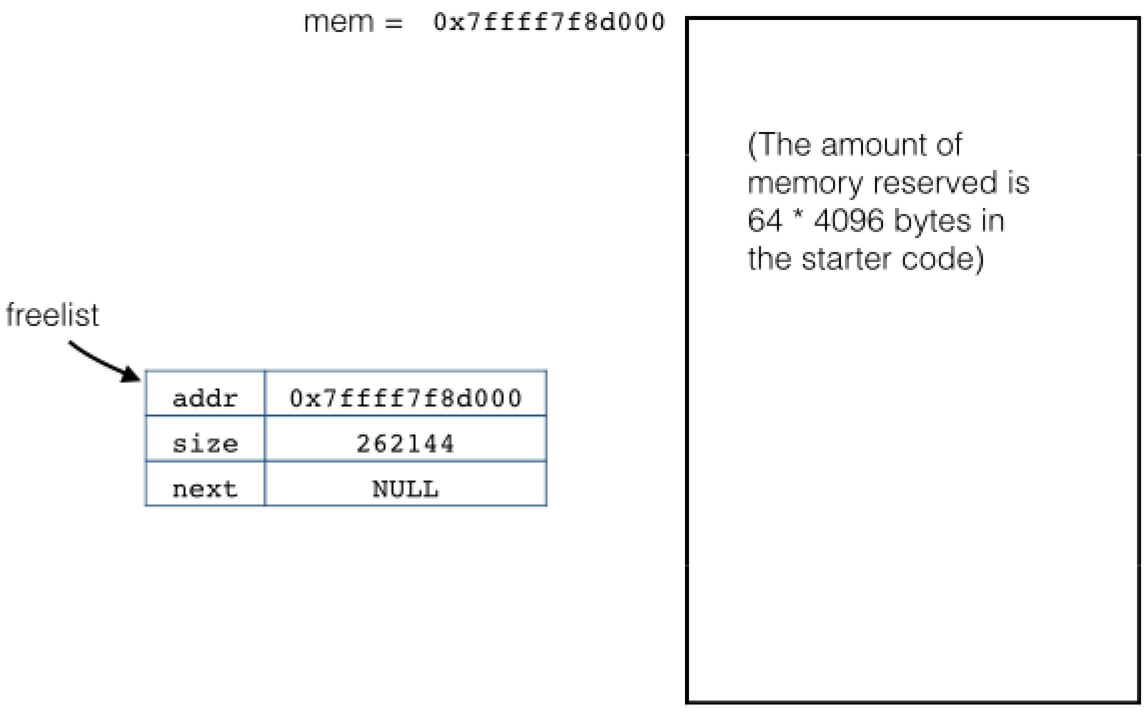
There are three global variables that define the data structures required:

- `mem` stores the starting address of the memory region that is reserved by `mem_init`.

- `freelist`: A linked list of `struct block`s that identify the portions of the memory region that are free (not in use). Blocks in this list are stored in increasing address order.
- `allocated_list`: A linked list of `struct block`s that identify portions of memory that have been reserved by calls to `smalloc`. When a block is allocated it is placed at the front of this list, so the list is unordered.

You can think of the address returned by `mem_init` as the start of a large array of bytes, and it is your job to partition it up when `smalloc` is called. Your program will keep track of two linked lists of blocks (using `struct block`): a list of allocated blocks (`allocated_list`) and a list of freed blocks (`freelist`). To complete `mem_init`, you need to create a block node with the starting address as given by `mmap`. Therefore, after `mem_init` completes, the `allocated_list` will be empty (but initialized!), and `freelist` will have one block in it where the address contained in that block is the address returned by `mmap`.

The following diagram shows the state of memory after `mem_init` has been called.

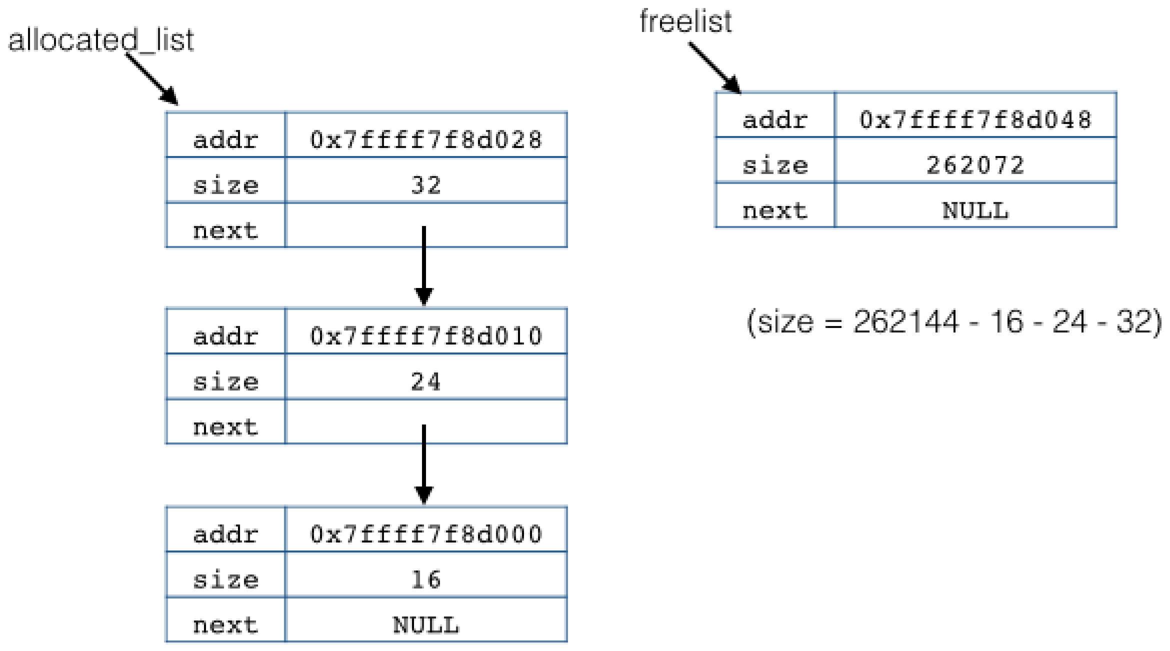

mem = 0x7ffff7f8d000
(The amount of memory reserved is 64 * 4096 bytes in the starter code)
freelist
addr 0x7ffff7f8d000
size 262144
next NULL

When `smalloc(size)` is called, it searches the `freelist` for a block that is at least `size` bytes in size. There are two possibilities for success: it might find a block of exactly the required size, or it might find a block that is larger than the required size. If it finds a block that is larger than `size` bytes, then it will split the block into two blocks. The first block, containing the address and size of the allocated block, is placed at the beginning of the `allocated_list`, and the block containing the address and size of the remaining memory stays in the `freelist`.

Example

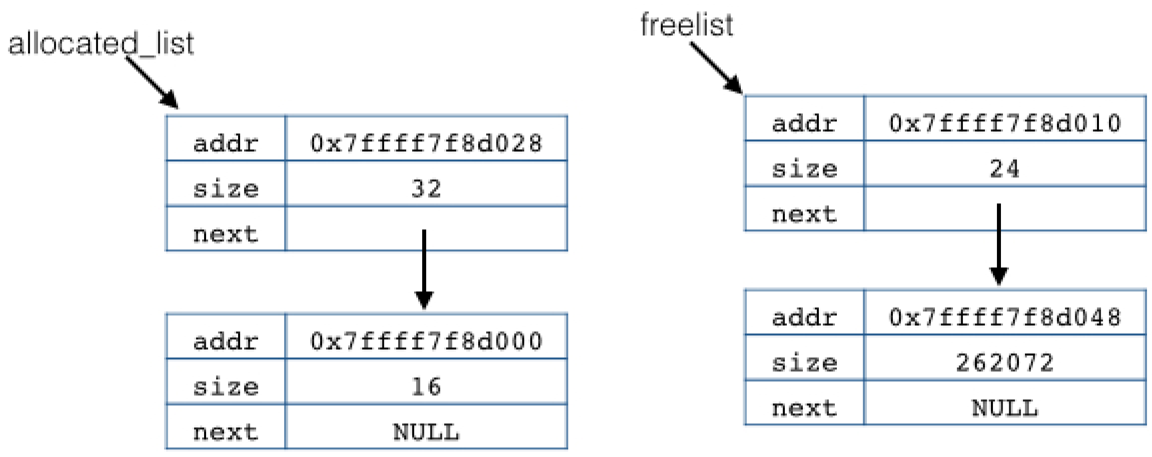
Suppose we have a test program that makes the following three calls to `smalloc`. The state of the lists are shown in the diagram below:

```
void *ptrs[3];
ptrs[0] = smalloc(16);
ptrs[1] = smalloc(24);
ptrs[2] = smalloc(32);
```



The next diagram shows the state of the two lists after the following call to `sfree`:

```
sfree(ptrs[1]);
```



Testing (15%)

(10%) In addition to writing the four functions to implement the dynamic memory system, you will also write one test program in `mytest.c` that tests your functions. You may use`simpletest.c` as a guide for how to write a test, but your program must test an interesting test case. Determining what makes an interesting case is a decision you need to make, and part of what we are marking. Your `mytest.c` program must include a comment at the top of the file explaining the case that is tested and why it is an interesting test case.

To make `simpletest` fully work, you will need to complete the `print_list` function.

(5%) Add a rule (or rules) to the `Makefile` to build an executable called `mytest`.

Add another rule to the `Makefile` with the target `tests` and the prerequisites `simpletest` and `mytest`that runs both test programs.

Valgrind

`valgrind` is a very useful tool for checking for memory errors. We will primarily use it to make sure that there are no memory leaks in our programs. Run it as:

```
valgrind simpletest
```

The output includes the output from simpletest. The output from valgrind will be prefixed with something that looks like "`==10320== `" (The number is the process id of the simpletest process and will change from run to run.) The information from valgrind that is most useful is near the bottom:

```
==10320==
==10320== HEAP SUMMARY:
==10320==     in use at exit: 0 bytes in 0 blocks
==10320==   total heap usage: 5 allocs, 5 frees, 120 bytes allocated
```

```
    ==10320==
    ==10320== All heap blocks were freed -- no leaks are possible
    ==10320==
    ==10320== For counts of detected and suppressed errors, rerun with: -v
    ==10320== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
```

The message you want to see is "All heap blocks were freed -- no leaks are possible." (The suppressed errors are not important.)

Code Quality (10%)

Code quality is always important. Use clear variable names, consistent formatting, informative comments, and good design. Compared to A1, there are more clear opportunities for modularization in your code for A2.

What to submit

You will commit to your repository in the `a2` directory **all** files required to build your program. This includes the `Makefile`, all source code (`.c`) and header files (`.h`).

Make sure that you've done `svn add` on every file. Check MarkUs, or checkout your repository into a new directory, to verify that everything is there. There's nothing we can do if you forget to submit a file.

You should be able to run `valgrind simpletest` and see the message *All heap blocks were freed -- no leaks are possible*.

Submission checklist:

You can make your life a lot simpler by ensuring that your submission is complete. When you first sit down to do some work on this assignment, check out the repository. Notice which files are already in the repository. Every time you create a new file, run `svn add` immediately to ensure that future commit operations will commit the new file as well.

Use version control as it was meant to be used. Commit your work frequently -- at least once every 2-3 hours that you are working on the assignment. This will prevent any last minute svn problems, and provide a record of your work.

Here is a list of things that you should do at least an hour before the assignment deadline:

- Create a temporary directory in your account (not one that is a subdirectory of your working directory for your repository).
- Check out your repository into this empty directory to be sure that the correct files have been committed.
- Run `make` to be sure that at least the `simpletest` program compiles without warnings or error.
- Run `simpletest` and your own tests to be sure that everything works as expected.

- Run `valgrind simpletest` to make sure that no memory is left allocated in the heap.