Simon Fraser University
School of Computing Science
CMPT 300: Assignment #3

The General Dynamic Storage Allocation Problem

Reminder: The rules of academic conduct apply as described in the course outline. All coding is to be done **alone or in pairs**. We will be using electronic software to compare all assignments handed in by the class to each other, as well as to assignments handed in for previous terms that this, or a similar, assignment may have been given in.

Be sure that this assignment compiles on the Linux computers in the CSIL lab using the gcc compiler. You can access the Linux server remotely as detailed on the course website.

What to Hand In: Include your source code files (including .h files), makefile, and the required documentation file(s) described below. Please ensure that all code/makefile files are in plain ASCII format, and the documentation is in a standard format (e.g., .txt/.pdf/.docx).

Introduction

For this assignment you are going to implement a solution to the general dynamic storage allocation problem. That is, you are going to provide the interface routines `my_malloc()` and `my_free()`. These should have the same parameters and return types as `malloc()` and `free()` as described in the UNIX man pages. As far as algorithms go, I'd like you to use your noodle and come up with an efficient and elegant algorithm. I am not really interested in existing algorithms that may be found in texts. Primarily I am interested in an algorithm which is as close to O(1) as possible for memory allocation and freeing. Don't worry too much (within reason) about fragmentation. If you don't have the energy, you are welcome to use one of the simple algorithms described in class (list-based first-fit, best-fit, or worst-fit). I'll leave it up to you, but bold new approaches and brave attempts at originality (even if failed) will be well rewarded and naive algorithms will limit your mark-earning potential.

Once you start to think about it, you will find that this is primarily a data structure problem. The key to efficiency will be the data structure you choose. Be creative in your use of data structures to make allocation and freeing very efficient.

Implementation Details

Whatever your algorithm, it should start out (on initialization) by requesting some large contiguous block of memory from UNIX (make the size configurable). Subsequent requests for memory should be satisfied by taking chunks out of this block. A request that cannot be satisfied should return an appropriate error value.

Your implementation should include the routine `mem_init()` which does whatever initialization you deem necessary. You will also have to decide when and how to join free blocks of memory together when they are adjacent to each other.

How will this assignment be judged? On several points:

- originality
- execution speed efficiency (i.e. - is it O(N)? O(1)?, etc)
- internal and external fragmentation (if they are deemed to be excessive)

Often these goals are traded off. For example linear-list-based first-fit is slow, but creates little fragmentation. No algorithm can be perfect in every respect, so don't twist your brain too hard looking for utopia.

When you hand in this assignment you are expected to also hand in a description of the algorithm and data structures used. Be sure to provide reasonable detail so that your ideas may be understood. I would expect the description to be between one and two pages - give or take. Feel free to use diagrams to detail your data structures. A significant bulk of the grade will be based on your data structure.

Submission

This will be handed in electronically. I'd like you to hand in the following items:

- All your source code: `my_malloc.c`, `my_malloc.h`, and a `mem_test.c` file (with the `main()` function) that tests your implementation.
  - We may replace your `mem_test.c` file with our own copy, so do make sure your files are named exactly as above.
- The separate documentation file(s) describing your algorithm/data structures.
- A makefile that compiles your .c files and links the objects to any necessary libraries. The makefile should produce the executable *mem_test*, and should have a clean rule.
- Do NOT hand in any executables or .o files (unless you choose to use the instructor-provided list.o implementation).

Have fun and good luck.