Assignment 2: Garbage Collection

- Due Mar 6 by 10p.m.
- Points 1

Introduction

Note (Feb 17): Please see below for a fix to the list starter code. The starter code on MarkUs has been updated to reflect this change. If you have already added the starter files to your repo and started work, you should **first ensure** that you have added, committed and pushed all work you have done on the assignment. Next, **download** list.c and do_list.c. (Don't click on "Add Starter Files to Repository" -- this will overwrite your work.)

Note 2 (Feb 19): There is another small fix to the do_fstree.c starter code and the list_trans1 file. The same instructions as above apply.

The first two programming languages we teach at the University of Toronto, Python and Java, use garbage collection to clean up dynamically allocated memory. This means that programmers don't need to worry about freeing the memory they have allocated. But this comes at a cost -- the garbage collection algorithms require both space to keep track of memory allocations and time to execute the algorithms that identify and free unused memory. In a low-level language like C, the burden of freeing unused memory falls to the programmer.

In this assignment, you will implement a simple mark-and-sweep garbage collector for C. With CSC209-level knowledge, we can't implement a general approach, so our garbage collector has to know about the dynamically allocated data structures one which we want to perform garbage collection. In other words, the garbage collection algorithm needs to know how to traverse specific data structures.

Mark and sweep is a commonly used garbage collection algorithm, and you will implement a relatively simple and inefficient version of it for this assignment. To be able to test your algorithm and to see how it operates, we need some sample data structures and a driver program that can perform a variety of adds and deletes on the data structure to test whether the garbage collector is working correctly.

Running the linked list program

The starter code provides two different data structures and the code to run them. We will focus first on the linked list code because it is the easiest place to get starter.

The files list.h and list.c implement functions on a simple linked list. The driver program, do_list.c, reads a sequence of operations from a transactions file and performs each operation (or transaction) on the linked list. This allows us to use the same program to run many different tests by writing different transaction files.

Let's take a look at a few transactions. Each line of a transaction file is either a comment line, or a transaction.

```
# add and delete several nodes
a 2777
r 2777
a 5386
a 2362
r 2362
p
g
```

The first character of a comment line is #. There are 4 different types of transactions on lists: a for add_node, r for remove_node, p for print_list, and g for mark_and_sweep (the function that does the garbage collection). The a and r transactions have a second argument on the line which indicates the value stored in the list node. The p and g transactions don't have any other arguments. Read the code in do_list.c to see how this works. We will only correctly formatted transaction files to run and test your code.

After you have your garbage collector working for the list data structure, you can add the extra code to support the fstree structure which is described below.

FIX to list.c and do_list.c:

As was pointed out on piazza in @601

Links to an external site. and @602

Links to an external site. there is a flaw in the tostring function in list.c. The function calls gc_malloc to allocate memory to store the string representation of the list. However, because it isn't part of the list data structure, it will always be marked and freed when the mark_and_sweep function is called. While this is not wrong because we use the string immediately in do_list to print it, and it is fine for it to be freed, it is a hidden side effect that could lead to problems if tostring was used in a different context. It also means that your memory chunk counts will not be the same as the number of list notes allocated.

Here are the following changes to make to address this:

1. Change `tostring` in list.c so that it calls `malloc` instead of `gc_malloc`.
2. Change the PRINT_LIST case in the main function in do_list.c so that it frees str before the break:

```
case PRINT_LIST :
    str = tostring(list_head);
    printf("List is %s\n", str);
    free(str);
    break;
```

End FIX

FIX TO do_fstree.c and list_trans1

There are two other small issues.

The first is in the do_fstree.c code, where the opened file is not closed. To fix this, you can add the fclose call after the transactions are processed:

```
do_transactions(fp, root);
//print_tree(root, 0);
fclose(fp);
```

The second issue is the transactions/list_trans1 file. The second comment in the file was not updated after the previous change to the starter files. Instead of "# there should be 4 nodes to free and 5 nodes reachable" it should read "# there should be 3 nodes to free and 5 nodes reachable". Since this is just a comment it should not affect any functionality.

End FIX

Starter Code

Take some time to read through the starter files working out what is already provided for you and how your code will fit into the existing framework. The files you should focus on first are those that are used to compile the `do_list` program. You can read the `Makefile` to identify these files.

All of your programming work will be in 3 files: `gc.c`, `gc_list.c`, and `gc_fstree.c`. IMPORTANT: none of the functions you are asked to implement should print to standard output. They may print to the log file, or print to `stderr`. (See below for a recommendation on how to add debugging print statements.)

Step 1: Keeping track of dynamically allocated memory

Step 1 a)

If you look at the functions in `list.c`, you will see that instead of calling `malloc`, they call `gc_malloc` when they need dynamically allocated memory. Your first small task is to add the call to `malloc` in `gc_malloc`. Once `gc_malloc` returns a pointer to dynamically allocated memory, you can compile and run the `do_list` program to see how it operates. No garbage collection is happening yet.

Step 1 b)

Now add the infrastructure to track dynamically allocated memory. You will use a linked list to store every address returned by `gc_malloc` together with a flag that indicates whether it is still used. Since the memory reference list will be used to track all calls to `gc_malloc`, you will never need more than one list. Since we need only one list, and since the memory reference list is really a system-level data structure, the global variable `memory_list_head` is provided for you. You will use the `struct mem_chunk` type from `gc.h` to implement your list.

After this step, run and test your code. "How do I test it?", you might ask. You probably want to create some small transaction files similar to the one above, and print out the memory reference list to confirm that you have the right nodes in the memory reference list.

Step 2: Mark and Sweep

To be able to tell whether a chunk of dynamically allocated memory is still in use, we must be able to traverse the program's data structures looking for all the memory that is currently accessible, and *marking* the memory chunks that are still reachable. Then we can *sweep* across the linked list of all dynamically allocated memory. Anything not reachable by traversing the program's data structures can be freed.

There are three steps to the mark and sweep algorithm:

1. Reset: set all of the used flags to "not used" in the memory reference list.

2. Mark: traverse the program's dynamically allocated data structures and set each address that we reach to "used" in the memory reference list. Note that we need a different mark function for every data structure. Cycles can be avoided by noting when a reference has already been marked as "in use" and halting the traversal of that portion of the data structure.

3. Sweep: traverse the memory reference list, freeing each address that is not marked as "used", and removing freed nodes from the list.

The file `gc.h` contains the function prototypes for the external interface of the garbage collector. Note that `mark_and_sweep` takes a function pointer as an argument. This is the mark function that traverses a specific data structure and knows which components have been dynamically allocated. However, this mark function does not need to know how the garbage collector is implemented, so we provide a `mark_one` function that will set the address `vptr` to "in use".

The mark function specific to the linked list should go into a separate file called `gc_list.c`. Similarly, the mark functions for other data structures should go into similarly named files.

Remember to free memory allocated for the nodes of the memory reference list. That memory is not managed by the mark and sweep algorithm. Valgrind should show that all heap memory has been freed if there are no nodes of the data structures that are reachable.

As an example, consider a linked list named `lst` with four nodes:

Suppose we then remove the third node with the transaction `r 3`. The memory diagram then changes to the following:

A call to the garbage collection (with the transaction `g`) might be visualized by the following steps:

1. The reset step. All the `in_use` flags are set to false.

2. The mark step. We traverse the linked list and mark each piece of memory that is reached as in use (using the `mark_one` function) in the memory reference list.

3. The sweep step. We traverse the memory reference list and remove all the not-in-use nodes, freeing any malloc-ed memory.

Sample programs

As described above, you are given two programs on which to implement your garbage collector. The only changes you need to make are to implement the `mark_list()` and `mark_fstree()` functions. The linked list code and its transactions are described above.

Linked list

The first program implements a series of operations on a simple linked list. The program `mktrans` creates a random series of operations on a linked list: adding integer nodes, deleting nodes, and printing the list. The program `do_list` takes a file produced by `mktrans` as a command line argument and executes the operations on a linked list. The files `list.h` and `list.c` contain the list operations.

The second data-structure: `fstree`

The second data-structure on which you will implement your garbage collector is based on a file-system tree with directories and subdirectories. In addition to a strict tree hierarchy, it allows links. This means that the `fstree` can contain cycles. Read the code in `fstree.h` and `fstree.c` carefully to understand how it works before you begin to implement the garbage collection.

The program `do_fstree.c` does a series of transactions on an `fstree` and follows a pattern similar to the transactions on lists.

Step 3: Report generation and testing

How do you tell whether your garbage collection algorithm is working correctly?

Your garbage collection code will write to a log file summary data each time mark_and_sweep is executed, including the number of chunks of memory that have currently been allocated, and the number of chunks that were freed. You will need to open the log file in append mode to be able to store multiple mark_and_sweep calls. Several print statements are given for you to use.

Rather than just hard-coding the name of the log file, you will put it in a define constant called `LOGFILE`. Make gives us the ability to set define constants (macros) by passing them in as an argument to the compiler. Then, in the program you can set a value of `LOGFILE` if one has not already been set. (See `gc.h` for the syntactic details.)

You may also want to print out other information while your program is running to help you debug and test it.

For example:

```
int debug = 1; //global variable

// inside some function
if(debug) {
  fprintf(stderr, "Here is something I want to print for debugging purposes.\n")
}
```

Generating some reports doesn't actually tell you whether your program is working correctly. You need to be able to verify that memory that is supposed to be freed actually is freed, and that memory that should not be reclaimed is not, and that all memory allocated by `malloc` that should be in the memory reference list is in the list.

Testing

In a file named testing.txt, provide a description of 6 test cases (3 for lists and 3 for fstrees) that demonstrate that your garbage collector is working correctly. At most one of the 6 tests may test for an error condition. For each test case, include the name of the transaction file used for the test case and a description of the case that is being tested that explains why it is an interesting case. The transaction file for each test case should be as small as possible to demonstrate the particular test case. You should not use the transaction files provided in the starter code.

Style and Error checking

Unlike A1, TAs will be reading your programs. You will be marked for programming style and error checking. For programming style, we are looking for readability. Features that make code more readable are good use of variable and function names, good use of white space, and appropriate level of comments to guide the reader, and good use of functions to break your program into logical pieces. In this assignment you may not find much need for helper functions.

For error checking and handling we are primarily checking to make sure that **all** system calls and I/O calls are checked, using `perror` where appropriate. A general rule of thumb is that your program should never crash. You may not be able to do anything on encountering an error other than calling exit, but that is fine. You only need to add error checking to code that you write. You do not need to add error checking to the starter code.

You may assume that the transaction files are correctly formatted. In other words, you do not have to add error checking to the code that reads the transaction files, and can assume that valid values are passed as arguments to the program.

What to submit

Commit all your code to the `a2` directory of your repository. All of the starter files should be in the repository. Don't forget to add your transaction files and testing.txt file. Test that you have submitted everything that you intended to by checking out your repository into another empty directory, and rerun your tests. Remember that you must not submit .o files or executable files.

Code that does not compile on teach.cs using your submitted Makefile will receive a grade of 0. So work on the program incrementally, and keep compiling it and testing it in pieces. When you have something working, commit it. For example, after you have written `gc_malloc`, run it and test it. Convince yourself that it works before moving to the next step.

References

Here are several of references that go into more detail about the differences between various algorithms for garbage collections. You may find it useful to read some of the descriptions of mark and sweep.

The Very Basics of Garbage Collection

Links to an external site.
Wikipedia on Garbage Collection

Links to an external site.