Linux Kernel Programming

Lab 01: Dusting Off Your C Skills

AUTHOR
Redha Gouicem & Julien Sopena

This first exercise is the occasion for you to check your C programming skills by implementing a version control system. Based on circular doubly-linked lists, these assignments will have you manipulate C structures and their padding, manage memory allocation, manipulate pointers and solve type issues.

Task 1: Structures

In this first task, you will implement the mechanisms related to the version numbering of commits. Each commit will be identified by three integers:

- **major**: a 2-byte integer incremented for every major change;
- **minor**: an 8-byte integer reset to zero after each major change, and incremented by one after each minor change. As in many projects, Linux included in a distant past, odd minor versions are *unstable* versions while even minor versions are *stable*;
- **flags**: a set of flags stored on 1 byte.

Thus, the file `version.h` defines the following structure:

```
struct version {
    unsigned short major;
    unsigned long minor;
    char flags;
};
```

Question 1

First of all, download the sources on Moodle, compile and execute the program `testVersion`. Is the stability test (*w.r.t. minor version numbers*) correct?

Question 2

The computation to find the minor version is syntactically valid and looks correct. With the help of a debugger, explain why the program produces this output.

Question 3

Using the name of the structure members, fix the `is_unstable()` function.

Question 4

What is the memory footprint of `struct version`? Is it optimal?

Question 5

Modify `struct version` to optimise the memory footprint.

Task 2: Computing Offsets

Let's define the structure associated to a commit. It will use the same version numbering as in Task 1 by embedding a `struct version`. In addition to the version, a commit contains:

- **id**: an 8-byte unsigned integer, unique for each commit;
- **comment**: a pointer to a string that contains a comment about the commit.

The structure looks as follows:

```
struct commit {
    unsigned long id;
    struct version version;
    char *comment;
    struct commit *next;
    struct commit *prev;
};
```

Question 1

First of all, implement a small program `testOffset.c` that allocates and initialises a `struct commit` and prints the addresses of every member.

Question 2

We now want to implement a function that returns the address of the `struct commit` that contains a given `struct version`. This new function must not rely on the order and number of members in `struct commit`, with the following prototype:

```
struct commit *commit_of(struct version *version);
```

As seen previously, the compiler can introduce padding in a structure in order to align members in memory. However, this padding is deterministic and identical across all instances of the structure.

First, print the offset between the start of the structure `struct commit` and the member `version`.

Question 3

From this offset, you can now implement the function `commit_of()` and compare the result to the addresses you obtained previously.

Task 3: Homemade circular doubly-linked list

Maintaining the history of version not only requires to save the commits, but also their order. While version numbers can be enough, using an appropriate data structure will improve the performance of our system.

Let’s modify the `struct commit` to store it into a circular doubly-linked list, and then implement a set of features related to the addition and deletion of commits.

To simplify the management of this list, we introduce a new structure `struct history` that contains a name, a list of commits and the number of commits.

```
struct history {
    unsigned long commit_count;
    char *name;
    struct commit *commit_list;
};
```

Question 1

Add the necessary rules in the `Makefile` to build the `testHistory` binary.

Question 2

In order to simplify the assignment, let’s consider that an empty list of commits is a phantom commit (that corresponds to no real commit) and points at itself. This phantom commit will always stay in the list and be its first element.

Implement the `new_commit()` and `new_history()` functions that allocate and initialise a commit and a history respectively. Then, implement the `last_commit()` function that returns the last commit from a history.

Question 3

We can now create an empty history and access its last element, but we still need to add new commits to this history. To do so, implement the `add_minor_commit()` and `add_major_commit()` functions, using the `static insert_commit()` function to effectively insert a commit created by these functions.

Question 4

Now, let’s use the circulare doubly-linked list to iterate over the commit history. Implement the `display_history()` printing function to get the following output when running the `testHistory` binary:

```
0: 0.0 (stable) 'DO NOT PRINT ME!!!'

History of 'Circle of Life':
```

```
History of 'Circle of Life':
1: 0.1 (unstable) 'Work 1'
2: 0.2 (stable) 'Work 2'
3: 0.3 (unstable) 'Work 3'
4: 0.4 (stable) 'Work 4'

History of 'Circle of Life':
1: 0.1 (unstable) 'Work 1'
2: 0.2 (stable) 'Work 2'
3: 0.3 (unstable) 'Work 3'
4: 0.4 (stable) 'Work 4'
5: 1.0 (stable) 'Release 1'
6: 1.1 (unstable) 'Work 1'
7: 1.2 (stable) 'Work 2'
8: 2.0 (stable) 'Release 2'
9: 2.1 (unstable) 'Work 1'

History of 'Circle of Life':
1: 0.1 (unstable) 'Work 1'
2: 0.2 (stable) 'Work 2'
3: 0.3 (unstable) 'Work 3'
4: 0.4 (stable) 'Work 4'
10: 0.5 (unstable) 'Security backport!!!'
5: 1.0 (stable) 'Release 1'
6: 1.1 (unstable) 'Work 1'
7: 1.2 (stable) 'Work 2'
8: 2.0 (stable) 'Release 2'
9: 2.1 (unstable) 'Work 1'
```

Question 5

Implement the `infos()` function that iterates over a history to search for a specific commit version. If found, it should display the commit, else it should print `Not here!!!`

```
Searching for commit 1.2 :   7: 1.2 (stable) 'Work 2'

Searching for commit 1.7 :   Not here!!!

Searching for commit 4.2 :   Not here!!!
```

Task 4: On the use of list_head

To optimise the search for a commit, we could start by iterating over major versions only until we reach the correct one, and then only iterate on minor versions starting there (*i.e., a skip list*).

Question 1

To implement this optimisation, we need a second doubly-linked list that only contains major commits. Can we imagine a simple code refactor for insertion and deletion from what we already have? Can this work be easily re-used for other structures?

> **Note**
>
> We did not implement a `del_commit(struct commit *victim)` function earlier. Now is a good time to do it and add some tests in `testHistory`.

Question 2

Instead of reinventing the wheel, we will use the linked list implementation from the kernel. You can find a modified version that works in user space in `sources/task4/list.h`.

Have a look, with a particular focus on `INIT_LIST_HEAD`, `list_add` and `list_del`, as well as `list_for_each` and `list_for_each_entry` macros.

Question 3

Replace the `next` and `prev` pointers in you `struct commit` with a `struct list_head`.

Appreciate the simplicity of this new code.

Task 5: The skip list

We can now easily add a second list that only contains major commits. You will need to add two members to your `struct commit`:

- *major_list*: a `struct list_head` that will link major commits;
- *major_parent*: a pointer that allows minor commits to find their major commit quickly.

Question 1

Modify your `struct commit` with this new members, update your insertion functions and test them. You can ignore your deletion function for now.

Question 2

Implement a new version of the `infos()` function that makes use of the skip list to find commits faster.

Task 6: Memory audit and freeing lists

Memory leaks are frequent in C, and removing a commit from the list is not enough to free resources. If memory leaks are problematic in a “regular” program, they become critical in the kernel.

Question 1

Afte recompiling your code with the `-g` option of gcc (adding debugging information), audit it with valgrind.

```
valgrind --leak-check=full ./testHistory
```

Question 2

If needed, fix your memory leaks.

You can also implement a `void free_history(struct history *)` that frees a history completely from memory.

Task 7: Function pointers and interfaces

We now want to change the display output depending on the type of commit, *i.e., display major commits differently than minor ones*.

```
0: 0.0 (stable) 'DO NOT PRINT ME!!!'

History of 'Circle of Life':

History of 'Circle of Life':
1: 0.1 (unstable) 'Work 1'
2: 0.2 (stable) 'Work 2'
3: 0.3 (unstable) 'Work 3'
4: 0.4 (stable) 'Work 4'

History of 'Circle of Life':
1: 0.1 (unstable) 'Work 1'
2: 0.2 (stable) 'Work 2'
4: 0.4 (stable) 'Work 4'

History of 'Circle of Life':
1: 0.1 (unstable) 'Work 1'
2: 0.2 (stable) 'Work 2'
4: 0.4 (stable) 'Work 4'
5: ### version 1 : 'Release 1'
6: 1.1 (unstable) 'Work 1'
7: 1.2 (stable) 'Work 2'
8: ### version 2 : 'Release 2'
9: 2.1 (unstable) 'Work 1'
```

```
        History of 'Circle of Life':
        1: 0.1 (unstable) 'Work 1'
        2: 0.2 (stable) 'Work 2'
        4: 0.4 (stable) 'Work 4'
        10: 0.5 (unstable) 'Security backport!!!'
        5: ### version 1 : 'Release 1'
        6: 1.1 (unstable) 'Work 1'
        7: 1.2 (stable) 'Work 2'
        8: ### version 2 : 'Release 2'
        9: 2.1 (unstable) 'Work 1'
```

Question 1

A first approach would be to add a test in `display_commit()` and use the proper output depending on the commit. What design issues could this cause?

Question 2

Like in the kernel, we will instead associate to each commit a different display function. To do so, we will add a `display` member to the `struct commit` that will point to the proper display function. The other functions that wish to display a commit will then call its embedded display function.

Implement the `display_major_commit` function. Fix the rest of your code to use function pointers for display the commits like in the example previously shown.

Question 3

We now want to use the same technique to adapt the deletion of a commit depending on its type: a minor commit will be deleted alone, while deleting a major commit should also delete all its associated minor commits.

Example: deleting commit 1.2 only deletes 1.2, while deleting 1.0 deletes 1.0, 1.1 and 1.2.

Implement such a behaviour by adding the function pointer member `extract` to the `struct commit` and implementing the functions `extract_minor()` and `extract_major()`.

Question 4

While using function pointers in structures allows for more flexibility and improves maintainability, it can make the structure quite large if the number of functions grows.

To overcome this, the idea is to use interfaces similar to object-oriented languages, and introduce a `struct commit_ops` that defines a set of functions for a commit. Each instance of this structure defines a set of functions suited for a specific case. Here, we will instantiate two of these structures: one for major commits, and one for minor ones. Your commits will only have a pointer to an instance of these commit operations instead of pointers to every function needed.

Change your code to use the following interface:

```c
struct commit_ops {
    void (*display)(struct commit *);
    void (*extract)(struct commit *);
};
```

Task 8: Error handling

We now want to improve the commit comments by adding a title and an author to the text. We will therefore replace the string `comment` by this structure:

```c
struct comment {
    int title_size;
    char *title;
    int author_size;
    char *author;
    int text_size;
    char *text;
};
```

Question 1

Copy the files `comment.h`, `comment.c`, `testComment.c` from `sources/task8/`, update your `Makefile` and analyse the memory usage of the `testComment` test. What are the issues of this implementation?

Question 2

Edit the `new_comment()` function to make it resilient. Your changes should allow the test to run without any modification.