Advanced Operating Systems (labs)

Vittorio Zaccaria | Politecnico di Milano | '24/25

Kernel concurrency and atomic ops

`lab-3-th-locking` is aimed at showing concurrency issues in the kernel when creating multiple kthreads modifying a shared variable.

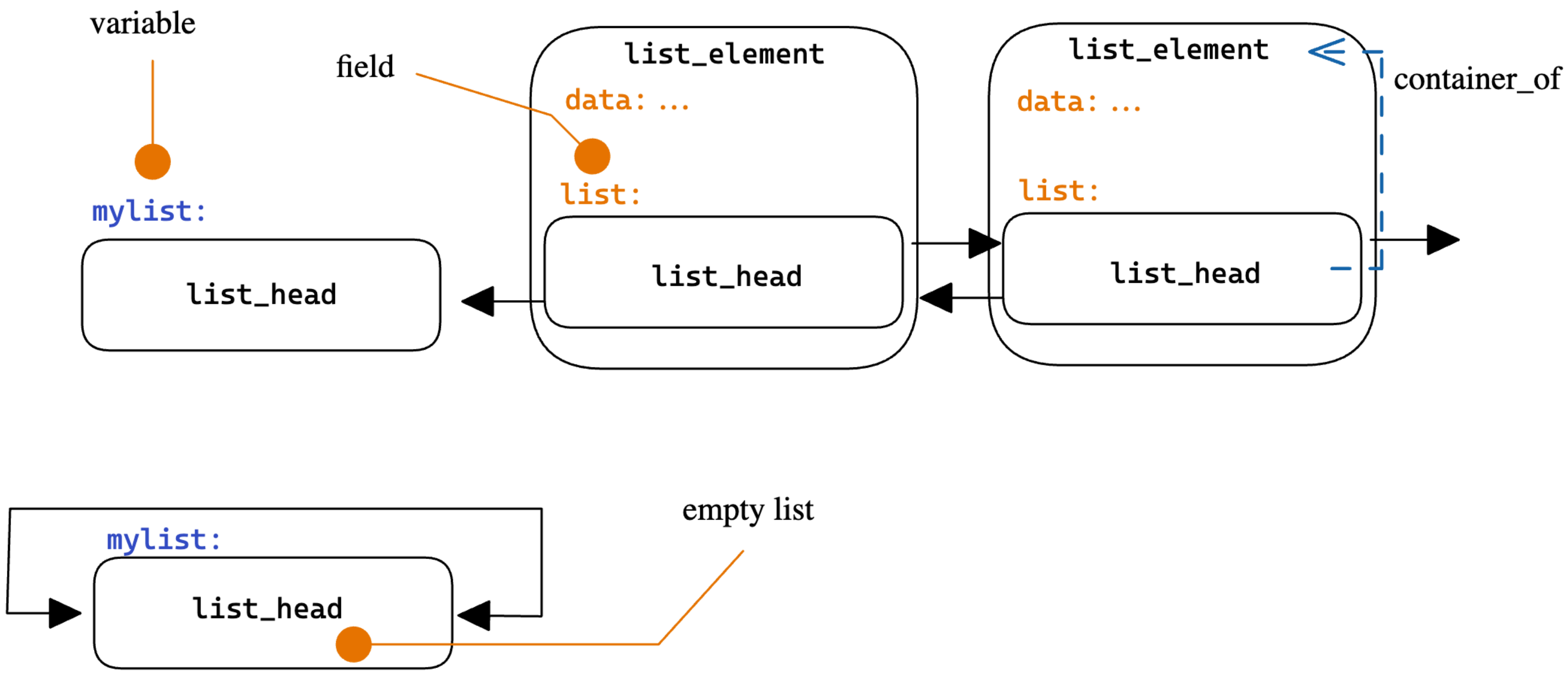
It is an educational example in two parts: the "Broken" section demonstrates concurrency issues, such as race conditions, when manipulating a shared variable without proper synchronization, and the "Fixed" section shows how to address these issues using atomic operations. This is a useful illustration of the importance of proper synchronization in kernel-level code to prevent data corruption and unpredictable behavior in concurrent scenarios.

Broken and fixed variants can be chosen with a module param when `insmod`-ing it (see source).

(preamble) Kernel lists

The `list_head` struct

Double linked lists in Linux are created by adding a `list_head` field in the structure of any element.
This structure contains two pointers: `next` and `prev`, which point to the next and previous elements in the list respectively.



> Structure of a list based on list_head

The `list_head` struct

Here's how you can define a doubly linked list in Linux:

```c
struct list_element {
    int data;
    struct list_head list;
};
```

In this case, `list_element` is a structure that has an integer variable `data` and a `list_head` variable `list`. The latter serves as an anchor point for the double linked list.

List manipulation

You can manipulate (insert, delete, search) elements in this linked list using macros provided by Linux such as `list_add()` , `list_del()` , etc. For instance, if you want to add a new element to this list:

```c
struct list_element *new_node;
new_node = (struct list_element *)kmalloc(sizeof(struct list_element), GFP_KERNEL);
new_node->data = 100;

INIT_LIST_HEAD(&new_node->list);
list_add(&new_node->list, &mylist);
```

In the above code snippet, we create a new node (`new_node`), initialize its `list` field, and then add it to an existing list (`myList`). This is how the doubly linked lists in Linux are created and manipulated.

The use of macros allows these operations on the lists to be somewhat polymorphic - meaning they can work with any type of data as long as it has the required `list_head` field.

Kernel RCU

`lab-3-th-rcu` we will demonstrate two variants of reader/writer threads manipulating a shared list. The initialisation code fills up a shared list with initial data, creates and starts reader and writer kthreads.

The `manipulate_list_thread_*()` kthread, manipulates the shared list by removing the first element if any, incrementing its value and adding it back. `read_list_thread_*` just prints the current list.

Variant 0 (broken)
Variant 0 of the reader-writer threads (`read_list_thread_norcu` and `manipulate_list_thread_norcu`) lacks proper synchronization, which may result in list corruption. Just check by running it!

Variant 1 (fixed with rcu)
Variant 1 of of the reader-writer threads (`read_list_thread_rcu` and `manipulate_list_thread_rcu`) employes Read-Copy-Update (RCU) synchronization to avoid such issues.

Broken and fixed variants can be chosen with a module param when `insmod` -ing it (see source).