National University of Computer and Emerging Sciences Chiniot-Faisalabad Campus

Lab 06

CL2006 – Operating System - Lab

Course Instructor	Juhinah Batool Asif
Lab Instructor	Juhinah Batool Asif
Semester	Fall 2024

FAST School of Computing

Department of AI & DS

Instructions

1. Make a PDF document with the convention "ROLLNO_ LAB#_ SECTION" and put all your source code and snapshots of its output in it.
2. Plagiarism is strictly prohibited, if you take a code snippet off the internet, mention its reference.
3. Do not discuss solutions with one another. Copying the solution from any source can lead to ZERO marks.

Kernel Data Structures

We covered various data structures that are common in operating systems. The Linux kernel provides several of these structures. Here, we explore using the circular, doubly linked list that is available to kernel developers. Much of what we discuss is available in the Linux source code— in this instance, the include file —and we recommend that you examine this file as you proceed through the following steps. Initially, you must define a struct containing the elements that are to be inserted in the linked list. The following C struct defines a color as a mixture of red, blue, and green:

```
struct color {

        int red;

        int blue;

        int green;

        struct list head list;

        };
```

Notice the member struct list head list. The list head structure is defined in the include file , and its intention is to embed the linked list within the nodes that comprise the list. This list head structure is quite simple—it merely holds two members, next and prev, that point to the next and previous entries in the list. By embedding the linked list within the structure, Linux makes it possible to manage the data structure with a series of macro functions.

Inserting Elements into the Linked List

We can declare a list head object, which we use as a reference to the head of the list by using the LIST HEAD() macro:

```
static LIST HEAD(color list);
```

This macro defines and initializes the variable color list, which is of type struct list head. We create and initialize instances of struct color as follows:

```
struct color *violet;

violet = kmalloc(sizeof(*violet), GFP KERNEL);

 violet->red = 138;
```

```
violet->blue = 43;

 violet->green = 226;

INIT LIST HEAD(&violet->list);
```

The kmalloc() function is the kernel equivalent of the user-level malloc() function for allocating memory, except that kernel memory is being allocated. The GFP KERNEL flag indicates routine kernel memory allocation. The macro INIT LIST HEAD() initializes the list member in struct color. We can then add this instance to the end of the linked list using the list add tail() macro:

```
list add tail(&violet->list, &color list);
```

Traversing the list involves using the list for each entry() macro, which accepts three parameters:

- A pointer to the structure being iterated over
- A pointer to the head of the list being iterated over
- The name of the variable containing the list head structure

The following code illustrates this macro:

```
struct color *ptr;

list for each entry(ptr, &color list, list) {

/* on each iteration ptr points */ /* to the next struct color */

}
```

III. Removing Elements from the Linked List

Removing elements from the list involves using the list del() macro, which is passed a pointer to struct list head:

```
list del(struct list head *element);
```

This removes element from the list while maintaining the structure of the remainder of the list.

Perhaps the simplest approach for removing all elements from a linked list is to remove each element as you traverse the list. The macro list for each entry safe() behaves much like list for each entry() except that it is passed an additional argument that maintains the value of the next pointer of the item being deleted. (This is necessary for preserving the structure of the list.) The following code example illustrates this macro:

```
struct color *ptr, *next;

list for each entry safe(ptr,next,&color list,list) {

/* on each iteration ptr points */

/* to the next struct color */

list del(&ptr->list); kfree(ptr);

}
```

Notice that after deleting each element, we return memory that was previously allocated with kmalloc() back to the kernel with the call to kfree().

Part I—Task

In the module entry point, create a linked list containing four struct color elements. Traverse the linked list and output its contents to the kernel log buffer. Invoke the dmesg command to ensure that the list is properly constructed once the kernel module has been loaded. In the module exit point, delete the elements from the linked list and return the free memory back to the kernel. Again, invoke the dmesg command to check that the list has been removed once the kernel module has been unloaded.

Part II—Parameter Passing

This portion of the project will involve passing a parameter to a kernel module. The module will use this parameter as an initial value and generate the Collatz sequence as described in Exercise 3.21.

Passing a Parameter to a Kernel Module

Parameters may be passed to kernel modules when they are loaded. For example, if the name of the kernel module is collatz, we can pass the initial value of 15 to the kernel parameter start as follows:

```
sudo insmod collatz.ko start=15
```

Within the kernel module, we declare start as a parameter using the following code:

```
#include<linux/moduleparam.h>

 Static int start = 25;

module param(start, int, 0);
```

The module param() macro is used to establish variables as parameters to kernel modules. module param() is provided three arguments:

(1) the name of the parameter,

(2) its type, and

(3) file permissions.

Since we are not using a file system for accessing the parameter, we are not concerned with permissions and use a default value of 0. Note that the name of the parameter used with the insmod command must match the name of the associated kernel parameter. Finally, if we do not provide a value to the module parameter during loading with insmod, the default value (which in this case is 25) is used.

Part II—Task

Design a kernel module named collatz that is passed an initial value as a module parameter. Your module will then generate and store the sequence in a kernel linked list when the module is loaded. Once the sequence has been stored, your module will traverse the list and output its contents to the kernel log buffer. Use the dmesg command to ensure that the sequence is properly generated once the module has been loaded. In the module exit point, delete the contents of the list and return the free memory back to the kernel. Again, use dmesg to check that the list has been removed once the kernel module has been unloaded.