Assignment 1. RB Tree and Dynamic Probe in Linux Kernel (200 points)

Related Subjects

1. Linux kernel RB tree
2. Linux module and device driver
3. Kprobe
4. x86's TSC (Time stamp counter) to measure elapse time
5. Multi-threaded programs.

Project Assignment

Part 1: Accessing a kernel RB tree via device file interface

Linux kernel consists of several generic data structures. One of them is RB tree (red-black tree) which is a form of semi-balanced binary tree. To form a binary tree, each node in the tree contains up to two children. A node in a RB tree should consist of a value that is greater than that of all children in the "left" child branch, and less than that of all children in the "right" branch. Thus, it is possible to organize a red-black tree by performing a depth-first, left-to-right traversal. The implementation is provided in include/linux/rbtree.h and lib/rbtree.c.

In this assignment, you are requested to develop a Linux kernel module which initiates an empty RB tree in Linux kernel and allows the tree being accessed as a device file. We will name the tree as "rbt530". The objects of type *rb_object_t* can be added to or removed from the RB tree according to their "*key*" value

```
typedef struct rb_object {
        int key;
        int data ;
} rb_object_t;
```

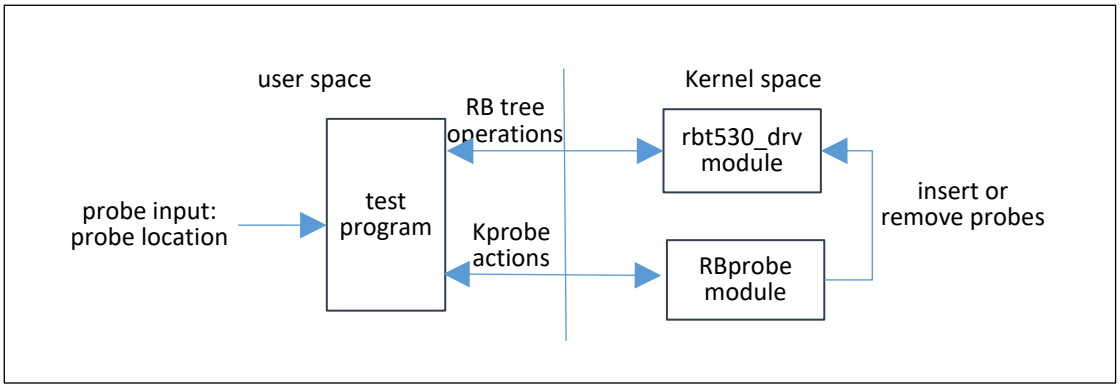
The RB tree is implemented in kernel space as a character device "rbt530_dev"and managed by a device driver "rbt530_drv". When the device driver is installed, the tree "rbt530" is created and a device "rbt530_dev" is added to Linux device file systems. The device driver should be implemented as a Linux kernel module and enable the following file operations:

- *open:* to open a device (the device is "rbt530_dev").
- *write:* if the input object of *rb_object_t* has a non-zero data field, a node is created and added to the *rbt530*. If an object with the same key already exists in the tree, it should be replaced with the new one. If the data field is 0, any existing object with the input *key* is deleted from the table.
- *read:* to retrieve the next object in either ascending or descending order (to be set by an *ioctl* call) from the RB tree. If the RB tree is empty or the next object is *null*, -1 is returned and errno is set to EINVAL.
- *loctl:* The command "read_order" to set up ascending or descending order that objects are to be read. If the argument is 0, read calls retrieve the objects in ascending order starting from the object with the minimal key. It if is 1, read calls get the objects in descending order starting from the object with the maximal key. Otherwise, -1 is returned and errno is set to EINVAL.
- *release:* to close the descriptor of an opened device file.

To test your driver, two RB tree devices, "rbt530_dev1" and "rbt530_dev2", should be created by your driver module to manage two RB trees, "*rbt530-1*", "*rbt530_2*". A user program should be developed in which the main program creates 2 threads to populate (by calling write operation) each RB tree with a total of 50 objects and then invoke read, write, and ioctl operations randomly. The threads are set with different real-time priorities and consecutive file operations are invoked after a random delay. After a total of 100 read and write operations are done at each tree, the threads should terminate and the main program dumps out all objects of the two trees. Note that the "*rbt530-1*", "*rbt530_2*" and their associated objects should be deleted when the driver module is removed.

Part 2: Dynamic instrumentation in kernel modules

Linux has static and dynamic tracing facilities with which callback functions can be invoked when trace points (or probes) are hit. In this part of the assignment, you are required to develop a kernel module that create a character device, named as "RBprobe". RBprobe uses kprobe API to add and remove dynamic probes in any kernel programs. With the module's device file interface, a user program can place a kprobe on a specific line of kernel code, access kernel information and variables. Integrated with part 1 of the assignment, you need to demonstrate the scenario depicted in the following diagram:



While exercising the two RB trees, "*rbt530-1*" and "*rbt530_2*", your test program reads in kprobe request information from console and then invokes *RBprobe* device file interface to register/unregister a kprobe at a given location of *rbt530_drv* module. When the kprobe is hit, the handler should retrieve few trace data items in a buffer such that they can be read out via *RBprobe* module interface. In the scenario, the user input request consists of the location (offset) of a source line of code on the execution path of read and write functions of *rbt530_drv*. The buffer is with a fixed size and holds one set of trace data, i.e. any old trace data will be overwritten when new trace data is generated. The trace data items to be collected by kprobe handler include: the address of the kprobe, the pid of the running process that hits the probe, time stamp (x86 TSC), and all *rb_object* objects traversed in the RB tree while performing the corresponding functions. Other than open and close file operations, the read and write operations of *RBprobe* device can be defined as:

- *write:* to register or unregister a kprobe. The location (offset) of the kprobe is passed in the buffer **buf* along with an integer flag. A kprobe is registered if the flag is 1, or unregistered if 0.
- *read:* to retrieve the trace data items saved in the buffer. If the buffer is empty, -1 is returned and errno is set to EINVAL.

You can reuse the test program in part 1 for the scenario in part 2. For instance, besides the 4 threads that exercise the RB trees, an additional thread can be created to receive input from console, to set up kprobes in *rbt530_drv*, and to read out any collected data items. Using proper synchronizations, you can control how the 4 threads invoke the operations to *rbt530* device file which can result in a hit at the kprobe point.

Due Date

The due date is 11:59pm, Feb. 21.

What to Turn in for Grading

- Create a working directory, named "EOSI-LastName-FirstInitial-assgn01", that consists of two sub-directories, "part1" and "part2", to include your source files (.c and .h), makefile(s), and readme.
- Compress the directory into a zip archive file named EOSI-LastName-FirstInitial-assgn01.zip and submit the zip archive to Canvas by the due date. Note that any object code or temporary build files should not be included in the submission.
- Please make sure that you comment the source files properly and the readme file includes a description about how to make and use your software. A sample result from your test run can be included in readme file. Don't forget to add your name and ASU id in the readme file.
- There will be 20 points penalty per day if the submission is late. Note that submissions are time stamped by Canvas. If you have multiple submissions, only the newest one will be graded. If needed, you can send an email to the instructor and TA to drop a submission.
- The assignment must be done individually. No collaboration is allowed, except the open discussion in the forum on Canvas. The instructor reserves the right to ask any student to explain the work and adjust the grade accordingly.
- Here are few general rule for deductions:
    - No make file or compilation error -- 0 point for the part of the assignment.
    - Must have "–Wall" flag for compilation -- 5-point deduction for each warning.
    - 10-point deduction if no compilation or execution instruction in README file.
    - Source programs are not commented properly -- 10-20-point deduction.
- ASU Academic Integrity Policy (http://provost.asu.edu/academicintegrity), and FSE Honor Code (http://engineering.asu.edu/integrity) are strictly enforced and followed.