Assignment 1. Hash Table and Dynamic Probe in Linux Kernel (200 points)

Related Subjects

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

Project Assignment

Part 1: Accessing a kernel hash table via device file interface

Linux kernel consists of several generic data structures. One of them is hash table which is based on chaining objects upon a collision. So, a hash table contains a predefined number of buckets and every bucket is a linked list which will link all objects that are hashed to the same bucket. The implementation is provided in Linux/include/linux/hashtable.h.

In this assignment, you are requested to develop a Linux kernel module which initiates two hash tables in Linux kernel. Each hash table is with 128 buckets and can be accessed as a device file. We will name the table as "ht530_tbl_0" and "ht530_tbl_1" and the objects to be embedded in the table have a type of

```
typedef struct ht_object {
        int key;
        int data ;
} ht_object_t;
```

The hash tables are implemented in kernel space as devices "ht530-0" and "ht530-1"and managed by a device driver "ht530_drv". The hash tables are created and the devices are added to Linux device file systems when the device driver is installed. 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 "ht530-0" or "ht530-1").
- *write:* if the input object has a non-zero data field, a ht_object is created and added it to the hash table. If an old object with the same key already exist in the hash table, 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 an object based on an input key. If no such object exists in the table, -1 is returned and errno is set to EINVAL.
- *Ioctl:* a new command "dump" to dump all objects hashed to bucket *n.* If n is out of range, -1 is returned and errno is set to EINVAL.
- *release:* to close the descriptor of an opened device file.

In a write call, **buf* points to a valid object and *count* gives the number of bytes of the object. The same parameters appear in read calls. However, for read function, the key value in the object pointed by **buf* is used to search the hash table. If the object is found, it is returned in the same buffer. For the *dump ioctl* command, you will need to generate a new ioctl number for it. The argument of the dump ioctl is a pointer to a buffer which is defined as

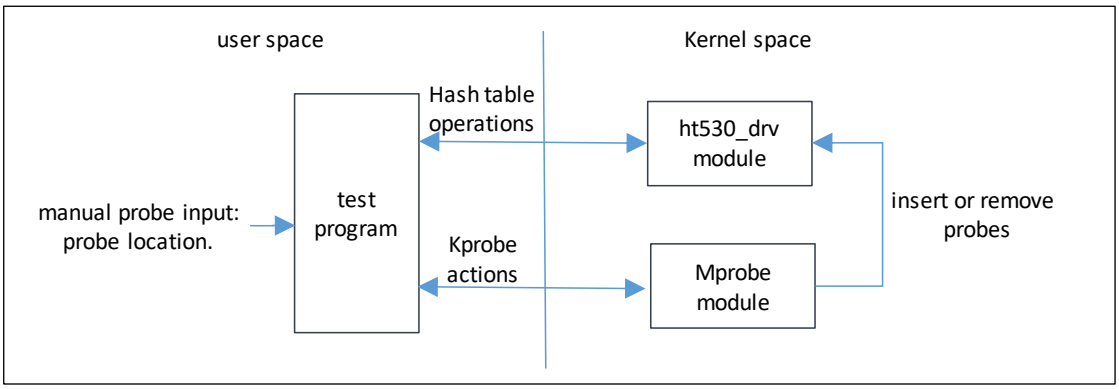
```
struct dump_arg {
```

```
        int n;              // the n-th bucket (in) or n objects retrieved (out)
        ht_object_t    object_array[8] ;  // to retrieve at most 8 objects from the n-th bucket
};
```

To test your driver, a user program should be developed in which the main program creates 5 threads to populate the tables with 150-200 objects and then invoke search, add, or delete operations to the tables. The threads are set with different real-time priorities and consecutive file operations are invoked after a random delay. After a total of 100 search, addition, or deletion operations are done, the threads should terminate and the main program dump out all objects in the tables. Note that the tables and its associated objects should be deleted when the driver module is removed. So, the tables are empty only at the first time your test program runs after the driver module is installed.

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, named as "Mprobe", that uses kprobe API to add and remove dynamic probes in any kernel functions. 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 hash tables in part 1, your test program reads in kprobe request information from console and then invokes *Mprobe* device file interface to register a kprobe at a given location of the write file operation function of ht530_drv module. When the kprobe is hit, the handler should retrieve few trace data items in a ring buffer such that they can be read out via Mprobe module. In the scenario, the user input request consists of the location (offset) of a source line of code in the write file operation function of ht530_drv module. 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 objects hashed to the bucket where the new object is to be added. Other than open and close file operations, the read and write operations of Mprobe device can be defined as:

- *write:* to register a new kprobe. The location (offset) of the new kprobe is passed in the buffer **buf*.
- *read:* to retrieve the trace data items collected in a probe hit and saved in the ring buffer. If the ring buffer is empty, -1 is returned and errno is set to EINVAL.

- *Ioctl:* a command "unregister" to unregister any existing kprobe that is registered previously by Mprobe module.

You can reuse the test program in part 1 for the scenario in part 2. For instance, besides the 5 threads that exercise the hash table, an additional thread can be created to receive input from console, to set up kprobes in *ht530-drv*, and to read out any collected data items. Using proper synchronizations, you can control how the 5 threads invoke the operations to *ht530* device file which can result in a hit at the kprobe point.

Due Date

The due date is 11:59pm, Sep. 23.

What to Turn in for Grading

- Create a working directory, named "EOSI-teamX-assgn01", for the assignment to include your source files (.c and .h), makefile(s), and readme. Compress the directory into a zip archive file named EOSI-teamX-assgn01.zip. Note that any object code or temporary build files should not be included in the submission. Submit the zip archive to Blackboard by the due date and time.
- 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 each team member's 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 Blackboard. 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.
- Your team must work on the assignment without any help from other teams and is responsible to the submission in Blackboard. No collaboration between teams is allowed, except the open discussion in the forum on Blackboard.
- Failure to follow these instructions may cause deduction of points.
- 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 there is no instruction on compilation or execution 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.