Linux Kernel Development

Akshat Malik - 906110838

Project 3: Latency Profiler

Big Picture:

Our latency profiler should run in kernel space and measure sleeping time of all running tasks in a system. The profiling results should include the call stack information, where a task sleeps, accumulated sleeping time in CPU clock cycles in a descending order of sleeping time.

Task 1:

Do not modify existing kernel data structures and measure latency.

Solution:

Used modules which implement kprobe handlers for probing certain functions. The functions which have to probed are:-

- *enqueue_entity()* – This is the function which the scheduler calls when a task wakes up.
- *dequeue_entity()* – This is the function which the scheduler calls when the tasks is put to sleep.

To calculate the latency of any process, we need to get the time elapsed between *dequeue_entity()* and enqueue_entity(). But as these two functions are declared static, we cannot probe them. So to get pass this, I used :-

- *activate_task()*
- *deactivate_task()*

These task are basically called in whenever *enqueue_entity()* and *dequeue_entity()* are called, so they can act as a replacement to calculate the latency. These are found in */kernel/sched/core.c*

Task 2:

Maintain sleeping time information for each pid and kernel call stack combination

Solution:

By accessing the *pt_regs* registers passed through the pre_handler function of kprobe, we can access the current task's PID and stack information.

Design strategy for data structures:

I have used the following data structures:

- Hash Table – It has O(1) look up and insertion time which makes searching and inserting elements very fast.
- RB Tree – Due to its pre-ordered nature it covers up for the hash table being unordered. Though it has O(log(n)) insertion time.

In order to use hash table for searching and RB Tree for printing saved data, I utilized a single structure which incorporates the hlist_node for the hash table, rb_node for the RB Tree and a pointer to another structure which contains the PID, stack strace and sleep time of a task.

Hash Key Generation

The key to the hash table is calculated based on the combination of a task's PID and stack trace. The values are passed through the *Jenkins hash function*, which gives the key. The key is then used to add the element to the hash table.

Walking through the code

As soon as the task activates, the kprobe is called. My code then stores all the required data for the current task in the combined data structure and then passes the pointer to it to both hash table and the RB Tree (as all are in a single data structure). The value of sleep time is set to 0 as the task has just been activated. When the task is deactivated, the task is searched through the hash table to see if it exists or not. If it exists, the CPU cycles are recorded through rdtsc() and if it doesn't, it is ignored as we do not have an entry for it. Later when the task is activated again, we search through the hash table to get the pointer to the data structure containing the values and update the sleep time based on the time recorded when the deactivate function was called. Hence the latency is calculated and stored.

Task 3:

Print out results.

Solution:

A virtual file called *lattop* is created in proc file system to print out all values contained in the RB Tree when the user calls cat */proc/lattop*.

The proc_show function is utilized to achieve this. In there, an iteration function runs to iterate over the whole RB Tree structure which prints out values in descending order.

Clearing memory:

I'm deleting the entire hash table in the exit module. While deleting all connections of the nodes, I'm freeing up the memory allocated to the data structure which contains the task related information and then I clear the memory allocated to the data structure which contains the hlist_node, rb_node and the pointer to my previously cleared data structure. This ensures that there is no memory leakage. As the the header nodes of RB Tree are also in this data structure, by iterating over the hash table by using *hash_for_each_safe()* and deleting each node, RB Tree is also being deleted.

Task 5:

User call Stack

Solution:

I utilized *save_stack_trace_user()* function to save the user stack along with the kernel stacktrace. The function is found in *include/linux/stacktrace.h*

Task 5:

Analysis of results while cloning the Linux kernel repository:

Latency profiler result:

```
Latency Profiler!

Begins...

Sleep : 2226844108361

Name : git

PID : 2048

     exit_to_usermode_loop+0x57/0x90

     prepare_exit_to_usermode+0x2f/0x40

     retint_user+0x8/0xd

     0xffffffffffffffff

Sleep : 1464701333651

Name : rcu_sched

PID : 8

     rcu_gp_kthread+0x611/0xd40

     kthread+0x104/0x140

     ret_from_fork+0x22/0x30

     0xffffffffffffffff

Sleep : 1154952211850

Name : sshd

PID : 2008

     exit_to_usermode_loop+0x57/0x90

     syscall_return_slowpath+0x53/0x60

     entry_SYSCALL_64_fastpath+0x92/0x94

     0xffffffffffffffff
```

It is seen that the following three task sleep the most:

1. git – Cloning is an I/O bound task, it sleeps whenever it needs to write or request data from the net, hence it has the largest sleeping time.
2. rcu_sched - It stands for 'read-copy-update', it's a lockless kernel structure duplication mechanism that allows for changes to be atomically committed. It is usually involved in situations where normal locking is far too slow.
3. Sshd – It is the daemon program for ssh. This is also I/O bound and sleeps whenever it is waiting for user input. It listens for connections from clients.