CS353: Linux Kernel Project 2(b) Report

Weichen Li
5120309662
eizo.lee@sjtu.edu.cn
Department of Computer
Science and Engineering
School of Electronic Information
and Electrical Engineering
Shanghai Jiao Tong University

CONTENTS

I. OBJECT

Add ctx, a new member to $task\_struct$ to record the schedule in times of the process;

- When a task is scheduled in to run on a CPU, increase ctx of the process;
- Export ctx under $/proc/XXX/ctx$;

II. COMPILE STEPS

A. Preprocess Working

At the very beginning, to save some trivial operations on authority verification, we need get the root authority first. (Need to input the key)

```
su
```

1) Modify kernel source code: We have three places to modify:

1. In source file $include/linux/sched.h$, find where $task_struct$ is, and add a member ctx to it. (Shown in Fig 1)

```
struct task_struct {
        unsigned int ctx;
        volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
        void *stack;
        atomic_t usage;
        unsigned int flags;     /* per process flags, defined below */
        unsigned int ptrace;
```

Fig. 1. add ctx

2. In source file $kernel/fork.c$, find the do_fork function, initialize ctx in it. (Shown in Fig 2)
3. In source file $kernel/sched/core.c$, find the place where processes are switched, plus ctx by 1. (Shown in Fig 3)
4. In source file $fs/proc/base.c$, create a new entry in $staticpid\_entrytgid\_base\_stuff[]$ array. For example:

```
ONE("ctx", S_IRUSR, proc_pid_my_file),
```

```
if (!IS_ERR(p)) {
        struct completion vfork;

        p->ctx = 0;

        struct pid *pid;

        trace_sched_process_fork(current, p);

        pid = get_task_pid(p, PIDTYPE_PID);
        nr = pid_vnr(pid);

        if (clone_flags & CLONE_PARENT_SETTID)
                put_user(nr, parent_tidptr);
```

Fig. 2. initialize ctx

```
        if (likely(prev != next)) {
                rq->nr_switches++;
                rq->curr = next;
                ++*switch_count;

                next->ctx++;
```

Fig. 3. add ctx by 1

5. In source file $fs/proc/base.c$, create a callback function that will be invoked when the new proc entry is accessed ($*task\_struct$ is passed to this function).

```
struct pid *pid, struct task_struct *task)
{
        seq_printf(m, "%d\n", task->ctx);
        return 0;
}
```

2) Compile the kernel: In the root directory of the linux kernel (Say, $linux-4.0/$), input commands below:

```
make
make modules_install
make install
```

The time needed to execute the first command may be very long, as well as the time for the next two commands, so please be patient when the kernel is being compiled.

After the compilation of our modified kernel, next we can obtain the output of our modification–ctx.

B. Base Solution

1) Create relevant files: To start our work, we need to create a new directory to store our work files.

```
mkdir /usr/src/Project_2B
cd /usr/src/Project_2B
```

Then we create a $.c$ file.

```
vi block.c
```

Then add the content below to $block.c$:

```
#include <stdio.h>

int main() {

while (1) getchar();

return 0;
}
```

Next, in command line, compile the $block.c$ file.

```
gcc block.c -o block -Wall
```

2) Compile and execute the block file: Open another terminal B, execute:

```
./block
```

Open a new terminal C, execute:

```
ps -e | grep block
```

And in ternimal C we can see some output shown in the ternimal like:

```
51?             00:00:00 kblockd/0
7711 pts/0      00:00:00 block
```

In ternimal C, execute:

```
cd /proc/7711
cat ctx
```

We can get the ctx value of process 7711.
In terminal B, press 'a', then press 'Enter'.
In terminal C, execute:

```
cat ctx
```

We will see that the value of ctx has been increased by 1.
To terminate the process, execute:

```
kill -9 7711
```

C. Module Solution

1) Create relevant files: To start our work, we need to create a new directory to store our work files.

```
mkdir /usr/src/Project_2B
cd /usr/src/Project_2B
```

Then we create a $.c$ file.

```
vi traverse.c
```

Every time the "vi" command is called, the terminal will jump into the command window of VIM. We can simply input ":wq" and press "Enter" button to get back to the terminal.
Next we need to create a Makefile file:

```
vi Makefile
```

Do not quit the VIM in a hurry, we need to add some content in the Makefile file, press the "i" button on the keyboard first, then input:

```
obj-m := traverse.o

KDIR := /lib/modules/$(shell uname -r)/build

PWD := $(shell pwd)

all:

make -C $(KDIR) M=$(PWD) modules

clean:

rm *.o *.ko *.mod.c Module.symvers modules.order -f
```

Then press the "ESC" button, input ":wq", press "Enter" button, the Makefile file is built. (This is the typical way to modify and save text files using VIM, "wq" means save and quit)
Next, input:

```
vi traverse.c
```

Add content below , save and quit.

```
#include <linux/module.h>
#include <linux/list.h>
#include <linux/init.h>
#include <linux/sched.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>

MODULE_LICENSE("Dual BSD/GPL");

static int hello_proc_show(struct seq_file *m, void *v) {

        struct task_struct *task, *p;
        struct list_head *pos;
        int count=0;

        seq_printf(m, "test module init\n");

        task=&init_task;
        list_for_each(pos, &task->tasks)
        {
                p=list_entry(pos, struct task_struct, tasks);
                count++;
                seq_printf(m, "%-15s\t[pid: %d]\t[ctx: %d]\n", p->comm, p->pid,p->ctx);
        }
        seq_printf(m, "Total %d tasks\n", count);

        return 0;
}
static int hello_proc_open(struct inode *inode, struct file *file) {
        return single_open(file, hello_proc_show, NULL);
}
static const struct file_operations hello_proc_fops = {
        .owner = THIS_MODULE,
        .open = hello_proc_open,
        .read = seq_read,
        .llseek = seq_lseek,
        .release = single_release,
};

static int test_init(void)
{
        proc_create("hello_proc",0,NULL,&hello_proc_fops);

return 0;
}

static void test_exit(void)
{
        remove_proc_entry("hello_proc", NULL);
}

module_init(test_init);
module_exit(test_exit);
```

2) Compile and insert the modules: So far we have completed all the preparation. Next, input:

```
make
```

Then input:

```
insmod traverse.ko
cat /proc/hello_proc
```

We can see that the $traverse$ module has been successfully inserted into the kernel. (Shown in Figure 4)

At this moment, we have successfully completed all the tasks for experiment 2(a), congratulations!

Tips: you can use “rmmod xxx.ko” command to remove module xxx from the kernel, or use “make clean” to delete all the files produced when compiling, then only the initial files left.

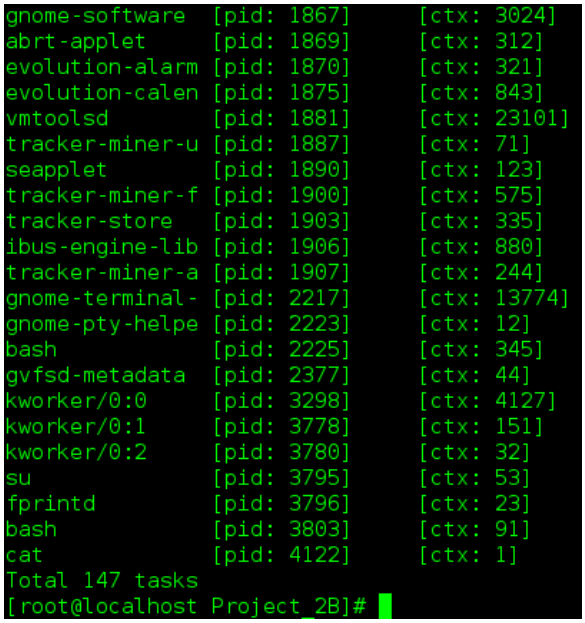

Fig. 4. *traverse* module