Project 2A: Linux Process Management

Yesheng Ma

April 6, 2017

Abstract

In this project, we are going to look into the Linux process scheduler, of which the most important one is the kernel `task_struct` data structure. We will add a counter to each process and make the counter available to users via the `proc` file system.

1 Some Preparations

1.1 Transfer to Arch Linux

Initially I use Ubuntu Linux to write kernel programs. However, since Ubuntu is not such a lightweight operating system, compiling the kernel source code is quite time-consuming and modifying the kernel is more dangerous. Thus, I decided to do the following experiments on Arch Linux, which is a simple and lightweight operating system.

The installation of Arch Linux is not that simple as Ubuntu. You have to partition disks, make file systems and do some configurations by yourself. I learned a lot from this and find Arch Wiki a valuable reference for Linux programmers.

1.2 Kernel Build in Arch Linux

Since building the Linux kernel can be time-consuming, when I read an article on kernel build in Arch Wiki, I find a quite useful command to generate a configuration file for kernel build, which will build those required parts in the kernel source code. The command is `make localmodconfig` and don't forget to run `make mrproper` beforehand. In this way, you can save much time on compiling the kernel.

2 Counting Schedule Times

In this section, we will mainly discuss how to add a member of `ctx` in the kernel data structure `task_struct` and make this information available to users via the `proc` file system.

2.1 Add ctx in task_struct

In Linux kernel, the headers regarding process scheduler is defined in include/linux/sched.h. What we need to do is add an item in the task_struct data structure.

```
struct task_struct {
    ...
    int ctx;              /* counter for scheduler */
    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;
    ...
}
```

Listing 1: add ctx to task_struct

Here we add the ctx member near state variable, which is closely related to scheduling. In this way, we may achieve better performance by spatial locality.

2.2 Initialize ctx of a process

For our counter to work properly, we need to set the ctx to 0 when a process is created. As we all know, a process is created by the system call fork in a Unix operating system, thus we can trace into the implementation of fork.

Actually, we should add the initialization code before the process is woken up, i.e. we should set ctx to 0 immediately after copy_process. The related code is shown as follows:

```
long _do_fork(...)
{
    ...
    p = copy_process(clone_flags, stack_start, stack_size,
            child_tidptr, NULL, trace, tls, NUMA_NO_NODE);
    p->ctx = 0;    /* initialize ctx to 0 */
    ...
}
```

Listing 2: Initialize ctx to 0

2.3 Increment ctx on scheduling

Since our task is to record the schedule number, we should increment the counter once the process is scheduled to the CPU. The core implementation of Linux scheduler is in kernel/sched/core.c.

As we can see in the documentation of core.c, the main scheduler routine is __scheduler. Since the way the scheduler works is it adds a task to be scheduled into a run queue if the process qualifies and thus we should increment the counter immediately after that. The code is shown as follows:

```c
static void __sched notrace __schedule(bool preempt)
{
    ...
    if (likely(prev != next)) {
        rq->nr_switches++;
        next->ctx++;      /* increment the counter */
        rq->curr = next;
        ++*switch_count;
    ...
}
```

Listing 3: Increment the counter on scheduling

2.4 Add ctx to proc file system

The base process directory entries under the proc file system is implemented in the kernel source file `fs/proc/base.c`. To add a new file in proc file system, we should declare it using `ONE` macro in `tgid_base_stuff`, where the `ONE` macro needs three parameters: file name, file permission bit and a file operation to show it. The source code is shown as follows:

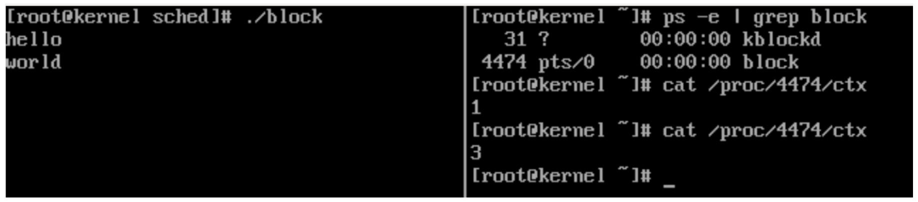
```c
/* the proc_show file operation */
int proc_pid_ctx(struct seq_file *m, struct pid_namespace *ns,
        struct pid *pid, struct task_struct *task)
{
    seq_printf(m, "%d\n", task->ctx);
    return 0;
}

/* create ctx proc file */
static const struct pid_entry tgid_base_stuff[] = {
    /* CHANGE */
    ONE("ctx", S_IRUGO, proc_pid_ctx),
    DIR("task",     S_IRUGO|S_IXUGO, proc_task_inode_operations,
        proc_task_operations),
    DIR("fd",       S_IRUSR|S_IXUSR, proc_fd_inode_operations,
        proc_fd_operations),
    ...
}
```

Listing 4: Create ctx file under proc file system

Thus we are done with the basic part of project 2B.

3 Demo of this project

To show the result of this project, we will write a simple program to read input character from console. Since IO in an OS is usually implemented by interrupt, the process will get scheduled to CPU when there is characters input to the program. The following figure shows that the scheduler counter is correctly implemented.



```
[root@kernel sched]# ./block
hello
world
```

```
[root@kernel ~]# ps -e | grep block
   31 ?        00:00:00 kblockd
 4474 pts/0    00:00:00 block
[root@kernel ~]# cat /proc/4474/ctx
1
[root@kernel ~]# cat /proc/4474/ctx
3
[root@kernel ~]# _
```

Figure 1: Demo of the scheduler counter

As we can see here, when the program is first loaded into memory, it get scheduled to CPU once. Then we typed two lines of text to it and we can find that the `ctx` counter goes up to 3, which is the expected result.

4 Bonus Part: `top` for `ctx`

One bonus part of this project is to implement a program that behaves like the Linux command `top`. This can be easily done by interacting with proc file system. In this project, I simply use Python to do this since Python is a quite handy script programming language, which means that we can do formatting and interact with file system easily.

To make the program display the real-time information of the scheduler counter, we can use the Linux library *curses*. This library can make the display in console much easier and we can repaint the console as we want. The source code is shown as follows:

```python
import os, time, curses

def get_info():
    dirs = [s for s in os.listdir('/proc') if s.isdigit()]
    info = []
    for d in dirs:
        pid = comm = ctx = ''
        pid = d
        with open('/proc/' + d + '/comm') as f:
            comm = f.read().strip('\n')
        with open('/proc/' + d + '/ctx') as f:
            ctx = f.read().strip('\n')
        info.append((pid, comm, ctx))
    return info
```

```

def info_to_strs():
    top_info = sorted(get_info(), key=lambda i: int(i[2]),
        reverse=True)[:20]
    strs = []
    for i in top_info:
        strs.append(i[0].rjust(5) + ' ' + i[1].ljust(15) + i[2].rjust(6))
    return strs


screen = curses.initscr()
screen.keypad(1)
screen.nodelay(1)
curses.noecho()

while (True):
    strs = info_to_strs()
    screen.clear()
    for s in strs:
        screen.addstr(s + "\n")
        screen.refresh()
    key = screen.getch()
    if key == ord('q'):
        break;
    time.sleep(1.0)

curses.endwin()
```

Listing 5: top for ctx

5 Conclusion

In this project, we mainly deal with the Linux process scheduler. I learned a lot from the Linux's completely fair scheduler and have more hands-on with regard to the proc file system. Also, with the help of Arch Linux, now I have a more compact understanding of the Linux philosophy — *keep it simple, stupid*.

Acknowledgement

Thanks Prof. Chen for guidance on Linux kernel and TAs for their hard work.