Linux Kernel Project2 Report

蒋逸伟 517161910005

1 实验平台

云平台：华为鲲鹏云
系统：`Ubuntu 18.10`
Linux Kernel version：`5.6.7`
CPU 架构：`ARM64`

2 进程管理的源代码修改

2.1 为 *task_struct* 结构添加数据成员 int ctx，每当进程被调用一次，$ctx++$。

2.1.1 Step1 在 linux/sched.h 中添加数据成员 ctx

`task_struct` 的局部定义如下：

```
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
        /*
         * For reasons of header soup (see current_thread_info()), this
         * must be the first element of task_struct.
         */
        struct thread_info              thread_info;
#endif
        /* -1 unrunnable, 0 runnable, >0 stopped: */
        volatile long                   state;


        /*
         * This begins the randomizable portion of task_struct. Only
         * scheduling-critical items should be added above here.
         */
        randomized_struct_fields_start

        void                            *stack;
        refcount_t                      usage;
        /* Per task flags (PF_*), defined further below: */
        unsigned int                    flags;
        unsigned int                    ptrace;
```

在该结构体中提示`thread_info`必须放在第一个可以选择 `state`成员前增加`ctx`变量修改后如下：

```
struct task_struct {
#ifdef CONFIG_THREAD_INFO_IN_TASK
        /*
```

```
     * For reasons of header soup (see current_thread_info()), this
     * must be the first element of task_struct.
     */
    struct thread_info              thread_info;
#endif
    /* This element is generated with new process built, it will add
    1 when the process called*/
    unsigned int            ctx;
    /* -1 unrunnable, 0 runnable, >0 stopped: */
    volatile long                   state;
```

2.1.2 Step2 将 ctx 在进程创建时初始化修改 kernel/fork.c

由于 Linux 系统采用写时拷贝技术是通过复制父进程的方式创建一个新进程它依赖于`do_fork()`函数而`do_fork()`调用了`copy_process()`创建子进程的`task_struct`。所以在`copy_process`之后完成子进程`task_struct`中`ctx`的初始化即可。

```
long _do_fork(struct kernel_clone_args *args)
{
        u64 clone_flags = args->flags;
        struct completion vfork;
        struct pid *pid;
        struct task_struct *p;
        int trace = 0;
        long nr;

        /*
         * Determine whether and which event to report to ptracer.  When
         * called from kernel_thread or CLONE_UNTRACED is explicitly
         * requested, no event is reported; otherwise, report if the event
         * for the type of forking is enabled.
         */
        if (!(clone_flags & CLONE_UNTRACED)) {
                if (clone_flags & CLONE_VFORK)
                        trace = PTRACE_EVENT_VFORK;
                else if (args->exit_signal != SIGCHLD)
                        trace = PTRACE_EVENT_CLONE;
                else
                        trace = PTRACE_EVENT_FORK;

                if (likely(!ptrace_event_enabled(current, trace)))
                        trace = 0;
```

```
        }

        p = copy_process(NULL, trace, NUMA_NO_NODE, args);
        add_latent_entropy();

        if (IS_ERR(p))
                return PTR_ERR(p);

        /* initial the ctx after the child process created*/
        p->ctx = 0;
```

2.2 Step3 在调度进程的函数中，找到合适的位置，ctx++

在/kernel/sched/core.c源代码中__schedule()函数下一个进程的选择和，进程切换的功能。它调用了pick_next_task()从run queue中选择下一个进程，同时调用context_switch()完成进程的切换。这里决定在pick_next_task()函数调用后加入next->ctx++;

```
next = pick_next_task(rq, prev, &rf);
next->ctx++;
clear_tsk_need_resched(prev);
clear_preempt_need_resched();
if (likely(prev != next)) {
                rq->nr_switches++;
                /*
                 * RCU users of rcu_dereference(rq->curr) may not see
                 * changes to task_struct made by pick_next_task().
                 */
                RCU_INIT_POINTER(rq->curr, next);
                /*
                 * The membarrier system call requires each architecture
                 * to have a full memory barrier after updating
                 * rq->curr, before returning to user-space.
                 *
                 * Here are the schemes providing that barrier on the
                 * various architectures:
                 * - mm ? switch_mm() : mmdrop() for x86, s390, sparc, PowerPC.
                 *   switch_mm() rely on membarrier_arch_switch_mm() on PowerPC.
                 * - finish_lock_switch() for weakly-ordered
                 *   architectures where spin_unlock is a full barrier,
                 * - switch_to() for arm64 (weakly-ordered, spin_unlock
                 *   is a RELEASE barrier),
                 */
```

```
            ++*switch_count;
            trace_sched_switch(preempt, prev, next);

            /* Also unlocks the rq: */
            rq = context_switch(rq, prev, next, &rf);
```

2.2.1 Step4 把 ctx 输出到/proc/<PID>/ctx 下，通过 cat /proc/<PID>/ctx 可以查看当前指定进程的 ctx 的值。

`struct pid_entry tgid_base_stuff[]`数组中记录着/proc/<pid>/下的文件名称、种类和操作。

```
#define REG(NAME, MODE, fops)                                   \
        NOD(NAME, (S_IFREG|(MODE)), NULL, &fops, {})
#define ONE(NAME, MODE, show)                                   \
        NOD(NAME, (S_IFREG|(MODE)),                             \
                NULL, &proc_single_file_operations,             \
                { .proc_show = show } )
```

这两种宏定义了对应目录下的文件类型这里选择REG型，在base stuff数组中添加前需要加入文件的读写实现。文件定义如下：

REG("ctx",S_IRUSR|S_ IWUSR,proc_pid_ctx_operations) 添加的文件操作函数如下：

```
static int ctx_show(struct seq_file *m, void *v)
{
        struct inode *inode = m->private;
        struct task_struct *p;

        p = get_proc_task(inode);
        if (!p)
                return -ESRCH;
    seq_puts(m, "The ctx in task_struct : ");
    seq_printf(m, " %d\n", p->ctx);
    put_task_struct(p);
        return 0;
}

static int ctx_open(struct inode *inode, struct file *filp)
{
        return single_open(filp, ctx_show, inode);
}

static const struct file_operations proc_pid_ctx_operations = {
        .open           = ctx_open,
```

```
        .read              = seq_read,
        .llseek            = seq_lseek,
        .release           = single_release,
};
```

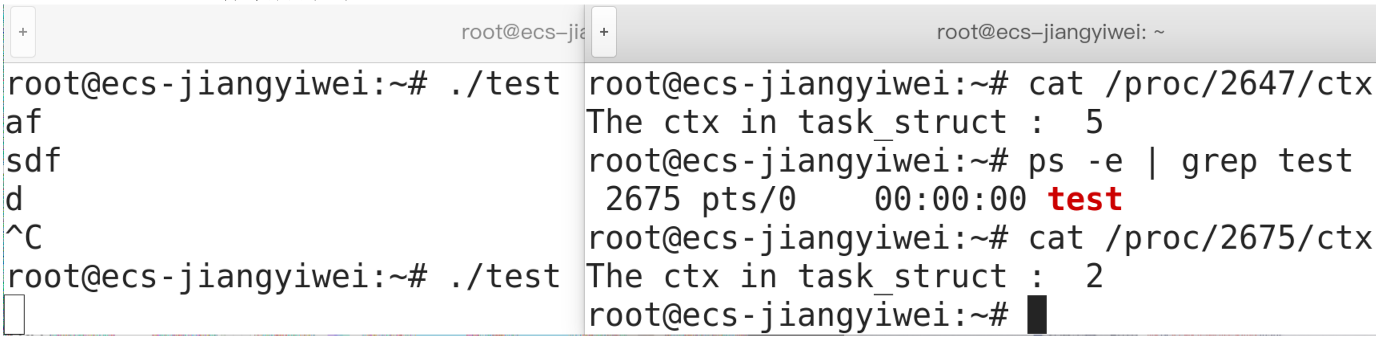
在`ctx_show`函数中利用seqAPI完成文件内容的显示，同时定义`ctx_open()`，调用`ctx_show()`和`single_open()`完成文件的打开，最后定义相应的`file_operation()`实现 ctx 文件的建立和读取。

3 程序调用的测试

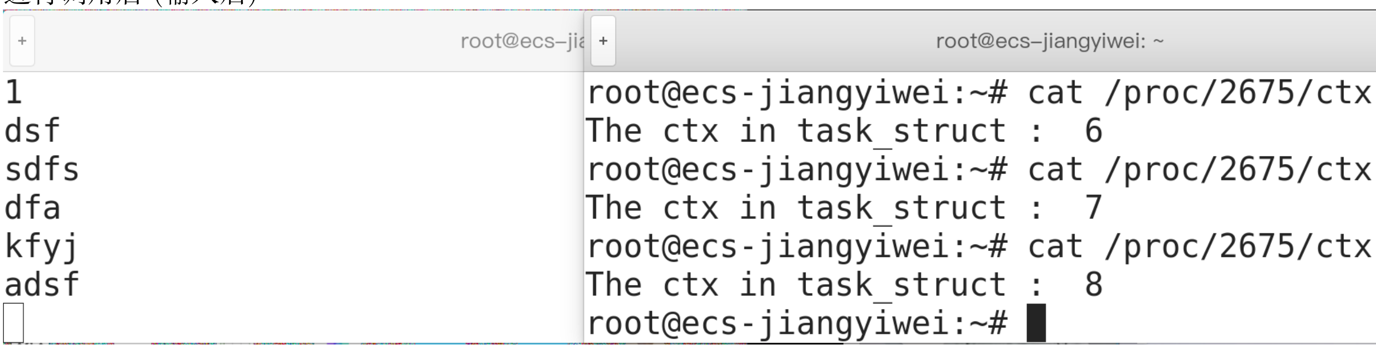
写一个 test.c 程序来循环接收输入，每接收输入一次，它得到一次调度，对应 ctx 加一。利用以下程序完成实验

```
#include<stdio.h>
int main()
{
  while(1) getchar();
  return 0;
}
```

`gcc test.c -o test`编译后得到`test`



进行调用后（输入后）



4 总结

本次实验主要了解了`task_struct`的数据成员，同时学习了进程创建的方法`do_fork()`他会完成对应进程的`task_struct`的数据成员，同时学习了进程创建的方法的创建。同时实现了 `\proc\<pid>`下文件的建立，和相应文件操作函数的实现！