CS353 Linux内核 Project2 报告

517030910214 刘宏洲

0. 简介

在本Project中，我们将研究Linux内核调度有关的代码，并修改内核代码，对进程的调度次数进行计数，并利用proc文件系统进行输出。

本次Project的实验平台为华为云ECS服务器，它采用双核鲲鹏处理器，系统环境配置如下：

- Ubuntu 18.04.3 LTS (GNU/Linux 5.6.6 aarch64)
- GNU Make 4.1
- gcc 7.4.0

1. 实现

1.1 `task_struct`

我们知道，在Linux内核中与进程相关的最重要的数据结构就是`task_struct`。为了实现调度次数的计数，首先要在`task_struct`中声明一个计数器`ctx`。`task_struct`定义在`include/linux/sched.h`中，`task_struct`中定义了各种各样有关的数据成员。根据注释可以知道，定义主要分为三个部分：

- scheduling-critical items
- randomized struct fields
- CPU-specific state

因此，我们需要将

```
1   int ctx;
```

写在randomized struct fields中，并且注意不要写在`#ifdef`和`#endif`构成的语句块之间。我选择将`ctx`声明在randomized struct fields的最后。

1.2 创建进程

声明了`ctx`变量之后，我们还需要对它进行初始化为0的操作。初始化必须在进程创建的最开始进行。因此我找到了`kernel/fork.c`这个文件，其中有创建进程有关的函数。我了解到，`fork()`系统调用实际上嵌套调用了许多相关的内核代码，他们的关系如下：

`fork()` → `clone()` → `do_fork()` → `_do_fork()` → `copy_process()`

其中，`_do_fork()`完成了主要的工作，它调用`copy_process()`完成子进程对父进程的上下文内容以及各种环境数据（包括`task_struct`）的拷贝，然后启动子进程。

在`copy_process()`中，会调用`dup_task_struct()`来得到一个基于父进程`task_struct`的新`task_struct p`，然后对`p`进行一系列的初始化，如果成功，会返回`p`。返回之前的代码如下：

```
1     total_forks++;
2     hlist_del_init(&delayed.node);
3     spin_unlock(&current->sighand->siglock);
4     syscall_tracepoint_update(p);
5     write_unlock_irq(&tasklist_lock);
6
```

```
    proc_fork_connector(p);
    cgroup_post_fork(p);
    cgroup_threadgroup_change_end(current);
    perf_event_fork(p);

    trace_task_newtask(p, clone_flags);
    uprobe_copy_process(p, clone_flags);
    p->ctx = 0;  //Initialize here
    return p;
```

我们在返回之前对`ctx`进行了初始化，程序一旦执行到此处，便预示着新的`task_struct`创建成功，因此在返回前初始化是合理的。

1.3 进程调度

完成对`ctx`的初始化后，就需要在进程被调度的时候增加`ctx`的值。与调度有关的函数在`kernel/sched/core.c`中实现，其调用顺序为：

`schedule()` → `__schedule()`

在`schedule()`函数中，首先调用`sched_submit_work(tsk)`将当前进程提交。然后进入循环，关闭抢占并调用`__schedule()`，再打开抢占，检测是否还需要重新调度。最后使用`sched_update_worker(tsk)`更新`worker`。

在`__schedule()`函数中，首先要获取当前cpu以及对应的runqueue，然后调用`schedule_debug()`进行错误检测。随后使用`pick_next_task(rq, prev, &rf)`选出最适合的下一个进程，得到它的`task_struct` `next`。注意到`next`可能仍然是当前进程的`task_struct`，因此需要判断后，再进行上下文切换，相关代码如下：

```
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);
        rq->curr->ctx++;
    } else {
        rq->clock_update_flags &= ~(RQCF_ACT_SKIP|RQCF_REQ_SKIP);
        rq_unlock_irq(rq, &rf);
    }

    balance_callback(rq);
```

可以看到，在`if`语句块中，调用`context_switch`进行了上下文切换，切换后的runqueue `rq`的`curr`就是被调度上来的新进程。此时我们对其`ctx`计数器自增即可。在这里，我认为，如果`next`和`prev`是同一个进程，那么就相当于没有被调度，因此没有在`if-else`语句块之前或之后添加`next->ctx++`。但我也尝试过在获得`next`后对`ctx`自增，这可能会导致测试程序刚运行时`ctx`值的一些差异，但每次输入输出时`ctx`增加1的效果并没有差距。

1.4 输出至`/proc/pid/ctx`

我们知道`/proc`文件夹下有许多数字命名的文件夹，这些数字就是一个个进程的pid。现在我们需要将`ctx`的数值输出到某个进程pid的文件夹下的`ctx`文件。为此，我们需要生成有关的文件并定义读操作。在`fs/proc/base.c`中定义了进程对应目录下的文件及文件夹。

在`base.c`中定义了一个数组`tgid_base_stuff`，它是由一个个`pid_entry`构成的。`pid_entry`的定义如下：

```
struct pid_entry {
    const char *name;
    unsigned int len;
    umode_t mode;
    const struct inode_operations *iop;
    const struct file_operations *fop;
    union proc_op op;
};
```

其中定义了文件名、长度、读写模式、inode操作、文件操作等。同时，为了简单起见，还定义了`DIR`、`REG`、`ONE`、`LNK`等宏，用于产生不同类型的`pid_entry`。`DIR`代表文件夹，`REG`代表普通文件，`ONE`代表只有一种文件操作的文件，`LNK`代表链接。由于只需要读文件内容，使用`ONE`和`REG`都可以。为了避免`#ifdef`可能的影响，我将`ctx`的`pid_entry`写在数组的最后一项，即

```
REG("ctx", S_IRUSR|S_IWUSR, proc_ctx_operations)
```

其中`S_IRUSR|S_IWUSR`表示用户的读写权限，即0600权限，也可以只声明为只读（0400）权限，`proc_ctx_operations`是一个`file_operations`指针，指向了文件操作相关的结构体。它的声明如下：

```
static ssize_t ctx_read(struct file *file, char __user *buf,
        size_t count, loff_t *ppos)
{
    struct task_struct *task;
    int ctx;
    size_t len;
    char buffer[64];
```

```
    task = get_proc_task(file_inode(file));
    if (!task)
        return -ESRCH;
    ctx = task->ctx;
    len = snprintf(buffer, sizeof(buffer), "%d\n", ctx);
    return simple_read_from_buffer(buf, count, ppos, buffer, len);
}
static const struct file_operations proc_ctx_operations = {
    .read = ctx_read,
};
```

这个结构体在Project1中使用过，这里只需要赋值`.read`即可。`ctx_read`函数和Project1 Module3中的读函数差别不大。但这里使用了`get_proc_task`以及`file_inode`，根据传入的`file`参数得到`task_struct`。然后即可读取当前进程的`ctx`值。再使用`snprintf`将`ctx`转换为字符串打印至`buffer`。最后要返回调用`simple_read_from_buffer`的结果，这个函数将`buffer`内容复制给了用户空间的`buf`。这样的写法参考了本文件中其他读函数的写法。如果使用原始的`copy_to_user`，然后返回`len`，则会造成无限打印的bug。

至此，本次Project对内核代码的修改就完成了。

2. 结果

为了测试对内核代码修改的效果，我编写了一个简单的C程序`test.c`，它接受一个输入，然后马上打印，代码如下：

```
#include <stdio.h>

int main(void)
{
    while(1)
    {
        char c;
        c = getchar();
        putchar(c);
    }
    return 0;
}
```

运行这个程序，然后使用`ps -e | grep test`获得`test`的`pid`，再利用`sudo cat /proc/pid/ctx`，就可以得到`ctx`的当前值，结果如下：

图1. 结果1

可以看到，`ctx`在程序未接收输入时的初始值为0，每一次输入输出都会使得`ctx`增加1。

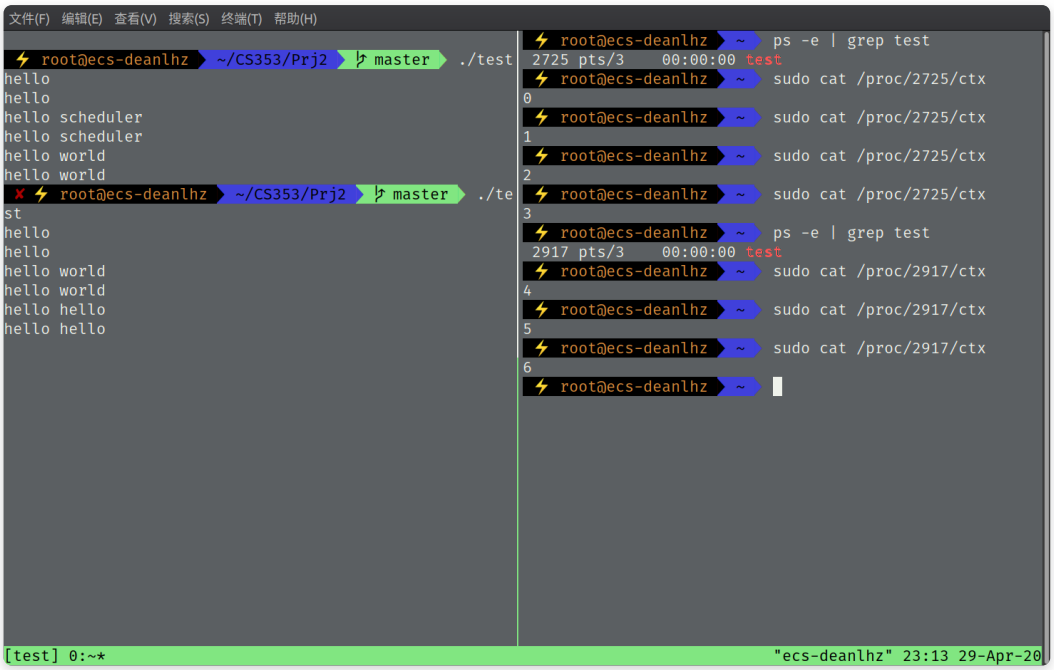

图2. 结果2

结束`test`再运行，发现`ctx`的初始值发生了变化，但每一次输入输出仍然会使得`ctx`增加1。

图3. 结果3

将`test.c`中的`putchar(c)`注释掉再编译运行，发现初始值又发生了变化，但输入仍然会导致`ctx`增加1。

对于以上实验结果，我发现无论只执行输入还是输入后输出，每一次`ctx`只增加1，但初始值不相同。甚至两次运行同样的程序，`ctx`的初值都不相同。初始值不同的现象可能有以下原因：

- 在输入`./test`运行，然后切换到右边，再到首次输出之间发生了一些调度，导致`test`被调度了几次
- 输入输出相关的动态库可能需要被调用，导致`test`的调度（注意到代码中有`io_schedule`函数，它也调用了`schedule`）

至于为何只执行输入和执行输入输出都只会使`ctx`增加1，我猜想是由于这样紧密连续执行的I/O操作并不会引起`test`被调度，但每一次开始I/O操作都会使得`test`被调度。但要注意的是，如果两次输入之间时间间隔较大，会出现`ctx`不止增加1的情况。因为其他高优先级的进程会抢占`test`，从而发生多次调度。

3. 总结与感想

在本次Project中，我对内核代码进行了修改，并且详细地了解了Linux内核中有关进程以及进程调度的实现，并且加深了对proc文件系统的理解。虽然代码量很小，但阅读内核源码并寻找相关内容的过程也具有一定挑战性。在此过程中，我惊讶于Linux内核的庞大、复杂但不失精巧，激发了我进一步探索Linux内核的兴趣。