Linux Kernal Lab Report 2

Process Management

朱祥辉 5180519100018
网络空间安全学院
954305012@qq.com

一、实验内容

此次实验内容为Linux内核的进程管理，具体是：

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. 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。

二、实验简介

（1）Linux内核编译

内核编程相对于普通的编程来说，需要接触到的内容更加的低层。需要编程者对于操作系统的知识有一个较为深刻的了解。不仅要考虑到写出来的模块与操作系统的兼容性，还需要考虑进程之间是如何通信的，并且有时候还要求考虑如何管理硬件。最后，接口的稳定性必不可少。

（2）华为云弹性服务器

此次实验环境为华为云ECS弹性服务器，又叫“云服务器”，是由CPU、内存、镜像、云硬盘组成的一种可随时获取、弹性可扩展的计算服务器。同时结合VPC、安全组、数据多副本保存，支持负载均衡等能力，确保服务持久稳定运行。

（3）Linux进程管理

进程是正在运行的程序实体，并且包括这个运行的程序中占据的所有系统资源，比如说CPU（寄存器），IO,内存，网络资源等。操作系统的职能之一，主要是对处理机进行管理。为了提高CPU的利用率而采用多道程序技术。通过进程管理来协调多道程序之间的关系，使CPU得到充分的利用。

三、实验过程与效果截图

1、修改进程描述符（task_struct）

`task_struct` 是存在于内核代码的 `include/linux/sched.h` 文件中用来描述进程的数据结构，可以理解为进程的属性。比如进程的状态、进程的标识（PID）等，都被封装在了进程描述符这个数据结构中。下面我将选取部分成员进行分析：

- **state成员**

```
volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */
```

该值的可取值为以下值，分别表示**进程的不同状态**。（运行态、可打扰态、不可打扰态、停止执行态、监控态 etc.）

```
/* Used in tsk->state: */
#define TASK_RUNNING                    0x0000
#define TASK_INTERRUPTIBLE              0x0001
#define TASK_UNINTERRUPTIBLE            0x0002
#define __TASK_STOPPED                  0x0004
#define __TASK_TRACED                   0x0008
/* Used in tsk->exit_state: */
#define EXIT_DEAD                       0x0010
#define EXIT_ZOMBIE                     0x0020
#define EXIT_TRACE                      (EXIT_ZOMBIE | EXIT_DEAD)
/* Used in tsk->state again: */
#define TASK_PARKED                     0x0040
#define TASK_DEAD                       0x0080
#define TASK_WAKEKILL                   0x0100
#define TASK_WAKING                     0x0200
#define TASK_NOLOAD                     0x0400
#define TASK_NEW                        0x0800
#define TASK_STATE_MAX                  0x1000
```

- **pid和tgid成员**

```
    pid_t    pid;   /*进程的唯一标识*/
    pid_t    tgid;  /*线程组的领头线程的pid成员的值*/
```

Unix系统通过`pid`来标识进程，linux把不同的`pid`与系统中每个进程或轻量级线程关联，而unix程序员希望**同一组线程具有共同的pid**，遵照这个标准linux引入线程组的概念。一个线程组所有线程与领头线程具有相同的`pid`，存入`tgid`字段，getpid()返回当前进程的`tgid`值而不是`pid`的值。

★ 具体修改

对于每一个进程而言，**进程描述符task_struct**都是该进程独有的。为了管理进程，内核必须知道每个进程的信息与其所作的事情（进程优先级、分配的地址空间等）。每次执行程序时，内核都会根据**进程描述符**对进程进行相应的操作，同样的，也能对进程描述符中的值进行修改。在此次实验中，由于需要记录某个进程被调用的次数，因此需要**修改task_struct结构体中成员**。

```
struct task_struct {
        //在task_struct此处添加成员ctx
        int     ctx;
#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
```

由于**ifdef**等命令会导致某些成员没有被定义，因此我将ctx成员变量添加到`#ifdef`~`#endif`之外。

2、修改_do_fork()函数

- 首先我观察到**fork()**和**vfork()**函数的调用。

```c
#ifdef __ARCH_WANT_SYS_FORK
SYSCALL_DEFINE0(fork)
{
#ifdef CONFIG_MMU
    struct kernel_clone_args args = {
        .exit_signal = SIGCHLD,
    };

    return _do_fork(&args);
#else
    /* can not support in nommu mode */
    return -EINVAL;
#endif
}
#endif

#ifdef __ARCH_WANT_SYS_VFORK
SYSCALL_DEFINE0(vfork)
{
    struct kernel_clone_args args = {
        .flags      = CLONE_VFORK | CLONE_VM,
        .exit_signal    = SIGCHLD,
    };

    return _do_fork(&args);
}
#endif
```

- **fork()和vfork()的区别**：**fork()**函数用于创建新进程，且创建出的子进程是父进程的副本，**完全复制**父进程的数据空间，堆、栈的副本。而**vfork()**创建的进程并不将父进程的地址空间完全复制倒子进程，子进程在调用exec之前在运行在**父进程的空间**，并且**vfork()**保证子进程**先于父进程运行**，调用exec之后才会使得父进程开始执行。
- 同样地，除了上述的fork()和vfork()函数之外，还有如下函数也会直接调用_do_fork()。

```c
#ifndef CONFIG_HAVE_COPY_THREAD_TLS
/* For compatibility with architectures that call do_fork directly rather than
 * using the syscall entry points below. */
long do_fork(unsigned long clone_flags,
             unsigned long stack_start,
             unsigned long stack_size,
             int __user *parent_tidptr,
             int __user *child_tidptr)
{
    struct kernel_clone_args args = {
        .flags      = (clone_flags & ~CSIGNAL),
        .pidfd      = parent_tidptr,
        .child_tid  = child_tidptr,
        .parent_tid = parent_tidptr,
        .exit_signal    = (clone_flags & CSIGNAL),
        .stack      = stack_start,
        .stack_size = stack_size,
    };

    if (!legacy_clone_args_valid(&args))
        return -EINVAL;

    return _do_fork(&args);
}
#endif
```

do_fork函数调用_do_fork()

```
/*
 * Create a kernel thread.
 */
pid_t kernel_thread(int (*fn)(void *), void *arg, unsigned long flags)
{
    struct kernel_clone_args args = {
        .flags      = ((flags | CLONE_VM | CLONE_UNTRACED) & ~CSIGNAL),
        .exit_signal    = (flags & CSIGNAL),
        .stack      = (unsigned long)fn,
        .stack_size = (unsigned long)arg,
    };

    return _do_fork(&args);
}
```

kernel_thread函数调用_do_fork()

- 从**上述定义**中可以看出进程创建的**大部分工作**由**_do_fork()**函数完成。仔细观察可以发现在函数中首先建立了**kernel_clone_args**类型的结构体，作用是将父进程的参数存进去，从而传给**_do_fork()**函数。接下来的工作由**_do_fork()**函数完成。
- 最后我将目标定位到**_do_fork()**函数中。

★ 具体修改

```
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;
    ...
    /*
     * copy_process（）会用当前进程的一个副本来创建新进程并分配pid，但不会实际启动这个新进程。
     * 它会复制寄存器中的值、所有与进程环境相关的部分，以及每个clone标志。
     * 新进程的实际启动由调用者来完成。
     */
    p = copy_process(NULL, trace, NUMA_NO_NODE, args);
    add_latent_entropy();

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

    //在此处初始化新进程的ctx变量.
    p->ctx = 0;
```

- 在通过copy_process()进行得到子进程的**进程描述符指针**后，程序通过IS_ERR(p)来判断该指针是否出错。初始化ctx变量部分就**放在判断之后**。

3、修改__schedule函数

- 首先观察**schedule**函数。schedule就是**主调度器**的函数, 在内核中的许多地方, 如果要将CPU分配给与当前活动进程不同的另一个进程, 都会直接调用主调度器函数schedule。

```
asmlinkage __visible void __sched schedule(void)
{
    struct task_struct *tsk = current;

    sched_submit_work(tsk);                 /*  避免死锁  */
    do {
        preempt_disable();                  /*  关闭内核抢占  */
        __schedule(false);                  /*  完成调度  */
        sched_preempt_enable_no_resched();  /*  开启内核抢占  */
    } while (need_resched());   //内核在即将返回用户空间时检查进程是否需要重新调度。如果设置了，就会发生调度。
    sched_update_worker(tsk);
}
EXPORT_SYMBOL(schedule);
```

- 说明一下**__sched**前缀，该前缀可以用于调用schedule函数，包括schedule本身。该前缀定义在include/kernel/sched/debug.h文件中。其中**attribute**((*section*("...")))是一个gcc的编译属性，其目的是在将相关的函数放在代码编译之后，放到目标文件特定的段内（sched.txt)

```
/* Attach to any functions which should be ignored in wchan output. */
#define __sched     __attribute__((__section__(".sched.text")))
```

- 为什么要在**完成调度之前关闭抢占**?
  - 在内核完成调度器过程中, 如果发生了内核抢占, 我们的调度会被中断, 而调度却还没有完成, 这样会丢失我们调度的信息。
- schedule函数完成如下工作:
  - 确定当前就绪队列, 并在保存一个指向当前(仍然)活动进程的task_struct指针。
  - 检查死锁, 关闭内核抢占后调用__schedule完成内核调度。
  - 恢复内核抢占, 然后检查当前进程是否设置了重调度标志TLF_NEDD_RESCHED,如果该进程被其他进程设置了TIF_NEED_RESCHED标志, 则函数重新执行进行调度。

★ 具体修改

```
static void __sched notrace __schedule(bool preempt)
{
    struct task_struct *prev, *next;
    unsigned long *switch_count;
    struct rq_flags rf;
    struct rq *rq;
    int cpu;

    cpu = smp_processor_id();       //获取当前CPU
    rq = cpu_rq(cpu);               //获取当前CPU的使用队列
    prev = rq->curr;                //将当前运行的队列存入Prev中

    //此处进行自增
    prev->ctx++;

    schedule_debug(prev, preempt);
```

- 在__schedule函数中得到**当前cpu的runqueue队列**，然后从中得到当前运行的进程描述符指针**prev**。随后在此处将当前调度的进程的计数器自增。

4、修改tgid_base_stuff结构体数组 && 添加proc_pid_ctx函数

★ 具体修改

```
static const struct pid_entry tgid_base_stuff[] = {
    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),
    DIR("map_files",  S_IRUSR|S_IXUSR, proc_map_files_inode_operations,
proc_map_files_operations),
    DIR("fdinfo",     S_IRUSR|S_IXUSR, proc_fdinfo_inode_operations,
proc_fdinfo_operations),
    DIR("ns",     S_IRUSR|S_IXUGO, proc_ns_dir_inode_operations,
proc_ns_dir_operations),
    //在此处添加ctx目录
    ONE("ctx",  S_IRUSR, proc_pid_ctx),
    ...
```

- 通过与/proc/<PID>/文件夹下的路径名进行对比，可以很清晰地看出tgid_base_stuff结构体数组的**整体构造**——tgid_base_stuff结构体数组主要组成元素为三个类别的结构体。
  - **DIR(NAME, MODE, iops, fops)**：表示为在/proc/<PID>/文件夹下创建名为NAME的**文件夹**。并且定义了**索引节点操作**以及**文件读写操作**。
  - **REG(NAME, MODE, fops)**：在/proc/<PID>/文件夹下创建名为NAME的文件，并定义了**文件读写操作**。
  - **ONE(NAME, MODE, show)**：在/proc/<PID>/文件夹下创建名为NAME的文件，并且只提供**文件读操作**。

  由于此次实验只需要读ctx文件，因此添加ONE(NAME, MODE, show)即可。

★ 具体修改

```
//ctx文件的读操作
static int proc_pid_ctx(struct seq_file *m, struct pid_namespace *ns,
                struct pid *pid, struct task_struct *task)
{
    int err = lock_trace(task);
    if (!err) {
        seq_printf(m, "%d\n", task->ctx);    //将该进程的ctx值显示出来
        unlock_trace(task);
    }
    return err;
}
```

- 在tgid_base_stuff结构体数组的上方定义**proc_pid_ctx**函数即可。该函数的作用是在ctx文件里显示task->ctx的值。

四、反思与总结

思考：

总的来说，此次实验难度主要体现在**两个方面**。

1、面对庞大的内核源码，**阅读起来非常费劲**。完全不清楚应该从何处开始下手是本次实验的常态。更由于此次实验需要修改四个文件中的内容，且每个文件的代码含量都可以说非常庞大，于是从何处开始、如何读懂是非常困难的。

2、尽管此次对内核源码的修改并不多，但是每一次修改都需要对**整个内核**重新进行编译。每次编译的耗费时长是巨大的。这导致的直接原因是每一次对内核的修改都必须要小心翼翼，再三检查多次，才可以进行编译。好在此次实验需要修改的代码量很少，从而可以让我保证质量的完成此次实验。

总结：

在此次实验过程中，我逐渐掌握到了阅读源码的诀窍。特别是面对几百万行的源码，一开始我是望而却步的，完全找不到下手的位置。后来通过google等搜索引擎，找到了不少内核开发者对内核源码的**解析和注解**。这让我对各个文件、各个函数的作用有了较为清晰的认知。随着对代码阅读的不断加深，我渐渐发现读内核代码似乎不是想象中这么困难。我也因此掌握了以下几点诀窍：

1、内核源码的**函数名**大多是指向性非常强的。很多时候仅仅通过函数名就可以知道该函数的作用是什么。例如此次实验中的_do_fork函数、schedule函数等，仅仅看到名字就知道其在进程创建、进程调度中的作用。

2、**善用插件。**本次源码阅读过程中，我使用的文本编辑器是Vscode，并且下载了**C/C++插件**。这一插件大大降低了我阅读源码的难度。每次看到没有找到的函数，通过**Go to definition**选项可以很快找到该函数的定义位置。这一功能比直接Ctrl+F搜索要方便得多。