Linux Kernel Project 2: Process Management

薛春宇 518021910698

1 实验要求

- 为 `task_struct` 结构添加数据成员 `int ctx`，每当进程被调用一次，`ctx++`
- 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中 *ctx* 的值

2 实验环境

- *Linux OS* 版本：*Ubuntu 18.04*
- *Kernel* 版本：*Linux-5.5.11*
- *Gcc* 版本：*7.4.0*

3 实验内容

本节中，将分别介绍 *Linux* 进程管理数据结构、进程创建、进程调度、*proc entry* 创建的具体实现函数，并通过添加并观察 *ctx* 变量的方式验证实验效果。

3.1 进程管理数据结构 *task_struct*

Task_struct 是 *Linux* 进程描述符的实现，包含一个进程所需的所有信息，被用来管理进程。*Task_struct* 定义在 *Linux* 源码中的 `/include/linux/sched.h` 头文件中，结构头位于 *line 629*。由于本次实验的 *ctx* 变量无论在何环境下都应该被创建，即不应被某对 `#ifdef` 和 `#endif` 所包含，因此选择在 *line 675* 插入 *int* 型变量 *ctx* 的声明。

```
struct task_struct {
  // ...

    /*  declare ctx in line 675 of /include/linux/sched.h  */
    int        ctx;

    int        on_rq;

    int        prio;
    int        static_prio;
    int        normal_prio;
    unsigned int       rt_priority;

  // ...
```

```
}
```

3.2 进程创建

Linux 进程创建的实现源码位于 *Linux* 源码中的 `/kernel/fork.c` 文件中。

首先分析 *Linux* 关于进程创建的三个系统调用 `fork()`、`vfork()` 和 `clone()`，源码位于 *line 2514 ~ 2579*，发现三种系统调用在进行参数设置后，均将参数引用 `&args` 通过 `_do_fork()` 函数返回，猜测进程创建的相关实现与该函数相关。

继续寻找 `_do_fork()` 函数的定义，位于 *line 2397 ~ 2461*，源码中给出了该函数的解释：

```
/*
 *  Ok, this is the main fork-routine.
 *
 * It copies the process, and if successful kick-starts
 * it and waits for it to finish using the VM if required.
 *
 * args->exit_signal is expected to be checked for sanity by the caller.
 */
long _do_fork(struct kernel_clone_args *args)     // Line 2397
{
  // ...
}
```

可以看出，`_do_fork()` 接收相关参数，并完成进程的复制。阅读该函数中的 *line 2424 ~ 2434*，发现 `_do_fork()` 首先调用了 `copy_process()` 函数，接着添加 *latent entropy*，随后调用 `trace_sched_process_fork()` 函数唤醒新的线程。因此推测进程创建的工作会在 `copy_process()` 函数中完成。

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

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

  /*
   * Do this prior waking up the new thread - the thread pointer
   * might get invalid after that point, if the thread exits quickly.
   */
  trace_sched_process_fork(current, p);
```

接着寻找 `copy_process()` 函数的定义，函数头位于 *line 1824*，同时给出该方法的解释：

```
/*
 * This creates a new process as a copy of the old one,
 * but does not actually start it yet.
 *
```

```
 * It copies the registers, and all the appropriate
 * parts of the process environment (as per the clone
 * flags). The actual kick-off is left to the caller.
 */
static __latent_entropy struct task_struct *copy_process(
          struct pid *pid,
          int trace,
          int node,
          struct kernel_clone_args *args)
{
  // ...
}
```

由注释信息可知，该函数基于已有进程创建一个拷贝，包含寄存器等所有进程环境中合适的部分，但并不会立刻运行该拷贝进程。通过阅读该函数的实现（*line 1824 ~ 2354*）判断该方法是 *Linux* 进程创建的底层实现，因此在该函数中选择合适的位置进行 *ctx* 的初始化（注意不要被某对 `#ifdef` 和 `#endif` 所包含）：

```
static __latent_entropy struct task_struct *copy_process(
          struct pid *pid,
          int trace,
          int node,
          struct kernel_clone_args *args)
{
  //...

  /*  initialize ctx in line 2042 of /kernel/fork.c */
    p->ctx = 0;

    /* Perform scheduler related setup. Assign this task to a CPU. */
    retval = sched_fork(clone_flags, p);
    if (retval)
      goto bad_fork_cleanup_policy;

  // ...
}
```

3.3 进程调度

```
  *Linux* 进程调度的实现源码位于 `/kernel/sched/core.c` 文件中。找到 `schedule()` 函数，位于 *line 4154 ~ 4167*，可以发现进程调度的宏观控制均发生在该函数中，因此，直接在该函数中寻找合适的地方插入 *ctx* 的更新语句：
```

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

  /* increment ctx in line 4159 */
```

```
  tsk->ctx ++;

  sched_submit_work(tsk);
  do {
    preempt_disable();
    __schedule(false);
    sched_preempt_enable_no_resched();
  } while (need_resched());
  sched_update_worker(tsk);
}
EXPORT_SYMBOL(schedule);
```

注意到，该函数会首先创建一个指向当前进程的 *task_struct* 指针 *tsk*，并调用 `sched_submit_work()` 函数进行进程运行态以及死锁的检测，然后在循环中反复进行该进程的调度，首先使用 `preempt_disable()` 函数禁用内核抢占，然后调用 `__schedule()` 函数进行进程调度。`__schedule()` 是真正进行进程调度实现的函数，位于 *line 4007 ~ 4094*。

3.4 *proc entry* 创建

Linux /proc 文件系统的 *PID* 对应目录下的文件/文件夹创建实现位于 `/fs/proc/base.c` 文件中，该文件 *line 3011* 中定义的 *pid_entry* 类型的结构体 `tgid_base_stuff[]` 中定义了 *PID* 下可访问的变量。参照 *personality* 变量的声明格式，我们还需要定义一个 *handle function*，用来在调用 `cat /proc/[PID]/ctx` 命令时使用 `seq_printf(m, "%d\n", task->ctx);` 将 *ctx* 的值打印到命令行：

```
/* get task->ctx in line 2994 */
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);
    unlock_trace(task);
  }
  return err;
}
```

选择以 *personality* 变量相应的函数 `proc_pid_personality()` 作为参考的原因是，该函数含有错误处理机制，具有更好的容错性。同时，我们需要在静态常量数组 `tgid_base_stuff[]` 中使用 `ONE("ctx", S_IRUSR, proc_pid_ctx)` 指令添加 *proc entry*，其中 *ONE* 指令用于只读文件的声明。

3.5 重新编译、安装内核

- 使用国内源下载 *5.5.11* 版本的 *Linux Kernel*：

  ```
  wget http://mirror.bjtu.edu.cn/kernel/linux/kernel/v5.x/linux-5.5.11.tar.xz
  ```

  解压到特定目录：

```
tar xvf linux-5.5.11.tar.xz -C /usr/src
```

- 安装相关依赖：

```
apt-get install gcc make libncurses5-dev openssl libssl-dev
apt-get install build-essential
apt-get install pkg-config
apt-get install libc6-dev
apt-get install bison
apt-get install flex
apt-get install libelf-dev
```

- 替换文件：

```
cp sched.h /usr/src/linux-5.5.11/include/linux/sched.h
cp fork.c /usr/src/linux-5.5.11/kernel/fork.c
cp core.c /usr/src/linux-5.5.11/kernel/sched/core.c
cp base.c /usr/src/linux-5.5.11/fs/proc/base.c
```

- 编译及安装：

```
cd /usr/src/linux-5.5.11
make menuconfig
make
make modules_install
make install
shutdown -r now
```

- 验证内核版本：

```
Last login: Thu Apr 29 00:17:03 2021 from 59.78.43.80
root@linux-kernel-server:~# uname -a
Linux linux-kernel-server 5.5.11 #1 SMP Thu Apr 29 00:04:18 CST 2021 aarch64 aarch64 aarch64 GNU/Linux
root@linux-kernel-server:~#
```

---

4 实验结果

创建测试程序 `test.c`：

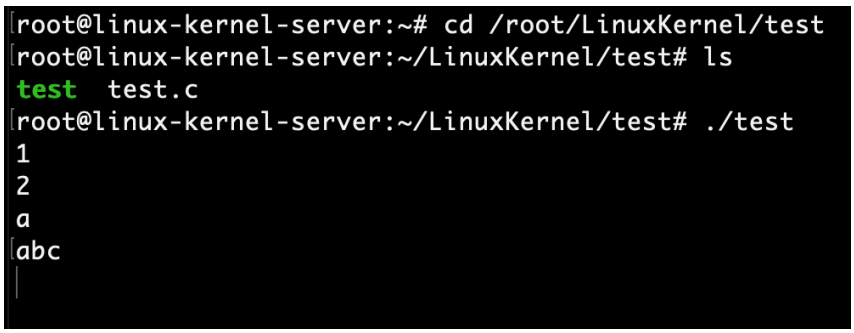
```
#include <stdio.h>
int main(){
  while(1) getchar();
  return 0;
}
```

并使用 *gcc* 进行编译：

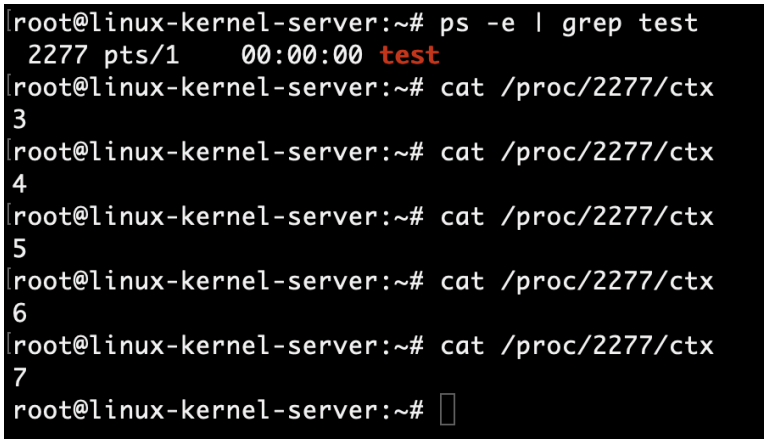
```
gcc test.c -o test
```

在本机同时与云服务器建立两个连接：

- 连接 *A* 进入目录 `/root/LinuxKernel/test`，运行编译好的程序 `./test`，并连续输入字符和回车



- 连接 *B* 使用 `ps -e | grep test` 指令获取 *test* 的 *PID*，并连续使用 `cat /proc/[PID]/test` 来检查 *ctx* 的值



---

5 实验心得

本次实验是 *Linux* 内核课程的第二次正式 project，旨在掌握面向 *Linux* 进程管理数据结构 *task_struct*、进程创建和调度，以及 *proc entry* 创建等的基本知识。在实验过程中，我首先在 *ECS* 上下载了 *linux-5.5.11* 版本的 *kernel* 源码，然后将 `sched.h`、`fork.c` 等需要修改的文件通过 *SFTP get* 到本机，修改完之后再 *put* 传回服务器。为了更加高效地阅读源码，我首先对每份代码的关键字进行了检索，例如在 `fork.c` 中搜索关键词 *fork*，然后对结果进行分析，结合该部分的代码和注释筛选出有效信息，简单了解左右后再逐行阅读该部分的源码，找到更深层次的函数调用。由于目前网上关于修改 *sched.h* 等文件的教程少之又少，因此我必须进行不断的试错。例如，究竟是将代码添加在更加宏观的函数调用中，还是深入到底层的实现中添加（典型的例子是 `schedule()` 和其中的 `__schedule()` 函数）。再经过三次试错后的重新编译后，我终于完成了本实验的要求。

本次实验中，我不仅一定程度上掌握了 *Linux* 源码阅读的方法，增强了自身阅读源码的能力，还提高了发现问题、解决问题的能力。希望能在下面的实验中也能收获满满!