МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Федеральное государственное автономное
образовательное учреждение высшего образования
«Национальный исследовательский университет ИТМО»

ФАКУЛЬТЕТ ПРОГРАММНОЙ ИНЖЕНЕРИИ И КОМПЬЮТЕРНОЙ ТЕХНИКИ

ЛАБОРАТОРНАЯ РАБОТА №2

по дисциплине

‘Операционные системы’

Вариант: ioctl: signal_struct, syscall_info

Выполнил:

Студент группы P33312

Соболев Иван
Александрович

Преподаватель:

Пашнин Александр
Денисович

Санкт-Петербург, 2023

Задание:

Разработать комплекс программ на пользовательском уровне и уровне ярда, который собирает информацию на стороне ядра и передает информацию на уровень пользователя, и выводит ее в удобном для чтения человеком виде. Программа на уровне пользователя получает на вход аргумент(ы) командной строки (не адрес!), позволяющие идентифицировать из системных таблиц необходимый путь до целевой структуры, осуществляет передачу на уровень ядра, получает информацию из данной структуры и распечатывает структуру в стандартный вывод. Загружаемый модуль ядра принимает запрос через указанный в задании интерфейс, определяет путь до целевой структуры по переданному запросу и возвращает результат на уровень пользователя.

Интерфейс передачи между программой пользователя и ядром и целевая структура задается преподавателем. Интерфейс передачи может быть один из следующих:

1. syscall - интерфейс системных вызовов.
2. ioctl - передача параметров через управляющий вызов к файлу/устройству.
3. procfs - файловая система /proc, передача параметров через запись в файл.
4. debugfs - отладочная файловая система /sys/kernel/debug, передача параметров через запись в файл.

Целевая структура может быть задана двумя способами:

1. Именем структуры в заголовочных файлах Linux
2. Файлом в каталоге /proc. В этом случае необходимо определить целевую структуру по пути файла в /proc и выводимым данным.

Описание структур:

1. signal_struct

Defined in 1 files as a struct:

include/linux/sched/signal.h, line 93 *(as a struct)*

```c
/*
 * NOTE! "signal_struct" does not have its own
 * locking, because a shared signal_struct always
 * implies a shared sighand_struct, so locking
 * sighand_struct is always a proper superset of
 * the locking of signal_struct.
 */
struct signal_struct {
        refcount_t              sigcnt;
        atomic_t                live;
        int                     nr_threads;
        int                     quick_threads;
        struct list_head        thread_head;

        wait_queue_head_t       wait_chldexit;  /* for wait4() */

        /* current thread group signal load-balancing target: */
        struct task_struct      *curr_target;

        /* shared signal handling: */
        struct sigpending       shared_pending;

        /* For collecting multiprocess signals during fork */
        struct hlist_head       multiprocess;

        /* thread group exit support */
        int                     group_exit_code;
        /* notify group_exec_task when notify_count is less or equal to 0 */
        int                     notify_count;
        struct task_struct      *group_exec_task;

        /* thread group stop support, overloads group_exit_code too */
        int                     group_stop_count;
        unsigned int            flags; /* see SIGNAL_* flags below */

        struct core_state *core_state; /* coredumping support */

        /*
         * PR_SET_CHILD_SUBREAPER marks a process, like a service
         * manager, to re-parent orphan (double-forking) child processes
         * to this process instead of 'init'. The service manager is
         * able to receive SIGCHLD signals and is able to investigate
         * the process until it calls wait(). All children of this
         * process will inherit a flag if they should look for a
         * child_subreaper process at exit.
         */
        unsigned int            is_child_subreaper:1;
        unsigned int            has_child_subreaper:1;

#ifdef CONFIG_POSIX_TIMERS

        /* POSIX.1b Interval Timers */
        unsigned int            next_posix_timer_id;
        struct list_head        posix_timers;

        /* ITIMER_REAL timer for the process */
        struct hrtimer real_timer;
        ktime_t it_real_incr;

        /*
         * ITIMER_PROF and ITIMER_VIRTUAL timers for the process, we use
         * CPUCLOCK_PROF and CPUCLOCK_VIRT for indexing array as these
         * values are defined to 0 and 1 respectively
         */
        struct cpu_itimer it[2];

        /*
         * Thread group totals for process CPU timers.
         * See thread_group_cputimer(), et al, for details.
         */
        struct thread_group_cputimer cputimer;

#endif
        /* Empty if CONFIG_POSIX_TIMERS=n */
        struct posix_cputimers posix_cputimers;

        /* PID/PID hash table linkage. */
        struct pid *pids[PIDTYPE_MAX];

#ifdef CONFIG_NO_HZ_FULL
        atomic_t tick_dep_mask;
#endif

        struct pid *tty_old_pgrp;

        /* boolean value for session group leader */
        int leader;

        struct tty_struct *tty; /* NULL if no tty */

#ifdef CONFIG_SCHED_AUTOGROUP
        struct autogroup *autogroup;
#endif
        /*
         * Cumulative resource counters for dead threads in the group,
         * and for reaped dead child processes forked by this group.
         * Live threads maintain their own counters and add to these
         * in __exit_signal, except for the group leader.
         */
        seqlock_t stats_lock;
        u64 utime, stime, cutime, cstime;
        u64 gtime;
        u64 cgtime;
        struct prev_cputime prev_cputime;
        unsigned long nvcsw, nivcsw, cnvcsw, cnivcsw;
        unsigned long min_flt, maj_flt, cmin_flt, cmaj_flt;
        unsigned long inblock, oublock, cinblock, coublock;
        unsigned long maxrss, cmaxrss;
        struct task_io_accounting ioac;

        /*
         * Cumulative ns of schedule CPU time fo dead threads in the
         * group, not including a zombie group leader, (This only differs
         * from jiffies_to_ns(utime + stime) if sched_clock uses something
         * other than jiffies.)
         */
        unsigned long long sum_sched_runtime;

        /*
         * We don't bother to synchronize most readers of this at all,
         * because there is no reader checking a limit that actually needs
         * to get both rlim_cur and rlim_max atomically, and either one
         * alone is a single word that can safely be read normally.
         * getrlimit/setrlimit use task_lock(current->group_leader) to
         * protect this instead of the siglock, because they really
         * have no need to disable irqs.
         */
        struct rlimit rlim[RLIM_NLIMITS];

#ifdef CONFIG_BSD_PROCESS_ACCT
        struct pacct_struct pacct;      /* per-process accounting information */
#endif
#ifdef CONFIG_TASKSTATS
        struct taskstats *stats;
#endif
#ifdef CONFIG_AUDIT
        unsigned audit_tty;
        struct tty_audit_buf *tty_audit_buf;
#endif

        /*
         * Thread is the potential origin of an oom condition; kill first on
         * oom
         */
        bool oom_flag_origin;
        short oom_score_adj;            /* OOM kill score adjustment */
        short oom_score_adj_min;        /* OOM kill score adjustment min value.
                                         * Only settable by CAP_SYS_RESOURCE. */
        struct mm_struct *oom_mm;       /* recorded mm when the thread group got
                                         * killed by the oom killer */

        struct mutex cred_guard_mutex;  /* guard against foreign influences on
                                         * credential calculations
                                         * (notably. ptrace)
                                         * Deprecated do not use in new code.
                                         * Use exec_update_lock instead.
                                         */
        struct rw_semaphore exec_update_lock;   /* Held while task_struct is
                                                 * being updated during exec,
                                                 * and may have inconsistent
                                                 * permissions.
                                                 */
} __randomize_layout;
```

В структуре signal_struct в ядре Linux хранятся информация и настройки, связанные с обработкой сигналов.

2. syscall_info

Defined in 1 files as a struct:

include/linux/ptrace.h, line 15 *(as a struct)*

```
/* Add sp to seccomp_data, as seccomp is user API, we don't want to modify it */
struct syscall_info {
        __u64                   sp;
        struct seccomp_data     data;
};

 struct seccomp_data {
         int nr;
         __u32 arch;
         __u64 instruction_pointer;
         __u64 args[6];
 };
```

Структура syscall_info в ядре Linux предназначена для хранения информации о системных вызовах и их реализации. Она отвечает за связь между номерами системных вызовов и их соответствующими функциями в ядре.

Выполнение:

Исходный код разработанных модулей лежит по ссылке: Ivanio1/itmo-os (github.com)

Результаты:

```
ivan@ivan-VirtualBox:~/Desktop/itmo-os/lab2/code$ sudo ./userapp 3231

###################################

Opening Driver...
Writing data...


syscall_info for PID 3231:
        Stack pointer: 140724039143792
        Architecture: 3221225534
        Instruction pointer: 140432090900895
        The system call number: 7
        Syscall arguments:
                1. 140431782264256
                2. 5
                3. 4294967295
                4. 140431879014496
                5. 0
                6. 140432089774704

signal_struct_info for PID 3231:
        Nr threads = 108
        Group exit code = 0
        Notify count = 0
        Group stop count = 0
        Flags = 0

Closing Driver...

###################################
```

Вывод:

Во время выполнения лабораторной работы я углубился в работу ядра linux. Написал собственный модуль ядра и клиентское приложение, для работы с этим модулем, реализовав общение между ними с помощью ioctl.