Семинар 15: процессы

26 марта, 2020

Процессы

- Сам термин «процесс» довольно сложно определить
- POSIX: «*A process is an abstraction that represents an executing program. Multiple processes execute independently and have separate address spaces. Processes can create, interrupt, and terminate other processes, subject to security restrictions.*»
- В самом ядре Linux нет понятия «процесс», вместо этого оно оперирует «тасками»
- То, что в POSIX процесс, в Linux называется thread group
- Дальше мы всё таки будем говорить процессы :)
- Процессы объединяются в *группы процессов*
- Группы процессов объединяются в *сессии*

Аттрибуты процесса

- Сохранённых контекст процесса (регистры)
- Отображения памяти (VMA) и стек для ядра
- Файловые дескрипторы
- Current working directory (cwd) и текущий корень (`man 2 chroot`)
- umask
- PID, PPID, TID, TGID, PGID, SID
- Resource limits
- Priority
- Capabilities
- Namespaces

Аттрибуты процесса: PID, PPID, TGID, ...

- PID = process ID
- PPID = parent process ID
- TID = thread ID
- TGID = thread group ID
- PGID = process group ID
- SID = session ID
- `pid_t getpid()`, `pid_t getppid()`, `pid_t gettid()`
- `pid_t getpgid()`, `int setpgid(pid_t pid, pid_t pgid)`
- `pid_t setsid()`, `pid_t getsid(pid_t pid)`
- Всё это можно найти в `/proc/<pid>/status` или в `/proc/<pid>/stat`

Аттрибуты владельца процесса

- UID (user ID или real user ID) — ID владельца процесса, `void setuid(uid_t)`
- EUID (effective user ID) используется для проверок доступа, `void seteuid(uid_t)`
- SUID (saved user ID) используется, чтобы можно было временно понизить привилегии
- FSUID (file system user ID) обычно совпадает с EUID, но может быть отдельно изменён через `int setfsuid(uid_t fsuid)`
- Непривилегированный процесс может выставлять EUID равный только в SUID, UID или опять в EUID
- Также есть понятие setuid/setgid флагов (правильно они называются sticky-флагами), в отличие от обычных файлов, владельцем такого процесса станет *владелец файла*, а не текущий пользователь
- Есть аналогичные GID, EGID, SGID, FSGID

Приоритет и nice процессов

- В Linux есть понятие приоритета процесса
- Всего существует 140 различных приоритетов двух типов: real-time priroity и nice
- Приоритет реального времени от 1 до 99
- Приоритет для пользователей (через nice): от 100 до 139
- $priority = 20 + niceValue$
- Чем выше nice value, тем менее он приоритетен (более вежливый) и тем реже он выполняется
- По-умолчанию nice value равен 0

Лимиты ресурсов процессов

- Лимиты деляется на два типа: soft и hard
- Soft-лимит или текущий лимит – по нему вычисляются проверки
- Hard-лимит — максимальное значение soft-лимита
- CPU-время, если процесс его превысит ему ядро отошлёт **SIGXCPU**, если превысит hard limit, то **SIGKILL**
- Размер записываемых файлов: write будет возвращать **EFBIG**
- Размер записываемых coredump-файлов
- На максимальный размер виртуальной памяти, выделенной процессу
- Количество одновременных процессов пользователя
- Количество файловых дескрипторов
- И ещё кучу всего :)

Лимиты ресурсов процессов

```c
#include <sys/time.h>
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlim);
int setrlimit(int resource, const struct rlimit *rlim);

int prlimit(pid_t pid, int resource,
      const struct rlimit *new_limit,
            struct rlimit *old_limit);
```

Механизмы изоляции

- `man 7 namespaces` и `man 7 cgroups`
- Linux namespaces изолируют части отдельные процессов
- Существующие неймспейсы: user, network, mount, cgroup, pid и ut
- Механизм Linux для ограничения ресурсов процессов: в основном это более тонкие ограничения для потребления процессорного времени и памяти

Работа с процессами: fork

- В Linux выбран не самый обычный подход для запуска новых процессов
- Создать новый процесс можно только *скопировав* текущий с помощью `pid_t fork()`
- fork выйдет в двух процессах одновременно, но в однём вернёт PID ребёнка, а в другом — 0.
- Ребёнок будет полностью идентичен родителю, но файловые дескрипторы и адресное пространство будут *скопированы*
- Копирование адресного пространства довольно затратная штука, поэтому используется copy-on-write
- Реально копироваться страница будет только при первой записи в родителе или ребёнке

Dirty COW (CVE-2016-5195)

Работа с процессами: fork

```c
#include <unistd.h>

pid_t pid = fork();
if (pid == -1) {
    // fork сломался
} else if (pid == 0) {
    // ребёнок
    // текущий pid можно получить через getpid
} else {
    // родитель, pid содержит pid ребёнка
}
```

Работа с процессами: execve

- Создавать копии недостаточно, нужно уметь запускать произвольные файлы
- Для этого используется системный вызовов execve
- Он заменяет текущий процесс, процессом, созданным из указанного файла
- Это называется заменой образа процесса: заменяются только части адресного пространства
- Аттрибуты процесса сохраняются (в том числе: лимиты, неймспейсы, всевозможные *ID, ...), хотя бывают исключения
- Также в новом образе процесса останутся текущие файловые дескрипторы, не помеченные флагом `O_CLOEXEC`

Работа с процессами: execve

```c
#include <unistd.h>

extern char **environ;
int execl(const char *path, const char *arg0,
          ... /*, (char *)0 */);
int execle(const char *path, const char *arg0,
          ... /*, (char *)0, char *const envp[]*/);
int execlp(const char *file, const char *arg0,
          ... /*, (char *)0 */);
int execv(const char *path, char *const argv[]);
int execve(const char *path, char *const argv[],
          char *const envp[]);
int execvp(const char *file, char *const argv[]);
int fexecve(int fd, char *const argv[], char *const envp[]);
```

Работа с процессами: exit

- Служит для завершения текущего процесса
- `exit` vs `_exit`
- Чтобы завершить текущую thread group можно воспользоваться `exit_group`
- После вызова происходит закрытие всех открытых файловых дескрипторов, освобождение страниц в физической памяти
- Если у процесса были дети, то их родителем станет процесс с PID == 1.

Работа с процессами: wait

- Кроме запуска процессов нужно ещё и дожидаться их завершения
- Для этого используются системные вызовы семейства wait*
- Они дожидаются завершения процесса (конкретного или любого) и возвращают специальный *exit status*
- Обычно exit status содержит то, что передали в `exit`, но иногда процесс может завершиться не сам, а с помощью сигнала
- Для того, чтобы различать такие случаи, используются специальные макросы
- Зомби-процесс (Z) — такой процесс, который завершился, но wait в родителе ещё не собрал информацию из него

Работа с процессами: wait

```c
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/wait.h>

pid_t wait(int *wstatus);
pid_t waitpid(pid_t pid, int *wstatus, int options);

pid_t wait3(int *wstatus, int options,
            struct rusage *rusage);

pid_t wait4(pid_t pid, int *wstatus, int options,
            struct rusage *rusage);
```

Работа с процессами: макросы для status

```
WIFEXITED(status) // если процесс завершился сам
WEXITSTATUS(status) // exit status
WIFSIGNALED(status) // если процесс был завершён сигналом
WTERMSIG(status) // сигнал, завершивший процесс
WCOREDUMP(status) // отложил ли процесс coredump?
```

Gratias ago!