Процессы

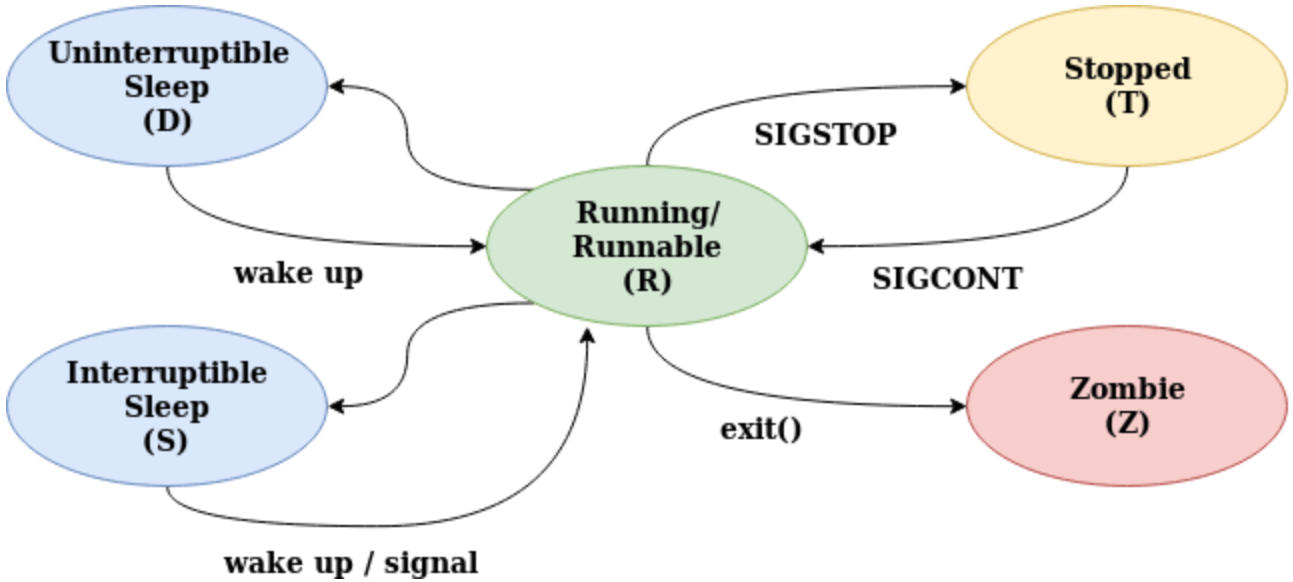
Процесс

- 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
- Процессы объединяются в *группы процессов*
- Группы процессов объединяются в *сессии*

Аттрибуты процессора

- Сохранённый контекст процессора (регистры)
- Виртуальная память (анонимные, private/shared, файловые)
- Файловые дескрипторы
- Current working directory (cwd)
- Текущий корень (`man 2 chroot`)
- umask
- PID, PPID, TID, TGID, PGID, SID
- Resource limits
- Priority
- Capabilities
- Namespaces

Состояния процессов



`fork`

- 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`
- Для оптимизации потребления памяти используется copy-on-write подход для копирования памяти

```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
}
```

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
- Также в новом образе процесса останутся незакрытые файловые дескрипторы, не помеченные флагом `O_CLOEXEC`

```c
#include <unistd.h>

extern char **environ;
int execl(const char *path, const char *arg0,
        ... /*, NULL */);
int execle(const char *path, const char *arg0,
        ... /*, NULL, char *const envp[]*/);
int execlp(const char *file, const char *arg0,
        ... /*, NULL */);
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[]);
```

`execve`: сохраняемые аттрибуты

- В отличие от `fork` сохраняет меньше аттрибутов
- Файловые дескрипторы (не помеченные флагом `O_CLOEXEC`)
- cwd и root

```c
#include <unistd.h>

pid_t pid = fork();
if (pid == -1) {
    // fork сломался
} else if (pid == 0) {
    char* argv[] = { "ls", "-lah", NULL };
    char* envp[] = { "FOO=bar", "XYZ=abc", NULL };
    execve("/usr/bin/ls", argv, envp);
    // если оказались здесь, то что-то пошло не так во время execve
} else {
    // родитель, pid содержит PID ребёнка
}
```

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

- PID = process ID
- PPID = parent process 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)`
- `/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 (sticky flags), в отличие от обычных файлов, EUID такого процесса будет выставлен как UID *владельца файла*, а не *текущий пользователь*
- Есть аналогичные GID, EGID, SGID, FSGID

Работа с процессами: exit

- Завершает текущий процесс с определённым *кодом возврата* (exit code)
- `exit` vs `_exit`
- Чтобы завершить текущую thread group можно воспользоваться `exit_group`
- exit закрывает все открытые файловые дескрипторы, освобождает выделенные страницы, etc
- Если у процесса были дети, то их родителем станет процесс с PID == 1
- После этого процесс становится *зомби-процессом*
- Ядро не хранит огромную структуру для него, а только его PID и exit code

Работа с процессами: wait

- Дожидается, пока процесс будет остановлен
- Для этого используются системные вызовы семейства `wait*`
- Они дожидаются завершения процесса (конкретного или любого) и возвращают специальный \emph{exit status}
- Обычно exit status содержит то, что передали в \mintinline{c}{exit}, но иногда процесс может завершиться не сам, а с помощью сигнала
- Для того, чтобы различать такие случаи, используются специальные макросы

```c
#include <sys/wait.h>

pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
pid_t wait3(int *stat_loc, int options, struct rusage *rusage);
pid_t wait4(pid_t pid, int *stat_loc, int options, struct rusage *rusage);
```

`wait4`

- `pid < -1`: ждёт любой дочерний процесс в группе процессов `-pid`
- `pid == -1`: ждёт любой дочерний процесс
- `pid == 0`: ждёт любой дочерний процесс в текущей группе процессов
- `pid > 0`: ждёт конкретного ребёнка

Макросы для `wait4`

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

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

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

struct rlimit {
    rlim_t rlim_cur;  /* Soft limit */
    rlim_t rlim_max;  /* Hard limit (ceiling for rlim_cur) */
};

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

Механизмы изоляции

- `man 7 namespaces` и `man 7 cgroups`
- Linux namespaces изолируют части отдельные процессов
- CGroups (control groups) обычно ограничивают потребляемое процессорное время и память
- Существующие неймспейсы: cgroup, IPC, network, mount, PID, time, и UTS

Вопросы?