ОПЕРАЦИОННЫЕ СИСТЕМЫ И СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ

Лекция № 03 — Иерархия процессов

Преподаватель: Поденок Леонид Петрович, 505а-5

+375 17 293 8039 (505а-5)
+375 17 320 7402 (ОИПИ НАНБ)
prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by

Кафедра ЭВМ, 2024

2023.02.29

Оглавление

Иерархия процессов

В операционной системе *NIX все процессы кроме одного, создающегося при старте операционной системы, могут быть порождены только какими-либо другими процессами.

В качестве процесса прародителя всех остальных процессов в разных *NIX-ообразных системах могут выступать процессы с номерами 1 или 0.

В операционной системе Linux таким родоначальником, существующим только при загрузке, является процесс **`kernel`** с идентификатором 0.

ps — снимок текущих процессов в системе

```
$ ps -e
    PID TTY          TIME CMD
      1 ?        00:00:08 systemd
      2 ?        00:00:00 kthreadd
      3 ?        00:00:00 pool_workqueue_release
      4 ?        00:00:00 kworker/R-rcu_g
      5 ?        00:00:00 kworker/R-rcu_p
...
 375280 ?        00:00:00 kworker/7:1-mm_percpu_wq
 375339 ?        00:00:00 kworker/5:1-events
 375431 ?        00:00:00 kworker/7:2-mm_percpu_wq
 375544 ?        00:00:00 kworker/0:0
 375584 ?        00:00:00 kworker/1:2
 375586 pts/19   00:00:00 ps
```

Все процессы в *NIX связаны отношениями процесс-родитель — процесс-потомок, образуя генеалогическое дерево процессов. Для сохранения целостности генеалогического дерева в ситуациях, когда процесс-родитель завершает свою работу до завершения выполнения процесса-потомка, идентификатор родительского процесса в данных ядра процесса- потомка (PPID - Parent Process IDentificator) изменяет свое значение на значение 1, соответствующее идентификатору процесса **init** (или **systemd**), время жизни которого определяет время функционирования операционной системы. Тем самым процесс **init** (или **systemd**) как бы усыновляет «осиротевшие» процессы.

Некоторые полагают, что логичнее было бы изменять PPID не на значение 1, а на значение идентификатора ближайшего существующего процесса-прародителя умершего процесса-родителя, но в UNIX такая схема реализована не была по причине того, что прародитель может ничего не знать об осиротевшем процессе и это все равно приведет к тому, что он «поднимется» к «Адаму».

```
$ ps -el --forest
F S UID  PID PPID C PRI  NI ADDR SZ WCHAN TTY     TIME CMD
1 S   0    2    0 0  80   0 -     0 -     ?   00:00:00 kthreadd
1 I   0    3    2 0  60 -20 -     0 -     ?   00:00:00  \_ rcu_gp
1 I   0    4    2 0  60 -20 -     0 -     ?   00:00:00  \_ rcu_par_gp
1 I   0    6    2 0  60 -20 -     0 -     ?   00:00:00  \_ kworker/0:0H-events_highpri
1 I   0    8    2 0  60 -20 -     0 -     ?   00:00:00  \_ mm_percpu_wq
1 S   0    9    2 0  80   0 -     0 -     ?   00:00:00  \_ rcu_tasks_kthre
1 S   0   10    2 0  80   0 -     0 -     ?   00:00:00  \_ rcu_tasks_rude_
1 S   0   11    2 0  80   0 -     0 -     ?   00:00:00  \_ rcu_tasks_trace
1 S   0   12    2 0  80   0 -     0 -     ?   00:00:10  \_ ksoftirqd/0
```

```
$ ps -A --forest
    PID TTY          TIME CMD
      2 ?        00:00:00 kthreadd
      3 ?        00:00:00  \_ rcu_gp
      4 ?        00:00:00  \_ rcu_par_gp
      6 ?        00:00:00  \_ kworker/0:0H-events_highpri
...
   8557 ?        00:00:00 lightdm
   8564 tty1     03:02:22  \_ Xorg
   8630 ?        00:00:00  \_ lightdm
   8667 ?        00:02:08      \_ xfce4-session
   8824 ?        00:00:00          \_ ssh-agent
   8941 ?        00:16:30          \_ xfce4-panel
   9020 ?        00:00:02          |   \_ panel-6-systray
....
   8987 ?        00:01:56          \_ xfdesktop
   8998 ?        01:04:30          \_ xfce4-terminal
   9114 pts/0    00:00:00          |   \_ bash
  15983 pts/0    00:00:00          |   |   \_ sudo
  15985 pts/0    00:00:00          |   |       \_ su
  15988 pts/0    00:00:00          |   |           \_ bash
  16027 pts/0    00:07:53          |   |               \_ mc
  16029 pts/5    00:00:00          |   |                   \_ bash
 776082 pts/5    00:00:00          |   |                       \_ mkfs.ext4
 776083 pts/5    00:00:40          |   |                           \_ badblocks
   9115 pts/1    00:00:00          |   \_ bash
```

Системные вызовы getppid() и getpid()

Данные ядра, находящиеся в контексте ядра процесса, не могут быть прочитаны процессом непосредственно. Для получения информации о данных, находящихся в контексте ядра, процесс должен совершить соответствующий системный вызов.

Значение идентификатора текущего процесса может быть получено с помощью системного вызова **`getpid(2)`**, а значение идентификатора родительского процесса для текущего процесса - с помощью системного вызова **`getppid(2)`**.

В операционной системе UNIX новый процесс может быть порожден единственным способом — с помощью системного вызова **`fork(2)`**. При этом вновь созданный процесс будет являться практически полной копией родительского процесса. У порожденного процесса по сравнению с родительским процессом изменяются значения следующих параметров:

- идентификатор процесса — PID;
- идентификатор родительского процесса — PPID;

Дополнительно к ним может измениться поведение порожденного процесса по отношению к некоторым сигналам.

В процессе выполнения системного вызова **`fork(2)`** порождается копия родительского процесса и возвращение из системного вызова будет происходить уже как в родительском, так и в порожденном процессах.

`fork(2)` — единственный системный вызов, который вызывается один раз, а при успешной работе возвращается два раза — один раз в процессе-родителе и один раз в процессе-потомке.

После выхода из системного вызова оба процесса продолжают выполнение регулярного пользовательского кода, следующего за системным вызовом.

fork — создает дочерний процесс

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

fork создает процесс-потомок, который отличается от родительского только значениями **PID** (идентификатор процесса) и **PPID** (идентификатор родительского процесса), а также тем фактом, что счетчики использования ресурсов установлены в **0**.

Блокировки файлов и сигналы, ожидающие обработки, не наследуются.

Под Linux **fork** реализован с помощью "копирования страниц при записи" (copy-on-write, COW), поэтому расходы на **fork** сводятся к копирования таблицы страниц родителя и созданию уникальной структуры, описывающей задачу.

Возвращаемое значение

При успешном завершении родителю возвращается **PID** процесса-потомка, а процессу-потомку возвращается **0**.

При неудаче родительскому процессу возвращается **-1**, процесс-потомок не создается, а значение **errno** устанавливается должным образом.

Ошибки

EAGAIN — **fork** не может выделить достаточно памяти для копирования таблиц страниц родителя и для выделения структуры описания процесса-потомка.

ENOMEM — **fork** не может выделить необходимые ресурсы ядра, потому что памяти слишком мало.

Соответствие стандартам

Системный вызов **fork** соответствует SVr4, SVID, POSIX, X/OPEN, BSD 4.3.

Программа с одинаковой работой родителя и потомка

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main( ) {

    pid_t pid, ppid, chpid;
    int a = 0;
    chpid = fork( );
// При успешном создании нового процесса с этого места псевдопараллельно
// начинают работать 2 процесса: старый и новый
// Перед выполнением следующего выражения a в обоих процессах равно 0
    a = a + 1;
// Узнаем идентификаторы текущего и родительского процесса в каждом из них
    pid  = getpid( );
    ppid = getppid( );
// Печатаем значения PID, PPID и вычисленное значение a в каждом из процессов
    printf("My pid = %d, my ppid = %d, result = %d\n", (int)pid, (int)ppid, a);
    return 0;
}
```

```
$ ./a.out
My pid = 777792, my ppid = 487976, result = 1
My pid = 777793, my ppid = 777792, result = 1
```

Завершение процесса. Функция exit()

Существует два способа корректного завершения процесса в программах, написанных на языке С.

1) процесс корректно завершается по достижении конца функции **`main()`** или при выполнении оператора **`return`** в функции **`main()`**;

2) второй способ применяется при необходимости завершить процесс в каком-либо другом месте программы.

Для этого применяется функция **`exit(3)`** из стандартной библиотеки функций для языка С.

При выполнении этой функции происходит сброс всех частично заполненных буферов ввода-вывода с закрытием соответствующих потоков, после чего инициируется системный вызов прекращения работы процесса и перевода его в состояние «закончил исполнение».

Возврата из функции в текущий процесс не происходит и функция ничего не возвращает.

Значение параметра функции **`exit(3)`** — кода завершения процесса — передается ядру операционной системы и может быть затем получено процессом, породившим завершившийся процесс.

На самом деле при достижении конца функции **`main()`** также неявно вызывается эта функция со значением параметра 0.

Если процесс завершает свою работу раньше, чем его родитель, и родитель явно не указал, что он *не хочет* получать информацию о статусе завершения порожденного процесса, то завершившийся процесс не исчезает из системы окончательно, а остается в состоянии «закончил исполнение» либо до завершения процесса-родителя, либо до того момента, когда родитель соблаговолит получить эту информацию. Процессы, находящиеся в состоянии «закончил исполнение», в операционной системе принято называть ***процессами-зомби*** (zombie, defunct).

```
$ ps -A –forest
...
 488604 pts/9    00:00:00           |   \_ bash
 488646 pts/9    00:00:02           |   |   \_ mc
 488648 pts/10   00:00:00           |   |       \_ bash
 529276 pts/9    00:00:00           |   |       \_ ssh <defunct>
 638985 pts/9    00:00:00           |   |       \_ ssh <defunct>
 638993 pts/9    00:00:00           |   |       \_ ssh <defunct>
 493395 pts/11   00:00:00           |   \_ bash
 494953 pts/11   00:00:14           |   |   \_ mc
 494955 pts/12   00:00:00           |   |       \_ bash
 525214 pts/11   00:00:00           |   |       \_ ssh
```

Пример

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/wait.h>

_Noreturn int child_foo(void); // (std 9899-2011) спец-ры inline и _Noreturn
int child_status;

int main() {

    fprintf(stdout, "Родительский процесс стартовал ...\n");
    pid_t pid = fork();
    if (pid == -1) {
        fprintf(stdout, "Ошибка, код ошибки - %d\n", errno);
    }
    if (pid == 0) { // дочерний процесс
        fprintf(stdout, "Это дочерний процесс. Вызываем child_foof()...\n");
        child_foo();
    }
    fprintf(stdout, "Родительский процесс продолжает выполнение\n");
    wait(&child_status);
    fprintf(stdout, "Дочерний процесс завершился с кодом завершения %d\n",
            WEXITSTATUS(child_status));
    exit(0);
}
```

```
_Noreturn int child_foo(void) {

    fprintf(stdout, "%s()\n", __func__);
    exit(123);
}
```

Выход

```
Родительский процесс стартовал ...
Родительский процесс продолжает выполнение
Это дочерний процесс. Вызываем child_foof()...
child_foo()
Дочерний процесс завершился с кодом завершения 123
```

Параметры функции main() в языке C.

Переменные среды и аргументы командной строки

У функции **`main()`** в языке программирования C в общем случае существует ***три параметра***, которые могут быть переданы ей операционной системой.

Стандарт ISO/IEC 9899:2011 в 5.1.2.2.1 Program startup приводит прототип функции **`main()`**

```
int main(int argc, char *argv[]) { /* ... */ }
```

ISO/IEC 9899:2011 Annex J (informative) Portability issues

J.5.1 Environment arguments (Аргументы среды)

*В размещающей среде функция **`main()`** получает третий аргумент, **`char *envp[]`**, который является массивом указателей на **`char`**, завершающимся нулем, каждый из которых указывает на строку, которая предоставляет информацию о среде, в которой выполняется программа.*

Полный прототип функции **`main()`** выглядит следующим образом:

```
int main(int argc, char *argv[], char *envp[]);
```

Первые два параметра при запуске программы на исполнение командной строкой позволяют узнать полное содержание командной строки.

Вся командная строка рассматривается как ***набор слов, разделенных пробелами***.

Через параметр **`argc`** передается количество слов в командной строке, которой была запущена программа.

Параметр **argv** является массивом указателей на отдельные слова. Так, например, если программа была запущена командой

```
$ prog 12 abcd "a b c d"
```

то значение параметра **argc** будет равно 4,
argv[0] будет указывать на имя программы — первое слово **"prog"**,
argv[1] — на слово **"12"**,
argv[2] - на слово **"abcd"**.
argv[3] - на слово **"a b c d"**.

Имя программы всегда присутствует на первом месте в командной строке, поэтому **argc** всегда больше 0, а **argv[0]** всегда указывает на имя запущенной программы.

Анализируя в программе содержимое командной строки, мы можем предусмотреть ее различное поведение в зависимости от слов следующих за именем программы.

wp --> office // запускается вордпроцессор
ss --> office // запускаются электронные таблицы

Третий параметр — **envp** — является массивом указателей на параметры окружающей среды процесса.

Начальные параметры окружающей среды процесса задаются в специальных конфигурационных файлах для каждого пользователя и устанавливаются при входе пользователя в систему. В последующем они могут быть изменены с помощью специальных команд операционной системы.

Каждый параметр имеет вид:

```
переменная=строка
```

Такие переменные используются для изменения долгосрочного поведения процессов, в отличие от аргументов командной строки.

Размер массива аргументов командной строки в функции **main()** мы получали в качестве ее параметра. Так как для массива ссылок на параметры окружающей среды такого параметра нет, то его размер определяется другим способом. Последний элемент этого массива содержит указатель **NULL**.

```
$ env
SHELL=/bin/bash
HOSTNAME=z-book
HOME=/home/podenok
LANG=ru_RU.utf8
TERM=xterm-256color
USER=podenok
PATH=/home/podenok/.local/bin:/home/podenok/bin:/usr/lib64/qt-3.3/bin:/usr/
lib64/ccache:/usr/local/bin:/usr/bin:/bin:/usr/local/sbin:/usr/sbin
GDMSESSION=xfce
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
MAIL=/var/spool/mail/podenok
```

execve — выполнить программу

extern

```
#include <unistd.h>

int execve(const char *filename, //
           char *const argv[],   // --> argv
           char *const envp[]);  // --> envp
```

execve() выполняет программу, заданную параметром **filename**.

Программа должна быть или двоичным исполняемым файлом, или скриптом, начинающимся со строки вида

```
"#! интерпретатор [аргументы]"
```

В последнем случае интерпретатор — это правильный путь к исполняемому файлу, который не является скриптом; этот файл будет выполнен как **интерпретатор [arg] filename**.

argv — это массив строк, аргументов новой программы.

envp — это массив строк в формате **key=value**, которые передаются новой программе в качестве окружения (*environment*).

Как **argv**, так и **envp** завершаются нулевым указателем.

К массиву аргументов и к окружению можно обратиться из функции **main()**, которая объявлена как

```
int main(int argc, char *argv[], char *envp[]);
```

```
#include <unistd.h>

int execve(const char *filename,
           char *const argv[],  // аргументы и
           char *const envp[]); // параметры среды исполнения,
                                // передаваемые процессу (программе) при запуске
```

С другой стороны, у нас есть функция **main()**, с которой программа (процесс) начинает исполнение:

```
int main(int argc,      // количество переданных параметров
         char *argv[],  // массив указателей на символы (строки параметров)
         char *envp[]); // массив указателей на символы (строки среды)
```

execve() не возвращает управление при успешном выполнении, а код, данные, bss и стек вызвавшего процесса перезаписываются кодом, данными и стеком загруженной программы.

Новая программа также наследует от вызвавшего процесса его идентификатор и открытые файловые дескрипторы, на которых не было флага '*закрыть-при-завершении*' (close-on-exec, COE).

Сигналы, ожидающие обработки, удаляются.

Переопределённые обработчики сигналов возвращаются в значение по умолчанию.

Обработчик сигнала **SIGCHLD** (когда установлен в **SIG_IGN**) может быть сброшен или не сброшен в **SIG_DFL**.

Если текущая программа выполнялась под управлением **ptrace**, то после успешного **execve()** ей посылается сигнал **SIGTRAP**.

Если на файле программы **filename** установлен setuid-бит, то фактический идентификатор пользователя вызывавшего процесса меняется на идентификатор владельца файла программы.

Точно так же, если на файле программы установлен setgid-бит, то фактический идентификатор группы устанавливается в группу файла программы.

Если исполняемый файл является динамически-скомпонованным файлом в формате **a.out**, содержащим заглушки для вызова совместно используемых библиотек, то в начале выполнения этого файла вызывается динамический компоновщик **ld.so**(8), который загружает библиотеки и компонует их с исполняемым файлом.

Если исполняемый файл является динамически-скомпонованным файлом в формате **ELF**, то для загрузки разделяемых библиотек используется интерпретатор, указанный в сегменте **PT_INTERP**. Обычно это **/lib/ld-linux.so.1** для программ, скомпилированных под Linux **libc** версии 5, или же **/lib/ld-linux.so.2** для программ, скомпилированных под GNU **libc** версии 2.

Возвращаемое значение

При успешном завершении **execve( )** не возвращает управление, при ошибке возвращается **-1**, а значение **errno** устанавливается должным образом.

Коды ошибок

EACCES — интерпретатор файла или скрипта не является обычным файлом.
EACCES — нет прав на выполнение файла, скрипта или ELF-интерпретатора.
EACCES — файловая система смонтирована с флагом **noexec**.

EPERM — файловая система смонтирована с флагом **nosuid**, пользователь не является суперпользователем, а на файле установлен бит **setuid** или **setgid**.

EPERM — процесс работает под отладчиком, пользователь не является суперпользователем, а на файле установлен бит **setuid** или **setgid**.

E2BIG — список аргументов слишком велик.

ENOEXEC — исполняемый файл в неизвестном формате, для другой архитектуры, или же встречены какие-то ошибки, препятствующие его выполнению.

EFAULT — **filename** указывает за пределы доступного адресного пространства.

ENAMETOOLONG — **filename** слишком длинное.

ENOENT — файл **filename**, или интерпретатор скрипта или ELF-файла не существует, или же не найдена разделяемая библиотека, требуемая файлу или интерпретатору.

ENOMEM — недостаточно памяти в ядре.

ENOTDIR — компонент пути **filename**, или интерпретатору скрипта или ELF-интерпретатору не является каталогом.

EACCES — нет прав на поиск в одном из каталогов по пути к **filename**, или имени интерпретатора скрипта или ELF-интерпретатора.

ELOOP — слишком много символьных ссылок встречено при поиске **filename**, или интерпретатора скрипта или ELF-интерпретатора.

ETXTBSY — исполняемый файл открыт для записи одним или более процессами.

EIO — произошла ошибка ввода-вывода.

ENFILE — достигнут системный лимит на общее количество открытых файлов.

EMFILE — процесс уже открыл максимально доступное количество открытых файлов.

EINVAL — исполняемый файл в формате ELF содержит более одного сегмента **PT_INTERP** (то есть, в нем указано более одного интерпретатора).

EISDIR — ELF-интерпретатор является каталогом.

ELIBBAD — ELF-интерпретатор имеет неизвестный формат.

Соответствие стандартам

SVr4, SVID, X/OPEN, BSD 4.3. POSIX не документирует поведение, связанное с **#!**, но в остальном совершенно совместимо. SVr4 документирует дополнительные коды ошибок **EAGAIN**, **EINTR**, **ELIBACC**, **ENOLINK**, **EMULTIHOP**; POSIX не документирует коды ошибок **ETXTBSY**, **EPERM**, **EFAULT**, **ELOOP**, **EIO**, **ENFILE**, **EMFILE**, **EINVAL**, **EISDIR** и **ELIBBAD**.

Замечания

SUID и **SGID** процессы не могут быть оттрассированы **ptrace()**.

Linux игнорирует **SUID** и **SGID** биты на скриптах.

Результат монтирования файловой системы с опцией **nosuid** различается в зависимости от версий ядра Linux: некоторые ядра будут отвергать выполнение **SUID/SGID** программ, когда это должно дать пользователю те возможности, которыми он уже не обладает (и возвращать **EPERM**), некоторые ядра будут просто игнорировать **SUID/SGID** биты, но успешно производить запуск программы.

Первая строка (строка с **#!**) исполняемого скрипта не может быть длиннее 127 символов.

Пример

parent.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/wait.h>

int main() {

    int child_status;
    char *args[] = {"pr0gramm", "parm_1", (char*)0};
    pid_t pid = fork();

    if (pid == -1) {
        fprintf(stdout, "Error occured, error code - %d\n", errno);
        exit(errno);
    }
    if (pid == 0) {      // Ветка дочернего процесса
        fprintf(stdout, "Child process created. Please, wait...\n");
        execve("./child", args, NULL);
    }
    wait(&child_status); // Ветка родительского процесса
    fprintf(stdout, "Child process have ended with %d exit status\n", child_status);
    exit(0);
}
```

child.c

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {

    fprintf(stdout, "Child process begins...\n");
    for (int i = 0; i < argc; i++) {
        fprintf(stdout, "%s\n", argv[i]);
    }
    exit(0);
}
```

makefile

```
CC = gcc
CFLAGS = -W -Wall -Wextra -std=c11
.PHONY: clean

all: parent child
parent: parent.c makefile
        $(CC) $(CFLAGS) parent.c -o parent
child: child.c makefile
        $(CC) $(CFLAGS) child.c -o child
clean:
        rm parent child
```

Компиляция и сборка

```
$ make
gcc -W -Wall -Wextra -std=c11 parent.c -o parent
gcc -W -Wall -Wextra -std=c11 child.c -o child
$ls -1
child
child.c
makefile
parent
parent.c
```

Запуск

```
$ ./parent
Parent process begins...
Child process created. Please, wait...
Child process begins...
pr0gramm
parm_1
Child process have ended with 0 exit status
```

`environ` — среда пользователя

```
extern char **environ; // массив указателей
```

Переменная **environ** указывает на массив указателей на строки, называемый «окружением». Последний указатель в этом массиве имеет значение **NULL**.

Этот массив строк предоставляется процессу вызовом **execve(2)** при запуске новой программы по умолчанию (никто не мешает его заполнить чем угодно, как и argv).

Когда дочерний процесс создается с помощью **fork(2)**, он наследует копию своего родительского окружения.

По соглашению строки в окружении имеют форму **"имя=значение"**.

Имя чувствительно к регистру и не может содержать символ **"="**.

Значение может быть любым, что может быть представлено в виде строки.

Имя и значение не могут содержать встроенный нулевой байт (**'\0'**), так как предполагается, что он завершает строку.

Переменные среды могут быть помещены в среду оболочки командой **export** в **sh(1)** или командой **setenv**, если используется **csh(1)**.

Начальная среда оболочки заполняется различными способами, например определениями из **/etc/environment**.

Кроме того, различные сценарии инициализации оболочки, такие как общесистемный сценарий **/etc/profile** и сценарий инициализации для каждого пользователя, могут включать команды, добавляющие переменные в среду оболочки.

Для создания определений переменной среды в рамках процесса, выполняющего команду, оболочки в стиле Bourne (bash) поддерживают синтаксис:

```
NAME=value command
```

Команде может предшествовать несколько определений переменных, разделенных пробелом.

```
$ LC_MESSAGES=C /opt/slickeditor/bin/vse
$ rpm -qa | LANG=C sort
```

Аргументы также могут быть помещены в окружение при вызове **exec(3)**.

Семейство функций **exec()** – обертки функции **execve(2)**.

Программа на C может манипулировать ***своим*** окружением с помощью функций из **<stdlib.h>** **getenv(3)**, **putenv(3)**, **setenv(3)** и **unsetenv(3)**.

Итак, три способа передачи информации о среде выполнения:

1) из переданного в функцию **main()**;

2) из внешней переменной **environ**;

3) с помощью **getenv()**.

Переменные окружения, которые обычно встречаются в системе

Список неполный и включает только общие переменные, которые обычные пользователи видят в своей повседневной жизни.

Переменные среды, специфичные для конкретной программы или библиотечной функции, задокументированы в разделе **ENVIRONMENT** соответствующей страницы руководства.

USER — имя вошедшего в систему пользователя (используется некоторыми программами, производными от BSD).

LOGNAME — имя вошедшего в систему пользователя (используется некоторыми программами, производными от System-V). Устанавливается при входе в систему.

HOME — «домашний» каталог пользователя. Устанавливается при входе в систему.

LANG — имя, используемое для категорий локалей, если оно не переопределено переменной **LC_ALL** или более конкретными переменными среды, такими как **LC_COLLATE**, **LC_CTYPE**, **LC_MESSAGES**, **LC_MONETARY**, **LC_NUMERIC** и **LC_TIME** (информацию о переменных среды **LC_*** можно найти **locale(7)**).

PATH — последовательность префиксов каталогов, которую **sh(1)** и многие другие программы используют при поиске исполняемого файла, указанного как простое имя файла (т. е. путь, не содержащий в начале `'/'`). Префиксы разделяются двоеточиями `':'`.

Список префиксов просматривается от начала до конца путем проверки имени пути, образованного объединением префикса, косой черты и имени файла, до тех пор, пока не будет найден файл с разрешением на выполнение.

В качестве устаревшей функции префикс нулевой длины (указанный как два соседних двоеточия или начальное или завершающее двоеточие) интерпретируется как текущий рабочий каталог. Однако использование этой функции устарело, и в POSIX отмечено, что соответствующее приложение для указания текущего рабочего каталога должно использовать явное имя пути (например, '.').

Аналогично тому, как используется **PATH**, некоторыми оболочками для поиска цели команды изменения каталога (cd <target>) используется **CDPATH**.

Для поиска справочных страниц man(1) использует **MANPATH**, и так далее.

PWD — текущий рабочий каталог. Устанавливается некоторыми оболочками.

SHELL — абсолютный путь к оболочке, используемой пользователем при входе. Устанавливается при входе в систему.

TERM — тип терминала, для которого должен быть подготовлен вывод.

PAGER — предпочитаемая пользователем утилита для отображения текстовых файлов. Это может быть любая строка, допустимая в качестве операнда командной строки для команды **sh -c**. Если **PAGER** имеет значение **null** или не установлен, то приложения, запускающие пейджер, по умолчанию будут использовать программу, такую как **less(1)** или **more(1)**.

EDITOR/VISUAL — предпочитаемая пользователем утилита для редактирования текстовых файлов. Это может быть любая строка, допустимая в качестве операнда командной строки для команды **sh -c**.

Наличие или значение определенных переменных среды влияет на поведение многих программ и библиотечных процедур:

Переменные **LANG**, **LANGUAGE**, **NLSPATH**, **LOCPATH**, **LC_ALL**, **LC_MESSAGES** и т. д. влияют на обработку локали (**catopen(3)**, **gettext(3)** и **locale(7)**).

TMPDIR влияет на префикс пути к имени, созданному **tempnam(3)** и другими подпрограммами, а также на временный каталог, используемый **sort(1)** и другими программами.

LD_LIBRARY_PATH, **LD_PRELOAD** и другие переменные **LD_*** влияют на поведение динамического загрузчика/компоновщика (**ld.so(8)**).

```
export LD_LIBRARY_PATH=$HOME/lib64
```

POSIXLY_CORRECT заставляет определенные программы и библиотеки следовать предписаниям POSIX.

На поведение **malloc(3)** влияют переменные **MALLOC_***.

TZ и **TZDIR** предоставляют информацию о часовом поясе, используемую **tzset(3)** и через нее такими функциями, как **ctime(3)**, **localtime(3)**, **mktime(3)**, **strftime(3)**, **tzselect(8)**.

PRINTER или **LPDEST** могут указать желаемый принтер для использования **lpr(1)**.

Переменная **HOSTALIASES** задает имя файла, содержащего псевдонимы, которые будут использоваться с **gethostbyname(3)**.

```
$ alias
alias egrep='egrep --color=auto'
alias fgrep='fgrep --color=auto'
alias grep='grep --color=auto'
alias l.='ls -d .* --color=auto'
alias ll='ls -l --color=auto'
alias ls='ls --color=auto'
alias mc='. /usr/libexec/mc/mc-wrapper.sh'
alias mingw32-env='eval `rpm --eval %{mingw32_env}`'
alias mingw64-env='eval `rpm --eval %{mingw64_env}`'
alias which='(alias; declare -f) | /usr/bin/which --tty-only --read-alias --read-
functions --show-tilde --show-dot'
alias xzegrep='xzegrep --color=auto'
alias xzfgrep='xzfgrep --color=auto'
alias xzgrep='xzgrep --color=auto'
alias zegrep='zegrep --color=auto'
alias zfgrep='zfgrep --color=auto'
alias zgrep='zgrep –color=auto'
```

TERMCAP дает информацию о том, как обращаться к данному терминалу (или предоставляет имя файла, содержащего такую информацию).

COLUMNS и **LINES** сообщают приложениям размер окна, возможно, переопределяя фактический размер.

Использование среды может иметь угрозу безопасности. Многие системные команды были обмануты пользователем, указанием необычных значений для **`LD_LIBRARY_PATH`**.

Существует также риск загрязнения пространства имен. Такие программы, как **`make`** и **`autoconf`**, позволяют переопределять имена утилит по умолчанию с помощью переменных с аналогичными именами во всех регистрах. Например, CC используется для выбора нужного компилятора C (и аналогично MAKE, AR, AS, FC, LD, LEX, RM, YACC и т. д.). Однако в некоторых традиционных случаях такая переменная среды дает опции для программы вместо имени пути. Такое использование считается ошибочным, и его следует избегать в новых программах.

wait, waitpid — ожидает завершения процесса

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

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

Функция **wait** приостанавливает выполнение текущего процесса до тех пор, пока какой нибудь из дочерних процессов не завершится, или до появления сигнала, который либо завершает текущий процесс, либо требует вызвать функцию-обработчик.

Системные ресурсы, связанные с дочерним процессом, освобождаются.

Функция **waitpid** приостанавливает выполнение текущего процесса до тех пор, пока дочерний процесс, указанный в параметре **pid**, не завершит выполнение, или пока не появится сигнал, который либо завершает текущий процесс, либо требует вызвать функцию-обработчик.

Если указанный дочерний процесс к моменту вызова функции уже завершился (так называемый «*зомби*» — "zombie"), то функция немедленно возвращает управление.

Системные ресурсы, связанные с дочерним процессом, освобождаются.

Параметр **pid** может принимать несколько значений:

< -1 — означает, что нужно ждать любого дочернего процесса, идентификатор группы процессов которого равен абсолютному значению **pid**.

-1 — означает ожидание любого дочернего процесса; функция **wait** ведет себя точно так же.

0 — означает ожидание любого дочернего процесса, идентификатор группы процессов которого равен идентификатору текущего процесса.

> 0 — означает ожидание дочернего процесса, чей идентификатор равен **pid**.

options

Значение **options** создается путем логического сложения нескольких следующих констант:

WNOHANG — означает немедленное возвращение управления, если ни один дочерний процесс не завершил выполнение.

WUNTRACED — означает возврат управления и для остановленных (но не отслеживаемых) дочерних процессов, о статусе которых еще не было сообщено.

Статус для отслеживаемых остановленных подпроцессов обеспечивается без этой опции.

Возвращаемые значения

Возвращает идентификатор дочернего процесса, который завершил выполнение, или ноль, если использовался **WNOHANG** и ни один дочерний процесс пока еще недоступен, или **-1** в случае ошибки (в этом случае переменной **errno** присваивается соответствующее значение).

status

Если **status** не равен **NULL**, то функции **wait** и **waitpid** сохраняют информацию о статусе в переменной, на которую указывает **status**.

Состояние **status** можно проверить с помощью макросов, которые принимают в качестве аргумента буфер (типа **int**), а не указатель на буфер!:

WIFEXITED(status) — не равно нулю, если дочерний процесс успешно завершился.

WEXITSTATUS(status) — возвращает восемь младших битов значения, которое вернул завершившийся дочерний процесс.

Эти биты могли быть установлены в аргументе функции **exit()** или в аргументе оператора **return** функции **main()**. Этот макрос можно использовать, только если **WIFEXITED** вернул ненулевое значение.

WIFSIGNALED(status) — возвращает истинное значение, если дочерний процесс завершился из-за необработанного сигнала.

WTERMSIG(status) — возвращает номер сигнала, который привел к завершению дочернего процесса. Этот макрос можно использовать, только если **WIFSIGNALED** вернул ненулевое значение.

WIFSTOPPED(status) — возвращает истинное значение, если дочерний процесс, из-за которого функция вернула управление, в настоящий момент остановлен; это возможно, только если использовался флаг **WUNTRACED** или когда подпроцесс отслеживается (см. **ptrace(2)**).

WSTOPSIG(status) — возвращает номер сигнала, из-за которого дочерний процесс был остановлен. Этот макрос можно использовать, только если **WIFSTOPPED** вернул ненулевое значение.

Ошибки

ECHILD — процесс, указанный в **pid**, не существует или не является дочерним процессом текущего процесса. (Это может случиться и с собственным дочерним процессом, если обработчик сигнала **SIGCHLD** установлен **в SIG_IGN**. См. главу ЗАМЕЧАНИЯ по поводу многозадачности процессов.)

EINVAL — аргумент **options** неверен.

EINTR — использовался флаг **WNOHANG**, и был получен необработанный сигнал или **SIGCHLD**. Стандарт Single Unix Specification описывает флаг **SA_NOCLDWAIT** (не поддерживается в Linux), если он установлен или обработчик сигнала **SIGCHLD** устанавливается в **SIG_IGN**, то завершившиеся дочерние процессы не становятся зомби, а вызов **wait()** или **waitpid()** блокируется, пока все дочерние процессы не завершатся, а затем устанавливает переменную errno равной **ECHILD**.

Стандарт POSIX оставляет неопределенным поведение при установке **SIGCHLD** в **SIG_IGN**. поздние стандарты, включая SUSv2 и POSIX 1003.1-2001, определяют поведение, только что описанное как опция совместимости с XSI.

Linux не следует второму варианту — если вызов **wait()** или **waitpid()** сделан в то время, когда **SIGCHLD** игнорируется, то вызов ведет себя, как если бы **SIGCHLD** не игнорировался, то есть вызов блокирует до завершения работы следующего подпроцесса и возврата идентификатора процесса PID и статуса этого подпроцесса.

Замечания по linux

В ядре Linux задачи, управляемые ядром, по внутреннему устройству не отличаются от процесса. Задача (*thread*) — это простой процесс, который создан специфичным для Linux системным вызовом **clone(2)**; другие процедуры, типа портируемой **pthread_create(3)**, также реализованы с помощью **clone(2)**.

До Linux 2.4, задачи были частным случаем процесса, и последовательность одной группы задач не могла ожидать дочерний процесс или другую группу задач, даже если воследняя принадлежит к этой-же группе задач. Однако, POSIX предопределяет такие особенности, и с Linux 2.4 группа задач может, и по умолчанию будет ожидать дочерний процесс другой группы задач в этой же группе задач.

Следующие специфичные для Linux параметры существуют для использования с дочерними процессами и **clone(2)**.

__WCLONE — только ожидает дочерние процессы. Если отменяется, то ожидает только дочерних неклонированных процессов. («клонированным» дочерним процессом является подпроцесс, не отправляющий сигналов, или выдающий сигналы, отличающиеся от **SIGCHLD** своему родителю до завершения.) Эта опция игнорируется, если определено **__WALL**.

__WALL — (с Linux 2.4) Ожидает все дочерние процессы, независимо от их типа («клон» или «не-клон»).

__WNOTHREAD — (С Linux 2.4) Не ожидать дочерние процессы или остальные задачи в идентичной группе задач. Было параметром по умолчанию до Linux 2.4.

Взаимодействие со средой (размещающее окружение)

abort — аварийное завершение программы

```
#include <stdlib.h>
_Noreturn void abort(void); // C11
```

Если сигнал **SIGABRT** не перехватывается и обработчик сигнала не возвращается (**longjmp(3)**) функция **abort()** сначала разблокирует сигнал **SIGABRT**, а затем инициирует этот сигнал для вызывающего процесса (как если бы была вызвана функция **raise(3)**). Это приводит к аварийному завершению процесса,

Если сигнал **SIGABRT** игнорируется или перехватывается обработчиком, который возвращает управление, функция **abort()** все равно завершит процесс. Он делает это, восстанавливая для **SIGABRT** поведение по умолчанию, и генерируя сигнал во второй раз.

Сбрасываются ли открытые потоки с незаписанными буферизованными данными, закрываются ли открытые потоки или удаляются ли временные файлы, определяется реализацией.

Возвращает

Управление вызывающей стороне не возвращает.

Особенности POSIX-реализации

Соответствует в целом стандарту ISO C.

Функция **abort()** вызывает аварийное завершение процесса в том случае, если не перехватывается сигнал **SIGABRT** и обработчик сигнала не возвращает управление.

Обработка аномального завершения должна включать действия по умолчанию, определенные для **SIGABRT**, и может включать попытку применить **fclose()** ко всем открытым потокам.

Сигнал **SIGABRT** отправляется вызывающему процессу как бы с помощью функции **raise()** с аргументом **SIGABRT**.

Статус для **wait()**, **waitid()** или **waitpid()**, сформированный с помощью **abort()**, является статусом процесса, завершенного сигналом **SIGABRT**.

Функция **abort()** отменяет блокировку или игнорирование сигнала **SIGABRT**.

Перехват сигнала предназначен для того, чтобы предоставить разработчику приложения переносимое средство для обработки прерывания, свободное от возможных помех со стороны каких-либо функций, предоставляемых реализацией.

atexit — регистрирует функцию, вызываемую при завершении

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

Функция **atexit()** регистрирует функцию, на которую указывает **func**, для вызова без аргументов при нормальном завершении программы.

Регистрации функции **atexit()** отличаются от регистраций **at_quick_exit()**, поэтому приложениям может потребоваться вызывать обе функции регистрации с одним и тем же аргументом.

Реализация обычно поддерживает регистрацию не менее 32 функций.

Возвращает

Функция **atexit()** возвращает ноль, если регистрация прошла успешно, и ненулевое значение, если она не удалась.

at_quick_exit — регистрирует функцию, вызываемую при завершении

```
#include <stdlib.h>
int at_quick_exit(void (*func)(void));
```

Функция **at_quick_exit()** регистрирует функцию, на которую указывает **func**, для вызова без аргументов в случае вызова **quick_exit()**.

Регистрация функции **at_quick_exit()** отличается от регистрации **atexit()**, поэтому приложениям может потребоваться вызывать обе функции регистрации с одним и тем же аргументом.

Реализация обычно поддерживает регистрацию не менее 32 функций.

Возвращает

Функция **at_quick_exit()** возвращает ноль, если регистрация прошла успешно, и ненулевое значение, если она не удалась.

exit — вызывает обычное завершение программы

```
#include <stdlib.h>
_Noreturn void exit(int status);
```

Функция **exit()** вызывает обычное завершение программы.

Функции, зарегистрированные функцией **at_quick_exit()**, не вызываются.

1) Сначала вызываются все функции, зарегистрированные функцией **atexit()**, в порядке, обратном их регистрации.

Каждая функция вызывается столько раз, сколько она была зарегистрирована, и в правильном порядке по отношению к другим зарегистрированным функциям.

2) Затем сбрасываются все открытые потоки с незаписанными буферизованными данными, все открытые потоки закрываются, а все файлы, созданные функцией **tmpfile()**, удаляются.

3) Управление возвращается в хост-среду.

Если значение **status** равно нулю или **EXIT_SUCCESS**, возвращается определяемая реализацией форма успешного завершения состояния.

Если значением статуса является **EXIT_FAILURE**, возвращается определенная реализацией форма статуса неудачного завершения.

В противном случае возвращаемый статус определяется реализацией.

Не возвращает управление вызывающей стороне – _Noreturn.

Если программа вызывает функцию выхода более одного раза или вызывает функцию **quick_exit()** в дополнение к функции выхода, поведение не определено.

_Exit — нормальное завершение программы

```
#include <stdlib.h>
_Noreturn void _Exit(int status);
```

Функция **_Exit()** вызывает нормальное завершение программы и возврат управления в хост-среду.

Никаких функций, зарегистрированных функцией **atexit()**, функцией **at_quick_exit()** или обработчиков сигналов, зарегистрированных функцией **signal()**, не вызывается.

Статус, возвращаемый в хост-среду, определяется так же, как и для функции **exit()**.

Судьба открытых потоков с незаписанными буферизованными данными (сбрасываются/не сбрасываются, закрываются/не закрываются), удаляются ли временные файлы, определяется реализацией.

Управление вызывающей стороне не возвращает.

quick_exit — нормальное завершение программы

```
#include <stdlib.h>
_Noreturn void quick_exit(int status);
```

Функция **quick_exit()** вызывает нормальное завершение программы.

Никакие функции, зарегистрированные функцией **atexit()**, или обработчики сигналов, зарегистрированные функцией **signal()**, не вызываются.

Если во время выполнения функции **quick_exit()** возникает сигнал, поведение также не определено.

Сначала функция **quick_exit()** вызывает все функции, зарегистрированные функцией **at_quick_exit()**, в порядке, обратном их регистрации.

Каждая функция вызывается столько раз, сколько она была зарегистрирована, и в правильном порядке по отношению к другим зарегистрированным функциям.

После этого посредством вызова функции **_Exit(status)** возвращается управление в хост-среду.

Если программа вызывает функцию **quick_exit()** более одного раза или вызывает функцию **exit()** в дополнение к функции **quick_exit()**, поведение не определено.

Не возвращает управление вызывающей стороне.

_exit — функция, завершающая работу программы

Помимо нескольких функций **exit** — **exit(3)** из стандартной библиотеки C существует более низкоуровневая **exit(2)**.

```
#include <unistd.h>
#include <stdlib.h>
```

```
void _exit(int status);
```

_exit "немедленно" завершает работу программы.

Все дескрипторы файлов, принадлежащие процессу, закрываются, все его дочерние процессы начинают управляться процессом 1 (init или systemd), а родительскому процессу посылается сигнал **SIGCHLD**.

Значение **status** возвращается родительскому процессу как статус завершаемого процесса, который он может быть получен в родительском процессе с помощью одной из функций семейства **wait**.

Возвращаемые значения

Эти функции никогда не возвращают управление вызвавшей их программе.

Соответствие стандартам

SVr4, SVID, POSIX, X/OPEN, BSD 4.3. Функция **_Exit()** была представлена C99.

Замечания

Для рассмотрения эффектов завершения работы, передачу статуса выхода, зомби-процессов, сигналов и т.п., следует смотреть документацию по **exit(3)**.

Функция **_exit** аналогична **exit()**, но не вызывает никаких функций, зарегистрированных с функцией C **atexit**, а также не вызывает никаких зарегистрированных обработчиков сигналов.

Будет ли выполняться сброс стандартных буферов ввода-вывода и удаление временных файлов, созданных **tmpfile(3)**, зависит от реализации.

С другой стороны, **_exit** закрывает открытые дескрипторы файлов, а это может привести к неопределенной задержке для завершения вывода данных.

Если задержка нежелательна, то может быть полезным перед вызовом **_exit()** вызывать функции типа **tcflush()**. Будет ли завершен ввод-вывод, а также какие именно операции ввода-вывода будут завершены при вызове **_exit()**, зависит от реализации.

	abort()	**_Exit()**	**exit(3)**	**quick_exit(3)**	**exit(2)**
Вызов функций **atexit(3)**	нет	нет	**да**	нет	нет
Вызов функций **at_quick_exit(3)**	нет	нет	нет	**да**	нет
Вызов обработчиков сигналов	нет	нет	нет	нет	нет
Сбрасываются все открытые потоки	?	?	**да**	**да**	n/a
Открытые дескрипторы закрываются	n/a	n/a	n/a	n/a	**да**
Открытые потоки закрываются	?	?	**да**	**да**	n/a
Все файлы, созданные **tmpfile()** удаляются	?	?	**да**	**да**	?
Управление возвращается в хост-среду/ОС	**да**	**да**	**да**	**да**	**да**

? – зависит от реализации

n/a – не имеет отношения

getenv — получить строку окружения

```
#include <stdlib.h>
char *getenv(const char *name);
```

Функция **getenv()** ищет в списке окружения (environment list), предоставленном хост-средой, строку, совпадающую со строкой, на которую указывает имя.

Набор имен окружения и метод изменения списка окружения определяются реализацией.

Многие реализации предоставляют нестандартные функции, которые изменяют список окружения.

Функция **getenv()** может быть нереентерабельной и в ряде случаев могут возникать условия гонки с другими потоками исполнения, которые изменяют список окружения.

Возвращает

Функция **getenv()** возвращает указатель на строку, связанную с соответствующим членом списка.

Строка, на которую указывает указатель, не должна изменяться программой, а также может быть перезаписана последующим вызовом функции **getenv()**.

Если указанное имя не может быть найдено, возвращается нулевой указатель.

setenv – изменить или добавить переменную окружения

```
#include <stdlib.h>
int setenv(const char *name, const char *value, int overwrite);
int unsetenv(const char *name);
```

Если переменная **name** ещё в окружении не существует, **setenv()** ее добавляет в окружение со значением **value**.

Если переменная **name** в окружении существует и **overwrite** имеет ненулевое значение, то её значение изменяется на **value.**

Если переменная **name** в окружении существует, но **overwrite** равно нулю, то значение **name** не изменяется, а **setenv()** завершается без ошибки.

Эта функция делает копию строк, указанных в **name** и **value**.

Функция **unsetenv()** удаляет переменную **name** из окружения.

Если **name** в окружении не существует, то функция завершается без ошибки и окружение не изменяется.

Возвращает

При успешном выполнении функций **setenv()** и **unsetenv()** возвращает ноль.

При ошибке возвращается -1, а код ошибки содержится в **errno**.

Ошибки

EINVAL – Значение **name** равно **NULL**, указывает на строку нулевой длины или содержащую символ '**=**'.

ENOMEM – Недостаточно памяти для добавления новой переменной в окружение.

system — выполнить программу

```
#include <stdlib.h>
int system(const char *string);
```

Если **string** является нулевым указателем, функция **system( )** определяет, есть ли в хост-среде командный процессор.

Если **string** не является нулевым указателем, функция **system( )** передает строку, на которую указывает **string** , этому командному процессору для выполнения.

Способ выполнения определяется реализацией и это может привести к тому, что программа, вызванная функцией **system( )** будет вести себя несоответствующим образом или несоответствующим образом завершит работу.

В библиотечной функции **system( )** в *NIX используется **fork(2)** для создания процесса-потомка, в котором посредством **execl(3)** запускается команда оболочки, указанная в **string**:

```
execl("/bin/sh", "sh", "-c", command, (char *)NULL);
```

Функция **system( )** возвращает результат после завершения работы команды.

execl(3)

```
int execl(const char *pathname, const char *arg, ...); // (char *)NULL
```

Возвращает

Если аргумент является нулевым указателем, функция **system()** возвращает ненулевое значение, если командный процессор доступен, и 0, если недоступен.

Если аргумент не является нулевым указателем, а функция **system()** возвращает значение, оно возвращает значение, определенное реализацией.

В linux:

- если процесс-потомок не может быть создан или его состояние невозможно вернуть, то возвращается значение -1, а **errno** присваивается код ошибки.

- если оболочка не может выполниться в процессе-потомке, то возвращаемое значение будет таким же как если бы оболочка-потомок завершилась вызовом **_exit(2)** с состоянием 127.

- если все системные вызовы выполнены без ошибок, то возвращается значение состояния завершения процесса-потомка, использовавшегося для выполнения **string** (состояние завершения оболочки — это состояние завершения последней выполнявшейся команды).

В последних двух случаях возвращаемое значение — это «состояние ожидания», которое можно определить с помощью макроса описанного в **waitpid(2)** (т. е., **WIFEXITED()**, **WEXITSTATUS()** и т. п.).

Лабораторная работа No 3. Понятие процессов.

Изучение системных вызовов **fork(), execve(), getpid(), getppid(), getenv()**.

Создаются две программы – **parent** и **child**.

Перед запуском программы **parent** в окружении создается переменная среды **CHILD_PATH** с именем каталога, где находится программа **child**.

Родительский процесс (программа **parent**) после запуска получает переменные среды, сортирует их в **LC_COLLATE=C** и выводит в **stdout**. После этого входит в цикл обработки нажатий клавиатуры.

Символ «**+**» порождает дочерний процесс, используя **fork()** и **execve()**, запускает очередной экземпляр программы **child**., используя информацию о каталоге из окружения, которую получает, используя функцию **getenv()**. Имя программы (**argv[0]**) устанавливается как **child_XX**, где XX – порядковый номер от 00 до 99. Номер инкрементируется родителем.

Символ «*****» порождает дочерний процесс аналогично предыдущему случаю, однако информацию о его расположении получает, сканируя массив параметров среды, переданный в третьем параметре функции **main()**.

Символ «**&**» порождает дочерний процесс аналогично предыдущему случаю, однако информацию о его расположении получает, сканируя массив параметров среды, указанный в переданный во внешней переменной **extern char **environ**, установленной хост-средой при запуске (см. IEEE Std 1003.1-2017).

При запуске дочернего процесса ему передается сокращенное окружение, включающее набор переменных, указанных в файле, который передается родительскому процессу как параметр ко-

мандной строки. Минимальный набор переменных должен включать SHELL, HOSTNAME, LOGNAME, HOME, LANG, TERM, USER, LC_COLLATE, PATH. Дочерний процесс открывает этот файл, считывает имена переменных, получает из окружения их значение и выводит в `stdout`.

Дочерний процесс (программа **`child`**) выводит свое имя, pid, ppid, открывает файл с набором переменных, считывает их имена, получает из окружения, переданного ему при запуске, их значение способом, указанным при обработке нажатий, выводит в stdout и завершается.

Символ «**&**» завершает выполнение родительского процесса.

Программы компилируются с ключами

```
-W -Wall -Wno-unused-parameter -Wno-unused-variable -std=c11 -pedantic
```

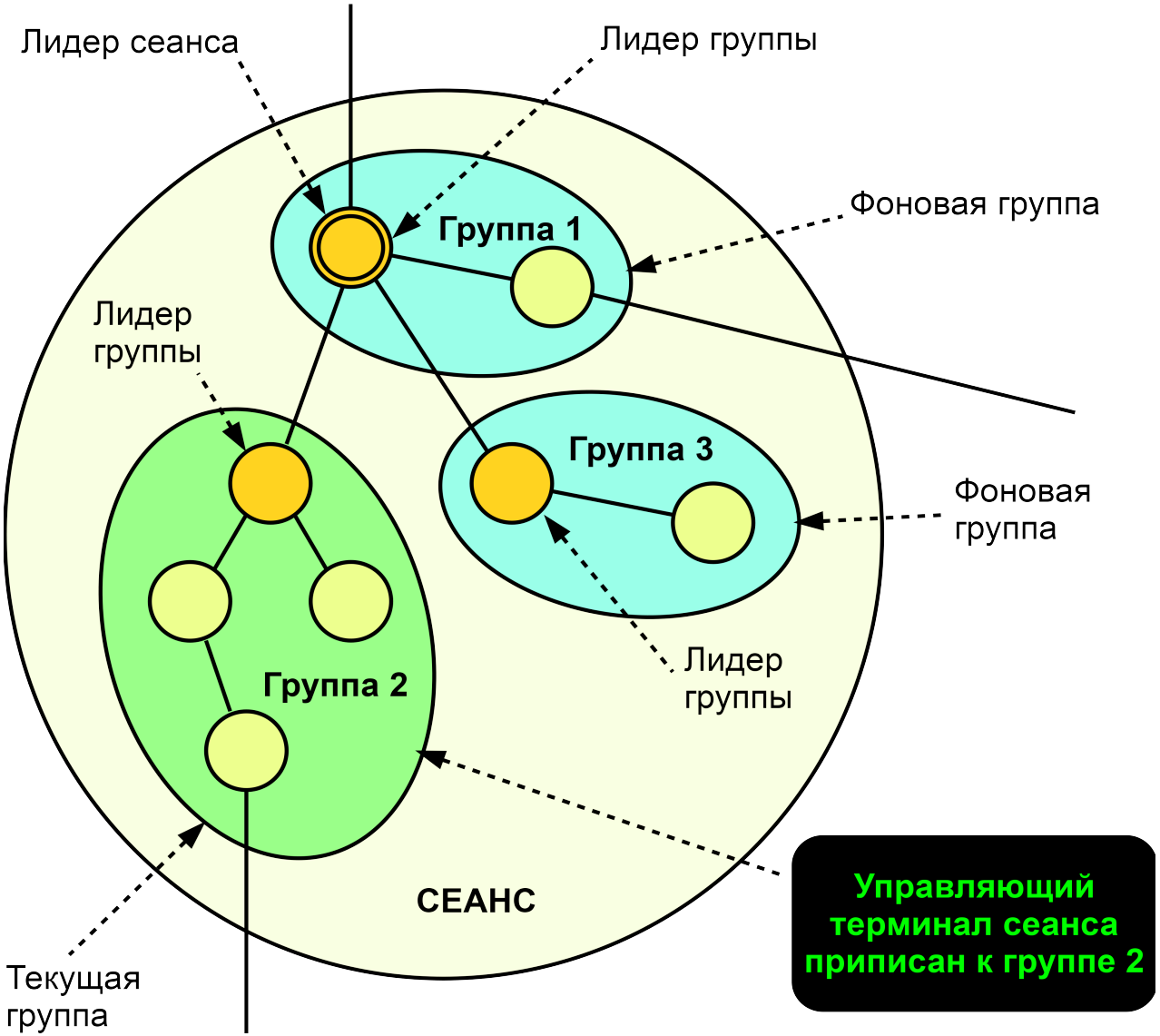
Для компиляции, сборки и очистки используется make.

Группы процессов

Понятия группы процессов, сеанса, лидера группы, лидера сеанса, управляющего терминала сеанса. Системные вызовы **`getpgrp()`**, **`setpgrp()`**, **`getpgid()`**, **`setpgid()`**, **`getsid()`**, **`setsid()`**.

Все процессы в системе связаны родственными отношениями, образуя генеалогическое дерево или лес из таких деревьев, где в качестве узлов деревьев выступают сами процессы, а связями служат отношения родитель-потомок.

Все эти деревья принято разделять на группы процессов.



Группа процессов

Группа процессов включает в себя один или более процессов и существует, пока в группе присутствует хотя бы один процесс.

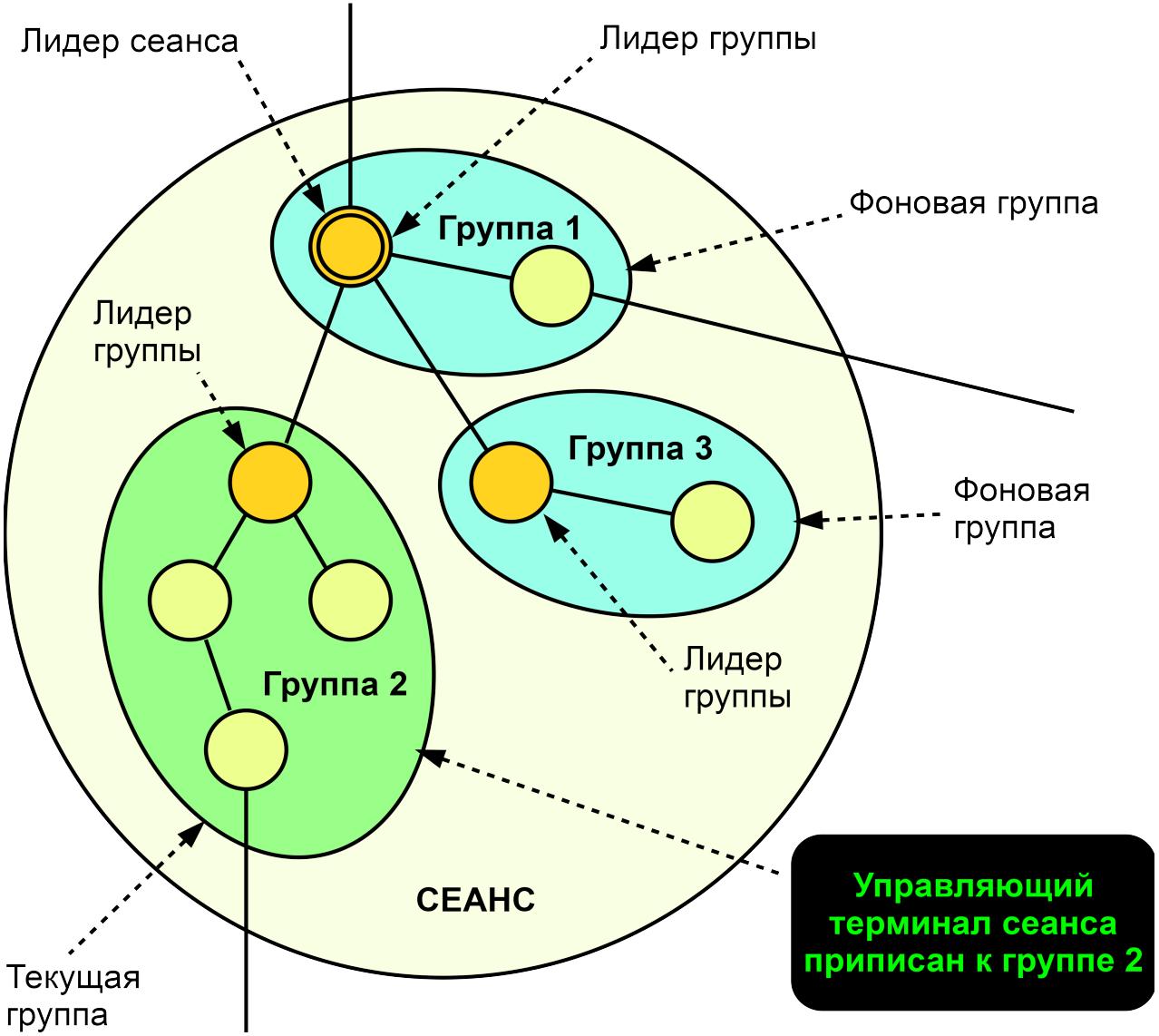
Каждый процесс обязательно включен в какую-нибудь группу.

При рождении нового процесса он попадает в ту же группу процессов, в которой находится его родитель (cgid = pgid).

Процессы могут мигрировать из группы в группу по своему собственному желанию или по желанию другого процесса (в зависимости от версии UNIX).

Многие системные вызовы могут быть применены не к одному конкретному процессу, а ко всем процессам в некоторой группе (waitpid).

Поэтому способ объединения процессы в группы определяется тем, как их собираются использовать.



Сеансы

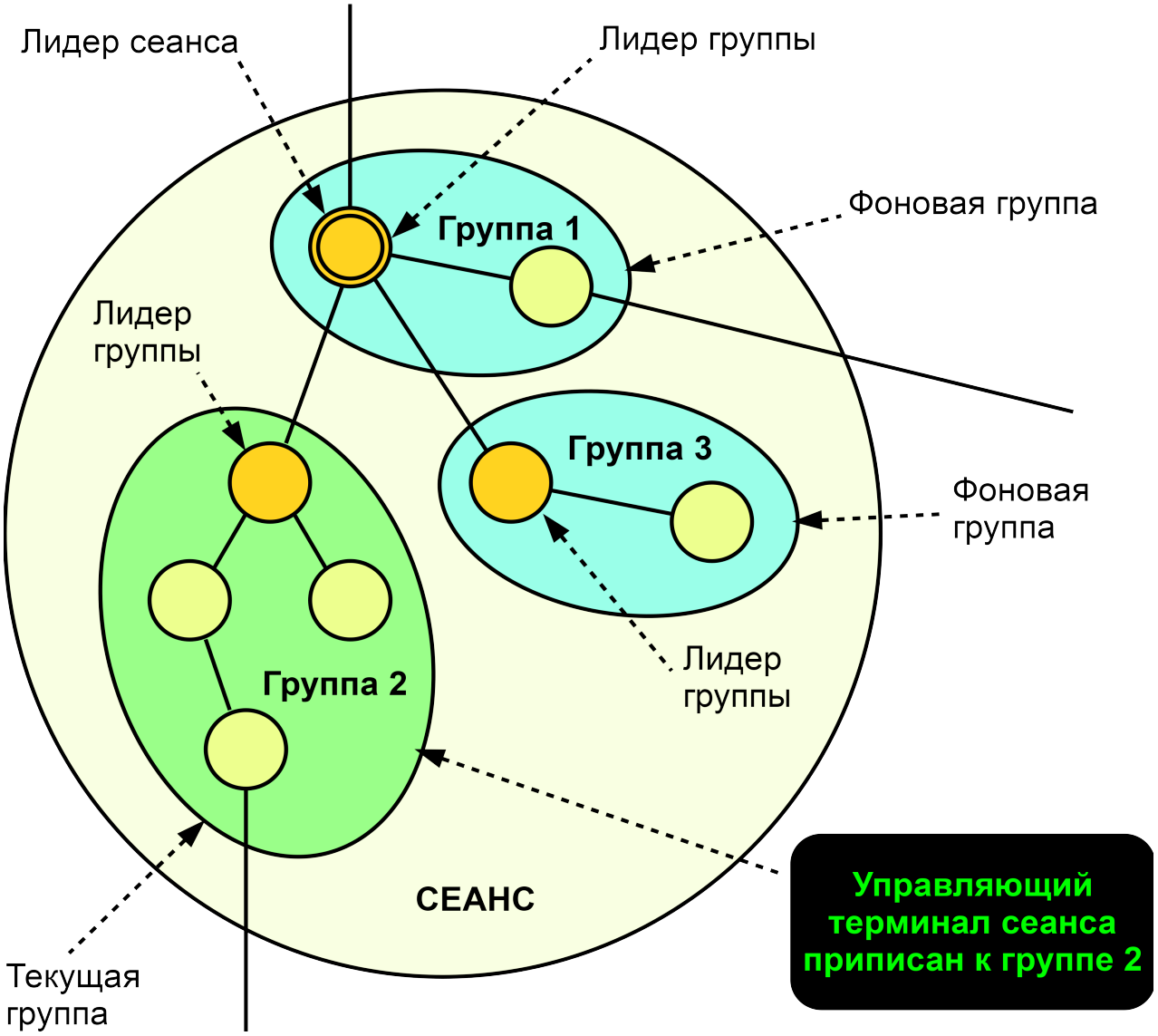
Группы процессов объединяются в сеансы.

Понятие сеанса изначально было введено в UNIX для логического объединения групп процессов, созданных в результате каждого входа и последующей работы пользователя в системе.

В связи с этим с каждым сеансом в системе может быть связан терминал, называемый управляющим терминалом сеанса, через который обычно и общаются процессы сеанса с пользователем.

Сеанс не может иметь более одного управляющего терминала, и один терминал не может быть управляющим для нескольких сеансов.

В то же время могут существовать сеансы, вообще не имеющие управляющего терминала.



Идентификаторы процесса

ID (PID) процесса

Каждый процесс имеет уникальный неотрицательный целочисленный идентификатор (PID), который ему назначается при создании с помощью **fork(2)**.

Процесс может узнать свой PID с помощью вызова **getpid(2)**.

PID имеет тип **pid_t** (определён в **<sys/types.h>**).

PID используется в различных системных вызовах для указания процесса, с которым работает вызов, например, **kill(2)**, **ptrace(2)**, **setpriority(2)**, **setpgid(2)**, **setsid(2)**, **sigqueue(3)** и **waitpid(2)**.

PID процесса сохраняется после вызова **execve(2)**.

Родительский ID (PPID) процесса

ID родительского процесса — это ID процесса, который создал данный процесс с помощью **fork(2)**. Процесс может получить свой PPID с помощью **getppid(2)**. PPID имеет тип **pid_t**.

PPID процесса сохраняется после вызова **execve(2)**.

ID группы процессов и сеанса

У каждого процесса есть ID сеанса и ID группы процессов – они тоже имеют тип **pid_t**.

Процесс может получить ID ***своего сеанса*** с помощью **getsid(2)**, а ID ***своей группы*** процессов с помощью **getpgrp(2)**.

Потомок, создаваемый с помощью **fork(2)**, наследует ID сеанса и группы процессов своего родителя. Идентификатор сеанса и группы сохраняется после **execve(2)**.

Сеансы и группы процессов — это абстракции, предназначенные для поддержки управления заданиями оболочки. Группа процессов (иногда называемая «заданием» (job)) — это набор процессов, у которых одинаковый ID группы процессов.

Оболочка создаёт новую группу процессов для процессов, используемых в одной команде или конвейере (например, два процесса, созданные командой «**`dirwalk | wc`**», помещаются в одну группу процессов).

Членство в группе процессов может быть установлено с помощью **`setpgid(2)`**.

Процесс, чей ID процесса совпадает с его ID группы процессов, называется лидером группы процессов этой группы.

Сеанс

Сеанс – это набор процессов, у которых одинаковый ID сеанса.

Все члены группы процессов имеют одинаковый ID сеанса – они всегда принадлежат одному сеансу и сеансы и группы процессов формируют из процессов жёсткую двухуровневую иерархию.

Новый сеанс создаётся вызовом **`setsid(2)`**. ***ID созданного сеанса совпадает с PID процесса***, который вызвал **`setsid(2)`**. Создатель сеанса также называется лидером сеанса.

Все процессы в сеансе используют общий управляющий терминал.

Управляющий терминал назначается в момент, когда лидер сеанса впервые открывает терминал (если при вызове **`open(2)`** не указан флаг **`O_NOCTTY`**).

Терминал может быть управляющим терминалом не более чем для одного сеанса.

В сеансе может быть только одно активное задание (foreground job), все остальные задания в сеансе считаются фоновыми заданиями (background jobs).

Только активное задание может читать данные из терминала.

Когда процесс в фоне пытается прочитать данные с терминала, его группе процессов посылается сигнал **`SIGTTIN`**, который приостанавливает (suspends) задание.

Если у терминала установлен флаг **`TOSTOP`** (**`termios(3)`**), то только активное задание может писать в терминал. Попытка записи из фонового задания приводит к генерации сигнала **`SIGTTOU`**, который приостанавливает задание.

Если нажимаются клавиши терминала, которые генерируют сигнал (например клавиша interrupt, обычно это комбинация Ctrl-C), то сигнал посылается процессам в активном задании.

С членами группы процессов могут работать различные системные вызовы и библиотечные функции и операции **`fcntl(2)`**, (**`kill(2)`**, **`waitpid(2)`**...).

Идентификаторы пользователя и группы

С каждым процессом связаны идентификатор пользователя и различных групп.

Эти идентификаторы представляются в виде целых чисел с типами **`gid_t`** и **`uid_t`**, соответственно (определены в **`<sys/types.h>`**).

В Linux каждый процесс имеет следующие идентификаторы пользователя и групп:

1) Реальный ID пользователя (real user) и реальный ID группы.

Эти ID определяют кто владелец процесса.

Реальный ID пользователя и группы процесса можно получить с помощью вызовов **`getuid(2)`** и **`getgid(2)`**.

2) Эффективный[1] ID пользователя (effective user) и эффективный ID группы.

Эти ID используются ядром для определения прав, которые будет иметь процесс при доступе к общим ресурсам, таким как очереди сообщений, общая память и семафоры.

В большинстве систем UNIX эти ID также определяют права доступа к файлам (см. п. 4)).

Однако в Linux для этой задачи используются ID файловой системы.

Эффективный ID пользователя и группы процесса можно получить с помощью вызовов **`geteuid(2)`** и **`getegid(2)`**.

---

1 действующий

3) Сохранённые set-user-ID и set-group-ID.

Эти ID используются в программах с установленными битами **set-user-ID** и **set-group-ID** для сохранения копии соответствующих эффективных ID, которые устанавливаются в момент запуска программы (**execve(2)**).

Программа с **set-user-ID** может повышать и понижать права, переключая свой ID эффективного пользователя туда и обратно между значениями её ID реального пользователя и сохранённым **set-user-ID**.

Такое переключение производится с помощью вызовов **seteuid(2)**, **setreuid(2)** или **setresuid(2)**.

Программа с **set-group-ID** выполняет аналогичные задачи с помощью **setegid(2)**, **setregid(2)** или **setresgid(2)**.

Сохранённый **set-user-ID** и **set-group-ID** процесса можно получить с помощью **getresuid(2)** и **getresgid(2)**.

4) **ID пользователя файловой системы и ID группы файловой системы**[2].

Эти ID используются для определения прав доступа к файлам (path_resolution(7)).

Каждый раз при изменении ID эффективного пользователя (группы) ядро также автоматически изменяет ID пользователя (группы) файловой системы на то же значение. Следовательно, ID файловой системы, обычно, равны соответствующим эффективным ID, а семантика проверки прав доступа к файлам в Linux такая же как и у других систем UNIX.

ID файловой системы можно сделать отличным от эффективных ID с помощью вызова **setfsuid(2)** и **setfsgid(2)**.

---

2 есть только в Linux

5) ID дополнительных групп (supplementary group).

Определяет ID добавочных групп, которые используются при проверке доступа к файлам и другим общим ресурсам.

В ядрах Linux до версии 2.6.4 процесс может быть членом 32 дополнительных групп.

Начиная с версии 2.6.4 процесс может быть членом 65536 дополнительных групп. В помощью вызова **sysconf(_SC_NGROUPS_MAX)** можно узнать количество дополнительных групп, в которых процесс может быть членом.

Процесс может получить список ID дополнительных групп с помощь **getgroups(2)**, и изменить этот список с помощью **setgroups(2)**.

Дочерний процесс, созданный **fork(2)**, наследует копии ID пользователя и все группы своего родителя.

При **execve(2)** сохраняются ID реального пользователя и группы процесса, а также ID дополнительных групп, в то же время эффективный и сохранённый ID могут измениться (описано в **execve(2)**).

Кроме целей, отмеченных выше, идентификаторы пользователя процесса также используются:

- при определении права на отправку сигналов (**kill(2)**;
- при определении права на установку параметров планировщика процесса (значение уступчивости, политика и приоритет планирования в реальном времени, увязывание ЦП, приоритет ввода-вывода);
- при проверке ограничения по ресурсам (**getrlimit(2)**);
- при проверке ограничения на количество экземпляров **inotify**[3], которые процесс может создать (**inotify(7)**).

3 Наблюдает за событиями файловой системы

getpgid() – получить идентификатор группы процессов

Каждая группа процессов в системе получает свой собственный уникальный номер – **gid**. Узнать этот номер можно с помощью системного вызова **getpgid()**.

Используя его, процесс может узнать номер группы для себя самого или для процесса из своего сеанса.

Данный системный вызов присутствует не во всех версиях UNIX. Это следствие разделения UNIX на линию BSD и линию System V. Вместо вызова **getpgid()** в таких системах существует системный вызов **getpgrp()**, который возвращает номер группы только для текущего процесса. Linux поддерживает оба системных вызова.

setpgid() – установить идентификатор группы процессов

Для перевода процесса в другую группу процессов, возможно с одновременным ее созданием, применяется системный вызов **setpgid()**.

Перевести в другую группу процесс может либо самого себя (не во всякую и не всегда), либо своего процесса-потомка, который не выполнял системный вызов **exec()**, т.е. не запускал на выполнение другую программу.

При определенных значениях параметров системного вызова создается новая группа процессов с идентификатором, совпадающим с идентификатором переводимого процесса, состоящая первоначально только из одного этого процесса.

Новая группа может быть создана только таким способом, поэтому идентификаторы групп в системе уникальны. Переход в другую группу без создания новой группы возможен только в пределах одного сеанса.

В некоторых разновидностях UNIX системный вызов **setpgid()** отсутствует, а вместо него существует системный вызов **setpgrp()**, который умеет только создавать новую группу процессов с идентификатором, совпадающим с ID текущего процесса, и переводить текущий процесс в нее.

В некоторых разновидностях UNIX, где совместно сосуществуют вызовы **setpgrp()** и **setpgid()**, например в Solaris, вызов **setpgrp()** ведет себя иначе – он аналогичен **setsid()**.

Лидер группы

Процесс, идентификатор которого совпадает с идентификатором его группы, называется лидером группы.

Одно из ограничений на применение вызовов **setpgid()** и **setpgrp()** состоит в том, что лидер группы не может перебраться в другую группу.

Сеанс

Каждый сеанс в системе также имеет свой собственный номер – **sid**.

Для того, чтобы узнать его можно воспользоваться системным вызовом **getsid()**. В разных версиях UNIX на него накладываются различные ограничения.

В Linux такие ограничения отсутствуют.

Лидер сеанса

Использование системного вызова **setsid()** приводит к созданию новой группы, состоящий только из процесса, который его выполнил (он становится лидером новой группы), и нового сеанса, идентификатор которого совпадает с идентификатором процесса, сделавшего вызов.

Такой процесс называется лидером сеанса. Этот системный вызов может применять только процесс, не являющийся лидером группы.

Установить/получить ID группы процесса

```
#include <sys/types.h>
#include <unistd.h>

int   setpgid(pid_t pid, pid_t pgid);
pid_t getpgid(pid_t pid);

pid_t getpgrp(void);                  // по версии POSIX.1
pid_t getpgrp(pid_t pid);             // по версии BSD

int   setpgrp(void);                  // по версии System V
int   setpgrp(pid_t pid, pid_t pgid); // по версии BSD
```

Все перечисленные интерфейсы доступны в Linux и используются для получения и установки идентификатора группы процессов (PGID).

Для получения PGID вызывающего процесса предпочтительней использовать версию POSIX.1 – **`getpgrp(void)`**, а для установки PGID вызывающего процесса – **`setpgid()`**.

Вызов **`setpgid()`** устанавливает PGID у процесса с идентификатором **`pid`** равным **`pgid`**.

Если значение **`pid`** равно 0, то используется идентификатор вызывающего процесса.

Если значение **`pgid`** равно 0, то PGID процесса, указанного в **`pid`**, становится равным его идентификатору процесса.

Если **setpgid()** используется для перевода процесса из одной группы в другую (это делают некоторые оболочки командной строки для объединения каналов процессов), то обе группы процессов должны быть частью одного сеанса (**setsid(2)**). В этом случае в **pgid** указывается существующая группа процессов, в которую нужно выполнить перевод и идентификатор сеанса этой группы должен совпадать с идентификатором сеанса переводимого процесса.

В версии POSIX.1 вызов **getpgrp()** без аргументов возвращает PGID вызывающего процесса.

В версии System V вызов **setpgrp()** без аргументов эквивалентен **setpgid(0, 0)**.

Вовращают

При успешном выполнении **setpgid()** и **setpgrp()** возвращают 0.

В случае ошибки возвращается -1, а **errno** устанавливается в соответствующее значение.

Вызов **getpgrp()** (POSIX.1) всегда возвращает PGID вызывающего процесса.

Вызовы **getpgid()** и **getpgrp()** (BSD) при успешном выполнении возвращают ID группы процессов. При ошибке возвращается -1, а значение **errno** устанавливается соответствующим образом.

setsid() – создает сеанс и устанавливает идентификатор группы процесса

```
#include <sys/types.h>
#include <unistd.h>

pid_t setsid(void);
```

Если вызывающий процесс не является лидером группы процессов, вызов **setsid()** создаёт новый сеанс. Вызывающий процесс становится лидером нового сеанса (то есть, его ID сеанса становится равным ID самого процесса).

Вызывающий процесс также становится лидером группы процессов новой группы процессов в сеансе (то есть, его ID группы процессов становится равным ID самого процесса).

Вызывающий процесс будет единственным в новой группе процессов и новом сеансе.

Изначально, новый сеанс не имеет управляющего терминала.

Управляющий терминал назначается в момент, когда лидер сеанса впервые открывает терминал (если при вызове **open(2)** не указан флаг **O_NOCTTY**).

Терминал может быть управляющим терминалом не более чем для одного сеанса.

Возвращает

При успешном выполнении возвращается идентификатор (нового) сеанса вызывающего процесса. В случае ошибки возвращается **(pid_t)-1**, а **errno** устанавливается в соответствующее значение.

Ошибки

EPERM – вызывающий процесс является лидером группы процессов.

Лидер группы процессов — это процесс, идентификатор группы процессов которого равен идентификатору самого процесса (PID).

Отказ лидера группы процессов выполнять **setsid()** предотвращает возможность того, что сам лидер группы процессов переместится в новый сеанс, в то время как другие процессы в группе останутся в первоначальном сеансе, что поломало бы жёсткую двухуровневую иерархию сценариев и групп процессов.

Для того, чтобы **setsid()** выполнился успешно, следует вызвать **fork(2)** и в родителе **_exit(2)**, а затем в дочернем процессе (который по определению не может быть лидером группы процессов) следует вызвать **setsid()**.

Если сеанс имеет управляющий терминал, у которого не установлен флаг **CLOCAL** и возникает зависание (hangup) терминала, то лидеру сеанса посылается **SIGHUP**.

Если завершается процесс, который является лидером сеанса, то сигнал **SIGHUP** посылается каждому процессу в приоритетной (foreground) группе процессов управляющего терминала.