**9**

**Системные вызовы, функции стандартной библиотеки**

Все версии UNIX предоставляют строго определенный ограниченный набор входов в ядро операционной системы, через которые прикладные задачи имеют возможность воспользоваться базовыми услугами, предоставляемыми UNIX. Эти точки входа получили название системных вызовов. Системный вызов, таким образом, определяет функцию, выполняемую ядром операционной системы от имени процесса, выполнившего вызов, и является интерфейсом самого низкого уровня взаимодействия прикладных процессов с ядром. В среде программирования UNIX они определяются как функции С, независимо от фактической реализации вызова функции ядра операционной системы. В UNIX используется подход, при котором каждый системный вызов имеет соответствующую функцию (или функции) с тем же именем, хранящуюся в стандартной библиотеке языка С (в дальнейшем эти функции будем для простоты называть системными вызовами). Функции библиотеки выполняют необходимое преобразование аргументов и вызывают требуемую процедуру ядра, используя различные приемы. Помимо системных вызовов программисту предлагается большой набор функций общего назначения. Эти функции не являются точками входа в ядро операционной системы, хотя в процессе выполнения многие из них выполняют системные вызовы. Функции, о которых идет речь, хранятся в стандартных библиотеках С и наряду с системными вызовами составляют основу среды программирования в UNIX. Таким образом, часть библиотечных функций является "надстройкой" над системными вызовами, обеспечивающей более удобный способ получения системных услуг. К этим функциям можно отнести функции библиотеки ввода/вывода, функции распределения памяти, часть функций управления процессами и т.д.

**Компиляция, создание, запуск, завершение**

Исходный код + заголовки

|  |  |  |  |
| --- | --- | --- | --- |
| ctype | симв типы | stdio | ввод вывод |
| dirent | струк данных кат | sys/types | примитивы сист данных |
| errno | ошибки | sys/stat | метаданные |
| fcntl | файлов управление | unistd | сист вызовы |
| signal | сигналы | sem | семафоры |

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

Первый аргумент (argc) определяет число параметров, переданных программе, включая ее имя. Указатели на каждый из параметров передаются в массиве argv[], таким образом, через argv[0] адресуется строка, содержащая имя программы, argv[1] указывает на первый параметр и т.д.. до argv[argc-1]. Массив envp[] содержит указатели на переменные окружения, передаваемые программе

Выход – return exit

**Обработка ошибок**

<errno.h> содержит коды ошибок, значения которых может принимать переменная errno. Две функции, помогающие сообщить причину ошибочной ситуации: strerror и perror.

Функция strerror(3C) имеет вид:

#include <string.h>

char \*strerror(int errnum);

Функция принимает в качестве аргумента errnum номер ошибки и возвращает указатель на строку, содержащую сообщение о причине ошибочной ситуации.

Функция perror(3C) объявлена следующим образом:

#include <errno.h>

#include <stdio.h>

void perror(const char \*s);

Функция выводит в стандартный поток сообщений об ошибках информацию об ошибочной ситуации, основываясь на значении переменной errno. Строка s, передаваемая функции, предваряет это сообщение и может служить дополнительной информацией, например содержа название функции или программы, в которой произошла ошибка.

**10**

В среде программирования UNIX существуют два основных интерфейса для файлового ввода/вывода: 1. Интерфейс системных вызовов, предлагающий системные функции низкого уровня, непосредственно взаимодействующие с ядром операционной системы. 2. Стандартная библиотека ввода/вывода, предлагающая функции буферизованного ввода/вывода. Второй интерфейс является "надстройкой" над интерфейсом системных вызовов, предлагающей более удобный способ работы с файлами.

везде fcntl.h

**int open(const char \*path, int oflag, mode\_t mode);**

Первый аргумент (path) является указателем на имя файла. Это имя может быть как абсолютным (начинающимся с корневого каталога /), так и относительным (указанным относительно текущего каталога). Аргумент oflag указывает на режим открытия файла и представляет собой побитное объединение флагов, приведенных в табл. с помощью операции ИЛИ (O\_RDONLY, O\_WRONLY, O\_RDWR и т.д.). Аргумент mode, определяющий права доступа к файлу, используется только при создании файла.

**int creat(const char \*path, mode\_t mode);**

аргумент path определяет имя файла в файловой системе, a mode — устанавливаемые права доступа к файлу. При этом выполняется ряд правил: 1) Если идентификатор группы (GID) создаваемого файла не совпадает с эффективным идентификатором группы (EGID) или идентификатором одной из дополнительных групп процесса, бит SGID аргумента mode очищается (если он был установлен). 2) Очищаются все биты, установленные в маске процесса. 3) Очищается флаг Sticky bit.

далее unistd.h

**int close(int fildes);**

В случае успеха возвращает нулевое значение, в противном случае возвращается -1, а значение переменной errno указывает на причину неудачи.

**int dup(int fildes);**

В случае успешного завершения возвращается новый файловый дескриптор, свойства которого идентичны свойствам дескриптора fildes. Оба указывают на один и тот же файл, одно и то же смещение, начиная с которого будет производиться следующая операция чтения или записи (файловый указатель), и определяют один и тот же режим работы с файлом. Правило размещения нового файлового дескриптора аналогично используемому в функции open(2).

**int dup2(int fildes, int fildes2);**

Функция dup2(2) делает то же самое, однако позволяет указать номер файлового дескриптора, который требуется получить после дублирования. Файловый дескриптор, подлежащий дублированию, передается в первом аргументе (fildes), а новый дескриптор должен быть равен fildes2.

**off\_t lseek(int fildes, off\_t offset, int whence);**

С помощью функции lseek(2) можно установить файловый указатель на любое место файла и тем самым обеспечить прямой доступ к любой части файла. Интерпретация аргумента offset зависит от аргумента whence, который может принимать следующие значения: SEEK\_CUR, SEEK\_END, SEEK\_SET.

**ssize\_t read(int fildes, void \*buf, size\_t nbyte);**

Позволяют считывать данные из файла, на который указывает файловый дескриптор. Аргументы, передаваемые функции read(2), указывают, что следует считать nbyte байт из файла, связанного с дескриптором fildes, начиная с текущего значения файлового указателя. Считанные данные помещаются в буфер приложения, указатель на который передается в аргументе buf. После завершения операции значение файлового указателя будет увеличено на nbyte.

**ssize\_t write(int fildes, void \*buf, size\_t nbyte);**

Аргументы указывают, что следует записать nbyte байт в файл, связанный с дескриптором fildes, начиная с текущего значения файлового указателя. Данные для записи находятся в буфере приложения, указанном аргументом buf. После завершения операции значение файлового указателя будет увеличено на nbyte.

буферизованный ввод-вывод – stdio.h

extern FILE \*stdin; extern FILE \*stdout; extern FILE \*stderr;

fopen(3S) Открывает файл с указанным именем и возвращает файловый указатель, ассоциированный с данным файлом

fclose(3S) Закрывает поток, освобождая буферы

fflush(3S) Очищает буфер потока, открытого на запись

getc(3S) Считывает символ из потока

putc(3S) Записывает символ в поток

gets(3S) Считывает строку из потока

puts(3S) Записывает строку в поток

fread(3S) Считывает указанное число байтов из потока (бинарный ввод)

fwrite(3S) Записывает указанное число байтов в поток (бинарный вывод)

fseek(3S) Позиционирует указатель в потоке

printf(3S) Производит форматированный вывод

scanf(3S) Производит форматированный ввод

fileno(3S) Возвращает файловый дескриптор данного потока

Жесткая связь создается с помощью системного вызова link(2):

#include <unistd.h>

**int link(const char \*existing, const char \*new);**

При этом будет образована новая запись каталога с именем new и номером inode указывающим на метаданные файла existing. Также будет увеличено число связей.

Для удаления жесткой связи используется системный вызов unlink(2):

#include <unistd.h>

**int unlink(const char \*path);**

Символическая связь позволяет косвенно адресовать другой файл файловой системы. Системный вызов symlink(2) служит для создания символической связи. Этим вызовом, кстати, пользуется команда ln -s.

#include <unistd.h>

**int symlink (const char \*name, const char \*synmame);**

После создания символической связи, доступ к целевому файлу name может осуществляться с помощью symname.

**11**

**Файлы отображаемые в памяти**

Системный вызов mmap(2) предоставляет механизм доступа к файлам, альтернативный вызовам read(2) и write(2). С помощью этого вызова процесс имеет возможность отобразить участки файла в собственное адресное пространство. После этого данные файла могут быть получены или записаны путем чтения или записи в память. Функция mmap(2) определяется следующим образом:

#include <sys/types.h>

#include <sys/mman.h>

**caddr\_t mmap(caddr\_t addr, size\_t len, int prot, int flags, int fildes, off\_t off);**

Этот вызов задает отображение len байтов файла с дескриптором fildes, начиная со смещения off, в область памяти со стартовым адресом addr. Аргумент prot определяет права доступа к области памяти, которые должны соответствовать правам доступа к файлу, указанным в системном вызове open(2).

Значения prot:

PROT\_READ Область доступна для чтения r

PROT\_WRITE Область доступна для записи w

PROT\_EXEC Область доступна для исполнения x

PROT\_NONE Область недоступна -

Обычно значение addr задается равным 0, что позволяет операционной системе самостоятельно выбрать виртуальный адрес начала области отображения. В любом случае, при успешном завершении возвращаемое системным вызовом значение определяет действительное расположение области памяти.

Аргумент flags определяет дополнительные особенности управления областью памяти. В табл. 2.13 приведены возможные типы отображения, определяемые аргументом flags.

Значение flags

MAP SHARED Область памяти может совместно использоваться несколькими процессами

MAP PRIVATE Область памяти используется только вызывающим процессом

MAP\_FIXED Требует выделения памяти, начиная точно с адреса addr

MAP\_NORESERVE He требует резервирования области свопинга

Отображение автоматически снимается при завершении процесса. Процесс также может явно снять отображение с помощью вызова munmap(2). Закрытие файла не приводит к снятию отображения.

**Владение и права доступа**

Владелец-пользователь и владелец-группа файла могут быть изменены с помощью системных вызовов:

#include <unistd.h>

#include <sys/types.h>

**int chown(const char \*path, uid\_t owner, gid\_t group);**

**int fchown(int fildes, uid\_t owner, gid\_t group);**

**int lchown(const char \*path, uid\_t owner, gid\_t group);**

Все три вызова работают одинаково за исключением ситуации, когда адресуемый файл является символической связью. В последнем случае вызов lchown действует на сам файл — символическую связь, а не на целевой файл (т.е. не следует символической связи). В функциях chown и lchown файл адресуется по имени, а в fchown — по файловому дескриптору. Если значение owner или group установлено равным -1, соответствующий владелец файла не изменяется.

Права доступа к файлу могут быть изменены с помощью системных вызовов chmod и fchmod:

#include <sys/types.h>

#include <sys/stat.h>

**int chmod(const char \*path, mode\_t mode);**

**int fchmod(int fildes, mode\_t mode);**

Флаги аргумента mode

S\_ISUID 04000 Установить бит SUID

S\_ISGID 020#0 Установить бит SGID, если # равно 7, 5, 3 или 1. Установить обязательное блокирование файла, если # равно 6, 4, 2 или 0

S\_ISVTX 01000 Установить Sticky bit

S\_IRWXU 00700 Установить право на чтение, запись и выполнение для владельца-пользователя

S\_IRUSR 00400 Установить право на чтение для владельца-пользователя

S\_IWUSR 00200 Установить право на запись для владельца-пользователя

S\_IXUSR 00100 Установить право на выполнение для владельца-пользователя

S\_IRWXG 00070 Установить право на чтение, запись и выполнение для владельца-группы

S\_IRGRP 00040 Установить право на чтение для владельца-группы

S\_IWGRP 00020 Установить право на запись для владельца-группы

S\_IXGRP 00010 Установить право на выполнение для владельца-группы

S\_IRWXO 00007 Установить право на чтение, запись и выполнение для остальных пользователей

S\_IROTH 00004 Установить право на чтение для остальных пользователей

S\_IWOTH 00002 Установить право на запись для остальных пользователей

S\_IXOTH 00001 Установить право на выполнение для остальных пользователей

**Метаданные**

Каждый файл помимо собственно данных содержит метаданные, описывающие его характеристики, например, владельцев, права доступа, тип и размер файла, а также содержащие указатели на фактическое расположение данных файла. Метаданные файла хранятся в структуре inode. Часть полей этой структуры могут быть получены с помощью системных вызовов stat:

#include <sys/types.h>

#include <sys/stat.h>

**int stat(const char \*path, struct stat \*buf);**

**int lstat (const char \*path, struct stat \*buf);**

**int fstat(int fildes, struct stat \*buf);**

В качестве аргумента функции принимают имя файла или файловый дескриптор (fstat(2)) и возвращают заполненные поля структуры stat, которые приведены в табл. 2.15.

Поля структуры stat:

mode\_t st\_mode Тип файла и права доступа

ino\_t st\_ino Номер inode. Поля st\_ino и st\_dev однозначно определяют обычные файлы

dev\_t st\_dev Номер устройства, на котором расположен файл (номер устройства файловой системы)

dev\_t st\_rdev Для специального файла устройства содержит номер устройства, адресуемого этим файлом

nlink\_t st\_link Число жестких связей

uid\_t st\_uid Идентификатор пользователя-владельца файла

gid\_t st\_gid Идентификатор группы-владельца файла

off\_t st\_size Размер файла в байтах. Для специальных файлов устройств это поле не определено

time\_t st\_atime Время последнего доступа к файлу

time\_t st\_mtime Время последней модификации данных файла

time\_t st\_ctime Время последней модификации метаданных файла

long st\_blksize Оптимальный размер блока для операций ввода/вывода.

long st\_blocks Число размещенных 512-байтовых блоков хранения данных.

Определение типа файла

S\_ISFIFO(mode) FIFO

S\_ISDIR(mode) Каталог

S\_ISREG(mode) Обычный файл

S\_ISLNK(mode) Символическая связь

S\_ISSOCK(mode) Сокет

**12**

**Процессы**

Каждый процесс характеризуется набором атрибутов и идентификаторов, позволяющих системе управлять его работой. Важнейшими из них являются идентификатор процесса PID и идентификатор родительского процесса PPID. PID является именем процесса в операционной системе, по которому мы можем адресовать его, например, при отправлении сигнала. PPID указывает на родственные отношения между процессами, которые (как и в жизни) в значительной степени определяют его свойства и возможности. Еще идентификатор пользователя UID, эффективный идентификатор пользователя EUID, идентификатор группы GID и эффективный идентификатор группы EGID. Эти идентификаторы определяют права процесса в файловой системе, и как следствие, в операционной системе в целом.

Для получения значений идентификаторов процесса используются следующие системные вызовы:

#include <sys/types.h>

#include <unistd.h>

**uid\_t getuid(void);**

**uid\_t geteuid(void);**

**gid\_t getgid(void);**

**gid\_t getegid(void);**

Процесс также может изменить значения этих идентификаторов с помощью системных вызовов:

#include <sys/types.h>

#include <unistd.h>

**int setuid(uid\_t uid);**

**int setegid(gid\_t egid);**

**int seteuid(uid\_t euid);**

**int setgid(gid\_t gid);**

Системные вызовы setuid(2) и setgid(2) устанавливают сразу реальный и эффективный идентификаторы, а системные вызовы seteuid(2) и setegid(2) — только эффективные.

**Выделение памяти:**

Существует четыре способа выделения памяти:

1. Переменная объявлена как глобальная, и ей присвоено начальное значение в исходном тексте программы

2. Значение глобальной переменной неизвестно на этапе компиляции

3. Переменные автоматического класса хранения, используемые в функциях программы, используют стек.

4. Выделение памяти явно запрашивается некоторыми системными вызовами или библиотечными функциями.

Дополнительная память выделяется из хипа — области виртуальной памяти, расположенной рядом с сегментом данных, размер которой меняется для удовлетворения запросов на размещение. Следующий за сегментом данных адрес называется разделительным или брейк. Изменение размера сегмента данных по существу заключается в изменении брейк-адреса. Для изменения его значения UNIX предоставляет процессу два системных вызова:

#include <unistd.h>

**int brk(void \*endds);**

**void \*sbrk(int incr);**

Системный вызов brk(2) позволяет установить значение брейк-адреса равным endds и, в зависимости от его значения, выделяет или освобождает память. Функция sbrk(2) изменяет значение брейк-адреса на величину incr. Если значение incr больше 0, происходит выделение памяти, в противном случае, память освобождается.

Существуют четыре стандартные библиотечные функции, предназначенные для динамического выделения/освобождения памяти.

#include <stdlib.h>

**void \*malloc(size\_t size);**

**void \*calloc(size\_t nelem, size\_t elsize);**

**void \*realloc(void \*ptr, size\_t size);**

**void free(void \*ptr);**

Функция malloc(3C) выделяет указанное аргументом size число байтов. Функция calloc(3C) выделяет память для указанного аргументом nelem числа объектов, размер которых elsize. Выделенная память инициализируется нулями. Функция realloc(3C) изменяет размер предварительно выделенной области памяти (увеличивает или уменьшает, в зависимости от знака аргумента size). Увеличение размера может привести к перемещению всей области в другое место виртуальной памяти, где имеется необходимое свободное непрерывное виртуальное адресное пространство. Функция free(3C) освобождает память, предварительно выделенную с помощью функций malloc(3C), calloc(3C) или realloc(3C), указатель на которую передается через аргумент ptr.

**Создание процессов**

Новый процесс порождается с помощью системного вызова fork(2):

#include <sys/types.h>

#include <unistd.h>

**pid\_t fork(void);**

Порожденный, или дочерний процесс, хотя это кажется странным, является точной копией процесса, выполнившего этот вызов, или родительского процесса. В частности, дочерний процесс наследует такие атрибуты родителя, как:идентификаторы пользователя и группы, переменные окружения, диспозицию сигналов и их обработчики, ограничения, накладываемые на процесс, текущий и корневой каталог, маску создания файлов, все файловые дескрипторы, включая файловые указатели, управляющий терминал. Легче перечислить немногочисленные различия между этими процессами, а именно: дочернему процессу присваивается уникальный идентификатор PID. идентификаторы родительского процесса PPID у этих процессов различны, дочерний процесс свободен от сигналов, ожидающих доставки, значение, возвращаемое системным вызовом fork(2) различно для родителя и потомка.

Операционная система предоставляет процессу ряд функций, позволяющих ему контролировать выполнение потомков.

#include <sys/types.h>

#include <sys/wait.h>

**pid\_t wait(int\* stat\_loc);**

**int waitpid(idtype\_t idtype, id\_t id, siginfo\_t \* infop, int options);**

**pid\_t waitpid(pid\_t pid, int \*stat\_loc, int options);**

Первый из этих вызовов wait(2) обладает самой ограниченной функциональностью — он позволяет заблокировать выполнение процесса, пока кто-либо из его непосредственных потомков не прекратит существование. Вызов wait(2) немедленно возвратит состояние уже завершившегося дочернего процесса в переменной stat\_loc, если последний находится в состоянии зомби. Системный вызов waitid(2) предоставляет больше возможностей для контроля дочернего процесса. Функция waitpid(2), как и функции wait(2) и waitid(2), позволяет контролировать определенное множество дочерних процессов.

**Группы и сеансы:**

Процесс имеет возможность определить идентификатор собственной группы процессов или группы процесса, который является членом того же сеанса. Для этого используются два системных вызова: getpgrp(2) и getpgid(2):

#include <sys/types.h>

#include <unistd.h>

**pid\_t getpgrp(void);**

**pid\_t getpgid(pid\_t pid);**

Аргумент pid, который передается функции getpgid(2), адресует процесс, идентификатор группы которого требуется узнать. Если этот процесс не принадлежит к тому же сеансу, что и процесс, сделавший системный вызов, функция возвращает ошибку.

Системный вызов setpgid(2) позволяет процессу стать членом существующей группы или создать новую группу.

#include <sys/types.h>

#include <unistd.h>

**int setpgid(pid\_t pid, pid\_t pgid);**

Функция устанавливает идентификатор группы процесса pid равным pgid. Процесс имеет возможность установить идентификатор группы для себя и для своих потомков (дочерних процессов). Однако процесс не может изменить идентификатор группы для дочернего процесса, который выполнил системный вызов exec(2), запускающий на выполнение другую программу.Если значения обоих аргументов равны, то создается новая группа с идентификатором pgid, а процесс становится лидером (group leader) этой группы. Поскольку именно таким образом создаются новые группы, их идентификаторы гарантированно уникальны. Заметим, что группа не удаляется при завершении ее лидера, пока в нее входит хотя бы один процесс.

Идентификатор сеанса можно узнать с помощью функции getsid(2):

#include <sys/types.h>

#include <unistd.h>

**pid\_t getsid(pid\_t pid);**

Как и в случае с группой, идентификатор pid должен адресовать процесс, являющийся членом того же сеанса, что и процесс, вызвавший getsid(2). Заметим, что эти ограничения не распространяются на процессы, имеющие привилегии суперпользователя.

Вызов функции setsid(2) приводит к созданию нового сеанса:

#include <sys/types.h>

#include <unistd.h>

**pid\_t setsid(void);**

Новый сеанс создается лишь при условии, что процесс не является лидером какого-либо сеанса. В случае успеха процесс становится лидером сеанса и лидером новой группы.

**Демоны:**

Демоны играют важную роль в работе операционной системы. Достаточно будет сказать, что возможность терминального входа пользователей в систему, доступ по сети, использование системы печати и электронной почты, — все это обеспечивается соответствующими демонами — неинтерактивными программами, составляющими собственные сеансы (и группы) и не принадлежащими ни одному из пользовательских сеансов (групп). Некоторые демоны работают постоянно, наиболее яркий пример такого демона — процесс init(1M), являющийся прародителем всех прикладных процессов в системе. При описании взаимодействия процессов с терминалом и пользователем в разделе "Группы и сеансы", отмечалось особое место демонов, которые не имеют управляющего терминала. Теперь в отношении демонов можно сформулировать ряд правил, определяющих их нормальное функционирование, которые необходимо учитывать при разработке таких программ:

1. Демон не должен реагировать на сигналы управления заданиями, посылаемые ему при попытке операций ввода/вывода с управляющим терминалом. Начиная с некоторого времени, демон снимает ассоциацию с управляющим терминалом, но на начальном этапе запуска ему может потребоваться вывести то или иное сообщение на экран.

2. Необходимо закрыть все открытые файлы (файловые дескрипторы), особенно стандартные потоки ввода/вывода. Многие из этих файлов представляют собой терминальные устройства, которые должны быть закрыты, например, при выходе пользователя из системы. Предполагается, что демон остается работать и после того, как пользователь "покинул" UNIX.

3. Необходимо снять его ассоциацию с группой процессов и управляющим терминалом. Это позволит демону избавиться от сигналов, генерируемых терминалом (SIGINT или SIGHUP), например, при нажатии определенных клавиш или выходе пользователя из системы.

4. Сообщения о работе демона следует направлять в специальный журнал с помощью функции syslog(3), — это наиболее корректный способ передачи сообщений от демона.

5. Необходимо изменить текущий каталог на корневой. Если этого не сделать, а текущий каталог, допустим, находится на примонтированной файловой системе, последнюю нельзя будет размонтировать. Самым надежным выбором является корневой каталог, всегда принадлежащий корневой файловой системе.