Содержание

1 Обзор ОС UNIX: архитектура, вход в систему, файлы и каталоги, ввод и вывод.

1.1 Архитектура UNIX

1.1.1 Определение ОС

Операционная система (ОС) – это программное обеспечение (ПО), которое управляет аппаратными ресурсами компьютера и предоставлет среду выполнения прикладных программ (*application*). Обычно это ПО называют ядром (kernel), т.к. оно имеет относительно небольшой объем и состовляет основу системы (см. рисунок ниже).

1.1.2 Интерфес ядра

Интерфес ядра – это слой ПО, называемый системными вызовами (*system calls*). Библиотеки функций общего пользования (*library routines*) строятся на основе интерфеса системных вызовов, но прикладная программа (*application*) может свободно пользоваться как теми, так и другими. Т.е. прикладная программа может использовать как системные вызовы, так и библиотечные функции.

1.1.3 Коммандая оболочка

Коммандая оболочка (*shell*) – это особое приложение, которое предоставлет интерфес для запуска других приложений.

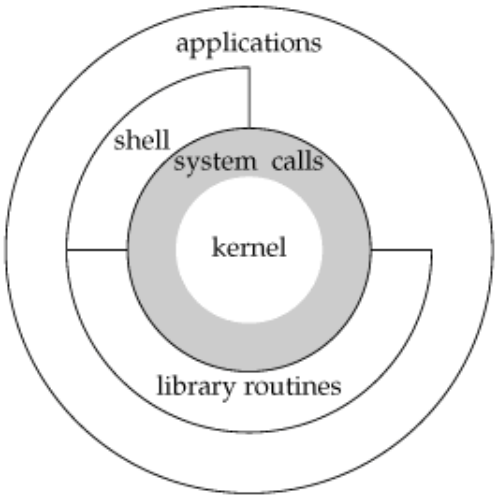
1.1.4 Обобщение

В общем, ОС – это ядро и всё остальное ПО, которое делает компьютера пригодным к использованию. В состов этого ПО входят системные утилиты (например ls, cat), прикладные программы (например GIMP), коммандые оболочки (например bash, fish), библиотеки функций общего пользования (например stdio для Си) и т.п..

1.1.5 Linux

Linux – ядро ОС GNU (или же GNU/Linux). Отдельно Linux – это ядро, называть его ОС не совсем корректно.

1.1.6 Схема, отражающая структуру UNIX



1.2 Вход в систему

При входе в систему UNIX мы вводим имя пользователя и пароль. Послеэ того система отыскивает введенное имя в файле паролей; обычно это файл `/etc/passwd`. Файл паролей содержит записи, каждая из которых состоит из семи полей, разделенных двоеточиями: имя пользователя, зашифрованный пароль, числовой идентификатор пользователя (`205`), числовой идентификатор группы (`105`), поле комментария, домашний каталог (`/home/sar`) икомандный интерпретатор (`/bin/ksh`).
Пример записи:
`sar:x:205:105:Stephen Rago:/home/sar:/bin/ksh`
Все современные системы хранят пароли в отдельном файле.
После входа в систему пользователь получает доступ к коммандной оболочке (например к bash или sh).

1.3 Файлы и каталоги

1.3.1 Файловая система

- Файловая система UNIX представляет собой иерархическую древовидную структуру, состоящую из каталогов и файлов. Начинается она с каталога, который называется корнем (`root`), а имя этого каталога представлено единственным символом - `/`.

- Каталог представляет собой файл, в котором содержатся каталожные записи. Логически каждую такую запись можно представить в виде струк-

туры,состоящей из имени файла и дополнительной информации, описывающей атрибуты файла.

- Атрибуты файла - это такие характеристики, как тип файла (обычный файл или каталог), размер файла, владелец файла, права доступа к файлу (есть ли у других пользователей доступ к файлу), время последней модификации файла.
  Для получения атрибутов файла используются функции `stat()` и `fstat()`.

1.3.2 Имя файла

Имена элементов каталога называются именами файлов. Имя файла не может содержать слеш `/` или нулевой символ `0`. Файл «точка» `.` – это текущий каталог, файл «точка-точка» `..` – родительский. При создании нового каталога в нём создаются файлы «точка» и «точка-точка». Для корневого каталога «точка-точка» – это то же самое что и «точка».
Современные ОС позволяют создавать файлы с длинной имени не менее 255 символов.

1.3.3 Путь к файлу

Последовательность одного или более имен файлов (каталог – это тоже файл), разделенных слэшами, образует строку пути к файлу. Эта строка может также начинаться с символа слэша, и тогда она называется строкой **абсолютного пути**, в противномслучае – строкой **относительного пути**. В случае относительного пути маршрут начинается от текущего каталога.
Также в пути могут присутсвовать «точка» и «точка-точка», например путь `../qwe` – это файл `qwe` в родительском каталоге.

1.3.4 Рабочий каталог

У каждого процесса имеется свой рабочий каталог, который иногда называют текущим рабочим каталогом. Это каталог, от которого отсчитываются все относительные пути, используемые в программе. Процесс может изменить свой рабочий каталог с помощью функции `chdir`.

1.3.5 Домашний каталог

Когда пользователь входит в систему, рабочим каталогом становится его домашний каталог. Домашний каталог пользователя определяется всоответствии с записью в файле паролей.

1.4 Ввод и вывод

1.4.1 Дескрипторы файлов

Дескрипторы файлов - это, как правило, небольшие целые положительные числа, используемые ядром для идентификации файлов, к которым обращается конкретный процесс. Всякий раз, когда процесс открывает существующий или создает новый файл, ядро возвращает его дескриптор, который затем используется для выполнения над файлом операций чтения или записи.

1.4.2 Стандартный ввод, стандартный вывод,Стандартный вывод сообщений об ошибках

По принятым соглашениям все командные оболочки при запуске новой программы открывают для нее три файловых дескриптора: файл стандартного ввода (`stdin`), файл стандартного вывода (`stdout`) и файл стандартного вывода сообщений об ошибках (`stderr`).
`ls > file.out` – перенаправление вывода `stdout` в файл `file.out`
`ls 2> file.err` – перенаправление вывода ошибок `stderr` в файл `file.err`

1.4.3 Небуферизованный ввод-вывод

Небуферизованный ввод-вывод осуществляется функциями `open` , `read` , `write` , `lseek` и `close` . Все эти функции работают с файловыми дескрипторами.
Заголовочный файл `<unistd. h>` , подключаемый из файла `apue.h` , и константы `STDIN_FILENO` и `STDOUT_FILENO` являются частью стандарта **POSIX.**
Константы `STDIN_FILENO` и `STDOUT_FILENO` , определенные в файле `<unistd. h>` ,устанавливают дескрипторы файлов стандартного ввода и стандартного вывода.Обычные значения этих констант - соответственно 0 и 1.

1.4.4 Стандартные функции ввода-вывода

Стандартные функции ввода-вывода предоставляют буферизованный интерфейс к функциям небуферизованного ввода-вывода. Использование стандартных функций ввода-вывода избавляет нас от необходимости задумываться о выборе оптимального размера буфера.
Другое преимущество стандартных функций ввода-вывода в том, что они значительно упрощают обработку пользовательского ввода (например `fgets()` или `printf()`).
`<stdio.h>` содержит прототипы всех стандартных функций ввода-вывода.

2 Обзор ОС UNIX: программы и процессы, обработка ошибок, идентификация пользователя.

2.1 Программы и процессы

2.1.1 Программа

Программа – это исполняемый файл, размещенный на диске. Программа считывается в память и затем выполняется ядром через вызов одной из шести функций семейства `exec`.

2.1.2 Процессы и идентификаторы процессов

Программа, находящаяся в процессе исполнения, называется процессом.
В некоторых ОС для обозначения выполняемой в данный момент программы используется термин задача.
UNIX обеспечивает присвоение каждому процессу уникального числового идентификатора, который называется идентификатором процесса. Идентификатор процесса – всегда целое неотрицательное число.

2.1.3 getpid()

Получить собственный идентификатор процесса можно при помощи функции `getpid()` (get process id)

2.1.4 Управление процессами

Три основные функции отвечают за управление процессами:

- `fork()` – создает копию вызывающего процесса
- `exec()` – замещает дочерний процесс некоторой программой. Функция `exec` имеет шесть разновидностей
- `waitpid()` – ждёт завершения некоторого процесса с определенным `pid`

2.2 Обработка ошибок

2.2.1 errno

Очень часто при возникновении ошибки в любой из функций системы UNIX эта функция возвращает отрицательное число, а в глобальную переменную `errno` записывается некоторое целое, которое несёт дополнительную информацию о возникшей ошибке.

Например, функция open возвращает либо файловый дескриптор – неотрицательное число, либо `-1` в случае возникновения ошибки. Некоторые функции следуют иному соглашению. Например, большинство функций, которые должны возвращать указатель на какой-либо объект, в случае ошибки возвращают пустой указатель (`NULL`).
Определения переменной `errno` и констант всех возможных кодов ошибок находятся в заголовочном файле `<errno.h>`. Имена констант начинаются с символа E (например `EACCES` – возникли проблемы с правами доступа, например при открытии файла).
Стандарты POSIX и ISO C определяют `errno` как символ, раскрывающийсяв изменяемое выражение `lvalue` (то есть выражение, которое может стоять слева от оператора присваивания) целого типа.

2.2.2 Вывод сообщения об ошибке

Для вывода сообщений об ошибках стандарт C предусматривает две функции:

- `char *strerror(int errnum);` – преобразует `errnum` (равный `errno`) в строку сообщения об ошибке и возвращает указатель на нее
- `void perror(const char *msg);` – на основе `errno` выводит сообщение об ошибке с префиксом `msg:`, т.е. `msg: error_text`. Вывод заканчивается символом перевода строки.

2.2.3 Восстановление после ошибок

Ошибки, определенные в `<errno.h>`, могут быть разделены на две категории - фатальные и нефатальные. Восстановление нормальной работы после фатальных ошибок невозможно. Самое лучшее, что мы можем сделать, – это вывести сообщение об ошибке на экран или записать его в файл журнала и завершить работу приложения. Нефатальные ошибки допускают нормальное продолжение работы. Большинство нефатальных ошибок по своей природе носят временный характер (например, нехватка ресурсов), и их можно избежать при меньшей загруженности системы.

2.3 Идентификация пользователя

2.3.1 Идентификатор пользователя

Идентификатор пользователя из записи в файле паролей представляет собой числовое значение, которое однозначно идентифицирует пользователя в системе. Идентификатор пользователя назначается системным администратором при создании учетной записи и не может быть изменен пользователем. Как правило,

каждому пользователю назначается уникальный идентификатор. Ядро использует идентификаторпользователя для проверки прав на выполнение определенных операций.
Пользователь с идентификатором `0` называется суперпользователем, или `root`.
В файле паролей этому пользователю обычно присвоено имя `root`.
Если процесс имеетпривилегии суперпользователя,большинство проверок прав доступа к файлам просто не выполняется. Некоторые системные операции доступны только суперпользователю. Суперпользователь обладает неограниченной свободой действий в системе.

2.3.2 Идентификатор группы

Кроме всего прочего, запись в файле паролей содержит числовой идентификатор группы. Он также назначается системным администратором при создании учетной записи.
Обычно группы используются для распределения пользователей по проектам или отделам. Это позволяет организовать совместное использование ресурсов, например файлов, членами определенной группы.
В системе существует файл групп, в котором указаны соответствия имен группих числовым идентификаторам. Обычно этот файл называется `/etc/group`.
Каждый пользователь может состоять в нескольких группах.

3 Обзор ОС UNIX: сигналы, представление времени, системные вызовы и библиотечные функции.

3.1 Сигналы

Сигналы используются, чтобы известить процесс о наступлении некоторого состояния. Например, если процесс попытается выполнить деление на ноль,он получит уведомление в виде сигнала `SIGFPE` (floating-point exception – ошибка выполнения операции с плавающей точкой). Процесс может реагировать на сигнал тремя способами:

- Игнорировать сигнал
- Разрешить выполнение действия по умолчанию
- Определить функцию, которая будет вызвана для обработки сигнала (такие функции называют перехватчиками (handler) сигналов).

`Ctrl+c` – генерирует сигнал прерывания.
При помощи функции `kill` процесс может послать сигнал другому процессу.

При этом процесс который посылает сигнал должен быть владельцем процесса которому посылается сигнал.

3.2 Представление времени

Исторически в системе UNIX поддерживается два различных способа представления временных интервалов:

- Календарное время. Значения в этом представлении хранят число секунд, прошедших с начала Эпохи: 00:00:00 1 января 1970 года по согласованному всемирному времени (Coordinated Universal Time - UTC)
- Время работы процесса. Оно еще называется процессорным временем и измеряет ресурсы центрального процессора, использованные процессом.Значения в этом представлении измеряются в тактах (ticks). Исторически сложилось так, что в различных системах в одной секунде может быть 50, 60 или 100 тактов. Для хранения времени в этом представлении используется тип данных `clock_t`.

При измерении времени выполнения процессасистема UNIX хранит три значения для каждого процесса:

- Общее время (**Clock time**) – это отрезок времени, затраченный процессом от момента запуска до завершения
- Пользовательское время (**User CPU time**) – это время, затраченное на исполнение машинных инструкций самой программы
- Системное время (**System CPU time**) – это время, затраченное на выполнение ядром машинных инструкций от имени процесса

Сумму пользовательского и системного времени часто называют процессорным временем.

3.3 Системные вызовы и библиотечные функции

Любая операционная система обеспечивает прикладным программам возможность обращения к системным службам. Во всех реализациях UNIX имеется строго определенное число точек входа в ядро, которые называются системными вызовами (system calls, см. картинку в начале).
В ОС Linux имеется от 240 до 260 системных вызовов в зависимости от версии.
В ОС FreeBSD около 320 системных вызовов.
В системе UNIX для каждого системного вызова предусматривается одноименная функция в стандартной библиотеке языка C. Пользовательский процесс вызывает эту функцию стандартными средствами языка C.

С точки зрения разработчика системы между системным вызовом и библиотечной функцией имеются коренные различия. Но с точки зрения пользователя эти различия носят непринципиальный характер. В контексте нашей книги и системные вызовы, и библиотечные функции можно представлять как обычные функции языка С.

4 Стандарты и реализации ОС UNIX: пределы ISO C, пределы POSIX, функции sysconf(), pathconf() и fpathconf(), элементарные системные типы данных.

4.1 Стандартизация UNIX

4.1.1 ISO C

ISO C, ANSI C – стандарт языка С, опубликованный Американским национальным институтом стандартов (American National Standards Institute – ANSI).
ISO C включает в себя следующие заголовочные файлы (библиотеки):

- `<assert.h>` – Проверка программных утверждений
- `<complex.h>` – Поддержка арифметики комплексных чисел
- `<ctype.h>` – Типы символов
- `<errno.h>` – Коды ошибок
- `<fenv.h>` – Окружение операций с плавающей запятой
- `<float.h>` – Арифметика с плавающей запятой
- `<inttypes.h>` – Преобразования целочисленных типов
- `<iso646.h>` – Альтернативные макросы операторов отношений
- `<limits.h` – Константы реализации (см. ниже, пределы ISO C)
- `<locale.h` – Классы региональных настроек (локалей)
- `<math.h>` – Математические константы (например π)
- `<setjmp.h>` – Нелокальные переходы

- `<signal.h>` – Сигналы (см. ниже)
- `<stdarg.h` – Списки аргументов переменной длины
- `<stdbool.h>` – Логический тип и значения
- `<stddef.h>` – Стандартные определения
- `<stdint.h>` – Целочисленные типы
- `<stdio.h>` – Стандартная библиотека ввода/вывода
- `<stdlib.h>` – Функции общего назначения
- `<string.h>` – Операции над строками
- `<tgmath.h>` – Макроопределения математических операций
- `<time.h>` – Время и дата
- `<wchar.h>` – Расширенная поддержка многобайтных символов («широкие» символы)
- `<wctype.h>` – Классификация и функции преобразования многобайтных символов

4.1.2 IEEE POSIX

POSIX – это семейство стандартов, разработанных организацией IEEE (Institute of Electrical and Electronics Engineers). POSIX (Portable Operating System Interface – переносимый интерфейс операционных систем) – набор стандартов, описывающих интерфейсы между операционной системой и прикладной программой (системный API), библиотеку языка C и набор приложений и их интерфейсов. Стандарт POSIX.1 не определяет понятие суперпользователя. Вместо этого он требует, чтобы некоторые операции были доступны только при наличии «соответствующих привилегий», но определение этого термина POSIX.1 оставляет на усмотрение конкретной реализации.

4.1.3 Single UNIX Specification

Single Unix Specification (Единая спецификация UNIX) — это надмножество стандарта POSIX.1 и определяет дополнительные интерфейсы, расширяющие возможности, предоставляемые базовой спецификацией POSIX.1. Стандарт POSIX.1 является эквивалентом раздела Base Specification (базовые спецификации) спецификации Single UNIX Specification.

Расширение X/Open System Interface (XSI) определяет дополнительные интерфейсы POSIX.1, которые должны поддерживаться реализацией, чтобы она получила право именоваться «XSI-совместимой». В их число входят: синхронизация файлов, адрес и размер стека потока, синхронизация потоков между процессами и символьная константа _XOPEN_UNIX.

4.1.4 FIPS

Аббревиатура FIPS означает Federal Information Processing Standard (Федеральный стандарт обработки информации). Этот стандарт опубликован правительством США, которое использовало его при покупке компьютерных систем.
На данный момент стандарт FIPS отменён.

4.2 Реализации UNIX

ISO C, IEEE POSIX и Single UNIX Specification – это стандарты которые описывают только `спецификации интерфейса`. На основе этих спецификаций реализуются реальные ОС.

4.2.1 UNIX System V Release 4

Версия UNIX System V Release 4 (SVR4) была выпущена подразделением AT&T. Версия SVR4 объединила функциональность AT&T UNIX System Release 3.2 (SVR3.2), SunOS – операционной системы от Sun Microsystems, 4.3BSD, выпущенной Калифорнийским университетом, и Xenix — операционной системы от корпорации Microsoft – в единую операционную систему.
Исходные тексты SVR4 были опубликованы в конце 1989 года, а первые копии стали доступны конечным пользователям в 1990 году.

4.2.2 4.4BSD

Версии Berkeley Software Distribution (BSD) разрабатывались и распространялись Computer Systems Research Group (CSRG) – Группой исследования компьютерных систем Калифорнийского университета в Беркли.
Изначально BSD-системы содержали исходный код, запатентованный AT&T, и подпадали под действие лицензий AT&T. Чтобы получить исходный код BSD-системы, требовалась лицензия AT&T на UNIX.

4.2.3 FreeBSD

Операционная система FreeBSD базируется на 4.4BSD-Lite. Проект FreeBSD образован с целью дальнейшего развития линейки BSD-систем после того, как в

Беркли было принято решение о прекращении работ над BSD-версиями операционной системы UNIX и проект 386BSD оказался заброшенным.
Все программное обеспечение, разработанное в рамках проекта FreeBSD, является свободно распространяемым как в исходных текстах, так и в виде бинарных дистрибутивов.

4.2.4 Linux

Linux – это операционная система, которая предоставляет все богатства программного окружения UNIX и свободно распространяется в соответствии с Общественной лицензией GNU (GNU Public License).
ОС Linux была создана Линусом Торвальдсом в 1991 году в качестве замены ОС MINIX.

4.2.5 Mac OS X

Ядро Mac OS X – «Darwin» представляет собой комбинацию ядра Mach, ОС FreeBSD и объектно-ориентированного фреймворка для драйверов и других расширений ядра.

4.2.6 Solaris

Solaris — это разновидность ОС UNIX, разработанная в Sun Microsystems. Основанная на System V Release 4, она совершенствовалась инженерами из Sun Microsystems более 10 лет. Это единственный коммерчески успешный потомок SVR4, формально сертифицированный как UNIX-система.

4.2.7 Прочие версии UNIX

- AIX, версия UNIX от IBM
- HP-UX, версия UNIX от Hewlett-Packard
- IRIX, UNIX-система, распространяемая компанией Silicon Graphics
- UnixWare, версия UNIX, которая происходит от SVR4 и продается корпорацией SCO

4.3 Пределы ISO C

Все пределы, определяемые стандартом ISO C, являются пределами времени компиляции. Пределы ISO C определены в заголовочном файле `<limits.h>`.

4.4 Пределы POSIX

Стандарт POSIX.1 определяет многочисленные константы, связанные с предельными значениями. Пределы POSIX также определены в `<limits.h>`, какие-то пределы определены всегда, каки-то только при опрееленных условиях.

4.5 Функции sysconf, pathconf и fpathconf

Функции sysconf, pathconf и fpathconf – позволяют получить значение пределов на этапе выполнения программы. Все три функции возвращают значение соответствующего предела в случае успеха или –1 в случае ошибки.

```
#include <unistd.h>
long sysconf(int name);
long pathconf(const char *pathname, int name);
long fpathconf(int fd, int name);
```

Различие между двумя последними функциями состоит в том, что первая получает в аргументе строку пути к файлу, а вторая – файловый дескриптор. Пример использования `sysconf()`:

```
void print_sysconf(const char *con_name, int name)
{
    long val;

    errno = 0;
    if ((val = sysconf(name)) < 0) {
        if (errno != 0) {
            if (errno == EINVAL) {
                printf("%s is not supported\n", con_name);
            } else {
                perror("Sysconf error: ");
            }
        } else {
            printf("%s is unlimited\n", con_name);
        }

    } else {
        printf("%s = %li\n", con_name, val);
    }
}

int main(int argc, char* argv[])
{
#ifdef  _SC_ARG_MAX
    print_sysconf("ARG_MAX", _SC_ARG_MAX);
#else
    printf("_SC_ARG_MAX is not defined.\n");
#endif
}
```

4.6 Элементарные системные типы данных

Заголовочный файл <sys/types.h> определяет ряд зависящих от реализации типов данных, которые называются элементарными системными типами данных:

- clock_t – Счетчик тактов системных часов (время работы процесса, раздел 1.10)
- comp_t – Счетчик тактов в упакованном виде
- dev_t – Номер устройства
- fd_set – Набор файловых дескрипторов
- fpos_t – Позиция в файле
- gid_t – Числовой идентификатор группы
- ino_t – Номер индексного узла (i-node)
- mode_t – Тип файла, режим создания файла
- nlink_t – Счетчик ссылок для записей в каталоге
- off_t – Размер файла и смещение в файле (со знаком)
- pid_t – Числовой идентификатор процесса и идентификатор группы процессов
- pthread_t – Числовой идентификатор потока выполнения
- ptrdiff_t – Разность двух указателей (со знаком)
- rlim_t – Предельное значение для ресурса
- sig_atomic_t – Тип данных, доступ к которому может выполняться атомарно
- sigset_t – Набор сигналов
- size_t – Размер объекта (например, строки) (без знака)
- ssize_t – Возвращаемый функциями результат, представляющий счетчик байтов (со знаком)
- time_t – Счетчик секунд календарного времени
- uid_t – Числовой идентификатор пользователя
- wchar_t – Может представлять символы в любой кодировке

5 Файловый ввод-вывод: дескрипторы файлов, функция open(), функция creat(), функция close().

Большинство операций файлового ввода/вывода в UNIX можно выполнить с помощью всего пяти функций: `open`, `read`, `write`, `lseek` и `close`.

5.1 Дескрипторы файлов

Все открытые файлы представлены в ядре файловыми дескрипторами. Файловый дескриптор – это неотрицательное целое число. Когда процесс открывает существующий файл или создает новый, ядро возвращает ему файловый дескриптор.
В соответствии с соглашениями командные оболочки UNIX ассоциируют файловый дескриптор 0 (`STDIN_FILENO`) с устройством стандартного ввода процесса, 1 (`STDOUT_FILENO`) – с устройством стандартного вывода и 2 (`STDERR_FILENO`) – с устройством стандартного вывода сообщений об ошибках.
Под файловые дескрипторы отводится диапазон чисел от `0` до `OPEN_MAX-1`.

5.2 Функции open и openat

Файл создается (если не существует) или открывается функцией open или openat.

```
#include <fcntl.h>
int open(const char *path, int oflag, ... /* mode_t mode */);
int openat(int fd, const char *path, int oflag, ... /* mode_t mode */);
```

Последний аргумент обозначен многоточием (...), таким способом стандарт ISO C указывает, что количество остальных аргументов и их типы могут варьироваться.
`path` – имя файла
`oflag` – комбинация флагов (целых чисел) `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_EXEC`, `O_APPEND`, `O_CREAT` и др. при помощи объедения ИЛИ (OR, `|`)
Функции open и openat гарантируют, что возвращаемый ими дескриптор файла будет иметь наименьшее неиспользуемое положительное числовое значение.

5.2.1 Параметр fd в функции openat

Возможны три разных варианта:

- Параметр `path` содержит строку абсолютного пути. В этом случае параметр `fd` игнорируется и `openat` действует подобно функции `open`.

- Параметр `path` содержит строку относительного пути, а параметр `fd` содержит дескриптор файла, определяющего местоположение в файловой системе, откуда будет откладываться относительный путь.
- Параметр `path` содержит строку относительного пути, а параметр `fd` содержит специальное значение `AT_FDCWD`. В этом случае путь начинает откладыватьсяот текущего рабочего каталога и функция `openat` действует подобно функции `open`.

5.3 Функция creat

Новый файл можно также создать с помощью функции creat.

```
#include <fcntl.h>
int creat(const char *path, mode_t mode);
```

Эта функция эквивалентна вызову

```
open(path, O_WRONLY | O_CREAT | O_TRUNC, mode);
```

5.4 Функция close

Закрытие открытого файла производится вызовом функции close.

```
#include <unistd.h>
int close(int fd);
```

Закрытие файла приводит также к снятию любых блокировок, которые могли быть наложены процессом. При завершении процесса все открытые им файлы автоматически закрываются ядром.

6 Файловый ввод-вывод: Функция lseek(), функция read(), функция write()

6.1 Функция lseek

С любым открытым файлом связано такое понятие, как текущая позиция в файле. Как правило, это неотрицательное целое число, которым выражается смещение в байтах от начала файла. По умолчанию при открытии файла текущая позиция инициализируется числом 0, если не был установлен флаг `O_APPEND`. Явное изменение текущей позиции в файле выполняется с помощью функции lseek.

```
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

Интерпретация аргумента `offset` зависит от значения аргумента `whence`:

- `SEEK_SET` – offset интерпретируется как смещение от начала файла
- `SEEK_CUR` – offset интерпретируется как смещение от текущей позиции в файле. В этом случае offset может принимать и положительные, и отрицательные значения.
- `SEEK_END` – offset интерпретируется как смещение от конца файла. В этом случае offset может принимать и положительные, и отрицательные значения

Поскольку в случае успеха функция lseek возвращает новую текущую позицию в файле, в аргументе offset можно передать 0, чтобы узнать текущую позицию:

```
1 off_t currpos;
2 currpos = lseek(fd, 0, SEEK_CUR);
```

Таким же способом можно определить, поддерживается ли свободное перемещение текущей позиции файла.

6.2 Функция read

Чтение данных из открытого файла производится функцией read.

```
1 #include <unistd.h>
2 ssize_t read(int fd, void *buf, size_t nbytes);
```

Возвращает количество прочитанных байтов, 0 – если достигнут конец файла, -1 – в случае ошибки.

6.3 Функция write

Запись данных в открытый файл производится функцией write.

```
1 #include <unistd.h>
2 #include <unistd.h>
3 ssize_t write(int fd, const void *buf, size_t nbytes);
```

Возвращает количество записанных байтов в случае успеха, -1 – в случае ошибки.

7 Файловый ввод-вывод: эффективность операций ввода-вывода

Пример копирования из стандартного ввода в стандартный вывод

```c
#include "apue.h"
#define BUFFSIZE 4096

int main()
{
    int n;
    char buf[BUFFSIZE];

    while((n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
        if(write(STDOUT_FILENO, buf, n) != n)
            err_sys("write error!");

    if(n < 0)
        err_sys("read error!");

    exit(0);
}
```

Размера буфера `buf` влияет на производительность, т.к. переход ядра в привилегированный режим для чтения и записи занимает определенное время.
Производительность операции чтения с различными размерами буфера в ОС Linux (пример из учебника):

BUFFSIZE	Пользовательское время (секунды)	Системное время (секунды)	Общее время (секунды)	Количество циклов
1	20,03	117,50	138,73	516 581 760
2	9,69	58,76	68,60	258 290 880
4	4,60	36,47	41,27	129 145 440
8	2,47	15,44	18,38	64 572 720
16	1,07	7,93	9,38	32 286 360
32	0,56	4,51	8,82	16 143 180
64	0,34	2,72	8,66	8 071 590
128	0,34	1,84	8,69	4 035 795
256	0,15	1,30	8,69	2 017 898
512	0,09	0,95	8,63	1 008 949
1024	0,02	0,78	8,58	504 475
2048	0,04	0,66	8,68	252 238
4096	0,03	0,58	8,62	126 119
8192	0,00	0,54	8,52	63 060
16 384	0,01	0,56	8,69	31 530
32 768	0,00	0,56	8,51	15 765
65 536	0,01	0,56	9,12	7883
131 072	0,00	0,58	9,08	3942
262 144	0,00	0,60	8,70	1971
524 288	0,01	0,58	8,58	986

Наилучшее системное время – при размере буфера равном размеру дискового блока (`st_blksize`).
Для повышения производительности большинство файловых систем поддерживают опережающее чтение. Обнаружив ряд последовательных операций чтения, система пытается прочитать больший объем данных, чем было запрошено приложением, предполагая, что программа вскоре продолжит чтение.

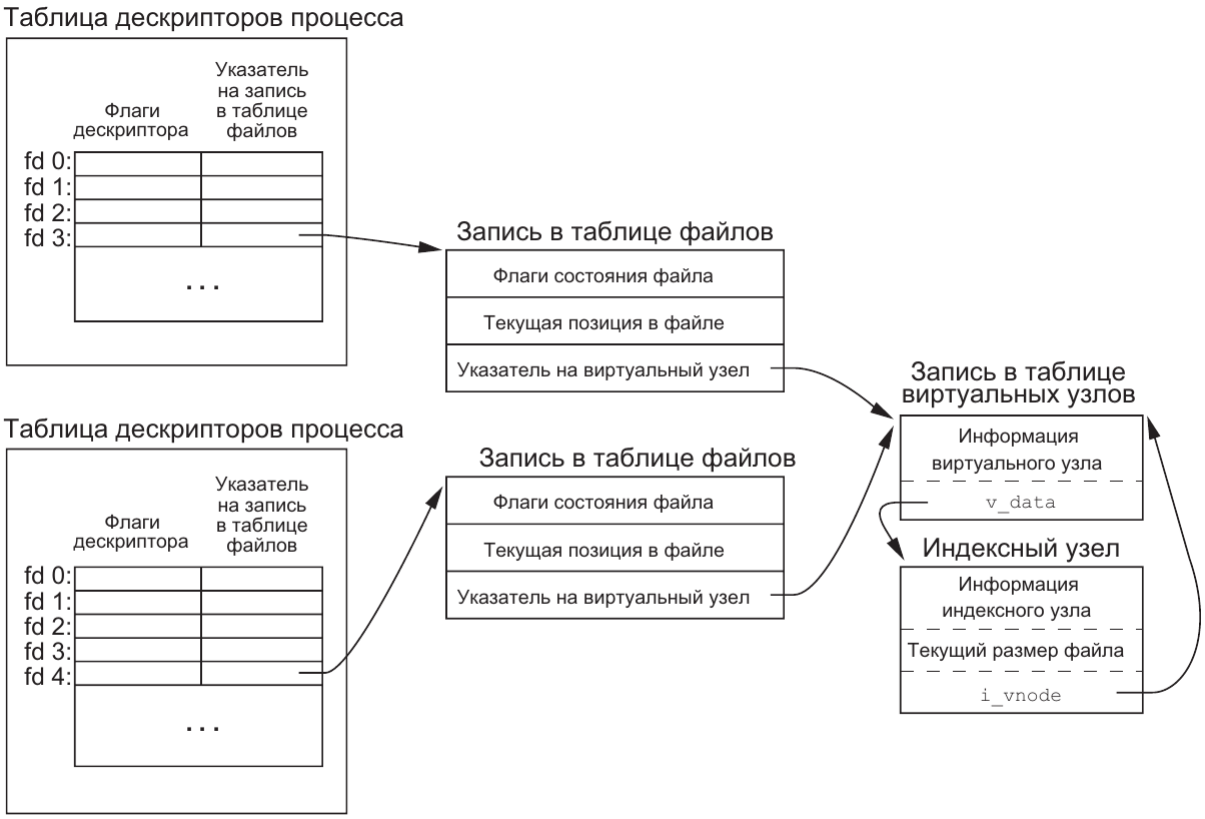
8 Файловый ввод-вывод: совместное использование файлов, атомарные операции, функции dup() и dup2()

8.1 Совместное использование файлов

ОС UNIX поддерживает совместное использование открытых файлов несколькими процессами.
Ядро использует три структуры данных для представления открытого файла, а отношения между ними определяют взаимовлияние процессов при совместном использовании файлов:

1. Каждому процессу соответствует запись в таблице процессов. С каждой записью в таблице процессов связана таблица открытых файловых дескрипторов. Для каждого дескриптора хранится следующая информация:
   - флаги дескриптора
   - указатель на запись в таблице файлов
2. Все открытые файлы представлены в ядре таблицей файлов. Каждая запись в таблице содержит:
   - флаги состояния файла
   - текущая позиция в файле
   - указатель на запись в таблице виртуальных узлов (v-node)
3. Каждому открытому файлу (или устройству) соответствует структура виртуального узла (v-node) с информацией о типе файла и указатели для функций, работающих с файлом

В Linux отсутствует понятие виртуальных узлов (vnode). Вместо него используются структуры индексных узлов (inode). Хотя реализация их различна, концептуально они представляют собой одно и то же. В обоих случаях индексный узел хранит информацию, специфичную для конкретной файловой системы.
Ситуация, когда два независимых процесса открывают один и тот же файл:



Каждый процесс, открывающий файл, создает собственную запись в таблице файлов, но двум этим записям соответствует единственная запись в таблице виртуальных узлов.

8.2 Атомарные операции

8.2.1 Добавление данных в конец файла

ОС UNIX выполняет операцию `write` атомарно (только один процесс может получить доступ), если указан флаг `O_APPEND`.

8.2.2 Функции pread и pwrite

`pread` и `pwrite` – это расширения XSI для стандарта Single UNIX Specification. Эти расширения позволяют процессам атомарно выполнять ввода/вывода.

```
#include <unistd.h>
ssize_t pread(int fd, void *buf, size_t nbytes, off_t offset);
ssize_t pwrite(int fd, const void *buf, size_t nbytes, off_t offset);
```

Вызов функции pread эквивалентен двум последовательным вызовам функций `lseek` и `read` со следующими отличиями:

- При использовании pread нет возможности прервать выполнение этих двух операций
- Значение текущей позиции в файле не изменяется

Вызов функции pwrite эквивалентен двум последовательным вызовам функций `lseek` и `read` с аналогичными отличиями.

8.2.3 Создание файла

При указании флага `O_CREAT` функция `open()` будет атомарно проверять существование и создавать файл.

8.3 Функции dup и dup2

Дубликат дескриптора существующего файла можно создать с помощью одной из следующих функций:

```
#include <unistd.h>
int dup(int fd);
int dup2(int fd, int fd2);
```

Возвращают новый дескриптор файла или –1 — в случае ошибки.
Функция dup гарантирует, что возвращаемый ею новый файловый дескриптор будет иметь наименьшее возможное значение.
Вызывая функцию dup2, мы указываем значение нового дескриптора в аргументе fd2. Если дескриптор fd2 перед вызовом функции уже был открыт, он предварительно закрывается.
Если значения аргументов fd и fd2 равны, функция dup2 вернет дескриптор fd2, не закрывая его.

9 Файловый ввод-вывод: функции sync(), fsync(), fdatasync(), fcntl(), ioctl(), /dev/fd

9.1 Функции sync, fsync и fdatasync

Традиционные реализации UNIX имеют в своем распоряжении буферный кэш, или кэш страниц, через который выполняется большинство дисковых операций ввода/вывода. При записи данных в файл они, как правило, сначала помещаются ядром в один из буферов, а затем ставятся в очередь для записи на диск в более позднее время. Этот прием называется отложенной записью.
Ядро обычно записывает отложенные данные на диск, когда возникает необходимость в повторном использовании буфера. Для синхронизации файловой

системы на диске и содержимого буферного кэша существуют функции `sync`, `fsync` и `fdatasync`:

```
#include <unistd.h>
#include <unistd.h>
int fsync(int fd);
int fdatasync(int fd);
void sync(void);
```

Функция sync просто ставит все измененные блоки буферов в очередь для записи и возвращает управление — она не ждет, пока данные фактически запишутся на диск.
Функция sync, как правило, вызывается периодически (обычно каждые 30 секунд) из системного демона, часто называемого `update`.
Функция fsync применяется только к одному файлу, который определяется файловым дескриптором fd, кроме того, она ожидает завершения физической записи данных на диск, прежде чем вернуть управление.
Функция fdatasync похожа на функцию fsync, но воздействует только на содержимое файла. (При использовании fsync также синхронно обновляются атрибуты файла).

9.2 Функция fcntl

С помощью функции fcntl можно изменять свойства уже открытого файла.

```
#include <fcntl.h>
int fcntl(int fd, int cmd, ... /* int arg */ );
```

Функция fcntl используется в пяти различных случаях:

1. Создание дубликата существующего дескриптора (cmd = `F_DUPFD` или `F_DUPFD_CLOEXEC`)
2. Получение/установка флагов дескриптора (cmd = `F_GETFD` или `F_SETFD`)
3. Получение/установка флагов состояния файла (cmd = `F_GETFL` или `F_SETFL`)
4. Проверка/установка владельца для асинхронных операций ввода/вывода (cmd = `F_GETOWN` или `F_SETOWN`)
5. Получение/установка блокировки на отдельную запись в файле (cmd = `F_GETLK`, `F_SETLK` или `F_SETLKW`)

9.3 Функция ioctl

Функция `ioctl` всегда была универсальным инструментом ввода/вывода. Все, что невозможно выразить с помощью функций, описанных в этой главе, как

правило, делается с помощью `ioctl`. Возможности этой функции чаще всего использовались в операциях терминального ввода/вывода.

```
#include <unistd.h>     /* System V */
#include <sys/ioctl.h> /* BSD и Linux */
int ioctl(int fd, int request, ...);
```

9.4 /dev/fd

В современных операционных системах имеется каталог `/dev/fd`, в котором находятся файлы с именами 0, 1, 2 и т. д. Открытие файла `/dev/fd/n` эквивалентно созданию дубликата дескриптора с номером n, при условии, что дескриптор n открыт.
В некоторых системах имеются файлы `/dev/stdin`, `/dev/stdout` и `/dev/stderr`, эквивалентные файлам `/dev/fd/0`, `/dev/fd/1` и `/dev/fd/2` соответственно.

10 Файлы и каталоги: функции stat(), fstat(), lstat(), содержимое struct stat.

Функции `stat()`, `fstat()`, `lstat()` возвращают информацию об указанном файле в структуру `stat`. В случае успеха возвращается ноль. При ошибке возвращается -1, а переменной errno присваивается номер ошибки

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

int stat(const char *file_name, struct stat *buf);
int fstat(int filedes, struct stat *buf);
int lstat(const char *file_name, struct stat *buf);
```

`stat()` возвращает информацию о файле `file_name`.
`lstat()` идентична `stat()`, но в случае символьных сылок она возвращает информацию о самой ссылке, а не о файле.
`fstat()` идентична `stat()`, только возвращается информация об открытом файле, на который указывает `filedes` (дескриптор файла).

Структура `stat` содержит следующие поля:

```
struct stat {
    dev_t     st_dev;     // устройство
    ino_t     st_ino;     // inode
    mode_t    st_mode;    // режим доступа
    nlink_t   st_nlink;   // количество жестких ссылок
    uid_t     st_uid;     // идентификатор пользователя-владельца
    gid_t     st_gid;     // идентификатор группы-владельца
```

```
8     dev_t          st_rdev;      // тип устройства
9                                  // (если это устройство)
10    off_t          st_size;      // общий размер в байтах
11    blksize_t      st_blksize;   // размер блока ввода-вывода
12                                 // в файловой системе
13    blkcnt_t       st_blocks;    // количество выделенных блоков
14    time_t         st_atime;     // время последнего доступа
15    time_t         st_mtime;     // время последней модификации
16    time_t         st_ctime;     // время последнего изменения
17 };
```

11 Файлы и каталоги: типы файлов, права доступа к файлу, функция umask().

11.1 Типы файлов

Большинство файлов в UNIX являются либо обычными файлами, либо каталогами, но есть и другие типы файлов:

- Обычный файл – наиболее распространенный тип файлов, хранящих данные в некотором виде. Ядро UNIX не делает различий между текстовыми и двоичными файлами. Интерпретация содержимого файла полностью зависит от прикладной программы, обрабатывающей файл.
  Одно из наиболее известных исключений из этого правила – выполняемые файлы.

- Файл каталога. Файлы этого типа содержат имена других файлов и ссылки на информацию о них. Любой процесс, обладающий правом на чтение каталога, может прочитать его содержимое, но только ядро обладает правом на запись непосредственно в файл каталога.

- Специальный файл блочного устройства. Этот тип файлов обеспечивает буферизованный ввод/вывод фиксированными блоками для таких устройств, как жесткие диски.

- Специальный файл символьного устройства. Этот тип файлов обеспечивает небуферизованный ввод/вывод для устройств с переменным размером блока. Все устройства в системе являются либо специальными файлами блочных устройств, либо специальными файлами символьных устройств.

- FIFO, или именованный канал. Этот тип файлов используется для организации обмена информацией между процессами.

- Сокет. Этот тип файлов используется для обмена информацией между процессами через сетевые соединения. Сокеты можно также применять для обмена информацией между процессами на одной и той же машине.
- Символическая ссылка. Файлы этого типа являются ссылками на другие файлы.

Тип файла хранится в поле `st_mode` структуры `stat`. Определить тип файла можно с помощью макроопределений:

- `S_ISREG()` – Обычный файл
- `S_ISDIR()` – Каталог
- `S_ISCHR()` – Специальный файл символьного устройства
- `S_ISBLK()` – Специальный файл блочного устройства
- `S_ISFIFO()` – Канал
- `S_ISLNK()` – Символическая ссылка
- `S_ISSOCK()` – Сокет

11.2 Права доступа к файлу

Поле `st_mode` содержит также биты прав доступа к файлу.
Права доступа к файлу определяются девятью битами. Пример:
`110 100 100` = `6 4 4`
`rwx rwx rwx`
`rw- r-- r--` – `rw, read-write` для владельца, `r, read` для группы и остальных. `x, execute` – не разрешено
Биты прав доступа из файла <sys/stat.h>:
Пользователь, владелец:

- `S_IRUSR`
- `S_IWUSR`
- `S_IXUSR`

Группа:

- `S_IRGRP`
- `S_IWGRP`

- `S_IXGRP`

Остальные:

- `S_IROTH`
- `S_IWOTH`
- `S_IXOTH`

11.3 Функция umask

Функция `umask()` устанавливает маску режима создания файлов для процесса и возвращает предыдущее значение.

```
#include <sys/stat.h>
mode_t umask(mode_t cmask);
```

Аргумент `cmask` – это набор констант из списка выше объеденённых при помощи ИЛИ (OR, `|`)

12 Файлы и каталоги: функции chmod(), fchmod(), chown(), fchown(), lchown().

12.1 Функции chmod, fchmod

Функции chmod, fchmod позволяют изменять права доступа к существующим файлам.

```
#include <sys/stat.h>
int chmod(const char *pathname, mode_t mode);
int fchmod(int fd, mode_t mode);
```

12.2 Функции chown, fchown, lchown

Функции семейства chown позволяют изменять идентификаторы пользователя и группы файла, но если в аргументе owner или group передается –1, соответствующий идентификатор не изменяется.

```
#include <unistd.h>
int chown(const char *pathname, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
int lchown(const char *pathname, uid_t owner, gid_t group);
```

`lchown` – изменяет владельца символической ссылки, а не файла.

13 Файлы и каталоги: размер файла, дырки в файлах, усечение файлов, файловые системы, функции link(), unlink(), remove(), rename().

13.1 Размер файла

Поле `st_size` структуры stat содержит размер файла в байтах.
Обычные файлы могут иметь размер, равный нулю. В этом случае первая же операция чтения вернет признак конца файла. Для каталогов размер файла обычно кратен некоторому числу, такому как 16 или 512.
Поле `st_blocks` определяет фактическое количество 512-байтных блоков, занимаемых файлом.

13.2 Дырки в файлах

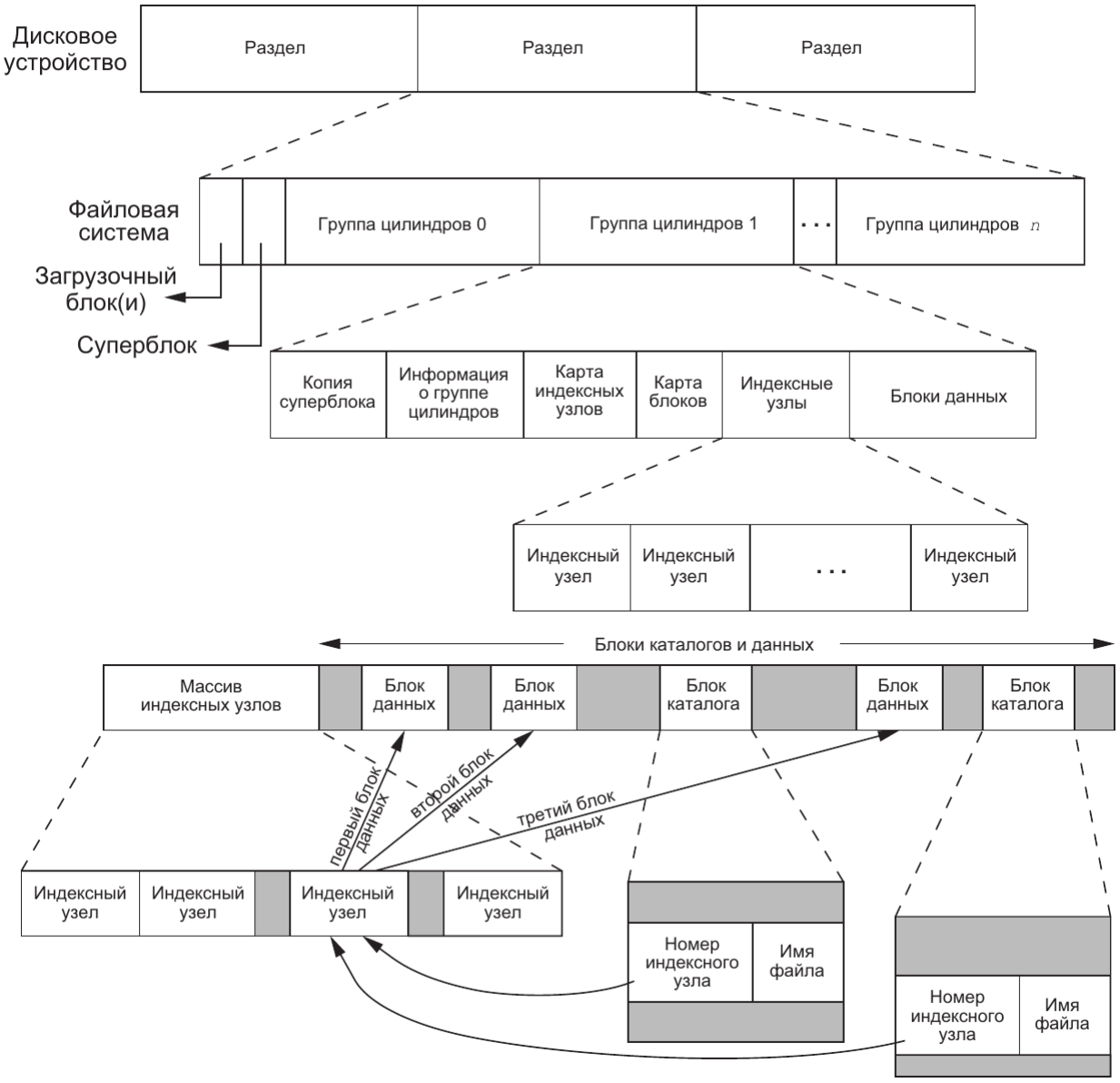
Обычные файлы могут содержать «дырки». Дырки создаются в результате переноса текущей позиции за пределы файла и последующей записи некоторых данных.
Если скопировать файл с «дырками», например, с помощью утилиты `cat`, дырки будут скопированы как обычные байты данных со значением 0.
Чтобы узнать реальный объем занимаемый файлом на диске необходимо использовать поле `st_blocks`, т.е. `real_size = st_blocks * 512`.

13.3 Усечение файлов

Иногда возникает необходимость отсечь некоторые данные, расположенные в конце файла. Усечение размера файла до нуля, которое осуществляется при использовании флага `O_TRUNC` в вызове функции `open`, есть частный случай усечения файла.
Эти две функции усекают существующий файл до размера, определяемого аргументом `length`:

```
1 #include <unistd.h>
2 int truncate(const char *pathname, off_t length);
3 int ftruncate(int fd, off_t length);
```

13.4 Файловые системы



13.4.1 link, unlink

На индексный узел любого файла могут указывать несколько каталожных записей. Такие ссылки создаются с помощью функции `link`:

```
1 #include <unistd.h>
2 int link(const char *existingpath, const char *newpath);
```

Удаление записей из каталога производится с помощью функции `unlink`:

```c
#include <unistd.h>
int unlink(const char *pathname);
```

13.4.2 remove

Удалить жесткую ссылку на файл или каталог можно также с помощью функции `remove`. Для файлов функция remove абсолютно идентична функции `unlink`, для каталогов — функции `rmdir`:

```c
#include <stdio.h>
int remove(const char *pathname);
```

13.4.3 rename

Для переименования файла или каталога используется функция `rename`:

```c
#include <stdio.h>
int rename(const char *oldname, const char *newname);
```

14 Файлы и каталоги: символические ссылки, функции symlink() и readlink().

14.1 Символические ссылки

Символическая ссылка — это косвенная ссылка на файл в отличие от жесткой ссылки, которая является прямым указателем на индексный узел файла. Символические ссылки придуманы с целью обойти ограничения, присущие жестким ссылкам:

- Жесткие ссылки обычно требуют, чтобы ссылка и файл размещались в одной файловой системе
- Только суперпользователь имеет право создавать жесткие ссылки на каталоги (если это поддерживается файловой системой)

По сути символическая ссылка – это буквально ссылка на путь к файлу. Например:
Есть файл: `/home/user/code/qwe.c`
Пусть символическая ссылка `symlink.c` на это файл размещена в `/home/user/somedir`, тогда путь `/home/user/somedir/symlink.c` ссылается на путь `/home/user/code/qwe.c`

14.2 symlink()

Символические ссылки создаются с помощью функции `symlink`:

```
#include <unistd.h>
int symlink(const char *actualpath, const char *sympath);
```

14.3 readlink()

Поскольку функция open следует по символическим ссылкам, нам необходим инструмент, с помощью которого можно было бы открыть саму символическую ссылку, чтобы прочитать имя файла, на который она ссылается. Эти действия выполняют функции readlink:

```
#include <unistd.h>
ssize_t readlink(const char *restrict pathname, char *restrict buf,
    size_t bufsize);
```

15 Файлы и каталоги: временные характеристики файлов, функция utime().

15.1 Временные характеристики файлов

Поля структуры `stat`:

- `st_atim` – Время последнего доступа к содержимому файла (`read`)
- `st_mtim` – Время последнего изменения содержимого файла (`write`)
- `st_ctim` – Время последнего изменения статуса индексного узла (i-node) (`chmod`, `chown`)

15.2 utime()

Функция `utime()` изменяет время создания (или последней модификации) файла, чье имя определяется указателем `fname`:

```
#include <utime.h>
int utime(char *fname, struct utimbuf *t);
```

Новое время задается структурой, на которую указывает параметр `t`. Структура `utimbuf` определена следующим образом:

```
struct utimbuf {
    time_t actime;
    time_t modtime;
}
```

16 Файлы и каталоги: функции mkdir() и rmdir(), чтение каталогов, функции chdir(), fchdir(), getcwd().

16.1 mkdir()

Создание каталогов производится с помощью функции `mkdir`:

```
#include <sys/stat.h>
int mkdir(const char *pathname, mode_t mode);
```

16.2 rmdir()

Удаление пустого каталога производится с помощью функции `rmdir`. Пустым называется каталог, который содержит только две записи: «точка» и «точка-точка»

```
#include <unistd.h>
int rmdir(const char *pathname);
```

16.3 Чтение каталогов

Прочитать информацию из файла каталога может любой, кто имеет право на чтение этого каталога. Но только ядро может выполнять запись в каталоги, благодаря чему обеспечивается сохранность файловой системы.

16.4 Рабочий каталог

Для каждого процесса определен текущий рабочий каталог. Относительно этого каталога вычисляются все относительные пути (то есть пути, которые не начинаются с символа слеша)

16.4.1 chdir(), fchdir()

Процесс может изменить текущий рабочий каталог вызовом функции `chdir()` или `fchdir()`:

```
#include <unistd.h>
int chdir(const char *pathname);
int fchdir(int fd);
```

Получить текущий рабочий каталог можно при помощи функции `getcwd()`:

```
#include <unistd.h>
char *getcwd(char *buf, size_t size);
```

Возвращает указатель на `buf` в случае успеха, `NULL` — в случае ошибки

17 Стандартная библиотека ввода-вывода: потоки и объекты FILE, стандартные потоки ввода, вывода и сообщений об ошибках, буферизация.

Стандартная библиотека ввода/вывода сама производит размещение буферов и выполняет операции ввода/вывода блоками оптимального размера, что избавляет от необходимости задумываться о правильности выбора.

17.1 Потоки и объекты FILE

При обсуждении стандартной библиотеки ввода/вывода мы будем отталкиваться от термина поток ввода/вывода, или просто поток, НЕ дискриптор файла.
Открывая или создавая файл средствами стандартной библиотеки ввода/вывода, мы говорим, что связали поток с файлом.

17.2 Стандартные потоки ввода, вывода и сообщений об ошибках

Для любого процесса автоматически создается три предопределенных потока: стандартный поток ввода, стандартный поток вывода и стандартный поток сообщений об ошибках. Эти потоки связаны с теми же файлами, что и дескрипторы `STDIN_FILENO`, `STDOUT_FILENO` и `STDERR_FILENO`.
Эти три потока доступны посредством предопределенных указателей на файлы `stdin`, `stdout` и `stderr`. Определения файловых указателей находятся в заголовочном файле `<stdio.h>`.

17.3 Буферизация

Поддержка буферизации в стандартной библиотеке ввода/вывода реализована с целью уменьшить количество обращений к функциям read и write.
Библиотека поддерживает три типа буферизации:

1. Полная буферизация. В этом случае фактический ввод/вывод осуществляется, только когда будет заполнен стандартный буфер ввода/вывода. Обычно стандартная библиотека ввода/вывода использует полную буферизацию для файлов на диске.

2. Построчная буферизация. В этом случае фактический ввод/вывод осуществляется, когда в потоке встречается символ перевода строки. Построч-

ная буферизация обычно используется для потоков, связанных с терминальными устройствами, например для стандартного ввода и стандартного вывода.

3. Отсутствие буферизации. Стандартная библиотека ввода/вывода не буферизует операции с символами

18 Стандартная библиотека ввода-вывода: открытие потока, чтение из потока и запись в поток, функции ввода, функции вывода.

18.1 Открытие потока

Функции fopen, freopen и fdopen открывают стандартный поток ввода/вывода:

```
#include <stdio.h>
FILE *fopen(const char *restrict pathname, const char *restrict type);
FILE *freopen(const char *restrict pathname, const char *restrict type,
    FILE *restrict fp);
FILE *fdopen(int fd, const char *type);
```

- Функция fopen открывает указанный файл
- Функция freopen открывает указанный файл и связывает его с указанным потоком, предварительно закрывая поток, если он уже был открыт
- Функция fdopen принимает открытый дескриптор файла, полученный вызовом функций open, dup, dup2, fcntl, pipe, socket, socketpair или accept,и связывает его с потоком ввода/вывода. Часто эта функция вызывается с дескриптором, который получен в результате создания канала или сетевого соединения

Возможные значения аргумента type при открытии потока:

- r или rb – Открыть для чтения
- w или wb – Усечь размер файла до 0 или создать и открыть на запись
- a или ab – Открыть для записи в конец файла или создать для записи
- r+, или r+b, или rb+ – Открыть для чтения и для записи
- w+, или w+b, или wb+ – Усечь размер файла до 0 или создать и открыть для чтения и для записи

- a+, или a+b, или ab+ – Открыть или создать для чтения и для записи в конец файла

Закрывается открытый поток с помощью функции `fclose`:

```
#include <stdio.h>
int fclose(FILE *fp);
```

18.2 Чтение из потока и запись в поток

После открытия потока можно выбрать один из трех типов неформатированного ввода/вывода:

- Посимвольный ввод/вывод
- Построчный ввод/вывод
- Прямой ввод/вывод

18.3 Функции ввода

18.3.1 Посимвольный ввод

Три функции позволяют читать по одному символу за одно обращение. Все три возвращают очередной символ в случае успеха, `EOF` — в случае ошибки:

```
#include <stdio.h>
int getc(FILE *fp);
int fgetc(FILE *fp);
int getchar(void);
```

18.3.2 Построчный ввод

```
#include <stdio.h>
char *fgets(char *restrict buf, int n, FILE *restrict fp);
char *gets(char *buf);
```

18.3.3 Прямой ввод (двоичные данные)

Функции прямого ввода/вывода используются для работы с двоичными данными, т.к. они не обрабатывает нулевые байты и спецсимволы как строки.

```
#include <stdio.h>
size_t fread(void *restrict ptr, size_t size, size_t nobj, FILE *
    restrict fp);
```

18.4 Функции вывода

18.4.1 Посимвольный вывод

```
#include <stdio.h>
int putc(int c, FILE *fp);
int fputc(int c, FILE *fp);
int putchar(int c);
```

18.4.2 Построчный вывод

```
#include <stdio.h>
int fputs(const char *restrict str, FILE *restrict fp);
int puts(const char *str);
```

18.4.3 Прямой вывод (двоичные данные)

```
#include <stdio.h>
size_t fwrite(const void *restrict ptr, size_t size, size_t nobj, FILE
    *restrict fp);
```

19 Стандартная библиотека ввода-вывода: эффективность стандартных операций ввода-вывода, позиционирование в потоке.

19.1 Эффективность стандартных функций ввода/вывода

Стандартная библиотека ввода/вывода не намного медленнее, чем прямое обращение к функциям read и write. В большинстве сложных приложений наибольшее количество пользовательского времени уходит на выполнение самого приложения, а не на обращения к стандартным функциям ввода/вывода. Но стандартная библиотека ввода/вывода поддерживает буферизацию.

19.2 Позиционирование в потоке

Существует три способа позиционирования в потоке ввода/вывода:

- С помощью функций `ftell` и `fseek` – позиция в файле представлена как long int

```
    #include <stdio.h>
    long ftell(FILE *fp);
    int fseek(FILE *fp, long offset, int whence);

```

- С помощью функций `ftello` и `fseeko` – позиция в файле представлена как off_t

```
#include <stdio.h>
off_t ftello(FILE *fp);
int fseeko(FILE *fp, off_t offset, int whence);

```

- С помощью функций `fgetpos` и `fsetpos` – позиция в файле представлена как абстрактный тип fpos_t. Этот тип может быть увеличен настолько, насколько это необходимо для представления позиции в файле

```
#include <stdio.h>
int fgetpos(FILE *restrict fp, fpos_t *restrict pos);
int fsetpos(FILE *fp, const fpos_t *pos);

```

20 Стандартная библиотека ввода-вывода: форматированный вывод, форматированный ввод, временные файлы.

20.1 Форматированный ввод/вывод

20.1.1 Форматированный вывод

Форматированный вывод производится с помощью пяти функций из семейства printf.

```
#include <stdio.h>
int printf(const char *restrict format, ...);
int fprintf(FILE *restrict fp, const char *restrict format, ...);
int dprintf(int fd, const char *restrict format, ...);
int sprintf(char *restrict buf, const char *restrict format, ...);
int snprintf(char *restrict buf, size_t n, const char *restrict format,
    ...);
```

20.1.2 Форматированный ввод

Форматированный ввод выполняется с помощью трех функций из семейства scanf.

```
#include <stdio.h>
int scanf(const char *restrict format, ...);
int fscanf(FILE *restrict fp, const char *restrict format, ...);
int sscanf(const char *restrict buf, const char *restrict format, ...);
```

20.2 Временные файлы

Стандарт ISO C определяет две вспомогательные функции для создания временных файлов.

```
#include <stdio.h>
char *tmpnam(char *ptr);
FILE *tmpfile(void);
```

Функция tmpnam генерирует строку с уникальным полным именем файла, не существующего в данный момент в системе. При каждом вызове эта функция генерирует неповторяющиеся имена файлов до `TMP_MAX` раз. Константа `TMP_MAX` определена в файле `<stdio.h>`.

21 Управление процессами: идентификаторы процесса, функция fork(), совместное использование файлов.

21.1 Идентификаторы процесса

Любой процесс обладает уникальным идентификатором процесса, который представляет собой целое положительное число. Поскольку идентификатор процесса – это единственный широко используемый идентификатор, уникальность которого гарантируется системой, он часто присоединяется к другим идентификаторам для придания им уникальности. Например, приложения иногда включают идентификатор процесса в имена файлов, чтобы обеспечить их уникальность.
Но, несмотря на уникальность, идентификаторы процесса могут использоваться многократно. По завершении процесса его идентификатор может использоваться повторно для другого процесса.
Процесс с идентификатором 0 – это, как правило, процесс-планировщик, который часто называют swapper (программа подкачки).
Процесс с идентификатором 1 – это обычно процесс init, который запускается ядром в конце процедуры начальной загрузки.

21.2 Функция fork

Любой процесс может создать новый процесс, вызвав функцию `fork`:

```
#include <unistd.h>
pid_t fork(void);
```

Возвращает 0 в дочернем процессе, идентификатор дочернего процесса – в родительском, -1 – в случае ошибки.
Новый процесс, созданный функцией `fork`, называется дочерним процессом,

или процессом-потомком.
Современные версии UNIX не производят немедленного полного копирования сегмента данных, стека и кучи, потому что часто вслед за вызовом `fork` сразу же следует вызов `exec`

21.3 Совместное использование файлов

Одна из особенностей функции `fork` в том, что она передает дочернему процессу дубликаты всех дескрипторов, открытых в родительском процессе.
Важно заметить, что родительский и дочерний процессы совместно используют текущую позицию в файле.

22 Управление процессами: функция exit(), функции wait() и waitpid().

22.1 Функции exit

Эта функция определена стандартом ISO C, она производит вызов всех функций – обработчиков выхода, зарегистрированных функцией `atexit`, и закрывает все стандартные потоки ввода/вывода.

22.2 Функции wait и waitpid

Когда процесс завершается, обычным образом или аварийно, ядро извещает об этом родительский процесс с помощью сигнала `SIGCHLD`.
Функции `wait` и `waitpid`, вызванные родительским процессом, могут:

- Заблокировать процесс, если все его дочерние процессы продолжают работу
- Сразу же вернуть управление с кодом завершения дочернего процесса, если он уже закончил работу и ожидает, пока родительский процесс заберет код завершения
- Сразу же вернуть управление с признаком ошибки, если у вызвавшего процесса нет ни одного дочернего процесса

Если процесс вызывает wait по получении сигнала SIGCHLD, функция сразу же вернет управление. Но если wait была вызвана в любой произвольный момент времени, она может заблокировать родительский процесс.

```
1 #include <sys/wait.h>
2 pid_t wait(int *statloc);
3 pid_t waitpid(pid_t pid, int *statloc, int options);
```

Функция `waitpid` не ждет первого завершившегося дочернего процесса – можно указать, завершения какого процесса она должна ожидать.
В обеих функциях аргумент `statloc` является указателем на целое число. Если в аргументе передается непустой указатель, по заданному адресу будет записан код завершения дочернего процесса.
Код завершения дочернего процесса обрабатывается различными макроопределениями.

23 Управление процессами: семейство функций exec().

Функция fork часто используется для создания нового процесса, который затем запускает другую программу с помощью одной из функций семейства exec.

```
#include <unistd.h>
int execl(const char *pathname, const char *arg0, ... /* (char *)0 */ );
int execv(const char *pathname, char *const argv[]);
int execle(const char *pathname, const char *arg0, ... /* (char *)0, char *const envp[] */ );
int execve(const char *pathname, char *const argv[], char *const envp[]);
int execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );
int execvp(const char *filename, char *const argv[]);
int fexecve(int fd, char *const argv[], char *const envp[]);
```

Одно из отличий между этими функциями заключается в том, что первые четыре принимают полный путь к файлу, следующие две – только имя файла и последняя – дескриптор файла.
Если функция `execlp` или `execvp` находит выполняемый файл, используя один из префиксов пути, но этот файл не является двоичным выполняемым файлом, сгенерированным редактором связей, функция предположит, что найденный файл является сценарием командной оболочки, и попытается вызывать `/bin/sh` с именем файла в качестве аргумента.

Таблица 8.6. Различия между семью функциями семейства exec

Функция	pathname	filename	fd	Список аргументов	argv[]	environ	envp[]
execl	✓			✓		✓	
execlp		✓		✓		✓	
execle	✓			✓			✓
execv	✓				✓	✓	
execvp		✓			✓	✓	
execve	✓				✓		✓
fexecve			✓		✓		✓
буква в имени		p	f	l	v		e

24 Управление процессами: изменение идентификаторов пользователя и группы, функции setuid(), setgid(), seteuid(), setegid().

24.1 Изменение идентификаторов пользователя и группы

В системе UNIX предоставление привилегий (таких, как возможность изменять текущую дату) и управление доступом к файлам (например, право на чтение или запись) основаны на идентификаторах пользователя и группы.
Когда программе необходимы дополнительные привилегии, чтобы получить доступ к ресурсам, недоступным в настоящее время, она должна изменить свой идентификатор пользователя или группы на идентификатор, который имеет соответствующие привилегии.

24.1.1 setuid, setgid

Изменить реальный и эффективный идентификаторы пользователя можно с помощью функции setuid. Аналогично с помощью функции setgid можно изменить реальный и эффективный идентификаторы группы.

```
1 #include <unistd.h>
2 int setuid(uid_t uid);
3 int setgid(gid_t gid);
```

Изменить реальный идентификатор пользователя может только процесс, обладающий привилегиями суперпользователя.
Эффективный идентификатор пользователя устанавливается функцией exec, только когда файл программы имеет установленный бит set-user-ID.
Функция exec копирует эффективный идентификатор пользователя в сохраненный.

24.1.2 seteuid, setegid

Стандарт POSIX.1 включает еще две функции: seteuid и setegid. Они очень похожи на функции setuid и setgid, но изменяют только эффективный идентификатор пользователя и группы.

```
#include <unistd.h>
int seteuid(uid_t uid);
int setegid(gid_t gid);
```

25 Управление процессами: интерпретируемые файлы, функция system().

25.1 Интерпретируемые файлы

Все современные версии UNIX поддерживают интерпретируемые файлы. Это обычные текстовые файлы, которые начинаются со строки вида:
`#! pathname [optional-argument]`
Например: `#!/bin/bash`
Распознавание интерпретируемых файлов производится ядром в процессе выполнения системного вызова exec. В действительности ядро запускает на выполнение не сам интерпретируемый файл, а программу, указанную в параметре pathname, в первой строке.

25.2 Функция system

Функция system дает удобный способ выполнения команд внутри программы.

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

Например, допустим, что требуется записать строку с датой и временем в некоторый файл:

```
system("date > file");
```

26 Сигналы: концепция сигналов, функция signal(), ненадежные сигналы.

26.1 Концепция сигналов

Прежде всего, каждый сигнал имеет собственное имя. Имена всех сигналов начинаются с последовательности SIG. Например, `SIGABRT` – это сигнал прерывания, который генерируется, когда процесс вызывает функцию `abort`. Сигнал `SIGALRM` генерируется, когда таймер, установленный функцией `alarm`, отмерит указанный промежуток времени.
Все имена сигналов определены как константы с положительными числовыми значениями (номерами сигналов) в заголовочном файле `<signal.h>`.
Сигнала с номером 0 не существует. Функция kill использует номер сигнала 0 в особых случаях. Стандарт POSIX.1 называет такой сигнал null signal (пустой сигнал).
Сигналы являют собой классический пример асинхронных событий. Сигнал может быть передан процессу в любой момент.

26.2 Функция signal

Функция `signal` – это простейший интерфейс к сигналам UNIX.

```
1 #include <signal.h>
2 void (*signal(int signo, void (*func)(int)))(int);
```

Аргумент signo — это имя сигнала.
В качестве аргумента func можно передать (а) константу `SIG_IGN`, или (б) константу `SIG_DFL`, или (в) адрес функции, которая будет вызвана при получении сигнала.

26.3 Ненадежные сигналы

В ранних версиях UNIX (таких, как Version 7) сигналы были ненадежными. То есть сигналы могли теряться: иными словами, процесс мог не получить посланный ему сигнал.
Кроме того, процесс имел весьма ограниченные возможности управления сигналами: он мог либо перехватить сигнал, либо игнорировать его. Иногда может возникнуть потребность заблокировать сигнал, то есть не игнорировать его, а просто отложить посылку сигнала до момента, когда приложение будет готово принять его.

27 Сигналы: прерванные системные вызовы, реентерабельные функции.

27.1 Прерванные системные вызовы

Ранние версии UNIX обладали одним свойством: если процесс, заблокированный в «медленном» системном вызове, перехватывал сигнал, выполнение системного вызова прерывалось. В этом случае системный вызов возвращал признак ошибки с кодом `EINTR` в переменной errno. Так было сделано в предположении, что раз сигнал был послан и процесс перехватил его, возможно, произошло что-то, из-за чего системный вызов должен прервать ожидание и вернуть управление процессу.
Для поддержки такого поведения системные вызовы были разделены на «медленные» и все остальные. Медленные системные вызовы – это такие вызовы, которые могут заблокировать процесс навсегда. В эту категорию попали:

- Операция чтения
- Операция записи
- Операция открытия
- Функция `pause` и функция `wait`
- Некоторые операции функции ioctl
- Некоторые функции межпроцессного взаимодействия

27.2 Реентерабельные функции

Когда процесс обрабатывает перехваченный сигнал, нормальная последовательность выполнения инструкций временно нарушается обработчиком сигнала. После этого процесс продолжает работу, но выполняет инструкции уже в функции-обработчике.
Чтобы изюежать различных коллизий при обработке асинхронных сигналов (например использование `malloc`). Стандарт Single UNIX Specification определяет перечень функций, которые должны обеспечивать безопасность вызова из обработчиков сигналов. Эти функции являются реентерабельными и в стандарте Single UNIX Specification называются безопасными для использования в обработчиках асинхронных сигналов. Помимо обеспечения реентерабельности, они блокируют доставку любых сигналов до своего завершения.

28 Сигналы: функции kill(), raise(), alarm(), pause().

28.1 kill и raise

Функция kill посылает сигнал процессу или группе процессов. Функция raise позволяет процессу послать сигнал себе самому.

```
#include <signal.h>
int kill(pid_t pid, int signo);
int raise(int signo);
```

Вызов `raise(signo);` эквивалентен вызову `kill(getpid(), signo)`;
Для `kill`:

- `pid > 0` – Сигнал посылается процессу с идентификатором pid
- `pid == 0` – Сигнал посылается всем процессам с идентификатором группы процессов, равным идентификатору группы процессов посылающего процесса, которым данный процесс имеет право посылать сигналы
- `pid < 0` – Сигнал посылается всем процессам с идентификатором группы процессов, равным абсолютному значению pid, которым данный процесс имеет право посылать сигналы
- `pid == -1` – Сигнал посылается всем процессам в системе, которым посылающий процесс имеет право посылать сигналы

28.2 Функции alarm и pause

Функция `alarm` позволяет установить таймер, который по истечении заданного времени сгенерирует сигнал `SIGALRM`.

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

Каждый процесс может обладать только одним таким таймером. Если функция `alarm` вызывается до истечения таймера, установленного ранее, она возвращает количество оставшихся секунд, а ранее установленный интервал времени заменяется новым.
По умолчанию сигнал SIGALRM завершает процесс, но большинство приложений перехватывают его.
Функция `pause` приостанавливает вызывающий процесс, пока не будет перехвачен сигнал:

```
#include <unistd.h>
int pause(void);
```

29 Сигналы: надежные сигналы, терминология и семантика, наборы сигналов.

29.1 Надежные сигналы. Терминология и семантика

Мы должны определить некоторые термины, используемые при обсуждении сигналов. Прежде всего для процесса генерируется (или процессу посылается) сигнал, когда происходит некоторое событие.
Когда выполняется действие, предусмотренное для сигнала, мы говорим, что сигнал доставлен процессу.
Процесс может заблокировать доставку сигнала.
Каждый процесс имеет маску сигналов, с помощью которой определяется множество блокируемых сигналов.
Процесс может проверить и изменить маску с помощью функции `sigprocmask`.

29.2 Наборы сигналов

Для представления множества сигналов нам необходим специальный тип данных – набор сигналов. Он используется такими функциями, как `sigprocmask`, чтобы передать ядру набор сигналов, которые должны быть заблокированы.
Стандарт POSIX.1 определяет для этих целей специальный тип `sigset_t`, который может хранить набор сигналов и с которым работают следующие пять функций:

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember(const sigset_t *set, int signo);
```

Функция `sigemptyset` инициализирует пустой набор сигналов, на который указывает аргумент `set`. Функция `sigfillset` инициализирует набор сигналов, в который включены все сигналы. Все приложения должны вызывать функцию `sigemptyset` или `sigfillset`.
Добавление одного сигнала в существующий набор производится функцией `sigaddset`, а удаление сигнала из набора – функцией `sigdelset`.

30 Сигналы: маска сигналов процесса и функция sigprocmask(), функция sigpending(),

30.1 Маска сигналов процесса и функция sigprocmask()

30.1.1 Маска сигналов

Каждый процесс имеет маску сигналов, с помощью которой определяется множество блокируемых сигналов. Ее можно представлять как битовую маску, в которой каждый бит соответствует отдельному сигналу. Если некоторый бит включен, доставка соответствующего ему сигнала блокируется.

30.1.2 sigprocmask()

Процесс может получить текущее значение маски, изменить маску или выполнить сразу обе операции с помощью следующей функции.

```
1 #include <signal.h>
2 int sigprocmask(int how, const sigset_t *restrict set, sigset_t *
    restrict oset);
```

Прежде всего, если в аргументе `oset` передается непустой указатель, через него возвращается текущая маска сигналов процесса.
Далее, если в аргументе `set` передается непустой указатель, то значение аргумента `how` определяет, как должна измениться маска сигналов.
Значения `how`:

- SIG_BLOCK – Новая маска сигналов является объединением текущей маски сигналов с набором, на который указывает аргумент set
- SIG_UNBLOCK – Новая маска сигналов является пересечением текущей маски сигналов с набором, на который указывает аргумент set
- SIG_SETMASK – Новая маска сигналов в аргументе set замещает текущую маску сигналов

Если в аргументе set передается пустой указатель, маска сигналов процесса не изменяется и значение аргумента how игнорируется.

30.2 Функция sigpending

Функция sigpending возвращает набор сигналов, доставка которых заблокирована и которые в данный момент ожидают обработки. Набор сигналов возвращается через аргумент set:

```
1 #include <signal.h>
2 int sigpending(sigset_t *set);
```

31 Сигналы: функция sigaction().

Функция sigaction позволяет проверить действие, связанное с определенным сигналом, изменить его или выполнить обе эти операции. Эта функция служит заменой функции signal из ранних версий UNIX.

```
#include <signal.h>
int sigaction(int signo, const struct sigaction *restrict act, struct
    sigaction *restrict oact);
```

Через аргумент signo передается номер сигнала, диспозицию которого требуется получить или изменить. Если в аргументе act передается непустой указатель, диспозиция сигнала изменяется. Если в аргументе oact передается непустой указатель, функция возвращает предыдущее значение диспозиции сигнала, помещая его по указанному в oact адресу.
Структура sigaction:

```
struct sigaction {
    void (*sa_handler)(int); // адрес функции — обработчика сигнала,
                             // или SIG_IGN, или SIG_DFL
    sigset_t sa_mask;        // дополнительные блокируемые сигналы
    int sa_flags;            // флаги

    // альтернативный обработчик сигнала
    void (*sa_sigaction)(int, siginfo_t *, void *);
}
```

Если при изменении диспозиции сигнала поле sa_handler содержит адрес функции-обработчика (а не константы SIG_IGN или SIG_DEL), поле sa_mask определяет набор сигналов, которые будут добавлены к маске сигналов процесса перед вызовом функции-обработчика.

32 Сигналы: функция sigsuspend().

`sigsuspend` временно блокирует сигнал. Если процессу придёт времеено заблокированный сигнал, то он будет «висеть в очереди» пока процесс не разблокирует его:

```
#include <signal.h>
int sigsuspend(const sigset_t *sigmask);
```