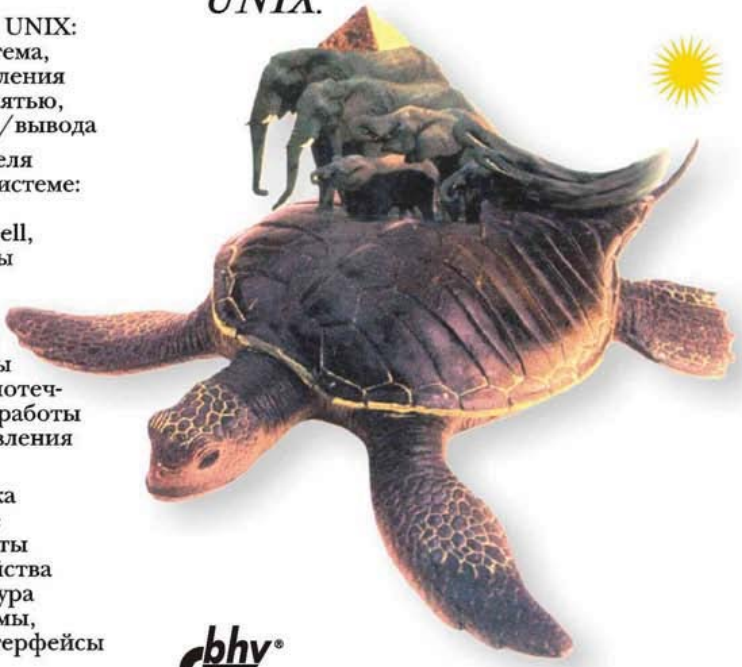
Андрей РОБАЧЕВСКИЙ, Сергей НЕМНЮГИН, Ольга СТЕСИК

ОПЕРАЦИОННАЯ СИСТЕМА UNIX®

2-Е ИЗДАНИЕ

Принципы организации, идеология и архитектура, пользовательский и программный интерфейсы — все, что объединяет различные версии операционной системы под общим названием UNIX.

- Архитектура ядра UNIX: файловая подсистема, подсистема управления процессами и памятью, подсистема ввода/вывода
- Работа пользователя в операционной системе: командный интерпретатор shell, основные команды и утилиты
- Программный интерфейс UNIX: системные вызовы и основные библиотечные функции для работы с файлами и управления процессами
- Сетевая поддержка в UNIX: описание и принципы работы протоколов семейства TCP/IP, архитектура сетевой подсистемы, программные интерфейсы сокетов и TLI



bhv®

УДК 681.3.06
ББК 32.973.26-018.2
Р58

Робачевский А. М., Немнюгин С. А., Стесик О. Л.

Р58 Операционная система UNIX. — 2-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2005. — 656 с.: ил.

ISBN 5-94157-538-6

Книга является вторым изданием рекомендованного Министерством общего и профессионального образования России одноименного учебного пособия.

Даны основы организации, идеологии и архитектуры, объединяющие различные версии UNIX. Рассматриваются: архитектура ядра (подсистемы ввода/вывода, управления памятью и процессами, а также файловая подсистема), программный интерфейс (системные вызовы и основные библиотечные функции), пользовательская среда (командный интерпретатор shell, основные команды и утилиты) и сетевая поддержка (протоколы семейства TCP/IP, архитектура сетевой подсистемы, программные интерфейсы сокетов и TLI).

Во второе издание включен новый материал по операционным системам Linux и FreeBSD, удалены темы, утратившие актуальность, скорректирован набор описываемых системных вызовов и библиотечных функций, добавлен глоссарий англоязычных терминов.

Для студентов, преподавателей, пользователей и системных администраторов

УДК 681.3.06
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. гл. редактора	*Евгений Рыбаков*
Зав. редакцией	*Григорий Добин*
Редактор	*Алия Амирова*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Виктория Пиотровская*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 21.07.05.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 52,89.
Тираж 4000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 5-94157-538-6

Оглавление

Глава 2

Среда программирования UNIX

Одной из целей, которые изначально ставились перед разработчиками ОС UNIX, являлось создание удобной среды программирования. Во многом это справедливо и сегодня.

Разговор в данной главе пойдет о программировании в UNIX. Может показаться, что предлагаемый материал интересен лишь разработчикам программного обеспечения. Это не совсем так. Безусловно, разработка программ невозможна без знания интерфейса системных вызовов и без понимания внутренних структур и функций, предоставляемых операционной системой. Однако осмысленное администрирование системы также затруднительно без представления о том, как работает UNIX. Программный интерфейс UNIX позволяет наглядно показать внутренние механизмы этой операционной системы.

В начале главы дана общая характеристика программного интерфейса UNIX и связанной с ним среды разработки; затронуты такие важные темы, как обработка ошибок, различия между системными вызовами и функциями стандартных библиотек, форматы исполняемых файлов и размещение образа программы в памяти; также описано, как происходит запуск и завершение программы с точки зрения программиста.

Следующие два раздела посвящены подробному обсуждению программного интерфейса двух важнейших подсистем операционной системы UNIX: файловой подсистемы и подсистемы управления процессами и памятью. В них рассматриваются важнейшие системные вызовы работы с файлами, функции стандартной библиотеки ввода-вывода, системные вызовы создания процесса, запуска новой программы и управления процессами.

В заключение приводятся два типичных приложения: демон и командный интерпретатор, на примере которых проиллюстрированы темы, затронутые в данной главе.

Программный интерфейс ОС UNIX

Системные вызовы и функции стандартных библиотек

Все версии UNIX предоставляют строго определенный ограниченный набор входов в ядро операционной системы, через которые прикладные задачи имеют возможность воспользоваться базовыми услугами 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. Седьмая редакция UNIX включала около 50 системных вызовов, современные версии, например, Linux RedHat 7.3 и FreeBSD 4.4, предлагают более 160.

Системные вызовы обычно документированы *в разделе 2* электронного справочника. В среде программирования UNIX они определяются как функции С, независимо от фактической реализации вызова функции ядра операционной системы. В ОС 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. Заметим, что в этом случае библиотечный код выполняет только роль оболочки, в то время как фактические инструкции расположены в ядре операционной системы.

Помимо системных вызовов программисту предлагается большой набор библиотечных функций общего назначения. Эти функции не являются точками входа в ядро операционной системы, хотя в процессе выполнения многие из них выполняют системные вызовы. Например, функция `printf(3S)`[1] использует системный вызов `write(2)` для записи данных в файл, в то время как функции `strcpy(3C)` (копирование строки) или `atoi(3C)` (преобразование символа в его числовое значение) вообще не прибегают к услугам операционной системы. Функции, о которых идет речь, хранятся в стандартных библиотеках С и наряду с системными вызовами составляют основу среды программирования в UNIX. Подробное описание этих функций приведено *в разделе 3* электронного справочника.

---

[1] Подразделы электронного справочника, помеченные буквами, – особенность System 5, в частности, Solaris. В других системах подразделы, как правило, не выделяются.

Таким образом, часть библиотечных функций является "надстройкой" над системными вызовами, обеспечивающей более удобный способ получения системных услуг. В качестве примера рассмотрим процесс получения текущей даты и времени. Соответствующий системный вызов `time(2)` возвращает время в секундах, прошедшее с момента Epoch: 1 января 1970 года. Дополнительная интерпретация этого значения, такая как преобразование в вид, удобный для восприятия (дата и время) с учетом временной зоны, осуществляется библиотечными функциями (`ctime(3C)`, `localtime(3C)` и т. д.). К этим функциям можно отнести функции библиотеки ввода-вывода, функции распределения памяти, часть функций управления процессами и т. д.

На рис. 2.1 показана схема взаимодействия приложения с ядром операционной системы при использовании системных вызовов и библиотечных функций.

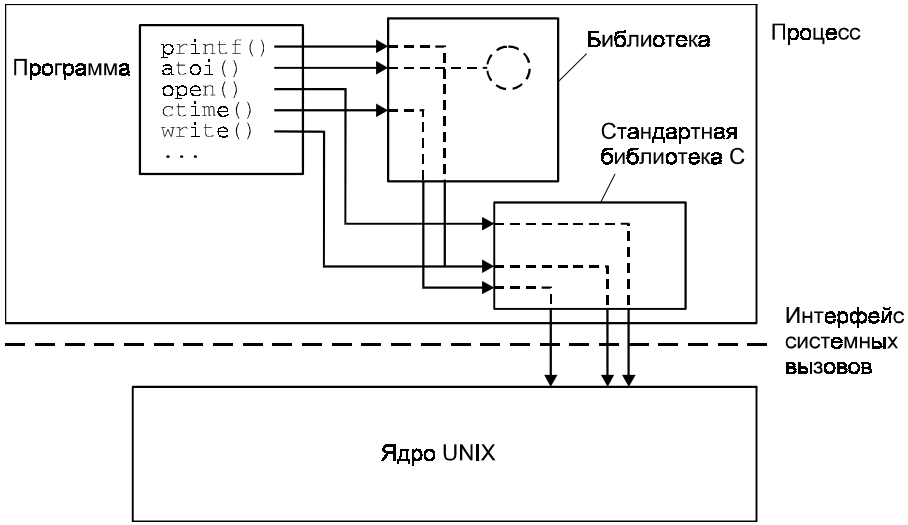


Рис. 2.1. Системные вызовы и библиотечные функции

Обработка ошибок

В предыдущем разделе мы обсудили разницу между системными вызовами и библиотечными функциями. Они также различаются по способу передачи процессу информации об ошибке, произошедшей во время выполнения системного вызова или функции библиотеки.

Обычно в случае возникновения ошибки системные вызовы возвращают -1 и устанавливают значение переменной `errno`, указывающее причину возникно-

вения ошибки. Так, например, существует более десятка причин завершения вызова `open(2)` с ошибкой, и все они могут быть определены с помощью переменной `errno`. Файл заголовков <errno.h> содержит коды ошибок, значения которых может принимать переменная `errno`, с краткими комментариями.

Библиотечные функции, как правило, не устанавливают значение переменной `errno`, а код возврата различен для разных функций. Для уточнения возвращаемого значения библиотечной функции необходимо обратиться к электронному справочнику `man(1)`.

Поскольку базовым способом получения услуг ядра являются системные вызовы, рассмотрим более подробно обработку ошибок в этом случае.

Переменная `errno` определена следующим образом:

```
external int errno;
```

Следует обратить внимание, что значение `errno` не обнуляется следующим, нормально завершившимся системным вызовом. Таким образом, значение этой переменной имеет смысл только после системного вызова, который завершился с ошибкой.

Стандарт ANSI C определяет две функции, помогающие сообщить причину ошибочной ситуации: `strerror(3C)` и `perror(3C)`.

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, добавляя, например, название функции или программы, в которой произошла ошибка.

Использование этих двух функций иллюстрирует пример, приведенный в листинге 2.1.

Листинг 2.1. Использование функций `strerror(3C)` и `perror(3C)`

```
#include <errno.h>
#include <stdio.h>
main(int argc, char *argv[])
{
        fprintf(stderr, "ENOMEM: %s\n", strerror(ENOMEM));
        errno = ENOEXEC;
        perror(argv[0]);
}
```

Запустив программу, получим на экране следующий результат:

```
$ a.out
ENOMEM: Not enough space
a.out: Exec format error
```

Эти функции используются, в частности, командным интерпретатором и большинством стандартных утилит UNIX. Например:

```
$ rm does_not_exist
rm: Cannot remove does_not_exist: No such file or directory      ошибка
                                                                 ENOENT
$ pg do_not_read
do_not_read: Permission denied                                   ошибка
                                                                 EACCESS
```

Список кодов ошибок основных системных вызовов для ОС UNIX стандартизован (POSIX.1). Однако с появлением новых системных вызовов определяются новые ошибки, толкования которых в разных версиях UNIX различаются. Обычно полный список определений номеров и кодов ошибок системных вызовов находится в файле /usr/include/errno.h, но не всегда: RedHat Linux размещает этот список в /usr/include/asm/errno.h.

В таблице 2.1 приведены наиболее общие ошибки системных вызовов, включая сообщения, которые обычно выводят функции `strerror(3C)` и `perror(3C)`, а также их краткое описание.

Таблица 2.1. *Некоторые ошибки системных вызовов*

Код ошибки и сообщение	Описание
`E2BIG` `Arg list too long`	Размер списка аргументов, переданных системному вызову `exec(2)`, плюс размер экспортируемых переменных окружения превышает `ARG_MAX` байт

Таблица 2.1 *(продолжение)*

Код ошибки и сообщение	Описание
EACCESS Permission denied	Попытка доступа к файлу с недостаточными правами для данного класса (определяемого эффективными UID и GID процесса и соответствующими идентификаторами файла)
EAGAIN Resource temporarily unavailable	Превышен предел использования некоторого ресурса, например, переполнена таблица процессов или пользователь превысил ограничение по количеству процессов с одинаковыми UID. Причиной также может являться недостаток памяти или превышение соответствующего ограничения (*см. раздел "Ограничения" далее в этой главе*)
EALREADY Operation already in progress	Попытка выполнения операции с неблокируемым объектом, уже обслуживающим некоторую операцию
EBADF Bad file number	Попытка выполнения операции с файловым дескриптором, не адресующим никакой файл; также попытка выполнения операции чтения или записи с файловым дескриптором, полученным при открытии файла на запись или чтение, соответственно
EBADFD File descriptor in bad state	Файловый дескриптор не адресует открытый файл или попытка выполнения операции чтения с файловым дескриптором, полученным при открытии файла только на запись
EBUSY Device busy	Попытка монтирования устройства (файловой системы), которое уже примонтировано; попытка размонтировать файловую систему, имеющую открытые файлы; попытка обращения к недоступным ресурсам (семафоры, блокираторы и т. д.)
ECHILD No child processes	Вызов функции `wait(2)` процессом, не имеющим дочерних процессов или процессов, для которых уже был сделан вызов `wait(2)`
EDQUOT Disk quota exceeded	Попытка записи в файл, создание каталога или файла при превышении квоты пользователя на дисковые блоки, попытка создания файла при превышении пользовательской квоты на число индексных дескрипторов
EEXIST File exists	Имя существующего файла использовано в недопустимом контексте, например, сделана попытка создания символической связи с именем уже существующего файла

Таблица 2.1 *(продолжение)*

Код ошибки и сообщение	Описание
EFAULT Bad address	Аппаратная ошибка при попытке использования системой аргумента функции, например, в качестве указателя передан недопустимый адрес
EFBIG File too large	Размер файла превысил установленное ограничение RLIMIT_FSIZE или максимально допустимый размер для данной файловой системы (*см. раздел "Ограничения" далее в этой главе*)
EINPROGRESS Operation now in progress	Попытка выполнения длительной операции (например, установление сетевого соединения) для неблокируемого объекта
EINTR Interrupted system call	Получение асинхронного сигнала, например, сигнала SIGINT или SIGQUIT, во время обработки системного вызова. Если выполнение процесса будет продолжено после обработки сигнала, прерванный системный вызов завершится с этой ошибкой
EINVAL Invalid argument	Передача неверного аргумента системному вызову. Например, размонтирование устройства (файловой системы), которое не было примонтировано. Другой пример — передача номера несуществующего сигнала системному вызову kill(2)
EIO I/O error	Ошибка ввода-вывода физического устройства
EISDIR Is a directory	Попытка операции, недопустимой для каталога, например, запись в каталог с помощью вызова write(2)
ELOOP Number of symbolic links encountered during path name traversal exceeds MAXSYMLINKS	При попытке трансляции имени файла было обнаружено недопустимо большое число символических связей, превышающее значение MAXSYMLINKS
EMFILE Too many open files	Число открытых файлов для процесса превысило максимальное значение OPEN_MAX
ENAMETOOLONG File name too long	Длина полного имени файла (включая путь) превысила максимальное значение PATH_MAX
ENFILE File table overflow	Переполнение файловой таблицы

Таблица 2.1 *(продолжение)*

Код ошибки и сообщение	Описание
ENODEV No such device	Попытка недопустимой операции для устройства. Например, попытка чтения устройства только для записи или операция для несуществующего устройства
ENOENT No such file or directory	Файл с указанным именем не существует или отсутствует каталог, указанный в полном имени файла
ENOEXEC Exec format error	Попытка запуска на выполнение файла, который имеет права на выполнение, но не является файлом допустимого исполняемого формата
ENOMEM Not enough space	При попытке запуска программы (`exec(2)`) или размещения памяти (`brk(2)`) размер запрашиваемой памяти превысил максимально возможный в системе
ENOMSG No message of desired type	Попытка получения сообщения определенного типа, которого не существует в очереди (*см. раздел "Сообщения" в главе 3*)
ENOSPC No space left on device	Попытка записи в файл или создания нового каталога при отсутствии свободного места на устройстве (в файловой системе)
ENOSR Out of stream resources	Отсутствие очередей или головных модулей при попытке открытия устройства STREAMS. Это состояние является временным. После освобождения соответствующих ресурсов другими процессами операция может пройти успешно
ENOSTR Not a stream device	Попытка применения операции, определенной для устройств типа STREAMS (например, системного вызова `putmsg(2)` или `getmsg(2)`), для устройства другого типа
ENOTDIR Not a directory	В операции, предусматривающей в качестве аргумента имя каталога, было указано имя файла другого типа (например, в пути для полного имени файла)
ENOTTY Inappropriate ioctl for device	Попытка системного вызова `ioctl(2)` для устройства, которое не является символьным
EPERM Not owner	Попытка модификации файла способом, разрешенным только владельцу и суперпользователю и запрещенным остальным пользователям. Попытка операции, разрешенной только суперпользователю

Таблица 2.1 *(окончание)*

Код ошибки и сообщение	Описание
EPIPE Broken pipe	Попытка записи в канал, для которого не существует процесса, принимающего данные. В этой ситуации процессу обычно отправляется соответствующий сигнал. Ошибка возвращается при игнорировании сигнала
EROFS Read-only file system	Попытка модификации файла или каталога для устройства (файловой системы), примонтированного только на чтение
ESRCH No such process	Процесс с указанным PID не существует в системе

Создание программы

Создание любой программы обычно начинается с базовой идеи (но не всегда), разработки ее блок-схемы (современные программисты часто пропускают этот этап), интерфейса пользователя (весьма ответственный процесс) и написания исходного текста. Далее следуют этапы компиляции и отладки.

В этом разделе рассмотрен процесс создания приложения на языке C для операционной системы UNIX. Предвидя обвинения в архаизме, мы все-таки остановимся на добротном ANSI C и базовой среде разработки UNIX, во-первых, полагая, что старый друг лучше новых двух, а во-вторых потому, что объектом нашего обсуждения все же является UNIX, а не современные средства создания приложений. Заметим также, что язык программирования C является "родным" языком UNIX, поскольку ядро операционной системы написано на этом языке[2]. Это, безусловно, не ограничивает возможности других языков и технологий программирования, которые сегодня, наверное, используются даже чаще, чем обсуждаемый нами традиционный подход.

Примечание

Несмотря на то, что многие современные версии UNIX (особенно коммерческие) поставляются без исходных текстов, основная часть кода ядра в них получена путем компиляции C-модулей.

Опустим также процесс рождения базовой идеи и разработку блок-схем, полагая, что все это уже сделано. Итак, начнем с исходного текста будущей программы.

Исходный текст

Исходные тексты программы, разработанной для UNIX, по большому счету мало отличаются от текстов приложений, создаваемых для других операционных систем. Можно уверенно сказать, что синтаксис языка определяется не операционной системой. Все, что вам потребуется, это хорошее знание самого языка и особенностей системы UNIX, а именно — ее системных вызовов.

Во-первых, не забудьте включить в исходный текст необходимые файлы заголовков. Во-вторых, уточните синтаксис вызова библиотечных и системных функций. В-третьих, используйте их по назначению. В-четвертых, не пренебрегайте комментариями.

В этом (за исключением, пожалуй, четвертого совета) вам помогут электронный справочник `man(1)`, ваш опыт и, надеемся, эта книга.

Заголовки

Использование системных функций обычно требует включения в текст программы файлов заголовков, содержащих определения функций — число передаваемых аргументов, типы аргументов и возвращаемого значения. Большинство системных файлов заголовков расположено в каталогах /usr/include или /usr/include/sys. Если вы планируете использовать малознакомую системную функцию, будет нелишним изучить соответствующий раздел электронного справочника `man(1)`. Там же, помимо описания формата функции, возвращаемого значения и особых ситуаций, вы найдете указание, какие файлы заголовков следует включить в программу.

Файлы заголовков включаются в программу с помощью директивы `#include`. При этом если имя файла заключено в угловые скобки (<>), это означает, что поиск файла будет производиться в общепринятых каталогах хранения файлов заголовков. Если же имя файла заголовка заключено в кавычки, то используется явно указанное абсолютное или относительное имя файла.

Например, системный вызов `creat(2)`, служащий для создания обычного файла, объявлен в файле <fcntl.h> следующим образом:

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

Включение в исходный текст прототипа системного вызова `creat(2)` позволяет компилятору произвести дополнительную проверку правильности использования этой функции, а именно — числа аргументов и их типов. Можно заметить, что наряду со стандартными типами языка C, например `char`, для второго аргумента `creat(2)` используется производный тип — `mode_t`. В ранних версиях ОС UNIX большинство системных вызовов использовали стандартные типы, например, `creat(2)` для второго аргумента охотно принимала тип `int`. Производные типы переменных, имеющие окончание `_t`, которые в большом количестве встречаются при программировании в UNIX, получили название примитивов системных данных. Большинство этих типов определены в файле <sys/types.h>, а их назначение заключается в улучшении переносимости написанных программ. Вместо конкретных типов данных, каковыми являются `int`, `char` и т. д., приложению предлагается набор системных типов, гарантированно неизменных в контексте системных вызовов. Другими словами, во всех версиях UNIX сегодня и спустя десять лет, системный вызов `creat(2)` в качестве второго аргумента будет принимать переменную типа `mode_t`. Фактический размер переменных этого типа может быть разным для различных версий системы, но это отразится в изменении соответствующего файла заголовков и потребует только перекомпиляции вашей программы.

Среда программирования UNIX определяется несколькими стандартами, обсуждавшимися во введении, и может незначительно различаться для разных версий системы. В частности, стандарты ANSI C, POSIX.1 и XPG4 определяют названия и назначения файлов заголовков, приведенных в табл. 2.2.

Таблица 2.2. *Стандартные файлы заголовков*

Файл заголовка	**Назначение**
<assert.h>	Содержит прототип функции `assert(3C)`, используемой для диагностики при отладке
<cpio.h>	Содержит определения, используемые для файловых архивов `cpio(1)`
<ctype.h>	Содержит определения символьных типов, а также прототипы функций определения классов символов (ASCII, печатные, цифровые и т. д.) — `isascii(3C)`, `isprint(3C)`, `isdigit(3C)` и т. д.
<dirent.h>	Содержит определения структур данных каталога, а также прототипы функций работы с каталогами `opendir(3C)`, `readdir(3C)` и т. д.
<errno.h>	Содержит определения кодов ошибок (*см. раздел "Обработка ошибок" в начале данной главы*)

Таблица 2.2 *(продолжение)*

Файл заголовка	Назначение
<fcntl.h>	Содержит прототипы системных вызовов `fcntl(2)`, `open(2)` и `creat(2)`, а также определения констант и структур данных, необходимых при работе с файлами
<features.h>	Содержит определения констант, описывающих соответствия стандартам
<ftw.h>	Содержит прототипы функций, используемых для сканирования дерева файловой системы `ftw(3C)` и `nftw(3C)`, а также определения используемых констант
<grp.h>	Содержит прототипы функций и определения структур данных, используемых для работы с группами пользователей: `getgrnam(3C)`, `getgrent(3C)`, `getgrgid(3C)` и т. д.
<langinfo.h>	Содержит определения языковых констант: дни недели, названия месяцев и т. д., а также прототип функции `langinfo(3C)`
<limits.h>	Содержит определения констант, устанавливающих значения ограничений для данной реализации: минимальные и максимальные значения основных типов данных, максимальное значение файловых связей, максимальная длина имени файла и т. д.
<locale.h>	Содержит определения констант, используемых для создания пользовательской среды, зависящей от языковых и культурных традиций (форматы дат, денежные форматы и т. д.), а также прототип функции `setlocale(3C)`
<math.h>	Содержит определения математических констант (π, e, $\sqrt{2}$ и т. д.)
<nl_types.h>	Содержит определения для каталогов сообщений, а также прототипы функций `catopen(3C)` и `catclose(3C)`
<pwd.h>	Содержит определение структуры файла паролей /etc/passwd, а также прототипы функций работы с ним: `getpwnam(3C)`, `getpwent(3C)`, `getpwuid(3C)` и т. д.
<regex.h>	Содержит определения констант и структур данных, используемых в регулярных выражениях, а также прототипы функций для работы с ними: `regcomp(3C)`, `regexec(3C)` и т. д.
<search.h>	Содержит определения констант и структур данных, а также прототипы функций, необходимых для поиска: `hsearch(3C)`, `hcreate(3C)`, `hdestroy(3C)`
<setjmp.h>	Содержит прототипы функций перехода `setjmp(3C)`, `sigsetjmp(3C)`, `longjmp(3C)`, `siglongjmp(3C)`, а также определения связанных с ними структур данных

Таблица 2.2 *(продолжение)*

Файл заголовка	Назначение
<signal.h>	Содержит определения констант и прототипы функций, необходимых для работы с сигналами: `sigsetops(3C)`, `sigemptyset(3C)`, `sigaddset(3C)` и т. д. (*см. раздел "Сигналы" далее в этой главе*)
<stddef.h>	Содержит стандартные определения (например `size_t`)
<stdio.h>	Содержит определения стандартной библиотеки ввода-вывода
<stdlib.h>	Содержит определения стандартной библиотеки
<string.h>	Содержит прототипы функций работы со строками `string(3C)`, `strcasecmp(3C)`, `strcat(3C)`, `strcpy(3C)` и т. д.
<tar.h>	Содержит определения, используемые для файловых архивов `tar(1)`
<termios.h>	Содержит определения констант, структур данных и прототипы функций для обработки терминального ввода-вывода
<time.h>	Содержит определения типов, констант и прототипы функций для работы со временем и датами: `time(2)`, `ctime(3C)`, `localtime(3C)`, `tzset(3C)`, а также определения, относящиеся к таймерам `getitimer(2)`, `setitimer(2)`. Таймеры будут рассмотрены *в главе 3*
<ulimit.h>	Содержит определения констант и прототип системного вызова `ulimit(2)` для управления ограничениями, накладываемыми на процесс. *См. раздел "Ограничения" далее в этой главе*
<unistd.h>	Содержит определения системных символьных констант, а также прототипы большинства системных вызовов
<utime.h>	Содержит определения структур данных и прототип системного вызова `utime(2)` для работы с временными характеристиками файла (временем доступа и модификации)
<sys/ipc.h>	Содержит определения, относящиеся к системе межпроцессного взаимодействия (IPC), которые рассматриваются *в главе 3*
<sys/msg.h>	Содержит определения, относящиеся к подсистеме IPC (сообщениям). *См. также раздел "Сообщения" главы 3*
<sys/resource.h>	Содержит определения констант и прототипы системных вызовов управления размерами ресурсов, доступных процессу: `getrlimit(2)` и `setrlimit(2)`. Более подробно ограничения на ресурсы обсуждаются *в разделе "Ограничения" далее в этой главе*

Таблица 2.2 *(окончание)*

Файл заголовка	Назначение
<sys/sem.h>	Содержит определения, относящиеся к подсистеме IPC (семафорам). *См. также раздел "Семафоры" главы 3*
<sys/shm.h>	Содержит определения, относящиеся к подсистеме IPC (разделяемой памяти). *См. также раздел "Разделяемая память" главы 3*
<sys/stat.h>	Содержит определения структур данных и прототипы системных вызовов, необходимых для получения информации о файле: `stat(2)`, `lstat(2)`, `fstat(2)`. Подробнее эти системные вызовы рассмотрены *в разделе "Метаданные файла" далее в этой главе*
<sys/times.h>	Содержит определения структур данных и прототипа системного вызова `times(2)`, служащего для получения статистики выполнения процесса (времени выполнения в режиме ядра, задачи и т. д.)
<sys/types.h>	Содержит определения примитивов системных данных
<sys/utsname.h>	Содержит определения структур данных и прототип системного вызова `uname(2)`, используемого для получения имен системы (компьютера, операционной системы, версии и т. д.)
<sys/wait.h>	Содержит определения констант и прототипы системных вызовов `wait(2)`, `waitpid(2)`, используемых для синхронизации выполнения родственных процессов

Компиляция

Процедура создания большинства приложений является общей, ее схема приведена на рис. 2.2.

Первой фазой является стадия компиляции, когда файлы с исходными текстами программы, включая файлы заголовков, обрабатываются компилятором C: `cc(1)` или, если в системе используется GNU-версия компилятора, `gcc(1)`. Параметры компиляции задаются либо в файле makefile (или Makefile), либо явным указанием необходимых ключей компилятора в командной строке. В итоге компилятор создает набор промежуточных объектных файлов. Традиционно имена созданных объектных файлов имеют суффикс o.

На следующей стадии эти файлы с помощью редактора связей `ld(1)` связываются друг с другом и с различными библиотеками, включая стандартную библиотеку по умолчанию и библиотеки, указанные пользователем в качестве параметров. При этом редактор связей может выполняться в двух режимах: статическом и динамическом, что задается соответствующими опциями.

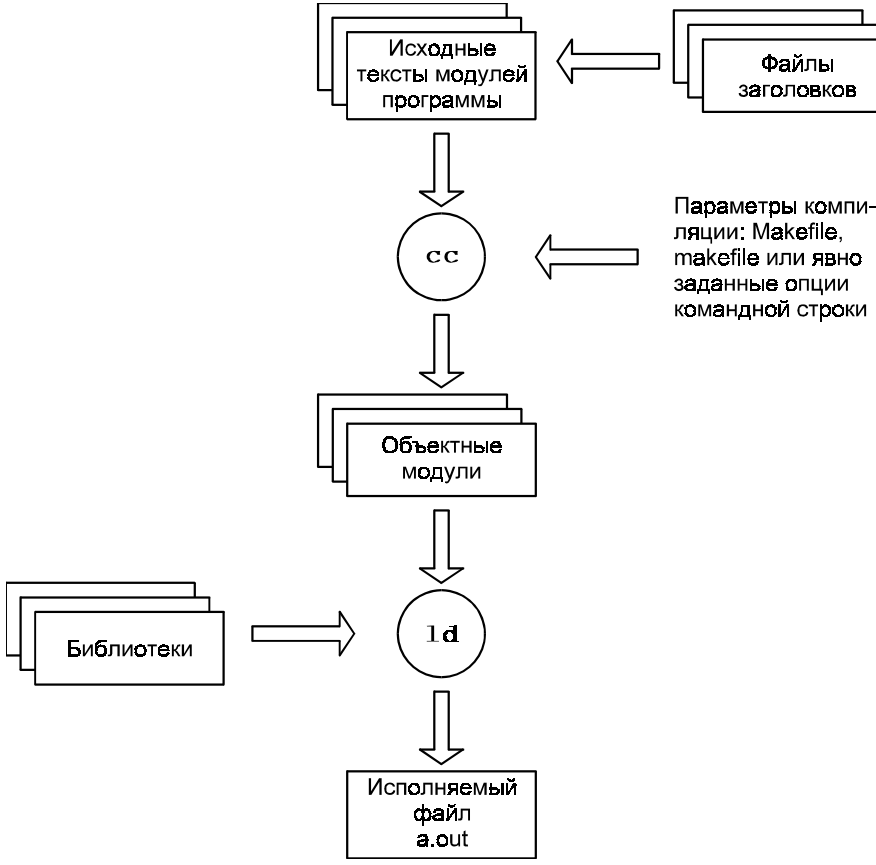


Рис. 2.2. Схема компиляции программы

В статическом режиме связываются все объектные модули и статические библиотеки (их имена имеют суффикс a), производится разрешение всех внешних ссылок модулей и создается единый исполняемый файл, содержащий весь необходимый для выполнения код. Во втором случае, редактор связей, разрешая ссылки, по возможности подключает *разделяемые библиотеки* (имена этих библиотек имеют суффикс so), содержащие необходимые функции в виде разделяемых объектных файлов. В результате создается исполняемый файл, к которому все необходимые объектные файлы будут подключены в процессе выполнения. В обоих случаях создается исполняемый файл с именем a.out, если не указано другое имя. По умолчанию выполняется динамическое связывание.

Для достаточно простых задач (если, например, вся программа помещается в файле с именем prog.c) все фазы создания исполняемого модуля с именем prog автоматически выполняются вызовом команды:

```
$ make prog
```

или эквивалентной ей

```
$ cc -o prog prog.c
```

Альтернативное имя prog вместо имени a.out, задаваемого по умолчанию, назначается с помощью ключа `-o`.

Указанные стадии можно выполнять и раздельно, с использованием команд `cc(1)` и `ld(1)`. Заметим, что на самом деле команда `cc(1)` является программной оболочкой компилятора и редактора связей, которую рекомендуется использовать при создании программ.

Проиллюстрируем процесс создания более сложной программы с помощью конкретных вызовов команд (табл. 2.3).

Таблица 2.3. *Команды вызова компиляторов*

Команда	Назначение
`$ cc -c file1.c file2.c`	Создадим промежуточные объектные файлы file1.o и file2.o
`$ cc -o prog file1.o file2.O -lnsl`	Создадим исполняемый файл с именем prog, используя промежуточные объектные файлы и библиотеку libnsl.a или libnsl.so

Форматы исполняемых файлов

Виртуальная память процесса состоит из нескольких *сегментов* или *областей* памяти. Размер, содержимое и расположение сегментов в памяти определяется как самой программой, например, использованием библиотек, размером кода и данных, так и форматом исполняемого файла этой программы. В подавляющем большинстве современных операционных систем UNIX используется один стандартный формат исполняемых файлов — ELF (Executable and Linking Format). Формат COFF (Common Object File Format), распространившийся вместе с System V Release 3 и достаточно популярный еще в середине 90-х годов 20-го века, теперь развивается в операционных системах фирмы Microsoft. Базовый формат исполняемых файлов, изначально использовавшийся в UNIX, a.out[2], практически совсем вышел из употребления после перехода FreeBSD на ELF (начиная с версии 3).

---

[2] Название этого формата исполняемых файлов совпадает с названием по умолчанию выходного файла компилятора.