Библиотека языка C GNU glibc

Справочное руководство
ред. 0.06 24.10.1994

Сандра Лузмор (Sandra Loosemore)
Ричард Сталлман (Richard M. Stallman)
Роланд Макграх (Roland MacGrath)
Андрей Орам (Andrew Oram)

1. Введение

Язык C не обеспечивает никаких встроенных средств для выполнения таких общих операций как ввод-вывод, управление памятью, обработка строк, и т.п.. Вместо этого, такие средства определены в стандартной библиотеке, которую Вы подключаете к вашим программам.

Библиотека GNU C, описанная в этом документе, содержит описание всех библиотечных функций, которые определены в соответствии с ANSI C стандартом, учитывая дополнительные особенности, специфические для POSIX-стандарта операционной системы UNIX, и расширений, специфических для GNU-разработок.

Цель этого руководства состоит в том, чтобы сообщить Вам, как использовать средства GNU библиотеки. Мы упомянули особенности ее стандартов, чтобы помочь Вам разобраться с вещами, которые являются потенциально непереносимыми на другие системы. Но переносимость не является основным вопросом данного руководства.

1.1 Начало

Это руководство написано с учетом того, что Вы по крайней мере немного знакомы с языком программирования C и базисными понятиями программирования. Вообще считается, что стандарт ANSI C (см. раздел 1.2.1 [ANSI C]) проще для понимания, чем "традиционные", предшествующие ANSI C, диалекты.

Если Вы читаете это руководство впервые, Вам следует подробно прочитать весь вводный материал, а остальное главы только просмотреть. В

библиотеке GNU C имеется множество функций, и Вы не запомните, как использовать каждую из них. Но Вы в общих чертах ознакомитесь с видами средств, которые обеспечивает библиотека, поэтому когда Вы будете писать ваши программы, Вы сможете распознавать, когда использовать библиотечные функции, и где в этом руководстве Вы сможете найти более специфическую информацию по ним.

1.2 Стандарты и переносимость

Этот раздел обсуждает различные стандарты и другие источники, на которых базируется GNU C. Эти источники включают стандарты ANSI и POSIX, а так же реализации System V и Berkeley Unix.

Первичная цель этого руководства - сообщить Вам, как эффективно использовать средства библиотеки GNU. Но если Вас интересует создание ваших программ, совместимых с определенными стандартами, или переносимых на операционные системы отличные от GNU, то оно укажет Вам направление использования библиотеки. Этот раздел дает Вам краткий обзор выше перечисленных стандартов, так, чтобы Вы знали, что они представляют, когда они упоминаются в других частях руководства.

См. Приложение B [Обзор библиотеки], для получения алфавитного списка функций и других символов, обеспечиваемых библиотекой. Этот список указывает также, на какие стандарты опирается каждая функция или символ.

1.2.1 ANSI C

Библиотека GNU C совместима со стандартом C, принятым Американским Институтом Национальных Стандартов (ANSI): American National Standard X3.159-1989_"ANSI C". Заголовочные файлы и библиотечные средства, которые составляют библиотеку GNU - надмножество средств, определенных в соответствии с ANSI C стандартом.

1.2.2 Библиотека GNU C

Если Вы следуете относительно строгой приверженности к ANSI C стандарту, Вы должны использовать опцию ` -ansi ', когда Вы компилируете ваши программы компилятором GNU C. Эта опция указывает, чтобы транслятор определил только возможности стандарта ANSI из библиотечных заголовочных файлов, если Вы явно не попросите о дополнительных особенностях, для получения информации относительно того, как это сделать см. раздел 1.3.4 [Макрокоманды установки возможностей].

Способность ограничить библиотеку, чтобы включить только

возможности ANSI C важна, потому что ANSI C устанавливает ограничения, которым не удовлетворяют некоторые имена, определяемые библиотечной реализацией, т.е. GNU расширения не всегда удовлетворяют этим ограничениям. См. раздел 1.3.3 [Зарезервированные имена], где приводится более подробная информация относительно этих ограничений.

Это руководство не будет вдаваться в подробности различий между ANSI C и более старыми диалектами. Оно поможет вам написать программы, чтобы работать корректно под многими диалектами C, но не более.

Библиотека GNU также совместима с семейством стандартов IEEE POSIX, известных более формально как Переносной Интерфейс Операционной Системы. POSIX взят в основном из различных версий операционной системы UNIX.

Библиотечные средства, определенные POSIX стандартами - надмножество соответствующих стандарту ANSI C; POSIX определяет дополнительные особенности для функций ANSI C, и задает определения новых дополнительных функций. Вообще предпочтение отдается скорее дополнительным требованиям и функциональным возможностям, определяемым POSIX стандартами, обеспечивающим поддержку низшего уровня для определенной среды операционной системы, чем общей поддержке языка программирования, который может выполняться в многих разнообразных средах операционной системы.

GNU C библиотека осуществляет все функции, определенные в IEEE Std 1003.1-1990, Системе Интерфейса Прикладных программ POSIX, обычно упоминающемся как POSIX.1. Первичные расширения к ANSI C средствам, определенные этими стандартными примитивами интерфейса файловой системы (см. Главу 9 [Интерфейс файловой системы]), зависят от устройства функции управления терминалом (см. Главу 12 [Интерфейс терминала низкого уровня]), и функции управления процессом (см. Главу 23 [Порожденнные процессы]).

Некоторые средства из IEEE Std 1003.2-1992, Стандарта Оболочки и Утилит POSIX (POSIX.2) также представлены в библиотеке GNU. Они включают в себя утилиты для обработки регулярных выражений и других средств сопоставления с образцом (см. Главу 16 [Сопоставления с образцом]).

1.2.3 Berkeley Unix

GNU C библиотека определяет средства (которые формально не стандартизированы) из некоторых версий UNIX: 4.2 BSD, 4.3 BSD, и 4.4 BSD Unix системы (также известной как Berkeley Unix) и из SunOS (популярной 4.2 BSD производной, которая включает некоторые функциональные возможности Unix System V). Эта система поддерживает большинство ANSI и POSIX средств, 4.4 BSD и более новые выпуски SunOS, т.е. фактически поддерживает их все.

BSD средства включают символические связи (см. раздел 9.4 [Символические связи]), функцию выбора (см. раздел 8.6 [Ожидание ввода - вывода]), функции сигнала BSD (см. раздел 21.9 [BSD Обработка сигнала]), и межпроцессорные связи (см. Главу 11 [Гнезда]).

1.2.4 SVID (Описание интерфейса System V)

Описание Интерфейса System V (SVID) - документ, описывающий AT&T операционную систему Unix System V. Это -, в некоторой степени, надмножество стандарта POSIX (см. раздел 1.2.2 [POSIX])

GNU C библиотека определяет некоторых из средств, требуемых SVID, и не требуемых в соответствии с ANSI или POSIX стандартами, для совместимости с Unix System V и другой UNIX системой (типа SunOS), которые включают эти средства. Однако, многие из большего количества менее известных и менее полезных средств, требуемых SVID не включены. (Фактически, Unix System V сама не обеспечивает их все.)

1.3 Использование библиотеки

Этот раздел описывает некоторые из практические проблемы использования GNU C библиотек.

1.3.1 Заголовочные файлы

Библиотеки используемые C программами на самом деле состоят из двух частей: заголовочные файлы, которые определяют типы и макрокоманды и объявляют переменные и функции; и фактическая библиотека или архив,

который содержит определения переменных и функций.

(В Си, описание просто представляет информацию,что функция или переменная существует, и присваивает ей тип. Для функционального объявления, нужно указать информацию относительно типов параметров. Цель объявлений состоит в том, чтобы транслятор правильно обработал ссылки к объявленным переменным и функциям. С другой стороны, определение фактически распределяет память для переменных или указывает, что делает функция.)

Чтобы использовать средства библиотеки GNU C, Вы должны убедиться, что ваши исходные программы включают соответствующие заголовочные файлы, чтобы транслятор видел объявления доступных средств, и мог правильно обрабатывать ссылки на них. При компиляции вашей программы, компоновщик свяжет эти ссылки с фактическими определениями, находящимися в файле архива.

Файлы заголовка включаются в исходный файл директивой препроцессора "#include". Язык C поддерживает две формы этой директивы; первая,

```
#include "header"
```

обычно используется, чтобы включить заголовочный файл header, который Вы пишете самостоятельно; он обычно содержит определения и объявления, описывающие интерфейсы между различными частями вашей конкретной прграммы. Другая форма,

```
#include
```

обычно используется, чтобы включить заголовочный файл "file.h" который содержит определения и объявления для стандартной библиотеки. Этот файл обычно устанавливается в стандартное место вашим администратором системы. Вы должны использовать эту вторую форму для файлов заголовка библиотеки C.

Обычно, директивы "#include" помещены в начале исходного C файла, перед любым другим кодом. Если Вы начинаете ваши исходные файлы некоторыми комментариями, объясняя, что код в файле делает, поместите

директивы "#include" сразу после комментариев, согласно особенностям макроопределения (см. раздел 1.3.4 [Особенности макрокоманд]).

Для получения подробной информации относительно использования заголовочных и директивы "#include" см. раздел "Заголовочные файлы" в Руководстве по препроцессору GNU C.

GNU C библиотека обеспечивает несколько заголовочных файлов, каждый из которых содержит тип, макроопределения, переменную и описания функций для группы связанных средств. Это означает, что в ваши программы следует включить несколько файлов заголовка, в зависимости от того, какие средства Вы используете.

Некоторые библиотечные заголовочные файлы включают другие библиотечные заголовочные файлы. Однако, программируя в хорошем стиле, Вы не должны полагаться на это; лучше явно включить все заголовочные файлы, требуемые для библиотечных средств, которые Вы используете. GNU C библиотека была написана таким способом, что не имеет значения, включен ли заголовочный файл случайно больше чем один раз; при включении заголовочного файла второй раз ничего не происходит. Аналогично, если ваша программа многократно включает заголовочные файлы, порядок включения, не имеет значения.

Примечание относительно совместимости: включение стандартного заголовочного файла в любом порядке и любое число раз работает в любой ANSI C реализации. Однако, этого традиционно не было во многоих более старых C реализациях.

Строго говоря, Вы не должны включать заголовочный файл, чтобы использовать функцию, которую он объявляет; Вы можете объявить функцию явно, согласно спецификациям в этом руководстве. Но обычно лучше включить заголовочный файл, потому что он может определять типы и макрокоманды, которые иначе не доступны, или он может определять более эффективные макро замены для некоторых функций. Это - также верный способ иметь правильное объявление.

1.3.2 Макроопределения функций

Если мы в этом руководстве описываем что-нибудь как функцию, то она может иметь и макроопределение. Обычно это не имеет никакого значение, потому что ваше макроопределение делает ту же самую вещь, что и функция. В частности, макро-эквиваленты для библиотечных функций вычисляют параметры ровно один раз, так же образом, как и при обращение к функции. Основная причина для этих макроопределений состоит в том, что иногда они могутреализовывать встроенное расширение, которое является значительно быстрее, чем фактическое обращение к функции.

Взятие адреса библиотечной функции работает даже, если она определена как макрокоманда, потому что, в этом контексте, имя функции не сопровождается левой круглой скобкой, которая является синтаксически необходимой, чтобы распознать макрообращение.

Вы можете иногда не использовать макроопределение функции, возможно, чтобы сделать вашу программу более простой или для отладки. Имеются два способа, сделать это:

* Вы можете избегать макроопределения, включая имя функции в круглых скобках. Это работает, потому что имя функции не появляется в синтаксическом контексте, где оно распознаваемо как макрообращение.

* Вы можете подавить любое макроопределение для целого исходного файла, используя директиву препроцессора "#undef ", если в явно описании средства не указано обратное.

Например, предположим, что заголовочный файл "stdlib.h" объявляет функцию abs как

```
extern int abs ( int);
```

и также обеспечивает макроопределение для abs. Тогда при выполнении:

```
#include < stdlib.h >
int f ( int * i) {return (abs (++ * i));}
```

ссылка на abs может относиться или к макрокоманде или функции. С другой

стороны, в каждом из следующих примеров производится ссылка на функцию, а не на макрос.

```
#include < stdlib.h >
int g ( int * i) {return ((abs) (++ * i));}
#undef abs
int h ( int *i) { return (abs (++*i)); }
```

Так как макроопределения, дублирующие функции, ведут себя в точно так же, как фактическая версия функции, удаление макроопределений обычно делает вашу программу медленнее.

1.3.3 Зарезервированные имена

Имена всех библиотечных типов, макрокоманд, переменных и функций, которые исходят из ANSI C стандарта, зарезервированы безоговорочно; ваша программа не имеет права переопределять эти имена. Все другие библиотечные имена зарезервированы, если ваша программа явно включает заголовочный файл, который определяет или объявляет их. Для этих ограничений имеются несколько причин:

* Другие люди, читая ваш код могут запутаться, если Вы использовали функцию exit, чтобы делать что - нибудь полностью отличное от того, что делает стандартная функция выхода.

Предотвращение этой ситуации помогает делать ваши программы более простыми, для понимания и способствует модульности.

* Это лишает пользователя возможности случайно переопределить библиотечную функцию, которая вызывается в соответствии с другими библиотечными функциями. Если переопределение позволялось, те другие функции не будут работать.

* Это позволяет транслятору делать любые специальные оптимизации опираясь на обращения к этим функциям, без риска, что они могут быть переопределены пользователем. Некоторые библиотечные средства, такие как variadic функции (с переменным числом аргументов, см. раздел A.2 [Variadic функции]) и нелокальные выходы (см. Главу 20 [Нелокальные

выходы]), фактически требуют более близкого сотрудничества с частью C транслятора, и для транслятора проще обрабатывать их как встроенные части языка.

В дополнение к именам, описываемым в этом руководстве, зарезервированные имена включают в себя все внешние идентификаторы (глобальные функции и переменные) начинающиеся с подчеркивания ("_") и все идентификаторы независимо от использования, которые начинаются с двух подчеркиваний, или с подчеркивания сопровождаемого заголовочной буквой. Это для того, чтобы библиотека и заголовочные файлы могли определять функции, переменные, и макрокоманды для внутренних целей без риска конфликта с именами в программах пользователя.

Некоторые дополнительные классы имен идентификатора зарезервированы для будущих расширений языка C. Использовании этих имен для ваших собственных целей прямо сейчас создает возможность конфликта с будущими версиями стандарта C, так что Вам следует избегать этих имен.

* Имена, начинающиеся с прописной буквы "E" с последующей цифрой, или символом верхнего регистра могут использоваться для дополнительных имен кода ошибки. См. Главу 2 [Сообщения об ошибках].

* Имена, которые начинаются с "is' или "к" сопровождаемые символом нижнего регистра, могут использоваться для дополнительного тестирования символа и функций преобразования. См. Главу 4 [Обработка символов].

* Имена, которые начинаются с "LC_" с последующим символом верхнего регистра, могут использоваться для дополнительных макрокоманд, определяя атрибуты стандарта. См. Главу 19 [Положения].

* Названия всех существующих функций математики (см. Главу 13 [Математика]) с прибавленным "f" или "l" зарезервированы для соответствующих функций, которые применяют к числам с плавающей точкой и параметрам двойной длины, соответственно.

* Имена, которые начинаются с "SIG", сопровождаемые символом верхнего регистра, зарезервированы для дополнительных имен сигнала. См. раздел 21.2 [Стандартные сигналы].

* Имена, которые начинаются с "SIG_" сопровождаемые символом верхнего регистра, зарезервированы для дополнительных действий сигнала. См. раздел 21.3.1 [Обработка видеосигнала].

* Имена, начинающиеся со "str", "mem", или "wcs", сопровождаемые символом нижнего регистра, зарезервированы для дополнительной строки и функций массива. См. Главу 5 [Уилиты для работы со строками и массивами].

* Имена, которые заканчиваются на "_t" зарезервированы для дополнительных имен типа.

Кроме того, некоторые индивидуальные заголовочные файлы резервируют имена вне тех, что они фактически определяют. Если ваша программа включает такой специфический заголовочный файл, то у Вас есть повод для беспокойства.

* Заголовочный файл "dirent.h" резервирует имена с предстоящим "d_".

* Заголовочный файл "fcntl.h" резервирует имена с предстоящими "l_", "F_", "O_", и "S_".

* Заголовочный файл "grp.h" резервирует имена с предстоящим "gr_".

* Заголовочный файл "limits.h" резервирует имена, с приписанным "_MAX".

* Заголовочный файл "pwd.h" резервирует имена с предстоящим "pw_".

* Заголовочный файл "signal.h" резервирует имена с предстоящим "sa_" и "SA_".

* Заголовочный файл "sys/stat.h" резервирует имена с предстоящим "st_" и "S_".

* Заголовочный файл "sys/times.h" резервирует имена с предстоящим

"tms_".

* Заголовочный файл "termios.h" резервирует имена с предстоящим "c_", "V", "I", "O", и "TC", а также имена с предстоящим "B", сопровождаемым цифрой.

1.3.4 Макрокоманды управления особенностями

Управляя макрокомандами, Вы определяете точный набор особенностей, доступный, когда при компиляции исходного файла.

Если Вы компилируете ваши программы, используя "gcc -ansi", Вы получаете только ANSI C библиотечные особенности, если Вы явно не запрашиваете дополнительные особенности, определяя одну или большее количество макрокоманд особенностей. См. раздел "Опции Команд GNU CC" в GNU CC Руководстве, для уточнения информации относительно опций GCC.

Вы должны определить эти макрокоманды, используя директивы препроцессора "#define" в начале ваших файлов. Эти директивы должны стоять перед любым #include заголовочного файла системы. Лучше всего указывать их самыми первыми в файле, только после комментариев. Вы могли бы также использовать опцию `-D' для GCC, но лучше, если Вы создаете исходные файлы с указанием их собственного значения замкнутым способом.

_POSIX_SOURCE (макрос)

Если Вы определяете эту макрокоманду, то функциональные возможности из стандарта POSIX.1 (Стандарт ИИЭРа 1003.1) доступны, также как все ANSI C средства.

_POSIX_C_SOURCE (макрос)

Если Вы определяете эту макрокоманду со значением 1, то будут определены функциональные возможности из стандарта POSIX.1 (Стандарт ИИЭР 1003.1). Если Вы определяете эту макрокоманду со значением 2, то станут доступны функциональные возможности из стандарта POSIX.1 и функциональные возможности из стандарта POSIX.2 (Стандарт ИИЭР 1003.2). Это - в дополнение к ANSI C средствам.

_BSD_SOURCE (макрос)

Если Вы определяете эту макрокоманду, включаются функциональные возможности 4.3 BSD Unix, также как ANSI C, POSIX.1, и POSIX.2.

Некоторые из особенностей происходящие от 4.3 BSD Unix конфликтуют с соответствующими особенностями, определенными стандартом POSIX.1. Если эта макрокоманда определена, 4.3 BSD определения, имеют больший приоритет чем POSIX определения.

Из-за некоторых конфликтов между 4.3 BSD и POSIX.1, Вы должны использовать специальную BSD библиотеку совместимости при компоновании программ, компилируемых для BSD совместимости. Т. к. некоторые функции должны быть определены двумя различными способами, один из них в нормальной библиотеке C, а другой в библиотеке совместимости. Если ваша программа определяет _BSD_SOURCE, Вы должны указать опцию "-lbsd-compat" транслятору или компоновщику при компоновании программы, чтобы он нашел функции в этой специальной библиотеке совместимости перед поиском их в нормальной библиотеке C.

_SVID_SOURCE (макрос)

Если Вы определяете эту макрокоманду, функциональные возможности из SVID, включены также как ANSI C, POSIX.1, и POSIX.2.

_GNU_SOURCE (макрос)

Если Вы определяете эту макрокоманду, включены все: ANSI C, POSIX.1, POSIX.2, BSD, SVID, и расширения GNU. В случаях, конфликтов

POSIX.1 с BSD, POSIX определения берут верх.

Если Вы хотите получить полный эффект _GNU_SOURCE, но установить BSD определениям больший приоритет необходимо примеенить следующую последовательность определений:

```
#define _GNU_SOURCE
```



```
#define _BSD_SOURCE
#define _SVID_SOURCE
```

Обратите внимание, что, если Вы делаете это, Вы должны компоновать вашу программу с BSD библиотекой совместимости, указывая опцию `-lbsd-compat' транслятору или компоновщику. Обратите внимание: если Вы забудете сделать это, Вы можете получить очень странные ошибки во время выполнения.

Мы рекомендуем, чтобы Вы использовали _GNU_SOURCE в новых программах. Если Вы не определяете опцию `-ansi' для GCC и не определяете никакую из этих макрокоманд явнно, то _GNU_SOURCE дает тот же эффект.

Когда Вы определяете макрокоманду, чтобы запросить больший класс особенностей, безобидно определить кроме того макрокоманду для подмножества этих особенностей. Например, если Вы определяете _POSIX_C_SOURCE, то определение _POSIX_SOURCE не имеет никакого эффекта. Аналогично, если Вы определяете _GNU_SOURCE, определяя затем либо _POSIX_SOURCE либо _POSIX_C_SOURCE либо _SVID_SOURCE, также не будет никакого эффекта.

Обратите внимание, что особенности _BSD_SOURCE не подмножество любой другой из обеспечиваемых макрокоманд особенностей. Потому что эта макрокоманда определяет особенности BSD, которые берут верх над особенностями POSIX, которые запрашиваются другими макрокомандами. По этой причине, определение _BSD_SOURCE в дополнение к другим макрокомандам особенностей заставляет особенности BSD брать верх при конфликтое с особенностями POSIX .

1.4 Путеводитель по руководству

Имеется краткий обзор содержания оставшихся глав этого руководства.

* Глава 2 [Сообщения об ошибках], описывает, как сообщаются ошибки, обнаруженные при использовании библиотекой.

* Глава 3 [Распределение памяти], описывает средства библиотеки GNU

для динамического распределения памяти. Если Вы не знаете заранее сколько памяти потребует ваша программа , Вы можете распределять память динамически, и управлять этим через указатели.

* Глава 4 [Обработка символов], содержит информацию относительно символьных функций классификаций (типа isspace) и функций для выполнения преобразования по выбору.

* Глава 5 [Утилиты для работы со строками и массивами], имеет описания функций для управления строками (символьными массивами, завершенными пустым указателем) и общими массивами байтов, включая операции типа копирования и сравнения.

* Глава 6 [Краткий обзор ввода - вывода], рассматривают средства ввода и вывода в библиотеке, и содержит информацию относительно базисных понятий, таких как имя файла.

* Глава 7 [Потоки ввода - вывода], описывает операции ввода - вывода, включая потоки (или FILE* объекты). Это обычные библиотечные функции C из "stdio.h".

* Глава 8 [Ввод - вывод низкого уровня], содержит информацию относительно операций ввода - вывода с описателями файла. Описатели

файла - механизм низшего уровня, специфический для UNIX семейства операционных систем.

* Глава 9 [Интерфейс файловой системы], имеет описания операций на всех файлах, например функций для их удаления и переименования и для создания новых каталогов. Эта глава также содержит информацию, относительно того, как Вы можете обращаться к атрибутам файла, например к режимам защиты файла.

* Глава 10 [Каналы и FIFO (первый зашел - первый вышел)], содержит информацию относительно простых межпроцессорных механизмов связи. Каналы обеспечивают связь между двумя связанными процессами (типа "родитель-порожденнным"), в то время как FIFO обеспечиваю связь между процессами, совместно используя общую файловую систему.

* Глава 11 [Гнеза], описывает более сложный межпроцессорный механизм связи, который позволяет процессам, выполняющимся на различных машинах, связаться через сетью. Эта глава также содержит информацию относительно Межсетевой адресации главной ЭВМ и как использовать базы данных сетей системы.

* Глава 12 [Интерфейс терминала низкого уровня], описывает, как Вы можете изменять атрибуты устройства терминала. Например, если Вы хотите отключать отображение символов, печатаемых пользователем, читайте эту главу.

* Глава 13 [Математика], содержит информацию относительно математических библиотечных функций. Они включают такие вещи как генераторы случайных чисел и функции остатка, а также обычные тригонометрические и показательные функции на числах с плавающей запятой.

* Глава 14 [Функции арифметики низкого уровня], описывают функции для простой арифметики, анализ чисел с плавающей запятой, и чтения чисел из строк.

* Глава 15 [Поиск и сортировка], содержит информацию относительно функций для поиска и сортировки массивов. Вы можете использовать эти функции для любого вида массива, обеспечивая соответствующую функцию сравнения.

* Глава 16 [Сопоставление с образцом], функции для определения соответствия регулярных выражений и образцов имени файла оболочки, и для расширяющихся слов, как делает оболочка.

* Глава 17 [Дата и время], описывает функции для измерения календарного времени и процессорного времени, также как функции для установки будильников и таймеров.

* Глава 18 [Расширения символов], содержит информацию относительно управления символами, и строками, используя набор символов больший чем, представлен в обычном типе данных char.

* Глава 19 [Положение], описывает как выбор специфической страны или языка воздействует на поведение библиотеки. Например, эффекты положения воздействуют на последовательности объединений для строк и на то как форматируются валютные значки.

* Глава 20 [Нелокальные выходы], содержит описания функций setjmp и longjmp. Эти функции обеспечивают средства для goto-подобных переходов, возможно из одной функции в другую.

* Глава 21 [Обработка сигнала], сообщает Вам все относительно сигналов, что это такое, как установить драйвер, который вызывается, когда специфический вид сигнала поставлен, и как предотвратить сигналы во время выполнения критических разделов вашей программы.

* Глава 22 [Запуск процесса], сообщает, как ваши программы могут обращаться к их параметрам командной строки и системным переменным.

* Глава 23 [Порожденнные процессы], содержит информацию относительно того, как начать новые процессы и выполнять программы.

* Глава 24 [Управление заданиями], описывает функции для управления группами процесса. Этот материал, вероятно, представляет интерес только, если Вы пишете оболочку.

* Раздел 25.12 [База данных пользователей], и раздел 25.13 [База данных групп], сообщают Вам, как обратиться к базам данных групп и пользователей системы.

* Глава 26 [Информация системы], описывает функции для получения информации относительно аппаратных средств и конфигурации программного обеспечения,когда выполняется ваша программа.

* Глава 27 [Конфигурация системы], сообщает Вам, как Вы можете получить информацию относительно различных ограничений операционной системы. Большинство этих параметров предусмотрено для совместимости с POSIX.

* Приложение A [Особенности языка], содержит информацию

относительно библиотечной поддержки для стандартных частей C языка, включая такие вещи как оператор sizeof или символическая постоянная NULL, форма записи функций, принимающих переменное числа параметров, и констант, описывающих диапазоны и другие свойства числовых типов. Имеется также простой механизм отладки, который разрешает поместить утверждения в ваш код, и напечатать диагностические сообщения, если тесты не прошли.

* Приложение B [Обзор библиотеки], дает обзор всех функций, переменных, и макрокоманд в библиотеке, с полными типами данных и функциональными прототипами, и говорит, от какого стандарта или системы каждая происходит.

* Приложение C [Сопровождение], объясняет, как формировать и установить GNU C библиотеку на вашей системе, куда сообщать о всех ошибках, которые Вы можете найти, и как добавить новые функции или перенести библиотеку на новоую систему.

Если Вы уже знаете имя средства, которое Вас интересует, Вы можете найти его в Приложении B [Обзор библиотеки]. Оно даст Вам обзор синтаксиса и указатель, по которому Вы можете найти более детализированное описание. Это приложение особенно полезно, если Вы хотите только проверить порядок и тип параметров функции. Оно также сообщает Вам, от какого стандарта или системы происходит каждая функция, переменная, или макрокоманда.

2. Сообщения об ошибках

Много функций в GNU C библиотеке обнаруживают и выводят ошибки условий, и иногда ваши программы должны проверить эти ошибки условий. Например, когда Вы открываете входной файл, Вы должны проверить, что файл был фактически открыт правильно, и печатать сообщение об ошибках или выполнять другое соответствующее действие, если обращение к библиотечной функции потерпело неудачу.

Эта глава описывает, как работают средства сообщений об ошибках. Чтобы использовать эти средства, ваша программа должна включить заголовочный файл "errno.h".

2.1 Проверка Ошибок

Большинство библиотечных функций возвращает специальное значение, чтобы указать, что они потерпели неудачу. Специальное значение типично -1, нулевой указатель, или константа типа EOF, которая определена для той цели. Но это значение возврата сообщает Вам только то, что ошибка произошла. Чтобы выяснять что это было, Вы должны рассмотреть код ошибки, сохраненный в переменной errno. Эта переменная объявлена в заголовочном файле "errno.h".

Переменная errno содержит номер ошибки системы. Вы можете изменять значение errno.

С тех пор как errno объявлена изменяемой, она может быть асинхронно изменена драйвером сигнала; см. раздел 21.4 [Определение драйверов]. Однако, правильно написанный драйвер сигнала сохраняет и восстанавливает значение errno, так что Вы вообще не должны волноваться относительно этой возможности, разве что при наaписании драйверов сигнала.

Начальное значение errno при запуске программы - нуль. Большинство библиотечных функций, когда они сталкиваются с некоторыми видами ошибок, помещают туда некоторое отличное от нуля значение. Эти ошибки условий перечислены для каждой функции. Если эти функции успешно выполняются они не изменяют errno; таким образом, значение errno после успешного обращения не обязательно нулевое, и Вы не должны использовать errno,

чтобы определить, потерпело ли обращение неудачу. Соответствующий способ проверки зарегистрирован для каждой функции. Если обращение неудачно, Вы может исследовать errno.

Многие библиотечные функции могут устанавливать errno отличным от нуля в результате вызова других библиотечных функций, которые возможно и установили ошибку. Т. е. если функция возвращает ошибку, то любая библиотечная функция могла изменять errno.

Примечание относительно переносимоси: ANSI C определяет errno скорее как "модифицируемое именуемое выражение", чем как переменную, разрешая ему выполняться как макрокоманде. Например, его расширение может включать обращение к функции, подобно * _errno (). Фактически, это встроено систему GNU непосредственно. GNU библиотека, на не-GNU системах, делает то, что правильно для этой специфической системы.

Имеются несколько библиотечных функций, подобно sqrt и atan, которые в случае ошибки возвращают ожидаемое значение, устанавивая также errno. Для этих функций, если Вы хотите выяснить, произошла ли ошибка, рекомендуется обнулить errno перед вызовом функции, и затем проверить значение позже.

Все коды ошибки имеют символические имена; т. е. это макрокоманды, определенные в "errno.h". Имена начинаются с "E" и символа верхнего регистра или цифры; Вы должны рассмотривать имена такой формы, как зарезервированные имена. См. раздел 1.3.3 [Зарезервированные имена].

Значения кода ошибки - это различные положительные целые числа, с одним исключением: EWOULDBLOCK и EAGAIN - имеют одинаковый код. Так как значения отличны, Вы можете использовать их как метки в утверждении выбора; только не используйте, и EWOULDBLOCK и EAGAIN. Ваша программа не должна иметь никаких сомнений относительно специфических значений этих символических констант.

Значение errno не обязательно должно соответствовать одной из этих макрокоманд, так как некоторые библиотечные функции могут возвращать другие их собственные коды ошибки для других ситуаций. Единственные значения, которые будут важны для специфической библиотечной функции -

это списки кодов ошибок для этой функции.

На не-GNU системах, почти любой системный вызов может возвратить EFAULT, если как параметр задан недопустимый указатель. Так как это могло случиться только в результате ошибки в вашей программе, и так как этого не будет в системе GNU, мы сэкономили место не упоминая EFAULT в описаниях индивидуальных функций.

2.2 Коды ошибок

Макрокоманды кода ошибки определены в заголовочном файле "errno.h". Каждая из них преобразуется в константное целое значение. Некоторые из этих ошибок не могут произойти в системе GNU, но они могут происходить при использовании библиотеки GNU в других системах.

int EPERM (макрос)

Не разрешенная операция; только владелец файла (или другого обьекта) или процессы со специальными привилегиями могут выполнять эту операцию.

int ENOENT (макрос)

Нет такого файла или каталога. Это - ошибка типа "файл не существует" для обычных файлов, которые вызваны в контекстах, где они, как ожидается, уже существуют.

int ESRCH (макрос)

Нет процесса соответствующего заданному.

int EINTR (макрос)

Прерванное обращение к функции; асинхронный сигнал предотвратил завершение обращения. Когда это случается, Вы должны попробовать снова вызвать функцию.

Вы можете выбрать показ резюме функций после сигнала EINTR; см.

раздел 21.5 [Прерванные Примитивы] .

int EIO (макрос)

Ошибка ввода-вывода, обычно используется для ошибок физического чтения или записи.

int ENXIO (макрос)

Нет такого устройства или адреса. Система попробовала использовать устройство, указанное файлом, который Вы определили, и не смогло найти это устройство. Это может означать, что файл устройства был установлен неправильно, или физическое устройство отсутствует или не правильно присоединено к компьютеру.

int E2BIG (макрос)

Список параметров слишком длинный; используется, когда параметры, переданные одной из функций (см. раздел 23.5 [Выполнение файла]) занимают слишком много пространства памяти. Это условие никогда не возникает в системе GNU.

int ENOEXEC (макрос)

Недопустимый формат исполняемого файла. Это условие обнаруживается запускаемыми функциями; см. раздел 23.5 [Выполнение файла].

int EBADF (макрос)

Плохой описатель файла; например, ввод - вывод на описателе, который был закрыт или чтение из описателя, открытого только для записи (или наоборот).

int ECHILD (макрос)

Не имеется никаких порожденнных процессов. Эта ошибка случается при

операциях, которые, как предполагается, управляют порожденнными процессами, когда не имеется ни каких процессов, для управления.

int EDEADLK (макрос)

Тупик; распределение ресурсов системы оценено как ситуация тупика. Система не гарантирует, что она будет обращать внимание на все такие ситуации. Эта ошибка означает, что Вам повезло; система могла зависнуть. См. раздел 8.11 [Блокировки файла].

int ENOMEM (макрос)

Нет доступной памяти. Система не может распределять виртуальную память, потому она полна.

int EACCES (макрос)

Отклоненное право; права файла не позволяют предпринятую операцию.

int EFAULT (макрос)

Плохой адрес; был обнаружен недопустимый указатель.

int ENOTBLK (макрос)

Не специальный файл, был дан в ситуации, которая требует блок файов. Например, при попытке установить обычный файл как файловую систему в UNIX дает эту ошибку.

int EBUSY (макрос)

Ресурсы заняты; ресурс системы, который не может быть разделен уже используется. Например, если Вы пробуете удалить файл, который является корнем установленной в настоящее время файловой системы, Вы получаете эту ошибку.

int EEXIST (макрос)

Файл существует; существующий файл был определен в контексте, где имеет смысл определять только новый файл.

int EXDEV (макрос)

Была обнаружена попытка, сделать неподходящую компоновку файловой системы. Это случается не только, когда Вы используете связи (см. раздел 9.3 [Сложные связи]) но также, когда Вы переименовываете файл (см. раздел 9.6 [Переименование файлов]).

int ENODEV (макрос)

Функции, которая ожидает специфический тип устройства, был дан неправильный тип устройства.

int ENOTDIR (макрос)

Был определен файл, а не каталог, когда требуется каталог.

int EISDIR (макрос)

Указан каталог; попытка открыть каталог для записи дает эту ошибку.

int EINVAL (макрос)

Недопустимый параметр. Используется, чтобы указать различные виды проблем с указанием неправильного параметра для библиотечной функции.

int EMFILE (макрос)

Текущий процесс имеет слишком много открытых файлов и не может открыть больше. Двойные описатели приводят к этому ограничению.

int ENFILE (макрос)

Имеется слишком много различных открытых экземпляров файла во всей системе. Обратите внимание, что любое число связанных каналов считается только одним открытым экземпляром файла; см. раздел 8.5.1 [Связанные каналы]. Эта ошибка никогда не происходит в системе GNU.

int ENOTTY (макрос)

Несоответствующая операция управления ввода - вывода, типа попытки устанавливать режимы терминала в обычном файле.

int ETXTBSY (макрос)

Попытка выполнить файл, который является в настоящее время открытым для записи, или записи в файл который в настоящее время выполняется. Это не является ошибкой в системе GNU; текст по мере необходимости копируется.

int EFBIG (макрос)

Файл слишком большой; размер файла больше чем позволено системой.

int ENOSPC (макрос)

Нет места на устройстве; операция записи в файл потерпела неудачу, потому что диск полон.

int ESPIPE (макрос)

Недопустимая операция установки.

int EROFS (макрос)

Была сделана попытка изменить что - нибудь в файловой системе только для чтения.

int EMLINK (макрос)

Слишком много связей; число связей одиночного файла слишком велико. Переименование может вызывать эту ошибку, если переименовываемый файл уже имеет максимальное число связей (см. раздел 9.6 [Переименование файлов]).

int EPIPE (макрос)

Разрушенный канал; не имеется процесса читающего с другого конца канала. Каждая библиотечная функция, которая возвращает этот код ошибки также генерирует сигнал SIGPIPE; если этот сигнал не обработан или не блокирован, то он завершает программу . Таким образом, ваша программа фактически не будет никогда видеть EPIPE, если она не обработала или не блокировала SIGPIPE.

int EDOM (макрос)

Ошибка области; использование математических функций, когда значение параметра не относится к области над которой функция определена.

int ERANGE (макрос)

Ошибка диапазона; использование математических функций, когда значение результата не представимо из-за переполнения.

int EAGAIN (макрос)

Ресурс временно недоступен; обращение может работать, если Вы пробуете позже.

int EWOULDBLOCK (макрос)

Операция, которая бы была блокировала предпринята на объекте, который имеет выбранный режим не-блокирования.

Примечание относительно переносимости: В 4.4BSD и GNU, EWOULDBLOCK

и EAGAIN - совпадают. Более ранние версии BSD (см. раздел 1.2.3 [Berkeley UNIX]) имеют два различных кода, и используют EWOULDBLOCK, чтобы указать операцию ввода-вывода, которая блокировала бы объект с набором режимов неблокирования, а EAGAIN для других видов ошибок.

int EINPROGRESS (макрос)

Операция, которая не может завершаться немедленно, была инициализирована в объекте, который имеет выбранный режим неблокирования. Некоторые функции, которые должны всегда блокировать (типа, connect ; см., раздел 11.8.1 [Соединение]) никогда не возвращает EWOULDBLOCK. Взамен, они возвращают EINPROGRESS, чтобы указать, что операция начата и займет некоторое время. Попытайтесь управлять объектом прежде, чем обращение завершится возвратив EALREADY.

int EALREADY (макрос)

Операция уже выполняется в объекте, который имеет выбранный режим неблокирования.

int ENOTSOCK (макрос)

Был определен файл,а не гнездо, когда требуется гнездо.

int EDESTADDRREQ (макрос)

Нет был обеспечен адрес адресата для операции гнезда.

int EMSGSIZE (макрос)

Размер сообщения, посланного на гнездо был больше чем обеспечиваемый максимальный размер.

int EPROTOTYPE (макрос)

Тип гнезда не поддерживает запрашиваемый протокол связи.

int ENOPROTOOPT (макрос)

Вы определили опцию гнезда, которая не имеет смысла для специфического протокола, используемого гнездом. См. раздел 11.11 [Опции гнезда].

int EPROTONOSUPPORT (макрос)

Область гнезда не поддерживает запрашиваемый протокол связи (возможно, потому что запрашиваемый протокол полностью недопустим.) См. раздел 11.7.1 [Создание гнезда].

int ESOCKTNOSUPPORT (макрос)

Тип гнезда не установлен.

int EOPNOTSUPP (макрос)

Операция, которую Вы запросили, не обеспечивается. Некоторые функции гнезда не имеют смысла для всех типов гнезд, а другие не имеют права выполнения для всех протоколов связи.

int EPFNOSUPPORT (макрос)

Семейство протоколов связи гнезда, которое Вы запросили, не обеспечивается.

int EAFNOSUPPORT (макрос)

Семейство адресов, заданное для гнезда несогласованно с протоколом, используемым на гнезде. См. Главу 11 [Гнезда].

int EADDRINUSE (макрос)

Запрашиваемый адрес гнезда - уже используется. См. раздел 11.3 [Адреса гнезда].

int EADDRNOTAVAIL (макрос)

Запрашиваемый адрес гнезда не доступен; например, Вы попробовали дать гнезду имя, которое не соответствует местному главному имени. См. раздел 11.3 [Адрес Гнезда].

int ENETDOWN (макрос)

Операция с гнездом потерпела неудачу, потому что нет сети.

int ENETUNREACH (макрос)

Операция гнезда потерпела неудачу, потому что подсеть, содержащая главную ЭВМ была недоступна.

int ENETRESET (макрос)

Сетевое соединение было сброшено, потому что отдаленная главная ЭВМ умерла.

int ECONNABORTED (макрос)

Сетевое соединение было прервано локально.

int ECONNRESET (макрос)

Сетевое соединение было закрыто по внешним причинам контроля над местной главной ЭВМ, например из-за неисправимого нарушения протокола.

int ENOBUFS (макрос)

Буфера ядра для операций ввода - вывода занят весь.

int EISCONN (макрос)

Вы пытаетесь соединить гнездо, которое уже соединено. См. раздел 11.8.1 [Соединение].

int ENOTCONN (макрос)

Гнездо не соединено с чем - нибудь. Вы получаете эту ошибку, когда Вы пробуете передавать данные на гнездо, без первого определения адресата для данных.

int ESHUTDOWN (макрос)

Гнездо уже было закрыто.

int ETIMEDOUT (макрос)

Операция гнезда с заданной блокировкой по времени не получила никакого ответа в течение периода блокировки по времени.

int ECONNREFUSED (макрос)

Отдаленная главная ЭВМ отказала в сетевом соединении (обычно из-за того, что не запущено запрашиваемое обслуживание).

int ELOOP (макрос)

При поиске имени файла столкнулись со слишком многими уровнями символических связей. Часто это указывает на цикл символических связей.

int ENAMETOOLONG (макрос)

Имя файла слишком длинное (больше чем PATH_MAX; см. раздел 27.6 [Ограничения для файлов]) или главное имя слишком длинное (в gethostname или sethostname; см. раздел 26.1 [Главная идентификация]).

int EHOSTDOWN (макрос)

Отдаленная главная ЭВМ для запрашиваемого сетевого соединения не реагирует.

int EHOSTUNREACH (макрос)

Отдаленная главная ЭВМ для запрашиваемого сетевого соединения не доступна.

int ENOTEMPTY (макрос)

Каталог, не пустой, а ожидался пустой каталог. Обычно эта ошибка происходит, когда Вы пробуете удалять каталог.

int EUSERS (макрос)

Файловая система спутана, потому что имеется слишком много пользователей.

int EDQUOT

Пользовательское дисковое пространство превышено.

int ESTALE (макрос)

Просроченная NFS программа обработки файла. Это указывает на внутренний беспорядок в NFS системе, который появляется из-за перестановок файловой системы на главной ЭВМ станции. Восстановление этого условия обычно требует переустановки файловой системы NFS на местной главной ЭВМ.

int EREMOTE (макрос)

Была сделана попытка NFS-подсоединения удаленной файловой системой с именем файла, которое уже определяет установленный файл NFS. (Эта ошибка воникает на некоторых операционных системах, но мы думаем что это будет работать правильно на системе GNU, делающей этот код ошибки невозможным.)

int ENOLCK (макрос)

Нет доступной блокировки. Это используется средствами закрытия файла; см. раздел 8.11 [Блокировки файла]. Эта ошибка никогда не происходит в системе GNU.

int ENOSYS (макрос)

Функция не выполнена. Некоторые функции имеют команды или определяющие их опции, которые не могут обеспечиваться во всех реализациях, и это - ошибка, которую Вы получаете, если Вы запрашиваете то, что не обеспечивается.

int EBACKGROUND (макрос)

В системе GNU, станции, обеспечивающие протокол терминала возвращают эту ошибку для некоторых операций, когда вызывающий оператор не входит в группу приоритетного процесса терминала. Пользователи обычно не видят эту ошибку, потому что функции типа чтения и записи транслируют ее в SIGTTIN или SIGTTOU сигнал. См. Главу 24 [Управление заданиями], для уточнения информации относительно групп процессов и этих сигналов.

int ED(макрос)

Опытный пользователь будет знать, что неправильно.

int EGREGIOUS (макрос)

Что Вы делаете?!!!

int EIEIO (макрос)

Идите домой и выпейте стакан теплого молока.

int EGRATUITOUS (макрос)

Этот код ошибки не имеет никакой цели.

2.3 Сообщения об ошибках

Библиотека имеет функции и переменные, разработанные, чтобы облегчить для вашей программы вывод информативных сообщений об ошибках в обычном формате. Функции strerror и perror дают Вам стандартное сообщение об ошибках для данного кода ошибки; переменная program_invocation_short_name дает Вам удобный доступ к имени программы, которая столкнулась с ошибкой.

char * strerror (int errnum) (функция)

функция strerror отображает код ошибки (см. раздел 2.1 [Прверка Ошибок]) заданный параметром errnum в описательную строку сообщения об ошибках. Значение возврата - указатель на эту строку.

Значение errnum обычно исходит из переменной errno.

Вы не должны изменять строку, возвращаемую strerror. Также, если Вы делаете последующие обращения к strerror, новая строка могла быть записана поверх старой. (Но гарантируется, что никакая библиотечная функция не вызовет strerror за вашей спиной.) Функция strerror объявлена в "string.h".

void perror (const char * message) (функция)

Эта функция печатает сообщение об ошибках в поток stderr; см. раздел 7.2 [Стандартные Потоки].

Если Вы вызываете perror с сообщением, которое является или нулевым указателем или пустой строкой, perror печатает сообщение об ошибках, соответствуя errno, добавляя конечный символ перевода строки.

Если Вы обеспечиваете не-нулевой параметр сообщения, то perror начинает вывод с этой строки. Она добавляет двоеточие и пробел, чтобы отделить сообщение от строки ошибки, соответствующей errno.

Функция perror объявлена в "stdio.h".

strerror и perror производят точно то же самое сообщение для любого

данного кода ошибки; точный текст изменяется от системы до системы. На системе GNU, сообщения довольно коротки; не имеются никаких многострочных сообщений или вложенных символов перевода строки. Каждое сообщение об ошибках начинается заголовочной буквой и не включает ни какой пунктуации завершения.

Примечание относительно совместимости: функция strerror - новая особенность ANSI C, многие более старые C системы не поддерживают эту функцию.

Множество программ, которые не читают ввод с терминала, разработаны, чтобы выйти, если любой системный вызов терпит неудачу. В соответствии с соглашением, сообщение об ошибках из такой программы должно начаться с имени программы. Вы можете найти это имя в переменной program_invocation_short_name; полное имя файла сохранено в переменной program_invocation_name:

char * program_invocation_name (переменная)

Значение этой переменной - имя, которое использовалось, чтобы вызвать программу, выполняющуюся в текущем процессе. Аналогично argv [0]. Обратите внимание, что это не обязательно какое-то полезное имя файла; часто она не содержит никаких имен. См. раздел 22.1 [Аргументы программы].

char * program_invocation_short_name (переменная)

Значение этой переменной - имя, которое использовалось, чтобы вызвать программу, выполняющуюся в текущем процессе, без имен каталогов. (То есть то-же что и в program_invocation_name минус все до последней наклонной черты вправо, если что-то есть в наличии.)

Библиотечные код инициализации устанавливает обе из этих переменных перед вызовом main.

Примечание относительно переносимости: Эти две переменные - расширения GNU. Если Вы хотите, чтобы ваша программа работала с библиотеками, не относящимися к GNU, Вы должны сохранить значение argv

[0] в main (основной программе), и удалить имена каталогов самостоятельно. Мы добавили эти расширения, чтобы сделать возможной написание замкнутых сообщений об ошибках подпрограммы, которые не требуют никакого явного сотрудничества с основной программой.

Имеется пример, показывающий, как обработать отказ открывать файл. Функция open_ sesame пробует открывать указанный файл для чтения и возвращает поток. Библиотечная функция fopen возвращает нулевой указатель, если она не может открыть файл по некоторым причинам. В той ситуации, open_sesame создает соответствующее сообщение об ошибках, используя функцию strerror, и завершает программу. Если мы хотим сделать другие вызовы из библиотек перед передачей кода ошибки к strerror, мы должны сохранить его в местной переменной, потому что те другие библиотечные функции могут записывать поверх errno.

```
 #include
 #include
 #include < stdlib.h >
 #include
 FILE * open_sesame (char *name)
  {
    FILE *stream;
    errno = 0;
    stream = fopen (name, "r");
    if (stream == NULL)
     {
       fprintf (stderr, "%s: Couldn't open file %s; %s\n",
         program_invocation_short_name, name, strerror (errno));
```

```
            exit (EXIT_FAILURE);
          }
        else return stream;
      }
```

3. Распределение памяти

Система GNU обеспечивает несколько методов для распределения пространства памяти при явном управлении программы. Они различны по общности и по эффективности.

* malloc производит общее динамическое распределение. См. раздел 3.3 [Беспрепятственное распределение].

* obstacks - другое средство, менее общее чем malloc, но более эффективное и удобное для стеко-подобного распределения. См. раздел 3.4 [Obstacks].

* Функция alloca позволяет Вам динамически распределять память, которая будет освобождена автоматически. См. раздел 3.5 [Размер автоматической переменной].

3.1 Концепции динамического распределения памяти

Динамическое распределение памяти - методика, в которой программы определяют, где сохранить некоторую информацию. Вы нуждаетесь в динамическом распределении, когда число блоков памяти, в которых Вы нуждаетесь, или то, как долго Вы продолжаете нуждаться в них, зависит от данных, с которыми Вы продолжаете работать.

Например, Вы можете нуждаться в блоке, чтобы сохранить строку прочитанную из входного файла; так как не имеется никаких ограничений, как долго нужно будет хранить строку, Вы должны динамически распределить память и динамически ее увеличивать, поскольку Вы читаете большее количество строк.

Или, Вам может понадобиться блок для каждой записи или каждого определения во входных сведениях; так как Вы не можете знать заранее, сколько их будет, Вы должны определять новый блок для каждой записи или определения, поскольку Вы их читаете.

Когда Вы используете динамическое распределение, распределение блока памяти представляет собой действие, которое программа запрашивает явно. Когда Вы хотите зарезервировать место, Вы вызываете функцию или

макрокоманду и определяете размер аргументом. Если Вы хотите освободить место, Вы вызываете другую функцию или макркоманду. Вы можете делать это всякий раз, когда Вы хотите, и так часто, как Вы хотите.

3.2 Динамическое Распределение в C

Язык C поддерживает два вида распределения памяти через переменные в программах C:

* Статическое распределение - то, что случается, когда Вы объявляете статическую переменную. Каждая статическая переменная определяет один блок места, фиксированного размера. Место размещено один раз, когда ваша программа начата, и никогда не освобождается.

* Автоматическое распределение происходит, когда Вы объявляете динамическую локальную переменную, например аргумент функции или местную переменную. Пространство для динамической локальной переменной резервируется, когда начинается выполнение составного утверждения, содержащего объявление, и освобождается, когда это составное утверждение, покидается.

В GNU C, длина автоматической памяти может быть выражением, которое изменяется. В других C реализациях, это должна быть константа.

Динамическое распределение не обеспечивается C переменными; не имеется никакого класса памяти "dynamic", и не может быть переменной C, чье значение сохранено в динамически размещенном месте. Единственный способ обратиться к динамически размещенному месту - через указатель. Т.к. это менее удобно, и фактический процесс динамического распределения требует большего количества компьютерного времени, программисты использует динамическое распределение только когда ни статическое ни автоматическое распределение применить невозможно.

Например, если Вы хотите зарезервировать динамически некоторое место, чтобы разместить struct foobar, Вы не можете объявлять переменную типа struct foobar, чье содержание - динамически размещенное место. Но Вы можете объявить переменную struct foobar * типа указатель и назначить ей адрес. Тогда Вы можете использовать операторы "*" и "->" для этой

переменной- указателя, чтобы обратиться по адресу:

```
{
 struct foobar *ptr= (struct foobar *) malloc (sizeof (struct foobar));
 ptr->name = x;
 ptr->next = current_foobar;
 current_foobar = ptr;
 }
```

3.3 Беспрепятственное распределение

Наиболее общее динамическое средство распределения - malloc. Оно разрешает Вам зарезервировать блоки памяти любого размера в любое время, увеличивать или уменьшать их, и освободждать блоки индивидуально в любое время (или не освобождать) .

3.3.1 Базисное распределение памяти

Чтобы зарезервировать блок памяти, вызовите malloc. Прототип для этой функции находится в "stdlib.h".

```
void * malloc (size _t size) (функция)
```

Эта функция возвращает указатель к только что размещенныому блоку size байтов длиной, или нулевому указателю если блок не мог бы быть размещен.

Содержание блока неопределено; Вы должны инициализировать его непосредственно (или использовать calloc; см. раздел 3.3.5 [Распределение очищенного места]). Обычно Вы приводите значение указателя к виду объекта, который Вы хотите сохранять в блоке. Здесь мы показываем такой пример, и инициализации место нулями, используя библиотечную функцию memset (см. раздел 5.4 [Копирование и конкатенация]):

```
struct foo *ptr;
```



```
. . .
ptr = (struct foo *) malloc (sizeof (struct foo));
if (ptr == 0) abort ();
memset (ptr, 0, sizeof (struct foo));
```

Вы можете сохранять результат malloc в любую переменную-указатель без приведения, потому что ANSI C автоматически преобразовывает void* в другой тип указателя когда необходимо. Но приведение необходимо в контекстах отличных от операторов назначения или если Вы хотите, чтобы ваш код выполнился в традиционном C.

Не забудьте, что при распределении пространства для строки, аргумент malloc должен быть один плюс длина строки. Это - потому что строка завершена символом \0, который не учтен в "длине" строки, но

нуждается в месте. Например:

```
char *ptr;
. . .
ptr = (char *) malloc (length + 1);
```

3.3.2 Примеры malloc

Если место не доступно, malloc возвращает нулевой указатель. Вы должны проверить значение каждого обращения к malloc. Полезно написать подпрограмму, которая вызывает malloc и сообщает ошибку, если значение - нулевой указатель, и возвращает результат только если значение отлично от нуля. Эта функция традиционно называется xmalloc:

```
void * xmalloc (size_t size)
{
 register void *value = malloc (size);
 if (value == 0) fatal ("virtual memory exhausted");
 return value;
}
```

Это реальный пример использования malloc (через xmalloc). Функция savestring будет копировать последовательность символов в завершенную пустым указателем строку:


```
char * savestring (const char *ptr, size_t len)
{
 register char *value = (char *) xmalloc (len + 1);
 memcpy (value, ptr, len);
 value[len] = '\0';
 return value;
}
```

Блок, который malloc дает Вам, уже выровнен и может содержать любой тип данных. В системе GNU, адрес всегда делится на восемь; если размер блока - 16 или больше, то адрес всегда делится на 16. Более высокая граница (типа границы страницы) необходима гораздо реже; для этих случаев, используйте memalign или valloc (см. раздел 3.3.7 [Выравниваемые блоки памяти]).

Обратите внимание, что память, размещенная после конца блока, вероятно будет в использовании для чего - нибудь еще; возможно для блока, уже размещенный другим обращением к malloc. Если Вы пытаетесь обрабатывать блок как дольше чем Вы установили, Вы можете разрушить данные, что malloc использует, чтобы следить за блоками, или Вы можете разрушить содержимое другого блока. Если Вы уже зарезервировали блок и обнаруживаете, что Вам нужен больший, используйте перераспределение (см. раздел 3.3.4 [Изменение размеров блока]).

3.3.3 Освобождение памяти, размещенной malloc

Когда Вы больше не нуждаетесь в блоке, который Вы создали malloc, используйте функцию free чтобы сделать блок доступным, для следующего резервировния. Прототип для этой функции находится в "stdlib.h".

void free (void * ptr) (функция)

Функция освобождает блок памяти, указанной в ptr.

void cfree (void * ptr) (функция)

Эта функция делает то же самое что и предыдущая. Она предусматривает совместимость снизу вверх с SunOS.

Освобождение блока изменяет содержание блока. Не ищите какие-либо данные (типа указателя на следующий блок в цепочке блоков) в блоке после его освобождения. Копируйте все необхоимое во вне блока перед его

освобождением! Вот пример соответствующего способа освободить все блоки в цепочке, и строки на которые они указывают:

```
struct chain {struct chain *next; char *name;}
void free_chain (struct chain *chain)
{
 while (chain != 0)
  {
   struct chain *next = chain->next;
   free (chain->name);
   free (chain);
   chain = next;
  }
}
```

Иногда, free может фактически возвращать память в операционную систему и делать процесс меньшим. Обычно, все это позволяет, вызывая позже malloc, многократно использовать место. Тем временем, место остается в вашей программе в списе свободных, и используется внутри malloc.

В конце программы, не имеется никаких указателей на освобожденые блоки, потому что все место программы было отдано обратно системе, когда процесс завершвлся.

3.3.4 Изменение размера блока

Часто Вы не уверены, насколько большой блок Вам будетнужен в конечном счете. Например, блок может быть буфером, что Вы используете, чтобы содержать строку читаемую из файла; неважно какой длины Вы делаете буфер первоначально, Вы можете сталкнуться со строкой, которая будет длиннее.

Вы можете делать блок длиннее, вызывая realloc. Эта функция объявлена в "stdlib.h".

void * realloc (void * ptr, size _t newsize) (функция)

Функция realloc изменяет размер блока с адресом ptr, на newsize.

Если место после конеца блока используется, realloc скопизует блок по новому адресу, где доступно большее количество свободного пространства. Значение realloc - новый адрес блока. Если блок должен передвигаться, realloc, копирует старое содержимое.

Если Вы передаете пустой указатель для ptr, realloc, ведет себя точно так же как "malloc (newsize)". Это может быть удобно, но остерегайтесь более старых реализаций (до ANSI C) которые не имеют права поддерживать это поведение, и будут вероятно что-то разрушать , когда realloc получит пустой указатель.

Подобно malloc, realloc может возвращать пустой указатель, если никакое пространство памяти не доступно, чтобы увеличить блок. Когда это случается, первоначальный блок остается нетронутым; он не изменяется и не перемещается.

В большинстве случаев это не имеет значения, что случается с первоначальным блоком, когда realloc терпит неудачу, потому что прикладная программа не может продолжаться, когда не хватает памяти, и единственное что, нужно сделать - это выдать сообщение о фатальной ошибке. Часто бывает удобно написать и использовать подпрограмму, традиционно называемую xrealloc, которая заботится о сообщениях об ошибках, как xmalloc делает для malloc:

```
void * xrealloc (void *ptr, size_t size)
{
 register void *value = realloc (ptr, size);
 if (value == 0) fatal ("Virtual memory exhausted");
  return value;
}
```

Вы можете также использовать realloc, чтобы уменьшить блок. Чтобы не связывать много пространства памяти, когда необходимо немного. Создание меньшего блока иногда требует копировать его, так что это может неудаться, если никакое другое место не доступно.

Если новый размер, который Вы определяете - такой же как старый, realloc, не изменит ничего и возвратит тот же самый адрес, который Вы дали.

3.3.5 Распределение очищенного места

Функция calloc резервирует память и обнуляет ее. Она объявлена в "stdlib.h".

void * calloc (size_t count, size _t eltsize) (функция)

Эта функция резервирует блок достаточно длинный, чтобы содержать вектор из count элементов, каждый размера eltsize. Содержимое очищается обнулением прежде чем calloc сделает возврат.

Вы можете определять calloc следующим образом:

```
void * calloc (size_t count, size_t eltsize)
{
 size_t size = count * eltsize;
 void *value = malloc (size);
 if (value != 0) memset (value, 0, size);
 return value;
}
```

Мы редко используем calloc сегодня, потому что она эквивалентна

простой комбинации других средств, которые больее часто используются. Это историческое средство, которое устаревает.

3.3.6 Обсуждение эффективности malloc

Чтобы лучше использовать malloc, нужно знать, что GNU версия malloc всегда назначает наименьший объем памяти в блоках, чьи размеры являются степенями двойки. Она хранит раздельные пулы для каждой степени двойки. И каждый занимает столько места сколько страница. Следовательно, если у Вас есть свободный выбор размера маленького блока, чтобы сделать malloc более эффективным, делайте его степенью двойки.

Если страница разбита для определенного размеа блока, она не может многократно использоваться для другого размера, если все блоки этого размера не освобождены. В многих программах маловероятно, что это случается. Таким образом, Вы можете иногда делать программы использующие память более эффективно, используя блоки того же самого размера для многих различных целей.

Когда Вы запрашиваете о блоках памяти рамера страницы или больших, malloc использует различную стратегию; она округляет сверху размер до нескольких размеров странцы, и может обьединять и разбивать блоки как необходимо.

Причина для двух стратегий то, что важно зарезервировать и освободить маленькие блоки так быстро как возможно, но для большого блока быстродействие менее важно, так как программа обычно тратит больше времени используя его. Больших блоков обычно меньше. Следовательно, для больших блоков, имеет смысл использовать метод, который занимает большее количество времени, чтобы минимизировать потраченное впустую место.

3.3.7 Распределение выравниваемых блоков памяти

- 45 -

Адрес блока, возвращенного malloc или realloc в системе GNU - всегда делится на восемь. Если Вы нуждаетесь в блоке, чей адрес - делится на более высокой степень двойки чем этот, используйте memalign или valloc. Эти функции объявлены в "stdlib.h".

С библиотекой GNU, Вы можете использовать free, чтобы освободить блоки которые возвращают memalign и valloc. Это не работает в BSD, т. к. BSD не обеспечивает ни какого способа освободить такие блоки.

void * memalign (size _t size, size _t boundary) (функция)

Функция memalign зарезервирует блок size байтов чей адрес - делится на boundary. Граница boundary должна быть степенью двойки! Функция memalign работает, вызывая malloc, чтобы зарезервировать несколько больший блок, и возвращает адрес внутри блока, который находится на заданной границе.

void * valloc (size _t size) (функция)

Подобна memalign с размером страницы заданным как значение второго аргумента:

```
void * valloc (size_t size)
{
 return memalign (size, getpagesize ());
}
```

- 46 -

3.3.8 Проверка непротиворечивости кучи

Вы можете указать, чтобы malloc проверила непротиворечивость динамической памяти, используя функцию mcheck. Эта функция - расширение GNU, объявленное в "malloc.h".

int mcheck (void (* abortfn) (void)) (функция)

Вызывая mcheck сообщает, чтобы malloc выполнил случайные проверки непротиворечивости. Они будут захватывать вещи типа письменного соглашения после конца блока, который был размещен malloc.

abortfn аргумент - это функция, которая вызвается, когда несогласованность найдена. Если Вы обеспечиваете пустой указатель, используется функция аварийного прекращения работы.

Слишком поздно начинать проверку, если Вы уже зарезервировали что - нибудь с помощью malloc. Так как mcheck в этом случае ничего не делает. Функция возвращает -1, если Вы вызываете ее слишком поздно, и 0 иначе (в случае успеха).

Самый простой способ вызывать mcheck вовремя состоит в том, чтобы использовать опцию "-lmcheck" когда Вы компонуете вашу программу; то Вы вообще не должны изменять вашу исходную программу.

3.3.9 Ловушки для резервирования памяти

GNU C библиотека позволяет Вам изменять поведение malloc, realloc, и free, определяя соответствующие функции ловушки. Вы можете использовать эти ловушки, чтобы отладить программы, которые используют динамическое резервирование памяти.

Переменные-ловушки объявлены в "malloc.h".

__realloc_hook (переменная)

Значение этой переменной - указатель на функцию, которе realloc использует всякий раз, когда вызывается. Вы должны определить эту

функцию, как:

```
void * function (void * ptr, size _t size)
```

__ free_hook (переменная)

Значение этой переменной - указатель на функцию, которую free использует всякий раз, когда вызывается. Вы должны определить эту функцию, как:

```
void function (void * ptr)
```

Вы должны удостовериться, что функция, которую Вы устанавливаете как ловушку для одной из этих функций, не вызывает заменяемую функцию рекурсивно без того, чтобы сначала восстановить старое значение ловушки! Иначе, ваша программа будет эастревать в бесконечной рекурсии.

Это пример показывает как правильно использовать __ malloc_hook. Мы устанавливаем функцию, которая выводит информацию каждый раз когда вызывается malloc.

```
static void *(*old_malloc_hook) (size_t);
static void * my_malloc_hook (size_t size)
{
 void *result; __malloc_hook = old_malloc_hook;
 result = malloc (size);
 __malloc_hook = my_malloc_hook;
 printf ("malloc (%u) returns %p\n", (unsigned  int) size, result);
 return result;
}
main ()
{
 ...
 old_malloc_hook = __malloc_hook;
 __malloc_hook = my_malloc_hook;
 ...
}
```


Функция mcheck (см. раздел 3.3.8 [Непротиворечивость кучи]) в ходе своей работы устанавливает такие ловушки.

3.3.10 Статистика резервирования памяти при помощи malloc

Вы можете получить информацию относительно динамического резервирования памяти, вызывая функцию mstats. Эта функция и связанный тип данных объявлены в "malloc.h"; они являются расширением GNU.

struct mstats (тип данных)

Этот структурный тип используется, чтобы возвратить информацию относительно динамической программы распределения памяти. Она содержит следующие поля:

size_t bytes_total

Это - полный размер памяти, управляемой malloc, в байтах.

size_t chunks_used

Это - число используемых кусков. (Программа распределения памяти внутренне получает куски памяти из операционной системы, и преобразует их в такие, которые удовлетворяют индивидуальным запросам malloc; см. раздел 3.3.6 [Эффективность и malloc])

size_t bytes_used

Это число используемых байтов.

size_t chunks_free

Это - число кусков, которые являются свободными - то есть которые были размещены операционной системой вашей программы, но теперь не используются.

size_t bytes_free

Это - число свободных байт.

struct mstats mstats (void) (функция)

Эта функция возвращает информацию относительно текущего динамического использования памяти в структуре типа struct mstats.

3.3.11 Обзор функций, имеющих отношение к функции malloc

Это обзор функций, которые имеют отношение к malloc:

void * malloc (size _t size)

Резервирует блок из size байт. См. раздел 3.3.1 [Базисное резервирование].

void free (void *addr)

Освобождает блок, предварительно размещенный malloc. См. раздел 3.3.3 [Освобождение после malloc].

void * realloc (void * addr, size _t size)

Делает блок, предварительно размещенный malloc больше или меньше, возможно, копируя его по новому расположению. См. раздел 3.3.4 [Изменение размеров блока].

void * calloc (size _t count, size _t eltsize)

Резервирует блок в count * eltsize байт, используя malloc, и обнуляет содержимое. См. раздел 3.3.5 [Распределение очищенного места].

void * valloc (size _t size)

Зарезервирует блок в size байт, начинающийся на границе страницы. См. раздел 3.3.7 [Выравниваемые блоки памяти].

void * memalign (size _t size, size _t boundary)

Резервирует блок в size байт, начинающийся с адреса, который является делится на выравнивание. См. раздел 3.3.7 [Выравниваемые блоки памяти].

int mcheck (void (* abortfn) (void))

Указывает, чтобы malloc выполнил случайную проверку

непротиворечивости динамически размещенной памяти, и вызыватл abortfn, если найдена несогласованность. См. раздел 3.3.8 [Непротиворечивость кучи].

void * (* __ malloc_hook) (size _t size)

Указатель на функцию, которую malloc использует всякий раз, когда вызывается.

void * (* __ realloc _hook) (void * ptr, size _t size)

Указатель на функцию, которую realloc использует всякий раз, когда вызывается.

void (* __ free _hook) (void * ptr)

Указатель на функцию, которую free использует всякий раз, когда вызывается.

struct mstats mstats (void)

Возвращают информацию относительно текущего динамического исполнения памяти. См. раздел 3.3.10 [Статистика malloc].

3.4 obstacks

obstack - пул памяти, содержащей стек объектов. Вы можете создавать любое число различных obstacks, и зарезервировать объекты в заданном obstacks. Внутри каждого obstack, последний размещенный объект должен всегда быть первым освобожденным, но различные obstacks независят от друг друга.

Кроме этого одного ограничения на порядок освобождения, obstacks полностью свободен: obstack может содержать любое число объектов любого размера. Они выполнены как макрокоманды, так что резервирование обычно очень быстро, так как объекты обычно маленькие. И единственные дополнительные непроизводительные затраты на объект необходимы, чтобы начать каждый объект с подходящей границы.

3.4.1 Создание obstacks

Утилиты для управления obstacks объявлены в заголовочном файле "obstack.h".

struct obstack (тип данных)

obstack представляется структурой данных типа struct obstack. Эта структура имеет маленький фиксированный размер; она содержит состояния obstack и информацию о том, как как найти место, в котором размещены объекты. Она не содержит ни какой из объектов непосредственно. Вы не должны пробовать непосредственно обращаться к содержимому структуры; используйте только функции, описанные в этой главе.

Вы можете объявлять переменные типа struct obstack и использовать их как obstacks, или Вы можете зарезервировать obstacks динамически подобно любому другому виду объекта. Динамическое резервирование obstacks разрешает вашей программе иметь переменное число различных стеков. (Вы можете даже зарезервировать структуру obstack в другом obstack, но это редко требуется.)

Все функции, которые работают с obstacks, требуют, чтобы Вы определили, какой obstack использовать. Вы делаете это указателем типа struct obstack *. Далее, мы часто будем говорить "obstack" когда, строго

говоря, имеем в виду такой указатель.

Объекты в obstack упакованы в большие блоки называемые кусками. Структура struct obstack указывает на цепочку используемых в настоящее время кусков.

Библиотека obstack получает новый кусок всякий раз, когда Вы резервируете объект, которому не удовлетворяет предыдущий кусок. Так как библиотека obstack управляет кусками автоматически, Вы не должны уделять им много внимания, но Вы должны обеспечить функцию, которую библиотека obstack должна использовать, чтобы получить кусок. Обычно Вы обеспечиваете функцию, которая использует malloc непосредственно или косвенно. Вы должны также обеспечить функцию освобождения куска. Эти вопросы описаны в следующем разделе.

3.4.2 Подготовка к использованию obstacks

В каждый исходный файл, в котором Вы планируете использовать функции obstack, следует включить заголовочный файл "obstack.h":

```
#include
```

Также, если исходный файл использует макрокоманду obstack_init, он должен объявить или определить две функции или макрокоманды, которые будут вызываться библиотекой obstack. Это, obstack_chunk_alloc, которая используется, чтобы зарезервировать куски памяти, в которую объекты упакованы. Другая, obstack_chunk_free, которая используется, чтобы возвратить куски, когда объекты из них освобождены.

Обычно они определены, чтобы использовать malloc через посредника xmalloc (см. раздел 3.3 [Беспрепятственное Резервирование]). Это выполнено следующей парой макроопределений:

```
#define obstack_chunk_alloc xmalloc*

#define obstack_chunk_free free
```

Хотя память, которую Вы получаете используя obstacks действительно,

исходит из malloc, использование obstacks - быстрее, потому что malloc вызывается менее часто, для больших блоков памяти. См. раздел 3.4.10 [Куски obstack], для полной информации.

Во время выполнения, прежде, чем программа может использовать struct obstack object как obstack, она должна инициализировать obstack, вызывая obstack_init.

```
void obstack_init (struct obstack * obstack_ptr) (функция)
```

Инициализирует obstack obstack_ptr для резервирования объектов.

Имеются два примера того, как зарезервировать пространство для obstack и инициализировать его. Первый, obstack, который является статической переменной:

```
static struct obstack myobstack;
...
obstack_init (&myobstack);
```

Во-вторых, obstack, который самостоятельно динамически размещен:

```
struct obstack * myobstack_ptr= (struct obstack *) xmalloc (sizeof
(struct obstack));

obstack_init (myobstack_ptr);
```

3.4.3 Резервирование в obstack

Наиболее прямой способ зарезервировать объект в obstack - через obstack_alloc, которая вызывается почти как malloc.

```
void * obstack_alloc (struct obstack * obstack_ptr, size _t size)
(функция)
```

Резервирует неинициализированный блок size байт в obstack и возвращает адрес. Здесь obstack_ptr определяет, в каком obstack зарезервировать блок; это - адрес struct obstack object, который

представляет obstack. Каждая функция obstack или макркоманда требует, чтобы Вы определили obstack_ptr как первый аргумент.

Например функция, которая резервирует копию строки str в определенном obstack, который находится в переменной string_obstack:

```
struct obstack string_obstack;
char * copystring (char *string)
{
 char *s = (char *) obstack_alloc (&string_obstack, strlen (string) + 1);
 memcpy (s, string, strlen (string));
 return s;
}
```

Чтобы зарезервировать блок с заданным содержимым, используйте функцию obstack_copy, объявляемую так:

```
void * obstack_copy (struct obstack * obstack_ptr, void *address ,
size_t size) (функция)
```

Резервирует блок и инициализирует его, копируя size байтов данных, начинающихся по адресеу

```
void * obstack_copy0 (struct obstack * obstack_ptr, void *address ,
size _t size) (функция)
```

Подобна obstack_copy, но конкатенирует дополнительный байт, содержащий пустой символ. Этот дополнительный байт не учтен в размере аргумента.

Функция obstack_copy удобна для копирования последовательности символов в obstack как законченной пустым символом строки. Пример использования:

```
char * obstack_savestring (char * addr, size _t size)
```



```
{
 return obstack_copy (myobstack, addr, size);
}
```

Сравните это с предыдущим примером сохранения строки, использующим malloc (см. раздел 3.3.1 [Базисное резервирование]).

3.4.4 Освобождение объектов из obstack

Для освобждения объекта, размещенный в obstack, используйте функцию obstack_free. Так как obstack - стек объектов, освобождение одного объекта автоматически освобождает все другие объекты, зарезервированные позже в том же самом obstack.

```
void obstack_free (struct obstack * obstack_ptr, void * object)
(функция)
```

Если object - пустой указатель, все размещенные в obstack освобождаются. Если, object есть адрес объекта, размещенного в obstack, то объект освобождается, наряду с всеми размещенноми в obstack начиная с object.

Обратите внимание, что, если object является пустым указателем, то результат - неинициализированный obstack. Для освобождения всей памяти в obstack, с возможным его использованием для дальнейшего резервирования, вызовите obstack_free с адресом первого объекта, размещенного в obstack:

```
obstack_free (obstack_ptr, first_object_allocated_ptr);
```

Обратите внимание, что объекты в obstack сгруппированы в куски. Когда все объекты в куске становятся свободными, библиотека obstack автоматически освобождает кусок (см. раздел 3.4.2 [Подготовка obstack]). Тогда другое obstack-, или не obstack-резервирование, может

многократно использовать куска, занимаемое куском.

3.4.5 Функции и макросы obstack

Интерфейсы для использования obstacks могут быть определены или как функции или как макрокоманды, в зависимости от транслятора. obstack средство работает со всеми C трансляторами, включая, и ANSI C и традиционный C, но имеются предосторожности, которые Вы должны соблюдать, если Вы планируете использовать трансляторы отличные от GNU C.

Если Вы используете традиционный не ANSI C транслятор, все obstack "функции" фактически определены только как макрокоманды. Вы можете вызывать эти макркоманды подобно функциям, но Вы не можете использовать их любым другим способом (например, Вы не можете брать их адрес).

Вызов макрокоманд требует некоторых предосторожностей: а именно, первый операнд (указатель на obstack) не имеет права содержать никаких побочных эффектов, потому что он может быть вычислен больше чем один раз. Например, если Вы напишете: obstack_alloc (get_obstack (), 4), Вы увидите, что get_obstack может называться несколько раз. Если Вы используете * obstack_list_ptr ++ как аргумент-указатель obstack, Вы получите очень странные результаты, так как приращение может происходить несколько раз.

В ANSI C, каждая функция имеет, и макроопределение и определение функции. Определение функции используется, если Вы берете адрес функции без того, чтобы вызвать ее. Обычное обращение использует макроопределение по умолчанию, но Вы можете запрашивать определение функции, написав имя функции в круглых скобках, как показано здесь:

```
char * x; void * (* funcp) (); /* Используем макркоманду. */
x = (char *) obstack_alloc (obptr, size);  /* Вызываем функцию. */
x = (char *) (obstack_alloc) (obptr, size);  /* Берем адрес функции. */
funcp = obstack_alloc;
```

Это - та же самая ситуация, что и в ANSI C для стандартных библиотечных функций. См. раздел 1.3.2 [Определение макросов].

Предупреждение: Когда Вы используете макрокоманды, Вы должны

соблюдать предосторожности избавившись от побочных эффектов в первом операнде, даже в ANSI C.

Если Вы используете транслятор GNU C, эта предосторожность, необязательна, потому что различные расширения языка в GNU C разрешают определять макрокоманды, чтобы вычислять каждый аргумент только один раз.

3.4.6 Возрастающие объекты

Т.к. память в кусках obstack используется последовательно, возможно создать объект шаг за шагом, добавляя один или больше байт сразу к концу объекта. При таком способе, Вы не должны знать, сколько данных у Вас будет включено в объект до этого. Мы называем это методикой возрастастающих объектов. В этом разделе описаны специальные функции для добавления данных к возрастастающему объекту.

Вы не должны делать что-нибудь особенное, когда Вы начинаете наращивать объект. Использование одной из функций, чтобы добавить данные к объекту автоматически начинает его. Однако, необходимо явно указать, когда объект закончен. Это выполняется функцией obstack_finish.

Действительный адрес объекта, созданного таким образом не известен, пока объект не закончен. До тех пор, всегда остается возможность, что Вы добавите так много данных, что объект придется скопировать в новый кусок.

Поскольку obstack используется для возрастастающего объекта, Вы не можете использовать obstack для обычного резервирования другого объекта. Если, Вы попробуете это сделать, то место, уже добавленное к возрастастающему объекту, станет частью другого объекта.

void obstack_blank (struct obstack * obstack_ptr, size _t size) (функция)

Базисной функцией для добавления к возрастастающему объекту является obstack_blank, которая добавляет место без его инициализированя.

void obstack_grow (struct obstack * obstack_ptr, void *data , size _t size) (функция)

Добавляет блок инициализированного места, используя obstack_grow, которая является аналогом obstack_copy для возрастающих объектов. Она добавляет size байт данных к возрастастающему объекту, копируя содержимое из данных.

void obstack_grow0 (struct obstack * obstack_ptr, void *data , size_t size) (функция)

Это - аналог obstack_copy для возрастающих объектов. Она добавляет size байт копируемых из данных, добавляя дополнительный пустой символо.

void obstack_1grow (struct obstack * obstack_ptr, char c) (функция)

Чтобы добавить только один символ, используется функция obstack_1grow. Она добавляет одиночный байт с к возрастастающему объекту.

void * obstack_finish (struct obstack * obstack_ptr) (функция)

Когда Вы закончили увеличивать объект, используйте функцию obstack_finish, чтобы закрыть его и получить конечный адрес.

Если Вы закончили объект, obstack доступен для обычного резервирования или для увеличения другого объекта.

Когда Вы формируете объект, Вы будете вероятно должны знать, позже сколько места занял. Вы не должны следить за этим, потому что Вы можете выяснять длину из obstack перед самым окончанием объекта функцией obstack_object_size, объявленный следующим образом:

size _t obstack_object_size (struct obstack * obstack_ptr) (функция)

Эта функция возвращает текущий размер возрастастающего объекта, в

байтах. Не забудьте вызывать эту функцию перед окончанием объекта. После того, как он закончен, obstack_object_size будет возвращать нуль.

Если Вы начали увеличивать объект и желаете отменить это, Вы должны закончить его и тогда освободить его, примерно так:

obstack_free (obstack_ptr, obstack_finish (obstack_ptr));

Это не имеет никакого эффекта, если никакой объект не возрастастал.

Вы можете использовать obstack_blank с отрицательным аргументом size, чтобы делать текущий объект меньше. Только не пробуйте сокращать его до меньше нуля, не имеется никаких сведенний, что случиться, если Вы сделаете это.

3.4.7 Сверхбыстро возрастастающие объекты

Обычные функции для возрастастающих объектов делают непроизводительные затраты для проверки, имеется ли участок памяти для

нового роста в текущем куске. Если Вы часто создаете объекты в с маленькими шагами роста, эти непроизводительные затраты, могут быть значительны.

Вы можете уменьшать непроизводительные затраты, используя специальный "быстрый рост", т. е. функции, которые выращивают объект без проверки. Чтобы иметь здоровую программу, Вы должны делать проверку самостоятельно. Если Вы делаете это самым простым способом каждый раз. когда Вы собираетесь, добавлять данные к объекту, Вы ничего не приобретете, потому что это и делают обычные функции возрастания. Но если Вы можете проверить менее часто, или проверять более эффективно, то Вы сделаете программу быстрее.

Функция obstack_room возвращает количество памяти, доступной в текущем куске. Она объявлена следующим образом:

size _t obstack_room (struct obstack * obstack_ptr) (функция)

Возвращает число байтов которые могут быть добавлены к текущему

возрастастающему объекту (или к объекту, собирающемуся начаться) в obstack при использовании быстрых функций роста.

Если Вы знаете, что имеется участок памяти, Вы можете использовать эти быстрые функции роста для добавления данных к возрастастающему объекту:

void obstack_1grow_fast (struct obstack * obstack_ptr, char c) (функция)

Функция obstack_1grow_fast добавляет один байт, содержащий символ с к возрастастающему объекту в obstack obstack_ptr.

void obstack_blank_fast (struct obstack * obstack_ptr, size _t size) (функция)

Функция obstack_blank_fast добавляет size байтов к возрастастающему объекту в obstack obstack_ptr без их инициализации.

Если Вы проверяете место, используя obstack_room и нет достаточного участка памяти, для того, что Вы хотите добавлять, то быстрые функции роста не безопасны. В этом случае, просто используйте соответствующую обычную функцию роста. Если она будет копировать объект в новый кусок; то будет иметься больше доступной памяти.

Так, каждый раз когда Вы используете обычную функцию роста, проверяйте есть ли достаточного места, используя obstack_room. Как только объект скопирован в новый кусок, будет снова иметься много места, так что программа будет начинать использовать быстрые функции роста.

Вот пример:

```
void add_string (struct obstack * obstack, char * ptr, size _t len)
{
 while  (len > 0)
 {
  if (obstack_room (obstack) > len)
   {          /* Мы имеем достаточный участок памяти: добавляйте все
```



```
                    быстро. */
     while (len - > 0) obstack_1grow_fast (obstack, * ptr ++);
   }
  else
   {
/* Нет достаточного участка памяти. Добавьте один символ, он может
быть скопирован в новый кусок для создания места. */
    obstack_1grow (obstack, * ptr ++); len -;
   }
 }
}
```

3.4.8 Состояние obstack

Имеются функции, которые обеспечивают информацию относительно текущего состояния резервирования в obstack. Вы можете использовать их, чтобы узнать относительно объекта при все еще возрастании это.

void * obstack_base (struct obstack * obstack_ptr) (функция)

Эта функция возвращает временный адрес начала в настоящее время возрастастающего объекта в obstack_ptr. Если Вы заканчите объект немедленно, он будет иметь этот адрес. Если Вы сначала его увеличиваете, он может перерасти этот кусок и адрес будет изменяться!

Если никакой объект в данный момент не возрастастает, это значение показывает, где будет начинаться следующий объект, который Вы зарезервируете.

void * obstack_next_free (struct obstack * obstack_ptr) (функция)

Эта функция возвращает адрес первого свободного байта в текущем куске obstack obstack_ptr, т. е. - конца в настоящее время возрастастающего объекта. Если никакой объект не возрастастает, obstack_next_free, возвращает то же самое значение что и obstack_base.

size _t obstack_object_size (struct obstack * obstack_ptr)

(функция)

Эта функция возвращает размер в байтах возрастастающего в настоящее время объекта. Она эквивалентна toobstack_next_free (obstack_ptr) - obstack_base (obstack_ptr)

3.4.9 Выравнивание данных в obstacks

Каждый obstack имеет границу выравнивания; каждый объект, размещенный в obstack автоматически начинается на адресе, который делится на заданную границу (выравнивание). По умолчанию, эта граница - 4 байта.

Чтобы обращаться к границе выравнивания obstack, используйте макркоманду obstack_alignment_mask, чей функциональлный прототип

int obstack_alignment_mask (struct obstack * obstack_ptr) (макрос)

Его значение - битовая маска; установленная еденица, указывает, что соответствующий бит в адресе объекта должен быть 0. Значение маски должно быть на еденицу меньше чем степень 2; и все адреса объекта - делится на эту степень 2. По умолчанию значение маски - 3, чтобы адреса - делились на 4. Значение маски 0 означает, что объект может начинаться с любого целого адреса (то-есть никакое выравнивание не требуется).

Расширение макрокоманды obstack_alignment_mask - именующее выражение, так что Вы можете изменять маску назначением. Например, это утверждение:

obstack_alignment_mask (obstack_ptr) = 0;

Выключает обработку выравнивания в заданном obstack.

Обратите внимание, что изменение в маске выравнивания не действует до окончания или размещения следующего объекта в obstack. Если Вы не увеличиваете объект, Вы может заставить новую маску выравнивания воздействовать немедленно, вызывая obstack_finish. Она заканчит объект нулевой длины и то сделает соответствующее выравнивание для следующего

объекта.

3.4.10 Куски obstack

obstacks работает, резервируя пространство для себя в больших

кусках, и разбивает место снаружи на куски, чтобы удовлетворить ваши запросы. Куски - обычно длиной 4096 байтов, если Вы не определите другой размер куска. Размер куска включает 8 байтов непроизводительных затрат, которые фактически не используются для сохранения объектов. Независимо от заданного размера, длинные объекты будут размещены в более длинные куски, когда необходимо.

obstack библиотека зервирует куски, вызывая функцию obstack_chunk_alloc, которую Вы должны определить. Когда кусок больше не нужен, если Вы освободили в нем все объекты, obstack библиотека освобождает кусок, вызывая obstack_chunk_free, которыую Вы должны также определить.

Эти две функции должны быть определены (как макрокоманды) или объявляться (как функции) в каждом исходном файле, который использует obstack_init (см. раздел 3.4.1 [Создание obstacks]). Наиболее часто они определены как макркоманды подобно:

```
#define obstack_chunk_alloc xmalloc
#define obstack_chunk_free free
```

Обратите внимание, что это простые макркоманды (никаких аргументов). Определения макросов с аргументами работать не будут! Необходимо чтобы obstack_chunk_alloc или obstack_chunk_free, расширялась в имя функции, если она не является именем функции.

Функция, которая фактически осуществляет obstack_chunk_alloc, не может возвратить "отказ" в любом режиме, потому что obstack библиотека не подготовлена, чтобы обработать отказ. Следовательно, malloc непосредственно не подходит. Если функция не может получить место, она должна также завершить процесс (см. раздел 22.3 [Прерывание программ] или делать нелокальный выход, используя longjmp (см. Главу 20 [Нелокальные выходы]).

Если Вы зарезервируете куски с malloc, размер куска, должен быть степенью 2. Заданный по умолчанию размер куска - 4096, был выбран, достаточно большим чобы удовлетворить много типичных запросов на obstack однако достаточно коротким, чтобы не тратить впустую слишком много памяти.

```
size _t obstack_chunk_size (struct obstack * obstack_ptr) (макрос)
```

Он возвращает размер куска данного obstack.

Так как эта макркоманда расширяется до именующего выражения, Вы можете определить новый размер куска, назначая новое значение. Ее выполнение не подействует на куски, уже размещенные, но изменит размер кусков, размещенных в том конкретном obstack в будущем., вряд ли, будет полезно сделать размер куска меньше, но создание большего могло бы увеличивать эффективность, если Вы зарезервируете много объектов, чьи размеры являются сравнимыми с размером куска. Вот как это можно сделать:

```
if (obstack_chunk_size (obstack_ptr) < new_chunk_size)
 obstack_chunk_size (obstack_ptr) = new_chunk_size;
```

3.4.11 Обзор функций, имеющих отношение к obstack

Это обзор всех функций, связанных с obstack. Каждая берет в качестве первого аргумента адрес obstack (struct obstack *).

```
void obstack_init (struct obstack * obstack'ptr)
```

Инициализирует использование obstack. См. раздел 3.4.1 [Создание obstacks].

```
void * obstack_alloc (struct obstack * obstack'ptr, size_t size)
```

Резервирует объект как size неинициализированных байт. См. раздел 3.4.3 [Резервирование в obstack].

```
void * obstack_copy (struct obstack * obstack'ptr, void *address,
```

size_t size)

Резервирует объект из size байтов, с содержимым, скопированным из адреса address. См. раздел 3.4.3 [Резервирование в obstack].

void * obstack_copy0 (struct obstack * obstack'ptr, void *address, size_t size)

Резервирует объект из size + 1 байт, size из которых скопированы из адреса address, сопровождаемый пустым символом в конце. См. раздел 3.4.3 [Резервирование в obstack].

void obstack_free (struct obstack * obstack'ptr, void * object)

Освобождает обьект (и все размещенное в заданном obstack позже чем object). См. раздел 3.4.4 [Освобождение obstack объектов].

void obstack_blank (struct obstack * obstack'ptr, size_t size)

Добавляет size неинициализированных байтов к возрастастающему обьекту object. См. раздел 3.4.6 [Возрастающие объекты].

void obstack_grow (struct obstack * obstack'ptr, void * address, size _t size)

Добавляет size байт, скопированных из address, к возрастастающему обьекту object. См. раздел 3.4.6 [Возрастастающие объекты].

void obstack_grow0 (struct obstack * obstack'ptr, void * address, size _t size)

Добавляет size байт, скопированных из address, к возрастастающему обьекту object, и еще добавляет другой байт, содержащий пустой символ. См. раздел 3.4.6 [Возрастастающие объекты].

void obstack_1grow (struct obstack * obstack'ptr, char data'char)

Добавляет один байт данных к возрастастающему обьекту object. См. раздел 3.4.6 [Возрастастающие объекты].

void * obstack_finish (struct obstack * obstack'ptr)

Завершает объект, который возрастастает и возвращает постоянный address. См. раздел 3.4.6 [Возрастающие объекты].

size _t obstack_object_size (struct obstack * obstack'ptr)

Получает текущий размер в настоящее время возрастастающего объекта. См. раздел 3.4.6 [Возрастающие объекты].

void obstack_blank_fast (struct obstack * obstack'ptr, size _t size)

Добавляет size неинициализированных байт к возрастастающему объекту без проверки, что имеется достаточный участок памяти. См. раздел 3.4.7 [Сверхбыстро возрастастающие объекты].

vid obstack_1grow_fast (struct obstack * obstack'tr, char data'char)

Добавляет один байт к возрастастающему объекту без проверки, что имеется достаточный участок памяти. См. раздел 3.4.7 [Сверхбыстро возрастастающие объекты].

size _t obstack_room (struct obstack * obstack'ptr)

Получает участок памяти, теперь доступный для возрастания текущего объекта. См. раздел 3.4.7 [Сверхбыстро возрастающие объекты].

int obstack_alignment_mask (struct obstack * obstack'ptr)

Маска, используемая для выравнивания начала объекта. Это - именующее выражение (адрес переменной). См. раздел 3.4.9 [Выравнивание данных obstacks].

size _t obstack_chunk_size (struct obstack * obstack'ptr)

Размер распределяемых кусков. Это - именующее выражение. См. раздел 3.4.10 [Куски obstack].

void * obstack_base (struct obstack * obstack'ptr)

Пробный начальный адрес в настоящее время возрастастающего объекта. См. раздел 3.4.8 [Состояние obstack].

void * obstack_next_free (struct obstack * obstack'ptr)

Адрес следующий сразу за концом в настоящее время возрастастающего объекта. См. раздел 3.4.8 [Состояние obstack].

3.5 Автоматическая память с учетом размера переменной

Функция alloca поддерживает вид полудинамического резервирования, в котором блоки размещены динамически, но освобождаются автоматически.

Распределение блока alloca - явное действие; Вы можете зарезервировать так много блоков, как Вы желаете, и вычислять размеры во время выполнения. Но все блоки освобождатся, когда Вы выходите из функции из которой alloca вызывалась,как если бы они были динамические локальные переменные, объявленные в этой функции. Не имеется никакого способа освободить место явно.

Прототип для alloca находится в "stdlib.h". Эта функция - BSD расширение.

void * alloca (size _t size); (функция)

Возвращаемое значение alloca - адрес блока из size байтов памяти, размещенного в области данных вызывающей функции.

Не используйте alloca внутри аргументов обращения к функции, Вы

получите непредсказуемые результаты, потому что стек-пространство для alloca появится на стеке в середине пространства для аргументов функции. Пример того, что нужно избегать - foo (x, alloca (4), y).

3.5.1 Примеры alloca

Пример использования alloca, это функция, которая открывает имя файла, сделанное из связывания двух строк аргумента, и возвращает описатель файла или минус один выражая отказ:

```
intopen2 (char * str1, char * str2, int flags, int mode)
{
 char *name = (char *) alloca (strlen (str1) + strlen (str2) + 1);
 strcpy (name, str1);
 strcat (name, str2);
 return open (name, flags, mode);
}
```

А вот, как Вы получили бы те же самые результаты с malloc и free:

```
intopen2 (char * str1, char * str2, int flags, int mode)
{
 char *name = (char *) malloc (strlen (str1) + strlen (str2) + 1);
 int desc;
 if (name == 0) fatal ("превышенна виртуальная память ");
```

```
 strcpy (name, str1);
 strcat (name, str2);
 desc = (name, flags, mode);
 free (name);
 desc;
}
```

Вы видите, что это более просто с alloca. Но alloca имеет другие, более важные преимущества, и некоторые недостатки.

3.5.2 Преимущества alloca

Имеются причины, почему alloca может быть предпочтительнее malloc:

* Использование alloca занимает мало места и очень быстро. (Это открыто кодируется компилятором GNU C.)

* С тех пор alloca не имеет отдельных пулов для различных размеров блока, место, используемое для блока любого размера может многократно использоваться и для любого другого. alloca не вызывает фрагментацию памяти.

* Нелокальные выходы, выполненные через longjmp (см. Главу 20 [Нелокальные выходы]) автоматически освобождают место, размещенное alloca, когда они выходят из функции, которая вызвала alloca. Это - наиболее важная причина использовать alloca.

Чтобы иллюстрировать это, предположите, что Вы имеете функцию open_or_report_error, которая возвращает описатель открытого, если она успешно завершается, но не возвращается к вызывающему оператору, если она терпит неудачу. Если файл не может быть открыт, она печатает сообщение об ошибках и переходит с командного уровня вашей программы, используя longjmp. Давайте изменим open2 (см. раздел 3.5.1 [Примеры alloca]) чтобы использовать эту подпрограмму:

```
intopen2 (char * str1, char * str2, int flags, int mode)
{
 char *name = (char *) alloca (strlen (str1) + strlen (str2) + 1);
 strcpy (name, str1); strcat (name, str2);
 return open_or_report_error (name, flags, mode);
}
```

Из-за способа работы alloca, память, которую она резервирует, освобождается даже, когда происходит ошибка, без специального усилия.

А предыдущее определение open2 (которое использует malloc и free) допустило бы утечку памяти, если это было бы изменено таким образом,. Даже если Вы хотите сделать большее количество изменений, чтобы устранить это, не имеется никакого простого способа делать так.

3.5.3 Недостатки alloca

Здесь недостатки alloca по сравнению с malloc:

* Если, Вы пробуете зарезервировать большее количество памяти чем машина, может обеспечивать, Вы не получаете чистое сообщение об ошибках. Взамен Вы получаете фатальный сигнал подобно тому, который Вы получили бы из бесконечной рекурсии; возможно нарушение сегментации (см. раздел 21.2.1 [Сигналы об ошибках в программе]).

* Некоторые не-GNU системы будут не в состоянии поддерживать alloca, так что она менее переносима. Однако, более медленная эмуляция alloca, которую пишут на C доступна для использования в системах с этой неточностью.

3.5.4 GNU C массивы с переменным размером

В GNU C, Вы можете заменять большинство использований alloca с массивом переменного размера. Вот, как выглядела бы open2:

```
int open2(char * str1, char * str2, int flags, int mode)
{
 char name [strlen (str1) + strlen(str2) + 1];
 strcpy (name, str1);
 strcat (name, str2);
 return open (name, flags,mode);
}
```

Но alloca не всегда эквивалентна динамическому массиву, по следующим причинам:

* Место динамического массива освобождается в конце области действия имени массива. Место, размещенное alloca остается до конца этой функции.

* Возможно использовать alloca внутри цикла, резервируя дополнительный блок на каждой итерации. Это невозможно с динамическими массивами.

Обратите внимание: если Вы смешиваете использование alloca и динамических массивов внутри одной функции, выход из области, в который динамический массив был объявлен, освобождает все блоки, размещенные alloca во время выполнения этой области.

3.6 Настройка программы распределения

Любая система динамического распределения памяти имеет непроизводительные затраты: количество места, которое она использует - больше чем количество, о котором программа просит. Программа настройки распределения памяти достигает очень низких непроизводительных затрат, перемещая блоки в памяти по мере необходимости, по собственной инициативе.

3.6.1 Понятия настройки резервирования

Когда Вы резервируете блок malloc, адрес блока никогда не изменяется, если Вы не используете realloc, чтобы изменить размер. Таким образом, Вы можете безопасно сохранять адрес в различных местах, временно или постоянно, как захотите. Это не безопасно, когда Вы используете программу настройки распределения памяти, потому что любые переместимые блоки могут двигаться всякий раз, когда Вы зарезервируете память в любом режиме. Даже вызов malloc или realloc может перемещать переместимые блоки.

Для каждого переместимого блока, Вы должны делать дескриптор указываемого объекта в памяти, предназначенный для того, чтобы сохранять адрес этого блока. Программа настройки распределения знает, где находится дескриптор каждого блока, и модифицирует адрес, сохраненный там всякий раз, когда она перемещает блок, так, чтобы дескриптор всегда указывал на блок. Каждый раз Вы обращаетесь к содержимому блока, Вы должны брать адрес из дескриптора.

Вызов любой функции программы настройки распределения из обработчика сигнала почти всегда неправилен, потому что сигнал мог появиться в любое время. Единственый способ делать это безопасно состоит в том, чтобы блокировать сигнал для любого доступа к содержимому любого

переместимого блока, не удобен для работы. См. раздел 21.4.6 [Неповторное вхождение].

3.6.2 Распределение и освобождение переместимых блоков

В описаниях ниже, handleptr обозначает адрес дескриптора. Все функции объявлены в "malloc.h"; все являются расширениями GNU.

void * r_alloc (void ** handleptr, size _t size) (функция)

Эта функция резервирует переместимый блок размера size. Она

сохраняет адрес блока в * handleptr и возвращает непустой указатель в случае успеха.

Если r_alloc не может получить необходимое место, она сохраняет пустой указатель в *handleptr, и возвращает пустой указатель.

void r_alloc_free (void ** handleptr) (функция)

Эта функция - способ освободить переместимый блок. Она освобождает блок, на который указывает *handleptr, и сохраняет пустой указатель в *handleptr, чтобы показать что он больше не указывает на размещенный блок.

void * r_re_alloc (void ** handleptr, size _t size) (функция)

Функция r_re_alloc корректирует размер блока на который указывает *handleptr, делая его size байт длиной. Она сохраняет адрес измененного блока в *handleptr и возвращает непустой указатель в случае успеха.

Если достаточная память не доступна, эта функция, возвращает пустой указатель и не изменяет *handleptr.

3.7 Предупреждения относительно использования памяти

Вы можете просить о предупреждениях для программ исчерпывающих пространство памяти, вызывая memory_warnings. Она указывает, чтобы malloc проверял использование памяти, каждый раз когда он просит о большем количестве памяти из операционной системы. Это - расширение GNU, объявленное в "malloc.h".

void memory_warnings (void *start, void (* warn_func) (const char*)) (функция)

Вызывайте эту функцию, чтобы запросить предупреждения для приближающегося исчерпывания виртуальной памяти.

Аргумент start говорит, где начинается место данных в памяти. Программа распределения сравнивает его с последним используемым адресом и с пределом места данных, определяя долю доступной памяти. Если Вы указываете нуль как начало, то по умолчанию, используется наиболее вероятное значение.

malloc может вызывать функцию warn_func, чтобы предупредить Вас. Она вызывается со строкой (предупреждающим сообщением) как аргументом. Обычно она должнаот образить строку для пользователя.

Предупреждения приходят, когда память становится полной на 75%, на 85%, и на 95%. Если занято более чем 95 % Вы получаете другое предупреждение каждый раз увеличивая используемую память.

4. Обработчики символов

Программы которве работтают с символами и строками часто должны классифицировать символ как букву, цифру, пробел, и так далее и

выполнять операции замены регистра на символах. Функции в заголовочном файле "ctype.h" предусмотрены для этой цели.

Так как выбор стандарта и набора символов может изменять классификации специфических символьных кодов, все эти функции зависят от текущего стандарта. (Точнее, на них воздействует стандарт, в настоящее время выбранный для классификации символов LC_CTYPE; см. раздел 19.3 [Категории стандарта])

4.1 Классификация символов

Этот раздел объясняет библиотечные функции для классификации символов. Например, isalpha - функция, чтобы проверить буквенный ли символ. Она имеет один аргумент, символ для проверки, и возвращает целое число отличное от нуля, если символ буквенный, и нуль иначе. Ее можно использовть примерно так:

```
if (isalpha (c)) printf ("Символ \"%c\" является буквой.\n", c);
```

Каждая из функций в этом разделе проверяет на принадлежность специфическому классу символов; каждая имеет имя, начинающееся с 'is'. Каждая из них имеет один аргумент, который является проверяемым символом, и возвращает int, который обрабатывается как логическое значение. Символьный аргумент передан как int, и это может быть постоянное значение EOF вместо реального символа.

Атрибуты любого данного символа могут измениться между стандартами. См. Главу 19 [Стандарты], для уточнения информации относительно стандартов.

Эти функции объявлены в заголовочном файле "ctype.h".

int islower (int c) (функция)

Возвращает истину, если C - символ нижнего регистра.

int isupper (int c) (функция)

Возвращает истину, если C - символ верхнего регистра.

int isalpha (int c) (функция)

Возвращает истину, если C - буквенный символ (буква). Если islower или isupper - истина для символа, то isalpha - также истина.

В некоторых стандартах , могут иметься дополнительные символы, для которых isalpha является истинной, и которые не являются, ни строчными буквами, ни символами верхнего регистра. Но в стандарте "C", не имеется никаких таких дополнительных символов.

int isdigit (int c) (функция)

Возвращает истину, если C - десятичная цифра (от "0" до "9").

int isalnum (int c) (функция)

Возвращает истину, если C - алфавитно-цифровой символ (символ или цифра); другими словами, если или isalpha или isdigit - истина для символа, то isalnum - также истина.

int isxdigit (int c) (функция)

Возвращает истину, если C - шестнадцатеричная цифра. Шестнадцатеричные цифры включают нормальные десятичные цифры от "0" до "9" и символы от "A" до "F" и от "a" до "f".

int ispunct (int c) (функция)

Возвращает истину, если C - символ пунктуации. Это означает любой символ печати, который не алфавитно-цифровой или пробел.

int isspace (int c) (функция)

Возвращает истину, если C - символ пропуска. В стандартном расположении, isspace возвращает истину только для стандартных символов пробела:

" " пробел
"\f" перевод страницы
"\n" символ перевода строки
"\r" возврат каретки
"\t" горизонтальная метка табуляции
"\v" вертикальная метка табуляции
int isblank (int c) (функция)

Возвращает истину, если C - знак пропуска; то есть пробел или метка табуляции. Эта функция - расширение GNU.

int isgraph (int c) (функция)

Возвращает истину, если C - графический символ; то есть символ, который имеет glyph, связанный с этим. Символы пропуска не рассматриваются как графические символы.

int isprint (int c) (функция)

Возвращает истину, если C - символ печати. Символы печати включают все графические символы, плюс пробел (" ").

int iscntrl (int c) (функция)

Возвращает истину, если C - управляющий символ (то есть символ, который не является символом печати).

int isascii (int c) (функция)

Возвращает истину, если C 7-битовое символьное значение без знака, которое вписываться в US/UK ASCII набор символов. Эта функция - BSD расширение и - также SVID расширение.

4.2 Замена регистра

Этот раздел объясняет библиотечные функции для выполнения преобразований типа замены регистра символаов. Например, toupper преобразовывает любой символ в верхний регистр если возможно. Если символ не может быть преобразован, toupper, возвращает его неизменным.

Эти функции берут один аргумент типа int, который является символом, чтобы преобразовывать, и возвратить преобразованный символ как int. Если преобразование не применимо данному аргументу, аргумент, возвращается неизмененным.

Примечание относительно совместимости: В до-ANSI диалектах C, вместо того, чтобы возвращать неизмененный аргумент, эти функции могут терпеть неудачу, когда аргумент не подит для преобразования. Таким образом для переносимости, Вы должны написать islower (c)? toupper (c): c) а не просто toupper (c).

Эти функции объявлены в заголовочном файле "ctype.h".

int tolower (int c) (функция)

Если C - символ верхнего регистра, tolower возвращает соответствующий символ нижнего регистра. C если - не символ верхнего регистра, C возвращается неизменным.

int toupper (int c) (функция)

Если C - символ нижнего регистра, tolower возвращает соответствующий символ верхнего регистра. Иначе C возвращается

неизменным.

int toascii (int c) (функция)

Эта функция преобразовывает C в ASCII символ, очищая старшие биты. Эта функция - BSD расширение и - также SVID расширение.

int _tolower (int c) (функция)

Она идентична tolower, и предусматривает совместимость с SVID. См. раздел 1.2.4 [SVID].

int _toupper (int c) (функция)

Она идентична toupper, и предусматривает совместимость с SVID.

5. Утилиты для работы со строками и массивами.

Операции на строках (или массивах символов) - важная часть многих программ. Библиотека C GNU обеспечивает большой набор строковых сервисных функций, включая функции для копирования, связывания, сравнения, и поиска строк. Многие из этих функций могут также функционировать на произвольных областях памяти; например, функция memcpy может использоваться, чтобы копировать содержимое любого вида массива.

Для начинающихся C программистов довольно обычно "повторно изобретать колесо", дублируя эти функциональные возможности в их собственном коде, но знакомому с библиотечными функциями выгодно использовать их, так как это дает выгоды в эффективности и переносимости.

Например, Вы можете легко сравнивать одну строку с другим в двух строках кода C, но если Вы используете встроенную функцию strcmp, менее вероятно, что Вы сделаете ошибку. И, так как эти библиотечные функции обычно сильно оптимизированы, ваша программа может выполняться быстрее.

5.1 Представление строк

Этот раздел - быстрый обзор строковых понятий для начинающих программистов. Он описывает, как символьные строки представляются на C. Если Вы уже знакомы с этим материалом, Вы можете пропустить этот раздел.

Строка - массив объектов char. Но строковые переменные, обычно

объявляется, как указатели типа char *. Такие переменные не включают пространство для текста строки; он должен быть сохранен где-нибудь в переменной типа массив, строковой константе, или динамически размещенной памяти (см. Главу 3 [Распределение памяти]). Это позволяет Вам сохранить адрес выбранного пространства памяти в переменнуюуказатель. В качестве альтернативы Вы можете сохранять пустой указатель в переменной. Пустой указатель никуда не указывает, так что попытка сослаться на строку, на которую он указывает, получит ошибку.

Обычно, пустой символ, "\0", отмечает конец строки. Например, в тестировании, чтобы видеть, что переменная p указывает на пустой символ, отмечающий конец строки, Вы можете написать ! * P или * p == "\0".

Пустой символ - совершенно отличен от пустого указателя, хотя, и представляется целым числом 0.

Строковые литералы появляются в C программе как строки символов между символами кавычек ('"'). В ANSI C, строковые литералы могут также быть сформированы строковой конкатенацией: "a" "b" - то же что "ab". Изменение строковых литералов не допускается GNU C компилятором, потому что литералы помещены в памяти только для чтения.

Символьные массивы, которые являются объявленным const, также не могут изменяться. Это - вообще хороший стиль, объявить, что

немодифицируемые строковые указатели будут типа const char *, так как это часто позволяет компилятору C обнаружить случайные изменения, также как обеспечение некоторого количества документации относительно того, что ваша программа предполагает делать со строкой.

Объем памяти, размещенный для символьного массива может простираться после пустого символа, который обычно отмечает конец строки. В этом документе термин размер резервирования всегда используется, чтобы обратиться к общей сумме памяти, размещенной для строки, в то время как термин длина относится к числу символов до (но не, включая) пустого символа завершения.

Известным источником ошибок является то, что программа пробует

помещать большее количество символов в строку чем позволяет размещенный размер. При написании кода, который расширяет строки или перемещает символы в массив, Вы должны быть очень осторожны, чтобы следить за длиной текста и делать явную проверку переполнения массива. Многие из библиотечных функций не делают это для Вас! Не забудьте также, что Вы должны зарезервировать дополнительный байт, чтобы содержать пустой символ, который отмечает конец строки.

5.2 Соглашения относительно строк и массивов

Эта глава описывает функции, которые работают над произвольными массивами или блоками памяти, и функции, которые являются специфическими для массивов с нулевым символом в конце.

Функции, которые функционируют на произвольных блоках памяти, имеют имена, начинающиеся "mem" (типа memcpy) и неизменно имеют аргумент, который определяет размер (в байтах) блока рабочей памяти. Аргументы массива и возвращаемые значения для этих функций имеют тип void*, и как стиль, элементы этих массивов упоминаются как "байты". Вы можете передавать любой вид указателя на эти функции, а оператор sizeof полезен при вычислении значения аргумента size.

Напротив, функции, которые функционируют специально на строках, имеют имена, начинающиеся "str" (типа strcpy) и ищут пустой символ, чтобы завершить строку вместо того, чтобы требовать, чтобы был передан явный аргумент размера. (Некоторые из этих функций принимают заданную максимальную длину, но они также проверяют преждевременное окончание с пустым символом.) аргументы массива и возвращаемые значения для этих функций имеют тип char *, и элементы массива упоминаются как "символы".

В многих случаях, имеется, и "mem" и "str" версии функции. Которая является более подходящей, зависит от контекста. Когда ваша программа манипулирует произвольными массивами или блоками памяти, Вы должны всегда использовать "mem" функции. С другой стороны, когда Вы манипулируете строками с нулевым символом в конце, обычно более удобно использовать "str" функции, если Вы не знаете длину строки заранее.

5.3 Длина строки

Вы можете получить длину строки, используя функцию strlen. Эта функция объявлена в заголовочном файле "string.h".

size _t strlen (const char * s) (функция)

Функция strlen возвращает длину строки с нулевым символом в конце. (Другими словами, она возвращает смещение пустого символа завершения внутри массива.) Например,

```
strlen ("привет, мир")
=>12
```

Когда функция strlen применяется к символьному массиву, она возвращает длину сохраненной строки, а не размер резервирования. Вы можете получить размер резервирования символьного массива, который содержит строку, используя оператор sizeof:

```
char string[32] = "привет, мир";
sizeof (string)
=> 32
strlen (string)
=> 12
```

5.4 Копирование и конкатенация

Вы можете использовать функции, описанные в этом разделе, чтобы копировать содержимое строк и массивов, или конкатенировать содержимое одной строки с другой. Эти функции объявлены в заголовочном файле "string.h".

Все эти функции возвращают адрес целевого массива.

Большинство этих функций не работает правильно, если исходный и целевой массивы накладываются. Например, если начало массива адресата накладывается на конец исходного массива, первоначальное содержимое той части исходного массива может стать записанным поверх прежде, чем это скопировано. Даже хуже, в случае строковых функций, пустой символ,

отмечающий конец строки можно потерять, и функция копирования может эастревать в цикле, просматривая всю память зарезервированную для вашей программы.

Все функции, которые имеют проблемы при копировании между накладывающимися массивами, явно идентифицированы в этом руководстве. В дополнение к функциям в этом разделе, имеется несколько, других например sprintf (см. раздел 7.9.7 [Форматируемые функции вывода]) и scanf (см. раздел 7.11.8 [Форматируемые функции ввода]).

void * memcpy (void *to, void const *from, size _t size) (функция)

Функция memcpy копирует size байт из объекта, начинающегося в from в объект, начинающийся в to. Поведение этой функции неопределено если два массива перекрываются; используйте memmove взамен, если возможно перекрывание.

Значение, возвращенное memcpy - значение to.

Вот пример того, как Вы могли бы использовать memcpy, чтобы копировать содержимое массива:

```
struct foo *oldarray, *newarray;
int arraysize;
...
memcpy (new, old, arraysize * sizeof (struct foo));
```

void * memmove (void *to, void const *from, size _t size) (функция)

memmove копирует size байт из from в size в to, даже если те два блока памяти перекрываются.

void * memccpy (void *to, void const *from, int C, size _t size) (функция)

Эта функция копирует не больше, чем size байт из from в to, останавливая если найден байт соответствующий C. Возвращаемое значение - указатель в to на первый байт после C, или пустой указатель, если

никакой байт, соответствующий C не появился в первых size байтах from.

void * memset (void * block, int C, size _t size) (функция)

Эта функция копирует значение C (преобразованный в char без знака) в каждый из первых size байтов объекта, начинающегося с block. Она возвращает значение block.

char * strcpy (char *to, const char *from) (функция)

Она копирует символы из строки from (включая пустой символ завершения) в строку to. Подобно memcpy, эта функция имеет неопределенные результаты, если строки накладываются. Возвращаемое значение - значение to.

char * strncpy (char *to, const char *from, size _t size) (функция)

Эта функция подобна strcpy, но всегда копирует точно size символов в to.

Если длина from - больше чем size, то strncpy копирует только первые size символов.

Если длина from меньше чем size, то strncpy, копирует все from, сопровождая его достаточным количеством пустых символов, чтобы получить всего size символов. Это редко полезно, но зато определено в соответствии с ANSI стандартом.

Поведение strncpy неопределено, если строки накладываются.

Использование strncpy в противоположность strcpy - способ избежать ошибок в отношении соглашения о записи после конца размещенного пространства для to. Однако, это может также сделать вашу программу намного медленнее в одном общем случае: копирование строки, которая является возможно малой в потенциально большой буфер. В этом случае, size может быть большой, и когда это, strncpy будет тратить впустую значительное количество времени, копируя пустые символы.

char * strdup (const char * s) (функция)

Эта функция копирует строку с нулевым символом в конце в недавно размещенную строку. Строка размещена, используя malloc; см. раздел 3.3 [Беспрепятственное резервирование].

Если malloc не может зарезервировать пространство для новой строки, strdup возвращает пустой указатель. Иначе она возвращает указатель на новую строку.

char * stpcpy (char *to, const char *from) (функция)

Эта функция - подобно strcpy, за исключением того, что она возвращает указатель на конец строки to (то есть адрес пустого символа завершения) а не на начало.

Например, эта программа использует stpcpy, чтобы конкатенировать "foo" и "bar" и печатает "foobar".

```
#include
#include
intmain (void)
{
 char buffer[10];
 char *to = buffer;
 to = stpcpy (to, "foo");
 to = stpcpy (to, "bar");
 puts (buffer);
 return 0;
}
```

Эта функция - не часть ANSI или POSIX стандартов, и не обычна в системах UNIX, но мы не изобретали ее. Возможно она исходит из MS-DOS.

Поведение неопределено, если строки накладываются.

char * strcat (char * to, const char *from) (функция)

Функция strcat подобна strcpy, за исключением того, что символы из from добавляются или конкатенируются к концу to, вместо того, чтобы

записывать поверх него. То есть первый символ из from накладывается на пустой символ, отмечающий конец to.

Эквивалентное определение для strcat было бы:

```
char * strcat (char *to, const char *from)
{
 strcpy (to + strlen to, from);
 return to;
}
```

Эта функция имеет неопределенные результаты, если строки накладываются.

char * strncat (char *to, const char *from, size _t size) (функция)

Эта функция - подобна strcat за исключением того, что не больше чем size символов из from конкатенируются к концу to. Одиночный пустой символ также всегда конкатенируется к to, так что общая конечая длина to должна быть по крайней мере на size + 1 байт больше чем начальная длина.

Функция strncat могла быть выполнена примерно так:

```
char * strncat (char *to, const char *from, size_t size)
{
 strncpy (to + strlen (to), from, size);
 return to;
}
```

Поведение strncat неопределено, если строки накладываются.

Вот пример, показывающий использование strncpy и strncat. Обратите внимание, как вычислен параметр size, в обращении к strncat, чтобы избежать переполнять символьный массив buffer.


```
#include
#include
#define SIZE 10
static char buffer[SIZE];
main ()
{
 strncpy (buffer, "hello", SIZE);
 puts (buffer);
 strncat (buffer, ", world", SIZE - strlen (buffer) - 1);
 puts (buffer);
}
```

Вывод, произведенный этой программой будет:

```
hellohello, wo
```

void * bcopy (void *from, void const *to, size _t size) (функция)

Она является частично устаревшим вариантом для memmove, и происходит от BSD. Обратите внимание, что она не совершенно эквивалентна memmove, потому что аргументы не в том же самом порядке.

void * bzero (void *block, size _t size) (функция)

Она является частично устаренным аналогом memset, и происходит от BSD. Обратите внимание, что она не такая общая как memset, потому что единственое значение, которое она может сохранять - нуль. Некоторые машины имеют специальные команды для установки на нуль памяти, так что bzero может быть более эффективна чем memset.

5.5 Сравнение строк/массивов

Вы можете использовать функции в этого раздела, чтобы выполнить сравнение на содержимое строк и массивов. Также как для проверки равенства, эти функции могут использоваться как функции упорядочения для операций сортировки. См. Главу 15 [Поиск и сортировка].

В отличие от большинства операций сравнения в C, строковые функции

сравнения возвращают значение отличное от нуля, если строки - не эквивалентны. Знак значения указывает относительное упорядочение первых символов в строках, которые - не эквивалентны: отрицательное значение указывает, что первая строка - "меньше" чем вторая, в то время как положительное значение указывает, что первая строка "больше".

Если Вы используете эти функции только, чтобы проверить равенство, для большей чистоты программы, лучше задать их как макроопределения, примерно так:

```
#define str_eq (s1, s2) (! Strcmp ((s1), (s2)))
```

Все эти функции объявлены в заголовочном файле "string.h".

int memcmp (void const * a1, void const * a2, size _t size) (функция)

Функция memcmp сравнивает size байт памяти, начинающиеся в a1 с size байтами памяти, начинающимися в a2. Возвращенное значение имеет тот же самый знак как разность между первой отличающейся парой байтов (интерпретируемых как объекты char без знака).

Если содержимое двух блоков равно, memcmp, возвращает 0.

На произвольных массивах, функция memcmp обычно полезна для тестирования равенства. Обычно не имеет смысла делать байтовое сравнение упорядочения на не байтовых массивах. Например, байтовое сравнение на байтах, которые составляют числа с плавающей запятой не должно правдоподобно сообщить Вам что - нибудь относительно связи между значениями чисел с плавающей запятой.

Вы должны также быть внимательны при использовании memcmp, для сравнения объектов, которые могут содержать "дырки", типа дополнения, вставленного в объекты структуры, чтобы предписать требования выравнивания, дополнительного пространства в конце объединений, и дополнительных символов в концах строк, чьи длины меньше чем затребовано при размещении.

Содержимое этих "дырок" неопределено и может вызывать странное поведение при выполнении байтового сравнения. Для более предсказуемых результатов, выполните явное покомпонентное сравнение.

Например, пусть дано определение типов структуры подобно:

```
struct foo {
 unsigned char tag;
 union{double f; long i; char *p;}
 value;
};
```

Вам лучше написать специализированные функции сравнения, чтобы сравнить объекты struct foo вместо того, чтобы сравнить их memcmp.

int strcmp (const char * s1, const char * s2) (функция)

Функция strcmp сравнивает строку s1 с s2, возвращая значение, которое имеет тот же самый знак как различие между первой отличающейся парой символов (интерпретируемых как объекты char без знака).

Если две строки равны, strcmp, возвращает 0.

Следствие упорядочения, используемого strcmp - то, что, если s1 является начальной подстрокой s2, то s1 "меньше чем" s2.

int strcasecmp (const char * s1, const char * s2) (функция)

Эта функция подобна strcmp, за исключением того, что разногласия игнорируются.

strcasecmp происходит от BSD.

int strncasecmp (const char * s1, const char * s2) (функция)

Эта функция - подобна strncmp, за исключением того, что разногласия игнорируются.

strncasecmp - расширение GNU.

int strncmp (const char * s1, const char * s2, size _t size) (функция)

Эта функция подобна strcmp, за исключением того, что сравниваются не больше, чем size символов. Другими словами, если две строки совпадают в их первых size символах, возвращаемое значение - нуль.

Имеются некоторые примеры, показывающие использование strcmp и strncmp. Эти примеры подразумевают использование набора символов ASCII. (Если используется некоторый другой набор символов скажем, расширенный двоично-десятичный код обмена информацией, взамен, то glyphs связаны с различными числовыми кодами, и возвращаемые значения, и порядок могут отличиться.)

```
strcmp ("привет", "привет")
=> 0 /* Эти две строки одинаковы. */
strcmp ("привет", "Привет")
=> 33 /* Сравнение чувствительно к регистру. */
strcmp ("привет", "мир")
=> -2 /* символ "п" меньше чем "м". */
strcmp ("привет", "привет, мир")
=> -44 /* Сравнение пустого символа с запятой. */
strncmp ("привет", "привет, мир", 5)
=> 0 /* начальные 5 символов - те же самые. */
strncmp ("привет, мир","привет, глупый мир!!!", 5)
=> 0 /* начальные 5 символов - те же самые. */
```

int bcmp (void const * a1, void const * a2, size _t size) (функция)

Это - устаревший побочный результат исследования для memcmp, происходит от BSD.

5.6 Функции для объединений

В некоторых местах соглашения лексикографического упорядочения отличаются от строгого числового упорядочения символьных кодов. Например, в Испании большинство букв с диакритическими знаками, но буквы с диакритическими знаками не считаются различными символами в целях объединения. С другой стороны, последовательность с двумя символами "ll" обрабатывается как одна буква, которая объединена с "l".

Вы можете использовать функции strcoll и strxfrm (объявленные в заголовочном файле "string.h") чтобы сравнивать строки, использующие объединение упорядочивания соответствующее для данной местности. Стандарт, используемое этими функциями в частности может быть определено, устанавливая стандарт для класса LC_COLLATE; см. Главу 19 [Стандарты].

В стандартном расположении C, последовательность объединений для strcoll - таже что для strcmp.

Действительно, способ, которым эта функция работает применяя отображение, чтобы трансформировать символы в строку, в последовательность байтов которая представляет позицию строки в последовательности объединений текущего расположения. Сравнение двух таких последовательностей байтов в простом режиме эквивалентно сравнению

строк с последовательностью объединений расположения.

Функция strcoll выполняет эту трансляцию неявно, чтобы делать одно сравнение. А, strxfrm выполняет отображение явно. Если Вы делаете многократное сравнение, используя ту же самую строку или набор строк, то более эффективноиспользовать strxfrm, чтобы трансформировать все строки только один раз, и впоследствии сравнивать преобразованные строки strcmp.

int strcoll (const char * s1, const char * s2) (функция)

Функция strcoll подобна strcmp, но использует последовательность объединений данного расположения для объединения (LC_COLLATE стандарт).

Вот пример сортировки массива строк, с использованием strcoll,

чтобы сравнить их. Фактический алгоритм сортировки здесь не написан; он исходит из qsort (см. раздел 15.3 [Функции сортировки массива]). Работа кода показанного здесь, говорит, как сравнивать строки при их сортировке. (Позже в этом разделе, мы покажем способ делать это, более эффективно используя strxfrm.)

```
/* Это - функция сравнения, используемая в qsort. */
intcompare_elements (char **p1, char **p2)
{
 return strcoll (*p1,*p2);
}

/* Это - точка входа к функции сортировки строк, использующей
последовательность объединений расположения. */
void sort_strings (char **array, int nstrings)
{ /* Сортировка временного массива, сравнивая строки. */
 qsort (array, sizeof (char *), nstrings, compare_elements);
}
```

size _t strxfrm (char *to, const char *from, size _t size) (функция)

Функция strxfrm трансформирует строку, используя преобразование объединения, определенное стандартом, в настоящее время выбранным для объединения, и сохраняет преобразованную строку в массиве to.

Поведение неопределено если строки to и from перекрываются; см. раздел 5.4 [Копирование и конкатенация].

Возвращаемое значение - длина всей преобразованной строки. На это значение не воздействует значение size, но если она большее чем size, это означает, что преобразованная строка полностью не поместилась в массиве to.

Чтобы получать целую преобразованную строку, вызовите strxfrm снова с большим массивом вывода.

Преобразованная строка может быть больше чем первоначальная строка,

а может также быть более короткой.

Если size - нуль, никакие символы не сохранены в to. В этом случае, strxfrm просто возвращает число символов, которое было бы длиной преобразованной строки. Это полезно для определения какую строку зарезервировать. Не имеет значение, что будет в to, если size - нуль; может быть даже пустой указатель.

Вот пример того, как Вы можете использовать strxfrm когда Вы планируете делать много сравнений. Он делает то же самое что и предыдущий пример, но намного быстрее, потому что он должен трансформировать каждую строку только один раз, независимо от того сколько раз она сравнивается с другими строками. Даже времени для резервирования и освобождения памяти нужно намного меньше чем, когда имеются много строк.

```
     struct sorter { char *input; char *transformed; };

     /* Это - функция сравнения, используемая qsort для сортировки
massива структур. */
     int compare_elements (struct sorter *p1, struct sorter *p2)
     {
      return strcmp (p1->transformed, p2->transformed);
     }

     /* Это - точка входа функции сортировки строк, использующей
последовательность объединений расположения. */
     void sort_strings_fast (char **array, int nstrings)
     {
      struct sorter temp_array[nstrings];
      int i;
     /* Устанавливает temp_array. Каждый элемент содержит одну входную
строку и преобразованную строку. */
       for (i = 0; i < nstrings; i++)
       {
        size_t length = strlen (array[i]) * 2;
        temp_array[i].input = array[i];
     /* Трансформирует array[i] . Сначала, пробует буфер, возможно
```



```
достаточно большой. */
        while (1)
        {
         char *transformed = (char *) xmalloc (length);
         if (strxfrm (transformed, array[i], length) < length)
         {
          temp_array[i].transformed = transformed;
          break;
         } /* Попытка снова с еще большим буфером. */
         free (transformed); length *= 2;
        }
       }
     }

     /* Сортировка temp_array, сравнивая преобразованные строки. */
     qsort (temp_array, sizeof (struct sorter), nstrings,
compare_elements);

     /* Помещает элементы обратно в постоянный массив в их сортированном
порядке. */
     for (i = 0; i < nstrings; i++) array[i] = temp_array[i].input;

     /* Освобождают строки, которые мы зарезервировали. */
     for (i = 0; i < nstrings; i++) free (temp_array[i].transformed);}
```

Примечание относительно совместимости: функции объединения строк - новая возможность ANSI C., более старые диалекты C не имеют никакой эквивалентной возможности.

5.7 Функции поиска

Этот раздел описывает библиотечные функции, которые выполняют различные виды операций поиска в строках и массивах. Эти функции объявлены в заголовочном файле "string.h".

`void * memchr (void const *block, int c, size _t size)` (функция)

Эта функция находит первое вхождение байта c (преобразованного в char без знака) в начальных size байтах объекта, начинающегося в block. Возвращаемое значение - указатель на размещенный байт, или пустой указатель, если не было найдено никакого соответствия.

`char * strchr (const char *string, int c)` (функция)

Функция strchr находит первое вхождение символа c (преобразованного

в char) в строке с нулевым символом в конце, начинающейся в string. Возвращаемое значение - указатель на размещенный символ, или пустой указатель, если никакое соответствие не было найдено.

Например,
strchr ("привет, мир", "в")
=> "вет, мир"
strchr ("привет, мир", "?")
=> NULL

Пустой символ завершения является частью строки, так что Вы можете использовать эту функцию, для получения указателя на конец строки, определяя пустой символ как значение аргумента c.

char * index (const char *string, int c) (функция) - другое имя для strchr.

char * strrchr (const char *string, int c) (функция)

Функция strrchr - подобна strchr, за исключением того, что она ищет в обратном направлении, с конца строки (а не сначала).

Например,
strrchr ("привет, мир", "и")
=> "ир"

char * rindex (const char *string, int c) (функция)

rindex - другое имя для strrchr.

char * strstr (const char * haystack, const char * needle) (функция)

Она подобна strchr, за исключением того, что она ищет в haystack подстроку needle, а не только одиночный символ. Она возвращает указатель в строку haystack, который является первым символом подстроки, или пустой указатель, если никакого соответствия не было найдено. Если needle - пустая строка, то функция возвращает haystack.

Например,
strstr ("привет, мир","в")
=> "вет, мир"
trstr ("привет, мир","ми")
=> "мир"

void * memmem (const void *needle, size_t needle_len, const void *haystack, size_t haystack_len) (функция)

Она подобна strstr, но needle и haystack байтовые массивы, а не строки с нулевым символом в конце. needle_len - длина needle, а haystack_len - длина haystack.

Эта функция - расширение GNU.

size_t strspn (const char *string, const char *skipset) (функция)

Функция strspn ("строковый диапазон") возвращает длину начальной подстроки строки, которая состоит полностью из символов, которые являются элементами набора, заданного строкой skipset. Порядок символов в skipset не важен.

Например,
strspn ("привет, мир","абвгдежзийклмнопрстуфхцчшщъыьэюя"))
=> 6

size_t strcspn (const char *string, const char *stopset) (функция)

Функция strcspn ("строковый диапазон дополнения") возвращает длину начальной подстроки строки, которая состоит полностью из символов, которые - не элементы набора, заданного строкой stopset. (Другими словами, она возвращает смещение первого символа в строке, которая является элементом набора stopset.)

Например,
strcspn ("привет, мир","\t\n,.;!?")
=> 6

char * strpbrk (const char *string, const char *stopset) (функция)

strpbrk ("строковое прерывание указателя") то же что strcspn, за исключением того, что она возвращает указатель на первый символ в строке, который является элементом набора stopset вместо длины начальной подстроки. Это возвращает пустой указатель, если никакой символ из stopset не найден.

Например,
strpbrk ("привет, мир","\t\n,.;!?")
=> ",мир"

5.8 Поиск лексем в строке

В программах довольно часто возникает потребность сделать некоторые простые виды лексического и синтаксического анализа, типа разбивания командной строки в лексемы. Вы можете делать это функцией strtok, объявленной в заголовочном файле "string.h".

char * strtok (char *newstring, const char *delimiters) (функция)

Строку можно разбить в лексемы, делая ряд обращений к функции strtok.

Строка, которую нужно разбивать передается как newstring только при

первом обращении. Функция strtok использует ее, чтобы установить некоторую внутреннюю информацию о состоянии. Последующие обращения, чтобы получить дополнительные лексемы из той же самой строки обозначаются передачей пустого указателя как аргумента. Вызов strtok с другим не-пустым символом newstring повторно инициализирует информацию о состоянии. Гарантируется, что никакая другая библиотечная функция (которая смешала бы эту внутреннюю информацию о состоянии) никогда не вызовет strtok.

Аргумент delimiters - строка, которая определяет набор разделителей, которые могут окружать извлекаемую лексему. Все начальные символы, которые являются элементами этого набора, отбрасываются. Первый символ, который не является элементом этого набора разделителей, отмечает начало следующей лексемы. Конец лексемы находится, как следующий символ, который является элементом набора разделителей. Этот символ в первоначальной строке newstring зменяется пустым символом, и возвращается указатель на начало лексемы в newstring.

При следующем обращении к strtok, поиск начинается со следующего символа после того который отмечен концом предыдущей лексемы. Обратите внимание, что набор разделителей не должен быть тем же самым при каждом вызове strtok.

Если конец строки newstring достигнут, или если остаточный член от строки состоит только из символов - разделителей, strtok, возвращает пустой указатель.

Предупреждение: С тех пор как strtok изменит строку, которую она анализирует, всегда копируйте строку во временный буфер перед синтаксическим ее анализом strtok. Если Вы разрешаете, чтобы strtok изменил строку, которая исходила из другой части вашей программы, Вы создаете проблему; та строка может быть частью структуры данных, которая могла использоваться для других целей в течение синтаксического анализа, в то время как чередование strtok делает структуру данных, временно неточной.

Строка, на которой Вы действуете, могла быть и константой. И, когда

strtok попробует изменять ее, ваша программа получит фатальный сигнал о записи в память распределенную тоько для чтения. См. раздел 21.2.1 [Сигналы ошибки в программе].

Это - частный случай общего принципа: если часть программы не имеет цели изменить некоторую структуру данных, то ошибочно изменять структуру данных даже временно.

Функция strtok не повторно используема. См. раздел 21.4.6 [Неповторная входимость], для обсуждения где и почему повторная входимость важна.

Вот простой пример, показывающий использование strtok.

```
#include
#include
. . .
char string[] = "words separated by spaces -- and, punctuation!";
const char delimiters[] = " .,;:!-";
char *token;
. . .
token = strtok (string, delimiters);  /* token => "words" */
token = strtok (NULL, delimiters);   /* token => "separated" */
token = strtok (NULL, delimiters);   /* token => "by" */
token = strtok (NULL, delimiters);   /* token => "spaces" */
token = strtok (NULL, delimiters);   /* token => "and" */
token = strtok (NULL, delimiters);   /* token => "punctuation" */
token = strtok (NULL, delimiters);   /* token => NULL */
```


6. Краткий обзор ввода-вывода

Большинство программ должно делать некоторый ввод (чтение данных) или вывод, или наиболее часто оба, чтобы сделать что-нибудь полезное. Библиотека GNU C обеспечивает такой большой выбор функций ввода и функций вывода, что самая трудная часть решить, которая функция является наиболее подходящей!

Эта глава представляет понятия и терминологию, имеющие отношение к вводу и выводу. Другие главы, имеющие отношение к средствам ввода-вывода GNU:

* Глава 7 [Ввод-вывод на потоках], которая описывает функции выоского уровня, функционирующие на потоках, включая форматируемый ввод и выводит.

* Глава 8 [Ввод-вывод низкого уровня], которая описывает базисный ввод-вывод и функции управления на описателях файла.

* Глава 9 [Интерфейс файловой системы], которая описывает функции для операций на каталогах и для управления атрибутами файла, режимами доступа и монопольного использования.

* Глава 10 [Трубопроводы и FIFO (первый зашел - первый вышел)], которая включает информацию относительно базисных межпроцессорных средств связи.

* Глава 11 [Гнезда], которая описывает более сложное межпроцессорное средство связи с поддержкой для работы с сетями.

* Глава 12 [Интерфейс терминала низкого уровня], которая описывает функции для изменения как ввода так и вывода на терминал или другое последовательное устройство.

- 100 -

6.1 Понятия ввода-вывода

Прежде, чем Вы сможете читать или писать содержимое файла, Вы должны установить соединение или канал связи с файлом. Этот процесс называется открытием файла. Вы можете открывать файл для чтения, записи, или для того и другого.

Соединение с открытым файлом представляется или как поток или как описатель файла. Вы передаете его как аргумент функциям, которые делают фактическое чтение или операции записи. Некоторые функции ожидают потоки, а некоторые разработаны для взаимодействия с описателями файла.

Когда Вы закончили читать из или писать в файл, Вы можете завершать соединение, закрывая файл. Если только Вы закрыли поток или описатель файла, Вы не можете больше делать операции ввода-вывода на нем.

6.1.1 Потоки и описатели файла

Когда Вы хотите осуществлять ввод или выводить в файл, Вы имеете выбор из двух базисных механизмов для представления соединения между вашей программой и файлом. Это описатели файла и потоки. Описатели файла представляются как объекты типа int, в то время как потоки представляются как объекты FILE* (указатели).

Описатели файла обеспечивают примитивный интерфейс низкого уровня для операций ввода и вывода. И описатели файла, и потоки могут представлять соединение с устройством (типа терминала), или с трубопрводом или с гнездом для сообщения с другим процессом, также как с нормальным файлом. Но, если Вы хотите осуществлять операции управления, которые являются специфическими для специфического вида устройства, Вы должны использовать описатель файла; не имеется никаких средств, чтобы использовать для этого потоки. Вы должны также использовать описатели файла, если ваша программа должна делать ввод или выводить в специальных режимах, типа неблокированный (или опрошенный) ввод (см. раздел 8.10 [Флаги состояния файла]).

Потоки обеспечивают интерфейс более высокого уровня, основанный на примитивных средствах описателя файла. Интерфейс потока хорошо обрабатывает все виды файлов, единственная сложность - три стиля

- 101 -

буферизации, которую Вы можете выбирать (см. раздел 7.17 [Буферизация потока]).

Основное преимущество использования интерфейса потока - то, что набор функций для выполнения фактического ввода и вывода (в противоположность операциям управления) на потоках является намного более богатым и более мощным чем соответствующие средства для описателей файла. Интерфейс описателя файла обеспечивает только простые функции для пересылки блоков символов, а интерфейс потока обеспечивает мощный форматируемый ввод и вывода (printf и scanf) также как функции для символьно- и строко- ориентированного ввода и вывода.

Так как потоки выполнены в терминах описателей файла, Вы можете извлекать описатель файла из потока и выполнять операции низкого уровня непосредственно на описателе файла. Вы можете также первоначально открывать соединение как описатель файла, а потом делать поток, связанный с этим описателем файла.

Вообще, Вы должны ограничиться использованием потоков, если не имеется некоторой специфической операции, которая может быть выполнена только на описателе файла. Если Вы - начинающий программист и не уверены, какие функции использовать, мы предлагаем Вам концентрироваться

на форматируемых функциях (см. раздел 7.11 [Форматируемый ввод] и раздел 7.9 [Форматируемый вывод]).

Если Вы думаете о переносимости ваших программ на не-GNU системы, Вы должны также осознавать, что описатели файла не так переносимы как потоки. Вы можете ожидать что любая ANSI система поддерживает потоки, но не-GNU системы не могут поддерживать описатели файла вообще или могут выполнять только подмножество функций GNU, которые функционируют на описателях файла. Большинство функций описателя файла в библиотеке GNU включено в стандарт POSIX.1.

6.1.2 Позиция файла

Один из атрибутов открытого файла - позиция файла, которая следит, где в файле следующий символ должен прочитаться или написаться. В системе GNU, и всех POSIX.1 системах, позиция файла - просто целое число, представляющее число байтов от начала файла.

Позиция файла обычно устанавливается в начало файла, когда он открыт, и каждый раз когда символ читается или пишется, позиция файла увеличивается. Другими словами, доступ к файлу обычно последователен.

Обычные файлы разрешают чтение или запись в любую позицию внутри файла. Некоторые другие виды файлов могут также разрешать это. Файлы, которые разрешают это иногда упоминаются как файлы прямого доступа. Вы можете изменять позицию файла, используя функцию fseek на потоке (см. раздел 7.15 [Позиционирование файла]) или функцию lseek на описателе файла (см. раздел 8.2 [Примитивы ввода - вывода]). Если Вы пробуете изменять позицию файла в файле, который не поддерживает произвольный доступ, Вы получите ошибку ESPIPE.

Потоки и описатели, которые открыты для дописывания обрабатываются особенно для вывода: вывод в такие файлы всегда конкатенируется последовательно к концу файла, независимо от позиции файла. Но, позиция файла все еще используется, чтобы управлять, где в файле производить выполняемое чтение.

Если Вы подумаете относительно этого, вы поймете, что несколько программ могут читать данный файл в то же самое время. Каждая программа должна иметь собственный указатель файла, на который не воздействует ничего из того, что делают другие программы.

Фактически, каждое открытие файла создает отдельную позицию файла. Таким образом, если Вы открываете файл дважды даже в той же самой программе, Вы получаете два потока или описатели с независимыми позициями файла.

Напротив, если Вы открываете описатель а потом дублируете его, чтобы получить другой описатель, эти два описателя совместно используют

ту же самую позицию файла: изменение позиции файла одного описателя будет воздействовать на другой.

6.2 Имена файла

Чтобы открывать соединение с файлом, или выполнять другие операции типа удаления файла, Вы нуждаетесь в некотором способе обращения к файлу. Почти все файлы имеют имена, которые представляются строками.

Эти строки называются именами файла. Вы определяете имя файла, чтобы указывать, который файл Вы хотите открыть или обработать.

Этот раздел описывает соглашения для имен файла и как операционная система работает с ними.

6.2.1 Каталоги

Чтобы понимать синтаксис имен файла, Вы должны понять, как файловая система организована посредством иерархии каталогов.

Каталог - это файл, который содержит информацию, связыывающую имена других файлов; эти ассоциации называются выходами каталога или узами. Иногда, люди говорят "файлы в каталоге", но фактически, каталог только содержит указатели на файлы, а не файлы непосредственно.

Имя каталога содержащего файл, называется компонентом имени файла. Вообще, имя файла это последовательность из одного или большего количества таких компонентов, отделяемых символом наклонной черты вправо ("/").

Некоторые другие документы, типа POSIX стандарта, используют термин путь для того, что мы называем имя файла, и компонент пути для того, что это руководство вызывает компонент имени файла. Мы не используем эту терминологию, потому что "путь" - это что-нибудь полностью отличное (список каталогов для поиска), и мы думаем, что "имя пути", используемое для чего-нибудь еще будет запутывать пользователей. В документации GNU мы всегда используем "имя файла" и "компонент имени файла" (или иногда только "компонент", где контекст очевиден).

Вы можете найти более детализированную информацию относительно операций с каталогами в Главе 9 [Интерфейс файловой системы].

6.2.2 Назначение имени файла

Имя файла состоит из компонентов имени файла, отделяемых наклонной чертой вправо ("/"). В системах, которые поддерживает библиотека GNU C, многократные последовательные символы `/' эквивалентны одиночному символу `/'.

Процесс определения, к какому файлу относится имя файла называется назначение имени файла. Это реализуется, исследованием компонентов, которые составляют имя файла слева направо, и размещением каждого последовательныого компонента в каталоге, именованном предыдущим компонентом. Конечно, каждый из файлов, которые названы каталогами, должен фактически существовать, и быть каталогом вместо регулярного файла, и иметь соответствующие права, чтобы быть доступным различным процессам; иначе неправильно назначено имя файла.

Если, имя файла начинается с "/", первым компонентом в имени файла принимается корневой каталог процесса (обычно все процессы в системе имеют тот же самый корневой каталог). Такое имя файла называется абсолютным именем файла.

Иначе, первый компонент в имени файла есть текущий рабочий каталог (см. раздел 9.1 [Рабочий каталог]). Этот вид имени файла называется именем файла прямого доступа.

Компоненты имени файла "." ("точка") и ".." ("точка - точка") имеют специальные значения. Каждый каталог имеет входы для этих компонентов имени файла. Компонент имени файла "." относится к каталогу непосредственно, в то время как компонент имени файла ".." относится к каталогу предыдущего уровня (каталог, который содержит связь для рассматриваемого каталога). Как частный случай, ".." в корневом каталоге относится к корневому каталогу непосредственно, так как он не имеет никакого родителя; таким образом "/.." тоже что и "/".

Вот несколько примеров имен файла:

"/a" файл, с именем "a", в корневом каталоге.

"/a/b" файл, с именем "b", в каталоге с именем "a" в корневом каталоге.

"a" Файл, с именем "a" в текущем рабочем каталоге.

"/a/./b" Это то же, что и "/a/b".

"./a" файл с именем "a", в текущем рабочем каталоге.

"../a" файл с именем "a", в директории предыдущего уровня текущего рабочего каталога.

Имя файла, которое есть имя каталога может необязательно заканчиваться на "/". Вы можете определять имя файла "/" чтобы обратиться к корневому каталогу, но пустая строка - не имя файла. Если Вы хотите обратиться к текущему рабочему каталогу, используйте имя файла "." или "./".

В отличие от некоторых других операционных систем, система GNU не имеет ни какой встроенной поддержки типов файлов (или расширений) или версий как частей синтаксиса имени файла. Много программ и утилит используют соглашения для имен файлов, например, файлы, содержащие исходный текст C обычно имеют имена, с прибавленным ".c" но непосредственно в файловой системе не имеется ничего, что предписывает этот вид соглашения.

6.2.3 Ошибки, связанные с именами файлов

Функции, которые принимают как аргументы имена файлов обычно, обнаруживают эти errno - условия ошибки в отношении синтаксиса имени файла. Эти ошибки упоминаются в этом руководство как обычные синтаксические ошибки имени файла.

EACCES

Процесс не имеет права поиска для каталога - компонента имени файла.

ENAMETOOLONG

Эта ошибка используется, когда или общая длина имени файла большае чем PATH_MAX, или когда индивидуальный компонент имени файла имеет длину больше чем NAME_MAX. См. раздел 27.6 [Ограничения для файлов].

В системе GNU, не имеется никакого наложенного ограничения полной длины имени файла, но некоторые файловые системы могут иметь ограничения длины компонента.

ENOENT

Об этой ошибке сообщается, когда файл, вызванный как каталог - компонент в имени файла не существует, или когда компонент есть символическая связь, чей выходной файл не существует. См. раздел 9.4 [Символические связи].

ENOTDIR

Файл, который вызван как, каталог - компонент имени файла существует, но это не каталог.

ELOOP

Слишком много символических связей было рассмотрено при попытке поиска имени файла. Система имеет произвольное ограничение числа символических связей, которые могут быть запомнены при поиске одиночного имени файла, как примитивный способ обнаружить циклы. См. раздел 9.4 [Символические связи].

6.2.4 Переносимость имен файла

Правила для синтаксиса имен файла обсуждаемые в разделе 6.2 [Имена файла], являются правилами, обычно используемыми системой GNU и другими POSIX системами. Однако, другие операционные системы могут использовать другие соглашения.

Имеются две причины, почему для Вас важно осознавать проблемы переносимости имени файла:

* Если ваша программа делает предположения относительно синтаксиса имени файла, или содержит внедренные литеральные строки имени файла, то более трудно выполнить ее под управлением других операционных систем, которые используют различные соглашения синтаксиса.

* Даже если Вы не думаете о выполнении вашей программы на машинах, которые поддерживают другие операционные системы, Вы можете обращаться к файлам, которые используют различные соглашения наименования. Например, Вможете обратиться к файловым системам на другом компьютере, поддерживающем другую операционную систему, или читать и писать на диски в форматах, используемых другими операционными системами.

ANSI стандарт имеет очень немного ограничений относительно синтаксиса имени файла, только что имена файла являются строками. В дополнение к изменяющимся ограничениям на длину имен файла и какие символы могут законно появляться в имени файла, различные операционные системы, используют различные соглашения и синтаксис для понятий типа структурных каталогов и файловых типов или расширений. Некоторые понятия такие как версии файла обеспечиваются одними операционными системами и не обеспечиваются другими.

Стандарт POSIX.1 разрешает, чтобы реализации поместили дополнительные ограничения на синтаксис имени файла, относительно того, какие символы разрешаются в именах файла и на длину имени файла и строк компонентов имени файла. Однако, в системе GNU, Вы не должны волноваться относительно этих ограничений; любой символ за исключением пустого символа допускается в строке имени файла, и не имеется никаких ограничений на длины строк имени файла.

7. Ввод-вывод на потоках

Эта глава описывает функции для создания потоков и выполнения ввода и вывода на них. Как обсждается в Главе 6 [Краткий обзор ввода-вывода], поток - довольно абстрактное понятие, представляющее канал связи с файлом, устройством, или процессом.

7.1 Потоки

По историческим причинам, С тип структуры данных, которая представляет поток называется FILE,а не "поток". Так как большинство библиотечных функций имеет дело с объектами типа FILE *, иногда термина указатель на файл также используется, чтобы обозначить "поток". Это ведет к беспорядку терминологии во многих книгах о С, это руководство, однако, использует термины "файл" и "поток" только в техническом смысле.

Тип FILE объявлен в заголовочном файле "stdio.h".

FILE (тип_данных) - это тип данных, используемый, чтобы представить объекты потока. Объект FILE содержит всю внутреннюю информацию о состоянии относительно соединения со связанным файлом, включая такие вещи как индикатор позиции файла и информация буферизации. Каждый поток также имеет индикаторы ошибки и состояния конца файла, которые могут быть проверены функциями feof и ferror; см. раздел 7.13 [EOF и ошибки].

Объекты FILE размещены и управляются внутренне в соответствии с библиотечными функциями ввода -вывода. Не пробуйте создавать ваши собственные объекты типа FILE; пусть библиотеки, делают это. Ваши программы должны иметь дело только с указателями на эти объекты (то есть FILE*).

- 109 -

7.2 Стандартные потоки

Когда основная функция вашей программы вызывается, уже существуют три предопределенных потока. Они представляют "стандартные" входные и выходные каналы, которые были установлены для процесса.

Эти потоки объявлены в заголовочном файле "stdio.h".

FILE * stdin (переменная) стандартный входной поток, который является нормальным источником ввода для программы.

FILE * stdout (переменная) поток стандартного вывода, который используется для нормального вывода программы.

FILE * stderr (переменная) стандартный поток ошибки, который используется для сообщений об ошибках и диагностики, выданной программой.

В системе GNU, Вы можете определять то, какие файлы или процессы соответствуют этим потокам, использующим трубопровод и средства переадресации, обеспеченные оболочкой. (Использование примитивов оболочки описано в Главе 9 [Интерфейс файловой системы]) Большинство других операционных систем обеспечивает подобные механизмы, но подробности того, как их использовать могут изменяться.

В библиотеке GNU C, stdin, stdout, и stderr - обычные переменные, которые Вы можете устанавливать точно так же как любые другие. Например, чтобы переназначить стандартный вывод файла, нужно выполнить:

```
fclose (stdout);
stdout = fopen ("standard-output-file","w");
```

Обратите внимание, что в других системах stdin, stdout, и stderr являются макрокомандами, которые Вы не можете назначать обычным способом. Но Вы можете использовать freopen, чтобы получить эффект закрытия того и повторного открытия этого. См. раздел 7.3 [Открытие потоков].

- 110 -

7.3 Открытие потоков

Открытие файла функцией fopen создает новый поток и устанавливает соединение между потоком и файлом, возможно включая создание нового файла.

Все описанное в этом разделе объявлено в заголовочном файле "stdio.h".

FILE * fopen (const char *filename , const char * opentype) (функция)

функция fopen открывает поток для ввода - вывода в файл, и возвращает указатель на поток.

opentype аргумент - это строка, которая управляет открытием файла и определяет атрибуты возникающего в результате потока. Она должна начинаться с одной из следующих последовательностей символов:

"r" Открывает существующий файл для чтения.

"w" Открывает файл для записи. Если файл уже существует, его длина обнуляется. Иначе создается новый файл.

"a" Открывает файл для добавления; то есть записи в конец файла. Если файл уже существует начальное содержимое неизменяется, и вывод потока добавляется в конец файла. Иначе, создается новый пустой файл.

"r+" Открывает существующий файл, и для чтения и для заипси. Начальное содержимое файла неизменяется, и начальная позиция файла - в начале файла.

"w+" Открывает файл, и для чтения и для запмси. Если файл уже существует, он усекается, чтобы обнулить длину. Иначе, создается новый файл.

"a+" Открывает или создают файл, и для чтения и для добавления в конец. Если файл существует, начальное содержимое неизменяется. Иначе,

создается новый файл. Начальная позиция файла для чтения - в начале файла, но вывод всегда добавляется к концу файла.

Вы видите, что "a+" запрашивает поток, который может делать и ввод и вывод. Стандарт ANSI говорит, что при использовании такого потока, Вы должны вызвать fflush (см. раздел 7.17 [Буферизация потока]) или позиционирующую файл функцию типа fseek (см. раздел 7.15 [Позиционирование файла]) при переключении чтения на запись или наоборот. Иначе, внутренние буфера не будут освобождены правильно. Библиотека GNU C не имеет этого ограничения; Вы можете делать произвольное чтение и запись на потоке в любом порядке.

Библиотека GNU C определяет один дополнительный символ для использования в opentype: символ "x" настаивает на создании нового файла, если имя файла уже существует, fopen выдаст ошибку. Это эквивалентно O_EXCL опции открыавющей функции (см. раздел 8.10 [Флаги состояния файла]).

Символ "b" в opentype имеет стандартное значение; он запрашивает двоичный поток, а не текстовый поток. Но это не имеет никакое значения в POSIX системах (включая систему GNU). Если и "+" и "b" определен, они могут применяться в любом порядке. См. раздел 7.14 [Двоичные потоки].

Любые другие символы в opentype просто игнорируются. Они могут быть значимы в других системах.

Ели происходит ошибка , fopen возвращает пустой указатель.

Вы можете иметь многократные потоки (или описатели файла) указывающие на тот же самый файл, открытый в то же самое время. Если Вы только вводите, это работает правильно, но Вы должны быть внимательны если какой-то поток выходной. См. раздел 8.5 [Предосторожности, связанные с потоком/описателем].

Это верно в равной степени, в зависимости от того, находятся ли потоки в одной программе или в отдельных программах (что может легко случиться). Может оказаться более безопасным использование средств закрытия файла, для того, чтобы избежать одновременного доступа. См.

раздел 8.11 [Блокировки файла].

int FOPEN_MAX (макрос)

Значение этой макркоманды - целочисленное постоянное выражение, которое представляет минимальное число потоков, что могут быть открыты одновременно. Значение этой константы - по крайней мере восемь, включая три стандартных потока stdin, stdout, и stderr.

FILE * freopen (const char *filename, const char * opentype, FILE *stream) (функция)

Эта функция - подобна комбинации fclose и fopen. Она сначала закрывает упоминаемый поток, игнорируя любые ошибки, которые обнаружены в процессе. (Т.к. ошибки игнорируются, Вы не должны использовать freopen на выходном потоке, если Вы фактически выполнили вывод, используя поток.) А затем открывает файл filename с режимом opentype как в fopen, и связывает его с тем же самым потоком.

Если, если операция терпит неудачу, возвращается пустой указатель; иначе, freopen возвращает поток.

freopen традиционно используется, чтобы соединить стандартный поток типа stdin с файлом вашего собственного выбора. Это полезно в программах, в которых использование стандартного потока для некоторых целей является жестко закодированным. В библиотеке GNU C, Вы можете просто закрывать стандартные потоки и открывать новые через fopen. Но другие системы испытывают недостаток этой способности, так что использование freopen более переносимо.

7.4 Закрытие потоков

Когда поток закрывается с помощью fclose, соединение между потоком и файлом отменяется. После того, как Вы закрыли поток, Вы не можете выполнять какие-нибудь дополнительные операции на нем.

int fclose (FILE *stream) (функция)

Эта функция заркыает поток и прерывает соединение с соответствующим файлом. Любой буферизированный вывод дописывается, и любой буферизированный ввод отбрасывается. Функция fclose возвратит значение 0, если файл будет закрыт успешно, и EOF, если будет обнаружена ошибка.

Важно проверить ошибки, когда Вы вызываете fclose, чтобы закрыть выходной поток, потому что в это время могут быть обнаружены реальные, каждодневные ошибки. Например, когда fclose допишет остающийся буферизированный вывод, это мжет вызвать ошибку, потому что диск полон. Даже если Вы знаете, что буфер пуст, ошибки могут происходить при закрытии файла, если Вы используете NFS.

Функция fclose объявлена в "stdio.h".

Если ваша программа завершается, или если Вы вызываете функцию выхода (см. 22.3.1 [Нормальное окончание]), все открытые потоки автоматически закрываются.

Если ваша программа завершается каким-нибудь другим способом, типа, вызыва функции аварийного прекращения работы (см. раздел 22.3.4 [Прерывание выполнения программы]) или фатального сигнала (см. Главу 21 [Обработка сигналов]), открытые потоки могут быть закрыты неправильно. Буферизированный вывод может быть не дописан. Для подробной информации относительно буферизации потоков, см. раздел 7.17 [Буферизация потока].

7.5 Простой вывод символами или строками

Этот раздел описывает функции для выполнения символьно- и строчноориентированного вывода.

Эти функции объявлены в заголовочном файле "stdio.h".

int fputc (int C, FILE *stream) (функция)

Функция fputc преобразовывает символ C, чтобы напечатать char без знака, и запишет его в поток stream. EOF возвращается, если происходит

ошибка; иначе возвращается символ C.

int putc (int C, FILE *stream) (функция)

Это аналог fputc, за исключением того, что большинство систем выполняет ее как макркоманду, делая это быстрее. Одно из следствий - то, что она может оценивать аргумент потока больше чем один раз. putc - обычно лучшая функция, для записи одиночного символа.

int putchar (int c) (функция)

Функция putchar эквивалентна putc со stdout как значением аргумента потока.

int fputs (const char * s, FILE *stream) (функция)

Функция fputs запишет строку s в поток stream. Пустой символ завершения не пишется. Эта функция так же не добавляет символ перевода строки.

Эта функция возвращает EOF, если происходит оибк записи, а иначе неотрицательное значение.

Например:

```
fputs ("Are ", stdout);
fputs ("you ", stdout);
puts ("hungry?\n", stdout);
```

Выводит текст `Are you hungry?' сопровождаемый символом перевода строки.

int puts (const char * s) (функция)

Эта функция запишет строку s в поток stdout сопровождая ее символом перевода строки. Пустой символ завершения строки не запишет.

puts - наиболее удобная функция для печати простых сообщений.

Например:

```
puts ("Это - сообщение.");
```

int putw (int w, FILE *stream) (функция)

Эта функция напишет w (int) в поток stream. Она предусматривает совместимость с SVID, но мы рекомендуем, чтобы Вы использовали fwrite взамен (см. раздел 7.12 [Блочный ввод-вывод]).

7.6 Символьный ввод

Этот раздел описывает функции для выполнения символьно- и строчноориентированного ввода. Эти функции объявлены в заголовочном файле "stdio.h".

int fgetc (FILE * stream) (функция)

Эта функция читает следующий символ как char без знака из потока stream и возвращает значение, преобразованное в int. Если происходит условие конца файла или ошибка чтения, возвращается EOF.

int getc (FILE * stream) (функция)

Это - аналог fgetc, за исключением того, что для нее допустимо (и типично) выполнение как макрокоманды, которая оценивает аргумент stream больше чем один раз. getc часто сильно оптимизирована, так что это - обычно лучшая функция, чтобы читать одиночный символ.

int getchar (void) (функция)

Функция getchar эквивалентна getc с stdin вместо аргумента stream.

Вот пример функции, которая вводит используя fgetc. Она работала бы, точно также используя getc взамен, или используя getchar () вместо fgetc (stdin).


```
int y_or_n_p (const char *question)
{
 fputs (question, stdout);
 while (1)
 {
  int c, answer; /* Напишем пробел, чтобы отделить ответ от вопроса. */
  fputc (" ", stdout);
```

```
     /* Читаем первый символ строки. Это должен быть символ ответа, но
 может и не быть. */
       c = tolower (fgetc (stdin));
       answer = c /* Отбрасываем остальную входную строку. */
       while (c != '\n') c = fgetc (stdin);
     /* Примем ответ, если он был допустим. */
       if (answer == 'y') return 1;
       if (answer == 'n') return 0;
     /* Ответ был недопустим: просим о допустимом ответе. */
       fputs ("Please answer y or n:", stdout);
      }
     }
```

int getw (FILE * stream) (функция)

Эта функция читает word (то есть int) из stream. Она предусматривает совместимость с SVID. Мы рекомендуем, чтобы Вы использовали вместо этого fread (см. раздел 7.12 [Блочный ввод-вывод]) .

7.7 Строчно ориетированный ввод

Так как много программ интерпретируют ввод на основе строк, удобно иметь функции, чтобы читать строку из stream.

Стандартный C имеет функции, чтобы делать это, но они не очень безопасны: пустые символы и даже длинные строки могут сбивать их. Библиотека GNU обеспечивает нестандартную функцию getline, которая позволяет читать строки надежно.

Другое расширение GNU, getdelim, обобщает getline. Она читает разграниченную запись, определенную как все после следующего вхождения

заданного символа-разделителя.

Все эти функции объявлены в "stdio.h".

size_t getline (char ** lineptr, size _t * n, FILE * stream) (функция)

Эта функция читает всю строку из stream, сохраняя текст (включая символ перевода строки и пустой символ завершения) в буфере и хранит буферный адрес в * lineptr.

Перед вызовом getline, Вы должны поместить в *lineptr адрес буфера *n байт длиной, размещенный malloc. Если этот буфер достаточно большой чтобы вместить строку, getline, сохранит строку в этом буфере. Иначе, getline делает больший буфер используя realloc, сохраняя новый буферный адрес обратно в *lineptr и увеличенный size обратно в *n. См. раздел 3.3 [Беспрепятственное резервирование].

Если Вы устанавливаете *lineptr как пустой указатель, и обнуляете *n, перед обращением, то getline, зарезервирует начальный буфер для Вас, вызывая malloc.

В любом случае, когда getline завершается, *lineptr - это char * который указывает на текст строки.

Когда getline успешно завершвется, она возвращает число прочитанных символов (включая символ перевода строки, но не, включая пустой символ завершения). Это значение дает возможность Вам отличить пустые символы, которые являются частью строки от пустого символа, вставленного как признак конца.

Эта функция - расширение GNU, но это - рекомендуемый способ читать строки из stream. Альтернативные стандартные функции ненадежны.

Если происходитошибка происходит или достигнут конец файла, getline возвращает -1.

size_t getdelim (char ** lineptr, size _t * n, int delimiter, FILE * stream) (функция)

Эта функция - подобна getline за исключением того, что символ, который сообщает, чтобы она прекратила читать - не обязательно символ перевода строки. Аргумент delimiter определяет символ - разделитель; getdelim будет читать, пока не увидит этот символ (или конец файла).

Текст сохраняется в lineptr, включая символ - разделитель и пустой символ завершения. Подобно getline, getdelim делает lineptr большим, если он не достаточно большой.

getline фактически реализована в терминах getdelim, как показано ниже:

```
size_t getline (char **lineptr, size_t *n, FILE *stream)
{
 return getdelim (lineptr, n, '\n', stream);
}
```

char * fgets (char * s, int count, FILE * stream) (функция)

Функция fgets читает символы из потока stream включая символ перевода строки и сохраняет их в строке s, добавляя пустой символ, чтобы отметить конец строки. Вы должны обеспечить место для count символов в s, но читается count - 1 символов. Дополнительное символьное место используется, чтобы содержать пустой символ в конце строки.

Если, система уже в конце файла, когда Вы вызываете fgets, то содержимое массива s, неизменяется, и возвращается пустой указатель. Пустой указатель также возвращается, если происходит ошибка чтения, возвращаемое значение - указатель s.

Предупреждение: если входные данные имеют пустой символ, Вы не может использовать fgets. Так что не используйте fgets, если Вы не знаете, что данные не могут содержать пустой символ. Не используйте ее, чтобы читать файлы, отредактированные пользователем, потому что, если пользователь вставляет пустой символ, Вы должны обработать это правильно

или напечатать сообщение об ошибках. Мы рекомендуем использовать getline вместо fgets.

char * gets (char * s) (функция)

Эта функция читает символы из потока stdin до следующего символа символа перевода строки, и сохраняет их в строке s. Символ перевода строки отбрасывается (обратите внимание, что это отличает ее от поведения fgets, которая копирует символ перевода строки в строку). Если она сталкивается с ошибкой чтения или концом файла, она возвращает пустой указатель; иначе она возвращает s.

Предупреждение: эта функция очень опасна, потому что она не обеспечивает никакой защиты против переполнения строки s. Библиотека GNU включает ее только для совместимости. Вы должны всегда использовать fgets или getline взамен. Чтобы напомнить Вам это, компоновщик (при использовании GNU ld) выдаст предупреждение всякий раз, когда Вы используете gets.

7.8 Обратное чтение

В программах синтаксического анализатора часто полезно исследовать следующий символ во входном потокек без того, чтобы удалить его из потока. Это называется "заглядывание вперед" при вводе, потому что ваша программа бросает взгляд на то что она затем будет читать.

При использовании потока ввода - вывода, Вы можете заглядывать вперед при вводе первым чтением и затем обратным чтением (так называемм выталкиванием обратно в поток). Обратное чтение делает символ доступным для следующего обращения к fgetc или другой входной функции на этом потоке.

7.8.1 Что такое способ обратного чтения

Это иллюстрированное объяснение обратного чтения. Предположим, что Вы имеете поток, читая файл, который содержит только шесть символов, символы "foobar". Предположите, что Вы пока прочитали три символа. Ситуация выглядит следующим образом:

```
f o o b a r
    ^
```

так что следующий входной символ будет "b".

Если вместо того, чтобы читать "b" Вы выполняете обратное чтение символа "o", Вы получаете примерно такую ситуацию:

```
f o o b a r
    |
o-
^
```

так, что следующие входные символы будут "o" и "b".

Если Вы обратно читаете "9" вместо "o", Вы получите это:

```
f o o b a r
    |

9-
^
```

так, что следующие входные символы будут "9" и "b".

7.8.2 Использование ungetc для осуществления обратного чтения

Функция для чтения символа обратно называется ungetc, потому что она обращает действие getc.

int ungetc (int C, FILE *sream) (функция)

Функция ungetc помещает символ C обратно во входной поток. Так что следующий ввод из потока будет читать C прежде, чем что-нибудь еще.

Если C - EOF, ungetc не делает ничего и только возвращает EOF. Это позволяет Вам вызывть ungetc с возвращаемым значением getc без проверки ошибки из getc.

Символ, который Вы помещаете обратно, не обязательно тот который фактически читался из потока. Т. е. читать какой-нибудь символ из потока перед его обратным чтением не обязательно! Но это - странный способ писать программу; обычно ungetc используется только, чтобы читить обратно символ, который только что читался из того же самого потока.

Библиотека GNU C поддерживает только один символ pushback другими словами, нельзя вызвать ungetc дважды без ввода между вызовами. Другие системы могут позволять Вам, помещать обратно много символов; тогда чтение из потока восстанавливает символы в обратном порядке, от того как они были помещены.

Выталкивание обратных символов не изменяет файл; и воздействует только на внутреннюю буферизацию потока. Если вызывается позиционирующая файл функция (типа fseek или rewind; см. раздел 7.15 [Позиционирование файла]), все отложенные помещаемые-обратно символы отбрасываются.

Обратное чтение символа в поток, который находится в конце файла, стирает индикатор конца файла для потока. После того, как Вы читаете тот символ, пробуя читать снова Вы столкнетесь с концом файла.

Вот пример, показывающий использование getc и ungetc, чтобы

перескочить символы промежутков. Когда эта функция достигает символа не-промежутка, она читает обратно этот символ, и он будет замечен снова на следующей операции чтения из потока.

```
#include
#include

void skip_whitespace (FILE * stream)
{
 int c;
 do
```



```
  /* Нет нужды проверять EOF, потому что это - не isspace, а ungetc
игнорирует EOF. */
    c = getc (stream);
  while (isspace (c));
  ungetc (c, stream);
}
```

7.9 Форматированный вывод

Функции, описанные в этом разделе (printf и др.) обеспечивают, удобный способ выполнять форматированный вывод. Вы вызываете printf со строкой формата или строкой шаблона, которая определяет, как форматировать значения остающихся аргументов.

Если ваша программа не фильтр, который специально выполняет строчно- или символьно- ориентированную обработку, использование printf, или одной из других зависимых функций, описанных в этом разделе - обычно самый простой и наиболее краткий способ выполнить вывод. Эти функции особенно полезны для печати сообщений ошибок, таблицы данных, и т.п..

7.9.1 Основы форматированного вывода

Функция printf может использоваться, чтобы печатать любое число аргументов. Аргумент строки шаблона, который Вы обеспечиваете в обращении, обеспечивает информацию не только относительно числа дополнительных аргументов, но также относительно их типов и какой стиль должен использоваться для печати.

Обычные символы в строке шаблона просто записываются в выходной поток как есть, в то время как спецификации преобразования, представленные символом `%' в шаблоне заставляют последующие аргументы форматироваться при записи в выходной поток. Например:

```
int pct = 37;
char filename[] = "foo.txt";
printf ("Processing of `%s' is %d%% finished.\nPlease be
patient.\n", filename, pct);
```


Производит вывод:

```
Processing of `foo.txt' is 37% finished. Please be patient.
```

Этот пример показывает использование "%d" преобразования, чтобы определить, что int аргумент должен быть напечатан в десятичной записи, "%s" преобразования, чтобы определить печать строкового аргумента, и "%%" преобразования, чтобы печатать непосредственно символ "%".

Имеются также преобразования для печати целочисленного аргумента как значения без знака в восьмеричной, десятичной, или шестнадцатеричной системе счисления ("%o", "%u", или "%x", соответственно); или как символьного значения ("%c").

Числа с плавающей запятой могут быть напечатаны в нормальной, с фиксированной запятой записи, используя "%f" преобразование или в экспоненциальном представлении чисел, используя "%e" преобразование. "%g" преобразование использует или "%e" или формат "%f", в зависимости от того что более подходит для заданного числа.

Вы можете управлять форматированием более точно, написав модификаторы между "%" и символом, который указывает какое преобразование применить. Они немного изменяют обычное поведение преобразования. Например, большинство спецификаций преобразования разрешает Вам определять минимальную ширину поля и флаг, указывающий, хотите ли Вы чтобы результат выравнивался по правому или по левому краю поля.

Специфические флаги и модификаторы, которые разрешаются и их интерпретация, изменяются в зависимости от преобразований. Они все описаны более подробно в следующих разделах. Не волнуйтесь, если это все кажется чрезмерно сложным; Вы можете почти всегда получать приемлемый вывод без использования какого-нибудь из модификаторов вообще. Модификаторы обычно используются, чтобы делать просмотр вывода в таблицах.

7.9.2 Синтаксис преобразования вывода

Этот раздел обеспечивает подробности относительно точного синтаксиса спецификаций преобразования, которые могут появляться в printf строке шаблона.

Символы в строке шаблона, которые - не часть спецификации преобразования, печатаются как есть в выходной поток. Последовательности мультисимволов (см. Главу 18 [Расширенные символы]) разрешены в строке шаблона.

Спецификации преобразования в строке шаблона имеют общую форму:

% флаги ширины [. точность] тип преобразования

Например, в спецификаторе преобразования "%-10.8ld", "-" является флагом, "10" определяет ширину поля, точность - "8", символ "l" является модификатором типа, и "d" определяет стиль преобразования. (Этот специфический спецификатор типа говорит, что печатается long int аргумент в десятичной записи, с минимумом 8 цифр, выровненных по левому краю в поле по крайней мере шириной 10 символов.)

Более подробно, спецификации преобразования вывода состоят из начального символа `%', сопровождаемого последовательностью:

* Нуль или большее количество символов флага, которые изменяют нормальное поведение спецификации преобразования.

* Десятичное целое число, определяющее минимальную ширину поля. Если нормальное преобразование производит меньшее количество символов чем этот, поле, дополняется пробелами до заданной ширины. Это - минимальное значение; если нормальное преобразование производит большее количество символов чем этот, поле не усечено. Обычно, вывод выравнен по правому краю внутри поля.

Вы можете также определять ширину поля "*". Это означает что следующий аргумент в списке параметров (до фактического значения, которое будет напечатано) используется как ширина поля. Значение должно

быть int. Если значение является отрицательным, это означает, установить "-" флаг (см. ниже) и использовать абсолютное значение как ширину поля.

* Точность, чтобы определить число цифр, которые нужно написать для числовых преобразований. Если точность определена, она состоит из точки (".") сопровождаемой необязательно десятичным целым числом (если оно опущено - значение округляется).

Вы можете также определять точность "*". Это означает что следующий аргумент в списке параметров (до фактического значения, которое будет напечатано) используется как точность. Значение должно быть int и игнорируется, если оно отрицательное. Если Вы определяете "*" и для ширины и для точности, то аргумент ширны предшествует аргументу

точности. Другие C библиотеки могут не распознавать такой синтаксис.

7.9.3 Таблица форматов вывода Эта таблица содержит различные форматы вывода:

'%d', '%i': Вывод целого числа как десятичного числа со знаком. См. Раздел 7.9.4 [Целочисленные Форматы]. '%d' и '%i' являются синонимами для printf, но отличаются при использовании scanf для ввода (см. Раздел 7.11.3 [Таблица форматов ввода]).
'%o': Печатает целое число как восьмеричное число без знака. См. Раздел 7.9.4 [Целочисленные форматы].
'%u': Печатает целое число как десятичное число без знака. См. Раздел 7.9.4 [Целочисленные форматы].
'%Z': Печатает целое число как десятичное число без знака, принимая как тип size_t. Детали см. в Разделе 7.9.4 [Целочисленные Форматы]. Этот формат является расширением GNU.
'%x', '%X': Печатают целое число как шестнадцатеричное без знака. '%x' использует символы нижнего регистра а '%X' - верхнего регистра. См. Раздел 7.9.4 [Целочисленные форматы].
'%f': Печатает число с плавающей запятой в нормальной записи (с фиксированной запятой). См. подробности в Разделе 7.9.5 [Форматы с плавающей запятой].
'%e', '%E': Печатают число с плавающей запятой в экспоненциальном представлении чисел. '%e' использует символы нижнего

регистра, а '%E' - верхнего регистра.
'%g', '%G': Выводят число с плавающей запятой либо в нормальном, либо в экспоненциальном представлении. '%g' использует символы нижнего регистра, а '%G' - верхнего регистра.
'%c': Печатает одиночный символ. См. Раздел 7.9.6 [Другие форматы Вывода].
'%s': Печатает строку. См. Раздел 7.9.6 [Другие форматы Вывода].
'%p': Выводит значение указателя. См. Раздел 7.9.6 [Другие форматы Вывода].
'%n': Содержит число уже напечатанных символов. См. Раздел 7.9.6 [Другие Форматы Вывода]. Обратите внимание, что эта спецификация формата никогда не производит никакого вывода.
'%m': Печатает строку, соответствующую значению errno. (Этот формат формат является расширением GNU.) См. Раздел 7.9.6 [Другие Форматы Вывода].
'%%': Печатает символ `%'. См. Раздел 7.9.6 [Другие Форматы Вывода].

Если синтаксис спецификации формата вывода является недопустимым, то результат непредсказуем. Если не достаточно аргументов функции, чтобы обеспечить значения для всех спецификаций преобразования в строке шаблона, или если аргументы неправильных типов, результаты непредсказуемы. Если Вы обеспечиваете большее количество аргументов чем используется в спецификации формата, дополнительные аргументы, просто игнорируются; это иногда полезно.

7.9.4 Целочисленные Форматы Этот раздел описывает опции для '%d', '%i', '%o', '%u', '%x', '%X', и '%Z' спецификаций преобразования. Эти преобразования печатают целых числа в различных форматах. Форматы '%d' и '%i' печатают целочисленный аргумент как десятичное число со знаком; в то время как '%o', '%u', и '%x' печатают аргумент как восьмеричное, десятичное, или шестнадцатеричное без знака соответственно. Формат '%X' - точно то же что и '%x' за исключением того, что она использует символы 'ABCDEF' в качестве цифр вместо 'abcdef'. '%Z' подобна '%u' но принимает аргумент типа size_t.

Имеют значение следующие флаги:

'-' Выравнивание слева результата в поле вывода (вместо нормального выравнивания справа).

'+' Для знаковых форматов '%d' и '%i', печатает знак '+', если значение положительно.
' ' Для знаковых форматов '%d' и '%i', если результат не начинается со знака '+' или знака '-', то ставит перед ним пробел.
'*' Для формата '%o' ставит '0' первой цифрой, как будто, увеличивая точность. Не делает ничего полезного для форматов

'%d', '%i', или '%u'. Использование этого флага производит вывод, который может анализироваться функциями strtoul (см. Раздел 14.7.1 [Синтаксический анализ Целых чисел]) и scanf с форматом '%i' (см. Раздел 7.11.4 [Числовые форматы ввода]).

'0' Дополняют поле нулями вместо пробелов. Нули помещаются после какой-нибудь индикации относительно знака. Этот флаг игнорируется, если указан флаг '-' , или если указана точность.

Если указана точность, она определяет минимальное число цифр; в случае необходимости выводятся дополнительные нули вначале. Если Вы не указываете точность, печатается столько цифр числа, сколько требуется. Если Вы преобразовываете значение нуля с явной нулевой точностью, то никакие символы не выводятся вообще.

Без модификатора типа, соответствующий аргумент обрабатывается как int (для знаковых преобразований '%i' и '%d') или int без знака (для преобразований без знака '%o', '%u', '%x', и '%X'). Заметьте, что т. к. printf и ее производные функции, любой аргумент типа char и short автоматически приводится к типу int аргументами, заданными по умолчанию. Для аргументов других целочисленных типов, Вы можете использовать следующие модификаторы:

' H ' Определяет, что аргумент - short int или short unsigned int.

' l ' Определяет, что аргумент - long int или long unsigned int

' L ' Определяет, что аргумент - long long int. (Этот тип является расширением, обеспечиваемым компилятором GNU C. На системах, которые не поддерживают сверхдлинные целые числа, это - тоже что long int.)

Модификаторы для типа аргумента не применимы к '%Z', так как единственной целью '%Z' является указать тип данных size_t.

Вот пример использования строки шаблона:

'| %5d| %-5d| %+5d| %+-5d| %5d| %05d| %5.0d| %5.2d| %d|\n '

для печати числа, используя различные опции для преобразования '%d':

```
| 0    | 0    | + 0    | + 0  | 0    | 00000|      |  00  |  0 |
| 1    | 1    | + 1    | + 1  | 1    | 00001| 1    |  01  |  1 |
```



```
| -1   | -1   | -1     | -1   | -1   | -0001| -1   | -01  | -1 |
|100000|100000|+ 100000|100000|100000|100000|100000|100000|...
```

В частности обратите внимание на то, что получается в последнем случае, где аргумент слишком большой, чтобы поместиться в минимальной заданной ширине поля.

Вот еще несколько примеров, показывающих, как выводятся беззнаковые целые под различными опциями формата, используя строку шаблона:

'| %5u | %5o | %5x | %5X | %*5o | %*5x | %*5X | %*10.8x | \n'

```
|    0 |    0 |   0 |   0 |     0 |   0x0 |   0X0 |0x00000000|
|    1 |    1 |   1 |   1 |    01 |   0x1 |   0X1 |0x00000001|
|100000|303240|186a0|186A0|0303240|0x186a0|0X186A0|0x000186a0|
```

7.9.5 Преобразования с плавающей запятой

Этот раздел содержит спецификации форматов вывода чисел с плавающей запятой: '%f', '%e', '%E', '%g', и '%G'. Формат '%f' печатает аргумент в формате с фиксированной запятой, выводя его на экран в виде [-] ddd.ddd, где число цифр после десятичной точки определяется точностью, которую Вы указали.

Формат '%e' печатает аргумент в экспоненциальном представлении, выводя его на экран в виде [-] d.ddde [+ | -] dd. Число цифр после десятичной точки также определяется точностью. Экспонента всегда содержит по крайней мере две цифры. На этот формат похож '%E', но экспонента отмечена символом ' E ' вместо ' e '.

Форматы '%g' и '%G' печатают аргумент в стиле '%e' и '%E' соответственно, если экспонента меньше чем -4 или больше либо равна точности; в противном случае они используют стиль формата '%f'. Конечные нули из дробной части результата удаляются, а символ десятичной точки появляется только, если он сопровождается цифрой.

Чтобы изменить поведение функции, могут применяться следующие флаги:

'-' Выравнивание слева результата в поле вывода. Обычно результат выравнивается справа.

'+' Всегда выводится знак 'плюс' или 'минус'.

' ' Если результат не начинается со знака 'плюс' или 'минус', ставит перед ним пробел.

'#' Определяет, что результат должен всегда включать десятичную точку, даже если за ней не следует никаких цифр. Для форматов '%g' и '%G', конечные нули после десятичной точки удаляться не будут.
'0' Дополняет поле нулями вместо пробелов; нули помещаются после знака. Этот флаг игнорируется, если указан флаг '-'.

Точность определяет, сколько цифр следуют за символом десятичной точки для форматов '%f','%e', и '%E'. Точность, заданная по умолчанию для этих форматов - 6. Если она явно задана как 0, то символ десятичной точки подавляется. Для форматов '%g' и '%G', точность определяет сколько значащих цифр печатать. Значащие цифры это первая цифра перед десятичной точкой, и все цифры после нее. Если для '%g' или '%G' точность - 0 или не задана, то она обрабатывается как если бы была 1. Если напечатанное значение не может быть выражено точно заданным количеством цифр, то значение округляется до ближайшего подходящего значения.
Без модификатора типа, форматы с плавающей запятой используют аргумент двойного типа. (По умолчанию любой аргумент типа float автоматически преобразуется в double.) Поддерживаются следующие модификаторы типов:

'L' Определяет, что аргумент типа long double. Вот несколько примеров как при выводе используются различные форматы чисел с плавающей запятой. Все числа были напечатаны, используя следующий шаблон строки:
'|%12.4f|%12.4e|%12.4g|\n'

```
|      0.0000|  0.0000e+00|           0|
|      1.0000|  1.0000e+00|           1|
|     -1.0000| -1.0000e+00|          -1|
|    100.0000|  1.0000e+02|         100|
|   1000.0000|  1.0000e+03|        1000|
|  10000.0000|  1.0000e+04|       1e+04|
|  12345.0000|  1.2345e+04|   1.234e+04|
| 100000.0000|  1.0000e+05|       1e+05|
| 123456.0000|  1.2346e+05|   1.234e+05|
```


Обратите внимание как формат '%g' выводит конечные нули.

7.9.6 Другие Форматы Вывода Этот раздел описывает различные форматы, используемые printf.

Формат '%c' печатает одиночный символ. Аргумент типа int сначала преобразовывается в unsigned char. Может использоваться флаг '-' для задания выравнивания слева в поле вывода, но точность или модификатор типа не могут быть определены. Например:

```
printf ('%c%c%c%c%c', 'h', 'e', 'l', 'l', 'o');
```
выводит
`hello'.

```
printf ('%3s%-6s', 'no', 'where');
```
выводит
`nowhere'.

Если Вы случайно передаете в качестве аргумента для формата '%s' пустой указатель преобразования, библиотека GNU выведет '(null)'. Мы думаем, что это более полезно чем сообщение об ошибке.

```
fprintf (stderr, 'can't open `%s': %m\n', filename);
```
является эквивалентным:
```
fprintf (stderr, 'can't open `%s': %s\n', filename, strerror (errno));
```

Формат '%m' - расширение библиотеки GNU C.

Формат '%p' печатает значение указателя. Соответствующий аргумент должен иметь тип void*. Практически, Вы можете использовать любой тип

указателя.
В системе GNU, непустые указатели печатаются как integers unsigned, как при использовании формата '%#x'. Пустые указатели печатаются как '(nil)'. (В других системах указатели могут печататься по-другому.)

Например:
```
printf ('%p', 'testing');
```
печатает '0x' сопровождаемый шестнадцатеричным числом адреса строковой константы 'testing'.
Вы можете добавить флаг '-' к формату '%p', чтобы обеспечить выравнивание слева, но не должны определяться никакие другие флаги, точность, или модификаторы типа.

Формат '%n' - отличается от всех других форматов вывода. Он использует аргумент, который должен быть указателем на int, но вместо того, чтобы печатать что-нибудь, он содержит число уже напечатанных символов. Модификаторы типа 'h' и 'l' задают, что указывается аргумент типа short int* или long int * вместо int *, но не позволяется указывать никакие флаги, ширину поля, или точность.

Например,
```
int nchar; printf ('%d %s%n\n', 3, 'bears', &nchar);
```
печатает:
```
3 bears
```
и устанавливает nchar в значение 7, потому что строка '3 bears' содержит семь символов.

Формат '%%' выводит символ `%'. Этот формат не использует аргумент.

7.9.7 Функции Форматированного Вывода

Этот раздел описывает, как вызвать printf и относящиеся к ней функции. Прототипы для этих функций находятся в файле 'stdio.h'. Т. к. эти функции принимают переменное число аргументов, Вы должны объявить прототипы для них перед использованием. Конечно, самый простой способ удостовериться, что Ваши прототипы все правильные, это включить 'stdio.h'.

- Функция: int printf (const char *TEMPLATE, ...)
Функция printf выводит указываемые аргументы под управлением шаблона строки TEMPLATE в поток stdout. Она возвращает число напечатанных символов, или отрицательное значение при ошибке вывода.

- Функция: int fprintf (FILE *stream, const char *template, ...)
Эта функция - аналог printf, за исключением того, что вывод записывается в указанный поток вместо stdout.

- Функция: int sprintf (char *s, const char *template, ...)
Она подобна printf, за исключением того, что вывод сохраняется в символьном массиве s вместо записи в поток. Пустой символ записывается в конец строки.
Функция sprintf возвращает число символов, содержащихся в массиве s, исключая пустой символ завершения.
Поведение этой функции неопределено, если копирование происходит между пересекающимися объектами. Например, если s также является аргументом, который выводится под управлением формата '%s'. См. Раздел 5.4 [Копирование и Конкатенация].
Предупреждение: функция sprintf может быть опасна, т. к. она может потенциально выводить большее количество символов чем размер, зарезервированный для строки s. Не забудьте, что ширина поля, заданная в спецификации формата - только минимальное значение.
Чтобы избежать этой проблемы, Вы можете использовать snprintf или asprintf, описанные ниже.

- Функция: int snprintf (char *s, size_t size, const char *TEMPLATE, ...)
Функция snprintf подобна sprintf, за исключением того, что аргумент размера определяет максимальное число выводимых символов. Конечный пустой символ подпадает под это ограничение

Возвращаемое значение - число сохраненных символов, не включая пустой символ завершения. Если это значение равняется size-1, то в s недостаточно места для всего вывода. Вы должны пробовать снова с большей строкой вывода. Вот пример такого кода:

```
/* Выдаем сообщение, описывающее значение переменной
   с именем NAME и значением VALUE */ char * make_message
(char *name, char *value) {
  /* Предположим, что нам понадобится не больше чем 100
     символов. */ int size = 100; char *buffer = (char *)
  xmalloc (size);
```



```
while (1) {
      /* Попытаемся напечатать в отведенном пространстве. */
      int nchars = snprintf (buffer, size, 'value of %s is
                             %s', name, value);
      /* If that worked, return the string. */
      if (nchars < size) return buffer;
      /* Иначе попытаемся еще раз с удвоенным количеством
         символов */ size *= 2; buffer = (char *) xrealloc
      (size, buffer);
    }
}
```

На деле, часто проще использовать asprintf, см. ниже.

7.9.8 Форматируемый Вывод, Размещаемый Динамически

Функции в этом разделе форматируют вывод и помещают результаты в динамически размещенную память.

- Функция : int asprintf (char **ptr, const char *template, ...)

Эта функция похожа на sprintf, за исключением того, что она динамически распределяет строку для вывода (как malloc; см. Раздел 3.3 [Беспрепятственное Резервирование], вместо того, чтобы помещать вывод в буфер, определяемый заранее. Аргумент ptr должен быть адресом объекта char*, и asprintf сохраняет в нем указатель на размещенную строку.

```
char * make_message (char *name, char *value) {
  char *result; asprintf (&result, 'value of %s is %s', name,
  value); return result;
}
```

-Функция: int obstack_printf(struct obstack* obstack,const char* template,...)

Эта функция подобна asprintf, за исключением того, что она использует obstack, чтобы зарезервировать пространство в памяти. См. Раздел 3.4 [Obstack].

Символы дописываются в конец текущего объекта. Чтобы добраться до них, Вы должны закончить объект функцией obstack_finish (см. Раздел 3.4.6 [Возрастастающие Объекты]).

7.9.9 Переменные Аргументы Функций Вывода

Функции vprintf и подобные позволяют Вам определять ваши собственные различные printf-подобные функции, которые используют ту же самую внутреннюю организацию как и встроенные форматирующие функции вывода.

Наиболее естественный способ определять такие функции состоит в том, чтобы использовать конструкцию типа 'Вызвать printf и передать ему этот шаблон плюс все мои аргументы пропуская первые пять.' Но не имеется никакого способа сделать это на C, и было бы трудно это сделать, потому что на уровне Языка C не имеется никакого способа указать сколько аргументов получает ваша функция.

Так как тот метод невозможен, мы поставляем дополнительные функции типа vprintf, которые позволяет Вам передавать va_list, чтобы описать

'все мои аргументы после первых пяти.'

Перед вызовом vprintf или других функций, перечисленных в этом разделе, Вы должны вызвать va_start (см. Раздел A. 2 [Variadic Функции]) чтобы установить указатель на переменные аргументы. После этого Вы можете вызывать va_arg, чтобы выбрать аргументы, которые Вы хотели обработать.

Если ваш vуказатель a_list указывает на аргументы вашего выбора, Вы можете вызвать vprintf. Этот аргумент и все последующие аргументы, которые были переданы вашей функции, используются vprintf наряду с шаблоном, который Вы определили отдельно.

В некоторых других системах, указатель va_list может стать недопустимым после обращения к vprintf, так что Вы не должны использовать va_arg после того, как Вы вызываете vprintf. Вместо этого, Вы должны вызвать va_end, чтобы отменить указатель. Затем Вы можете безопасно вызывать va_start для другой переменной указателя и начинать выбирать аргументы снова через этот указатель. Вызов vprintf не разрушает список параметров вашей функции.

GNU C не имеет таких ограничений. Вы можете продолжать выбирать аргументы из списка va_list после выполнения через vprintf, тогда va_end

- пустая команда. (Примечание, последующие обращения va_arg выберут те же самые аргументы, как и те, что предварительно использованы vprintf.)

Прототипы для этих функций объявлены в 'stdio.h'.

- Функция: int vprintf (const char *template, va_list ap)

Эта функция подобна printf за исключением того, что ей, вместо переменной, содержащей число аргументов, необходимо передавать указатель на список параметров.

- Функция: int vfprintf (FILE *stream, const char *template, va_list ap)

Эта функция - эквивалент fprintf с переменным списком параметров, заданным непосредственно как и для vprintf.

- Функция: int vsprintf (char *s, const char *template, va_list ap)

Эта функция - эквивалент sprintf с переменным списком параметров, заданным непосредственно как в vprintf.

- Функция: int vsnprintf (char *s, size_t size, const char *template, va_list ap)

Это функция - эквивалент snprintf с переменным списком параметров, заданным непосредственно как в vprintf.

- Функция: int vasprintf (char **ptr, const char *template, va_list ap)

Функция vasprintf - эквивалент asprintf с переменным списком параметров, заданным непосредственно как и для vprintf.

- Функция: int obstack_vprintf (struct obstack *obstack, const char* template, va_list ap)

Функция obstack_vprintf - эквивалент obstack_printf с переменным списком параметров, заданным непосредственно как для vprintf.

Ниже приводится пример, показывающий, как Вы могли бы использовать vfprintf. Это - функция, которая выводит сообщения об ошибках в поток stderr, вместе с префиксом, указывающим имя программы (см. Раздел 2.3

[Сообщения об ошибках], описание program_invocation_short_name).

```
#include
#include

void
eprintf (const char *template, ...)
{
  va_list ap;
  extern char *program_invocation_short_name;

  fprintf (stderr, '%s: ', program_invocation_short_name);
  va_start (ap, count);
```

```
       vfprintf (stderr, template, ap);
       va_end (ap);
     }
```

Вы могли бы вызывать eprintf например так:

```
     eprintf ('file `%s' does not exist\n', filename);
```

7.9.10 Синтаксический разбор Строки Шаблона

Вы можете использовать функцию parse_printf_format, для получения информации относительно числа и типов аргументов, которые ожидаются данной строкой шаблона. Эта функция вызывает интерпретаторы, которые обеспечивают интерфейс в printf во избежание передачи недопустимых аргументов.

Все символы, описанные в этом разделе объявлены в файле заголовков 'printf.h'.

- Функция: size_t parse_printf_format (const char *template, size_t n, int* argtypes)

Эта функция возвращает информацию относительно числа и типов аргументов, ожидаемых printf в строке шаблонов. Информация сохраняется в массиве argtypes; каждый элемент этого массива описывает один аргумент. Эта информация предоставляется в виде различных 'PA_' макрокоманд, которые перечисляются ниже.

Аргумент n определяет число элементов в массиве argtypes. Это наибольшее число элементов, которые parse_printf_format пробует написать.

Parse_printf_format возвращает общее число аргументов, требуемых

шаблоном. Если это число больше n, то возвращаемая информация, описывает только первые n аргументов. Если Вы хотите получить информацию относительно большего, чем n, числа аргументов, зарезервируйте больший массив и вызовете parse_printf_format снова.

Типы аргумента закодированы как комбинация базисного типа и битов флага модификатора.

- Макрос: int PA_FLAG_MASK

Эта макрокоманда - маска для битов флага модификатора типа. Вы можете написать выражение (argtypes [i] & PA_FLAG_MASK) чтобы извлечь только биты флага для аргумента, или (argtypes [i] & ~PA_FLAG_MASK) чтобы извлечь только базисный код типа.

Имеются символические константы, которые представляют базисные типы; они устанавливаются для значений integer.

PA_INT Определяет, что исходный тип - int.

PA_CHAR Определяет, что исходный тип - int, приведеннное к char.

PA_STRING Определяет, что исходный тип - char *, строка с нулевым символом в конце.

PA_POINTER Определяет, что исходный тип - void *, произвольный указатель.

PA_FLOAT Определяет, что исходный тип с плавающей точкой.

PA_DOUBLE Определяет, что исходный тип - double.

PA_LAST Вы можете определять дополнительные исходные типы для ваших собственных программ как смещения из PA_LAST.

Например, если у Вас определены типы данных 'foo' и 'bar' с их собственными специализированными форматами для printf, то Вы можете определять эти типы как:

```
#define PA_FOO  PA_LAST
#define PA_BAR  (PA_LAST + 1)
```

Имеются биты флага, которые изменяют базисный тип. Они объединены с кодом для базисного типа, используя операцию или.

PA_FLAG_PTR

Если этот бит устанавливается, это указывает, что закодированный тип - указатель на исходный тип, а не непосредственное значение.

Например, 'PA_INT | PA_FLAG_PTR' представляет тип `int *'.

PA_FLAG_SHORT

Если этот бит устанавливается, это указывает, что исходный тип изменяется как short. (Соответствует модификатору типа 'h'.)

PA_FLAG_LONG

Если этот бит устанавливается, это указывает, что исходный тип изменяется как long. (Соответствует модификатору типа 'l'.)

PA_FLAG_LONG_LONG

Если этот бит устанавливается, это указывает, что исходный тип

изменяется как long long.
PA_FLAG_LONG_DOUBLE
Это - синоним для PA_FLAG_LONG_LONG, используемого обычно с исходным типом PA_DOUBLE для обозначения типа long double.

7.9.11 Пример Синтаксического анализа Строки Шаблона

```
/* Проверка, является ли NARGS, определяющий объекты вектора
   ARGS формату строки FORMAT:
   если да, возвращает 1.
   в противном случае возвращает 0 после вывода сообщения об ошибке */

int
validate_args (char *format, int nargs, OBJECT *args)
{
  int *argtypes;
  int nwanted;

  /* Получить информацию об аргументах.
     Каждый спецификатор формата должен быть длиной по крайней мере
     в два символа, поэтому не может быть спецификаторов длиной
     длиной больше половины длины строки.
  */

  argtypes = (int *) alloca (strlen (format) / 2 * sizeof (int));
  nwanted = parse_printf_format (string, nelts, argtypes);

  /* Проверим количество аргументов  */
  if (nwanted > nargs)
    {
      error ('too few arguments (at least %d required)', nwanted);
      return 0;
    }
```



```
  /* Проверим тип, требуемый для каждого аргумента,
     и соответствие ему данного аргумента.  */
  for (i = 0; i < nwanted; i++)
    {
      int wanted;

      if (argtypes[i] & PA_FLAG_PTR)
        wanted = STRUCTURE;
      else
        switch (argtypes[i] & ~PA_FLAG_MASK)
          {
          case PA_INT:
          case PA_FLOAT:
          case PA_DOUBLE:
            wanted = NUMBER;
            break;
          case PA_CHAR:
            wanted = CHAR;
            break;
          case PA_STRING:
            wanted = STRING;
            break;
          case PA_POINTER:
            wanted = STRUCTURE;
            break;
          }
      if (TYPE (args[i]) != wanted)
        {
          error ('type mismatch for arg number %d', i);
          return 0;
        }
    }
  return 1;
}
```

7.10 Настройка printf

Библиотека GNU C позволяет Вам определять ваши собственные спецификаторы преобразования для строк шаблона printf, т.е. научить printf выводить важные структуры данных вашей программы так, как Вам этого хочется.

Это можно сделать, указав формат преобразования с помощью функции register_printf_function; см. Раздел 7.10.1 [Указание Новых Форматов Вывода]. Один из аргументов, которые Вы передаете этой функции - указатель на функцию обработчика, которая производит фактический вывод; за более подробной информацией о написании этой функции обращайтесь к Разделу 7.10.3 [Определение Обработчика Вывода].

Вы можете также установить функцию, которая возвращает информацию относительно числа и типа аргументов, ожидаемых спецификатором формата преобразования. См. Раздел 7.9.10 [Синтаксический анализ Строки Шаблона], для получения дополнительной информации об этом.

Средства этого раздела объявлены в файле 'printf.h'.

Примечание о Переносимости: возможность изменения синтаксиса printf строк шаблона - расширение GNU. Стандарт ANSI C не имеет ничего подобного.

7.10.1 Указание Новых Форматов Вывода

Функция для регистрирования новых преобразований вывода - register_printf_function, объявлена в `printf.h'.

- Функция: int register_printf_function(int SPEC,printf_function HANDLER_FUNCTION, printf_arginfo_function ARGINFO_FUNCTION)

Эта функция определяет символ спецификатора преобразования SPEC. Так, если SPEC равен 'q', то определяется модификатор '%q'.

HANDLER_FUNCTION - функция, вызываемая printf, когда это модификатор появляется в строке шаблона. См. Раздел 7.10.3 [Определение Обработчика Вывода], для уточнения информации относительно того, как определить функцию в качестве этого аргумента. Если Вы задаете пустой указатель, существующая функция обработчика для спецификаций удаляется.

Arginfo_function - функция, вызываемая parse_printf_format, когда

это преобразование появляется в строке шаблона. См. Раздел 7.9.10 [Синтаксический анализ Строки Шаблона]. Обычно Вы устанавливаете обе функции обработки вывода одновременно, но если Вы никогда не обращаетесь к parse_printf_format, Вы не должны определять функцию arginfo_function.

Возвращаемое значение - 0 в случае успеха, и -1 при сбое (который происходит, если спецификации находятся вне диапазона).

Вы можете переопределять форматы стандартного вывода, но это - возможно не лучшая идея из-за потенциального путаницы. Если Вы это сделаете, могут пострадать библиотечные подпрограммы, написанные другими людьми.

7.10.2 Ключи Спецификатора Преобразования

Если Вы определяете какое-либо значение для '%q',то что произойдет, если шаблон содержит '%+23q' или '%-#q'? Чтобы реализовать обработку этого, обработчик должен быть способен получить ключи, определенные в шаблоне.

Оба аргумента функции register_printf_function - HANDLER_FUNCTION и ARGINFO_FUNCTION получает в качестве аргумента типа struct printf_info, который содержит информацию относительно ключей, появляющихся в образце спецификатора формата вывода. Этот тип данных объявлен в заглавном файле 'printf.h'.

- Тип: struct printf_info

Эта структура используется для передачи информации о ключах, появляющихся в образце спецификатора формата вывода в printf строке шаблона в обработчик и функции arginfo для их спецификатора. Она содержит следующие элементы:

'int prec'

Эта переменная содержит задаваемую точность. Значение -1, если

никакая точность не была определена. Если, точность была задана как '*', структура printf_info, переданная в функцию обработчика, содержит фактическое значение, взятое из списка параметров. Но структура, переданная в функция arginfo содержит значение INT_MIN, так как фактическое значение не известно.

'int width'
Эта переменная содержит задаваемую минимальную ширину поля вывода. Значение 0, если ширина не была определена. Если, ширина поля была задана как '*', структура printf_info, переданная в функцию обработчика, содержит фактическое значение, взятое из списка параметров. Но структура, переданная в функция arginfo содержит значение INT_MIN, так как фактическое значение не известно.
'char spec'
Эта переменная содержит заданный символ спецификатора формата вывода. Он содержится в структуре для того, чтобы Вы могли указать одну и ту же функцию обработчика для различных символов, но при этом иметь возможность их различать при вызове функции обработчика.
'unsigned int is_long_double'
Это - логическая переменная, которая содержит значение истина, если модификатор типа 'L' был определен.
'unsigned int is_short'
Это - логическая переменная, которая содержит значение истина, если модификатор типа 'h' был определен.
'unsigned int is_long'
Это - логическая переменная, которая содержит значение истина, если модификатор типа 'l' был определен.
'unsigned int alt'
Это - логическая переменная, которая содержит значение истина, если был определен флаг '#'.
'unsigned int space'
Это - логическая переменная, которая содержит значение истина, если был определен флаг ' '.
'unsigned int left'
Это - логическая переменная, которая содержит значение истина, если был определен флаг '-'.
'unsigned int showsign'
Это - логическая переменная, которая содержит значение истина, если был определен флаг '+'.
'char pad'
Это - символ, использующийся для дополнения вывода в минимальную ширину поля. Значение - '0' если был определен

определен флаг '0' , иначе ' '.

7.10.3 Определение Обработчика Вывода

Теперь рассмотрим, как определить функцию обработчика и функции arginfo, которые передаются как аргументы для register_printf_function.

Вы должны определить ваши функции обработчика с прототипом следующим образом:

```
int function (FILE *stream, const struct printf_info *info, va_list
*ap_pointer)
```

Аргумент stream, переданный функции обработчика - это поток, в который она должна записать вывод.
Аргумент info - указатель на структуру, которая содержит информацию относительно различных ключей, которые были включены в строку шаблона. Вы не должны изменять эту структуру внутри вашей функции обработчика. См. Раздел 7.10.2 [Опции Спецификатора Преобразования], для описания этой структуры данных.
Аргумент ap_pointer используется для передачи хвоста списка параметров, содержащего значения котторые Ваш обработчик должен напечатать. В отличие от большинства других функций, которым может быть передан явный список параметров, здесь передается указатель на va_list,

а не сам va_list. Таким образом, Вы должны обрабатывать аргументы с помощью va_arg(TYPE, * ap_pointer).
(Введение указателя здесь позволяет указать функцию, которая вызывает вашу функцию обработчика, чтобы модифицировать собственную переменную va_list, для обновления информации о параметрах, которые ваш обработчик обрабатывает. См. Раздел A. 2 [Variadic (функция)] 2.)
Ваша функция обработчика должна возвратить значение точно так же ка>

Transfer interrupted!

атить число символов, которое она
написала, или отрицательное значение, чтобы указать ошибку.

-Тип данных: printf_function
Это тип данных, который должна иметь функция обработчика. Если Вы собираетесь использовать parse_printf_format в вашем приложении,

то Вы должны также определить функцию, являющуюся параметром arginfo_function для каждого нового формата, который Вы устанавливаете с помощью register_printf_function.

Вот прототип подобной функции:
int function (const struct printf_info *info,size_t n,int *argtypes);

Функция должна возвращать число параметров, которые обрабатывает формат вывода. Кроме того, функция не должна заполнять больше, чем n элементов argtypes массива с информацией относительно типов каждого из этих параметров. Эта информация закодирована с помощью 'PA_' макрокоманд. (Обратите внимание, что это - то же самое что и соглашение о вызовах parse_printf_format.)

-Тип данных: printf_arginfo_function
Этот тип используется, для описания функций, которые возвращают информацию о числе и типе параметров, используемых спецификатором формата вывода.

7.10.4 Пример Расширения Printf

Вот пример, показывающий, как определять printf функцию обработчика. Эта программа определяет структуру данных называемую Widget и определяет формат '%M', для печати информации о параметре Widget* включая значение указателя и имя, содержащееся в структуре данных. Формат вывода '%W' поддерживает минимальную ширину поля и опции левого выравнивания, но игнорирует все остальные.

```
#include
#include
#include
typedef struct
  {
    char *name;
  } Widget;
```



```
int
print_widget (FILE *stream, const struct printf_info *info, va_list
*app)
{
  Widget *w;
  char *buffer;
  int len;

  /* Преобразуем выходную информацию в строку. */
  w = va_arg (*app, Widget *);
  len = asprintf (&buffer, '', w, w->name);
  if (len == -1)
```

```
          return -1;
        /* Заполняем поле минимальной длины и выводим в поток.*/
        len = fprintf (stream, '%*s',
                       (info->left ? - info->width : info->width),
                       buffer);

        /* Сброс и возврат. */
        free (buffer);
        return len;
      }


      int
      main (void)
      {

         / * Создаем widget, который необходимо напечатать. * /
         Widget mywidget;
         mywidget.name = 'mywidget';

         / * Теперь печатаем widget. * /
         printf ('|%W|\n',  &mywidget);
         printf ('|%35W|\n',  &mywidget);
         printf ('|%-35W|\n', &mywidget);
         return 0;
}
```


Программа выводит:

```
     ||
     |      |
     |      |
```

7.11 Форматируемый Ввод

Функции, описанные в этом разделе (scanf и ей подобные) обеспечивают средства для форматируемого ввода, аналогичного форматируемым средствам вывода. Эти функции обеспечивают механизм для чтения произвольных значений при контроле над строкой формата или строкой шаблона.

7.11.1 Основы Форматируемого Ввода

Вызов scanf на первый взгляд подобен обращениям к printf, в котором произвольные параметры считываются под управлением строки шаблона. В то время, как синтаксис спецификаций преобразования в шаблоне очень схож, синтаксис для printf, интерпретация шаблона ориентируется больше на ввод свободного формата и простое сопоставление с образцом, а не форматирование устанавливаемого поля. Например, большинство scanf пропускают любое преобразование над каким-нибудь количеством 'пробельных символов' (включая пробел, метки табуляции, и символы перевода строки) во входном файле, и нет никакого понятия точности для числовых входных преобразований которое имеется для соответствующих преобразований вывода. Обычно, непробельные символы в шаблоне, как ожидается, будут точно соответствовать символам во входном потоке, но соответствующая ошибка отличается от входной ошибки в потоке.

Другая отличие между scanf и printf - то, что Вы должны не забыть обеспечивать указатели а не непосредственные значения как необязательные параметры scanf; значения, которые читаются, сохраняются в объектах, на которые указывают указатели. Даже опытные программисты имеют тенденцию забывать это иногда, так если ваша программа получает странные ошибки, которые, кажется, связаны со scanf, Вам следует дважды проверить это.

Когда происходит ошибка несоответствия , scanf немедленно прерывается, оставляя первый символ несоответствия как следующий символ, который нужно читать из потока. Нормальное возвращаемое значение из scanf - число значений, которые были назначены, так что Вы можете использовать это, чтобы определить, случалась ли ошибка соответствия прежде, чем все прочитались ожидаемые значения.

Функция scanf обычно используется для программ вроде считывания содержания таблиц. Ниже приводится функция, которая использует scanf для того, чтобы инициализировать массив элементов double: void readarray (double *array, int n) {

```
        int i; for (i=0; i scanf ('%a[a-zA-Z0-9] = %a[^\n]\n',
```

```
                    &variable, &value))
        {
          invalid_input_error ();
          return 0;
        }

      ...
    }
```

7.11.7 Другие Входные Форматы

Этот раздел описывает разнообразные входные форматы.

Формат '%p' используется, для того чтобы считывать значение указателя. Оно распознает тот же самый синтаксис, как и формат вывода '%p' для printf (см. Раздел 7.9.6 [Другие Преобразования Вывода]). Соответствующий параметр должен иметь тип void **; то есть адрес места, где разместить указатель.

Формат '%n' возвращает число прочитанных символов. Соответствующий

параметр должен иметь тип int *. Это преобразование работает таким же образом как и формат '%n' для printf; см. примеры в Разделе 7.9.6 [Другие Форматы Вывода].

Формат '%n' - единственный механизм для определения успеха буквального соответствия или преобразования с подавляемыми назначениями. Если '%n' следует за ошибкой несоответствия, scanf возвратится, не успев обработать '%n'. Если Вы поместите -1 в этот параметр перед вызовом scanf, присутствие -1 после вызова scanf указывает, что ошибка произошла перед обработкой '%n'.

В заключение, формат '%%' соответствует символу `%' во входном потоке, без использования параметра. Это преобразование не позволяет определение никаких флагов, ширины поля, или модификаторов типа.

7.11.8 Форматируемые Входные Функции

Ниже приводятся описания функций для выполнения форматируемого ввода. Прототипы для этих функций находятся в файле 'stdio.h'.

-Функция: int scanf (const char *template, ...)
Функция scanf читает форматируемый ввод из потока stdin под управлением строки шаблона. Необязательные параметры - указатели на места, которые получают возникающие в результате значения.

Возвращаемое значение - обычно число успешных соответствий. Если условие конца файла обнаружено перед любым соответствием (включая соответствие пробельным символам и литеральных символов в шаблоне), то возвращается EOF.

-Функция: int fscanf (FILE *stream, const char *template, ...)
Эта функция - аналог scanf, за исключением того, что ввод осуществляется из вместо stdin указанного потока.

-Функция: int sscanf (const char *s, const char *template, ...)
Подобна scanf, за исключением того, что символы берутся из строки s с нулевым символом в конце, а не из потока. Достижение конца строки обрабатывается как условие конца файла.

Поведение этой функции неопределено, если копирование происходит между объектами, которые пересекаются, например, если s задан еще и как аргумент для получения считанной строки под управлением формата '%s'.

7.11.9 Функции Ввода С Переменными Аргументами

Функции vscanf и ей подобные работают так, чтобы Вы могли определять ваши собственные scanf-подобные функции, которые используют ту же самую внутреннюю организацию как встроенные форматируемые функции вывода. См. Раздел 7.9.9 [Вывод Аргументов Переменной].

Примечание о Переносимости: функции, перечисленные в этом разделе являются расширением GNU.

-Функция: int vscanf (const char *template, va_list ap)
Эта функция похожа на scanf за исключением того, что вместо того, чтобы принимать переменное число аргументов непосредственно, она берет указатель на список параметров - ap типа va_list (см. Раздел A.2

Variadic Функции]).

-Функция: int vfscanf (FILE *stream,const char *template,va_list ap)
Эта функция - эквивалент fscanf с переменным списком параметров, заданным непосредственно как для vscanf.

7.12 Блочный Ввод-Вывод

Этот раздел описывает операции ввода и вывода на блоках данных. Вы можете использовать эти функции для чтения и записи двоичных данных, также как читать и писать текст блоками устанавливаемого размера - а не символами или строками.

Сохранение данных в двоичной форме часто значительно более эффективно чем использование форматируемых функций ввода - вывода. Также, для чисел с плавающей запятой, двоичная форма избегает возможной потери точности в процессе преобразования. С другой стороны, двоичные файлы не могут быть легко исследованы или изменяться, используя много стандартных файловых утилит (вроде текстовых редакторов), и не переносимы между различными реализациями языка, или различными видами компьютеров.

Эти функции объявлены в ' stdio.h '.

-Функция:size_t fread (void *data, size_t size, size_t count, FILE *stream)

Эта функция читает до count объектов размера size в массив. Она возвращает число прочитанных объектов, которое может быть меньше чем

count, если происходит ошибка чтения, или достигнут конец файла. Эта функция возвращает значение нуль (и ничего не читает) если или size или count равен нулю.

Если fread достигает конца файла в середине объекта, она возвращает номер прочитанных полностью объектов, и отбрасывает несчитанные до конца.

-Функция: size_t fwrite (const void *data, size_t size, size_t count, FILE *stream)

Эта функция записывает до count объектов из массива данных в указанный поток. Возвращаемое значение - обычно count, если считывание успешно. Любой другое значение указывает какую-либо ошибку, например нехватку памяти.

7.13 КОНЕЦ ФАЙЛА и Ошибки

Многие из функций, описанных в этой главе возвращают значение макрокоманды EOF, указывающей неудачное завершение операции. Когда используется EOF, для сообщения о конце файла или о случайной ошибке, часто лучше использовать функцию feof, чтобы явно проверить конец файла и ferror, чтобы проверить наличие ошибки. Это контрольные индикаторные функции, которые являются частью внутреннего состояния объекта потока, индикаторы устанавливаются если соответствующее условие было обнаружено предыдущей операцией ввода - вывода на этом потоке.

Эти символы объявлены в заглавном файле ' stdio.h '.

- Макрос: int EOF
Этот макрос имеет целое значение, которое возвращается рядом функций, чтобы указать условие конца файла, или какую-нибудь другую ошибку. В библиотеке GNU, EOF имеет значение -1. В других библиотеках, значением может быть некоторое другое отрицательное число.

-Функция: void clearerr (FILE *stream)
Эта функция очищает индикаторы конца файла и ошибки для указанного потока. Позиционирующие файл функции (см. Раздел 7.15 [Позиционирование Файла]) также, очищают индикатор конца файла для потока.

-Функция: int feof (FILE *stream)
Функция feof возвращает отличное от нуля число, только если установлен индикатор конца файла для потока.

-Функция: int ferror (FILE *stream)
Функция ferror возвращает отличное от нуля число, только если индикатор ошибки для потока установлен, указывая что ошибка

произошла на предыдущей операции на потоке.

В дополнение к установке индикатора ошибки, связанного с потоком, функции, которые работают с потоками, устанавливают errno таким же образом как соответствующие функции низкого уровня, которые работают с дескрипторами файла. Например, все функции, которые выполняют вывод в поток, такие как fputc, printf, и fflush, осуществлены в терминах записи, и все условия ошибки errno, определенные для записи, имеют значение и для этих функций. Для получения более подробной информации о функциях ввода - вывода на уровне дескрипторов, см. Главу 8 [ввод - вывод низкого уровня] 3.

7.14 Текстовые и Двоичные Потоки

Система GNU и другие posix-совместимые операционные системы организовывает все файлы как однородные последовательности символов. Однако, некоторые другие системы делают различие между файлами, содержащими текст и файлами, содержащими двоичные данные, и средства ввода и вывода ANSI C предусматривают это различие. Этот раздел сообщает Вам, как написать программы, переносимые на такие системы.

Когда Вы открываете поток, Вы можете определять или текстовый поток или двоичный поток. Вы указываете, что Вы хотите двоичный поток, определяя модификатор 'b' в параметре opentype для fopen; см. Раздел 7.3 [Открытие Потоков]. Без этой опции, fopen открывает файл как текстовый поток.

Текстовые и двоичные потоки имеют различия:

* Данные из текстового потока разделены на строки, которые завершены символом перевода строки (' \n ') символы, в то время как двоичный поток - просто ряд символов. Текстовый поток может на некоторых системах быть не в состоянии обрабатывать строки больше чем 254 символа (включая символ перевода строки).

* На некоторых системах, текстовые файлы могут содержать только символы печати, горизонтальные символы табуляции, и символы перевода строки, и так что текстовые потоки не могут поддерживать другие символы. Однако, двоичные потоки могут обрабатывать любое символьное значение.
* Символы пробела, которые написаны перед символом перевода строки в текстовом потоке, могут исчезать.
* В более общем смысле, не обязательно имеется взаимнооднозначное отображение между символами, которые считываются или записываются в текстовый поток, и символами в фактическом файле.

Так как двоичный поток более общий и более предсказуемый чем текстовый поток, Вы могли бы задаться вопросом, в чем цель текстовых потоков. Почему не всегда используют двоичные потоки? Ответ в том, что на разных операционных системах, текстовые и двоичные потоки используют различные форматы файла, и единственный способ читать или записать 'обычный текстовый файл' который может работать с другими ориентируемыми текстом программами - через текстовый поток.

В библиотеке GNU, и на всех POSIX системах, не имеется никакого различия между текстовыми потоками и двоичными потоками. Когда Вы открываете поток, Вы получаете тот же самый вид потока, даже если Вы заказывали двоичный. Этот поток может обрабатывать любое содержание файла, и не имеет ни каких ограничений, которые в отличие от текстовые потоков.

7.15 Позиционирование Файла

Позиция файла потока описывает, где в файле поток в настоящее время читает или производит запись. Ввод-вывод на потоке продвигает позицию файла через весь файл. В системе GNU, позиция файла представляется как целое число, которое содержит число байтов от начала файла. См. Раздел 6.1.2 [Позиция Файла].

В течение ввода-вывода в обычный дисковый файл, Вы можете менять позицию в файле всякий раз, когда Вы желаете читать или записывать в любую часть файла. Некоторые другие виды файлов также позволяют делать это. Файлы, которые поддерживают изменение позиции файла иногда упоминается как файлы прямого доступа.

Вы можете использовать функции в этом разделе, чтобы исследовать или изменить индикатор позиции файла, связанный с потоком. Символы, перечисленные ниже объявлены в заглавном файле 'stdio.h'.

-Функция: long int ftell (FILE *stream)
Эта функция возвращает текущую позицию файла указанного потока. Эта функция может выдать ошибку, если поток не поддерживает позиционирование файла, или если позиция файла не может представляться как long int, или возможно по другим причинам.
Если происходит ошибка, возвращаемое значение -1.
-Функция: int fseek (FILE *stream, long int offset, int whence)
Функция fseek используется для изменения позиции файла указанного потока. Значение whence должно быть одной из констант SEEK_SET, SEEK_CUR, или SEEK_END, т. е. указывать является ли смещение относительно начала файла, текущей позиции файла, или конца файла, соответственно. Эта функция возвратит нуль, если операция была успешна, и значение, отличное от нуля чтобы указать отказ. Успешное обращение также очищает индикатор конца файла потока и отбрасывает любые символы, которые были 'помещены обратно' использованием ungetc. Fseek дописывает любой буферизированный вывод перед позиционированием файла, или еще запоминает его, так что он будет записан позже в соответствующем месте файла.

Примечание о Переносимости: В не-posix системах, ftell и fseek могут работать надежно только на двоичных потоках. См. Раздел 7.14 [Двоичные Потоки].
Следующие символические константы определены для использования в качестве аргумента whenceа для fseek. Они также используются функцией lseek (см. Раздел 8.2 [Примитивы ввода - вывода]) и для указания смещения для блокировок файла (см. Раздел 8.7 [Операции Управления]).

-Макрос: int SEEK_SET
Это целая константа которая, когда используется как аргумент whence функции fseek и определяет, что смещение указывается относительно начала файла.

-Макрос: int SEEK_CUR
Это целая константа которая используется как аргумент whence функции fseek и определяет, что смещение указывается относительно текущей позиции файла.

-Макрос: int SEEK_END
Это целая константа которая используется как аргумент whence функции fseek и определяет, что смещение указывается относительно конца файла.

-Функция: void rewind (FILE *stream)
Функция rewind позиционирует указанный поток в начало файла. Это эквивалентно вызову fseek на потоке с аргументом смещения 0L и аргументом whence SEEK_SET, за исключением того, что возвращаемое значение отбрасывается, и индикатор ошибки для потока сброшен.
Эти три побочных результата исследования для констант 'SEEK_...' существуют ради совместимости с более старыми BSD системами. Они определены в двух различных файлах: 'fcntl.h' и 'sys/file.h'.
L_SET синоним SEEK_SET.
L_INCR синоним SEEK_CUR.
L_XTND синоним SEEK_END.

7.16 Переносимые Функции позиционирования файла

В системе GNU, позиция файла - просто символьный счетчик. Вы можете задавать любое значение count как аргумента в fseek и получать надежные результаты для любого файла произвольного доступа. Однако, некоторые ANSI C системы не представляет позиции файла таким образом,.
На некоторых системах, где текстовые потоки отличаются от двоичных потоков невозможно представить позицию файла текстового потока как счетчик символов от начала файла. Например, позиция файла на некоторых системах должна кодировать, и смещение записи внутри файла, и смещение символа внутри записи.
Как следствие, если Вы хотите чтобы ваши программы были переносимы на эти системы, Вы должны соблюдать некоторые правила:
* Значение, возвращенное ftell на текстовом потоке не имеет никакой предсказуемостной связи с числом символов, которое Вы читали из пока. Единственная вещь, на которую Вы можете полагаться - то, что Вы сможете использовать его впоследствии, поскольку аргумент смещения в fseek двигается обратно в ту же самую позицию файла.

* При обращении к fseek на текстовом потоке, смещение должно быть либо нуль; либо SEEK_SET, и смещение должно быть результатом более раннего обращения к ftell на том же самом потоке.
* Значение индикатора позиции файла текстового потока неопределено, если имеются символы, которые были помещены обратно

функцией ungetc, которые не читались или не отбрасывались. См. Раздел 7.8 [Обратное чтение].

Но даже если Вы соблюдаете эти правила, Вы можете все еще иметь проблемы длинными файлами, т. к. ftell и fseek используют значение int long, для представления позицию файла. Этот тип может не иметь участка памяти, для кодирования всех позиций файла в большом файле.

Так, если Вы хотите поддерживать системы со специфическими кодированием для позиций файла, то лучше использовать функции fgetpos и fsetpos. Эти функции представляют позицию файла, используя тип данных fpos_t, чье внутреннее представление меняется от системы к системе.
Эти символы объявлены в заглавном файле ' stdio.h '.
-Тип данных: fpos_t
Это - тип объекта, который может кодировать информацию относительно файловой позиции потока, для использования функциями fgetpos и fsetpos.

В системе GNU, fpos_t эквивалентен off_t или long int. В других системах, он может иметь различное внутреннее представление.

-Функция: int fgetpos (FILE *stream, fpos_t *position)
Эта функция сохраняет значение индикатора файловой позиции для указанного потока в указанном объекте fpos_t. Обращение успешно если fgetpos возвращает нуль; иначе она возвращает значение отличное от нуля и сохраняет определенное реализацией положительное значение в errno.

-Функция: int fsetpos (FILE *stream, const fpos_t position)
Эта функция устанавливает индикатор файловой позиции для указанного потока в позицию position, которая должна определяться предыдущим обращением к fgetpos на том же самом потоке. Если обращение успешно, fsetpos очищает индикатор конца файла на потоке, отбрасывает любые символы, которые были 'помещены обратно' использованием ungetc, и возвращает значение нуля. Иначе, fsetpos возвращает значение отличное от нуля и сохраняет определенное реализацией положительное значение в errno.

7.17 Буферизация Потока

Символы, которые записаны в поток, обычно накапливаются и передаются в файл блоками асинхронно, вместо того, чтобы появляться, как только они выводятся прикладной программой. Аналогично, потоки часто восстанавливают ввод из главной среды в блоках а не по принципу символ-за-символ. Это называется буферизацией.

Если Вы напишете программы, которые делают интерактивный ввод и вывод используя потоки, Вы должны знать, как работает буферизация, когда Вы разрабатываете интерфейс пользователя в вашей программе. Иначе, вывод (типа подсказки) может не появиться, как ожидалось, и т. д.

Этот раздел имеет дело только с управлением передачей символов между потоком и файлом или устройством.

Вы можете обходить средства буферизации потока в целом(вполне), используя ввод и вывод низкого уровня, которые функционируют на описателях файла. См. Главу 8 [ввод - вывод низкого уровня].

7.17.1 Понятие Буферизации

Имеются три различных вида стратегий буферизации:

* Символы, записываемые или читаемые из небуферизованного потока, передаются индивидуально в файл или из файла как можно скорее.
* Символы, записываемые или читаемые из строчно буферизированного потока передаются в или из файла в блоках, когда прочитан с символ перевода строки.
* Символы, записываемые или читаемые из полностью буферизированного

потока, передаются в или из файла в блоках произвольного размера.
Вновь открываемые потоки обычно полностью буферизируются, с одним исключением: поток, соединенный с интерактивным устройством типа терминала - изначально буферизирован строчно. См. Раздел 7.17.3 [Управление Буферизацией], для уточнения информации о том, как в выбирают различные виды буферизации.
Использование строчной буферизации для интерактивных устройств подразумевает окончание вывода сообщения с символом перевода строки. Вывод, который не заканчивается на символе перевода строки, может и не быть обнаружен немедленно, так если Вы хотите вывести его немедленно, Вы должны очистить буферизированный вывод функцией fflush, как описано в Разделе 7.17.2 [Очистка буфера].

Буферизация строки - хорошее значение по умолчанию для ввода терминала, т. к. большинство интерактивных программ читают команды, которые являются обычно одиночными строками. Программа должна быть способна выполнять каждую строку сразу же. Буферизированный ввод также согласуется с обычными редактирующими вводом средствами большинства операционных систем, которые работают внутри строки ввода.
Они включают программы, которые читают одиночно - символьные команды (подобно Emacs) и программам, которые делают их собственное редактирование ввода (типа тех что используют readline). Чтобы просто читать символ, не достаточно выключить буферизацию во входном потоке; Вы должны также выключить редактирование ввода, в операционной системе. Это требует изменения режима терминала (см. Раздел 12.4 [Режимы Терминала]).

7.17.2 Промывание Буфера

Сброс вывода на буферизированном потоке, передает все накопленные символы в файл. Имеются много обстоятельств когда буферизированный вывод на потоке, сбрасывается автоматически:

* Когда Вы пробуете выводить, и буфер вывода полон.
* Когда поток закрыт. См. Раздел 7.4 [Закрывание Потоков].
* Когда программа завершается, вызывая exit. См. Раздел 22.3.1 [Нормальное Окончание].
* Когда введен символ перевода строки, если поток буферизирован строчно.
* Всякий раз, когда операция ввода на любом потоке фактически читает данные из файла.

Если Вы хотите сбросить буферизированный вывод в другой момент, вызывайте fflush, которая объявлен в заглавном файле ' stdio.h '.

-Функция: int fflush (FILE *stream)
Эта функция заставляет любой буферизированный вывод на потоке дописываться в файл. Если поток - нулевой указатель, то буферизированный вывод на всех открытых выходных потоках будет сброшен.
Эти функции возвращают EOF, если происходит ошибка записи, и нуль в другом случае.

Примечание о Совместимости: Некоторые поврежденные в уме операционные системы, как известно, были настолько помешаны на строчно ориентированном вводе, что для сброса строчно буферизированного потока должен быть введен символ перевода строки! К счастью, эта 'удобство', кажется, становится менее распространенным.
В системе GNU беспокоиться об этом Вам нет нужды.

7.17.3 Управление Видом Буферизации

После открытия потока (но прежде любой другой операции на нем), Вы можете явно определить какую буферизацию Вы хотите, используя функцию setvbuf.
Средства, перечисленные в этом разделе объявлены в файле 'stdio.h'.
-Функция: int setvbuf (FILE *stream, char *buf, int mode, size_t size)
Эта функция используется, чтобы определить, что указанный поток должен иметь заданный режим буферизации, который может быть: _IOFBF (для полной буферизации), _IOLBF (для буферизации строки), или _IONBF (для небуферизованного ввода -вывода).
Если Вы определяете нулевой указатель как параметр buf, то

setvbuf, распределяет буфер, непосредственно используя malloc. Этот буфер будет освобожден, когда Вы закроете поток.

Иначе, buf должен быть символьным массивом, который может содержать по крайней мере size символов. Вы не должны трогать пространство для этого массива, пока поток остается открытым и этот массив остается буфером. Использование автоматического массива - не очень хорошая идея, если Вы не закрываете файл перед выходом из блока, который объявляет массив.

В то время как массив остается буфером потоков, функции ввода - вывода потока используют буфер для их внутренних целей. Вы не должны пробовать обращаться к значениям в массиве непосредственно, в то время как поток использует его для буферизации.

Функция setvbuf возвращает ноль в случае успеха, или значение отличное от нуля, если значение режима не допустимо или если запрос не мог быть удовлетворен.

-Макрос: int _IOFBF
Значение этой макрокоманды - константа integer, которая может использоваться как параметр режима для функции setvbuf, чтобы определить, что поток должен быть полностью буферизирован.

-Макрос: int _IOLBF
Значение этой макрокоманды - константа integer, которая может использоваться как параметр режима для функции setvbuf, чтобы определить, что поток должен быть буферизирован строчно.

-Макрос: int _IONBF
Значение этой макрокоманды - константа integer, которая может использоваться как параметр режима для функции setvbuf, чтобы определить, что поток должен быть небуферизован.

-Макрос: int BUFSIZ
Значение этой макрокоманды - константа integer, которую удобно использовать как параметр size для setvbuf. Это значение, как гарантируют, будет по крайней мере 256.

Значение BUFSIZ выбрано в каждой системе таким, чтобы делать ввод - вывод потока наиболее эффективным.

Фактически, Вы можете лучшее значение, чтобы использовать для размера буфера посредством fstat системного вызова: его можно найти в st_blksize поле атрибутов файла. См. Раздел 9.8.1 [Значения Атрибута].

Иногда также используют BUFSIZ как размер распределения буферов, используемых для соответствующих целей, типа строк, используемых, чтобы получить строку ввода через fgets (см. Раздел 7.6 [Символьный Ввод]). Не имеется никакой специфической причины использовать BUFSIZ для этого вместо любого другого integer, за исключением того, что это могло бы привести к выполнению ввода - вывода в кусках эффективного размера.

-Функция: void setbuf (FILE *stream, char *buf)
Если buf - нулевой указатель, эффект этой функции эквивалентен вызову setvbuf с параметром режима _IONBF. Иначе, это эквивалентно вызову setvbuf с buf, и режимом _IOFBF и параметром size - BUFSIZ.
Функция setbuf предусмотрена для совместимости со старым кодом; используйте setvbuf во всех новых программах.

-Функция: void setbuffer (FILE *stream, char *buf, size_t size)
Если buf - нулевой указатель, эта функция делает поток небуферизованным. Иначе, это делает поток полностью буферизированным с использованием buf в качестве буфера. Параметр size определяет длину buf.

Эта функция предусмотрена для совместимости со старым BSD кодом. Используйте вместо нее setvbuf.

-Функция: void setlinebuf (FILE *stream)
Эта функция делает поток буферизированным строчно, и распределяет буфер для Вас. Эта функция предусмотрена совместимость со старым BSD кодом. Используйте setvbuf вместо нее.

7.18 Другие Виды Потоков

Библиотека GNU обеспечивает способы определить дополнительные виды потоков, которые не обязательно соответствуют открытому файлу.

Один такой тип потока берет ввод из или пишет вывод в строку. Эти виды потоков используются внутренне, чтобы выполнить функции sprintf и sscanf. Вы можете также создавать такой поток явно, при использовании функций, описанных в Разделе 7.18.1 [Строковые Потоки].

В более общем смысле, Вы можете определять потоки, которые делают ввод -вывод для произвольных объектов, используя функции, обеспеченные вашей программой. Этот протокол обсужден в Разделе 7.18.3 [Заказные Потоки] .

Примечание о Переносимости: средства, описанные в этом разделе специфические для GNU. Другие системы или реализации C могли и не обеспечивать эквивалентные функциональные возможности.

7.18.1 Строковые Потоки

Функции fmemopen и open_memstream делали ввод - вывод в буфер памяти или строку. Эти средства объявлены в 'stdio.h '.

-Функция: FILE * fmemopen (void *buf, size_t size, const char *opentype)

Эта функция открывает поток, который допускает доступ, определенный параметром opentype, и который считывается или записывается в буфер, определенный параметром buf. Этот массив должен быть по крайней мере size байтов длиной.

Если Вы определяете нулевой указатель как параметр buf, fmemopen

динамически распределяет (как с malloc; см. Раздел 3.3 [Беспрепятственное Распределение]) size байтовый массив. Это действительно полезно только, если Вы собираетесь записывать что-то в буфер и затем читать это обратно, т. к. Вы не имеете никакого способа фактически получить указатель на буфер (для этого, попробуете open_memstream, ниже). Буфер освобождается, когда открывается поток. Параметр opentype - такой же как в fopen (См. Раздел 7.3 [Открытие Потоков]). Если opentype определяет режим конкатенирования, то начальная файловая позиция устанавливается на первый символ пробела в буфере. Иначе начальная файловая позиция - в начале буфера.

Для потока, открытого для чтения, пустые символы (нулевые байты) в буфере не считаются концом файла. Операции чтения возвращают конец файла только когда файловая позиция продвигается за size байт. Так, если Вы хотите читать символы из строки с нулевым символом в конце, Вы должны обеспечить длину строки как аргумент size.

Вот пример использования fmemopen для создания потока для чтения из строки:

```
#include

static char buffer[] = 'foobar';

int
main (void)
{
  int ch;
  FILE *stream;

  stream = fmemopen (buffer, strlen (buffer), 'r');
  while ((ch = fgetc (stream)) != EOF)
    printf ('Got %c\n', ch);
  fclose (stream);

  return 0;
}
```


Эта программа производит следующий вывод:

```
Got f
Got o
Got o
```

```
     Got b
     Got a
     Got r
```

-Функция: FILE * open_memstream (char **ptr, size_t *sizeloc)
Эта функция открывает поток для записи в буфер. Буфер размещен динамически (как с malloc; см. Раздел 3.3 [Беспрепятственное Резервирование]) и растет по мере необходимости.

Когда поток закрывается с помощью fclose или сбрасывается с помощью fflush, указатели ptr и sizeloc модифицируются, и содержат указатель на буфер и size. Значения, таким образом сохраненные остаются допустимыми только, пока не происходит никакой дальнейший вывод на потоке. Если, Вы выводите еще, Вы должны промыть поток, чтобы сохранить новые значения прежде, чем Вы используете его снова.

Пустой символ записывается в конце буфера. Этот пустой символ не включен в значение size, сохраненное в sizeloc.

Вы можете перемещать файловую позицию потока функцией fseek (см. Раздел 7.15 [Позиционирование Файла]). Перемещение файловой позиции после конца уже записанных данных, заполняет захваченное пространство нулями.

Вот пример использования open_memstream:

```
     #include

     int
     main (void)
     {
       char *bp;
       size_t size;
       FILE *stream;

       stream = open_memstream (&bp, &size);
       fprintf (stream, 'hello');
```



```
       fflush (stream);
       printf ('buf = `%s', size = %d\n', bp, size);
       fprintf (stream, ', world');
       fclose (stream);
       printf ('buf = `%s', size = %d\n', bp, size);

       return 0;
     }
```

Эта программа производит следующий вывод:

```
     buf = `hello', size = 5
     buf = `hello, world', size = 12
```

7.18.2 Obstack Потоки

Вы можете открывать выходной поток, который помещает данные в obstack. См. Раздел 3.4 [Obstack].

-Функция: FILE * open_obstack_stream (struct obstack *obstack)

Эта функция открывает поток для записи данных в obstack. Она начинает объект в obstack и увеличивает его, при записывании данных (см. Раздел 3.4.6 [Возрастастающие Объекты]).

Вызов fflush на этом потоке модифицирует текущий размер объекта, в соответствии с количеством данных, которое было записано. После обращения к fflush, Вы можете исследовать объект.

Вы можете перемещать файловую позицию obstack потока функцией fseek (см. Раздел 7.15 [Позиционирование Файла]).

Чтобы сделать объект постоянным, модифицируйте obstack с fflush, и тогда используйте obstack_finish для завершения объекта и получения адреса. Следующая запись в поток, начинает новый объект в obstack.

Но как Вы узнаете, какой длины объект? Вы можете получать длину в байтах, вызывая obstack_object_size (см. Раздел 3.4.8 [Состояние Obstack]), или Вы можете пустым символом завершить объект, примерно так:

```
  obstack_1grow (obstack, 0);
```

Вот типичная функция, которая использует open_obstack_stream:

```
char *     make_message_string (const char *a, int b)
     {
       FILE *stream = open_obstack_stream (&message_obstack);
```

```
       output_task (stream);
       fprintf (stream, ': ');
       fprintf (stream, a, b);
       fprintf (stream, '\n');
       fclose (stream);
       obstack_1grow (&message_obstack, 0);
       return obstack_finish (&message_obstack);
     }
```

7.18.3 Программирование Ваших Собственных Потоков

Этот раздел описывает, как Вы можете создавать потоки, которые берут ввод из произвольного источника данных или пишут вывод в произвольный сток данных, программируемый Вами. Назовем эти потоки пользовательскими.

7.18.3.1 Пользовательские Потоки и Cookies

Внутри каждого пользовательского потока есть специальный объект называемый cookie. Это - объект, определяемый Вами, который указывает, где берутся или сохраняются данные.

Чтобы реализовать пользовательский поток, Вы должны указать, как выбирать или сохранять данные в заданном месте. Это делается определением функции ловушки для чтения, записи, изменения 'файловой позиции', и закрытия потока. Вся четыре из этих функций будут переданы cookie потока, так чтоони могут сообщать, где брать или сохранять данные. Библиотечные функции не знают что находится внутри cookie, но ваши функции будут знать.

Когда Вы создаете пользовательский поток, Вы должны задать указатель на cookie, а также четыре функции ловушки, содержащиеся в структуре типа struct cookie_io_functions.

Эти средства объявлены в ' stdio.h '.

-Тип даннаых: struct cookie_io_functions

Это - тип структуры, который содержит функции, которые определяют протокол связи между потоком и cookie. Он имеет следующие элементы:

cookie_read_function *read

Это функция, которая читает данные из cookie. Если значение является пустым указателем вместо функции, то операции чтения на потоке всегда возвращает EOF.

cookie_write_function *write

Это - функция, которая пишет данные в cookie. Если значение является пустым указателем вместо функции, то данные, записанные в поток отбрасываются.

cookie_seek_function *seek

Это функция, которая выполняет эквивалент позиционирования файла на cookie. Если значение является пустым указателем вместо функции, обращения к fseek на этом потоке, может искать расположения только внутри буфера; любая попытка искать снаружи буфера, возвратит ESPIPE ошибку.

cookie_close_function *close

Эта функция выполняет любую соответствующую сброс cookie при закрытии потока. Если значение является пустым указателем вместо функции, то при закрытии потока ничего не делается для закрытия cookie.

-Функция: FILE * fopencookie (void *cookie, const char *opentype, struct

Эта функция фактически создает поток для сообщения с cookie используя функции в io_functions аргументе. Opentype аргумент интерпретируется как для fopen; см. Раздел 7.3 [Открытие Потоков]. (Но отметьте что опция 'усечения при открытии' игнорируется.) Новый поток полностью буферизирован.

Функция fopencookie возвращает недавно созданный поток, или пустой указатель в случае ошибки.

7.18.3.2 Пользовательские Функции-Ловушки Потока

Имеется большое количество деталей определения четырех функций ловушки, которые необходимы для пользовательского потока.

Вы должны определить функцию для чтения данных из cookie как:

```
ssize_t reader (void *cookie, void *buffer, size_t size)
```

Это очень похоже на функцию read; см. Раздел 8.2 [Примитивы ввода - вывода]. Ваша функция должна передать до size байтов в буфер, и

возвращать число прочитанных байтов, или нуль указывая конец файла. Вы можете возвращать значение -1 для указания ошибки.

Вы должны определить функцию, для записи данных в cookie как:

ssize_t writer (void *cookie, const void *buffer, size_t size)
Это очень похоже на функцию write; см. Раздел 8.2 [Примитивы ввода - вывода]. Ваша функция должна передать до size байтов из буфера, и возвращать число записанных байтов.

Вы должны определить функцию, для выполнения операции позиционирования на cookie как:
int seeker (void *cookie, fpos_t *position, int whence)

Для этой функции, position и whence аргументы интерпретируются как для fgetpos; см. Раздел 7.16 [Переносимое Позиционирование]. В библиотеке GNU, fpos_t эквивалентен off_t или long int, и просто представляет число байтов от начала файла.
После выполнения операции установки, ваша функция должна сохранить возникающую в результате файловую позицию относительно начала файла в position. Ваша функция должна возвратить значение 0 в случае успеха, и -1 в случае ошибки.
Вы должны определить функцию, для операции очистки на cookie, при закрытии потока как:
int cleaner (void *cookie)
Ваша функция должна возвратить -1, указывая ошибку, или 0 иначе.

-Тип данных: cookie_read_function
Это - тип данных, который должна иметь функция чтения для пользовательского потока. Если Вы объявляете функцию как показано выше, это тип, который она будет иметь.

-Тип данных: cookie_write_function
Тип данных пишущей функции для пользовательского потока.

-Тип данных: cookie_seek_function
Тип данных функции инициализации для пользовательского потока.
-Тип данных: cookie_close_function
Тип данных функции close для заказного потока.

8. Ввод-Вывод низкого уровня

Эта глава описывает функции для выполнения операций ввод-вывода низкого уровня с помощью дескрипторов. Эти функции включают примитивы для функций ввода - вывода с более высоким уровнем, описанные в Главе 7 [Ввод - вывод на Потоках], также как функции для выполнения операций управления низкого уровня, для которых не имеется никаких эквивалентов на потоках.
Ввод-вывод на уровне потоков более гибок и обычно более удобен; поэтому, программисты используют функции дескрипторного уровня только при необходимости. Вот несколько ситуаций, в которых могут понабиться дескрипторы:
* Для чтения двоичных файлов в больших кусках.
* Для чтения всего файла в ядро перед его синтаксическим анализом.
* Для выполнения операций отличных от передачи данных, которая может быть выполнена только с дескриптором. (Вы можете использовать fileno, для получения дескриптора, соответствующего потоку.)
* Для передачи дескриптора в дочерний процесс. (Дочерний может создавать собственный поток, для использования наследуемого описателя, но не может наследовать поток непосредственно.)

8.1 Открытие и Закрытие Файлов

Этот раздел описывает примитивы для открытия и закрытия файлов,

используя дескрипторы файла. Функции creat и open объявлены в файле 'fcntl.h', в то время как close объявлена в 'unistd.h'.

-Функция: int open (const char *filename, int flags[, mode_t mode])
Функция open создает и возвращает новый описатель файла для указанного файла. Первоначально, индикатор файловой позиции для файла находится в начале файла. Аргумент mode используется только, когда файл создан.
Аргумент flags управляет тем, как файл будет открыт. Это - битовая маска; Вы создаете значение поразрядным ИЛИ соответствующих параметров (используя оператор `|' в C).

- 174 -

Аргумент flags должен включять точно одно из этих значений, для задания режима доступа к файлу:

O_RDONLY Открывает файл для чтения.
O_WRONLY Откроывает файл для записи.
O_RDWR Открывает файл, и для чтения и для записи.
Аргумент flags может также включать любую комбинацию этих флагов:
O_APPEND Если установлен, то все операции записи запишут данные в конец файла, расширяя его, независимо от текущей файловой позиции.
O_CREAT Если установлен, будет создан файл, если он еще не существует.
O_EXCL Если и O_CREAT и O_EXCL - установлены, то open выдает ошибку, если заданный файл уже существует.
O_NOCTTY Если filename имя терминала, не делайте его терминалом управления для процесса. См. Главу 24 [Управление заданиями], для уточнения информации относительно того, что означает терминал управления.
O_NONBLOCK Устанавливает режим неблокирования. Эта опция обычно полезна для специальных файлов типа FIFO (см. Главу 10 [Каналы и FIFO]) и устройств типа терминалов. Обычно, для этих файлов open блокируется, пока файл не 'готов'. Если O_NONBLOCK установлен, open возвращается немедленно.
O_NONBLOCK бит также воздействует на чтение и на запись: он разрешает им возвращаться немедленно с состоянием ошибки, если не имеется никакого доступного ввода, или если вывод не может быть записан.
O_TRUNC Если файл существует и открыт для записи, усекает это, до нулевой длины. Эта опция полезна только для регулярных файлов, а не специальных файлов типа каталогов или FIFO. Для получения более подробной информации об этих символических константах см. Раздел 8.10 [Флаги Состояния Файла].
Нормальное возвращаемое значение open - неотрицательный описатель файла типа integer. В случае ошибки возвращается значение -1. В дополнение к обычным синтаксическим ошибкам имени файла (см. Раздел 6.2.3 [Ошибки Имени файла]), следующие errno условия ошибки определены для этой функции:
EACCES
Файл существует, но не читаем/перезаписываем как запрошено аргументом flags.

- 175 -

EEXIST
И O_CREAT и O_EXCL - установлены, а именованный файл уже существует.
EINTR
Операция open была прервана сигналом.
См. Раздел 21.5 [Прерванные Примитивы] 3.
EISDIR
Аргумент flags задает доступ для записи, а файл - каталог.
EMFILE
Процесс имеет слишком много открытых файлов.
ENFILE
Вся система, или возможно файловая система, которая содержит каталог, не может поддерживать любые дополнительные файлы открытыми в настоящее время. (Эта проблема не может случаться в системе GNU.)
ENOENT

Именованный файл не существует, но O_CREAT не определен.
ENOSPC
Каталог или файловая система, которая содержала бы новый файл,не может быть расширена, т. к. там не осталось дискового пространства.
ENXIO
O_NONBLOCK и O_WRONLY установлены в аргументе flags, файл, именованный filename - FIFO (см. Главу 10 [Каналы и FIFO]), и никакой процесс имеет файл открытым для чтения.
EROFS
Файл постоянно находится в файловой системе только для чтения и любой из O_WRONLY, O_RDWR, O_CREAT, и O_TRUNC - установлены в аргументе flags.

Функция open - основной примитив для fopen и freopen функций, которые создают потоки.

-Функция: int creat (const char *filename, mode_t mode)
Эта функция устаревает. Обращение:
creat (filename, mode)
эквивалентно:
open (filename, O_WRONLY | O_CREAT | O_TRUNC, mode)

-Функция: int close (int filedes)
Функция закрывает дескриптор файла filedes. Закрытие файла имеет следующие последствия:

* Описатель файла освобожден.
* Любые блокировки записи, принадлежащие процессу на файле разблокирутся.
* Когда все описатели файла, связанные с каналом или в порядке поступления были закрыты, любые читаемые обратно данные отброшены.

Нормальное возвращаемое значение - 0; значение -1 возвращается в случае ошибки.
Следующие errno условия ошибки определены для этой функции:
EBADF
Filedes аргумент - не допустимый дескриптор файла.
EINTR
Обращение было прервано сигналом. См. Раздел 21.5 [Прерванные Примитивы]. Вот пример правильной обработки EINTR:

TEMP_FAILURE_RETRY (close (desc));
Для закрытия потока, вызовите
fclose (см. Раздел 7.4 [Закрытие Потоков]) вместо того, чтобы пробовать закрыть основной описатель файла.
Она дописывает любой буферизированный вывод и модифицирует объект потока указывая, что он закрыт.

8.2 Примитивы Ввода и Вывода
Этот раздел описывает функции для выполнения ввода и вывода с помощьюю дескрипторов файла: read, write, и lseek. Эти функции объявлены в файле ' unistd.h '.

-Тип данных: ssize_t
Этот тип данных используется для представления размеров блоков, которые могут быть прочитаны или записаны одиночной операцией. Он подобен size _t, но должен быть знаковым типом.

-Функция: ssize_t read (int filedes, void *buffer, size_t size)

Функция read читает до size байтов из файла с описателем filedes, сохраняя результаты в буфере. (Это - не обязательно символьная строка и не имеется никакого добавленного пустого символа завершения.)

Возвращаемое значение - число байтов фактически прочитанных. Оно может быть меньше чем size.

Нулевое значение указывает конец файла (за исключением того, если значение аргумента size является также нулем). Это не является ошибкой.

Если read возвращает по крайней мере один символ, не имеется никакого способа, которым Вы можете узнать, был ли конец файла достигнут. Но если Вы достигали конца, следующий read возвратит нуль.

В случае ошибки read возвращает -1. Следующие errno условия ошибки определены для этой функции:

EAGAIN
Обычно, когда никакой ввод недоступен, read ждет какого-нибудь ввода.
Но если установлен флаг O_NONBLOCK (см. Раздел 8.10 [Флаги Состояния Файла]), read возвращается немедленно не читая никаких данных, и сообщает эту ошибку.

Примечание о Совместимости: Большинство версий BSD UNIX использует различный код ошибки для EWOULDBLOCK. В библиотеке GNU, EWOULDBLOCK - синоним EAGAIN, так что не имеет значения, акое название Вы используете.

На некоторых системах, чтение большого количества данных из символьного специального файла может также выдавать ошибку EAGAIN, если ядро не может найти достаточно памяти для страниц пользователя. Это ограничение для устройств, которые передают с прямым доступом в память, т. е. не включают терминалы, так как они всегда используют отдельные

буфера внутри ядра.

EBADF Filedes аргумент - не допустимый описатель файла.

EINTR Чтение было прервано сигналом, в то время как он ждал ввода.
См. Раздел 21.5 [Прерванные Примитивы]. EIO Для многих
устройств, и для файлов на диске, этот код
ошибки указывает аппаратную ошибку. EIO также происходит,
когда фоновый процесс пробует читать с
терминала управления, и нормальное действие остановки процесса,
т. е. Посылка сигнала SIGTTIN не работает. Это может случиться,
если сигнал блокируется или игнорируется, или т. к. группа
процесса - orphaned. См. Главу 24 [Управление заданиями].

Функция read - основной примитив для всех функций, которые читают из потоков, типа fgetc.

-Функция: ssize_t write (int filedes, const void *buffer, size_t size)

Функция write пишет до size байтов из буфера в файл с описателем filedes. Данные в буфере - не обязательно символьная строка и вывод пустого символа подобен любому другому.

Возвращаемое значение - число байтов, фактически записанных. Это - обычно size, но могло бы быть и меньшее количество. В случае ошибки, write возвращает -1. Следующие errno условия ошибки определены для этой функции:

EAGAIN
Обычно, write блокируется до завершения операции записи. Но если для файла устанавлен O_NONBLOCK флаг (см. Раздел 8.7 [Операции Управления]), она возвращается немедленно не позводя записи любых данных, и сообщает эту ошибку. Пример ситуации, котаря могла бы вызывать блокировани процесса на выводе, запись в терминальное устройство, которое поддерживает управление потоком данных, где вывод был приостановлен получением символа STOP.

Примечание Совместимости: Большинство версий BSD UNIX использует различный код ошибки для EWOULDBLOCK. В библиотеке GNU, EWOULDBLOCK - синоним для EAGAIN, так что не имеет значения, какое название Вы используете.

EBADF
Аргумент Filedes - не допустимый дескриптор файла.

EFBIG
Размер файла больший чем реализация, может поддерживать.

EINTR
Операция write была прервана сигналом, в то время как

она была блокирована, и ждала завершения. См. Раздел 21.5 [Прерванные Примитивы].

EIO
Для многих устройств, и для файлов на диске, этот код ошибки указывает аппаратную ошибку. EIO также происходит, когда фоновый процесс пробует читать с терминала управления, и нормальное действие остановки процесса, посылая сигнал SIGTTIN не работает. Это может случиться, если сигнал блокируется или игнорируется, или т. к. группа процесса - orphaned. См. Главу 24 [Управление заданиями].

ENOSPC
переполнение устройства.

EPIPE
Эта ошибка возвращается, когда Вы пробуете писать в канал или в FIFO, который не открыт для чтения процессом. Когда это случается, сигнал SIGPIPE также посылается в процесс; см. Главу 21 [Обработка Сигнала].

Если Вы не предотвращаете EINTR ошибки, Вам следует проверять errno после каждого выдающего ошибку обращения к write, и если ошибка была EINTR, Вы должны просто повторить обращение. См. Раздел 21.5 [Прерванные Примитивы]. Простой способ реализовать это - макрокомандой TEMP_FAILURE_RETRY, следующим образом:

nbytes = TEMP_FAILURE_RETRY (write (desc, buffer, count));

Функция write - основной примитив для всех функций записи в поток, типа fputc.

8.3 Установка Файловой позиции Дескриптора

Точно как Вы можете устанавливать файловую позицию потока функцией fseek, Вы можете устанавливать файловую позицию дескриптора функцией lseek. Она определяет позицию в файле для следующей операции read или write. См. Раздел 7.15 [Позиционирование Файла], для подробной информации относительно файловой позиции и что это означает.

Для получения текущего значения файловой позиции из описателя, используйте lseek (desc, 0, SEEK_CUR).

-Функция: off_t lseek (int filedes, off_t offset, int whence)

Функция lseek используется, чтобы изменить файловую позицию файла с описателем filedes. Аргумент whence определяет, как смещение должно интерпретироваться, таким же образом как в функции fseek, и может быть одной из символических констант SEEK_SET, SEEK_CUR, или SEEK_END.

SEEK_SET
Определяет, что whence - яисло символов от начала файла.

SEEK_CUR
Определяет, что whence - число символов от текущей файловой позиции. Этот число может быть положительно или отрицательно.

SEEK_END
Определяет, что whence - число символов с конца файла. Отрицательное число определяет позицию внутри текущего тела файла; положительное число определяет позицию после текущего конца. Если Вы устанавливаете позицию после текущего конца, и фактически записываете данные, Вы расширяете файл нулями до этой позиции.

Возвращаемое значение lseek - обычно возникающая в результате файловая позиция, измеряемая в байтах от начала файла. Вы можете использовать это средство вместе с SEEK_CUR для чтения текущей файловой позиции.

Вы можете устанавливать файловую позицию после текущего конца файла. Это делает файл больше; lseek никогда не изменяет файл. Но последующий вывод в ту позицию расширит файла.

Если файловая позиция не может быть изменена, или операция выполняется некоторым недопустимым способом, lseek возвращает значение -1. Следующие errno условия ошибки определены для этой функции:

EBADF
Filedes - не допустимый описатель файла.

EINVAL
Значение аргумента whence не допустимо, или возникающее в результате смещение файла не допустимо.

ESPIPE filedes соответствует каналу или FIFO, который не может быть позиционирован.

(Могут иметься другие виды файлов, которые также не могут быть

позиционированы, но в этих случаях поведение не определено.)
Функция lseek - основной примитив для fseek, ftell и rewind функций, которые функционируют на потоках вместо описателей файла.
Вы можете иметь многократные описатели для того же самого файла, если Вы открываете файл больше чем один раз, или если Вы дублируете описатель с dup. Дескрипторы, которые исходят из отдельных обращений open, имеют независимые файловые позиции; использование lseek на одном дескрипторе не производит никакого эффекта на другой. Например,

```
  {
       int d1, d2;
       char buf[4];
       d1 = open ('foo', O_RDONLY);
       d2 = open ('foo', O_RDONLY);
       lseek (d1, 1024, SEEK_SET);
       read (d2, buf, 4);
     }
```

будет читать первые четыре символа файла 'foo'. (Код с обнаружением ошибок, необходимый для реальной программы был опущен здесь для краткости.)
Напротив, описатели, сделанные дублированием совместно используют общую файловую позицию с первоначальным описателем, который был дублирован. Изменение файловой позиции одного из дубликатов, включая чтение или запись данных, воздействует на все из них. Таким образом, например,

```
{
       int d1, d2, d3;
       char buf1[4], buf2[4];
       d1 = open ('foo', O_RDONLY);
       d2 = dup (d1);
       d3 = dup (d2);
       lseek (d3, 1024, SEEK_SET);
       read (d1, buf1, 4);
       read (d2, buf2, 4);
     }
```


Будет читать четыре символа, начиная с 1024-го символа 'foo', и еще четыре символа, начиная с 1028-го символа.

-Тип данных: off_t
Это - арифметический тип данных, используемый, чтобы представить размеры файла. В системе GNU, он эквивалентен fpos_t или long int. Эти три синонима константы 'SEEK_...' существуют ради совместимости с более старыми BSD системами. Они определены в двух различных файлах: 'fcntl.h' и 'sys/file.h'.
L_SET синоним для SEEK_SET.
L_INCR синоним SEEK_CUR.
L_XTND синоним SEEK_END.

8.4 Дескрипторы и Потоки

Определяя дескриптор файла с помощью open, Вы можете создавать поток для него функцией fdopen. Вы можете получить основной описатель файла для существующего потока функцией fileno. Эти функции объявлены в заглавном файле 'stdio.h'.

-Функция: FILE * fdopen (int filedes, const char *opentype)
Функция fdopen возвращает новый поток для описателя файла filedes. Opentype аргумент интерпретируется таким же образом как в функции fopen (см. Раздел 7.3 [Открытие Потоков]), за исключением того, что опция 'b' не разрешается; это оттого, что GNU не делает никакого различия между текстом и двоичными файлами. Также, 'w' и 'w+' не вызывают усечение файла; они воздействуют только при открытии файла, а в этом случае, файл уже открыт. Вы должны удостовериться, что opentype аргумент соответствует фактическому режиму дескриптора файла.
Возвращаемое значение - новый поток. Если поток не может быть создан (например, если режимы для файла, обозначенного дескриптором файла не разрешают доступ, заданный opentype аргументом), взамен возвращается

пустой указатель.
Для примера, показывающего использование функции fdopen, см. Раздел 10.1 [Создание Канала].

-Функция: int fileno (FILE *stream)
Эта функция возвращает описатель файла, связанный с указанным потоком. Если обнаружена ошибка (например, если поток не допустим) или

если поток, не делает ввод - вывод в файл, fileno возвращает - 1.
Имеются также символические константы, определенные в 'unistd.h' для описателей файла, принадлежащих к стандартным потокам stdin, stdout, и stderr; см. Раздел 7.2 [Стандартные Потоки].
STDIN_FILENO
Эта макрокоманда имеет значение 0, которое является дескриптором файла для стандартного ввода.
STDOUT_FILENO Эта макркоманда имеет значение 1, которое является дескриптором файла для стандартного вывода.
STDERR_FILENO Эта макркоманда имеет значение 2, которое является дескриптором файла для стандартного вывода ошибки.

8.5 Опасности Смешивания Потоков и Дескрипторов

Вы можете иметь многочисленные дескрипторы файла и потоки (назывем и потоки и дескрипторы 'каналами', для краткости), связанными с одним и тем же файлом, но Вы должны соблюдать осторожность, чтобы избежать путаницы между каналами. Имеются два случая, для рассмотрения: связанные каналы, которые совместно используют одино значение файловой позиции, и независимые каналы которые имеют свои собственные файловые позиции.
Самое лучшее использовать только один канал в вашей программе для фактической передачи данных в любой данный файл, за исключением того, когда весь доступ создан для ввода. Например, если Вы открываете канал (кое-что Вы можете делать только на уровне дескрипторов файла), или делать весь ввод - вывод через дескриптор, или создавать поток с fdopen, и тогда делать весь ввод - вывод через поток.

8.5.1 Связанные Каналы

Каналы, которые исходят из одного открытия, совместно используют ту же самую файловую позицию; мы называем их связанными каналами. Связанные каналы кончаются, когда Вы делаете поток из дескриптора, используя fdopen, и когда Вы получаете описатель из потока с fileno, и когда Вы копируете описатель с dup или dup2. Для файлов, которые не поддерживают, произвольный доступ, типа терминалов и канадов, все каналы действительно связаны. На файлах прямого доступа, все выходные потоки конкатенирующего типа действительно связаны друг с другом.
Если Вы использовали поток для ввода - вывода, и Вы хотите делать ввод - вывод, используя другой канал (или поток или дескриптор) который связан с этим, Вы должны сначала очистить поток, который Вы использовали. См. Раздел 8.5.3 [Очистка Потоков].
Завершение процесса, или выполнение новой программы в процессе, уничтожает все потоки в процессе. Если описатели, связанные с этими потоками сохраняются в других процессах, их файловые позиции станут неопределенными. Чтобы предотвратить это, Вы должны очистить потоки перед их разрушением.

8.5.2 Независимые Каналы

Когда Вы открываете каналы (потоки или описатели) отдельно,

каждый канал имеет собственную файловую позицию. Они называются независимыми каналами.

Система обрабатывает каждый канал независимо. В большинстве случаев, это совершенно предсказуемостно и естественно (особенно для ввода): каждый канал может читать или писать последовательно в собственном месте в файле. Однако, если некоторые из каналов - потоки, Вы должны соблюдать предосторожности:

* Вы должны очистить выходной поток после использования, перед выполнением чего - нибудь еще, что могло бы читать или писать в ту же самую часть файла.

* Вы должны очистить входной поток перед чтением данных, которые могут изменяться использованием независимого канала. Иначе, Вы можете читать устаревшие данные, которые были в буфере потоков.

Если Вы выводите по одному каналу в конец файла, это конечно оставит другие независимые каналы, позиционированные где-нибудь перед новым концом. Если Вы хотите, чтобы они вывели в конец, Вы должны установить их файловые позиции в конец файла. (В этом нет нужды, если Вы используете описатель конкатенирующего типа или поток; они всегда выводят в текущий конец файла.)

Двум каналам невозможно иметь отдельные указатели файла для файла, который не поддерживает произвольный доступ. Таким образом, каналы для чтения или записи в такие файлы всегда связаны. Каналы конкатенирующего типа также всегда связаны. Для этих каналов, соблюдайте правила связанных каналов; см. Раздел 8.5.1 [Связанные Каналы].

8.5.3 Очистка Потоков

В системе GNU, Вы можете очистить любой поток функцией fclean:

-Функция: int fclean (FILE *stream)

Очищает указанный поток так, чтобы буфер был пуст. На других системах, Вы можете использовать fflush, чтобы очистить поток в большинстве случаев. Вы можете пропускать fclean или fflush, если Вы знаете, что поток уже пуст. Поток пуст всякий раз, когда буфер пуст. Например, небуферизованный поток всегда пуст. Входной поток, который находится в конце файла, пуст. Буферизированный строчно поток пуст, когда последний выходной символ был символ перевода строки.

Имеется один случай, в котором сброс потока является невозможным на большинстве систем. Тогда, когда поток делает ввод из файла, не являющегося файлом прямого доступа. Такие потоки обычно читают вперед, и когда файл - не произвольного доступа, то нет никакого способа сбросить обратно данные излишка уже прочтитанные. Когда входной поток читается из файла прямого доступа, fflush очищает поток, но оставляет указатель файла в непредсказуемом месте; Вы должны установить указатель файла перед выполнением любого дальнейшего ввода - вывода. В системе GNU, используя fclean Вы избежите обе эти проблемы.

fflush также делает закрытие выходного потока, так что это - допустимый способ очистки выходного потока. В системе GNU, закрытие

входного потока вызывает fclean.

Вы не нуждаетесь в чистке потока перед использованием дескриптора для операций управления, типа установки режимов терминала; эти операции не воздействуют на файловую позицию и не зависят от нее. Вы можете использовать любой дескриптор для этих операций, и на все каналы воздействовать одновременно. Однако, текст уже 'выведенный' в поток но все еще буферизируемый потоком будет подчинен новым режимам терминала. Чтобы удостовериться что 'прошлый' вывод подчинен установкам терминала, которые были в действительности во время, сбрасывайте выходные потоки для того терминала перед установкой режимов. См. Раздел 12.4 [Режимы Терминала].

8.6 Ожидание Ввода или Вывода

Иногда программа должна принять ввод на нескольких входных каналах всякий раз, когда ввод прибывает. Например, некоторые рабочие станции могут иметь устройства типа планшета отцифровывания, поля

функциональной клавиши, или поля набора кода, которые соединены через нормальные асинхронные последовательные интерфейсы; хороший стиль интерфейса пользователя требует, чтобы ответ немедленно воздействовал на любое устройство. Другой пример - программа, которая действует как сервер для нескольких других процессов через каналы или гнезда.

Вы не можете обычно использовать для этой цели read, потому что она блокирует программу до ввода на одном указанном описателе файла; ввод на других каналах не будет иметь действия. Вы могли бы установить режим не блокирования и опрашивать каждый описатель файла по очереди, но это очень неэффективно.

Лучшее решение состоит в том, чтобы использовать функцию выбора. Она блокирует программу до ввода или вывода по заданному набору описателей файла, или по таймеру. Это средство объявлено в заглавном файле 'sys/types.h'.

Наборы описателей файла для функции выбора определены как объекты fd_set. Вот описание типа данных и некоторых макрокоманд для управления этих объектов.

fd_set (тип данных) Этот тип данных представляет наборы описателей файла для функции
выбора. Это - фактически битовый массив. int FD_SETSIZE (макрос)

Значение этой макрокоманды - максимальное число описателей файла, о которых объект fd_set может содержать информацию. На системах с фиксированным максимальным номером, FD_SETSIZE - по крайней мере это число. На некоторых системах, включая GNU, не имеется никакого абсолютного ограничения числа описателей, но эта макрокоманда все еще имеет постоянное значение, которые управляет числом битов в fd_set.

void FD_ZERO (fd_set *set) (макрос) Эта макрокоманда инициализирует набор наборов описателей файла,
как пустое множество. void FD_SET (int filedes, fd_set *set) (макрос)
Эта макрокоманда добавляет filedes к набору описателей файлов. void FD_CLR (int filedes, fd_set *set) (макрос) Эта макрокоманда удаляет filedes из набора дескрипторов файлов. int FD_ISSET (int filedes, fd_set *set) (макрос) Эта макрокоманда возвращает значение отличное от нуля (истина),
если filedes - элемент набора описателей файлов, и нуль (ложь) иначе.

Вот описание функции выбора непосредственно. int select (int nfds, fd_set *read_fds, fd_set *write_fds,
fd_set *except_fds, struct timeval *timeout) Функция выбора блокирует процесс вызова до наличия действий на
любом из заданных наборов описателей файла, или пока период блокировки по времени не истечет.

Описатели файла, заданные read_fds аргументом проверяются, являются ли они готовыми для чтения; write_fds описатели файла проверяются, являются ли они готовыми записи; и except_fds описатели файла задают исключительные условия. Вы можете передавать пустой указатель для любого из этих аргументов, если Вы не заинтересованы проверкой этого вида условия.

'Исключительные условия' не означают, что ошибки сообщаются немедленно, когда выполнен ошибочный системный вызов. Они включают условия, типа присутствия срочного сообщения на гнезде. (См. Главу 11 [Гнезда], для уточнения информации о срочных сообщениях.)

Функция выбора проверяет только первые nfds описателей файла. Обычно она передает FD_SETSIZE как значение этого аргумента.

Блокировка по времени определяет максимальное время ожидания. Если Вы передаете пустой указатель в качестве этого аргумента, это означает, что блокировать неопределенно, пока один из описателей файла не готов. Иначе, Вы должны обеспечить время в формате struct timeval; см. Раздел 17.2.2 [Календарь с высоким разрешением]. Определите нуль как время

(struct timeval содержащий все нули) если Вы хотите выяснять, какие описатели готовы в данный момент, без ожидания.

Нормальное возвращаемое значение - общее число готовых описателей файла во всех наборах. Каждый из наборов аргументов записан поверх информацией относительно описателей, которые являются готовыми для соответствующей операции. Таким образом, чтобы видеть, готов ли данный описатель desc для ввода, используйте FD_ISSET (desc, read_fds) после возврата select.

Если select возвращается, потому что период блокировки по времени истекает, то возвращаемое значение - нуль.

Любой сигнал заставит select возвращаться немедленно. Так, если ваша программа использует сигналы, Вы не можете полагаться на select, чтобы ждать полное заданное время. Если Вы хотите убедиться в правильности ожидания, Вы должны проверить EINTR и повторять select с расчетной блокировкой по времени, основанной на текущем времени. См. пример ниже. См. также Раздел 21.5 [Прерванные Примитивы].

Если происходит ошибка, select возвращает -1 и не изменяет аргумент наборов описателей файла. Следующие errno условия ошибки определены для этой функции:

EBADF
Один из набора описателей файла определил недопустимый дескриптор файла.

EINTR
Операция была прервана сигналом. См. Раздел 21.5 [Прерванные Примитивы]. EINVAL Аргумент блокировки по времени недопустим; один из компонентов отрицателен или слишком большой.

Примечание о Переносимости:
функция select - BSD возможность UNIX. Вот пример, показывающий, как Вы можете использовать select, чтобы установить период блокировки по времени для чтения из дескриптора файла.
Input_timeout функция блокирует процесс вызова, пока ввод не доступен (на описателе файла), или пока период блокировки по времени не истек.


```
#include
#include
#include
#include

int
input_timeout (int filedes, unsigned int seconds)
{
  fd_set set;
  struct timeval timeout;

  /* Инициализируем набор дескрипторов файлов. */
  FD_ZERO (&set);
  FD_SET (filedes, &set);

  /* Инициализируем структуру timeout. */
  timeout.tv_sec = seconds;
  timeout.tv_usec = 0;

  /* `select' returns 0 if timeout, 1 if input available, -1 if error.
*/
  return TEMP_FAILURE_RETRY (select (FD_SETSIZE,
                                     &set, NULL, NULL,
                                     &timeout));
}

int
main (void)
{
  fprintf (stderr, 'select returned %d.\n',
           input_timeout (STDIN_FILENO, 5));
  return 0;
}
```


8.7 Операции Управления Файлами

Этот раздел описывает, как Вы можете выполнять различные другие

операции с дескрипторами файла, типа запроса или установки флагов, описывающих состояние описателя файла, манипулирования блокировками записи, и т.п.. Все эти операции выполняются функцией fcntl.

Второй аргумент fcntl функции - команда, которая определяет которую операцию выполнить. Функция и макрокоманды, которые обозначают различные флаги, объявлены в заглавном файле 'fcntl.h'. (Многие из этих флагов также используются функцией open; см. Раздел 8.1 [Открытие и Закрытие Файлов].)

-Функция: int fcntl (int filedes, int command, ...)

Функция fcntl выполняет операцию, заданную командой, на описателе файла filedes. Некоторые команды требуют, чтобы были обеспечены дополнительные аргументы. Эти дополнительные аргументы и возвращаемое значение и условия ошибки даны в детализированных описаниях индивидуальных команд.

Вот краткий список различных команд.

F_DUPFD
Дублирует дескриптор файла (возвращает другой дескриптор файла, указывающий на тот же самый открытый файл). См. Раздел 8.8 [Дублирование Дескрипторов].

F_GETFD
Получает флаги, связанные с описателем файла. См. Раздел 8.9 [Дескрипторные Флаги]. F_SETFD Устанавливает флаги, связанные с дескриптором файла. См. Раздел 8.9 [Дескрипторные Флаги].

F_GETFL
Получает флаги, связанные с открытым файлом. См. Раздел 8.10 [Флаги Состояния Файла].

F_SETFL
Устанавливает флаги, связанные с открытым файлом. См. Раздел 8.10 [Флаги Состояния Файла].

F_GETLK
Получает блокировку файла. См. Раздел 8.11 [Блокировки Файла].

F_SETLK
Устанавливает или снимает блокировку файла. См. Раздел 8.11 [Блокировки Файла].

F_SETLKW
Подобен F_SETLK, но ждет завершения. См. Раздел 8.11 [Блокировки Файла].

F_GETOWN
Получает процесс или ID группы процессов, чтобы получить сигналы SIGIO. См. Раздел 8.12 [Ввод Прерывания].

F_SETOWN
Устанавливает процесс или ID группы процессов, чтобы получить сигналы SIGIO. См. Раздел 8.12 [Ввод Прерывания].

8.8 Дублирование Дескрипторов

Вы можете дублировать дескриптор файла, или зарезервировать другой дескриптор файла, который относится к тому же самому открытому файлу, что и первый. Двойные дескрипторы совместно используют одну и ту же файловую позицию и один набор флагов состояния файла (см. Раздел 8.10 [Флаги Состояния Файла]), но каждый имеет собственный набор флагов дескриптора файла (см. Раздел 8.9 [Дескрипторные Флаги]).

Основное использование дублирования дескриптора файла это переадресация ввода или вывода: то есть замена файла или канала, которому указанный дескриптор файла соответствует.

Вы можете выполнять эту операцию, используя fcntl функцию с командой F_DUPFD, но имеются также удобные функции dup и dup2 для дублирования дескрипторов.

Fcntl функция и флаги объявлена в 'fcntl.h', в то время как прототипы для dup и dup2 находятся в файле 'unistd.h'.

-Функция: int dup (int old)

Эта функция копирует дескриптор old в первый доступный дескрипторный номер (первый номер не открытый сейчас). Это эквивалентно fcntl (old, F_DUPFD, 0).

-Функция: int dup2 (int old, int new)
Эта функция копирует дескриптор old в дескриптор new.
Если old - недопустимый дескриптор, то dup2 не делает ничего. Иначе, новый дубликат old заменяет любое предыдущее значение дескриптора new, как будто он был закрыт сначала.
Если old и new - различные числа, и old - допустимый дескрипторный номер, то dup2 эквивалентен:

```
close (new);
fcntl (old, F_DUPFD, new)
```

Однако, dup2 делает это автоматически; нет никакого момента в середине вызова dup2, когда new закрыт, а дубликата old еще нет.
-Макрос int F_DUPFD
Эта макрокоманда используется как аргумент команды fcntl, для копирования дескриптора файла, данного как первый аргумент.
Форма обращения в этом случае:

```
fcntl (old, F_DUPFD, next_filedes)
```

Next_filedes аргумент имеет тип int и определяет, что возвращенный дескриптор файла должен быть следующим доступным больше или равным этому значению.
Возвращаемое значение из fcntl с этой командой - обычно значение нового дескриптора файла. Возвращаемое значение -1 указывает ошибку. Следующие errno условия ошибки определены для этой команды:
EBADF
Аргумент old недопустим.
EINVAL
Next_filedes аргумент недопустим.
EMFILE
Дескрипторов файла, доступных вашей программе, больще нет.
ENFILE - не возможный код ошибки для dup2, потому что dup2 не создает новое открытие файла; двойные дескрипторные подпадают под ограничение, которое указывает ENFILE.
EMFILE возможен, потому что это относится к ограничению различных дескрипторных чисел для использования в одном процессе.
Вот пример, показывающий, как использовать dup2, чтобы делать переадресацию. Обычно, переадресация стандартных потоков (подобно stdin) выполняется командным интерпретатором или подобной программой перед вызовом

одной из запускаемых функций (см. Раздел 23.5 [Выполнение Файла]) чтобы выполнить новую программу в дочернем процессе. Когда новая программа выполнена, она создает и инициализирует стандартные потоки, чтобы указать на соответствующие дескрипторы файла, прежде, чем функция main вызывается.
Так, чтобы переназначить стандартный ввод в файл, оболочка могла бы делать что - нибудь вроде:

```
pid = fork ();
if (pid == 0)
        {
                char *filename;
                char *program;
                int file;
                .       . .
                file = TEMP_FAILURE_RETRY (open
                (filename, O_RDONLY));
                dup2 (file, STDIN_FILENO);
                TEMP_FAILURE_RETRY (close (file));
                execv (program, NULL);
        }
```

8.9 Флаги Дескриптора Файла

Флаги дескриптора Файла - разнообразные атрибуты дескриптора файла. Эти флаги связаны со специфическими дескрипторами файла, так, что если Вы создаете двойные дескрипторы файла из одиночного открытия файла, то каждый дескриптор имеет собственный набор флагов.
В настоящее время имеется только один флаг дескриптора файла: FD_CLOEXEC, который заставляет дескриптор быть закрытым если Вы используете любую из запускаемых функций (см. Раздел 23.5

[Выполнение Файла]).
Символы в этом разделе определены в заглавном файле 'fcntl.h'.
int F_GETFD (макрос)
Эта макрокоманда используется как аргумент команды fcntl, чтобы определять, что она должна возвратить флаги дескриптора файла, связанные с filedes аргументом.
Нормальное возвращаемое значение из fcntl с этой командой - неотрицательное число, которое может интерпретироваться как

поразрядное OR индивидуальных флагов (за исключением того, что в настоящее время имеется только один флаг).
В случае ошибки, fcntl возвращает -1. Следующие errno условия ошибки определены для этой команды:
EBADF filedes аргумент недопустим.
int F_SETFD (макрос)
Эта макрокоманда используется как аргумент команды fcntl, для определения, что она должна установить флаги дескриптора файла, связанные с filedes аргументом. Она требует третий int аргумент, чтобы определить новые флаги, так что форма обращения:
fcntl (filedes, F_SETFD, new_flags)
Нормальное возвращаемое значение из fcntl с этой командой - неопределенное значение отличное от -1, которое указывает ошибку. Флаги и условия ошибки - также как для команды F_GETFD.
Следующая макрокоманда определена для использования как флаг дескриптора файла с fcntl функцией. Значение - константа integer пригодное для использования как значение битовой маски.
int FD_CLOEXEC (макрос)
Этот флаг определяет, что дескриптор файла должен быть закрыт, когда вызывается запускаемая функция; см. Раздел 23.5 [Выполнение Файл]. Когда дескриптор файла размещен (как с open или dup), этот бит первоначально очищен на новом описателе файла, означая, что дескриптор останется в живых в новой программе после запуска.
Если Вы хотите изменять флаги дескриптора файла, Вы должны получить текущие флаги с F_GETFD и изменять значение. Не думайте, что флаги, перечисленные здесь - единственные, которые реализованы, ваша программа может быть будет выполняться через годы, и тогда может существовать большее количество флагов. Например, вот функция, чтобы устанавливать или очистить флаг FD_CLOEXEC без того, чтобы изменять любые другие флаги:

```
int
set_cloexec_flag (int desc, int value)
{
        int oldflags = fcntl (desc, F_GETFD, 0);

        if (oldflags < 0)
```



```
                return oldflags;
        /* Установим только флаг который мы хотим
        установить. */
        if (value != 0)
                oldflags |= FD_CLOEXEC;
        else
                oldflags &= ~FD_CLOEXEC;

        return fcntl (desc, F_SETFD, oldflags);
}
```

8.10 Флаги Состояния Файла

Флаги состояния Файла используются, чтобы определить атрибуты открытия файла. В отличие от флагов дескриптора файла, обсужденных в Разделе 8.9 [Дескрипторные Флаги], флаги состояния файла разделяются дублированными дескрипторами файла, следующими из одиночного открытия файла.
Флаги состояния файла инициализируются функцией open из аргумента флагов функции open. Некоторые из флагов значимы только в open и не вспоминаются впоследствии; многие из оставшихся не могут впоследствии быть изменены, хотя Вы можете читать их значения, исследуя флаги состояния файла.
Несколько флагов состояния файла могут быть изменены, в любое

время используя fcntl. Они включают O_APPEND и O_NONBLOCK.
Символы в этом разделе определены в заглавном файле 'fcntl.h'.
int F_GETFL (макрос)
Эта макрокоманда используется как аргумент команды fcntl, для чтения флагов состояния файла для файла открытого с описателем filedes.
Нормальное возвращаемое значение из fcntl с этой командой - неотрицательное число, которое может интерпретироваться как поразрядное ИЛИ индивидуальных флагов. Флаги закодированы так же, как аргумент флагов для open (см. Раздел 8.1 [Открытие и Закрытие Файлов]), но здесь значимы только режимы доступа файла и O_APPEND и O_NONBLOCK флаги. Так как режимы доступа файла - не одноразрядные значения, Вы можете маскировать другие биты в возвращенных флагах с O_ACCMODE, чтобы сравнить их.

В случае ошибки, fcntl возвращает -1. Следующие errno условия ошибки определены для этой команды:
EBADF filedes аргумент недопустим.
-Макрос: int F_SETFL
Эта макрокоманда используется как аргумент команды fcntl, для установки флагов состояния файла для открытого файла, соответствующего filedes аргументу. Эта команда требует третьего int аргумента, чтобы определить новые флаги, так что обращение походит на это:

```
fcntl (filedes, F_SETFL, new_flags)
```

Вы не можете изменять режим доступа для файла таким образом; то есть был ли дескриптор файла открыт для чтения или для записи. Вы можете только заменять флаги O_APPEND и O_NONBLOCK.
Нормальное возвращаемое значение из fcntl с этой командой - неопределенное значение отличное от -1, которое указывает ошибку. Условия ошибки - такие же как для команды F_GETFL.
Следующие макрокоманды определены для анализа и построения значений флага состояния файла:
O_APPEND - бит, который делает возможным конкатенирующий режим для файла. Если установлен, то все операции write пишут данные в конце файла, расширяя его, независимо от текущей файловой позиции.
O_NONBLOCK
Бит, который делает возможным режим неблокирования для файла. Если этот бит установлен, запрос чтения в файле может возвращаться немедленно с состоянием отказа, если не имеется никакого немедленно доступного ввода кроме блокирования. Аналогично, запросы write могут также возвращаться немедленно с состоянием отказа, если вывод нельзя написать немедленно.
O_NDELAY Это - синоним для O_NONBLOCK, предусматривающий совместимость с BSD.

-Макрос: int O_ACCMODE
Эта макрокоманда замещает маску, которая может быть поразрядна AND со значением флага состояния файла, чтобы произвести значение, представляющее режим доступа файла. Режим будет O_RDONLY, O_WRONLY, или O_RDWR.

O_RDONLY Открывает файл для чтения.
O_WRONLY Открывает файл для записи.
O_RDWR Открывает файл, и для чтения и для записи.
Если Вы хотите изменить флаги состояния файла, Вы должны получить текущие флаги с F_GETFL и изменять их значение. Вот функция, чтобы устанавливать или очистить флаг O_NONBLOCK без изменения любых других флагов:

```
int
set_nonblock_flag (int desc, int value)
{
        int oldflags = fcntl (desc, F_GETFL, 0);

        if (oldflags < 0)
                        return oldflags;
```

```
                        if (value != 0)
                                oldflags |= O_NONBLOCK;
                        else
                                oldflags &= ~O_NONBLOCK;
                        return fcntl (desc, F_SETFL, oldflags);
                }
```

8.11 Блокировки Файла

Остающиеся команды fcntl используются, чтобы поддерживать блокировку записей, который разрешает многим сотрудничающим программам предохранять друг друга от одновременного доступа к частям файла ошибочным способом.

Исключительная блокировка или блокировка записи дает монопольный доступ процессу для записи в заданной части файла. В то время как установлена блокировка никакой другой процесс не может блокировать эту часть файла.

Общедоступная блокировка или блокировка чтения запрещает любому другому процессу запрос блокировки записи в заданной части файла. Однако, другие процессы могут запрашивать блокировки чтения.

Функции read и write фактически не выясняет, имеются ли блокировки в данном месте.

Если Вы хотите выполнять протокол блокировки для файла, общедоступного многократными процессами, ваше приложение, должно делать явные обращения fcntl, чтобы запрашивать и очистить блокировки в соответствующих пунктах.

Блокировки связаны с процессами. Процесс может иметь только один вид набора блокировок для каждого байта данного файла. Когда любой дескриптор файла для того файла закрыт процессом, все блокировки, которые обрабатывают связи с тем файлом, опущены, даже если блокировки были сделаны, используя другие описатели, которые остаются открытыми. Аналогично, блокировки опущены, когда процесс выходит, и не наследуются дочерними созданными процессами (см. Раздел 23.4 [Создание Процесса]).

При создании блокировки, используйте struct flock, чтобы определить какую блокировку и где Вы хотите выполнить. Этот тип данных и связанные
макрокоманды для fcntl функции объявлен в заглавном файле 'fcntl.h'.

struct flock (тип данных)

Эта структура используется с fcntl функцией, чтобы описать блокировку файла. Она имеет следующие элементы:

short int l_type

Определяет тип блокировки; один из F_RDLCK, F_WRLCK, или F_UNLCK.

short int l_whence

Соответствует к аргументу whence для fseek или lseek, и определяет то, относительно чего задано смещение. Значение может быть одно из SEEK_SET, SEEK_CUR, или SEEK_END.

off_t l_start

Определяет смещение начала области, к которой блокировка применяется, и дана в байтах относительно отметки, заданной элементом l_whence.

off_t l_len

Определяет длину области, которая будет блокирована. Значение 0 обрабатывается особенно; это означает область, что простирается до конца файла.

pid_t l_pid

Это поле - ID процесса (см. Раздел 23.2 [Понятия Создания Процесса]) процесса, содержащего блокировку. Оно задается вызовом fcntl с командой F_GETLK, но игнорируется при создании блокировки.

-Макрос int F_GETLK

Эта макрокоманда используется как аргумент команды fcntl, для определения, что она должна получить информацию относительно блокировки. Эта команда требует третьего аргумента типа struct flock *, чтобы быть переданной к fcntl, так, чтобы форма обращения была:

```
  fcntl (filedes, F_GETLK, lockp)
```

Вы должны определить тип блокировки F_WRLCK, если Вы хотите выяснять относительно блокировок чтения и записи, или F_RDLCK, если

Вы хотите выяснять относительно блокировок только записи.
Может иметься больше чем одна блокировка, воздействующая на область, заданную lockp аргументом, но fcntl только возвращает информацию относительно одной из них. l_whence элемент структуры lockp установлен в SEEK_SET и l_start и l_len набор полей, чтобы идентифицировать блокированную область.
Если никакая блокировка не применяется, единственное изменение для структуры lockp, должно модифицировать l_type в значение F_UNLCK.
Нормальное возвращаемое значение из fcntl с этой командой - неопределенное значение отличное от -1, которое зарезервировано, чтобы указать ошибку. Следующие errno условия ошибки определены для этой команды:
EBADF filedes аргумент недопустим.
E INVAL Или lockp аргумент не определяет допустимую информацию блокировки, или файл, связанный с filedes не поддерживает блокировки.
-Макрос int F_SETLK
Эта макрокоманда используется как аргумент команды fcntl, для определения, что она должна установить или снять блокировку. Эта команда требует третьего аргумента типа struct flock *, так, чтобы форма обращения была:
fcntl (filedes, F_SETLK, lockp)
Если процесс уже имеет блокировку на любой части области, старая блокировка на той части, заменяется на новую блокировку. Вы можете удалять блокировку, определяя тип блокировки F_UNLCK.
Если блокировка не может быть установлена, fcntl возвращается немедленно со значением -1. Эта функция не блокирует ожидание пока другие процессы снимут блокировки. Если fcntl преуспевает, она

возвращает значение отличное от -1.
Следующие errno условия ошибки определены для этой функции:

EACCES
EAGAIN
Блокировка не может быть установлена, потому что это блокировано существующей блокировкой на файле.
Некоторые системы используют EAGAIN в этом случае, и другие системы используют EACCES; ваша программа должна обработать их одинаково, после F_SETLK.
EBADF
Также: filedes аргумент недопустим; Вы запросили блокировку чтения, но filedes - не открыт для чтения; или, Вы запросили блокировку записи, но filedes - не открыт для записи.
EINVAL
Или lockp аргумент определяет недопустимую информацию блокировки, или файл, связанный с filedes не поддерживает блокировки.
ENOLCK
Система работает без ресурсов блокировки файла; имеются уже слишком много блокировок файла.
Хорошо разработанные файловые системы никогда не сообщают эту ошибку, потому что они не имеют никакого ограничения на число блокировок. Однако, Вы должны все еще принимать во внимание возможности этой ошибки, поскольку это может следовать из доступа к сети к файловой системе на другой машине.
int F_SETLKW (макрос)
Эта макрокоманда используется как аргумент команды fcntl, для определения, что она должна установить или снять блокировку. Это - тоже что команда F_SETLK, но заставляет процесс блокировать (или ждать) пока запрос не будет определен.
Эта команда требует третьего аргумента типа struct flock *, как для команды F_SETLK.
Fcntl возвращаемые значения и ошибки - те же что для команды F_SETLK, но эти дополнительные errno условия ошибки определены для этой команды:

EINTR
Функция была прервана сигналом, во время ожидания. См.

Раздел 21.5 [Прерванные Примитивы].

EDEADLK

Было обнаружено условие тупика. Это может случаться, если два процесса каждый из которых управляет уже заблокированной областью запрашивают блокировку на области, блокированной другим процессом.

Следующие макрокоманды определены для использования как значения для l_type элемента структуры flock. Значения - константы integer.

F_RDLCK

Эта макрокоманда используется, чтобы установить блокировку чтения.

F_WRLCK

Эта макрокоманда используется, чтобы установить блокировку записи.

F_UNLCK

Эта макрокоманда используется, чтобы установить, что область разблокируется.

Наличие многократных копий программы, одновременно пишущих в файл может сделать содержимое файла бессмысленным. Но Вы можете предотвратить эту проблему, устанавливая блокировку записи на файле перед фактической записью в файл.

Если программа также нуждается в чтении из файла и хочет удостовериться, что содержимое файла находится в непротиворечивом состоянии, то она может также использовать блокировку чтения. В то время как блокировка чтения установлена, никакой другой процесс не может блокировать эту часть файла для записи.

Не забудьте, что блокировки файла - только произвольный протокол для управления доступом к файлу. Все еще имеется возможность для доступа к файлу программами, которые не используют протокол блокировки.

8.12 Управляемый прерываниями Ввод

Если Вы устанавливаете FASYNC флаг состояния на дескрипторе файла (см. Раздел 8.10 [Флаги Состояния Файла]), сигнал SIGIO посланы всякий раз, когда производится ввод или вывод, становится возможным на этом описателе файла. Процесс или группа процессов, может быть выбрана чтобы получить сигнал, используя команду F_SETOWN для fcntl функции. Если дескриптор файла является гнездом, он также, выбирает получателя сигналов SIGURG, которые возникают, когда внепоточные данные прибывает в это гнездо; см. Раздел 11.8.8 [Внепоточные Данные].

Если дескриптор файла соответствует терминалу, то SIGIO сигналы посылаются группе приоритетного процесса терминала. См. Главу 24 [Управление заданиями].

Символы в этом разделе определены в заглавном файле 'fcntl.h'.

-Макрос int F_GETOWN

Эта макрокоманда используется как аргумент команды fcntl, для определения того, что она должна получить информацию относительно процесса или группы процессов, которой посланы сигналы SIGIO. (Для терминала, это - фактически приоритетный ID группы процессов, которую Вы можете получать использованием tcgetpgrp; см. Раздел 24.7.3 [Функции Доступа Терминала].)

Возвращаемое значение интерпретируется как ID процесса; если оно отрицательно, то абсолютное значение - ID группы процессов.

Следующее errno условие ошибки определено для этой команды:

EBADF

Filedes аргумент недопустим.

int F_SETOWN (макрос)

fcntl (filedes, F_SETOWN, pid)

Возвращаемое значение из fcntl с этой командой - -1 в случае ошибки и некоторое другого значения если обращение успешно. Следующие errno условия ошибки определены для этой команды:

EBADF

Filedes аргумент недопустим.

ESRCH

Не имеется никакого процесса или группы процессов,

соответствующей pid.

- 203 -

9. Интерфейсы Файловой системы

Эта глава описывает функции библиотеки GNU C для управления файлами. В отличие от функций ввода и функций вывода, описанных в Главе 7 [ввод - вывод на Потоках] и Главе 8 [ввод - вывод низкого уровня], эти функции имеют отношение к действиям на файлах непосредственно.

Среди средств, описанных в этой главе - функции для исследования или изменения каталогов, функции для переименования и удаления файлов, и функции для исследования и установки атрибутов файла типа прав доступа и изменения времени.

9.1 Рабочий каталог

Каждый процесс связанный с этим каталогом, называемым текущим рабочим каталогом или просто рабочим каталогом, который используется в уточнении имен файла прямого доступа (см. Раздел 6.2.2 [Уточнение Имени файла]).

Когда Вы регистрируетесь в системе и начинаете новый сеанс, ваш рабочий каталог первоначально устанавливается в исходный каталог, связанный с вашим логическим входом в систему в базе данных пользователей системы. Вы можете находить исходный каталог любого пользователя, используются getpwuid или getpwnam функции; см. Раздел 25.12 [База данных Пользователей].

Пользователи могут изменять рабочий каталог, используя команды оболочки подобно cd. Функции, описанные в этом разделе - примитивы, используемые командами и другими программами для исследования и изменения рабочего каталога.

Прототипы для этих функций объявлены в файле 'unistd.h'.

char * getcwd (char *buffer, size_t size) (функция)

Getcwd функция возвращает абсолютное имя файла, представляющее текущий рабочий каталог, сохраняя его в символьном буфере, который Вы указываете. Аргумент size - то, как Вы сообщаете системе размер резервирования буфера.

Версия GNU этой функции также разрешает Вам определять пустой указатель для буферного аргумента. Тогда getcwd зарезервирует буфер автоматически, как malloc (см. Раздел 3.3 [Беспрепятственное Резервирование]).

- 204 -

Возвращаемое значение - буфер в случае успеха и пустой указатель при отказе. Следующие errno условия ошибки определены для этой функции:

EINVAL аргумент size - нуль, и буфер - не пустой указатель.

ERANGE аргумент size - меньше чем длина имени рабочего каталога. Вы должны зарезервировать больший массив и попытаться снова.

EACCES Права на чтение или поиск компонент имени файла были превышены.

Вот пример, показывающий, как Вы могли бы реализовать GNU getcwd (NULL, 0) используя только стандартное поведение getcwd:

```
char *
gnu_getcwd ()
{
        int size = 100;
        char *buffer = (char *) xmalloc (size);
        while (1)
                {
                        char*value = getcwd(buffer,size);
                        if (value != 0)
                                return buffer;
                        size *= 2;
                        free (buffer);
                        buffer = (char *) xmalloc (size);
                }
}
```

См. Раздел 3.3.2 [Примеры Malloc], для уточнения информации относительно xmalloc, которая - не библиотечная функция, но - обычное имя, используемое в большинстве программного обеспечения GNU.

char * getwd (char *buffer) (функция)
Подобна getcwd. Библиотека GNU обеспечивает getwd для совместимости в обратном направлении с BSD. Буфер должен быть указатель на массив по крайней мере PATH_MAX байтов.
-Функция: int chdir (const char *filename)
Эта функция используется, чтобы установить рабочий каталог процесса filename.
Нормальное, успешное возвращаемое значение из chdir - 0. Значение -1 возвращается, чтобы указать ошибку. Errno условия

ошибки, определенные для этой функции - обычные синтаксические ошибки имени файла (см. Раздел 6.2.3 [Ошибки Имени файла]), плюс ENOTDIR, если filename файла - не каталог.

9.2 Доступ в Каталоги

Средства, описанные в этом разделе допускают Вам читать содержимое файла каталога. Это полезно, если Вы хотите, чтобы ваша программа перечислила все файлы в каталоге, возможно как часть меню.
Opendir функция открывает поток каталога, чьи элементы являются входами каталога. Вы используете readdir функцию на потоке каталога, чтобы отыскать эти входы, представляемые как объекты struct dirent. Имя файла для каждого входа сохранено в d_name элементе этой структуры. Имеются очевидные параллели здесь со средствами потока для обычных файлов, описанных в Главе 7 [ввод - вывод на Потоках].

9.2.1 Формат Входа в Каталог

Этот раздел описывает то, что Вы находите входя в каталога, поскольку Вы могли бы получать это из потока каталога. Все символы объявлены в заглавном файле 'dirent.h'.

-Тип данных: struct dirent
Это - тип структуры, используемый, чтобы возвратить информацию относительно входов в каталог. Она содержит следующие поля:
char *d_name
Это - компонента имени файла с нулевым символом в конце. Это - единственное поле, на которое Вы можете рассчитывать во всех POSIX системах.
ino_t d_fileno
Это - серийный номер файла. Для BSD совместимости, Вы можете также обратиться к этому элементу как к d_ino.
size_t d_namlen
Это - длина имени файла, исключая пустой символ завершения.
Эта структура в будущем может содержать дополнительные элементы.
Когда файл имеет многочисленные имена, каждое имя имеет собственный вход в каталог. Единственный способ которым Вы можете

сообщать, что входы каталога принадлежат одиночному файлу - это, если они имеют то же самое значение для d_fileno поля.
Атрибуты Файла типа размера, числа изменений, и т.п. - часть файла непосредственно, см. Раздел 9.8 [Атрибуты Файла].

9.2.2 Открытие Потока Каталога

Этот раздел описывает, как открыть поток каталога. Все символы объявлены в заглавном файле 'dirent.h'.
-Тип данных DIR
Тип данных DIR представляет поток каталога.
Вы не должны когда-либо резервировать объекты struct dirent или типов данных DIR, так как функции доступа каталога делают это для Вас. Вместо этого, Вы обращаетесь к этим объектам, используя указатели, возвращенные следующими функциями.
DIR * opendir (const char *dirname) (функция)
Opendir функция открывает и возвращает поток каталога для чтения каталога, чье имя dirname. Поток имеет тип DIR *.
Если происходит ошибка opendir возвращает пустой указатель. В дополнение к обычным синтаксическим ошибкам имени файла (см. Раздел 6.2.3 [Имя файла Errors]), следующие errno условия ошибки определены для этой функции:
EACCES право чтения отклонено для каталога dirname.

EMFILE процесс имеет слишком много открытых файлов.
ENFILE вся система, или возможно файловая система, которая содержит каталог, не может поддерживать любые дополнительные открытые файлы в настоящее время. (Эта проблема не может случаться в системе GNU.)
Тип DIR обычно реализован, используя дескриптор файла, а opendir функция в терминах функции open. См. Главу 8 [ввод - вывод низкого уровня]. Потоки Каталога и основные дескрипторы файла закрываются при запуске (см. Раздел 23.5 [выполнение Файла]).

9.2.3 Чтение и Закрытие Потока Каталога

Этот раздел описывает, как читать входы каталога из потока каталога, и как закрыть поток, когда Вы закончили работу с ним. Все символы объявлены в файле 'dirent.h'.

struct dirent * readdir (DIR *dirstream) (функция)
Эта функция читает следующий вход из каталога. Она обычно возвращает указатель на структуру, содержащую информацию относительно файла. Эта структура статически размещена и может быть перезаписана последующим обращением.
Примечание Переносимости: На некоторых системах, readdir не может возвращать входы для '. ' и ' .. ', даже если они - всегда допустимые имена файла в любом каталоге. См. Раздел 6.2.2 [Уточнение Имени файла].
Если нет больше входов в каталоге, или обнаружена ошибка, readdir возвращает пустой указатель. Следующие errno условия ошибки определены для этой функции:
EBADF dirstream аргумент не допустим.
int closedir (DIR *dirstream) (функция)
Эта функция закрывает поток каталога dirstream. Она возвращает 0 при успехе и -1 при отказе.
Следующие errno условия ошибки определены для этой функции:
EBADF dirstream аргумент не допустим.

9.2.4 Простая Программа Просмотра Каталога

Имеется простая программа, которая печатает имена файлов в текущем рабочем каталоге:

```
#include
#include
#include
#include
int
main (void)
{
        DIR *dp;
        struct dirent *ep;
        dp = opendir ('./');
        if (dp != NULL)
                {
                        while (ep = readdir (dp))
                                puts (ep->d_name);
                        (void) closedir (dp);
                }

               - 208 -

        else
                puts ('Couldn't open the directory.');
        return 0;
}
```

9.2.5 Произвольный доступ в Потоке Каталога

Этот раздел описывает, как повторно читать части каталога, который Вы уже читали из открытого потока каталога. Все символы объявлены в заглавном файле 'dirent.h'.
void rewinddir (DIR *dirstream) (функция)
Rewinddir функция используется, чтобы повторно инициализировать поток каталога dirstream, так, чтобы, если Вы вызвали readdir, она возвращала информацию относительно первого входа в каталоге снова.
off_t telldir (DIR *dirstream) (функция)

Telldir функция возвращает файловую позицию потока каталога dirstream. Вы можете использовать это значение с seekdir, чтобы восстановить поток каталога в эту позицию.

void seekdir (DIR *dirstream, off_t pos) Function

Seekdir функция устанавливает файловую позицию потока каталога dirstream в pos.

Значение pos должно быть результатом предыдущего обращения к telldir на этом специфическом потоке; закрытие и повторное открытие каталога может объявить неверным значения, возвращенные telldir.

9.3 Жесткие Связи

В POSIX системах, один файл может иметь много имен в то же самое время. Все имена в равной степени реальны, и никакое из них не лучше других.

Чтобы добавить имя к файлу, используйте функцию link. (Новое имя также называется жесткой связью с файлом.) Создание нового обращения к файлу не копирует содержимое файла; просто появляется новое имя, под которым файл может быть известен, в дополнение к существующему имени файла (или именам).

Один файл может иметь имена в отдельных каталогах, так что, организация файловой системы - не строгая иерархия или дерево.

Так как специфический файл существует внутри одиночной файловой системы, все имена должны быть в каталогах в этой файловой системе.

Link сообщает ошибку, если Вы пробуете делать жесткое обращение к файлу из другой файловой системы.

Прототип для функции link объявлен в заглавном файле 'unistd.h'.

int link (const char *oldname, const char *newname) (функция)

Функция link делает новую ссылка с существующим файлом, называемым oldname, под новым именем newname.

Эта функция возвращает значение 0, если она успешна и -1 при отказе. В дополнение к обычным синтаксическим ошибкам имени файла (см. Раздел 6.2.3 [Ошибки Имени файла]) и для oldname и newname, следующие errno условия ошибки определены для этой функции:

EACCES каталог, в котором link должна написать, нельзя перезаписывать.

EEXIST имеется уже файл, именованный newname. Если Вы хотите заменить эту ссылку на новую ссылку, Вы должны сначала явно удалить старую ссылку.

EMLINK имеются уже слишком много связей к файлу, именованному oldname. (Максимальное число связей к файлу - LINK_MAX; см. Раздел 27.6 [Ограничения для Файлов].)

Хорошо разработанные файловые системы никогда не сообщают эту ошибку, потому что они разрешают большее количество связей чем ваш диск мог бы содержать. Однако, Вы должны все еще принимать во внимание возможности этой ошибки, поскольку она могла бы следовать из доступа к сети к файловой системе на другой машине.

ENOENT файл, именованный oldname не существует. Вы не можете делать ссылок к файлу, который не существует.

ENOSPC каталог или файловая система, которая содержала бы, новую ссылка 'полна' и не может быть расширена.

EPERM Некоторые реализации позволяют только привилегированным пользователям делать связи к каталогам, а другие запрещают эту операцию полностью. Эта ошибка используется, чтобы сообщить эту проблему.

EROFS Каталог, содержащий новую ссылка не может изменяться, потому что он находится в файловой системе только для чтения.

EXDEV Каталог, заданный в newname находится в другой файловой системе чем существующий файл.

9.4 Символические Связи

Система GNU поддерживает связи программного обеспечения или символические связи. Это - вид 'файла', который является по существу указателем на другое имя файла. В отличие от жестких обращений, символические связи могут быть сделаны к каталогам или между файловыми системами без ограничений.

Функция open понимает, что Вы передали имя ссылки, и читает имя

файла, вместо этого открывает файл, указываемый ссылкой. Stat функция аналогично функционирует на файле, на который указывает символическая ссылка, а не на связи непосредственно. Как связывается, функция, которая делает жесткое обращение.

Наоборот, другие операции типа удаления или переименования файла воздействуют непосредственно на ссылки. Функции readlink и lstat также воздерживаются от следующих символических ссылок, потому что их цель - получить информацию относительно ссылки.

Прототипы для функций, перечисленных в этом разделе находятся в 'unistd.h'.

-Функция: int symlink (const char *oldname, const char *newname)

Symlink функция делает символическую ссылку к oldname, с именем newname.

Нормальное возвращаемое значение из symlink - 0. Возвращаемое значение -1 указывает ошибку.

В дополнение к обычным синтаксическим ошибкам имени файла (см. Раздел 6.2.3 [Ошибки Имени файла]), следующие errno условия ошибки определены для этой функции:

EEXIST имеется уже существующий файл по имени newname.

EROFS файл newname существовал бы на файловой системе только для чтения.

ENOSPC каталог или файловая система не может быть расширена, чтобы делать новую ссылка.

EIO аппаратная ошибка произошла при чтении или записи данных относительно диска.

-Функция: int readlink (const char *filename, char *buffer, size_t size)

Readlink функция получает значение символической filename связи. Имя файла, на которое ссылка указывает, скопировано в буфер. Эта строка имени файла не с нулевым символом в конце; readlink обычно возвращает скопированное число символов. Аргумент size определяет

максимальное число символов, для копирования, обычно размер резервирования буфера.

Если возвращаемое значение равняется size, Вы не можете распознать, имелся или нет участок памяти, чтобы возвратить все имя. Так что делайте больший буфер, и вызовите readlink снова. Вот пример:

```
char *
readlink_malloc (char *filename)
{
        int size = 100;
        while (1)
        {
                char *buffer = (char *) xmalloc (size);
                int nchars= readlink(filename,buffer,size);
                if (nchars < size)
                return buffer;
                free (buffer);
                size *= 2;
        }
}
```

Значение -1 возвращается в случае ошибки. В дополнение к обычным синтаксическим ошибкам имени файла (см. Раздел 6.2.3 [Ошибки Имени файла]), следующие errno условия ошибки определены для этой функции:

EINVAL именованный файл - не символическая ссылка.

EIO аппаратная ошибка произошла при чтении или записи данных относительно диска.

9.5 Удаление Файлов

Вы можете удалять файл функциями unlink или remove. (Эти имена синонимичны.)

Стирание фактически удаляет имя файла. Если это является только именем файла, то файл удален также. Если файл имеет другие имена также (см. Раздел 9.3 [Жесткие обращения]), то он остается доступным под другими именами.

-Функция: int unlink (const char *filename)

Функция unlink удаляет filename имя файла. Если это является единственным именем файла, файл непосредственно также удален. (Фактически, если любой процесс работает с файлом, когда это случается, стирание откладывается, пока все процессы не закрыли файл.)

Unlink функция объявлена в заглавном файле 'unistd.h'.

Эта функция возвращает 0 при успешном завершении, и -1 при ошибке. В дополнение к обычным синтаксическим ошибкам имени файла (см. Раздел 6.2.3 [Имя файла Errors]), следующие errno условия ошибки определены для этой функции:

EACCESS право записи отклонено для каталога из которого файл должен быть удален.

EBUSY Эта ошибка указывает, что файл используется системой таким способом, что он не может быть удален.

ENOENT имя файла, которое будет удалено не существует.

EPERM На некоторых системах, unlink не может использоваться, чтобы удалить имя каталога, или может использоваться только привилегированным пользователем. Чтобы избегать таких проблем, используйте rmdir, чтобы удалить каталоги.

EROFS Каталог, в котором имя файла должно быть удалено, находится в файловой системе только для чтения, и не может изменяться.

-Функция: int remove (const char *filename)

Функция remove - другое имя для unlink. Remove - имя ANSI C, в то время как unlink - POSIX.1 имя. Remove имя объявлено в 'stdio.h'.

-Функция: int rmdir (const char *filename)

Rmdir функция удаляет каталог. Каталог должен быть пуст прежде, чем он может быть удален; другими словами, он может только содержать элементы '. ' и ' .. '.

В большинстве других отношений, rmdir ведет себя подобно unlink. Имеются два дополнительных errno условия ошибки, определенные для rmdir:

EEXIST

ENOTEMPTY

Каталог, который будет удален не пуст.

Эти два кода ошибки синонимичны; некоторые системы используют

один, а некоторые - другой.

Прототип для этой функции объявлен в заглавном файле 'unistd.h'.

9.6 Переименование Файлов

Функция rename используется, чтобы заменить имя файла.

int rename (const char *oldname, const char *newname) (функция)

Функция rename переименовывает имя файла oldname в newname. Файл, прежде доступный под именем oldname позже доступен как newname. (Если файл имел любые другие имена кроме oldname, он продолжает иметь те имена.)

Каталог, содержащий имя newname должен быть в той же самой файловой системе как файл (что обозначен именем oldname).

Один частный случай для rename - то, когда oldname и newname - два имени для того же самого файла. Непротиворечивый способ обрабатывать этот случай состоит в том, чтобы удалить oldname. Однако, POSIX говорит, что rename в этом случае не делает ничего, и не сообщает об ошибке. Мы не знаем то, что ваша операционная система будет делать. Система GNU будет возможно делать правильно (удалять oldname) если Вы явно не запрашиваете строгую POSIX совместимость 'даже когда она причиняет вред'.

Если oldname - не каталог, то любой существующий файл, именованный newname, удален в течение операции переименования. Однако, если newname - имя каталога, происходит сбой rename в этом случае.

Если oldname является каталогом, то или newname, не должен существовать, или должен быть указан каталог, который является пустым. В последнем случае, существующий каталог, именованный newname удален сначала. Имя newname не должно определить подкаталог каталога oldname, который переименовывается.

Если rename терпит неудачу, она возвращает -1. В дополнение к обычным синтаксическим ошибкам имени файла (см. Раздел 6.2.3 [Имя файла Errors]), следующие errno условия ошибки определены для этой функции:

EACCES Один из каталогов, содержащих newname или oldname отказывает в записи; или newname и oldname - каталоги, и в праве записи отказано для одного из них.

EBUSY Каталог, именованный oldname или newname используется системой способом, который предотвращает переименование во время работы. Это включает каталоги, которые являются точками крепления для filesystems, и каталогов, которые являются текущими рабочими каталогами процессов.
EEXIST каталог newname существует.
ENOTEMPTY
Каталог newname не пуст.
EINVAL oldname - каталог, который содержит newname.
EISDIR newname называет каталог, а oldname нет.
EMLINK Каталог предыдущего уровня newname имел бы слишком много связей.
Хорошо разработанные файловые системы никогда не сообщают эту ошибку, потому что они разрешают большое количество связей чем ваш диск, мог бы содержать. Однако, Вы должны все еще принимать во внимание возможности этой ошибки, поскольку она могла бы следовать из доступа к сети к файловой системе на другой машине.
ENOENT файл, именованный oldname не существует.
ENOSPC каталог, который содержал бы, newname не имеет никакого участка памяти для другого входа, и нет никакого места, оставшегося в файловой системе, чтобы расширить его.
EROFS Операция включила бы запись в каталогу на файловой системе только для чтения.
EXDEV Два имени файла newname и oldnames находятся в различных файловых системах.

9.7 Создание Каталогов

Каталоги создаются функцией mkdir. (Имеется также команда оболочки mkdir, которая делает то же самое.)

-Функция: int mkdir (const char *filename, mode_t mode)
Mkdir функция создает новый, пустой каталог, с именем filename.
Mode аргумент определяет права файла для нового файла каталога. См. Раздел 9.8.5 [Биты Прав], для получения более подробной информации об этом.
Возвращаемое значение 0 указывает на успешное завершение, а -1 указывает на отказ. В дополнение к обычным синтаксическим ошибкам имени файла (см. Раздел 6.2.3 [Ошибки Имени файла]), следующие

errno условия ошибки определены для этой функции:
EACCES право записи отклонено для директории предыдущего уровня, в которую новый каталог должен быть добавлен.
EEXIST файл, именованный filename уже существует.
EMLINK директория предыдущего уровня имеет слишком много связей.
Хорошо разработанные файловые системы никогда не сообщают эту ошибку, потому что они разрешают большое количество связей чем ваш диск мог бы содержать. Однако, Вы должны все еще принимать во внимание возможности этой ошибки, поскольку она могла бы следовать из доступа к сети к файловой системе на другой машине.
ENOSPC файловая система не имеет достаточного участка памяти, чтобы создать новый каталог.
EROFS Директория предыдущего уровня создаваемого каталога находится в файловой системе только для чтения, и не может изменяться.
Чтобы использовать эту функцию, ваша программа должна включить заглавный файл 'sys/stat.h'.

9.8 Атрибуты Файла

Когда Вы применяете 'ls -l' команду оболочки на файле, это дает Вам информацию относительно размера файла, кто его хозяин, когда было последнее изменение, и т.п.. Этот вид информации называется атрибутами файла; они связаны с файлом непосредственно и не часть одного из имен.
Этот раздел содержит информацию относительно того, как Вы можете запрашивать и изменять эти атрибуты файлов.

9.8.1 Что Означают Атрибуты Файла

Когда Вы читаете атрибуты файла, они возвращаются в структуре, называемой struct stat. Этот раздел описывает имена атрибутов, их типов данных, и что они означают.

Заглавный файл 'sys/stat.h' объявляет все символы, определенные в этом разделе.

struct stat (тип данных)

Тип структуры stat используется, чтобы возвратить информацию относительно атрибутов файла. Она содержит по крайней мере следующие элементы:

mode_t st_mode

Определяет режим файла. Включая информацию о типе файла (см. Раздел 9.8.3 [Тестирование Типа Файла]) и биты прав файла (см. Раздел 9.8.5 [Биты Права]).

ino_t st_ino

Серийный номер файла, который отличает этот файл от всех других файлов на том же самом устройстве.

dev_t st_dev

Идентифицирует устройство, содержащее файл. St_ino и st_dev, вместе, однозначно идентифицируют файл.

nlink_t st_nlink

Число жестких связей с файлом. Этот счетчик следит, сколько каталогов имеют входы для этого файла. Если счетчик когда-либо уменьшится до нуля, то файл непосредственно отбрасывается. Символические связи не рассчитываются.

uid_t st_uid

ID владельца файла. См. Раздел 9.8.4 [Владелец Файла].

gid_t st_gid

ID группы файла. См. Раздел 9.8.4 [Владелец Файла].

off_t st_size

Это определяет размер файла в байтах. Для файлов, которые являются устройствами и т.п. это поле не значимо.

time_t st_atime

Это - время последнего доступа к файлу. См. Раздел 9.8.9 [Времена Файла].

unsigned long int st_atime_usec

Это - дробная часть времени последнего доступа к файлу.

time_t st_mtime

Это - время последней модификации содержимого файла.

unsigned long int st_mtime_usec

Это - дробная часть времени последней модификации содержимого файла.

time_t st_ctime

Это - время последней модификации атрибутов файла. См. Раздел 9.8.9 [Времена Файла].

unsigned long int st_ctime_usec

Это - дробная часть времени последней модификации атрибутов файла.

unsigned int st_nblocks

Это - количество дискового пространства, которое файл занимает, измеряемое в модулях (512-байтовых блоках).

Число блоков диска не строго пропорционально размеру файла, по двум причинам: файловая система может использовать, некоторые блоки для внутреннего хранения записи; и файл может быть разреженен, т. е. может иметь 'отверстия', которые содержат нули, но фактически не занимают пространство на диске.

Вы можете узнать (приблизительно) является ли файл разреженен, сравнивая это значение с st_size, примерно так:

(st.st_blocks * 512 < st.st_size)

Этот тест не совершенен, потому что файл, который только немного разрежен, не мог бы быть обнаружен как разреженный вообще. Для практических приложений, это - не проблема.

unsigned int st_blksize

Оптимальный размер блока для чтения или записи в этот файл. Вы могли бы использовать этот размер для распределения пространства буфера для чтения или для записи в файл.

Некоторые из атрибутов файла имеют специальные имена типа данных, которые существуют специально для этих атрибутов. (Они - все побочные результаты исследования для общеизвестных типов

integer, которые Вы знаете и любите.) Эти typedef-имена определены в заглавном файле 'sys/types.h' также как в 'sys/stat.h'. Вот их список:

mode_t (тип данных)
Это - тип данных integer, используемый, чтобы представить режимы файла. В системе GNU, это эквивалентно unsigned int.

ino_t (тип данных)
Это - арифметический тип данных, используемый, чтобы представить серийные номера файла. (В UNIX жаргоне они иногда называются inode числами.) В системе GNU, этот тип эквивалентен long unsigned int.

dev_t (тип данных)
Это - арифметический тип данных, используемый, чтобы представить числа файлового устройства. В системе GNU, это эквивалентно int.

nlink_t (тип данных)
Это - арифметический тип данных, используемый, чтобы представить число связей файла. В системе GNU, это эквивалентно short unsigned int.

9.8.2 Чтение Атрибутов Файла

Чтобы исследовать атрибуты файлов, используйте функции stat, fstat и lstat. Они возвращают информацию атрибута в объекте struct stat. Все три функции объявлены в заглавном файле 'sys/stat.h'.

int stat (const char *filename, struct stat *buf) (функция)
Stat функция возвращает информацию относительно атрибутов файла, именованного filename в структуре, указанной в buf.

Если filename - имя символической связи, атрибуты, которые Вы получаете, описывают файл, на который ссылка указывает. Если ссылка направляет на несуществующее имя файла, то stat сбоит, сообщая несуществующий файл.

Возвращаемое значение - 0, если операция успешна, и -1 при отказе. В дополнение к обычным синтаксическим ошибкам имени файла (см. Раздел 6.2.3 [Ошибки Имени файла], следующие errno условия ошибки определены для этой функции:

ENOENT файл, именованный filename не существует.

int fstat (int filedes, struct stat *buf) (функция)
Fstat функция - подобна stat, за исключением того, что она берет открытый дескриптор файла как аргумент, вместо имени файла. См. Главу 8 [ввод - вывод низкого уровня].

Подобно stat, fstat возвращает 0 при успехе и -1 при отказе. Следующие errno условия ошибки определены для fstat:

EBADF filedes аргумент - не допустимый дескриптор файла.

int lstat (const char *filename, struct stat *buf) (функция)
Lstat функция - подобна stat, за исключением того, что она не следует за символическими связями. Если filename - имя символической связи, lstat возвращает информацию относительно связи непосредственно; иначе, lstat работает подобно stat. См. Раздел 9.4 [Символические Связи].

9.8.3 Определение Типа Файла

Режим файла, сохраненный в st_mode поле атрибутов файла, содержит два вида информации: код типа файла, и биты прав доступа. Этот раздел обсуждает только код типа, который Вы можете использовать, чтобы сообщить является ли файл каталогом, является ли он гнездом, и так далее. Для уточнения информации относительно

права доступа см. Раздел 9.8.5 [Биты Прав].

Имеются два предопределенных способа, которыми Вы можете обращаться к типу файла. Прежде всего для каждого типа файла, имеется макрокоманда предиката, которая исследует значение режима файла и возвращает истину или ложь если файл того типа, или нет. Во-вторых, Вы можете маскировать снаружи остальную часть режима файла, чтобы получить только код типа файла. Вы можете сравнивать его с различными константами типов файлов.

Все символы, перечисленные в этом разделе определены в заглавном файле 'sys/stat.h'.

Следующие макрокоманды предиката проверяют тип файла, заданный значением m, которое является st_mode полем, возвращенным stat на этом файле:

int S_ISDIR (mode_t m) (макрос)

Эта макрокоманда возвращает отличное от нуля значение, если файл - каталог.
int S_ISCHR (mode_t m) (макрос)
Эта макрокоманда возвращает отличное от нуля значение, если файл - символьный специальный файл (устройство подобное терминалу).
int S_ISBLK (mode_t m) (макрос)
Эта макрокоманда возвращает отличное от нуля значение, если файл - блокированный специальный файл (устройство подобное диску).
int S_ISREG (mode_t m) (макрос)
Эта макрокоманда возвращает отличное от нуля значение, если файл - регулярный(правильный) файл.
int S_ISFIFO (mode_t m) (макрос)
Эта макрокоманда возвращает отличное от нуля значение, если файл - FIFO специальный файл, или канал. См. Главу 10 [Каналы и FIFO].
int S_ISLNK (mode_t m) (макрос)
Эта макрокоманда возвращает отличное от нуля значение, если файл - символическая ссылка. См. Раздел 9.4 [Символические Связи].
int S_ISSOCK (mode_t m) (макрос)
Эта макрокоманда возвращает отличное от нуля значение, если файл - гнездо. См. Главу 11 [Гнезда].
Альтернативный, не-POSIX метод тестирования типа файла обеспечивается для совместимости с BSD.
Режим можно поразрядно AND с S_IFMT, чтобы извлечь лод типа

файл, и сравнить с соответствующей константой кода типа. Например,
S_ISCHR (mode)
эквивалентно:
((mode & S_IFMT) == S_IFCHR)
int S_IFMT (макрос)
Это - битовая маска, используемая, чтобы извлечь код типа файла.
Вот символические имена для различных типов файлов:
S_IFDIR Эта макрокоманда представляет значение кода типа файла для файла каталога.
S_IFCHR Эта макрокоманда представляет значение кода типа файла для файла - устройства с символьной организацией.
S_IFBLK Эта макрокоманда представляет значение кода типа файла для блочно-ориентированного файла.
S_IFREG Эта макрокоманда представляет значение кода типа файла для регулярного(правильного) файла.
S_IFLNK Эта макрокоманда представляет значение кода типа файла для символической связи.
S_IFSOCK Эта макрокоманда представляет значение кода типа файла для гнезда.
S_IFIFO Эта макрокоманда представляет значение кода типа файла для FIFO или канала.

9.8.4 Владелец Файла

Каждый файл имеет владельца, который является одним из зарегистрированных имен пользователей, определенных в системе. Каждый файл также имеет группу, которая является одной из определенных групп. Владелец файла может часто быть полезен, но его основная цель - управление доступом.
Владелец файла и группа играет роль в определении доступа, потому что файл имеет набор битов права доступа для пользователя, который является владельцем, другой набор, для тех, кто принадлежат группе владельца файла, и третий набор битов, которые относятся ко всем остальным. См. Раздел 9.8.6 [Право Доступа]
Когда файл создан, его владелец определяется из пользовательского ID процесса, который создает его.
Вы можете изменять владельца и/или группу владельца существующего файла, используя chown функцию. Это - примитив для chown и chgrp команд оболочки.

Прототип для этой функции объявлен в 'unistd.h'.
int chown (const char *filename, uid_t owner, gid_t group) (функция)
Chown функция изменяет владельца filename файла, и группу.
Изменение владельца файла на некоторых системах очищает ID-пользователя и ID-группы биты прав файла. (Потому что эти биты не могут соответствовать новому владельцу.) другие биты прав файла не

изменяются.
Возвращаемое значение - 0 при успехе и -1 при отказе. В дополнение к обычным синтаксическим ошибкам имени файла (см. Раздел 6.2.3 [Ошибки Имени файла]), следующие errno условия ошибки определены для этой функции:
EPERM Этот процесс испытывает недостаток прав, чтобы делать запрошенное изменение.
Только привилегированные пользователи или владелец файла могут изменять группу файла. На большинстве файловых систем, только привилегированные пользователи могут изменять владельца файла; некоторые файловые системы позволяют Вам изменять владельца, если Вы - в настоящее время владелец. Когда Вы обращаетесь к отдаленной файловой системе, поведение, с которым Вы сталкиваетесь, определено системой, которая фактически содержит файл, а не системой, на которой ваша программа выполняется.
См. Раздел 27.7 [Опции для Файлов], для уточнения информации относительно _POSIX_CHOWN_RESTRICTED макрокоманды.
EROFS Файл находится в файловой системе только для чтения.
int fchown (int filedes, int owner, int group) (функция)
Подобна chown, за исключением того, что она изменяет владельца файла на описателе файла filedes.
Возвращаемое значение из fchown - 0 при успехе и -1 при отказе. Следующие errno коды ошибки определены для этой функции:
EBADF filedes аргумент - не допустимый дескриптор файла.
EINVAL filedes аргумент соответствует каналу или гнезду, а не обычному файлу.
EPERM Этот процесс испытывает недостаток прав, чтобы делать запрошенное изменение. Для подробностей, см. chmod, выше.
EROFS Файл постоянно находится в файловой системе только для чтения.

9.8.5 Биты Режима для Прав Доступа

Этот раздел обсуждает биты права доступа, которые управляют чтением и записью в файл. См. Раздел 9.8.3 [Проверка Типа Файла], для уточнения информации относительно кода типа файла.
Все символы, перечисленные в этом разделе определены в файле 'sys/stat.h'.
Эти символические константы определены для битов режима файла, которые управляют правом доступа для файла:

S_IRUSR
S_IREAD
бит права чтения для владельца файла. На многих системах, этот бит - 0400. S_IREAD - устаревший синоним, предусмотрен для BSD совместимости.
S_IWUSR
S_IWRITE
бит права записи для владельца файла. Обычно 0200. S_IWRITE - устаревший синоним, предусмотрен для совместимости с BSD.
S_IXUSR
S_IEXEC
бит права записи для владельца файла. Обычно 0100. S_IWRITE - устаревший синоним, предусмотрен для совместимости cBSD.
S_IRWXU
Это эквивалент ' (S_IRUSR | S_IWUSR | S_IXUSR) '.
S_IRGRP
бит права чтения для владельца группы файла. Обычно 040.
S_IWGRP
бит права записи для владельца группы файла. Обычно 020.
S_IXGRP
бит права выполнения или поиска для владельца группы файла. Обычно 010.
S_IRWXG
Это эквивалент ' (S_IRGRP | S_IWGRP | S_IXGRP) '.
S_IROTH
бит права чтения для других пользователей. Обычно 04.

S_IWOTH
бит права записи для других пользователей. Обычно 02.
S_IXOTH
бит права выполнения или поиска для других пользователей. Обычно 01.
S_IRWXO
Это эквивалент ' (S_IROTH | S_IWOTH | S_IXOTH) '.
S_ISUID
бит выполнения, устанавливающий ID-пользователя, обычно 04000.
S_ISGID
бит выполнения, устанавливающий ID-группы, обычно 02000.
S_ISVTX Это - бит, обычно 01000.
На исполняемом файле, он изменяет политику подкачки системы. Обычно, когда программа завершается, страницы в ядре немедленно освобождены и многократно используются. Если sticky бит установлен на исполняемом файле, система, хранит страницы в ядре некоторое время, как будто программа все еще выполняется. Это выгодно для программы, которая должна быть выполнена много раз последовательно.
На каталоге, sticky бит дает право удалить файл в каталоге, если Вы можете записывать в содержимое этого файла. Обычно, пользователь либо может удалять все файлы в каталоге либо не может удалять никакой из них (имеет ли пользователь право записи для каталога). Липкий бит делает возможным управлять стиранием для индивидуальных файлов.
Фактические битовые значения символов перечислены в таблице выше, так что Вы можете декодировать значения режима файла при отладке ваших программ. Эти битовые значения правильны для большинства систем, но это не гарантируется.
Предупреждение: Запись явных чисел для прав файла - плохая практика. Это не только непереносимо, но также требует от каждого, кто читает вашу программу, помнить то, что означают конкретные биты. Чтобы сделать вашу программу понятной, используйте символические имена.

9.8.6 Как Разрешается Доступ к Файлу

Операционная система обычно разрешает право доступа к файлу, основываясь на ID пользователя и группы процесса, и на дополнительном ID группы, вместе с битами владельца, группы и битами права файла. Эти понятия обсуждены подробно в Разделе 25.2 [Свойства процесса].

Если ID пользователя процесса соответствует ID владельца файла, то права для чтения, записи, и выполнения/поиска, управляются соответствующими 'пользовательскими' (или 'владельца') битами. Аналогично, если любой из ID группы или дополнительной группы процесса соответствует ID группы владельца файла, то права, управляется битами 'группы'. Иначе, права управляются 'другими' битами.
Привилегированные пользователи, подобно 'root', могут обращаться к любому файлу, независимо от битов права файла. Как частный случай, для выполнения файла даже для привилегированного пользователя, по крайней мере один из битов выполнения должен быть установлен.

9.8.7 Назначение Прав Файла

Примитивные функции для создания файлов (например, open или mkdir) воспринимают аргумент mode, который определяет права файла. Но заданный mode изменяется маской создания файла процесса, или перед использованием.
Биты, которые установлены в маске создания файла, идентифицируют права, которые должны всегда быть заблокированы для новых файлов. Например, если Вы устанавливаете все 'другие' биты доступа в маске, то новые файлы не доступны вообще для процессов в 'другом' классе, даже если аргумент mode, заданный в функции создания разрешил такой доступ. Другими словами, маска создания файла - дополнение обычных прав доступа, которые Вы хотите предоставить.
Программы которые создают файлы, обычно определяют аргумент mode, который включает все права, которые имеют смысл для специфического файла. Для обычного файла, это обычно право чтения и право записи для всех классов пользователей. Эти права ограничены как определено собственной маской создания файла индивидуального пользователя.
Чтобы изменять право существующего данного файла, вызовите chmod. Эта функция игнорирует маску создания файла; она использует только заданные биты права.

При нормальном использовании, маска создания файла инициализируется при входе пользователя в систему (использованием umask команды оболочки), и наследуется всеми подпроцессами. Прикладные программы обычно не должны заботиться о маске создания файла.

Когда ваша программа должна создать файл и обходить umask для прав доступа, самый простой способ сделать это состоит в том, чтобы использовать fchmod после открытия файла, а не изменять umask.

Фактически, изменение umask обычно делается только оболочками. Они используют umask функцию.

Функции в этом разделе объявлены в 'sys/stat.h'.

mode_t umask (mode_t mask) (функция)

Umask функция устанавливает маску создания файла текущего процесса, и возвращает предыдущее значение маски создания файла.

Вот пример, показывающий, как читать маску с umask без ее замены:

```
mode_t
read_umask (void)
{
        mask = umask (0);
        umask (mask);
}
```

mode_t getumask (void) (функция)

Возвращает текущее значение маски создания файла для текущего процесса. Эта функция - расширение GNU.

int chmod (const char *filename, mode_t mode) (функция)

Chmod функция устанавливает биты права доступа для файла, именованного filename как mode.

Если filename называет символическую ссылка, chmod изменяет право файла, указанного ссылкаю, а не связи непосредственно. Не имеется фактически никакого способа установить mode связи, который всегда - 1.

Эта функция возвращает 0 в случае успеха и -1 если нет. В дополнение к обычным синтаксическим ошибкам имени файла (см. Раздел 6.2.3 [Ошибки Имени файла]), следующие errno условия ошибки определены для этой функции:

ENOENT Именованный файл не существует.

EPERM Этот процесс не имеет права изменить право доступа этого файла. Только владелец файла или привилегированный пользователь может изменять их.

EROFS Файл постоянно находится в файловой системе только для чтения.

int fchmod (int filedes, int mode) (функция)

Подобна chmod, за исключением того, что она изменяет права файла в настоящее время открытого через дескриптор filedes.

Возвращаемое значение из fchmod - 0 при успехе и -1 при отказе. Следующие errno коды ошибки определены для этой функции:

EBADF filedes аргумент - не допустимый дескриптор файла.

EINVAL filedes аргумент соответствует каналу или гнезду, или кое-чему еще, которое не имеет права доступа.

EPERM Этот процесс не имеет права изменить право доступа этого файла. Только владелец файла или привилегированный пользователь может изменять их.

EROFS Файл постоянно находится в файловой системе только для чтения.

9.8.8 Тестирование Прав для Обращения к Файлу

Когда программа выполняется привилегированным пользователем, это разрешает ей обращаться к файлам без ограничений на пример, изменять ' /etc/passwd '. Программы разработанные, чтобы быть выполненным обычными пользователями, но обращаться к таким файлам используют setuid так, чтобы они всегда выполнялись под root'ом.

Программа должна явно проверить, имел ли пользователь необходимый доступ к файлу, прежде, чем она начнет читать или писать в файл.

Для этого используйте функции доступа, которые проверяет право доступа, основанное на ID пользователя. (setuid не

изменяет реального ID пользователя, так что это отражает пользователя, который фактически выполнил программу.)

Имеется другой способ, которым Вы могли бы проверить доступ, который является простым для описания, но очень интенсивно используемым. Нужно исследовать биты mode файла и подражать вычислению доступа системы. Этот метод нежелателен, потому что много систем имеют дополнительные возможности управления доступом; ваша программа не может корректно подражать им. Использование access просто и автоматически делает то, что соответствует системе, которую Вы используете.

Символы в этом разделе объявлены в ' unistd.h '.

int access (const char *filename, int how) (функция)

Функция выясняет, можно ли к файлу, именованному filename обращаться способом, заданным аргументом how. Этот аргумент может быть либо поразрядным ИЛИ флагов R_OK, W_OK, X_OK, либо проверка существования F_OK.

Эта функция использует ID пользователя и группы процесса, а не эффективный ID, для проверки права доступа. В результате, если Вы используете функцию из программы setuid или setgid (см. Раздел 25.4 [Как Изменить Права]), это дает информацию относительно пользователя, кто фактически выполнил программу.

Возвращаемое значение - 0, если доступ разрешается, и -1 иначе. (Другими словами, обрабатываемая как функция предиката, access возвращает истину, если запрошенный доступ отклонен.)

В дополнение к обычным синтаксическим ошибкам имени файла (см. Раздел 6.2.3 [Ошибки Имени файла]), следующие errno условия ошибки определены для этой функции:

EACCES доступ, заданный how, отклонен.

ENOENT файл не существует.

EROFS Право записи было запрошено для файла в файловой системе только для чтения.

Эти макрокоманды определены в заглавном файле 'unistd.h' для использования как аргументы функции access. Значения - константы integer.

int R_OK (макрос)
Аргумент проверки на право чтения.

int W_OK (макрос)
Аргумент проверки на право записи.

int X_OK (макрос)
Аргумент проверки направо выполнения/поиска.

int F_OK Macro
Аргумент проверки на существование файла.

9.8.9 Временные Характеристики Файла

Каждый файл имеет три временных метки, связанные с ним: время доступа, время изменения, и время изменения атрибута. Они соответствуют st_atime, st_mtime, и st_ctime элементам структуры stat; см. Раздел 9.8 [Атрибуты Файла].

Все эти времена представляются в календарном формате времени, как объекты time_t. Этот тип данных определен в 'time.h'. Для получения более подробной информации и манипулирования значениями времени см. Раздел 17.2 [Календарное Время].

Когда существующий файл открыт, его атрибуты, такие как время изменения, модифицируются. Чтение из файла модифицирует атрибут времени доступа, а запись модифицирует время изменения.

Когда файл создан, все три временных метки для этого файла установлены на текущее время. Кроме того, атрибут времени изменения каталога, который содержит новый вход, модифицируются.

Добавление нового имени для файла с функцией связи модифицирует атрибут поля времени изменения связываемого файла, и соответствующие атрибуты каталога, содержащего новое имя. Те же самые поля изменяются если имя файла удалено с unlink, remove, или rmdir. Переименование файла с rename воздействует только на атрибут времени изменения и изменения поля времени двух родительских включающих каталогов, а не на времена переименовываемого файла.

Изменение атрибутов файла (например, с chmod) модифицирует атрибут времени изменения.

Вы можете также изменять некоторые из временных меток файла,

явно используются utime, за исключением изменения атрибута времени изменения. Вы должны включить заглавный файл 'utime.h' чтобы использовать это средство.

struct utimbuf (тип данных)

Структура utimbuf используется с функцией utime, чтобы определить новый доступ или изменить времена для файла. Она содержит следующие элементы:

time_t actime

Это - время доступа(последнего) к файлу.

time_t modtime

Это - время изменения файла.

int utime (const char *filename, const struct utimbuf *times) (функция)

Эта функция используется, чтобы изменить файловые времена, связанные с файлом, именованным filename.

Если times являются пустым указателем, то время доступа и изменения файла устанавливаются на текущее время. Иначе, они устанавливаются как значения из actime и modtime элементов

(соответственно) структуры utimbuf, указанной times.

Utime функция возвращает 0 если обращение успешно и -1 при отказе. В дополнение к обычным синтаксическим ошибкам имени файла, следующие errno условия ошибки определены для этой функции:

EACCES имеется проблема права в случае, где пустой указатель был передан как аргумент времена. Чтобы модифицировать временную метку на файле, Вы должны также быть владельцем файла и иметь право записи в файл, или быть привилегированным пользователем.

ENOENT файл не существует.

EPERM Если аргумент times - не пустой указатель, Вы должны также быть владельцем файла или привилегированным пользователем.

EROFS Файл живет в файловой системе только для чтения.

Каждая из трех временных меток имеет соответствующую часть измеряющую микросекунды, которая расширяет разрешающую способность. Эти поля называются st_atime_usec, st_mtime_usec, и st_ctime_usec; каждое имеет значение от 0 до 999,999, которое указывает время в микросекундах. Они соответствуют tv_usec полю структуры timeval; см. Раздел 17.2.2 [Календарь с высоким разрешением].

Utimes функция - подобна utime, но также позволяет Вам определять дробную часть времен файла.

Прототип для этой функции находится в файле 'sys/time.h'.

int utimes (const char *filename, struct timeval tvp[2]) (функция)

Эта функция устанавливает доступ к файлу и времена изменения для файла, именованного filename.

Новый время доступа определено tvp [0], а новое время изменения tvp [1]. Эта функция происходит из BSD.

Возвращаемые значения и условия ошибки - такие же как для utime функции.

9.9 Создание Специальных Файлов

Mknod функция - примитив для создания специальных файлов, типа файлов, которые соответствуют устройствам. Библиотека GNU включает эту функцию для совместимости с BSD.

Прототип для mknod объявлен в 'sys/stat.h'.

int mknod (const char *filename, int mode, int dev) (функция)

Mknod функция создает специальный файл с именем filename. Mode определяет режим файла, и может включать различные специальные биты

файла, типа S_IFCHR (для символьного специального файла) или S_IFBLK (для блокированного специального файла). См. Раздел 9.8.3 [Тестирование Типа Файла].

Dev аргумент определяет, к которому устройству обращается специальный файл. Точная интерпретация зависит от вида создаваемого специального файла.

Возвращаемое значение - 0 при успехе и -1 при ошибке. В дополнение к обычным синтаксическим ошибкам имени файла (см. Раздел 6.2.3 [Ошибки Имени Файла]), следующие errno условия ошибки определены для этой функции:

EPERM Процесс вызова не привилегирован. Только суперпользователь может создавать специальные файлы.

ENOSPC каталог или файловая система, которая содержала бы,

новый файл 'полна' и не может быть расширена.
EROFS Каталог, содержащий новый файл не может изменяться, потому что он находится в файловой системе только для чтения.
EEXIST уже имеется файл, именованный filename. Если Вы хотите заменить этот файл, Вы должны сначала удалить старый файл.

9.10 Временные Файлы

Если Вы должны использовать временный файл в вашей программе, Вы можете использовать tmpfile функцию для его открытия. Или Вы можете использовать tmpnam функцию, что бы сделать имя для временного файла и тогда открыть его обычным способом через fopen.
Tempnam функция - подобна tmpnam, но допускает выбирать в какой каталог войдут временные файлы, и кое-что относительно того, на что их имена будут походить.
Эти средства объявлены в заглавном файле 'stdio.h'.
FILE * tmpfile (void) (функция)
Эта функция создает временный двоичный файл для режима модификации, как будто, вызывая fopen с режимом ' wb + '. Файл удаляется автоматически, когда он закрыт или когда программа завершается. (На некоторых других системах ANSI C файл может не быть удаленным, если программа завершается неправильно).
char * tmpnam (char *result) (функция)
Эта функция создает и возвращает имя файла, которое является допустимым именем файла, и не называет никакой существующий файл. Если аргумент result является пустым указателем, возвращаемое

значение - указатель на внутреннюю статическую строку, которая могла бы изменяться последующими обращениями. Иначе, аргумент result должен быть указателем на массив по крайней мере из L_tmpnam символов, и результат будет написан в этот массив.
Возможно tmpnam будет терпеть неудачу, если Вы вызываете ее слишком много раз. Потому что фиксированная длина временного имени файла дает участок памяти для только конечного числа различных имен. Если tmpnam сбоит, она возвращает пустой указатель.
int L_tmpnam (макрос)
Значение этой макрокоманды - константное выражение integer, которое представляет минимальный размер резервирования строки, достаточно большой, чтобы содержать имя файла, сгенерированное tmpnam функцией.
int TMP_MAX (макрос)
Макрокоманда TMP_MAX - нижняя граница для того, сколько временных имен Вы можете создавать с tmpnam. Вы можете полагаться на способность вызвать tmpnam по крайней мере столько раз прежде, чем она будет терпеть неудачу, говоря что Вы сделали слишком много временных имен файла.
С библиотекой GNU, Вы можете создавать очень большое количество временных имен файла, если Вы фактически создаете файлы, возможно дисковое пространство закончится прежде, чем закончатся имена. Некоторые другие системы имеют фиксированное, малое ограничение числа временных файлов. Ограничение никогда не меньше чем 25.
char * tempnam (const char *dir, const char *prefix) (функция)
Эта функция генерирует уникальное временное имя файла. Если prefix не пустой указатель, то до пяти символов этой строки используется как prefix для имени файла.
Prefix каталога для временного имени файла определяется проверкой следующей последовательности. Каталог должен существовать и быть перезаписываемым.
* Переменная среды TMPDIR, если она определена.
* Аргумент dir, если это - не пустой указатель.
* Значение P_tmpdir макрокоманды.
* Каталог ' /tmp '.
Эта функция определена для SVID совместимости.

char * P_tmpdir (SVID макрос)
Эта макрокоманда - имя заданного по умолчанию каталога для временных файлов.
Более старые системы UNIX не имели таких функций. Взамен они использовали mktemp и mkstemp. Обе из этих функций работают изменяя строку шаблона имени файла, заданную Вами. Последние шесть символов

этой строки должны быть ' XXXXXX '. Эти шесть X-ов заменятся на шесть символов, которые делают целую строку уникальным именем файла. Обычно строка шаблона - это что-нибудь вроде ' /tmp/prefixXXXXXX ', и каждая программа использует уникальный prefix.

Обратите внимание: Т. к. mktemp и mkstemp изменяют строку шаблона, Вы не должны передать строковые константы им. Строковые константы - обычно в памяти только для чтения, так что ваша программа разрушилась бы, если бы mktemp или mkstemp пробовали изменять строку.

char * mktemp (char *template) (функция)

Mktemp функция генерирует уникальное имя файла, изменяя шаблон как описано выше. В случае успеха она возвращает модифицированный шаблон. Если mktemp не находит уникальное имя файла, она делает шаблон пустой строкой и возвращает ее. Если шаблон не заканчивается на ' XXXXXX ', mktemp возвращает пустой указатель.

int mkstemp (char *template) (функция)

Mkstemp функция генерирует уникальное имя файла, точно так же как mktemp, но она также открывает файл для Вас через open (см. Раздел 8.1 [Открытие и Закрытие Файлов]). В случае успеха, она изменяет шаблон на месте и возвращает дескриптор файла открытый на этом файле для чтения и записи. Если mkstemp не может создать однозначно названный файл, она делает шаблон пустой строкой и возвращает -1. если шаблон не заканчивается на ' XXXXXX ', mkstemp возвращает -1 и не изменяет шаблон.

В отличие от mktemp, mkstemp, как гарантируют, создаст уникальный файл, который не может сталкиваться с любой другой программой, пробующей создать временный файл. Потому что она работает, вызывая open с O_EXCL битом флага, который говорит, что Вы хотите всегда создавать новый файл, и получать ошибку, если файл уже существует.

10. Каналы и FIFO

Канал - механизм для связи между процессами; данные, записывающиеся в канал одним процессом могут читаться другим процессом. Данные обрабатываются в порядке 'первым пришел' - 'первым ушел' (FIFO). Канал не имеет никакого имени; он создан для одного использования, и оба конца должны быть унаследованы от одиночного процесса, который создал канал.

FIFO специальный файл является подобным каналу, но вместо анонимного, временного соединения, FIFO имеет имя или имена подобно любому другому файлу. Процесс открывает FIFO по имени, чтобы связаться через него.

Канал или FIFO должен быть открыт с обоих концов одновременно. Если Вы читаете из канала или файла FIFO, в который никто ничего не пишет (возможно потому что, они все закрыли файл, или вышли), то чтение возвращает конец файла. Запись в канал или FIFO, который не имеет процесс считывания, обрабатывается как условие ошибки; это генерирует сигнал SIGPIPE, и сбои с кодом ошибки EPIPE, если сигнал обработан или блокируется.

Ни каналы ни FIFO специальные файлы не позволяют позиционирование файла. И чтение и запись происходит последовательно; чтение из начала файла и запись в конец.

10.1 Создание Канала

Примитив для создания канала - функция pipe. Она создает оба, и чтения и записи концы канала. Это не очень полезно для одиночного процесса, использовать канал, чтобы разговаривать с собой. В типичном использовании, процесс создает канал только прежде, чем он ветвится на один или более дочерних процессов (см. Раздел 23.4 [Создание Процесса]). Канал используется для связи или между родителем или дочерними процессами, или между двумя процессами братьями.

Функция pipe объявлена в заглавном файле 'unistd.h'.

int pipe (int filedes[2]) (функция)

Функция pipe создает канал и помещает дескрипторы файла для чтения и записи (соответственно) в filedes [0] и filedes [1].

При успехе pipe возвращает значение 0. При отказе, -1. Следующие errno условия ошибки определены для этой функции:

```
      EMFILE процесс имеет слишком много файлов открытыми.
      ENFILE имеются слишком много открытых файлов во всей системе.
```
См. Раздел 2.2 [Коды Ошибки], для получения более подробной информации о ENFILE.

Вот пример простой программы, которая создает канал. Эта программа использует функцию ветвления (см. Раздел 23.4 [Создание Процесса]) чтобы создать дочерний процесс. Родительский процесс напишет данные, которые читается дочерним процессом.

```
                #include
                #include
                #include
                #include
                void
                read_from_pipe (int file)
                {
                        FILE *stream;
                        int c;
                        stream = fdopen (file, 'r');
                        while ((c = fgetc (stream)) != EOF)
                                putchar (c);
                        fclose (stream);
                }
        /* Пишем некоторый произвольный текст в канал. */
                void
                write_to_pipe (int file)
                {
                        FILE *stream;
                        stream = fdopen (file, 'w');
                        fprintf (stream, 'hello, world!\n');
                        fprintf (stream, 'goodbye, world!\n');
                        fclose (stream);
                }
                int
                main (void)
                {
                        pid_t pid;
                        int mypipe[2];
                        /* Create the pipe. */
```



```
                        if (pipe (mypipe))
                                {
                                        fprintf(stderr,'Pipe failed.\n');
                                        return EXIT_FAILURE;
                                }
                        /* Создаем дочерний процесс. */
                        pid = fork ();
                        if (pid == (pid_t) 0)
                                {
                                        /* Это - дочерний процесс. */
                                        read_from_pipe (mypipe[0]);
                                        return EXIT_SUCCESS;
                                }
                        else if (pid < (pid_t) 0)
                                {
                                        /* The fork failed. */
                                        fprintf(stderr,'Fork failed.\n');
                                        return EXIT_FAILURE;
                                }
                        else
                                {
                                        /* Это - родительский процесс. */
                                        write_to_pipe (mypipe[1]);
                                        return EXIT_SUCCESS;
                                }
                }
```

10.2 Канал к Подпроцессу

Общее использование каналов должно послать данные к или получать данные из программы, выполняемой как подпроцесс.

Один из способов выполнения этого - использовать комбинацию pipe (чтобы создать канал), fork (чтобы создать подпроцесс), dup2 (чтобы вынудить подпроцесс использовать pipe как стандартный ввод

или канал вывода), и exec (чтобы выполнить новую программу). Или, Вы можете использовать popen и pclose.

Преимущество использования popen и pclose - в том, что интерфейс является намного более простым и более удобным для использования. Но они не предлагают так много гибкости, как использование функций

низкого уровня непосредственно.

FILE * popen (const char *command, const char *mode) (функция)

Popen функция близко связана с функцией системы; см. Раздел 23.1 [Выполнение Команд]. Она выполняет команду оболочки как подпроцесс.

Однако, вместо того, чтобы ждать завершения команды, она создает канал к подпроцессу и возвращает поток, который соответствует этому каналу.

Если Вы определяете аргумент режима 'r', Вы можете читать из потока, чтобы отыскать данные из канала стандартного вывода подпроцесса. Подпроцесс наследует канал стандартного ввода из родительского процесса.

Аналогично, если Вы определяете аргумент режима 'w', Вы можете писать в поток, чтобы посылать данные на канал стандартного ввода подпроцесса. Подпроцесс наследует канал стандартного вывода из родительского процесса.

В случае ошибки, popen возвращает пустой указатель. Это может случаться, если канал или поток не может быть создан, если подпроцесс не может быть раздвоен, или если программа не может быть выполнена.

int pclose (FILE *stream) (функция)

Pclose функция используется, чтобы закрыть поток, созданный popen. Она ждет завершения дочернего процесса, и возвращает значение состояния, что касается функции системы.

Вот пример, показывающий, как использовать popen и pclose, чтобы фильтровать вывод через другую программу.

```
#include
#include
void
write_data (FILE * stream)
{
        int i;
        for (i = 0; i < 100; i++)
                fprintf (stream, '%d\n', i);
        if (ferror (stream))
                {
                        fprintf (stderr, 'Output to
                        stream failed.\n');
                        exit (EXIT_FAILURE);
```



```
                }
}
int
main (void)
{
        FILE *output;
        output = popen ('more', 'w');
        if (!output)
                {
                        fprintf(stderr,'Could not run
                                        more.\n');
                        return EXIT_FAILURE;
                }
        write_data (output);
        pclose (output);
        return EXIT_SUCCESS;
}
```

10.3 FIFO Специальные Файлы

FIFO специальный файл подобен каналу, за исключением того, что он создан различным способом. Вместо анонимного канала связи, FIFO специальный файл введен в файловую систему, вызовом mkfifo.

Если Вы создали FIFO специальный файл таким образом,, любой процесс может открывать его для чтения или записи, таким же образом как обычный файл. Однако, он должен быть открыт в оба конца

одновременно прежде, чем Вы можете делать любой ввод или вывод на нем. Открытие FIFO для чтения обычно блокируется, пока некоторый другой процесс не открывает тот же самый FIFO для записи, и наоборот.

Mkfifo функция объявлена в заглавном файле 'sys/stat.h'.

int mkfifo (const char *filename, mode_t mode) (функция)

Mkfifo функция делает FIFO специальный файл с именем filename. Аргумент mode используется, чтобы установить права файла; см. Раздел 9.8.7 [Установка Прав].

Нормальное, успешное возвращаемое значение из mkfifo - 0. В случае ошибки возвращается -1. В дополнение к обычным синтаксическим ошибкам имени файла следующие errno условия ошибки

определены для этой функции:

EEXIST именованный файл уже существует.

ENOSPC каталог или файловая система не может быть расширен.

EROFS каталог, который содержал бы файл постоянно находится в файловой системе только для чтения.

10.4 Быстрота ввода-вывода Канала

Чтение или запись данных в канал мгновенны, если размер данных меньше чем PIPE_BUF. Это означает что передача данных кажется мгновенной, в этом случае ничто в системе не может наблюдать состояние, в котором он является частично полным. Быстрый ввод - вывод не может начинаться сразу же (может требоваться ждать пространство буфера или для данных), но если только он начинается, то он заканчивается немедленно.

Чтение или запись большего количества данных может не быть быстрым; например, выходные данные из других процессов, совместно использующих дескриптор могут быть разбиты на части.

См. Раздел 27.6 [Ограничения для Файлов], для уточнения информации относительно параметра PIPE_BUF.

11. Гнезда

Эта глава описывает средства GNU для межпроцессорной связи, используя гнезда.

Гнездо - обобщенный межпроцессорный канал связи. Подобно каналу, гнездо представляется как дескриптор файла. Но, в отличие от каналов, гнезда поддерживает ссылка между несвязанными процессами, и даже между процессами, выполняющимися на различных машинах, которые связываются по сети. Гнезда - первичный способ связи с другими машинами; telnet, rlogin, ftp, переговоры, и другие сетевые программы используют гнезда.

Не все операционные системы поддерживают гнезда. В библиотеке GNU, заглавный файл 'sys/socket.h' существует независимо от операционной системы, и функции гнезд всегда существуют, но если система действительно не поддерживает гнезда, эти функции всегда терпят неудачу.

Незавершенность: Мы в настоящее время не описали средства для передачи сообщений или для конфигурирования интерфейса Internet.

11.1 Понятие Гнезда

Когда Вы создаете гнездо, Вы должны определить стиль связи, который Вы хотите использовать и тип протокола, который должен поддерживать ее. Стиль связи гнезда определяет семантику пользовательского уровня посылки и получения данных через гнезда. Выбор стиля связи определяет ответы на вопросы типа:

* Каковы модули передачи данных? Некоторые стили связи расценивают данные как последовательность байтов, без большей структуры; другие группируют байты в записи (которые известны в этом контексте как пакеты).

* Могут данные быть потеряны в течение нормальной операции? Некоторые стили связи гарантируют, что все посланные данные прибывают в порядке, как они были посланы (страховка системы или сетевых сбоев); другие стили иногда теряют данные как нормальная часть операции, и могут иногда поставлять пакеты больше чем один раз или в неправильном порядке.

* Является ли ссылка полностью с одним партнером? Некоторые стили связи - подобно телефонному звонку, Вы делаете соединение с одним отдаленным гнездом, и тогда свободно обмениваетесь данными. Другие

стили - подобно отправке по почте символов, Вы определяете адрес адресата для каждого сообщения, которое Вы посылаете.

Вы должны также выбрать именное пространство для наименования гнезда. Имя гнезда ('адрес') значимо только в контексте частного namespace. Фактически, даже тип данных, используемый для имени гнезда может зависеть от именного. Именные пространства также называются 'областями', но мы избегаем этого слова, поскольку оно может быть спутано с другим использованием того же самого термина. Каждое именное пространство имеет символическое имя, которое начинается с 'PF_'. Соответствующее символическое имя, начинающееся с 'AF_' обозначает формат адреса для этого namespace.

В заключение Вы должны выбрать протокол, чтобы установить связь. Протокол определяет какой механизм низкого уровня используется, чтобы передавать и получить данные. Каждый протокол допустим для определенного именного пространства и стиля связи; именное пространство иногда называется совокупностью протоколов из-за этого имена именных пространств начинаются с 'PF_'.

Правила протокола относятся к данным, передающимся между двумя программами, возможно на различных компьютерах; большинство этих

правил обработано операционной системой, и Вы не нужно знать о них. Вот что Вы должны знать относительно протоколов:

* Чтобы иметь связь между двумя гнездами, они должны определить тот же самый протокол.
* Каждый протокол значим со специфическим стилем/именным пространством и не может использоваться с несоответствующими комбинациями. Например, TCP протокол удовлетворяет только стилю связи потока байтов и именному пространству Internet.
* Для каждой комбинации стиля и именного пространства, имеется заданный по умолчанию протокол, который Вы можете запрашивать, определяя 0 как номер протокола. И это - то, что Вы должны обычно делать - использовать значение по умолчанию.

11.2 Стили Связи

Библиотека GNU поддерживает различные виды сокетов. В этом разделе описываются различные типы сокетов предоставляемые в бибилотекой GNU. Упомянутые в данном разделе символические константы определены в "sys/socket.h".

int SOCK_STREAM (макрос)

Тип сокета SOCK_STREAM предназначен для передачи потоков байтов. Такой стиль передачи схож со стилем передачи данных через каналы (pipe) (см. Главу 10 [Трубопроводы и FIFO]). Он используется для передачи данных с отдаленным соединением. При таком стиле передачи данных обеспечивается высокая надежность.

Более подробное описание работы с использованием этого типа Вы найдете в Разделе 11.8 [Соединения].

int SOCK_DGRAM (макрос)

Тип сокета SOCK_DGRAM используется для посылки индивидуально адресованных пакетов (ненадежен). Этот тип принципиально отличается от типа SOCK_STREAM.

Каждый раз когда Вы пишите данные в сокет этого типа, данные становятся одним пакетом. Вы должны определить адрес получателя для каждого пакета.

Единственной гарантией, которую предоставляет Вам система относительно запросов передачи данных, является то, что она пробует наилучшим образом

посылать каждый пакет. У нее может получится посылка шестого пакета после неудачи с четвертыми и пятыми пакетами; седьмой пакет может прибыть перед шестым, также второй может прибыть как раз после шестого.

Типичное использование типа SOCK_DGRAM в ситуациях, где есть возможность повторной посылки пакетов, если ответ не был получен в приемлемое время.

int SOCK_RAW (макрос)

Этот тип обеспечивает доступ к сетевым протоколам и интерфейсам низкого уровня. Обычные пользовательские программы, обычно не имеют потребности использовать этот стиль.

11.3 Адреса сокетов

Имя сокета обычно называется адресом. Функции и символы для имеющихся адресов сокетов могли называться как с использованием термина, так и использованием термина "адрес". Вы можете расценивать эти термины как синонимичные в контексте обсуждения сокетов.

Сокет, созданный при помощи функции socket, не имеет никакого адреса. Другие процессы могут использовать его для связи только после того как Вы дадите ему адрес. Мы называем это - связывание адреса с сокетом. Связывание происходит при помощи функции bind.

В первый раз, когда Вы посылаете данные из сокета, или используете его, чтобы инициализировать соединение, система назначает адрес автоматически.

Подробности адресации сокетов изменяются, в зависимости от именного пространства, которое Вы используете. См. Раздел 11.4 [Именное пространство Файла], или Раздел 11.5 [Именное пространство Internet].

Независимо от именного пространства, Вы используете те же самые функции bind и getsockname, чтобы установить и исследовать адрес гнезда.

11.3.1 Форматы Адреса

Функции bind и getsockname используют обобщенный тип данных struct sockaddr *, чтобы представить указатель на адрес гнезда. Вы не можете использовать этот тип данных действительно, чтобы интерпретировать адрес или создавать его; для этого, Вы должны использовать соответствующий тип данных для именного пространства гнезда.

Таким образом, обычная нужно создать адрес в соответствующем именном пространстве специфического типа, и приводить указатель на struct sockaddr *, когда Вы вызываете bind или getsockname.

Единственная информация, которую Вы можете получить из структуры sockaddr - указатель формата адреса, который сообщает Вам какой тип данных использовать, чтобы понять адрес полностью.

Символы в этом разделе определены в заголовочном файле "sys/socket.h".

struct sockaddr (тип данных)

Тип структуры sockaddr непосредственно имеет следующие поля:

short int sa_family

Это код для формата адреса. Он идентифицирует формат данных.

char sa_data [14]

Это фактические данные адреса сокета, которые являются формато-зависимыми. Длина также зависит от формата, и может быть больше чем 14. Длина 14 из sa_data по существу произвольна.

Каждый формат адреса имеет символическое имя, которое начинается с "AF_ ". Каждый из них соответствует "PF_ " символу, который обозначает соответствующее именное пространство. Вот список названий форматов адресов:

AF_FILE Обозначает формат адреса, который идет с именным пространством файла. (PF_FILE - имя этого именного пространства.) См. Раздел 11.4.2 [Подробности Именного пространства Файла], для уточнения информации относительно этого формата адреса.

AF_UNIX Это синоним AF_FILE, для совместимости. (PF_UNIX - аналогично синоним для PF_FILE.)

AF_INET Обозначает формат адреса, который идет в именном пространстве Internet. (PF_INET - имя этого именного пространства.) См. Раздел 11.5.1 [Формат Адреса Internet].

AF_UNSPEC

Не обозначает никакой специфический формат адреса. Он используется только в редких случаях, когда необходимо очистить снаружи заданный по умолчанию адрес адресата от "соединенного" датаграмного сокета. См. Раздел 11.9.1 [Посылка Датаграмм].

Соответствующий символ указателя именного пространства PF_UNSPEC существует для законченности, но нет никакой причины использовать его в программе.

"Sys/socket.h" определяет символы, начинающиеся с " AF_ " для различных видов сетей, большинство из которых фактически не встречается. Мы будем документировать только то, что действительно используется на практике.

11.3.2 Установка Адреса сокета

Используйте функцию bind, чтобы для связывания адреса сокета. Прототип для bind находится в заголовочном файле "sys/socket.h". Для примеров использования см. Раздел 11.4 [Именное пространство Файла].

int bind (int socket, struct sockaddr *addr, size_t length) (функция)

Функция bind назначает адрес сокет socket. Аргументы Addr и length определяют адрес; детализированный формат адреса зависит от именного пространства. Первая часть адреса - всегда указатель формата, который определяет именное пространство, и говорит, что адрес находится в формате для этого именного пространства.

Возвращаемое значение - 0 при успехе и -1 при отказе. Для этой функции в переменной errno определены следующие виды ошибок:

EBADF аргумент - не допустимый описатель файла.
ENOTSOCK дескриптор socket - сокет.
EADDRNOTAVAIL заданный адрес не доступен на этой машине.
EADDRINUSE Существует другой сокет использующий заданный адрес.

EINVAL сокет уже имеет адрес.
EACCESS Вам не достаточно прав для обращения к запрошенному адресу. (В области Internet, только супер-пользователю позволяют определить номер порта в диапазоне от 0 до IPPORT_RESERVED минус один; см. Раздел 11.5.3 [Порты].) Дополнительные условия могут быть возможны в зависимости от специфического именного пространства

сокета.

11.3.3 Чтение Адреса сокета

Используйте функцию getsockname, чтобы исследовать адрес гнезда Internet. Прототип для этой функции находится в заголовочном файле "sys/socket.h".

int getsockname (int socket, struct sockaddr *addr, size_t *length_ptr) (функция)

Функция getsockname возвращает информацию относительно адреса сокета в заданного аргументами addr и length_ptr. Обратите внимание, что length_ptr - указатель; Вы должны инициализировать его, как размер резервирования addr, и по возвращении он содержит фактический размер данных адреса.

Формат данных адреса зависит от именного пространства сокета. Длина информации обычно устанавливается для данного именного пространства, так что обычно Вы можете знать точно, сколько места необходимо. Обычно нужно зарезервировать место для значения, используя соответствующий тип данных для именного пространства сокета, и тогда привести адрес к struct sockaddr *, чтобы передать его getsockname.

Возвращаемое значение - 0 при успехе и -1 при ошибке. Для этой функции в переменной errno определены следующие виды ошибок:

EBADF аргумент socket - не допустимый описатель файла.
ENOTSOCK дескриптор socket - не сокет.
ENOBUFS не имеется достаточных внутренних буферов, доступных для операции.

Вы не можете читать адрес сокета в именном пространстве файла. Это непротиворечиво с остальной частью системы; вообще, не существует способа найти имя файла из описателя для этого файла.

11.4 Именное пространство Файла

Этот раздел описывает подробности именного пространства файла, чье символическое имя (требуется, когда Вы создаете сокет) - PF_FILE.

11.4.1 Понятия Именного пространства Файла

В именном пространстве файла, адреса сокетов - имена файлов. Вы можете определять любое желаемое имя файла для адреса сокета, но Вы должны иметь право записи в каталоге, содержащем его. Для чтобы соединяться с сокетом, Вы должны иметь право чтения для него.
Обычно эти файлы помещаются в каталог `/tmp'.

Одна особенность именного пространства файла -- имя используется только при открытии соединения; если только оно было законченно, адрес не значим и может не существовать.

Другая особенность заключается в том, что Вы не можете соединяться с таким сокетом на другой машине, даже если другая машина совместно использует файловую систему, которая содержит имя это имя сокета. Вы можете видеть сокет в распечатке каталога, но соединение с ним никогда не произойдет.

После того, как Вы закрываете сокет в именном пространстве файла, Вы должны удалить имя файла из файловой системы. Используйте unlink или remove, чтобы делать это; см. Раздел 9.5 [Удаление Файлов].

Именное пространство файла поддерживает только один протокол для любого типа связи; 0 - номер протокола.

11.4.2 Подробности Именного пространства Файла

Чтобы создавать сокет в именном пространстве файла, используйте константу PF_FILE как аргумент именного пространства для socket или socketpair. Эта константа определена в "sys/socket.h".

int PF_FILE (макрос)

Он обозначает именное пространство файла, в котором адреса сокетов являются именами файлов, и связываются совокупностью протоколов.

int PF_UNIX (макрос)

Это - синоним PF_FILE используемый для совместимости.

Структура для определения имен сокетов в именном пространстве файла определена в заголовочном файле "sys/un.h":

struct sockaddr_un (тип данных)

Эта структура используется, чтобы определить адреса сокета именного пространства файла. Она имеет следующие поля:

short int sun_family

Это поле идентифицирует совокупность адреса или формат адреса сокета. Вы должны сохранить значение AF_FILE, чтобы обозначить именное пространство файла. См. Раздел 11.3 [Адреса Гнезда].

char sun_path[108]

Это имя используемого файла.

Незавершенность: Почему - 108? RMS предлагает делать его массивом нулевой длины и использовать alloc, чтобы зарезервировать соответствующее количество памяти, основываясь на длине filename.

Вы должны вычислить параметр длины для адреса сокета в именном пространстве файла как сумму размера компоненты sun_family и длины (не размера резервирования!) строки имени файла.

11.4.3 Пример файлового-именного пространства сокетов

Вот пример, показывающий, как создавать и связывать сокет в именном пространстве файла.

```
#include
#include
#include
#include
#include
#include
int
make_named_socket (const char *filename)
{
        struct sockaddr_un name;
        int sock;
        size_t size;
        sock = socket (PF_UNIX, SOCK_DGRAM, 0);
        if (sock < 0)
                        {
                                perror ("socket");
                                exit (EXIT_FAILURE);
                        }
        name.sun_family = AF_FILE;
        strcpy (name.sun_path, filename);
```

```
                        size=(offsetof(struct sockaddr_un, sun_path)
                                + strlen (name.sun_path) + 1);
                        if (bind (sock, (struct sockaddr *) &name,
                                        size) < 0) {
                                        perror ("bind");
                                        exit (EXIT_FAILURE);
                                }
                        return sock;
      }
```


11.5 Именное пространство Internet

Этот раздел описывает подробности протокола и соглашений именования сокетов, используемые в именном пространстве Internet.

Чтобы создать сокет в именном пространстве Internet, используйте символическое имя PF_INET этого именного пространства как аргумент именного пространства socket или socketpair. Эта макрокоманда определена в "sys/socket.h".

```
      int PF_INET         (макрос)
```

Обозначает именное пространство Internet и связанную совокупность протоколов.

Адрес сокета для именного пространства Internet включает следующие компоненты:

* Адрес машины с которой Вы хотите соединяться. Адреса в Internet могут быть определены разными способами; эти способы обсуждаются в Разделе 11.5.1 [Формат Адреса Internet] Разделе 11.5.2 [Главные Адреса], и Разделе 11.5.2.4 [Главные Имена].

* Номер порта для машины. См. Раздел 11.5.3 [Порты].

Вы должны гарантировать, что адрес и номер порта представляется в каноническим формате, называемом сетевым байтовым порядком. См. Раздел 11.5.5 [Порядок Байов], для уточнения информации относительно этого.

11.5.1 Формат Адреса сокета Internet

В именном пространстве Internet, адрес состоит из главного адреса и порта на этой главной ЭВМ. Кроме того, протокол, который Вы выбираете, служит как бы частью адреса, потому что местные числа порта значимы только внутри специфического протокола.

Тип данных для представления адресов в именном пространстве Internet определен в заголовочном файле "netinet/in.h".

```
   struct sockaddr_in  (тип данных)
```

Это тип данных, используемый, чтобы представить адреса в именном пространстве Internet. Он имеет следующие поля:

```
   short int sin_family
```

Это поле идентифицирует совокупность адресов или формат адреса сокета. Вы должны сохранить значение AF_INET в этом элементе. См. Раздел 11.3 [Адреса Гнезда].

```
   struct in_addr sin_addr
```

Это Internet адрес главной машины. См. Раздел 11.5.2 [Главные Адреса], и Раздел 11.5.2.4 [Главные Имена].

```
   unsigned short int sin_port

   Это номер порта. См. Раздел 11.5.3 [Порты].

   Когда Вы вызываете bind или getsockname, Вы должны определить
sizeof (struct sockaddr_in) как параметр длины при использовании
адреса в именном пространстве Internet.

         11.5.2 Главные Адреса

   Каждый компьютер в Internet имеет один, или большое количество
Internet адресов, т. е. числа, которые идентифицируют этот
компьютер среди остальных на Internet. Пользователи обычно записывают
число-адрес главной ЭВМ как последовательность из четырех чисел,
отделяемых точками, например "128.52.46.32".

   Каждый компьютер также имеет одно или большое количество главных
имен, которые являются строками слов, отделяемых точками, например
"churchy.gnu.ai.mit.edu".

                                - 251 -


   Программы, которые допускают пользователю определять главную ЭВМ
обычно принимают и числовые адреса и главные имена. Но для открытия соединения
программе необходим числовой адрес, так что для использования главного имени,
Вам нужно преобразовать его в числовой адрес.

         11.5.2.1 Адреса Главной ЭВМ Internet

   Адрес главной ЭВМ в Internet - это номер, содержащий четыре байта
данных. Они разделены на две части, сетевой номер и местный номер
внутри этой сети. Сетевой номер состоит из первых одного, двух или трех байт;
остальная часть байтов - местный адрес.

   Сетевые числа зарегистрированы в Сетевом Информационном Центре
(NIC), и разделены на три класса A, B, и C. Местные числа сетевого
адреса индивидуальных машин зарегистрированы  администратором
в локальной сети.

   Сеть класса A имеет одиночно-байтовые числа в диапазоне от 0 до
127. Сетей класса A не так уж много, но каждая из них может
поддерживать очень большое количество главных ЭВМ. Сети класса B
размера имеет Двух-байтовые сетевые числа, с первым байтом в
диапазоне от 128 до 191. Класс C самый маленький; адреса в нем они имеют
Трех-байтовые сетевые числа, с первым байтом в диапазоне 192-255. Таким
образом, первый 1, 2, или 3 байты адреса Internet определяют сеть.
Оставшиеся байты адреса Internet определяют адрес внутри этой сети.

   Нулевая сеть класса A зарезервирована для передачи по всем сетям.
Кроме того, главный номер 0 внутри каждой сети зарезервирован для
передачи на все главные ЭВМ в этой сети.

   127-ая сеть класса A зарезервирована для возврата цикла; Вы можете
всегда использовать адрес Internet "127.0.0.1", чтобы обратиться к
главной машине.

   Так как одиночная машина может быть элементом нескольких сетей,
она может иметь много адресов главной ЭВМ Internet. Однако,
предполагается,  что существует не более одной машины с тем же самым

                                - 252 -

главным адресом.

   Имеются четыре формы стандартного расположения чисел и точек для
Internet адреса:

   a.b.c.d  определяет все четыре байта адреса индивидуально.
   a.b.c последняя часть адреса, интерпретируется как 2-байтовое
число. Это полезно для определения главных адресов в сети класса B
с сетевым адресом a.

   a если дана только одна часть, то она соответствует непосредственно
```

числу главного адреса.

"0x" или "0X" подразумевает шестнадцатеричную систему счисления; "0" подразумевает восьмеричную; в противном случае десятичная система счисления.

11.5.2.2 Тип Данных Главного Адреса

Адреса главной ЭВМ Internet представляются в некоторых контекстах как integers (long unsigned int). В других контекстах, integer упакован внутри структуры типа struct in_addr. Было бы лучше, если бы использование было сделано непротиворечивым.

Следующие базисные определения для Internet адреса, появляются в файле "netinet/in.h":

struct in_addr (тип данных)

Этот тип данных используется в некоторых контекстах, чтобы содержать адрес главной ЭВМ Internet. Он имеет только одно поле, именованное s_addr, в которое записывается адрес как long unsigned int.

unsigned long int INADDR_LOOPBACK (макрос)

Вы можете использовать эту константу, в качестве адреса вашей машины вместо того, чтобы искать настоящий адрес. В Internet это адрес "127.0.0.1",

который обычно называется "localhost". Эта специальная константа сохраняет Вас от проблемы поиска адреса вашей собственной машины. Используя этот адрес можно имитировать передачу пакетов Internet в пределах одной машины.

unsigned long int INADDR_ANY (макрос)

Вы можете использовать эту константу вместо "любого входящего адреса". См. Раздел 11.3.2 [Установка Адреса]. Это обычный адрес, для указания в поле sin_addr структуры sockaddr_in, если Вы хотите установить соединение Internet.

unsigned long int INADDR_BROADCAST (макрос)

Эта константа - адрес, который Вы используете для посылки широковещательных сообщений.

unsigned long int INADDR_NONE (макрос)

Эта константа используется некоторыми функциями для отображения ошибок.

11.5.2.3 Функции Главного Адреса

Это дополнительные функции для управления Internet адресацией, объявленые в "arpa/inet.h". Они представляют Internet адреса в сетевом порядке байтов; это сетевые числа и числа локальных сетевых адресов в главном порядке байтов. См. Раздел 11.5.5 [Порядок Байтов], для объяснения сетевого и главного порядка байтов.

int inet_aton (const char *name, struct in_addr *addr) (функция)

Эта функция преобразовывает имя адреса главной ЭВМ Internet из стандарта числа-и-точки в двоичные данные. Inet_aton возвращает отличное от нуля чило, если адрес допустим, и нуль если нет.

unsigned long int inet_addr (const char *name) (функция)

Эта функция преобразовывает имя адреса главной ЭВМ Internet из стандарта числа-и-точки в двоичные данные. Если ввод не допустим,

inet_addr, возвращает INADDR_NONE. Это - устаревший интерфейс для inet_aton; устаревший, потому что INADDR_NONE - допустимый адрес (255.255.255.255), и inet_aton обеспечивает более чистый способ указать ошибку.

unsigned long int inet_network (const char *name) (функция)

Эта функция извлекает сетевой номер из имени адреса, данного в стандарте числа-и-точки. Если ввод не допустим, inet_network, возвращает -1.

char * inet_ntoa (struct in_addr addr) (функция)

Эта функция преобразовывает addr Internet адреса главной ЭВМ в строку в стандарте числа-и-точки. Возвращаемое значение - указатель на статически размещенный буфер. Последующие обращения запишут поверх в тот же самый буфер, так что Вы должны копировать строку, если Вы должны сохранить ее.

struct in_addr inet_makeaddr (int net, int local) (функция)

Эта функция создает Internet адрес главной ЭВМ , объединяя номер сети с местным номером.

int inet_lnaof (struct in_addr addr)

Эта функция возвращает локальную часть адреса, если Internet адрес главной ЭВМ - addr.

int inet_netof (struct in_addr addr) (функция)

Эта функция возвращает сетевую часть addr Internet адреса главной ЭВМ.

11.5.2.4 Главные Имена

Кроме стандарта числа-и-точки для Internet адреса, Вы можете также обратиться к главной ЭВМ символическим именем. Преимущество символического имени - то, что его обычно проще запомнить. Например, машина с адресом "128.52.46.32" также может иметь адрес "churchy.gnu.ai.mit.edu"; и другие машины в этом домене могут обратиться к ней просто как "churchy".

Система использует базу данных, чтобы следить за отображением между главными именами и главными числами. Эта база данных - файл, обычно "/etc/hosts" или эквивалент, обеспеченный блоком преобразования имен. Функции и другие символы для доступа к этой базе данных объявлены в "netdb.h". Возможности BSD могут использоваться при подключении файла "netdb.h".

struct hostent (тип данных)

Этот тип данных используется для представления доступа к базе данных главных ЭВМ. Он имеет следующие элементы:

char *h_name

Это "официальное" имя главной ЭВМ.

char **h_aliases

Это альтернативные имена для главной ЭВМ, представляемые как вектор с нулевым символом в конце строк.

int h_addrtype

Это тип главного адреса; практически, значение - всегда AF_INET. В принципе другие виды адресов могли бы представляться в базе данных, также как Internet адреса; если это было выполнено, Вы могли бы найти значение в этом поле отличным от AF_INET. См. Раздел 11.3 [Адреса Гнезда].

int h_length

Это длина, в байтах, каждого адреса.

char **h_addr_list

Это вектор адресов для главной ЭВМ. (Заметим, что главная ЭВМ могла бы быть соединенной с несколькими сетями и иметь различные адреса в каждой.) вектор завершен нулевым указателем.

char *h_addr

Это синоним для h_addr_list [0]; другими словами, это первый главный адрес.

В главной базе данных каждый адрес только блок памяти h_length байт длиной. Но в других контекстах имеется неявное предположение, что Вы можете преобразовывать его в struct addr_in или long unsigned int. Главные адреса в структуре struct hostent всегда даны в сетевом порядке байтов; см. Раздел 11.5.5 [Порядок Байт].

Вы можете использовать gethostbyname или gethostbyaddr, для уточнения инфрмации базы данных главных ЭВМ относительно специфической главной ЭВМ. Информация возвращена в статически размещенной структуре.

struct hostent * gethostbyname (const char *name) (функция)

Функция Gethostbyname возвращает информацию относительно главной ЭВМ, именованной name. Если происходит ошибка поиска, она возвращает пустой указатель.

struct hostent * gethostbyaddr (const char *addr, int length, (функция)

Функция Gethostbyaddr возвращает информацию относительно главной ЭВМ с адресом addr в Internet. Аргумент length - размер (в байтах) адреса addr. format определяет формат адреса; для адреса Internet,

определите значение AF_INET.

Если происходит сбой поиска, gethostbyaddr возвращает пустой указатель.

Если поиск имени gethostbyname или gethostbyaddr окончился неудачно, Вы можете выяснить причину, рассматривая значение переменной h_errno. (Было бы правильнее установить errno, но использование h_errno совместимо с другими системами.) Перед использованием h_errno, Вы должны объявить его примерно так:

extern int h_errno;

Имеются коды ошибок, которые Вы можете находить в h_errno:

HOST_NOT_FOUND

Нет такой главной ЭВМ в базе данных.

TRY_AGAIN

Это происходит, когда с блоком преобразования имен нельзя было бы входить в контакт. Если Вы попробуете сделать это позже, то возможно Вам повезет больше.

NO_RECOVERY

Произошла невосстанавливаемая ошибка .

NO_ADDRESS

Главная база данных содержит вход для имени, но он не имеет связанного Internet адреса .

Вы можете также просматривать всю базу данных главных ЭВМ используя sethostent, gethostent, и endhostent. Будьте внимательны при использовании этих функций, потому что они не допускают повторного использования.

void sethostent (int stayopen) (функция)

Эта функция открывает базу данных главных ЭВМ для просмотра. Затем Вы можете вызывать gethostent для ее чтения.

Если аргумент stayopen является отличным от нуля, она устанавливает флаг так, чтобы последующие обращения к gethostbyname или gethostbyaddr не закрыли базу данных (что они обычно сделали бы).

Это делается для эффективности, если Вы вызываете эти функции несколько раз, то избегаете повторного открытия базы данных для каждого обращения.

struct hostent * gethostent () (функция)

Эта функция возвращает следующий вход в базе данных главных ЭВМ. Она возвращает пустой указатель, если не имеется больше входов.

void endhostent () (функция)

Эта функция закрывает базу данных главных ЭВМ.

11.5.3 Порты Internet

Адрес сокета в именном пространстве Internet состоит из адреса Internet машины плюс номер порта, который отличает гнездо на данной машине (для данного протокола). Номера портов располагаются от 0 до 65535.

Номера портов меньше, зарезервированых IPPORT_RESERVED для стандартных серверов, типа finger и telnet. Имеется база данных, которая следит за ними, и Вы можете использовать функцию getservbyname для отображения сервисного номера порта; см. Раздел 11.5.4 [База данных Услуг].

Если Вы собираетесь устанавливать сервер, который не является стандартно определенным в базе данных, то Вам необходимо выбрать для него номер порта.

Используйте номера большие чем IPPORT_USERRESERVED; такие числа зарезервированы для серверов и никогда не будут генерироваться системой.

Когда Вы используете сокет без определения адреса, система генерирует номер порта для него. Этот номер попадает в интервал между IPPORT_RESERVED и IPPORT_USERRESERVED.

На Internet, фактически, законно иметь два различных сокета с одинаковыми номерами портов, пока они оба не попытаются связаться с тем этим адресом сокета (главный адрес плюс номер порта). Вы не должны дублировать номер порта за исключением специальных обстоятельств, где протокол с более высоким уровнем требует этого. Обычно, система не будет разрешать Вам делать это; bind требует различные номера портов. Чтобы многократно использовать номер порта, Вы должны установить опцию сокета SO_REUSEADDR. См. Раздел 11.11.2 [Опции Сокетов].

Эти макрокоманды определены в заголовочном файле "netinet/in.h".

int IPPORT_RESERVED (макрос)

Номера портов меньшие IPPORT_RESERVED зарезервированы для использования суперпользователем.

int IPPORT_USERRESERVED (макрос)

Номера портов большие или равные IPPORT_USERRESERVED зарезервированы для явного использования; они никогда не будут размещены автоматически.

11.5.4 База данных Услуг

База данных, которая следит за "общеизвестными" услугами - это обычно или файл "/etc/services" или эквивалент из блока преобразования имен. Вы можете использовать эти утилиты, объявленные в "netdb.h" для обращения к базе данных услуг.

struct servent (тип данных)

Этот тип данных содержит информацию относительно входов в базе данных услуг, он имеет следующие элементы:

char *s_name

Это "официальное" имя обслуживания.

char **s_aliases

Это альтернативные имена обслуживания, представляемые массивом строк.

Пустой указатель завершает массив.

int s_port

Это номер порта для обслуживания. Номера портов даны в сетевом порядке байтов; см. Раздел 11.5.5 [Порядок Байтов].

char *s_proto

Это имя протокола, для использования с этим обслуживанием. См. Раздел 11.5.6 [База данных Протоколов].

Чтобы получать информацию относительно специфического обслуживания, используйте функции getservbyname или getservbyport функции. Информация возвращается в статически размещенной структуре.

struct servent * getservbyname (const char *name, const char *proto) (функция)

Getservbyname функция возвращает информацию относительно обслуживания, именованного name, используя протокол proto. Если она не может найти такое обслуживание, она возвращает пустой указатель.

Эта функция полезна как для серверов так и для клиентов; серверы используют ее, чтобы определить, на каком порту они должны принимать пакеты (см. Раздел 11.8.2 [Прием]).

struct servent * getservbyport (int port, const char *proto) (функция)

Функция Getservbyport возвращает информацию относительно обслуживания на порте port, используя протокол proto. Если она не может найти такое обслуживание, она возвращает пустой указатель.

Вы можете также просматривать базу данных услуг, используя setservent, getservent, и endservent. Будьте внимательным в использовании этих функций, потому что они не предназначены для повторного использования.

void setservent (int stayopen) (функция)

Эта функция открывает базу данных услуг для просмотра.

Если аргумент stayopen является отличным от нуля, она

устанавливает флаг так, чтобы последующие обращения к getservbyname или getservbyport не закрыли базу данных (поскольку они обычно закрыли бы). Это делается для большей эффективности, если Вы вызываете эти функции несколько раз, избегая повторного открытия базы данных для каждого обращения.

struct servent * getservent (void) (функция)

Эта функция возвращает следующий вход базы данных услуг. Если там нет больше входов, она возвращает пустой указатель.

void endservent (void) (функция)

Эта функция закрывает базу данных услуг.

11.5.5 Преобразование Порядка Байтов

Различные виды компьютеров используют различные соглашения для упорядочения байтов внутри слова. Некоторые компьютеры помещают старший байт сначала (это называется "big-endian" порядком), а другие помещают его последним ("little-endian" порядок).

Так, чтобы машины с различными соглашениями порядка байтов могли связываться, протоколы Internet определяют каноническое соглашение порядка байтов для данных, переданных по сети. Оно известно как сетевой порядок байта.

При установлении соединения в Internet, Вы должны удостовериться, что данные в sin_port и sin_addr элементах структуры sockaddr_in представляются в сетевом порядке байта. Если Вы кодируете данные integer в сообщениях, посланных через сокет, Вы должны преобразовать их в сетевой порядок байта. Если Вы не делаете этого, ваша программа может работать не правильно при сообщении с другими типами машин.

Если Вы используете getservbyname и gethostbyname или inet_addr, для получения номера порта и главного адреса, то эти значения уже в сетевом порядке байта, и Вы можете копировать их непосредственно в структуру sockaddr_in.

Иначе, Вы должны преобразовать значения явно. Используйте htons и ntohs, чтобы преобразовать значения для sin_port элемента. Используйте htonl и ntohl, чтобы преобразовать значения для sin_addr элемента. (Помните, struct in_addr эквивалентен long unsigned int.) Эти функции описаны в "netinet/in.h".

unsigned short int htons (unsigned short int hostshort) (функция)

Эта функция преобразовывает short integer hostshort из главного порядка байтов в сетевой порядок байта.

unsigned short int ntohs (unsigned short int netshort) (функция)

Эта функция преобразовывает short integer netshort из сетевого порядка байта в главный порядок байта.

unsigned long int htonl (unsigned long int hostlong)

Эта функция преобразовывает long integer hostlong из главного порядка байтов в сетевой порядок байт.

unsigned long int ntohl (unsigned long int netlong) (функция)

Эта функция преобразовывает long integer netlong из сетевого порядка байт в главный порядок байт.

11.5.6 База данных Протоколов

Протокол связи используется для управления низкого уровня обмена данными. Например, протокол осуществляет вещи подобно контрольным суммам, чтобы обнаружить ошибки в передачах, и команды маршрутизации для сообщений.

Заданный по умолчанию протокол связи для именного пространства Internet зависит от стиля связи. Для потокового взаимодействия, значение по умолчанию - TCP ("протокол управления передачей"). Для датаграмной связи, значение по умолчанию - UDP ("протокол датаграммы пользователя"). Для надежной датаграмной связи значение по умолчанию - RDP ("надежный датаграмный протокол"). Вы должны почти всегда использовать это значение по умолчанию.

Протоколы Internet вообще определены именем вместо номера. Сетевые протоколы, которые знает главная ЭВМ, сохранены в базе данных. Она обычно происходит от файла "/etc/protocols", или может быть эквивалент, обеспеченный блоком преобразования имен. Вы можете искать номер протокола, связанный с именованным протоколом в базе данных, используя getprotobyname функцию.

Имеются детализированные описания утилит для доступа к базе данных протоколов. Они объявлены в "netdb.h".

struct protoent (тип данных)

Этот тип данных используется, чтобы представить входы в базе данных сетевых протоколов. Он имеет следующие элементы:

char *p_name

Это официальное имя протокола.

char **p_aliases

Это альтернативные имена для протокола, заданные как массив строк.

Последний элемент массива - пустой указатель.

int p_proto

Это номер протокола (в главном порядке байт); используйте этот элемент как аргумент protocol для socket.

Вы можете использовать getprotobyname и getprotobynumber, чтобы искать в базе данных протоколов специфический протокол. Информация возвращается в статически размещенной структуре; Вы должны копировать информацию, если Вы хотите сохранить ее для следующих обращений.

struct protoent * getprotobyname (const char *name) (функция)

Функция Getprotobyname возвращает информацию относительно сетевого протокола, именованного name. Если там нет такого протокола, она возвращает пустой указатель.

struct protoent * getprotobynumber (int protocol) (функция)

Getprotobynumber функция возвращает информацию относительно сетевого протокола с указанным номером. Если там нет такого протокола, она возвращает пустой указатель.

Вы можете также просматривать целую базу данных протоколов (по одному протоколу одновременно), используя setprotoent, getprotoent,

и endprotoent. Будьте внимательным в использовании этих функций, потому что они не предназначены для повторного использования.

void setprotoent (int stayopen) (функция)

Эта функция открывает для просмотра базу данных протоколов.

Если аргумент stayopen является отличным от нуля, она устанавливает флаг так, чтобы последующие обращения к getprotobyname или getprotobynumber не закрыли базу данных. Это делается для большей эффективности, если Вы вызываете эти функции несколько раз, избегая повторного открытия базы данных для каждого обращения.

struct protoent * getprotoent (void) (функция)

Эта функция возвращает следующий вход в базе данных протоколов. Она возвращает пустой указатель, если не имеется больше входов.

void endprotoent (void) (функция)

Эта функция закрывает базу данных протоколов.

11.5.7 Пример Internet сокета.

Вот пример, показывающий, как создавать и называть сокет в именном пространстве Internet. Созданный сокет существует на машине, на которой выполняется программа. Вместо поиска и использования адреса Internet машины, этот пример определяет INADDR_ANY как главный адрес.


```
#include
#include
#include
#include
int
make_socket (unsigned short int port)
{
        int sock;
        struct sockaddr_in name;
        sock = socket (PF_INET, SOCK_STREAM, 0);
        if (sock < 0)
                {
                        perror ("socket");
                        exit (EXIT_FAILURE);
                }
        name.sin_family = AF_INET;
        name.sin_port = htons (port);
        name.sin_addr.s_addr = htonl (INADDR_ANY);
        if (bind (sock, (struct sockaddr *) &name,
                                sizeof (name)) < 0)
                {
                        perror ("bind");
                        exit (EXIT_FAILURE);
                }
        return sock;
}
```

Вот другой пример, показывающий как Вы можете вносить в структуре sockaddr_in, данную строку главного имени и номер порта:

```
#include
#include
#include
#include
#include
void
init_sockaddr (struct sockaddr_in *name, const
```

```
        char *hostname, unsigned short int port)
```



```
{
        struct hostent *hostinfo;
        name->sin_family = AF_INET;
        name->sin_port = htons (port);
        hostinfo = gethostbyname (hostname);
        if (hostinfo == NULL)
                {
                        fprintf (stderr, "Unknown host
                                %s.\n", hostname);
                        exit (EXIT_FAILURE);
                }
        name->sin_addr = *(struct in_addr *)
        hostinfo->h_addr;
}
```

11.6 Другие именные пространства

Конечно другие именные пространства и связанные семейства протоколов также реализованы, но не описаны здесь, потому что они редко используются. PF_NS обращается к протоколам Программного обеспечения Сети Ксерокса (Xerox Network Software). PF_ISO замещает Открытые системы Связи (Open Systems Interconnect). PF_CCITT обращается к протоколам из MKKTT (CCITT). "Socket.h" определяет эти символы и другие протоколы.

PF_IMPLINK используется для связи между главными ЭВМ и Процессорами Сообщений Internet.

11.7 Открытие и Закрытие сокетов

Этот раздел описывает фактические библиотечные функции для открытия и закрытия сокетов. Те же самые функции работают для всех именных пространств и стилей соединения.

11.7.1 Создание сокета.

Примитив для создания сокета - функция socket, объявлена в "sys/socket.h".

int socket (int namespace, int style, int protocol) (функция)

Эта функция создает сокет и определяет style стиль связи, который должен быть одним из стилей сокетов, перечисленных в Разделе 11.2 [Стили Связи]. Аргумент namespace определяет именное пространство; это должно быть PF_FILE (см. Раздел 11.4 [Именное пространство Файла]) или PF_INET (см. Раздел 11.5 [Именное пространство Internet]). protocol обозначает специфический протокол (см. Раздел 11.1 [Понятия Гнезда]).

Возвращаемое значение из socket - описатель файла для нового сокета, или -1 в случае ошибки. Следующие errno условия ошибки определены для этой функции:

EPROTONOSUPPORT

Протокол или стиль не обеспечивается заданным именным пространством.

EMFILE процесс имеет слишком много открытых описателей файла.

ENFILE система имеет слишком много открытыми описателей файла

EACCESS процесс не имеет привилегии, чтобы создать сокет заданного стиля или протокола.

ENOBUFS в системе закончилось внутреннее пространство буфера.

Описатель файла, возвращенный функцией socket поддерживает и чтение и запись. Но, подобно трубопроводам, сокеты не поддерживают операции позиционирования файла.

Пример вызова функции socket см. Раздел 11.4 [Именное пространство Файла].

- 269 -

11.7.2 Закрытие сокета.

Когда Вы закончили использование сокета, Вы можете просто закрыть описатель файла примитивом close; см. Раздел 8.1 [Открытие и Закрытие Файлов].

Вы можете также выключать только прием или только передачу на соединении, вызывая shutdown, которая объявлена в "sys/socket.h".

int shutdown (int socket, int how) (функция)

Функция shutdown выключает соединение с сокетом socket. Аргумент how определяет какое действие выполнить:

0 Остановка при получения данных для этого сокета.
1 Остановка при передаче данных с этого сокета.
2 Остановка и приема и передачи.

Возвращаемое значение - 0 при успехе и -1 в случае неудачи. В переменной errno определяются следующие коды ошибок для этой функции:

EBADF socket - не допустимый описатель файла.
ENOTSOCK socket - не сокет.
ENOTCONN socket не соединен.

11.7.3 Пары сокетов

Пара socket состоит из пары соединенных (но неименованных) сокетов. Это очень похоже на трубопровод и используется аналогичным способом. Пары сокетов создаются функцией socketpair, описание в файле "sys/socket.h".

int socketpair (int namespace, int style, int protocol, int fields[Function2])

Эта функция создает пару сокетов, возвращая описатели файла в fields [0] и fields [1]. Пара сокетов - дуплексный канал связи, то есть и чтение и запись могут выполняться в любую сторону.

- 270 -

Аргументы namespace, style и protocol интерпретируется как в функции socket. style должен быть один из стилей связи, перечисленных в Разделе 11.2 [Стили связb]. Аргумент именного пространства определяет именное пространство, которое должно быть AF_FILE (см. Раздел 11.4 [Именное пространство Файла]); protocol определяет протокол связи.

Если style определяет стиль связи без установки логического соединения, то два сокета, которые Вы получаете, не соединены, строго говоря, но каждое из них знает другое как заданный по умолчанию адрес адресата, так что они могут посылать пакеты друг другу.

Функция Socketpair возвращает 0 при успехе и -1 при отказе. В переменной errno определяются следующие коды ошибок для этой функции:

EMFILE Процесс имеет слишком много открытых описателей файла.
EAFNOSUPPORT Не обеспечивается заданное именное пространство.
EPROTONOSUPPORT Не обеспечивается заданный протокол.
EOPNOTSUPP Заданный протокол не поддерживает создание пар сокетов.

11.8 Использование сокетов с соединениями.

Наиболее общие стили связи включают создание соединения с другим сокетом, и многократным обменом данными между этими сокетами. Создание соединения асимметрично; одна сторона (клиент) действует, чтобы запросить соединение, в то время как другая сторона (сервер) создает сокет и ждет запрос на соединение.

* Раздел 11.8.1 [Соединение], описывает то, что клиентская программа должна делать, чтобы инициализировать соединение с сервером.

* Раздел 11.8.2 [Прием], и Раздел 11.8.3 [Принятие Соединений], описывает то, что программа сервера должна делать, чтобы ждать и делать после запросов соединения от клиентов.

* Раздел 11.8.5 [Пересылка Данных], описывает, как данные перемещаются через соединенные сокеты.

11.8.1 Создание Соединения

В создании соединения, клиент делает соединение, в то время как сервер ждет и принимает соединение. Здесь мы обсуждаем то, что клиентская программа должна делать, используя функцию connect, которая объявлена в "sys/socket.h".

int connect (int socket, struct sockaddr *addr, size_t length)
(функция)

Функция connect инициализирует соединение из сокета socket, чей адрес определен аргументами length и addr. (Этот сокет обычно находится на другой машине, и он должен быть установлен как сервер.)
См. Раздел 11.3 [Адреса Сокетов], для уточнения инфрмации относительно того, как эти аргументы интерпретируются.

Обычно, connect ждет, пока сервер не отвечает на запрос прежде. Вы можете устанавливать режим неблокирования на сокете socket, чтобы заставить connect возвратиться немедленно без ожидания ответа. См. Раздел 8.10 [Флаги Состояния Файла], для уточнения инфрмации относительно неблокирования.

Нормальное возвращаемое значение connect - 0. Если происходит ошибка, connect возвращает -1. В переменной errno определяются следующие коды ошибок для этой функции:

EBADF сокет socket - не допустимый описатель файла.

ENOTSOCK указанный сокет - не сокет.

EADDRNOTAVAIL заданный адрес не доступен на отдаленной машине.

EAFNOSUPPORT именное пространство addr не обеспечивается этим сокетом.

EISCONN указанный сокет уже соединен.

ETIMEDOUT попытка установить соединение не состоялась.

ECONNREFUSED сервер активно отказался устанавливать соединение.

ENETUNREACH сеть данного addr не доступна с этой главной ЭВМ.

EADDRINUSE адрес сокета для данного addr уже используется.

EINPROGRESS указанный сокет не-блокируемый, и соединение не могло бы быть установлено немедленно.

EALREADY указанный сокет не-блокируемый и уже имеет отложенное соединение.

11.8.2 Ожидание Соединений

Теперь рассмотрим то, что процесс сервера должен делать, чтобы принять соединение из сокета. Это включает использование функции listen, чтобы дать возможность запросам на соединения через сокет, и позже использование функции accept (см. Раздел 11.8.3 [Принятие Соединений]) чтобы действовать по запросу. Функция listen используется только для уже установленного логического соединения.

В именном пространстве Internet, не сущестует специальных механизмов защиты управления доступом к порту; любой процесс на любой машине может установить соединение с вашим сервером. Если Вы хотите ограничивать доступ к вашему серверу, заставьте его исследовать адреса, связанные с запросами соединения или выполнять некоторое другое подтверждение связи или протокол идентификации.

В именном пространстве Файла, обычные биты защиты файла управляют доступом к сокету.

int listen (int socket, unsigned int n) (функция)

Функция listen дает возможность указанному сокету воспринимать соединения, таким образом создается сокет сервера.

Аргумент n определяет длину очереди для отложенных соединений.

Функция listen возвращает 0 при успехе и -1 в случае неудачи. В переменной errno определяются следующие коды ошибок для этой функции:

EBADF аргумент socket - не допустимый описатель файла.

ENOTSOCK аргумент socket - не сокет.

EOPNOTSUPP указанный сокет не поддерживает эту операцию.

11.8.3 Принятие Соединений

Когда сервер получает запрос соединения, он может создать соединение, принимая запрос. Для этих целей следует использовать функцию accept.

Сокет, который был установлен как сервер, может принимать запросы соединения от многих клиентов. Этот сокет сервера не станет частью соединения; взамен, accept делает новый сокет, который разделяет соединения. Accept возвращает описатель для этого сокета.

Исходный сокет сервера остается доступным для ожидания дальнейших запросов соединения.

Число отложенных запросов соединения на сокете сервера конечно. Если запросы соединения прибывают быстрее, чем сервер может их обработать, очередь может заполниться, и дополнительные запросы получат отказ с ошибкой ECONNREFUSED. Вы можете определять максимальную длину этой очереди как аргумент функции listen, хотя система может также наложить собственное внутреннее ограничение длины этой очереди.

int accept (int socket, struct sockaddr *addr, size_t *length_ptr)

Эта функция используется для принятия запроса на соединения в указанном сокете сервера.

Функция accept находится в состоянии ожидания, когда нет возможности принять соединение, если, конечно, указанный сокет не имеет набор режимов неблокирования. (Вы можете использовать select, чтобы ждать отложенное

соединение на неблокируемом сокете.) См. Раздел 8.10 [Флаги Состояния Файла], для уточнения информации относительно режима неблокирования.

Аргументы Addr и length_ptr используется, чтобы возвратить информацию относительно имени клиентского сокета, которое инициализировало соединение. См. Раздел 11.3 [Адреса сокетов], для уточнения информации относительно формата.

Сокет, который был установлен как сервер не станет частью соединения; взамен, accept сделает новый сокет. Accept возвращает описатель для этого сокета. Нормальное возвращаемое значение accept - описатель файла для нового сокета.

После accept, первоначально указанный сокет остается открытым и не связанным, и продолжает ожидать, пока Вы не закрываете его. Вы можете принимать дальнейшие соединения с этим сокетом, вызывая accept снова.

Если происходит ошибка, и accept возвращает -1.
В переменной errno определяются следующие коды ошибок для этой функции:

EBADF аргумент socket - не допустимый описатель файла.

ENOTSOCK дескрипторный аргумент socket - не сокет.

EOPNOTSUPP описанный сокет не поддерживает эту операцию.

EWOULDBLOCK сокет имеет набор режимов неблокирования, и нет никаких отложенных соединений.

Функцию accept не позволяется применять для сокета без установления логического соединения.

11.8.4 Кто соединен со Мной?

int getpeername (int socket, struct sockaddr *addr, size_t *length_ptr) (функция)

Функция Getpeername возвращает адрес сокета, с которым сокет соединен; она сохраняет адрес в пространстве памяти, заданном addr и length_ptr. Она сохраняет также длину адреса в *length_ptr.

См. Раздел 11.3 [Адреса Сокетов] , для уточнения информации относительно формата адреса. В некоторых операционных системах, getpeername работает только для сокетов в области Internet.

Возвращаемое значение - 0 при успехе и -1 в случае неудачи.
В переменной errno определяются следующие коды ошибок для этой функции:

EBADF аргумент socket - не допустимый описатель файла.

ENOTSOCK указанный сокет - не сокет.

ENOTCONN указанный сокет не соединен.

ENOBUFS нет внутренних доступных буферов.

11.8.5 Пересылка Данных

Если сокет был соединен с равным, Вы можете использовать обычные примитивы read и write (см. Раздел 8.2[Примитивы ввода - вывода]), чтобы передать данные. Сокет - канал двусторонней связеи, так что чтение и запись может выполняться в оба конца.

Имеются также некоторые режимы ввода - вывода, которые являются специфическими для операций с сокетами. Чтобы определять эти режимы, Вы

должны использовать функции recv и send вместо более обобщенного чтения и записи. Функции recv и send берут дополнительный

аргумент, который Вы можете использовать, чтобы определить различные флаги, для управления специальными режимами ввода - вывода. Например, Вы можете определить флаг MSG_OOB, чтобы читать или писать внепоточные данные, а также флаги MSG_PEEK или MSG_DONTROUTE.

11.8.5.1 Посылка Данных

Функция send объявлена в файле "sys/socket.h". Если ваш аргумент flags нуль, Вы можете точно также использовать write вместо send. Если сокет был соединен, но соединение прервано, Вы получаете сигнал SIGPIPE для каждого использования send или write (см. Раздел 21.2.6 [Разнообразные Сигналы]).

int send (int socket, void *buffer, size_t size, int flags) (функция)

Функция send - подобна write, но с дополнительными флагами Flags. Возможные значения flags описаны в Разделе 11.8.5.3 [Опции Данных сокетов].

Эта функция возвращает число переданных байтов, или -1 в противном случае. Если сокет неблокируемый, то send (подобно write) может возвращать после посылки только часть данных. См. Раздел 8.10 [Флаги Состояния Файла], для уточнения информации относительно режима неблокирования.

Обратите внимание, что успешное возвращаемое значение просто указывает, что сообщение было послано без ошибки, и не обязательно, что оно было получено без ошибки. В переменной errno определяются следующие коды ошибок для этой функции:

EBADF аргумент socket - не допустимый описатель файла.

EINTR операция был прервана сигналом прежде, чем любые данные были посланы. См. Раздел 21.5 [Прерванные Примитивы].

ENOTSOCK указанный сокет - не сокет.

EMSGSIZE тип сокетаа требует, чтобы сообщение было послано быстро, но сообщение слишком большое для этого.

EWOULDBLOCK на сокете был установле режим неблокирования, а операция записи блокирует. (Обычно send блокирует, пока операция не может быть завершена.)

ENOBUFS не имеется достаточного внутреннего доступного пространства буфера.

ENOTCONN Вы не соединили этот сокет.

EPIPE Этот сокет был соединен, но соединение теперь разбито. В этом случае send генерирует SIGPIPE сначала; если этот сигнал игнорируется или блокируется, или если обработчик возвращается, то происходит сбой send с EPIPE.

11.8.5.2 Получение Данных

Функция recv объявлена в файле "sys/socket.h". Если ваш аргумент flags является нулем, Вы можете точно также использовать read вместо recv; см. Раздел 8.2 [Примитивы ввода-вывода].

int recv (int socket, void *buffer, size_t size, int flags) (функция)

Функция recv подобна read, но с дополнительными флагами flags. Возможные значения flags описаны В Разделе 11.8.5.3 [Опции Данных сокетов].

Если режим неблокирования установлен для сокета, и никакие данные не доступны для чтения, recv не ожидает, а сразу возвращает код ошибки. См. Раздел 8.10 [Флаги Состояния Файла], для уточнения информации относительно режима неблокирования.

Эта функция возвращает число полученных байтов, или -1 в противном случае.

В переменной errno определяются следующие коды ошибок для этой функции:

EBADF аргумент socket - не допустимый описатель файла.

ENOTSOCK дескриптор socket - не сокет.

EWOULDBLOCK Режим неблокирования был установлен на сокете. (Обычно, recv блокирует пока не имеется входа, доступного для чтения.)

EINTR операция была прервана сигналом прежде, чем любые данные прочитались. См. Раздел 21.5 [Прерванные Примитивы].

ENOTCONN Вы не соединили этот сокет.

11.8.5.3 Опции Данных сокета.

Аргумент flags для send и recv - битовая маска. Вы можете объединить значения следующих макрокоманд вместе (через OR), чтобы получить значение для этого аргумента. Все они определены в файле "sys/socket.h".

int MSG_OOB (макрос)

Посылка или получение данных вне потока. См. Раздел 11.8.8 [Данные вне потока].

int MSG_PEEK (макрос)

Рассмотрение данных, но не удаление их из входной очереди. Это применимо только для функций типа recv (для send не подходят).

int MSG_DONTROUTE (макрос)

Не включать информацию о маршрутизации в сообщении. Это имеет смысл только с операциями вывода, и обычно представляет интерес только для диагностики программы.

11.8.6 Пример сокета с потоком байтов.

Вот пример программы клиента, которая устанавливает соединение для сокета в пространстве Internet с поточным типом передачи данных. Она не делает ни чего особенно интересного; если она соединилась с сервером, она посылает текстовую строку серверу и выходит.

```
#include
#include
#include
#include
#include
#include
#include
#include
#define PORT    5555
#define MESSAGE "Yow!!! Are we having fun yet?!?"
#define SERVERHOST              "churchy.gnu.ai.mit.edu"
void
write_to_server (int filedes)
{
        int nbytes;
        nbytes=write(filedes,MESSAGE,strlen(MESSAGE)+1);
        if (nbytes < 0)
                {
                        perror ("write");
                        exit (EXIT_FAILURE);
                }
}
```

```
int
main (void)
{
        extern void init_sockaddr(struct sockaddr_in*name,
                                  const char *hostname,
                                  unsigned short int port);
        int sock;
        struct sockaddr_in servername;
        sock = socket (PF_INET, SOCK_STREAM, 0);
```



```
        if (sock < 0)
                {
                  perror ("socket (client)");
                  exit (EXIT_FAILURE);
                }
        init_sockaddr (&servername, SERVERHOST, PORT);
        if (0 > connect (sock,
                         (struct sockaddr *) &servername,
                         sizeof (servername)))
                {
                  perror ("connect (client)");
                  exit (EXIT_FAILURE);
                }
        write_to_server (sock);
        close (sock);
        exit (EXIT_SUCCESS);
}
```

11.8.7 Пример соединения сервера.
(Тип соединения -поток байтов)

Текст программы сервера намного более сложен. Так как мы хотим предоставлять многим клиентам быть соединенными с сервером, но в то же самое время, было бы неправильно ждать ввод от одиночного клиента, просто вызывая read или recv. Взамен, нужно использовать select (см. Раздел 8.6 [Ждущий ввод - вывод]), чтобы ждать ввод на всех открытых сокетах. Это также позволяет серверу иметь дело с дополнительными запросами соединения.

Этот специфический сервер не делает хоть что-нибудь интересное, если он получил сообщение от клиента, то он закрывает сокет клиента по получению признака конца файла.

Эта программа использует make_socket и init_sockaddr для устанавления адреса сокета; см. раздел 11.5.7 [Inet Пример].


```
#include
#include
#include
#include
#include
#include
#include
#include
#define PORT   5555
#define MAXMSG  512
int
read_from_client (int filedes)
{
        char buffer[MAXMSG];
        int nbytes;
        nbytes = read (filedes,buffer,MAXMSG);
        if (nbytes < 0)
                {
                  perror ("read");
                  exit (EXIT_FAILURE);
                }
        else if (nbytes == 0)
```

```
                return -1;
        else
                {
                        fprintf (stderr, "Server: got message:
                                `%s'\n", buffer);
                        return 0;
                }
}
int
main (void)
{
        extern int make_socket (unsigned short int port);
        int sock;
        int status;
        fd_set active_fd_set, read_fd_set;
        int i;

                       - 282 -

        struct sockaddr_in clientname;
        size_t size;
        sock = make_socket (PORT);
        if (listen (sock, 1) < 0)
        {
                perror ("listen");
                exit (EXIT_FAILURE);
        }
        FD_ZERO (&active_fd_set);
        FD_SET (sock, &active_fd_set);
        while (1)
                {
                        if (select (FD_SETSIZE,
                                &read_fd_set, NULL,
                                NULL, NULL) < 0)
                        {
                                perror ("select");
                                exit (EXIT_FAILURE);
                        }
                if (FD_ISSET (i, &read_fd_set))
                {
                        if (i == sock)
                        {
                                if (accept (sock,
                (struct sockaddr *) &clientname, &size) < 0)

                        {
                                        perror ("accept");
                                        exit (EXIT_FAILURE);
                                }
                                fprintf (stderr,
        "Server: connect from host %s, port %hd.\n",
                                inet_ntoa (clientname.sin_addr),
                                ntohs (clientname.sin_port));
                                FD_SET (status, &active_fd_set);
                        }
                        else
                        {

                       - 283 -

                                if (read_from_client (i)<0)
                                {
                                        close (i);
                                        FD_CLR (i, &active_fd_set);
                                }
                        }
                }
        }
}
```

11.8.8 Данные Вне потока

Потоки с соединениями разрешающими данные вне потока имеют приоритет выше, чем обычные данные. Обычно причина для посылки данных вне потока - исключительные условия. Способ послать

данные вне потока использует send с флагом MSG_OOB (см. Раздел 11.8.5.1 [Посылка Данных]).

Данные вне потока посылаются с высшим приоритетом, плюс процесс получения не обрабатывает их в обыкновенное очереди, но чтобы читать доступные данные вне потока следует использовать recv с флагом MSG_OOB (см. Раздел 11.8.5.2 [Получение Данных]). Обычные операции чтения не воспринимают данные вне потока; они читают только обычные данные.

Когда сокет находит, что данные вне потока продвигаются, он посылает сигнал SIGURG процессу владельца или группе процессов сокета. Вы можете определять владельца, используя команду F_SETOWN для функции fcntl; см. Раздел 8.12 [Ввод Прерывания]. Вы должны также установить обработчик для этого сигнала, как описано в Главе 21 [Обработка Сигналов], для соответствующего действия типа чтения данных вне потока.

В качестве альтернативы, Вы можете проверять задержать данные вне потока, или ждать данные вне потока, при использовании функции select; она может ждать исключительное условие на гнезде. См. Раздел 8.6 [Ждущий ввод - вывод].

Уведомление о данных вне потока (с SIGURG или с select) обозначает, что данные вне потока находятся в пути; данные не могут фактически прибывать позже. Если Вы пробуете читать данные вне потока прежде, чем они ппребывают, то recv генерирует ошибку с кодом EWOULDBLOCK.

Посылка таких данных автоматически помещает "метку" в потоке обычных данных, показывающую, где в последовательности данных " были бы "данные вне потока. Это полезно, когда значение данных вне потока - " отменяет все посланное до ". Вот, как Вы можете в процессе получения проверять, были ли любые обычные данные посланы перед меткой:

```
success = ioctl (socket, SIOCATMARK, &result);
```

Имеется функция, чтобы отбросить любые обычные данные, предшествующие данным вне потока:

```
int
discard_until_mark (int socket)
{
        while (1)
        {
                char buffer[1024];
                int result, success;
                success = ioctl (socket, SIOCATMARK,
                &result); if (success < 0)
                perror ("ioctl");
                if (result)
                        return;
                success = read (socket, buffer, sizeof
                                buffer);
                if (success < 0)
                perror ("read");
        }
}
```


Если Вы не хотите отбрасывать обычные данные, предшествующие метке, Вам необходимо создать место во внутренних буферах систем для данных вне потока. Если Вы пробуете читать данные вне потока и получаете ошибку EWOULDBLOCK, попробуйте читать некоторые обычные данные (сохраняя их так, чтобы Вы могли использовать их позже) и смотрите появится ли необходимое место. Вот пример:

```
struct buffer
```

```
                {
                        char *buffer;
                        int size;
                        struct buffer *next;
                };
        struct buffer *
        read_oob (int socket)
        {
                struct buffer *tail = 0;
                struct buffer *list = 0;
                while (1)
                {
                        char *buffer = (char *) xmalloc (1024);
                        struct buffer *link;
                        int success;
                        int result;
                        success = recv (socket, buffer, sizeof
                buffer, MSG_OOB);
                        if (success >= 0)
                        {
                                link->size = success;
                                link->next = list;
                                return link;
                        }
                        success = ioctl (socket, SIOCATMARK,
                &result);
                         if (success < 0)
                                perror ("ioctl");
                        if (result)
                        {

                               - 286 -

                                sleep (1);
                                continue;
                        }
                        success = read (socket, buffer, sizeof
                        buffer);
                        if (success < 0)
                        perror ("read");
                        {
                                link->size = success;
                                if (tail)
                                        tail->next = link;
                                else
                                        list = link;
                                tail = link;
                        }
                }
        }
```

11.9 Датаграмные операции сокета

Этот раздел описывает, как использовать стили связи, которые не используют соединения (стили SOCK DGRAM и SOCK_RDM). При использовании этих стилей, Вы группируете данные в пакеты, и каждый пакет - независимая связь. Вы определяете адресата для каждого пакета индивидуально.

Датаграмные пакеты подобны письмам: Вы посылаете каждый независимо, с собственным адресом адресата, и они могут прибывать в неправильном порядке или вообще не прибывать.

Функции listen и accept не предназначены для сокетов, использующих стили связи без установки логического соединения.

11.9.1 Посылка Датаграмм

Нормальный способ посылки данных относительно датаграмного сокета использует функцию sendto, объявленную в "sys/socket.h".

!!! Вы можете вызывать connect на датаграмном сокете, но эта функция определяет заданного по умолчанию адресата для дальнейшей передачи данных на сокете. Когда сокет имеет по умолчанию заданного адресата, Вы можете использовать send (см. Раздел 11.8.5.1 [Посылка Данных]) или write (см. Раздел 8.2 [Примитивы ввода - вывода]) для посылки пакетов. Вы можете отменять заданного по умолчанию адресата, вызывая connect, и используя формат адреса AF_UNSPEC в аргументе addr. См. Раздел 11.8.1 [Соединение].

int sendto (int socket, void *buffer. size_t size, int flags, sockaddr *addr, size_t length)

Эта функция пересылает данные из buffer через сокет socket по заданному адресу. size задает число пересылаемых байт.

Flags интерпретируется также как и в send; см. Раздел 11.8.5.3 [Опции данных сокетов].

Возвращаемое значение и условия ошибок такие же как и для send, но Вы не можете полагаться на систему для обнаружения ошибок и сообщения о них; наиболее общая ошибка состоит в том, что пакет потеряется или не имеется никого в заданном адресе, чтобы получить его, и операционная система на вашей машине обычно не знает этого.

Также возможно, что для одного обращения к sendto она сообщит ошибку из-за проблемы связанной с предыдущим обращением.

11.9.2 Получение Датаграмм

Функция recvfrom читает пакет из датаграмного сокета и также сообщает Вам, откуда он был послан. Эта функция объявлена в "sys/socket.h".

int recvfrom (int socket, void *buffer, size_t size, int flags, struct sockaddr *addr, size_t *length_ptr)

Функция recvfrom читает один пакет из указанного сокета в указанный буфер.

Аргумент size определяет максимальное число байтов, которые нужно читать.

Если пакет является больше чем size байт, то, Вы получаете первые size байт пакета, а остальная часть пакета потеряна. Не существует способа прочитать остальную часть пакета. Таким образом, когда Вы используете протокол пакетов, Вы должны всегда знать длину ожидаемого пакета.

Аргументы addr и length_ptr используются для возвращения адреса источника пакета. См. Раздел 11.3 [Адреса сокетов]. Для сокета в области файла, информация адреса не будет значима, так как Вы не можете читать адрес такого сокета (см. Раздел 11.4 [Именное пространство Файла]). Вы можете определять пустой указатель как аргумент addr, если Вы не заинтересованы в этой информации.

Flags интерпретируется тем же самым способом как recv (см. Раздел 11.8.5.3 [Опции Данных сокета]). Возвращаемое значение и условия ошибки - такие же как для recv.

Вы можете использовать recv (см. Раздел 11.8.5.2 [Получение Данных]) вместо recvfrom, если знаете, что не должны выяснить, кто послал пакет. Даже read, может использоваться, если Вы не хотите определять flags (см. Раздел 8.2 [Примитивы ввода - вывода]).

- 289 -

11.9.3 Датаграмный Пример сокета.

Вот набор примеров программ, которые посылают сообщения используя датаграмный стиль. И клиент и сервер используют функцию make_named_socket, которая была предоставлена в Разделе 11.4 [Именное пространство Файла], для создания и связывания сокетов.

Сначала программа сервера. Очевидно, это не особенно полезная программа, но она показывает общие идеи.

```
#include
#include
#include
#include
#include
#define SERVER  "/tmp/serversocket"
#define MAXMSG  512
int
main (void)
{
        int sock;
        char message[MAXMSG];
        struct sockaddr_un name;
        size_t size;
        int nbytes;
        sock = make_named_socket (SERVER);
        while (1)
        {
                size = sizeof (name);
                nbytes=recvfrom(sock,message,MAXMSG,0,
                        (struct sockaddr*) & name,&size);
                if (nbytes < 0)
                {
                        perror ("recfrom (server)");
                        exit (EXIT_FAILURE);
                }
                fprintf (stderr, "Server: got message:
                %s\n", message);

               - 290 -

                nbytes=sendto(sock,message,nbytes,0,
                        (struct sockaddr*) & name,size);
                if (nbytes < 0)
                {
                        perror ("sendto (server)");
                        exit (EXIT_FAILURE);
                }
        }
}
```

11.9.4 Пример Чтения Датаграмм

Вот программа клиента, соответствующая серверу выше.

Она посылает датаграмму серверу и ждет ответ. Обратите внимание, что сокету клиента (также как для сервера) в этом примере должно быть дано имя. Так, чтобы сервер мог направлять сообщение обратно клиенту. Так как сокет не имеет никакого связанного состояния соединения, единственый способ, которым сервер может сделать это ссылаясь на имя клиента.

```
#include
#include
#include
#include
#include
#include
#define SERVER  "/tmp/serversocket"
```

```
                #define CLIENT  "/tmp/mysocket"
                #define MAXMSG  512
                #define MESSAGE "Yow!!! Are we having fun yet?!?"
                int
                main (void)
                {
                        extern int make_named_socket (const
                char *name);
                        int sock;
                        char message[MAXMSG];

                               - 291 -

                        struct sockaddr_un name;
                        size_t size;
                        int nbytes;
                        sock = make_named_socket (CLIENT);
                        name.sun_family = AF_UNIX;
                        strcpy (name.sun_path, SERVER);
                        size = strlen (name.sun_path) +
        sizeof (name.sun_family);
                        nbytes = sendto (sock, MESSAGE,
        strlen (MESSAGE) + 1, 0,                                (struct
sockaddr *) & name, size);
                        if (nbytes < 0)
                        {
                                perror ("sendto (client)");
                                exit (EXIT_FAILURE);
                        }
                nbytes =recvfrom(sock,message,MAXMSG,0,NULL,0);
                if (nbytes < 0)
                {
                        perror ("recfrom (client)");
                        exit (EXIT_FAILURE);
                }
                fprintf (stderr,"Client: got message:
        %s\n",message);
                        remove (CLIENT);
                        close (sock);
                }
```

Имейте в виду, что датаграмная связь сокетов ненадежна. В этом примере программа клиента ждет неопределенное время, если сообщение никогда не достигает сервера, или, если ответ сервера никогда не возвращается. Более автоматическое решение могло бы использовать select (см. Раздел 8.6 [Ждущий ввод - вывод]), чтобы установить период блокировки по времени для ответа, и в этом случае или снова послать сообщение, или выключить сокет и выйти.

11.10 Демон Inetd

Мы объяснили выше, как написать программу сервера, которая реализует собственное ожидание. Такой сервер должен уже выполниться для любого соединения с ним.

Другой способ обеспечивать обслуживание портов для Internet состоит в том, чтобы использовать в программе для ожидания демона inetd. Inetd - программа, которая выполняется все время и ждет (используя select) сообщения на заданном наборе портов. Когда она получает сообщение, она принимает соединение (если стиль сокета запрашивает соединение), и тогда запускает дочерний процесс, чтобы выполнить соответствующую программу сервера. Вы определяете порты и их программы в файле "/etc/inetd.conf".

11.10.1 Inetd Серверы

Написание программы сервера, которая будет выполнена inetd очень просто. Каждый раз когда кто-то запрашивает соединение с соответствующим портом, стартует новый процесс сервера. Соединение уже существует в это время; гнездо доступно как описатель

стандартного ввода и как описатель стандартного вывода (описатели 0 и 1) в процессе сервера. Так что программа сервера может начинать читать и писать данные сразу же. Часто программа нуждается только в обычных средствах ввода-вывода; фактически, универсальная программа-фильтр, которая не знает ничего относительно сокетов, может работать как сервер потока байтов, запускаемая inetd.

Вы можете также использовать inetd для серверов, которые используют стили связи без установления логического соединения. Для этих серверов, inetd не пробует принять соединение, так как никакое соединение не возможно. Она только начинает программу сервера, которая может читать входящий датаграмный пакет из описателя 0. Программа сервера может обрабатывать один запрос и выходить, или читать большое количество запросов. Вы должны определить, который из этих двух методов использования сервера удобен Вам при конфигурации inetd.

11.10.2 Конфигурирование inetd

Файл "/etc/inetd.conf " сообщает inetd, какие порты ожидает какой сервер для обработки пакетов. Обычно каждый вход в файле - это строка, но Вы можете разбивать его на много строк, если все, кроме первой строки входа, начинаются с пропуска. Строки, которые начинаются с "*" являются комментариями.

Имеются два стандартных входа в "/etc/inetd.conf":

```
ftp stream tcp nowait root /libexec/ftpd ftpd
talk dgram udp wait root /libexec/talkd talkd
```

Вход имеет формат:

```
service style protocol wait username program arguments
```

Поле service говорит, какое обслуживание обеспечивает эта программа. Это должно быть имя обслуживания, определенного в "/etc/services". Inetd использует обслуживание, чтобы решить какой порт слушать для этого входа.

style и protocol определяют стиль связи и протокол для использования ожидающего сокета. Стиль должен иметь имя стиля связи, преобразованного в строчные буквы и с удаленным "SOCK_", например, "stream" или "dgram". Протокол должен быть один из протоколов, перечисленных в "/etc/protocols". Типичные имена протокола - "tcp" для соединений потока байтов и "udp" для ненадежных датаграмм.

Поле wait должно быть, или "wait" или "nowait". "wait" используется, если стиль не требует установления логического соединения и обрабатывает многократные запросы. "nowait" используется, когда необходимо,чтобы inetd начинал новый процесс для каждого сообщения, или запроса, который приходит. Если используется соединение, то wait должен быть "nowait".

user - имя пользователя, под которым сервер должен выполняться. Inetd выполняется под пользователя root, так что она может устанавливать ID пользователей дочерних процессов произвольно. Лучше избегать

использования "root" для пользователя; но некоторые серверы, типа Telnet и FTP, читают username и пароль самостоятельно. Эти серверы должны быть запущены под пользователя root изначально, так как они могут регистрировать потоки данных передаваемых по сети.

program вместе с аргументами определяет команду, для запуска сервера. Это должно быть абсолютное имя файла, определяющее исполняемый файл для выполнения. Аргументы состоят из любого числа отделенных пробелами слов, которые станут аргументами командной строки программы.

Первое слово в аргументах - нуль, который должен быть именем программы непосредственно (каталоги sans).

Если Вы редактируете "/etc/inetd.conf", то Вы можете указывать необходимость повторного чтения файла для inetd и сообщения нового содержимого, посылая inetd сигнал SIGHUP. Вы будете должны использовать ps, чтобы определить ID процесса inetd, поскольку оно не фиксировано.

11.11 Опции сокетов.

Этот раздел описывает, как читать или установить различные опции, которые изменяют поведение сокетов и их основных протоколов связи.

Когда Вы манипулируете опциями сокета, Вы должны определить, к какому уровню они относятся, то есть применяется ли опция к интерфейсу сокета, или к интерфейсу протокола связи низшего уровня.

11.11.1 Функции Опций сокета.

Имеются функции для исследования и изменения опций сокета. Они объявлены в "sys/socket.h".

int getsockopt (int socket, int level, int optname, void *optval, size_t *optlen_ptr)

Функция getsockopt получает информацию относительно значения опции optname заданного уровня для указанного сокета.

Значение опции сохранено в буфере, на который указывает optval. Перед обращением, Вы должны обеспечить в * optlen_ptr размер этого буфера; по возвращении, он содержит число байтов информации, фактически сохраненной в буфере.

Большинство опций интерпретирует буфер optval как одиночное значение int.

Фактически возвращаемое значение getsockopt - 0 при успехе и -1 в случае неудачи. В переменной errno отражены следующие возможные причины:

EBADF аргумент socket - не допустимый описатель файла.

ENOTSOCK дескриптор socket - не сокет.

ENOPROTOOPT Optname не имеет смысла для данного уровня.

int setsockopt (int socket, int level, int optname, void *optval, size_t optlen)

Эта функция используется, чтобы установить опцию сокета optname заданного уровня для указанное сокета. Значение опции передано в буфере optval, который имеет размер optlen.

Возвращаемое значение и коды ошибки для setsockopt такие же как для getsockopt.

11.11.2 Опции уровня сокета.

int SOL_SOCKET (константа)

Используйте эту константу, как аргумент level для getsockopt или setsockopt, чтобы манипулировать опциями уровня сокета, описанными в этом разделе.

Вот таблица имен опций уровня сокета; все они определены в файле "sys/socket.h".

SO_DEBUG

Эта опция переключает запись информации об отладке в основных модулях протокола. Значение имеет тип int; значение отличное от нуля означает "да".

SO_REUSEADDR

Эта опция говорит bind (см. Раздел 11.3.2 [Установка Адреса]) разрешить многократное использование местных адресов для этого сокета. Если Вы пользуетесь этой опцией, Вы можете фактически иметь два сокета с тем же самым номером порта Internet. Необходимость в этой опции возникает, потому что что некоторые протоколы Internet с более высоким уровнем, такие FTP, требуют, чтобы Вы многократно использовали тот же самый номер сокета.

Значение имеет тип int; значение отличное от нуля означает "да".

SO_KEEPALIVE

Эта опция указывает, должен ли основной протокол периодически передавать сообщения на соединенный сокет. Если адресат будет не в состоянии отвечать на эти сообщения, соединение рассматривается разорваным. Значение имеет тип int; значение отличное от нуля

означает "да".

SO_DONTROUTE

Эта опция контролирует при посылке сообщения обход нормальных средств посылки сообщений. Если она установлена, сообщения посылаются непосредственно сетевому интерфейсу. Значение имеет тип int; значение отличное от нуля означает "да".

SO_LINGER

Эта опция определяет то, что должно случиться, в случае, когда сокет предоставляющий надежную выдачу все еще не передал сообщения, до закрытия; см. Раздел 11.7.2 [Закрытие сокета]. Значение имеет тип struct linger.

struct linger (тип данных)

Эта структура имеет следующие элементы:

int l_onoff

Это поле интерпретируется как булевское. Оно отлично от нуля, если блокировка закрыта, пока данные не переданы, или период блокировки по времени не истек.

int l_linger

Определяет период блокировки по времени, в секундах.

SO_BROADCAST

Эта опция определяет могут ли датаграммы быть широковещательно переданы из сокета. Значение имеет тип int; значение отличное от нуля означает "да".

SO_OOBINLINE

Если эта опция установлена, данные вне потока получаемые в сокет помещаются в нормальную входную очередь. Она разрешает читать их, используя read или recv без того, чтобы определить флаг MSG_OOB. См. Раздел 11.8.8 [Данные вне потока]. Значение имеет тип int; значение отличное от нуля означает "да".

SO_SNDBUF

Эта опция получает или устанавливает размер буфера вывода. Значение size_t является его размером в байтах.

SO_RCVBUF

Эта опция получает или устанавливает размер буфера ввода. Значение size_t является его размером в байтах.

SO_STYLE

SO_TYPE

Эта опция может использоваться только с getsockopt. Она используется, чтобы получить стиль связи сокета. SO_TYPE - историческое имя, а SO_STYLE - привилегированное имя в GNU. Значение имеет тип int, и обозначает стиль связи; см. Раздел 11.2 [Стили Связи].

SO_ERROR

Эта опция может использоваться только с getsockopt. Она используется, чтобы сбросить состояние ошибки сокета. Значение int представляет собой предыдущее состояние ошибки.

11.12 База данных Сетей

Много систем приходят с базой данных, которая записывает список сетей, известных разработчику системы. Она обычно сохраняется или в файле "/etc/networks" или в блоке преобразования имен. Эта база данных полезна для маршрутизации программ типа route, но бесполезна для программ, которые просто связываются по сети. Функции предназначенные для обращения к этой базе данных описаны в "netdb.h ".

struct netent (тип данных)

Этот тип данных используется, чтобы представить информацию относительно входов в базе данных сетей.

Он имеет следующие элементы:

char *n_name

Это - "официальное" имя сети.

char **n_aliases

Это альтернативные имена для сети, представляемые как вектор строк. Пустой указатель завершает массив.

int n_addrtype

Это - тип сетевого номера; он всегда равно AF_INET для сетей Internet.

unsigned long int n_net

Это сетевой номер. Сетевые числа представлены в главном порядке байтов; см. Раздел 11.5.5 [Порядок Байтов].

struct netent * getnetbyname (const char *name) (функция)

Getnetbyname функция возвращает информацию относительно сети

именованной name. Она возвращает пустой указатель, если нет никакой сети.

struct netent * getnetbyaddr (long net, int type) (функция)

Функция getnetbyaddr возвращает информацию относительно сети указанного типа с номером net. Вы должны определить значение AF_INET для аргумента type для сети Internet.

getnetbyaddr возвращает пустой указатель в случае отсутствия такой сети.

Вы можете также просматривать базу данных сетей, используя setnetent, getnetent, и endnetent. Будьте внимательным в использовании этих функций, потому что они не предназначены для повторного использования.

void setnetent (int stayopen)

Эта функция открывает базу данных сетей.

Если аргумент stayopen является отличным от нуля, то она устанавливает флаг так, чтобы последующие обращения к getnetbyname или getnetbyaddr не закрыли базу данных. Это делается для большей эффективности, если Вы вызываете эти функции несколько раз, избегая повторного открытия базы данных для каждого обращения.

struct netent * getnetent (void)

Эта функция возвращает следующий вход в базе данных сетей. Она возвращает пустой указатель, если не имеется больше входов.

void endnetent (void)

Эта функция закрывает базу данных сетей.

12. Интерфейс Терминала низкого уровня

Эта глава описывает функции, которые являются специфическими для терминальных устройств. Вы можете использовать эти функции для действий аналогичным выключению входного отображения на экране, установки характеристик строки типа быстродействия строки и управления потоком данных, и изменения символов используемых для обозначения конца файла, редактирования командной строки, посылки сигналов, и подобных функций управления.

Большинство функций в этой главе использует на описатели файлов. См. Главу 8 [ввод-вывод низкого уровня] для получения более подробной информации, чем описатель файла является и как открыть описатель файла для устройства терминала.

12.1 Идентификация Терминалов

Функции, описанные в этой главе работают с файлами, которые соответствуют терминальному устройству. Используя функцию isatty, Вы можете выяснять, связан ли описатель файла с терминалом.

Прототипы и для isatty и ttyname объявлены в файле "unistd.h".

int isatty (int filedes) (функция)

Эта функция возвращает 1, если filedes - описатель файла, связанный с открытым терминальным устройством, и 0 в противном случае.

Если описатель файла связан с терминалом, Вы может получать связанное имя файла, используя функцию ttyname. См. также ctermid функцию, описанную в Разделе 24.7.1 [Идентификация Терминала].

char * ttyname (int filedes) (функция)

Если описатель файла filedes связан с терминальным устройством, то функция ttyname возвращает указатель на статически размещенную строку с нулевым символом в конце, содержащую имя файла терминала. Значение - пустой указатель, если описатель файла не связан с терминалом, или имя файла не может быть определено.

12.2 Очереди Ввода-вывода

Многие из функций в этом разделе обращаются к очередям ввода и вывода терминального устройства. Эти очереди определяют форму буферизации внутри ядра, которое независит от буферизации, выполненной потоками ввода-вывода (см. Главу 7 [ввод-вывод на Потоках]).

Входная очередь терминала также иногда упоминается как буфер клавиатуры. Она содержит символы, которые были получены от терминала, но еще не прочитанны ни каким процессом.

Размер входной очереди терминала описан параметрами _POSIX _MAX_INPUT и MAX_INPUT; см. Раздел 27.6 [Ограничения для Файлов]. Если управление потоком данных ввода допускается установкой бита режима ввода IXOFF (см. Раздел 12.4.4 [Входные Режимы]), драйвер терминала передает символы STOP и START на терминал, когда необходимо предохранить очередь от переполнения. Иначе, ввод можно потерять при сликшом быстром поступлении данных с терминала. (Это маловероятно, если Вы печатаете ввод вручную!)

Очередь вывода терминала подобна входной очереди, но для вывода она содержит символы, которые написаны процессами, но еще не передавались на терминал. Если управление потоком данных вывода допускается установкой бита режима ввода IXON (см. Раздел 12.4.4 [Входные Режимы]), то драйвер терминала удовлетворяет условиям символов STOP и STOP, посланных терминалом, чтобы остановиться и перезапустить передачу вывода.

Очистка входной очереди терминала означает отбрасывание любых символов, которые были получены, но еще не прочитаны. Аналогично, очистка очереди вывода терминала означает отбрасывание любых символов, которые написаны, но еще не передавались.

12.3 Два Стиля Ввода: канонический и неканонический.

Системы POSIX поддерживают два базисных режима ввода: канонический и неканонический.

В каноническом входном режиме ввод с терминала обрабатывается построчно, строка оканчивается символом перевода строки ("\n"), символами EOF, или EOL. Ввод не может закончиться, пока вся строка не напечатана пользователем.

В каноническом входном режиме, операционная система обеспечивает входное редактирование: символы ERASE и KILL интерпретируются особо для выполнения операций редактирования внутри текущей строки текста. См. Раздел 12.4.9.1 [Редактирование Символов].

Константы _POSIX_MAX_CANON и MAX_CANON указывают максимальное число байтов, которые могут появляться в одиночной строке канонического ввода. См. Раздел 27.6 [Ограничения для Файлов].

В неканоническиом входном режиме обработки символы не сгруппированы в строки, и ERASE, и KILL не выполняются. Степень детализации, с которой байты читаются в неканоническиом входном режиме, управляется MIN и TIME. См. Раздел 12.4.10 [Неканонический Ввод].

Большинство программ использует канонический входной режим, потому что это дает пользователю способ редактировать входную строку. Обычной причиной для использования неканонического режима является необходимость программой принимать одиночно-символьные команды или предоставлять собственные средства редактирования.

Выбор канонического или неканонического ввода управляется флагом ICANON в элементе _lflag в struct termios. См. Раздел 12.4.7 [Автономные режимов].

12.4 Режимы Терминала

Этот раздел описывает различные атрибуты терминала, которые управляют вводом и выводом. Функции, структуры данных, и символические константы объявлены в файле " termios.h".

12.4.1 Типы Данных Режима Терминала

Вся коллекция атрибутов терминала сохранена в структуре типа struct termios. Эта структура используется функциями tcgetattr и tcsetattr, чтобы читать и установливать атрибуты.

struct termios (тип данных)

Это структура, которая записывает все атрибуты ввода-вывода терминала. Структура включает по крайней мере следующие элементы:

tcflag_t c_iflag

Битовая маска, определяющая флаги для режимов ввода; см. Раздел 12.4.4 [Режимы Ввода].

tcflag_t c_oflag

Битовая маска, определяющая флаги для режимов вывода; см. Раздел 12.4.5 [Режимы вывода].

tcflag_t c_cflag

Битовая маска, определяющая флаги для режимов управления; см. Раздел 12.4.6 [Режимы Управления].

tcflag_t c_lflag

Битовая маска, определяющая флаги для автономных режимов; см. Раздел 12.4.7 [Автономные режимы].

cc_t c_cc[NCCS]

Массив, определяющий, символы связаные с различными функциями управления; см. Раздел 12.4.9 [Специальные Символы].

Структура struct termios также содержит элементы, которые кодируют скорости передачи ввода и вывода, но представление не преднано. См. Раздел 12.4.8 [Быстродействие Строки], для того, как исследовать и сохранять значения быстродействия.

Следующие разделы описывают подробности элементов структуры struct termios.

tcflag_t (тип данных)

Это тип integer unsigned, используемый, чтобы представить различные битовые маски для флагов терминала.

cc_t (тип данных)

Это тип integer unsigned, используемый, чтобы представить символы, связанные с различными функциями управления терминала.

int NCCS (макрос)

Значение этой макрокоманды - число элементов в массиве c_cc.

12.4.2 Функции Режимов Терминала

int tcgetattr (int filedes, struct termios *termios_p) (функция)

Эта функция используется, чтобы исследовать атрибуты терминального устройства с описателем файла filedes. Атрибуты возвращены в структуре на которую указывает termios_p.

В случае успеха tcgetattr возвращает 0. Возвращаемое значение -1 указывает ошибку. Следующие условия ошибки errno определены для

этой функции:

EBADF filedes аргумент - не допустимый описатель файла.

ENOTTY filedes не связан с терминалом.

int tcsetattr (int filedes, int when, const struct termios *termios_p) (функция)

Эта функция устанавливает атрибуты устройства терминала с описателем файла filedes. Новые атрибуты принимаются из заданной структуры.

Аргумент when определяет, как поступать с вводом и выводом, уже поставленным в очередь. Это может быть одно из следующих значений:

TCSANOW Делать изменения немедленно.

TCSADRAIN Делать изменения после ожидания, пока весь поставленный в очередь вывод не написан.

Вы должны обычно использовать эту опцию при изменении параметров, которые воздействуют на вывод.

TCSAFLUSH подобен TCSADRAIN, но отбрасывает любой поставленный в очередь ввод.

TCSASOFT Это бит флага, который Вы можете добавлять к любому из вышеупомянутых вариантов. Значение должно запретить чередование состояния аппаратных средств терминала. Это BSD расширение; он не имеет никакого эффекта в не-BSD системах.

Если эта функция вызывается из фонового процесса на терминале управления, то все процессы в группе этого процесса посылается сигнал SIGTTOU, как будто процесс пробовал запись на терминал. Исключение - если процесс вызова непосредственно игнорирует или блокирует сигналы SIGTTOU См. Главу 24 [Управление заданиями].

При успехе tcsetattr возвращает 0. Возвращаемое значение -1 указывает ошибку. Следующие условия ошибки errno определены для этой функции:

EBADF filedes аргумент - не допустимый описатель файла.
ENOTTY filedes не связан с терминалом.

EINVAL или значение аргумента when не допустимо, или ошибка с данными в аргументе termios_p .

Хотя tcgetattr и tcsetattr определяют устройство терминала описателем файла, атрибуты - сами устройства терминала непосредственно, а не описателя файла. Это означает, что эффект изменения атрибутов терминала постоянен; если другой процесс открывает файл терминала позже, он будет видеть измененные атрибуты.

Аналогично, если одиночный процесс имеет многократные или дублированные описатели файла для того же самого устройства терминала, замена атрибутов терминала воздействует на ввод и вывод для всех этих описателей файлов.

12.4.3 Установка Режимов Терминала Правильно

Когда Вы устанавливаете режимы терминала, Вы должны сначала вызвать tcgetattr, чтобы получить текущие режимы специфического устройства терминала, и изменять только те режимы, в которых Вы действительно заинтересованы. Для сохранения результата испольуйте tcsetattr.

Плохая практика просто инициализировать структуру struct termios для выбранного набора атрибутов и передавать ее непосредственно в tcsetattr. Ваша программа может быть выполнена через годы на системах, которые поддерживают элементы, не зарегистрированные в этом руководстве.

Более того различные терминальные устройства могут требовать различных установок режима. Так что Вы должны избегать слепого копирования атрибутов из одного терминального устройства в другое.

Вот пример того, как устанавливать один флаг (ISTRIP) в структуре struct termios при правильном сохранении всех других данных в структуре:

```
int
set_istrip (int desc, int value)
{
        struct termios settings;
        int result;
        result = tcgetattr (desc, &settings);
        if (result < 0)
        {
                perror ("error in tcgetattr");
                return 0;
        }
        settings.c_iflag &= ~ISTRIP;
        if (value)
                settings.c_iflag |= ISTRIP;
        result = tcgetattr (desc, &settings);
        if (result < 0)
        {
                perror ("error in tcgetattr");
                return;
        }
        return 1;
}
```

12.4.4 Режимы Ввода

Этот раздел описывает флаги атрибутов терминала, которые управляют аспектами низкого уровня входной обработки: обработка ошибок контроля четности, сигналы останова, управление потоком данных, символы и RET и LFD.

Все эти флаги - биты в c_iflag элементе структуры struct termios. Элемент - integer, и Вы изменяете флаги, использующие операторами &, | и ^. Не пробуйте определять все значение для c_iflag_instead, изменяйте только специфические флаги и оставляйте

остаток нетронутым (см. Раздел 12.4.3 [Режимы Установки]).

INPCK если этот бит установлен, допускается входная проверка контроля четности. Если он не установлен, никакой проверки ошибок четности нет; символы просто передаются к приложению.

Если этот бит установлен, то при ошибки контроля четности ситуация зависит от того, установлены ли биты IGNPAR или PARMRK. Если никакой из этих битов не установлен, байт с ошибкой контроля четности, передается приложению как символ '\0'.

IGNPAR если этот бит установлен, любой байт с ошибкой четности игнорируется. Это полезно только, если также установлен INPCK.

PARMRK если этот бит установлен, входные байты с ошибкой четности или ошибкой синхронизации отмечены при передачи программе. Этот бит значим только, когда INPCK установлен, и IGNPAR не установлен.

Ошибочные байты отмечены двумя предшествующими байтами, 377 и 0. Таким образом, программа фактически читает три байта для одного ошибочного байта, полученного от терминала.

Если допустимый байт имеет значение 0377, и ISTRIP (см. ниже) не установлен, программа может путать его с префиксом, который отмечает ошибку контроля по четности. Так что допустимый байт 0377 передается программе как два байта, 0377 0377.

ISTRIP если этот бит установлен, допустимые входные байты урезаны до семи битов; иначе, все восемь битов доступны для программ.

IGNBRK если этот бит установлен, условия прерывания игнорируются.

Условие прерывания определено в контексте асинхронной последовательной передачи данных как ряд нулевых битов длиннее байта.

BRKINT если этот бит установлен, и IGNBRK не установлен, условие прерывания очищает ввод терминала и очереди вывода и поднимает сигнал SIGINT для группы приоритетного процесса, связанной с терминалом.

Если ни BRKINT ни IGNBRK не установлены, условие прерывания передается приложению как одиночный символ '\0', если PARMRK не установлен, или иначе как последовательность из трех символов "\377", "\0","\0".

IGNCR Если этот бит установлен, символы возврата каретки ("\r") отброшены на вводе. Отбрасывание возврата каретки может быть полезно на терминалах, которые посылают, и возврат каретки и перевод строки при нажатии клавишу RET.

ICRNL Если этот бит установлен, и IGNCR не установлен, символ возврата каретки (" \r ") передается к приложению как символ перевода строки (" \n ").

INLCR Если этот бит установлен, символ перевода строки ("\n") передан к приложению как символ возврата каретки ("\r").

IXOFF Если этот бит установлен, допускается start/stop контроль над вводом. Другими словами, компьютер посылает символы STOP и START по мере необходимости, чтобы предотвратить ввод от прибытия быстрее чем программы его читают. Идея состоит в том, что фактические аппаратные средства терминала, которые генерируют входные данные, отвечают на символ STOP, приостанавливая передачу, а на символ START, продолжая передачу. См. Раздел 12.4.9.4 [Start/Stop Символы].

IXON Если этот бит установлен, допускается start/stop контроль

над выводом. Другими словами, если компьютер получает символ STOP, он приостанавливает вывод, пока символ START не получен.

В этом случае, символы STOP и START никогда не переданы прикладной программе. Если этот бит не установлен, то START и STOP,

могут читаться как обычные символы. См. Раздел 12.4.9.4 [Start/Stop Символы].

IXANY если этот бит установлен, любой входной символ перезапускает вывод, когда вывод был приостановлен символом STOP. Иначе, только символ START перезапускает вывод.

IMAXBEL если этот бит установлен, то при заполнении буфера ввода терминала посылается символ BEL (код 007) на терминал (звенит звонок).

12.4.5 Режимы вывода

Этот раздел описывает флаги терминала и поля, которые управляют, как выводимые символы транслируются и дополняются для дисплея. Все они содержатся в c_oflag элементе struct termiosstructure.

C_oflag элемент непосредственно - integer, и Вы изменяете флаги и поля, используюя операторы &, |, и ^. Не пробуйте определять все значения для c_oflag , изменяйте только специфические флаги и оставляйте остаток нетронутым (см. Раздел 12.4.3 [Режимы Установки]).

int OPOST (макрос)

Если этот бит установлен, выходные данные обрабатываются некоторым неопределенным способом так, чтобы они отобразились соответственно на устройстве терминала. Это обычно включает отображение символа перевода строки ("\n") на пару перевод строки и возврат каретки.

Если этот бит не установлен, символы передаются как есть.

Следующие три бита возможности BSD, и они не имеют никакого эффекта на не-BSD системах. На всех системах, они эффективны только, если OPOST установлен.

int ONLCR (макрос)

Если этот бит установлен, преобразовывает символ перевода строки на выводе в пару символов: возврат каретки, сопровождаемый переводом строки.

int OXTABS (макрос)

Если этот бит установлен, преобразовывает символы табуляции на выводе в соответствующее число пробелов, чтобы эмулировать табулятор через каждые восемь столбцов.

int ONOEOT (макрос)

Если этот бит установлен, отбрасываются символы C-d (код 004) на выводе. Эти символы заставляют терминалы разъединиться.

12.4.6 Режимы Управления

Этот раздел описывает флаги терминала и поля, которые управляют параметрами, обычно связываемыми с асинхронной последовательной передачей данных. Эти флаги могут не иметь смысла для других видов портов терминала (типа псевдо-терминального сетевого соединения). Все они содержатся в элементе c_cflag структуры struct termios.

Элемент c_cflag непосредственно integer, и Вы изменяете флаги

и поля, используюя операторы &, |, и ^. Не пробуйте определять все значения для c_cflag взамен, изменяйте только специфические флаги и оставляйте остаток нетронутым (см. Раздел 12.4.3 [Режимы Установки]).

CLOCAL если этот бит установлен, это указывает, что терминал соединен "локально", и что строки состояния модема должны игнорироваться.

Если этот бит не установлен и Вы вызываете open без O_NONBLOCK набора флагов, open блокируются, пока соединение модема не установлено.

Если этот бит не установлен, а модем обнаружен, то посылается сигнал SIGHUP группе процесса управления терминала. Обычно, это вызывает завершение процессов; см. Главу 21 [Обработка Сигнала]. Чтение из терминала после разъединения дает условие конца файла, а запиись вызывает ошибку EIO. Для очищения бита устройство терминала должно быть закрыто и вновь открыто.

HUPCL Если этот бит установлен, то использующие это терминальное устройство процессы по окончании или закрытии файла вызовут разъединение с модемом.

CREAD Если этот бит установлен, ввод может читаться из терминала. Иначе, поступающий ввод отбрасывается.

CSTOPB если этот бит установлен, используются два стоповых бита. Иначе, используется только один стоповый бит .

PARENB если этот бит установлен, допускается порождение и обнаружение бита контроля четности. См. Раздел 12.4.4 [Входные Режимы], для уточнения информации о том, как обрабатываются входные ошибки контроля четности .

Если этот бит не установлен, никакой бит контроля четности не добавлен при выводе символов, и входные символы не проверены по четности.

PARODD Этот бит полезен только, если PARENB установлен. Если PARODD установлен, используется проверка на нечетность, иначе используется проверка на четность.

Флаги режима управления также включают поле для числа битов на символ. Вы можете использовать CSIZE макрокоманду как маску, чтобы извлечь значение, примерно так:

```
settings.c_cflag & CSIZE.
```


CSIZE Это маска для числа битов на символ.
CS5 Это определяет пять битов на байт.
CS6 Это определяет шесть битов на байт.
CS7 Это определяет семь битов на байт.
CS8 Это определяет восемь битов на байт.
CCTS_OFLOW Если этот бит установлен, он дает возможность управления потоком выводимых данных, основанного на CTS (RS232 протокол).

CRTS_IFLOW

Если этот бит установлен, он дает возможность управления потоком вводимых данных, основанного на RTS (RS232 протокол).

MDMBUF если этот бит установлен, он предоставляет возможность курьерскому управлению потоком выводимых данных.

12.4.7 Автономные режимы

Этот раздел описывает флаги для элемента c_lflag структуры struct termios. Эти флаги управляют аспектами с более высоким уровнем ввода, чем флаги режимов, описанные в Разделе 12.4.4 [Входные Режимы], типа отображения на экране, сигналов, и выбора канонического или неканонического ввода.

Элемент c_lflag непосредственно - integer, и Вы изменяете флаги и поля, используюя операторы &, |, и ^. Не пробуйте определять все значения для c_lflag, а изменяйте только специфические флаги и оставляйте остаток нетронутым (см. Раздел 12.4.3 [Режимы Установки]).

ICANON Этот бит, устанавливает канонический режим обработки ввода. Иначе, ввод обработан в неканоническом режиме. См. Раздел 12.3 [Каноническиий или Нет].

ECHO если этот бит установлен, допускается отображение входных символов обратно на терминал.

- 315 -

ECHOE если этот бит установлен, происходит отображение стирания ввода символом ERASE, уничтожается последний символ в текущей строке. Иначе, уничтоженный символ переотображен, чтобы показать что случилось.

Этот бит управляет только поведением дисплея; ICANON бит управляет фактическим распознаванием символа ERASE и стиранием ввода.

ECHOK Этот бит дает возможность специальному отображению символа KILL. Имеются два способа, которыми это может быть выполнено. Лучший способ стирать всю строку. Худший способ перемещение в новую строку после отображения на экране символа KILL.

Если этот бит не установлен, символ KILL отображается, точно как это было бы, если не было символа KILL. Ввод предшестующий символу KILL не отображается.

ECHONL если этот бит установлен, и бит ICANON также установлен, то символ перевода строки ("\n") отображается на экране, даже если бит ECHO не установлен.

ISIG Этот бит управляет распознаванием символов INTR, QUIT, и SUSP. Функции, связанные с этими символами выполняются только, если этот бит установлен.

В каноническом или неканоническом входном режиме терминал может не реагировать на интерпретацию этих символов.

IEXTEN Этот бит подобен ISIG, но контролирует определенные реализацией специальные символы. Если он установлен, он мог бы отменять заданное по умолчанию поведение для ICANON и ISIG флагов автономного режима, и IXON и флагов режима ввода IXOFF.

NOFLSH Обычно, символы INTR, QUIT, и SUSP очищают очереди ввода и вывода для терминала. Если этот бит установлен, очереди, не очищаются.

- 316 -

TOSTOP если этот бит установлен, и система поддерживает управление заданиями, то фоновыми процессами генерируются SIGTTOU сигналы, при попытке записи на терминал. См. Раздел 24.4 [Доступ к Терминалу].

Следующие биты представляют собой расширения BSD; библиотека GNU определяет эти символы в любой системе, но использоваться они будут только в системах BSD.

ECHOKE на системах BSD этот бит выбирает между двумя альтернативными способами отображения символа KILL, когда ECHOK установлен. Если ECHOKE установлен, то символ KILL стирает целую

экранную строку; иначе, символ KILL перемещается в следующую экранную строку. Установка ECHOKE не производит никакого эффекта, если бит ECHOK не установлен.

ECHOPRT этот бит дает возможность отображать символ ERASE заданным способом.

ECHOCTL если этот бит установлен, управляющие символы отображаются с "^" сопровождаемый соответствующим текстовым символом. Таким образом, control-A отображается на экране как "^A".

ALTWERASE Этот бит определяет сколько символов WERASE должен стереть. Символ WERASE стирает обратно к началу слова.

Если этот бит является неустановлен, то началом слова считается первый символ после символов пробел. Если бит установлен, то начало слова - алфавитно-цифровой символ или подчеркивание после символа, который не является одним из этих.

FLUSHO - бит, который переключается, когда пользователь печатает символ DISCARD. Если этот бит установлен то вывод не производится.

NOKERNINFO Установка этого бита отключает обработку символа STATUS.

PENDIN если бит установлен, то это означает, что имеется строка ввода, которая должна быть перпечатана.

Печать символа REPRINT устанавливает этот бит; бит остается установленным, пока перепечатывание не закончено. См. Раздел 12.4.9.2 [Редактирование в BSD].

12.4.8 Быстродействие Строки

Быстродействие строки терминала сообщает компьютеру как быстро читать и писать данные относительно терминала.

Если терминал соединен с реальной последовательной строкой, быстродействие терминала, которое Вы определяете фактически управляет строкой, если она не соответствует собственной концепции терминала относительно быстродействия, связь не работает.

Реальные последовательные порты воспринимают только некоторые стандартные скорости. Специфические аппаратные средства не могут поддерживать все стандартные скорости. При определении нулевой скорости "зависает " соединение телефонного вызова по номеру и выключаются сигналы управления модема.

Если терминал не реальная последовательная строка (например, если это сетевое соединение), то скорость не будет воздействовать на быстродействие передачи данных, но некоторые программы используют ее, чтобы определить количество необходимого дополнения. Самое лучшее определить значение быстродействия строки, которое соответствует фактическому быстродействию фактического терминала, но Вы можете безопасно экспериментировать с различными значениями, чтобы изменить количество дополнения.

Имеются фактически две скорости строки для каждого терминала, один для ввода и один для вывода. Вы можете устанавливать их независимо, но наиболее часто терминалы используют то же самое быстродействие для обоих направлений.

Значения быстродействия сохранены в структуре struct termios, но не пробуйте обращаться к ним в структуре struct termios

непосредственно. Взамен, Вы должны использовать следующие функции, чтобы читать и сохранять их:

speed_t cfgetospeed (const struct termios *termios_p) (функция)

Эта функция возвращает быстродействие строки вывода, сохраненное в структуре * termios_p.

speed_t cfgetispeed (const struct termios *termios_p) (функция)

Эта функция возвращает входное быстродействие строки, сохраненное в структуре * termios_p.

int cfsetospeed (struct termios *termios_p, speed_t speed) (функция)

Эта функция сохраняет быстродействие в * termios_p как быстродействие вывода. Нормальное возвращаемое значение 0; значение -1 указывает ошибку. Если speed - не быстродействие, cfsetospeed возвращает -1.

int cfsetispeed (struct termios *termios_p, speed_t speed) (функция)

Эта функция сохраняет быстродействие в * termios_p как входное быстродействие. Нормальное возвращаемое значение - 0; значение -1 указывает ошибку. Если speed - не быстродействие, cfsetospeed возвращает -1.

int cfsetspeed (struct termios *termios_p, speed_t speed) (функция)

Эта функция сохраняет быстродействие в * termios_p, и как скорости вывода и как скорость ввода. Нормальное возвращаемое значение 0; значение -1 указывает ошибку. Если speed не быстродействие, cfsetspeed возвращает -1. Эта функция расширение 4.4 BSD.

speed_t (тип данных)

Speed_t тип данных integer unsigned, используемый, чтобы представить скорости строки.

Функции cfsetospeed и cfsetispeed выдают ошибки только для значений быстродействия, которые система просто не может обрабатывать. Если Вы определяете значение быстродействия, которое является в основном допустимым, то эти функции выполняются успешно. Но они не проверяют, что специфическое аппаратное устройство может фактически поддерживать заданное быстродействие, они не знают, для которого устройства Вы планируете устанавливать быстродействие. Если Вы используете tcsetattr, чтобы установить быстродействие специфического устройства в значение, которое оно не может обрабатывать, tcsetattr, возвращает -1.

Примечание Переносимости: В библиотеке GNU, функции выше принимают скорости, измеряемые в битах в секунду как ввод, и возвращает значения быстродействия, измеряемые в битах в секунду. Другие библиотеки требуют, чтобы скорости были обозначены специальными кодами. Для POSIX. 1 переносимости, Вы должны использовать один из следующих символов, чтобы представить быстродействие; их точные числовые значения зависят от системы, но каждое имя, имеет фиксированное значение: B110 замещает 110 бит\сек, B300 для 300 бит\сек, и так далее. Нет никакого переносимого способа представить любое быстродействие, но это единственые скорости, которые последовательные строки могут поддерживать.

```
B0  B50  B75  B110  B134  B150  B200
B300  B600  B1200  B1800  B2400  B4800
B9600  B19200  B38400
```

BSD определяет два дополнительных символа быстродействия как побочные результаты исследования: EXTA - побочный результат исследования для B19200, и EXTB - побочный результат исследования для B38400. Эти побочные результаты исследования устаревшие.

int cfmakeraw (struct termios *termios_p) (функция)

Эта функция обеспечивает простой способ установить * termios_p для того, что традиционно вызвалось "необрабатываемый режим" в BSD. Она делает следующее:

```
termios_p->c_iflag &= ~(IGNBRK|BRKINT|PARMRK|ISTRIP
                  |INLCR|IGNCR|ICRNL|IXON);
termios_p->c_oflag &= ~OPOST;
termios_p->c_lflag &= ~(ECHO|ECHONL|ICANON|ISIG|IEXTEN);
termios_p->c_cflag &= ~(CSIZE|PARENB);
termios_p->c_cflag |= CS8;
```

12.4.9 Специальные Символы

В каноническом вводе, драйвер терминала распознает ряд специальных символов, которые выполняют различные функции управления. Они включают символ ERASE (обычно DEL) для редактирования ввода, и другие символы редактирования. Символ INTR (обычно C-c) для посылки сигнала SIGINT, и другие сигнальные символы, могут быть доступны либо в каноническом либо в неканоническом входном режиме. Все эти символы описаны в этом разделе.

Специфические используемые символы определены в c_cc элементе struct termios структуры. Этот элемент массив; каждый элемент определяет символ для специфической роли. Каждый элемент имеет символическую константу, которая замещает индекс этого элемента например, INTR - индекс элемента, который определяет символ INTR, так что сохранение "=" в termios.c_cc [INTR] определяет "=" как символ INTR.

На некоторых системах Вы можете отключать специфическую специальную символьную функцию, определяя значение _POSIX_VDISABLE. Это значение неравно любому возможному символьному коду. См. Раздел 27.7 [Опции Файлов] для получения более подробной информации, о том как сообщить операционной системе, которую Вы используете, поддержку _POSIX_VDISABLE.

12.4.9.1 Символы для Входного Редактирования

Эти специальные символы активны только в каноническом входном режиме. См. Раздел 12.3 [Канонический или Нет].

int VEOF (макрос)

Это индекс для символа EOF в специальном массиве управляющих символов. Termios.c_cc [VEOF] содержит символ непосредственно.

Символ EOF распознаваем только в каноническом входном режиме. Он действует как конец строки таким же образом как символ перевода строки, но если символ EOF печатается в начале строки, это заставляет read возвращать нулевой счетчик байтов, указывая конец файла. Символ EOF непосредственно отбрасывается.

Обычно, символ EOF - C-d.

int VEOL (макрос)

Это индекс для символа EOL в специальном массиве управляющих символов. Termios.c_cc [VEOL] содержит символ непосредственно.

Символ EOL распознаваем только в каноническом входном режиме. Он действует как признак конца строки, точно так же как символ перевода строки. Символ EOL не отбрасывается; он читается как последний символ во входной строке.

Вы не должны использовать символ EOL, чтобы заставить RET закончить строку. Просто установите ICRNL флаг. Фактически, это заданное по умолчанию состояние.

int VERASE

Это индекс для символа ERASE в специальном массиве управляющих символов. Termios.c_cc [VERASE] содержит символ непосредственно.

Символ ERASE распознаваем только в каноническом входном режиме. Когда пользователь печатает символ удаления, предыдущий печатаемый

символ отбрасывается. (Если терминал генерирует многобайтовые символьные последовательности, может быть отброшен больше чем один байт ввода.) Символ ERASE непосредственно отбрасывается.

Обычно, символ ERASE - DEL.

int VKILL (макрос)

Это индекс для символа KILL в специальном массиве управляющих символов. Termios.c_cc [VKILL] содержит символ непосредственно.

Символ KILL распознаваем только в каноническом входном режиме. Когда пользователь печатает этот символ, все содержимое текущей строки ввода отбрасывается. Сам символ непосредственно также отбрасывается.

Символ KILL - обычно C-u.

12.4.9.2 BSD Расширения Редактирующих Символов

Эти специальные символы активны только в каноническом входном режиме. См. Раздел 12.3 [Канонический или Нет]. Они являются расширениями BSD; библиотека GNU определяет символы на любой системе, но работают они только в системе BSD.

int VEOL2 (макрос)

Это индекс для символа EOL2 в специальном массиве управляющих символов. Termios.c_cc [VEOL2] содержит символ непосредственно.

Символ EOL2 работает точно так же как символ EOL (см. выше), но это может быть другой символ. Таким образом, Вы можете определять два символа, чтобы завершить входную строку, устанавливая EOL как один из них и EOL2 как другой.

int VWERASE (макрос)

Это индекс для символа WERASE в специальном массиве управляющих символов. Termios.c_cc [VWERASE] содержит символ непосредственно.

Символ WERASE распознаваем только в каноническом входном режиме. Он стирает слово предшествующего ввода.

int VREPRINT (макрос)

Это индекс для символа REPRINT в специальном массиве управляющих символов. Termios.c_cc [VREPRINT] содержит символ непосредственно.

Символ REPRINT распознаваем только в каноническом входном режиме. Он перепечатывает текущую входную строку.

int VLNEXT (макрос)

Это индекс для символа LNEXT в специальном массиве управляющих символов. Termios.c_cc [VLNEXT] содержит символ непосредственно.

Символ LNEXT распознаваем только, когда IEXTEN установлен. Он

отключает редактирующее значение следующего печатающегося символа. Это аналог команды C-q в Emacs.

Символ LNEXT - обычно C-v.

12.4.9.3 Символы вызывающие Сигналы

Эти специальные символы могут быть активны или в каноническом или в неканоническом входном режиме, но только, когда флаг ISIG установлен (см. Раздел 12.4.7 [Автономные режимы]).

int VINTR (макрос)

Это индекс для символа INTR в специальном массиве управляющих символов. Termios.c_cc [VINTR] содержит символ непосредственно.

INTR символ (прерывания) вызывает сигнал SIGINT для всех процессов в приоритетной работе, связанной с терминалом. Символ INTR непосредственно отбрасывается. См. Главу 21 [Обработка Сигнала] , для получения более подробной информации.

Обычно, символ INTR - C-c.

int VQUIT (макрос)

Это индекс для символа QUIT в специальном массиве управляющих символов. Termios.c_cc [VQUIT] содержит символ непосредственно.

Символ QUIT вызывает сигнал SIGQUIT для всех процессов в приоритетной работе, связанной с терминалом. Символ QUIT непосредственно отбрасывается. См. Главу 21 [Обработка Сигнала], дл получения более подробной информации.

Обычно, символ QUIT - C-\.

int VSUSP (макрос)

Это - нижний индекс для символа SUSP в специальном массиве управляющих символов.

Termios.c_cc [VSUSP] содержит символ непосредственно.

SUSP (приостанавливающий) символ, распознается только, если реализация поддерживает управление заданиями (см. Главу 24 [Управление заданиями]). Он посылает сигнал SIGTSTP всем процессам в приоритетной работе, связанной с терминалом. Символ SUSP непосредственно отбрасывается. См. Главу 21 [Обработка Сигнала], для получения более подробной информации.

Обычно, символ SUSP - C-z.

Некоторые приложения отключают нормальную интерпретацию символа SUSP. Если ваша программа делает это, она должна обеспечить какой-нибудь другой механизм для пользователя, чтобы остановить работу. Когда пользователь вызывает этот механизм, программа должна послать сигнал SIGTSTP группе процесса, а не только процессу непосредственно. См. Раздел 21.6.2 [Передача сигналов для Другого Процесса].

int VDSUSP (макрос)

Это индекс для символа DSUSP в специальном массиве управляющих символов. Termios.c_cc [VDSUSP] содержит символ непосредственно.

DSUSP (приостанавливающий) символ распознается только, если реализация поддерживает управление заданиями (см. Главу 24 [Управление заданиями]). Он посылает сигнал SIGTSTP, подобно символу SUSP, но не сразу, а когда программа пробует читать его как ввод.

Не все системы с управлением заданиями поддерживают DSUSP; только системы BSD. См. Главу 21 [Обработка Сигнала], для получения более подробной информации.

Обычно, символ DSUSP - C-y.

12.4.9.4 Специальные Символы для Управления потоком данных

Эти специальные символы могут быть активны или в каноническом или в неканоническом входном режиме, но их использование управляется флагами IXON и IXOFF (см. Раздел 12.4.4 [Входные Режимы]).

int VSTART

Это индекс для символа START в специальном массиве управляющих символов. Termios.c_cc [VSTART] содержит символ непосредственно.

Символ START используется, чтобы поддерживать режимы ввода IXON и IXOFF. Если IXON установлен, то получение символа START продолжает приостановленный вывод; символ START непосредственно отбрасывается. Если IXOFF установлен, то система может также передавать символы START на терминал.

Обычное значение для символа START - C-q. Вы не можете изменить это значение, т. к. аппаратные средства могут настаивать на использовании C-q независимо от того, что Вы определяете.

int VSTOP (макрос)

Это индекс для символа STOP в специальном массиве управляющих символов. Termios.c_cc [VSTOP] содержит символ непосредственно.

Символ STOP используется, чтобы поддерживать IXON и IXOFF режимы ввода. Если IXON установлен, то получение символа STOP заставляет риостановить вывод; символ STOP непосредственно отбрасывается. Если IXOFF установлен, система может также передавать символы STOP на терминал, предохранняя входную очередь от переполнения.

Обычное значение для символа STOP - C-s.

12.4.9.5 Другие Специальные Символы

Имеются два дополнительных специальных символа, которые являются значимыми на системах BSD.

int VDISCARD (макрос)

Это индекс для символа DISCARD в специальном массиве управляющих символов. Termios.c_cc [VDISCARD] содержит символ непосредственно.

Символ DISCARD распознаваем только, когда установлен IEXTEN . Он должен переключить флаг отбрасывания вывода. Когда этот флаг установлен, весь вывод программы отбрасывается. Установка флага также отбрасывает весь вывод находящийся в настоящее время в буфере вывода.

int VSTATUS (макрос)

Это индекс для символа STATUS в специальном массиве управляющих символов. Termios.c_cc [VSTATUS] содержит символ непосредственно.

Символ STATUS должен распечатать сообщение состояния относительно того, как текущий процесс выполняется.

Символ STATUS распознаваем только в каноническом режиме. Это специфическое решение задачи, так как значение символа STATUS ничего не делает со вводом.

12.4.10 Неканонический Ввод

В неканоническом входном режиме, специальные символы редактирования типа ERASE и KILL игнорируются.

Средства системы для редактирования ввода в неканоническом режиме заблокированы, так, чтобы все входные символы (если они не

специальные сигналы и не управляющие потоком данных) были переданы прикладной программе точно как печатались.

Неканонический режим предлагает специальные параметры называемые MIN и TIME для управления тем как долго ждать доступного ввода. Вы можете даже использовать их, чтобы избежать ожидания, чтобы возвратиться немедленно с любым вводом, или без ввода.

MIN и TIME сохранены в элементах c_cc массива, который является элементом struct termiosstructure. Каждый элемент этого массива имеет специфическую роль, и каждый элемент имеет символическую константу, которая замещает индекс этого элемента. VMIN и VMAX - имена для индексов TIME и MIN.

int VMIN

Это индекс для MIN в c_cc массиве. Таким образом, в termios.c_cc [VMIN] значение MIN хранится непосредственно.

MIN значим только в неканоническом входном режиме; он определяет минимальное число байтов, которые должны быть доступны во входной очереди для возвращения read.

int VTIME (макрос)

Это индекс для TIME в массиве c_cc. Таким образом, в termios.c_cc [VTIME] значение TIME представлено непосредственно.

TIME значим только в неканоническом входном режиме; он определяет, как долго ждать ввод перед возвращением, в единицах по 0.1 секунды.

Значения MIN и TIME взаимодействуют, чтобы определить критерий для возвращения read; их точные значения зависят, от того какой из них является отличным от нуля. Имеются четыре возможных случая:

О И MIN и TIME - нули.

В этом случае, read всегда возвращается немедленно со столькими символами, сколько доступно в очереди, до запрошенного числа. Если никакой ввод не является немедленно доступным, read возвращает

нуль.

О MIN - нуль, но TIME имеет значение отличное от нуля.

В этом случае, read ждет в течение времени TIME доступного ввода; доступность одиночного байта должна быть достаточна, чтобы удовлетворить запрос read. По возвращению функция принимает значение обозначающее количество доступных символов вплоть до запрошенного числа. Если никакой ввод не является доступным в заданное время, read возвращает нуль.

О TIME - нуль, но MIN имеет значение отличное от нуля.

В этом случае, read ждет, пока по крайней мере MIN байтов станут доступны в очереди. И возвращает количество доступных символов. Read может возвращать больше, чем MIN символов.

О И TIME и MIN отличны от нуля.

В этом случае, TIME определяет, как долго ждать следующего ввода после каждого входного символа. Read ждет, пока или MIN байтов прочитаны, или TIME истекает без дальнейшего ввода.

Read может не возвращать никакого ввода, если TIME истекает прежде, чем появится первый входной символ. Read может возвращать больше, чем MIN символов, если в очереди их больше чем MIN.

Что случается, если MIN - 50, и Вы запрашиваете прочитать только 10 байтов? Обычно, read ждет до 50 байтов в буфере (т. е. пока условие ожидания не удовлетварено), и тогда читает 10 из них, оставляя другие 40 буферизированными в операционной системе для последующего обращения read.

Примечание Переносимости: На некоторых системах, места MIN и TIME в массиве фактически такие же как у EOF и EOL. Это не вызывает

никакой серьезной проблемы, потому что MIN и TIME используется только в неканоническом вводе, а EOF и EOL используются только в каноническом вводе, но это не совсем хорошо. Библиотека GNU резервирует отдельные места для них.

12.5 Функции управления Строкой

Эти функции выполняют разнообразные управляющие действия на терминальных устройствах. Что касается доступа к терминалу, они обрабатываются подобно выполнению вывода: если любая из этих функций используется фоновым процессом на управляемом терминале, то обычно всем процессам в группе процесса посланы сигналы SIGTTOU. Исключение, если вызывающий процесс непосредственно игнорирует или блокирует сигналы SIGTTOU, тогда операция выполняется, и никакой сигнал не посылается. См. Главу 24 [Управление заданиями].

int tcsendbreak (int filedes, int duration)

Эта функция генерирует условие прерывания, передавая поток нулевых битов на терминал, связанный с описателем файла filedes. Продолжительность прерывания управляется аргументом duration. Если нуль, то продолжительность - между 0.25 и 0.5 секундами. Значение значения отличного от нуля зависит от операционной системы.

Эта функция ничего не делает, если терминал не асинхронный последовательный порт данных.

Возвращаемое значение обычно нуль. В случае ошибки, возвращается значение -1. Следующие условия ошибки errno определены для этой функции:

EBADF filedes - не допустимый описатель файла.
ENOTTY filedes не связан с устройством терминала.

int tcdrain (int filedes) (функция)

Tcdrain функция ждет, пока весь поставленный в очередь вывод на

терминал filedes не будет передан.

Возвращаемое значение обычно нуль. В случае ошибки, возвращается значение -1. Следующие условия ошибки errno определены для этой функции:

EBADF filedes - не допустимый описатель файла.
ENOTTY filedes не связан с устройством терминала.
EINTR операция была прервана сигналом. См. Раздел 21.5 [Прерванные Примитивы].

int tcflush (int filedes, int queue) (функция)

Эта функция используется, чтобы очистить очереди ввода и/или вывода, связанные с файлом терминала filedes. Аргумент queue определяет, какую очередь очищать, и может принимать одно из следующих значений:

TCIFLUSH очистить любые полученные, но еще не прочитанные, входные данные.

TCOFLUSH очистите любые выходные данные, которые написаны, но еще не переданный.

TCIOFLUSH Очистить, поставленный в очередь и ввод и вывод.

Возвращаемое значение обычно нуль. В случае ошибки, возвращаеися значение -1. Следующие условия ошибки errno определены для этой функции:

EBADF filedes - не допустимый описатель файла.
ENOTTY filedes не связан с устройством терминала.
EINVAL плохое значение было обеспечено как аргумент queue.

tcflush не совсем удачное название для это функции, так как термин "поток" обычно используется для совершенно другой операции. К сожалению, имя tcflush исходит из POSIX, и мы не можем изменять его.

int tcflow (int filedes, int action) (функция)

Эта функция используется, чтобы выполнить операции в отношении XON/XOFF управления потоком данных на файле терминала, заданном filedes.

Аргумент action определяет то, какую операцию выполнять, и может быть одним из следующих значений:

TCOOFF Приостанавливают передачу вывода.
TCOON Рестарт передачи вывода.
TCIOFF Передача символа STOP.
TCION Передача символа START.

Для получения более подробной информации о символах STOP и START, см. Раздел 12.4.9 [Специальные Символы].

Возвращаемое значение обычно нуль. В случае ошибки, возвращается значение -1. Следующие условия ошибки errno определены для этой функции:

EBADF filedes - не допустимый описатель файла.
ENOTTY filedes не связан с устройством терминала.
EINVAL плохое значение был обеспечен как аргумент action.

12.6 Пример Неканонического Режима

Вот пример программ, который показывает как Вы можете устанавливать устройство терминала для чтения одиночных символов в неканоническом входном режиме, без ECHO.

```
#include
#include
#include
#include
struct termios saved_attributes;
void
```

- 333 -

```
reset_input_mode (void)
{
        tcsetattr (STDIN_FILENO, TCSANOW,
&saved_attributes);
}
void
set_input_mode (void)
{
        struct termios tattr;
        char *name;
        if (!isatty (STDIN_FILENO))
                {
                        fprintf (stderr,
"Not a terminal.\n");
                        exit (EXIT_FAILURE);
                }
        tcgetattr (STDIN_FILENO, &saved_attributes);
        atexit (reset_input_mode);
        tcgetattr (STDIN_FILENO, &tattr);
        tattr.c_lflag &= ~(ICANON|ECHO);
        tattr.c_cc[VMIN] = 1;
        tattr.c_cc[VTIME] = 0;
        tcsetattr (STDIN_FILENO, TCSAFLUSH, &tattr);
}
int
main (void)
{
        char c;
        set_input_mode ();
        while (1)
                {
                        read (STDIN_FILENO, &c, 1);
                        if (c == '\004')        /* C-d */
                                break;
                        else
                                putchar (c);
                }
        return EXIT_SUCCESS;
```

- 334 -

```
}
```

Эта программа осторожна, она восстанавливает первоначальные режимы терминала перед выходом или завершением с сигналом. Она использует функцию atexit (см. Раздел 22.3.3 [Очистка на Выходе]) чтобы удостовериться в установке режимов.

Оболочка, как предполагается, заботится о сбросе режимов терминала, когда процесс остановлен или продолжается; см. Главу 24 [Управление заданиями]. Но некоторые существующие оболочки фактически не делают это, так что Вы можете установить обработчики для сигналов управления заданиями, которые сбрасывают режимы терминала. Вышеупомянутый пример делает так.

13. Математика

Эта глава содержит информацию относительно функций для выполнения математических вычислений, типа тригонометрических функций. Большинство этих функций имеет прототипы, объявленные в файле "math.h".

Все функции используют аргументы с плавающей запятой и возвращают результаты типа double. В будущем, могут появиться

дополнительные функции, которые используют значения long double и float. Например, cosf и cosl были бы версиями функци cos, которые используют аргументы типов float и long double, соответственно. Вы должны избегать использовать эти имена самостоятельно. См. Раздел 1.3.3 [Зарезервированные Имена].

13.1 Ошибки Области и Диапазона

Многие из функций, перечисленных в этой главе определены математически над областью, которая является только подмножеством вещественных чисел. Например, acos функция определена над областью от -1 до 1. Если Вы передаете аргумент одной из этих функций, который не находится в области, на которой она определена, функция устанавливает errno как EDOM, чтобы указать ошибку области. На

машинах, которые поддерживают формат ИИЭР (IEEE) с плавающей запятой, также возвращают ошибку EDOM и NaN.

Некоторые из этих функций определены математически, чтобы привести к комплексному значению над частями их областей. Наиболее знакомый пример это квадратный корень отрицательного числа.

Функции в этой главе берут аргументы только с плавающей точкой и возвращают значения соответственно того же типа.

Проблема возникает тогда, когда математический результат функции не может быть представим как число с плавающей запятой. Если величина результата слишком большая, функция устанавливает errno как ERANGE, чтобы указать ошибку диапазона, и возвращает "очень большое значение" (именованое макркомандой HUGE_VAL) или отрицание (- HUGE_VAL).

Если величина результата является слишком малой, возвращается нулевое значение. В этом случае, errno может быть или не быть установлена как ERANGE.

Единственый полностью надежный способ проверять ошибки области и диапазона состоит в том, чтобы установить errno как 0 прежде чем Вы вызываете математическую функцию и затем проверять errno. Следствие такого использования errno является то, что математические функции не могут быть многократно использованы с проверкой ошибок.

Ни одна из математических функций не генерирует сигналы в результате ошибок диапазона или области. В частности это означает что Вы не будете видеть сигналы SIGFPE, сгенерированные внутри этих функций. (См. Главу 21 [Обработка Сигнала], для получения более подробной информации.)

double HUGE_VAL (макрос)

Это выражение представляющее наибольшее число. На машинах, которые используют, ИИЭР формат с плавающей запятой это значение "бесконечность". На других машинах, это обычно самое большое положительное число, которое может быть представлено.

Значение этой макрокоманды используется как возвращаемое значение из различных математических функций в случаях переполнения.

Для получения более подробной информации см. Раздел A. 5.3.2 [Параметры с плавающей запятой]. В частности макркоманда DBL_MAX могла бы быть более удобной, чем HUGE_VAL для многих использований отличных от тестирования ошибки в математической функции.

13.2 Тригонометрические Функции

Это знакомые функции sin, cos, и tan. Аргументы всех этих функций измеряются в радианах; помните, что pi радиан равняются 180 градусам.

Математическая библиотека не определяет символическую константу для pi, но Вы можете определять вашу собственную, если Вы

нуждаетесь в этом:

```
#define PI 3.14159265358979323846264338327
```

Вы можете также вычислять значение pi выражением acos (-1.0).

```
double sin (double x)
```

Эта функция возвращает синус x, где x дан в радианах. Возвращаемое значение находится в диапазоне от -1 до 1.

```
double cos (double x)
```

Эта функция возвращает косинус x, где x дан в радианах. Возвращаемое значение находится в диапазоне от -1 до 1.

```
double tan (double x)
```

Эта функция возвращает тангенс x, где x дан в радианах.

Следующие errno условия ошибки определены для этой функции:

ERANGE Математически функция tan имеет особенности в точках (2*k+1)*pi/2. Если аргумент x близок к одной из этих особенностей, tan устанавливает errno как ERANGE и возвращает или положительный или отрицательный HUGE_VAL.

13.3 Обратные Тригонометрические Функции

Это обычные arcsin, arccos и arctan функции, которые являются обратными для синуса, косинуса и тангенса, соответственно.

```
double asin (double x)
```

Эта функция вычисляет арксинус x то есть значение, чей синус является x. Значение результата дается в радианах. Математически, имеется бесконечно много таких значений; но фактически возвращаются значения между -pi/2 и pi/2 (включая).

Asin устанавливает errno как EDOM, если x находится вне диапазона. Функция арксинуса определена математически только над областью от -1 до 1.

```
double acos (double x)
```

Эта функция вычисляет аркосинус x, то есть значение, чей косинус является x. Значение результата дается в радианах. Математически, имеется бесконечно много таких значений; но фактически возвращаются значения между 0 и pi (включая).

Acos устанавливает errno как EDOM, если x находится вне диапазона. Функция аркосинуса определена математически только над областью от -1 до 1.

```
double atan (double x)
```

Эта функция вычисляет арктангенс x то есть значение, чей тангенс

является x.

Значение результата дается в радианах. Математически, имеется бесконечно много таких значений; но фактически возвращаются значения между -pi/2 и pi/2 (включая).

```
double atan2 (double y, double x)
```

Это функция арктангенса двух аргументов. Она подобна вычислению арктангенса y/x, за исключением того, что знаки обоих

аргументов используются, чтобы определить квадрант результата, и x может быть нулем. Возвращаемое значение дано в радианах и находится в диапазоне от -pi до pi (включительно).

Если x и y координаты точки в плоскости, atan2 возвращает угол между линией от начала координат до этой точки и осью X. Таким образом, atan2 полезен для преобразования декартовых координат в полярные координаты. (Чтобы вычислять радиальную координату, используйте hypot; см. Раздел 13.4 [Экспоненты и Логарифмы])

Функция atan2 устанавливает errno как EDOM, если и x и y нули и возвращаемое значение не определено.

13.4 Возведение в степень и Логарифмы

double exp (double x)

Эта функция возвращает значение e (основание натуральных логарифмов) в степени x.

Если величина результата слишком большая, чтобы быть представимой функция устанавливает errno как ERANGE.

double log (double x)

Эта функция возвращает натуральный логарифм x. exp (log(x)) равняется x, точно в математике и приблизительно в Си.

Следующие условия ошибки errno определены для этой функции:

Функиця устанавливает EDOM, если аргумент x отрицателен. Функция log определена математически, чтобы возвращать результат только на положительных аргументах.

Функиця устанавливает ERANGE, если аргумент - нуль. Log нуля не определен.

double log10 (double x)

Эта функция возвращает логарифм x по основанию 10. Кроме основания , она подобна функции log. Фактически, log10 (x) равняется log (x) / log (10).

double pow (double base, double power)

Это общая функция возведения в степень, возвращающая base в степени power.

Следующие условия ошибки errnoопределены для этой функции:

Функиця устанавливает EDOM base, если аргумент отрицателен.

Функиця устанавливает ERANGE, если было обнаружено условие переполнения.

double sqrt (double x)

Эта функция возвращает квадратный корень x.

Sqrt устанавливает errno как EDOM, если x отрицателен. Математически, квадратный корень был бы комплексным числом.

double cbrt (double x)

Эта функция возвращает кубический корень x. Эта функция определена на всей вещественной оси и поэтому не изменяет переменную errno.

double hypot (double x, double y)

Эта функция возвращает sqrt (x * x + y * y). (Это длина гипотенузы прямоугольного треугольника со сторонами длины x и y, или расстояние точки (x, y) от начала.) См. также функцию cabs в Разделе 14.3 [Абсолютное значение].

double expm1 (double x)

Эта функция возвращает эквивалент значения exp (x) - 1. Оно вычислено точным способом, даже если значение x близко к нулю.

double log1p (double x)

Эта функция возвращает эквивалент значения log (1 + x). Оно вычислено точным способом, даже если значение x - близко к нулю.

13.5 Гиперболические функции

Функции в этом разделе связаны с экспоненциальными функциями; см. Раздел 13.4 [Экспоненты и Логарифмы].

double sinh (double x) (функция)

Эта функция возвращает гиперболический синус x, определенный математически как exp (x) - exp (-x) /.2 Функция устанавливает errno как ERANGE, если значение x слишком большое; то есть если происходит переполнение.

double cosh (double x) (функция)

Функция cosh возвращает гиперболический косинус x, определенный математически как exp (x) + exp (-x) /.2 Функция устанавливает errno как ERANGE, если значение x слишком большое; то есть если происходит переполнение.

double tanh (double x) (функция)

Эта функция возвращает гиперболический тангенс x, чье математическое определение sinh (x) / cosh (x).

double asinh (double x) (функция)

Эта функция возвращает обратный гиперболический синус x, чей гиперболический синус является x.

double acosh (double x) (функция)

Эта функция возвращает обратный гиперболический косинус x, чей гиперболический косинус является x. Если x - меньше чем 1, acosh возвращает HUGE_VAL.

double atanh (double x) (функция)

Эта функция возвращает обратный гиперболический тангенс x, чей гиперболический тангенс является x. Если абсолютное значение x больше или равно 1, atanh возвращает HUGE_VAL.

13.6 Псевдослучайные Числа

Этот раздел описывает средства GNU для получения ряда псевдослучайных чисел. Сгенерированные числа не совсем произвольные; обычно, они формируют последовательность, которая повторяется периодически, с периодом настолько большим, что Вы можете игнорировать его для обычных целей. Генератор случайных чисел работает, используя всегда значение начального числа, чтобы вычислить следующее случайное число и так далее.

Хотя сгенерированные чисала выглядят непредсказуемо при одном выполнении программы, последовательность чисел точно та же самая и при следующем выполении. Это происходит, потому что начальное число всегда одно то же . Это удобно, когда Вы отлаживаете программу, но

неподходит, если Вы хотите, чтобы программа вела себя

непредсказуемо. Если Вы хотите действительно случайные числа, а не только псевдослучайными, определяйте начальное число, основываясь на текущем времени.

Вы можете получать повторяющиеся последовательности чисел на машине определенного типа, определяя то же самое начальное значение начального числа для генератора случайных чисел. Не существует никакого стандарта для значения начального числа; то же самое начальное число, используемое в различных библиотеках C или на различных типах CPU , даст Вам различные случайные числа.

Библиотека GNU поддерживает стандартные функции случайного числа ANSI C плюс набор, который происходит от BSD. Мы рекомендуем, чтобы Вы использовали стандартные rand и srand.

13.6.1 Функции Случайного числа ANSI C

Этот раздел описывает функции случайного числа, которые являются частью стандарта ANSI C.

Чтобы использовать эти средства, Вы должны включить заглавный файл "stdlib.h" в вашей программе.

int RAND_MAX (макрос)

Значение этой макрокоманды - выражение константы integer, которое представляет максимальное возможное значение, возвращенное функцией rand. В библиотеке GNU, это - 037777777, которое является самым большим целым числом со знаком, представимым в 32 битах. В других библиотеках это может быть всего 32767.

int rand () (функция)

Функция rand возвращает следующее псевдослучайное число. Значение находится в диапазоне от 0 до RAND_MAX.

void srand (unsigned int seed) (функция)

Эта функция устанавливает seed как начальное число для нового ряда псевдослучайных чисел.

Если Вы вызываете rand перед тем как начальное число было установлен srand, то она использует значение 1 как заданное по умолчанию начальное число.

Чтобы производить случайные числа (не только псевдослучайные), делайте srand(time(0)).

13.6.2 BSD Функции Случайного числа

Этот раздел описывает набор функций для порождения случайного числа, являющиеся дополнением к BSD.

Нет особого преимущества при использовании этих функций с библиотекой GNU C; мы поддерживаем их только для совместимости BSD.

Прототипы для этих функций находятся в " stdlib.h ".

long int random () (функция)

Эта функция возвращает следующее псевдослучайное число. Диапазон возвращенных значений - от 0 до RAND_MAX.

void srandom (unsigned int seed) (функция)

Srandom функция устанавливает начальное число для random. Если Вы обеспечиваете значение начального числа 1, это заставит random воспроизводить набор случайных значений по умолчанию.

Чтобы производить действительно случайные числа (не только псевдослучайные), делайте srandom(time(0)).

void * initstate (unsigned int seed, void *state, size_t size) (функция)

Эта функция используется, чтобы инициализировать состояние генератора случайного числа. Аргумент state это массив size байтов, используемый для хранения информации о состоянии. Размер должен быть по крайней мере 8 байтов, а оптимальные размеры - 8, 16, 32, 64, 128, и 256. Больший массив состояния, лучше.

Возвращаемое значение - предыдущее значение массива информации о состоянии. Вы можете использовать это значение позже как аргумент setstate, чтобы восстановить состояние.

void *setstate (void *state)

Эта функция восстанавливает информацию о состоянии случайного числа. Аргумент должен быть результатом предыдущего обращения к initstate или setstate.

Возвращаемое значение - предыдущее значение массива информации о состоянии. Вы можете использовать это значение позже как аргумент setstate, чтобы восстановить состояние.

14. Арифметические функции низкого уровня

Эта глава содержит информацию относительно функций предназначенных для выполнения базисных арифметических операций, типа разбивания float на целую и дробную части. Эти функции объявлены в заголовочном файле "math.h ".

14.1 "Не Числовые" Значения

Формат ИИЭР с плавающей запятой используемый наиболее современными компьютерами, поддерживает значения, которые являются "не числами". Эти значения называются NaN. Эти значения следуют из некоторых операций, которые не имеют никакого значимого числового результата, типа нуля, деленого на нуль или бесконечности, деления

на бесконечность.

Одна примечательная особенность NaN'ов заключается в том, что они не равны себе. Таким образом, x == x может быть 0, если значение x - NaN. Вы можете использовать это, чтобы проверить является ли значение NaN или нет: если оно не равно себе, то это - NaN. Но рекомендуемый способ проверять NaN - isnan функцией (см. Раздел 14.2 [Предикаты на Значениях с Плаваюшей точкой]).

Почти любая арифметическая операция, в которой аргумент является NaN, возвращает NaN.

double NAN (макрос)

Выражение, представляющее значение, которое является "не числом". Эта макрокоманда является расширением GNU, доступное только на машинах, которые поддерживают значения "not a number" то есть на всех машинах поддерживающих формат с плавающей запятой.

Вы можете использовать " #ifdef NAN " чтобы проверить, поддерживает ли машина NaN. (Конечно, Вы должны принять меры, чтобы расширения GNU были видимыми, определяя _GNU_SOURCE, и Вы должны включить " math.h ".)

14.2 Предикаты на Float

Этот раздел описывает некоторые разнообразные функции-тесты double значений. Прототипы для этих функций появляются в "math.h". Это функции BSD, и таким образом доступны, если Вы определяете _BSD_SOURCE или _GNU_SOURCE.

int isinf (double x) (функция)

Эта функция возвращает -1, если x представляет отрицательную бесконечность, 1, если x представляет положительную бесконечность, и 0 иначе.

int isnan (double x) (функция)

Эта функция возвращает значение отличное от нуля, если x - значение " not a number ", и нуль иначе. (Вы можете точно также использовать x!= x, чтобы получить тот же самый результат).

int finite (double x) (функция)

Эта функция возвращает значение отличное от нуля, если x конечен или значение " not a number ", и нуль иначе.

double infnan (int error) (функция)

Эта функция предусмотрена для совместимости с BSD. Другие математические функции используют infnan, чтобы решить, что возвратить в случае ошибки. Аргумент - код ошибки, EDOM или ERANGE; infnan возвращает подходящее значение, чтобы указать ошибку. - ERANGE также допустим как аргумент, и соответствует -HUGE_VAL как значение.

В библиотеке BSD, на некоторых машинах, infnan вызывает фатальный сигнал во всех случаях. Библиотека GNU не делает аналогично, потому что это не удовлетворяет спецификации ANSI C.

Примечание Переносимости: функции, перечисленные в этом разделе - расширения BSD.

14.3 Абсолютное значение

Эти функции предусмотрены для получения абсолютного значения (или величины) числа. Абсолютное значение вещественного числа x - x если x положителен, -x, если x отрицателен. Для комплексного числа z, чья вещественная часть является x и чья мнимая часть является y, абсолютное значение - sqrt (x * x + y * y).

Прототипы для abs и labs находятся в " stdlib.h "; fabs и cabs объявлены в " math.h ".

int abs (int number) (функция)

Эта функция возвращает абсолютное значение числа.

Большинство компьютеров использует двоичное дополнение представления integer, в котором абсолютное значение INT_MIN (самый маленький возможный int) не может представляться; таким образом, abs (INT_MIN) не определен.

long int labs (long int number) (функция)

Подобна abs, за исключением того, что и аргумент и результат имеют тип long int a не int.

double fabs (double number) (функция)

Эта функция возвращает абсолютное значение числа с плавающей запятой.

double cabs (struct { double real, imag; } z) (функция)

Функция cabs возвращает абсолютное значение комплексного числа z, чья вещественная часть является z.real и чья мнимая часть является z.imag. (См. также функцию hypot в Разделе 13.4 [Экспоненты и Логарифмы].); значение:

sqrt (z.real*z.real + z.imag*z.imag)

14.4 Функции Нормализации

Функциям, описанные в этом разделе прежде всего обеспечивают способ эффективно выполнить некоторые манипулирования низкого уровня на числах с плавающей запятой, которые представляются, внутренне используя двоичную систему счисления. Эти функции требуются, чтобы реализовать эквивалентное поведение, даже если представление не использует основание системы счисления 2, но конечно они, вряд ли, будут особенно эффективны в тех случаях.

Все эти функции объявлены в " math.h ".

double frexp (double value, int *exponent) (функция)

Frexp функция используется, чтобы разбивать значение числа на нормализованную дробь и экспоненту.

Если значение аргумента value - не нуль, возвращаемое значение value - число степеней двойки, и всегда в диапазоне от 1/2 (включ.) до 1 (исключ.). Соответствующая экспонента сохранена в *exponent; возвращаемое значение, умноженное на 2 возведенное в эту экспоненту, равняется первоначальному значению числа.

Например, frexp (12.8, &exponent) возвращает 0.8 и сохраняет 4 в exponent.

double ldexp (double value, int exponent) (функция)

Эта функция возвращает результат умножения значения числа с плавающей запятой на 2 в степени exponent. (Это может использоваться, чтобы повторно транслировать числа с плавающей запятой, которые были демонтированы frexp.)

Например, ldexp (0.8, 4) Возвращает 12.8.

Следующие функции, которые исходят ИЗ BSD, обеспечивают эквиваленты средств ldexp и frexp:

double scalb (double value, int exponent)

Scalb функция - BSD имя для ldexp.

double logb (double x)

Эта BSD функция возвращает целочисленную часть логарифма x по осн. 2, целочисленное значение представляется в double. Знак x игнорируется. Например, logb (3.5) = 1.0 и logb (4.0) = 2.0.

Когда 2 в этой степени разделена на x, это дает частное между 1 (включ.) и 2 (исключ.).

Если x - нуль, значение - минус бесконечность (если машина поддерживает такое значение), или очень малое число. Если x - бесконечность, значение - бесконечность.

Значение, возвращенное logb на один меньше чем то что frexp

сохранил бы в *exponent.

double copysign (double value, double sign) Function

Copysign функция возвращает значение, чье абсолютное значение равно указанному значению, и чей знака противоположен исходному. Это - функция BSD.

14.5 Функции Округления и Остаточного члена

Функции, перечисленные здесь выполняют операции типа округления, усечения, и взятия остаточного члена от деления чисел с плавающей запятой. Некоторые из этих функций преобразовывают числа с плавающей запятой в целочисленные значения. Они все объявлены в " math.h ".

Вы можете также преобразовывать числа с плавающей запятой в integer просто, приводя их к int. Это отбрасывает дробную часть, действительно округляя к нулю. Однако, это работает только, если результат может фактически представляться как int и для очень больших чисел, это невозможно. Функции, перечисленные здесь возвращают результат как double, чтобы обойти эту проблему.

double ceil (double x)

Ceil функция округляет x вверх к самому близкому целому числу, возвращая это значение как double. Таким образом, ceil (1.5) = 2.0.

double floor (double x)

floor функция округляет x вниз к самому близкому целому числу, возвращая это значение как double. Таким образом, floor (1.5) = 1.0, a floor (-1.5) = -2.0.

double rint (double x)

Эта функция округляет x к целочисленному значению согласно текущему режиму округления.

Режим округления значения по умолчанию к самому близкому целому числу; некоторые машины поддерживают другие режимы, но этот не всегда используется если Вы явно выбираете другой.

double modf (double value, double *integer_part)

Эта функция разбивает значение аргумента на целочисленную часть и дробную часть (между -1 и 1, не включ.). Их сумма равняется значению. Каждая из частей имеет тот же самый знак как значение, так что округление целочисленной части - к нулю.

Modf сохраняет целочисленную часть в *integer_part, и возвращает дробную часть. Например, modf (2.5, &intpart) возвращает 0.5 и сохраняет 2.0 в integer_part.

double fmod (double numerator, double denominator)

Эта функция вычисляет остаточный член от деления numerator yf denominator. Специально, возвращаемое значение - numerator - n * denominator, где n - частное numerator/denominator, округленное к целому числу. Таким образом, fmod (6.5, 2.3) возвращает 1.9, который является 6.5-4.6.

Результат имеет тот же самый знак как numerator и имеет величину меньше чем величина denominator.

Если denominator - нуль, fmod сбоит и устанавливает errno как

EDOM.

double drem (double numerator, double denominator) (функция)

Функция drem - подобна fmod за исключением того, что она округляет внутреннее частное n к самому близкому целому числу a не к целому числу в сторону нуля. Например, drem (6.5, 2.3) возвращает -0.4, который является 6.5-6.9.

Абсолютное значение результата - меньше или равно половине абсолютного значения denominator. Различие между fmod (numerator, denominator) и drem (numerator, denominator) - всегда либо denominator, либо -denominator, либо нуль.

Если denominator - нуль, drem сбоит и устанавливает errno как EDOM.

14.6 Целочисленное деление

Этот раздел описывает функции для выполнения деления целых чисул. Эти функции избыточны в библиотеке GNU C, с тех пор в GNU C, оператор ` / ' всегда округляется к нулю. Но в других реализациях C, " / " может поступать по-другому с отрицательными аргументами. div и ldiv полезны, потому что они определяют как округляется частное: к нулю. Остаточный член имеет тот же самый знак как числитель.

Эти функции определены, чтобы возвратить результат r такой, что значение r.quot*denominator+r.rem равняется numerator.

Чтобы использовать эти средства, Вы должны включить заглавный файл " stdlib.h " в вашей программе.

div_t (тип данных)

Это - структура, используемая, чтобы содержать результат,

возвращенный функцией div. Она имеет следующие элементы:

Int quot частное от деления.
Int rem остаточный член от деления.

div_t div (int numerator, int denominator) (функция)

Эта функция вычисляет частное и остаточный член от деления numerator на denominator, возвращая результат в структуре типа div_t.

Если результат не может представляться (напр. деление на нуль), поведение не определено.

Вот пример, хотя и не очень полезный.

```
div_t result;
result = div (20, -6);
```

Теперь result.quot = -3, a result.rem = 2.

ldiv_t (тип данных)

Это - структура, используемая, чтобы содержать результат, возвращенный функцией ldiv. Она имеет следующие элементы:

long int quot

Частное от деления.

long int rem

Остаточный член от деления. (идентично div _t за исключением того, что компоненты имеют тип long int a не int.)

ldiv_t ldiv (long int numerator, long int denominator)
(функция)

Функция ldiv подобна div, за исключением того, что аргументы имеют тип long int, и результат возвращается как структура ldiv типа.

14.7 Синтаксический анализ Чисел

Этот раздел описывает функции для "чтения" целого числа и чисел с плавающей запятой из строки. Может быть более удобно в некоторых случаях использовать sscanf или одну из подобных функций; см. раздел 7.11 [Форматируемый Ввод]. Но часто Вы можете делать программу более надежной, находя лексемы в строке вручную, и преобразуя их в числа один за другим.

14.7.1 Последовательный Синтаксический анализ

Эти функции объявлены в " stdlib.h ".

long int strtol (const char *string, char **tailptr, int base)
(функция)

Strtol ("string-to-long") функция преобразовывает начальную часть строки в целое число со знаком, которое возвращено как значение long int.

Если строка является пустой, содержит только пропуск(и), или не содержит начальную подстроку, которая имеет ожидаемый синтаксис для целого числа с заданным base, никакое преобразование не выполняется. В этом случае, strtol возвращает нулевое значение, а значение, сохраненное в *tailptr - значение строки.

Если строка имеет допустимый синтаксис для целого числа, но значения не представимы из-за переполнения, strtol возвращает или LONG_MAX или LONG_MIN (см. Раздел A. 5.2 [Диапазон Типа]),

соответствующее знаку значения. Она также устанавливает errno как ERANGE, чтобы указать, что имелось переполнение.

См. пример в конце этого раздела.

unsigned long int strtoul (const char *string, char **tailptr, int base)

Strtoul ("string-to-unsigned-long") функция - подобна strtol за исключением того, что она возвращает значение с типа long unsigned int. Значение, возвращенное в случае переполнения - ULONG_MAX (см. Раздел A. 5.2 [Диапазон Типа]).

long int atol (const char *string) (функция)

Эта функция подобна strtol функции с аргументом base 10, за исключением того, что ей не требуется обнаруживать ошибки переполнения. Atol функция обеспечивается обычно для совместимости с существующим кодом; использование strtol более надежно.

int atoi (const char *string) (функция)

Эта функция - подобна atol, за исключением того, что она возвращает значение int а не long int. Atoi функция также рассматривается устаревшей; используйте strtol.

Вот функция, которая анализирует строку как последовательность целых и возвращает их сумму:

```
int
sum_ints_from_string (char *string)
{
        int sum = 0;
        while (1) {
                char *tail;
                int next;
                while (isspace (*string)) string++;
                if (*string == 0)

               - 355 -

                        break;
                errno = 0;
                next = strtol (string, &tail, 0);
                if (errno)
                        printf ("Overflow\n");
                else
                        sum += next;
                string = tail;
        }
        return sum;
}
```

14.7.2 Синтаксический анализ Float

Эти функции объявлены в " stdlib.h ".

double strtod (const char *string, char **tailptr) (функция)

Strtod ("string-to-double") функция преобразовывает начальную часть строки в число с плавающей запятой, которое возвращается как значение double.

Эта функция пытается разлагать строку следующим образом:

* (Возможно пустая) последовательность символов пропуска. Символы пропуска определяются isspace функцией (см. Раздел 4.1 [Классификация Символов]). Они отброшены.

* Не обязательный "плюс" или "минус" (" + " или " - ").

* Непустая последовательность цифр, необязательно содержащих десятичную точку ".", но это зависит от стандарта (см. Раздел 19.6 [Числовое Форматирование].

* Не обязательная часть экспоненты, состоящая из символа " e " или " E ", знака, и последовательности цифр.

* Если tailptr - не пустой указатель, указатель на этот хвост

списка строки сохранен в *tailptr.

Если строка является пустой, содержит только пропуски, или не содержит начальную подстроку, которая имеет ожидаемый синтаксис для числа с плавающей запятой, никакое преобразование не выполняется.

В этом случае, strtod возвращает нуль, а значение, возвращенное в *tailptr - значение строки.

В стандарте отличном от стандарта "C", эта функция может распознавать дополнительный синтаксис.

Если строка имеет допустимый синтаксис для числа с плавающей запятой, но значения, не представимы из-за переполнения, strtod возвращает или положительный или отрицательный HUGE_VAL (см. Главу 13 [Математика]), в зависимости от знака значения. Аналогично, если значение не представимо из-за близости к нулю, strtod возвращает нуль. Она также устанавливает errno как ERANGE, если имелось переполнение или обнуление.

double atof (const char *string) (функция)

Эта функция подобна функции strtod, за исключением того, что ей не требуется обнаруживать ошибки переполнения. Atof обеспечивают обычно для совместимости с существующим кодом; использование strtod более надежно.

15. Поиск и Сортировка

Эта глава описывает функции для поиска и сортировки массивов произвольных объектов. Вы определяете соответствующую функцию сравнения, которую нужно применить как аргумент, наряду с размером объектов в массиве и общим числом элементов.

15.1 Определение Функции Сравнения

Чтобы использовать библиотечные функции сортировки массива, Вы должны описать, как сравнить элементы массива.

Чтобы сделать это, Вы обеспечиваете функцию сравнения, для сравнения двух элементов массива. Библиотека вызовет эту функцию, передавая как указатели на аргументы два элемента массива, которые нужно сравнить. Ваша функция сравнения должна возвратить значение как strcmp (см. Раздел 5.5 [Сравнение СТРОКИ/МАССИВА]): отрицательное, если первый аргумент - "меньше" чем второй, нуль, если они "равны", и положительное если первый аргумент "больше".

Вот пример функции сравнения, которая работает с массивом чисел типа double:

```
int
compare_doubles (const double *a, const double *b)
{
        return (int) (*a - *b);
}
```

Заглавный файл " stdlib.h " определяет имя для типа данных функций сравнения. Этот тип - расширение GNU.

```
int comparison_fn_t (const void *, const void *);
```


15.2 Функция Поиска в Массиве

Чтобы искать в сортируемом массиве элемент, соответствующий ключу, используйте bsearch функцию. Прототип для этой функции находится в заглавном файле " stdlib.h ".

```
void * bsearch (const void *key, const void *array, size_t
count, size_t size, comparison_fn_t compare)
```

Bsearch функция ищет в сортируемом массиве объект, который является эквивалентным key. Массив содержит count элементов, каждый из которых имеет байты размера sise.

Возвращаемое значение - указатель на соответствующий элемент

массива, или пустой указатель, если никакое соответствие не найдено. Если массив содержит больше чем один подходящий элемент, неопределено который же возвращается.

Эта функция получила имя из предположения, что она выполнена, используя двоичный алгоритм поиска.

15.3 Функция Сортировки Массива

Для сортировки массива, используюя произвольную функцию сравнения, используйте qsort функцию. Прототип для этой функции находится в " stdlib.h ".

void qsort (void *array, size_t count, size_t size, comparison_fn_t compare)

Qsort функция сортирует заланный массив. Массив содержит count элементов, каждый из которых имеет размер size.

Функция compare используется, чтобы выполнить сравнение на элементах массива. Эта функция вызывается с двумя аргументами указателями и должна возвратить целое число меньше , равное, или больше нуля, если первый аргумент меньше , равен, или больше чем

второй аргумент.

Предупреждение: если, два объекта сравниваются как равные, их порядок после сортировки, непредсказуем. То есть сортировка не устойчива. Она может делать различие, когда сравнение рассматривает только часть элементов. А также, два элемента с тем же самым ключом сортировки могут отличиться в других отношениях.

Вот простой пример сортировки массива double значений в числовом порядке используя функцию сравнения, определенную выше (см. Раздел 15.1 [Функции Сравнения]):

```
{
        double *array;
        int size;
        . . .
        qsort(array,size,sizeof(double),compare_doubles);
}
```

Qsort функция получила имя из предположения, что она была первоначально выполнена, используя алгоритм "быстрой сортировки".

15.4 Пример Поиска и Сортировки

Вот пример, показывающий использование qsort и bsearch с массивом структур. Объекты в массиве сортируются, сравнивнением их name полей функцией strcmp.

```
#include
#include
#include
struct critter
{
        const char *name;
        const char *species;
};
struct critter muppets[] =
{
```



```
        {"Kermit", "frog"},
        {"Piggy", "pig"},
        {"Gonzo", "whatever"},
        {"Fozzie", "bear"},
        {"Sam", "eagle"},
        {"Robin", "frog"},
        {"Animal", "animal"},
        {"Camilla", "chicken"},
```

```
                {"Sweetums", "monster"},
                {"Dr. Strangepork", "pig"},
                {"Link Hogthrob", "pig"},
                {"Zoot", "human"},
                {"Dr. Bunsen Honeydew", "human"},
                {"Beaker", "human"},
                {"Swedish Chef", "human"}
        };
        int count=sizeof(muppets)/sizeof(struct critter);
        int
        critter_cmp (const struct critter *c1,
        const struct critter *c2)
        {
                return strcmp (c1->name, c2->name);
        }
        void
        print_critter (const struct critter *c)
        {
                printf ("%s, the %s\n", c->name,
        c->species);
        }
        void
        find_critter (const char *name)
        {
                struct critter target, *result;
                target.name = name;
                result = bsearch (&target, muppets, count,
                sizeof (struct critter), critter_cmp);
                if (result)
                        print_critter (result);

                       - 361 -

                else
                        printf ("Couldn't find %s.\n", name);
        }
        int
        main (void)
        {
                int i;
                for (i = 0; i < count; i++)
                        print_critter (&muppets[i]);
                printf ("\n");
                qsort (muppets, count, sizeof (struct
                critter), critter_cmp);
                for (i = 0; i < count; i++)
                        print_critter (&muppets[i]);
                printf ("\n");
                find_critter ("Kermit");
                find_critter ("Gonzo");
                find_critter ("Janice");
                return 0;
        }
```

Вывод этой программы:

```
        Kermit, the frog
        Piggy, the pig
        Gonzo, the whatever
        Fozzie, the bear
        Sam, the eagle
        Robin, the frog
        Animal, the animal
        Camilla, the chicken
        Sweetums, the monster
        Dr. Strangepork, the pig
        Link Hogthrob, the pig
        Zoot, the human
        Dr. Bunsen Honeydew, the human
        Beaker, the human
        Swedish Chef, the human
        Animal, the animal
        Beaker, the human

                       - 362 -

        Camilla, the chicken
```

```
Dr. Bunsen Honeydew, the human
Dr. Strangepork, the pig
Fozzie, the bear
Gonzo, the whatever
Kermit, the frog
Link Hogthrob, the pig
Piggy, the pig
Robin, the frog
Sam, the eagle
Swedish Chef, the human
Sweetums, the monster
Zoot, the human
Kermit, the frog
Gonzo, the whatever
Couldn't find Janice.
```

16. Сопоставления с образцом

Библиотека GNU C обеспечивает средства сопоставления с образцом для двух видов шаблонов: регулярные выражения и универсальные символы имени файла.

16.1 Универсальное сопоставление

Этот раздел описывает, как шаблон универсальных символов соответствует специфической строке. Результат - ответ да или никакого ответа: строка удовлетворяет шаблону или нет. Символы, описанные здесь объявлены в " fnmatch.h ".

```
int fnmatch (const char *pattern, const char *string, int flags)
```

Эта функция проверяет, соответствует ли указанная строка шаблону. Она возвращает 0, если они соответствуют; иначе, она возвращает значение FNM_NOMATCH отличное от нуля.

Flags - комбинация флаговых битов, которые изменяют подробности

соответствия. См. ниже список определенных флагов.

В Библиотеке GNU C, fnmatch не может испытывать "ошибку", она всегда возвращает ответ, преуспевает ли соответствие. Однако, другие реализации fnmatch могут иногда сообщать "ошибки", возвращая значения отличные от нуля, которые не равны FNM_NOMATCH.

Вот доступные флаги для аргумента flags:

FNM_FILE_NAME

Если этот флаг установлен, универсальные конструкции символов в шаблоне не могут соответствовать " / " в строке. Таким образом, единственый способ соответствовать " / " явно указать " / " в шаблоне.

FNM_PATHNAME

Это - побочный результат исследования для FNM_FILE_NAME; он исходит ИЗ POSIX.2. Мы не рекомендуем это имя, потому что мы не используем термин "имя пути" для имен файла.

FNM_PERIOD

Обрабатывает "." особенно, если она появляется в начале строки. Если этот флаг установлен, универсальные конструкции символов в шаблоне не могут соответствовать "." (первый символ строки).

Если Вы устанавливаете, и FNM_PERIOD и FNM_FILE_NAME, то "." после "/" трактуется также как к "." в начале строки. (Оболочка использует FNM_PERIOD и FNM_FILE_NAME флаги вместе для соответствия имен файлов.)

FNM_NOESCAPE

Не обрабатывает символ `\' в шаблонах. Обычно, " \ " означает (цитирует) следующий символ непосредственно. Когда цитирование допускается, шаблон " \? " означает строку "? ", потому что

вопросительный знак в шаблоне действует подобно обычному символу.

Если Вы используете FNM_NOESCAPE, то " \ " является обычным символом.

FNM_LEADING_DIR

Игнорирует конечную последовательность символов, начинающихся с " / " в строке.

Если этот флаг установлен, и " foo* " и " foobar " как шаблоны, соответствуют строке " foobar/frobozz ".

FNM_CASEFOLD

Игнорирует case при сравнении строки и шаблона.

16.2 Globbing

Типичное использование групповых символов - для соответствия файлов в каталоге, и создании списка всех соответствий. Это называется globbing.

Вы могли бы делать это используя fnmatch, читая входы каталога один за другим и проверяя каждый с fnmatch. Но это было бы медлено (и сложно, так как Вы будете должны обработать подкаталоги вручную).

Библиотека обеспечивает функцию glob, чтобы делать это с использованием удобных универсальных символов. Glob и другие символы в этом разделе объявлены в " glob.h ".

16.2.1 Вызов glob

Результат globbing - вектор имен файлов. Чтобы возвращать этот вектор, glob использует специальный тип данных, glob _t, который является структурой. Вы передаете glob адрес структуры, и она вносит поля структуры, чтобы сообщить Вам результаты.

glob_t (тип данных)

Этот тип данных содержит указатель на вектор слов. Более точно, он содержит, и адрес вектора слов и размер.

Gl_pathc Число элементов в векторе.
Gl_pathv Адрес вектора. Это поле имеет тип char **.
Gl_offs Смещение первого реального элемента вектора, от номинального адреса в gl_pathv поле. В отличие от других полей, это -- всегда ввод glob, а не вывод из нее (т. е. вы должны указать его).

Если Вы используете смещение отличное от нуля, то много элементов в начале вектора будут оставлены пустыми. (Функция glob заполняет их пустыми указателями.)

Gl_offs поле значимо только, если Вы используете GLOB_DOOFFS флаг.

Иначе, смещение - всегда нуль независимо от того, что находится в этом поле, и первый реальный элемент расположен в начале вектора.

int glob (const char *pattern, int flags, int (*errfunc) (const char *filename, int error-code), glob_t *vector_ptr)

Эта функция делает globbing, используя указанный шаблон в текущем каталоге. Она помещает результат в недавно размещенном векторе, и сохраняет размер и адрес этого вектора в *vector_ptr.

Flags аргумент - комбинация битовых флагов; см. Раздел 16.2.2 [Флаги для Globbing].

Результат globbing - последовательность имен файлов. Glob резервирует строку для каждого возникающего в результате слова, и вектор типа char **, чтобы сохранить адреса этих строк. Последний элемент вектора - пустой указатель. Этот вектор называется вектором слов.

Чтобы возвратить этот вектор, glob сохраняет и адрес и длину

(число элементов, не считая завершающий пустой символа) в *vector_ptr.

Обычно, glob сортирует имена файлов в алфавитном порядке перед их возвращением. Вы можете указать флаг GLOB_NOSORT, если Вы хотите получать информацию, с наибольшей скоростью.

Если glob преуспевает, она возвращает 0. Иначе, она возвращает один из этих кодов ошибки:

GLOB_ABORTED Имелась ошибка открытия каталога, и Вы использовали флаг GLOB_ERR, или ваша заданная errfunc возвратила значение отличное от нуля. См. ниже объяснение GLOB_ERR и errfunc.

GLOB_NOMATCH Шаблон не соответствовал ни каким существующим файлам. Если Вы используете GLOB_NOCHECK флаг, то Вы, никогда не получаете этот код ошибки, потому что этот флаг сообщает, чтобы glob симулировал что шаблону соответствует по крайней мере один файл.

GLOB_NOSPACE Было невозможно зарезервировать память, чтобы содержать результат.

В случае ошибки, glob сохраняет информацию в *vector_ptr относительно всей соответствий, которые она уже нашла.

16.2.2 Флаги для Glob

Этот раздел описывает флаги, которе Вы можете определять в аргументе flags в glob. Выберите флаги которые Вы хотите, и объедите их оператором OR (| в Си).

GLOB_APPEND

Добавлять слова от этого поиска к вектору слов, произведенных предыдущими обращениями к glob.

Для этого, Вы не должны изменить содержимое структуры вектора слов между обращениями к glob. И, если Вы устанавливаете

GLOB_DOOFFS в первом обращении к glob, Вы должны также установить его, когда Вы добавляете.

Обратите внимание, что указатель, сохраненный в gl_pathv может больше не быть допустимым после того, как Вы вызываете glob второй раз, потому что glob может переместить вектор. Так что всегда берите gl_pathv из структуры _t glob после каждого обращения к glob; никогда не сохраните указатель между обращений.

GLOB_DOOFFS

Оставьте пустые места в начале вектора слов. Gl_offs поле говорит сколько мест оставить. Пустые места содержат пустые указатели.

GLOB_ERR

Сразу же сообщает ошибку, если имеется любая трудность. Такие трудности могут включать каталог, в котором Вы не имеете необходимого доступа. Обычно, glob пробует продолжить несмотря на любые ошибки, и читает любые каталоги, какие может.

Вы можете осуществлять управление, определяя функцию обработчика ошибки errfunc, когда Вы вызываете glob. Если errfunc - не пустой указатель, то glob не отказывается сразу же, когда она не может читать каталог; взамен, она вызывает errfunc с двумя аргументами, примерно так:

(*errfunc) (filename, error-code)

Filename - имя каталога, который glob не может открыть, или не может читать, а error-code - значение errno, которое было сообщено glob.

Если функция обработчика ошибки возвращает не нуль, то glob завершается сразу же. Иначе, она продолжается.

GLOB_MARK Если шаблон соответствует имени каталога, конкатенирует " / " к

имени каталога при его возвращении.

GLOB_NOCHECK Если шаблон не соответствует ни каким именам файлов, возвращает шаблон непосредственно, как будто это было имя файла. (Обычно, когда шаблон не соответствует чему - нибудь, glob говорит что не имелось никакого соответствия.)

GLOB_NOSOR(***) Не сортирует имена файлов. (Практически, порядок будет зависеть от порядка входов в каталоге.) Единственая причина не сортировать состоит в том, чтобы сохранить время.

GLOB_NOESCAPE Не обрабатывает символ ` \ ' в шаблонах. Обычно, " \ " цитирует следующий символ, выключая специальное значение так, чтобы он соответствовал только непосредственно символу. Когда цитирование допускается, шаблон " \? " соответствует только строке "? ", потому что вопросительный знак в шаблоне действует подобно обычному символу.

Если Вы используете GLOB_NOESCAPE, то " \ " является обычным символом.

Glob делает это, вызывая функцию fnmatch. Она обрабатывает флаг GLOB_NOESCAPE, включая FNM_NOESCAPE флаг в обращениях к fnmatch.

16.3 Соответствия Регулярных Выражений

Библиотека GNU C поддерживает два интерфейса для соответствия регулярных выражений. Один - стандартный POSIX.2 интерфейс, а другой - тот, который система GNU использовала много лет.

Оба интерфейса объявлены в заглавном файле " regex.h ". Если Вы определяете _POSIX_C_SOURCE, то будут объявлены только POSIX.2 функции, структуры, и константы.

16.3.1 POSIX Регулярные Выражения

Прежде, чем Вы можете фактически использовать регулярное выражение, Вы должны откомпилировать его. Это - не истинная трансляция, это производит специальную структуру данных, а не машинные команды. Но это - подобно обычной трансляции, цель которой дать Вам возможность " выполнить" шаблон быстро. (См. Раздел 16.3.3 [Соответствие POSIX Регулярным Выражениям], для того, как использовать компилируемое регулярное выражение для соответствия.)

Имеется специальный тип данных для компилируемых регулярных выражений:

regex_t (тип данных)

Этот тип содержит компилируемое регулярное выражение. Это - фактически структура. Она имеет только одно поле, которое ваши программы должны рассмотреть: re_nsub . Это поле содержит некоторое число вводных подвыражений регулярного выражения.

Имеются и другие поля, но мы не описываем их здесь, потому что только функции в библиотеке должны использовать их.

После того, как Вы создаете объект regex_t, Вы можете компилировать регулярное выражение в ней, вызывая regcomp.

int regcomp (regex_t *compiled, const char *pattern, int cflags)

Функция regcomp "компилирует" регулярное выражение в структуру данных, которую Вы можете использовать с regexec, чтобы искать соответствующие строки. Компилируемый формат регулярного выражения разработан для эффективного соответствия. Regcomp сохраняет его в *compiled.

Вам нужно только зарезервировать объект типа regex_t и передать адрес regcomp.

Аргумент cflags допускает Вам, определять различные опции, которые управляют синтаксисом и семантикой регулярных выражений.

Если Вы используете флаг REG_NOSUB, то regcomp, опускает из компилируемого регулярного выражения информацию, необходимую для записи соответствий подвыражений. В этом случае, Вы можете также указывать 0 для matchptr и nmatch аргументов, когда Вы вызываете regexec.

Если Вы не используете REG_NOSUB, то компилируемое регулярное выражение имеет запись соответствия подвыражений. Также, regcomp сообщает Вам, сколько подвыражений имеет шаблон, сохраняя число в compiled->re_nsub. Вы можете использовать это значение, чтобы решить, какую длину массива зарезервировать, чтобы содержать информацию относительно соответствий подвыражений.

Regcomp возвращает 0, если она преуспевает в компилировании регулярного выражения; иначе, она возвращает код ошибки отличный от нуля (см. таблицу ниже). Вы можете использовать regerror, чтобы произвести строку сообщения об ошибках для значений отличных от нуля.

Имеются возможные значения отличные от нуля, которые regcomp может возвращать:

REG_BADBR Имеется недопустимая конструкция " \{. . .\} " в регулярном выражении. Допустимая " \{. . .\} " конструкция должна содержать или одиночное число, или два числа в увеличивающемся порядке, отделенные запятой.

REG_BADPAT Имелась синтаксическая ошибка в регулярном выражении.

REG_BADRPT Оператор повторения типа " ? " или " * " оказался, в плохой позиции (без предшествующего подвыражения).

REG_ECOLLATE Регулярное выражение сносится на недопустимый элемент объединения (не определенный в текущем стандарте для строкового

объединения). См. Раздел 19.3 [Категории Стандарта].

REG_ECTYPE Регулярное выражение ссылается на недопустимое символьное имя класса.

REG_EESCAPE Регулярное выражение законченно " \ ".

REG_ESUBREG Имелось недопустимое число в " \digit " конструкции.

REG_EBRACK Имелись несбалансированные квадратные скобки в регулярном выражении.

REG_EPAREN Расширенное регулярное выражение имело незакрытые скобки, или базисное регулярное выражение имело несбалансированные "\(" и "\)".

REG_EBRACE Регулярное выражение имело несбалансированные "\{" и "\}".

REG_ERANGE Одна из оконечных точек в выражении диапазона была недопустима.

REG_ESPACE Regcomp не хватает памяти.

16.3.2 Флаги для POSIX Регулярных Выражений

Это - битовые флаги, который Вы можете использовать в cflags операнде при компилировании регулярного выражения с regcomp.

REG_EXTENDED

Обрабатывает шаблон как расширенное регулярное выражение, а не как базисное регулярное выражение.

REG_ICASE

Игнорирует case при соответствии символов.

REG_NOSUB

Не сохраняет содержимое matches_ptr массива.

REG_NEWLINE

Обрабатывает символ перевода строки в строке как деление строки на многократные строки, так, чтобы " $ " мог соответствовать перед символом перевода строки, а " ^ " мог соответствовать после. Также, не разрешает "." соответствовать символу перевода строки, и не разрешает " [^. . .] " соответствовать символу перевода строки.

Иначе, символ перевода строки действует подобно любому другому обычному символу.

16.3.3 Соответствие Компилируемого POSIX Регулярного Выражения

Если только Вы компилировали регулярное выражение, как описано в Разделе 16.3.1 [Трансляция POSIX Регулярных выражений], Вы можете применять его к строкам используя regexec. Соответствие где-нибудь внутри строки считается как успех, если регулярное выражение не содержит символы " ^ " или " $ ".

int regexec (regex_t *compiled, char *string, size_t nmatch, regmatch_t matchptr [], int eflags) (функция)

Эта функция пробует подобрать соответствие компилируемому регулярному выражению *compiled.

Regexec возвращает 0 если tcnm соответствие выражению; иначе, она возвращает значение отличное от нуля. См. таблицу ниже для того, что означают значения отличные от нуля. Вы можете использовать regerror, чтобы произвести строку сообщения об ошибках для значений отличных от нуля.

Аргумент eflags - слово битовых флагов, которые дают возможность различным опциям.

Если Вы хотите получать информацию относительно части строки

которая фактически соответствовала регулярному выражению или подвыражению, используйте аргументы matchptr и nmatch. Иначе, укажите 0 для nmatch, и пустой указатель для matchptr.

Функция regexec принимает следующие флаги в аргументе eflags:

REG_NOTBOL Не расценивает начало заданной строки как начало строки; более вообще, не делает ни каких предположений относительно того, что текст мог бы предшествовать ей.

REG_NOTEOL Не расценивает конец заданной строки как конец строки; более обще, не делает ни каких предположений относительно того, что текст мог бы следовать за ней.

Имеются возможные значения отличные от нуля, которые regexec может возвращать:

REG_NOMATCH Шаблон не соответствовал строке. Это в общем не ошибка.

REG_ESPACE Regexec не хватило памяти.

16.3.4 Результаты Соответствия с Подвыражениями

Когда regexec находит соответствия подвыражениям шаблона, она записывает, которым частям строки они соответствуют. Она возвращает эту информацию, сохраняя смещения в массиве, чьи элементы являются структурами типа regmatch_t. Первый элемент массива (индекс 0) записывает часть строки, которая соответствовала всему регулярному выражению. Каждый другой элемент массива записывает начало и конец части, которая соответствовала одиночному вводному подвыражению.

regmatch_t

Это - тип данных matcharray массива, который Вы передаете к regexec. Он содержит два поля-структуры, следующим образом:

Rm_so - Смещение начала подстроки в строке. Добавьте это значение к строке, чтобы получить адрес этой части.

Rm_eo - Смещение конца подстроки в строке.

regoff_t

Regoff_t - побочный результат исследования для другого целого типа со знаком. Поля regmatch_t имеют тип regoff_t.

Regmatch_t элементы соответствуют подвыражениям позиционно; первый элемент (индекс 1) хранит, где первое согласованное подвыражение, второй элемент записывает второе подвыражение, и так далее. Порядок подвыражений - порядок, в котором они начинаются.

Когда Вы вызываете regexec, Вы определяете длину matchptr массива, с nmatch аргументом.

Это сообщает regexec сколько элементов сохранить. Если фактическое регулярное выражение имеет больше чем nmatch подвыражений, то, Вы не будет получать информацию о смещениях относительно остальной их части.

Еслио Вы не хотите, чтобы regexec возвращал любую информацию относительно подвыражений, Вы можете обеспечить 0 для nmatch, или использовать флаг REG_NOSUB, когда Вы компилируете шаблон с regcomp.

16.3.5 Осложнения в Соответствиях Подвыражений

Иногда подвыражение соответствует подстроке без символов. Это случается, когда " f\(o*\) " соответствует строке " fum ". (Оно действительно соответствует только " f ".) В этом случае, оба смещения идентифицируют отметку в строке, где была найдена пустая подстрока. В этом примере, оба смещения 1.

Иногда все регулярное выражение может соответствовать без использования некоторых из подвыражений вообще, например, когда " ba\(na\)* " соответствует строке " ba ", вводное подвыражение не используется. Когда это случается, regexec сохраняет -1 в обоих полях элемента для этого подвыражения.

Иногда при соответствии всего регулярного выражения некоторая подстрока может соответствовать специфическому подвыражению больше чем один раз например, когда " ba\(na\)* " соответствует строка " bananana ", вводное подвыражение соответствует три раза. Когда это

случается, regexec обычно сохраняет смещения последней части строки, которая соответствовала подвыражению. В случае " bananana ", эти смещения - 6 и 8.

16.3.6 Очистка POSIX Regexp Соответствий

Когда Вы закончили использование компилируемого регулярного выражения, Вы можете освободить память, которую оно использует, вызывая regfree.

void regfree (regex_t *compiled) (функция)

Вызов regfree освобождает всю память, на которую *compiled указывает. Включая различные внутренние поля структуры regex_t, которые не описаны в этом руководстве.

Regfree не освобождает объект *compiled непосредственно.

Вы должны всегда освобождать место в структуре regex_t с regfree перед использованием структуры, чтобы компилировать другое регулярное выражение.

Когда regcomp или regexec сообщает об ошибке, Вы можете использовать функцию regerror, преобразовать ее в строку сообщения об ошибках.

size_t regerror (int errcode, regex_t *compiled, char *buffer, size_t length) (функция)

Эта функция производит строку сообщения об ошибках для кода ошибки errcode, и сохраняет строку в length байтах памяти, начинающейся с buffer. Для аргумента compiled, обеспечьте то же самое регулярное выражение, с которой работал regcomp или regexec когда получил ошибку. В качестве альтернативы, Вы можете обеспечивать пустой указатель для compiled; Вы будете все еще получать значимое сообщение об ошибках, но оно может не быть детализировано.

Если сообщение об ошибках не помещается в length байтах (включая пустой символ завершения), то regerror усекает его. Эта строка всегда с нулевым символом в конце даже если она была усечена.

Возвращаемое значение regerror - минимальный length, для сохранения всего сообщения об ошибках. Если он меньше чем length, то сообщение об ошибках не было усечено, и Вы можете использовать его. Иначе, Вы должны вызвать regerror снова с большим buffer.

Вот функция, которая использует regerror, но всегда динамически, резервируя буфер для сообщения об ошибках:

```
char *get_regerror (int errcode, regex_t *compiled)
{
  size_t length = regerror(errcode,compiled,NULL,0);
  char *buffer = xmalloc (length);
  (void) regerror (errcode, compiled, buffer, length);
  return buffer;
}
```

16.4 Разложение Слов в стиле оболочки

Разложение Слов означает процесс разбивания строки в слова и замену переменных, команд, и универсальных символов, точно так как делает оболочка.

Например, когда Вы пишите " ls -l foo.c ", эта строка разбивается в три отдельных слова " ls ", " -l " и " foo.c ". Это - базисная функция разложения слов.

Когда Вы пишите " ls *.c ", это может стать многими словами, потому что слово " *.c " может быть заменено на любое число имен файлов. Это называется разложением универсального символа, и это - также часть разложения слова.

Когда Вы используете " ECHO $PATH " чтобы печатать ваш путь, Вы пользуетесь преимуществом замены переменной, которая является также частью разложения слова.

Обычные программы могут выполнять разложение слова точно так же как оболочка, вызывая библиотечную функцию wordexp.

16.4.1 Стадии Разложения Слова

Когда разложение слова применяется к последовательности слов, выполняются следующие преобразования в порядке, показанном здесь:

1. Разложение Тильды: Замена " ~foo " на имя исходного (home) каталога " foo ".
2. Затем, применяются три различных преобразования в том же самом шаге, слева направо:
   * Замена переменных: Переменные среды заменяются для ссылок типа " $foo ".
   * Замена Команд: Конструкции типа "" cat foo "" и эквивалент "

$(cat foo) " заменены на вывод внутренней команды.
   * Арифметическое разложение: Конструкции типа " $(($x-1)) " заменены на результат арифметического вычисления.
3. Разбивание Поля: разбиение текста в слова.
4. Разложение Универсальных символов: замена конструкции типа " *.c " на список " .c " имен файлов. Разложение Универсального символа применяется к всему слову одновременно, и заменяет это слово на 0 или большое количество имен файлов, которые являются самостоятельными словами.
5. Удаление Кавычек: стирание кавычек, теперь, когда они сделали их работу, запрещая вышеупомянутые преобразования когда нужно.

Для подробностей этих преобразований, и как написать использующие их конструкции, см. Руководство BUSH (должно появиться).

16.4.2 Вызов wordexp

Все функции, константы и типы данных для разложения слова объявлены в заглавном файле " wordexp.h ".

Разложение Слова производит вектор слов (строк). Чтобы возвращать этот вектор, wordexp использует специальный тип данных, wordexp_t, который является структурой. Вы передаете wordexp адрес структуры, и она вносит поля структуры, чтобы сообщить Вам результаты.

wordexp_t

Этот тип данных содержит указатель на вектор слов. Более точно, в нем записаны, и адрес вектора слов и размер.

We_wordc - Число элементов в векторе.

We_wordv - Адрес вектора. Это поле имеет тип char **.

We_offs - Смещение первого реального элемента вектора от номинального адреса в we_wordv поле. В отличие от других полей, это всегда ввод к wordexp, а не вывод из нее.

Если Вы используете смещение отличное от нуля, то многие элементы в начале вектора будут оставлены пустыми. (Wordexp функция заполняет их пустыми указателями.)

We_offs поле значимо только, если Вы используете WRDE_DOOFFS флаг. Иначе, смещение - всегда нуль независимо от того, что находится в этом поле, и первый реальный элемент находится в начале вектора.

int wordexp (const char *words, wordexp_t *word-vector-ptr, int flags)

Выполните разложение слова на строке слов, помещая результат в недавно размещенном векторе, и сохраните размер и адрес этого вектора в *word-vector-ptr. Аргумент flags - комбинация битовых флагов; см. Раздел 16.4.3 [Флаги для Wordexp].

Вы не должны использовать любой из символов " | & ; < > " в строке слов, если они не заключены в кавычки; аналогично для символа перевода строки. Если Вы используете эти символы без кавычек, Вы получите WRDE_BADCHAR код ошибки. Не используйте круглые скобки или фигурные скобки, если они заключены в кавычки или часть конструкции разложения слова. Если Вы используете кавычки " ' `, они должны войти попарно.

Результаты разложения слов - последовательность слов. Функция wordexp зарезервирует строку для каждого возникающего в результате слова, и вектор типа char **, чтобы сохранить адреса этих строк. Последний элемент вектора - пустой указатель. Этот вектор называется вектором слов.

Чтобы возвращать этот вектор, wordexp сохраняет, и адрес и длину

(число элементов, не считая завершающий пустой символ) в *word-vector-ptr.

Если wordexp завершает работу успешно, она возвращает 0. Иначе, она возвращает один из этих кодов ошибки:

WRDE_BADCHAR

Входные строковые слова содержат незащищенный кавычками недопустимый символ типа " | ".

WRDE_BADVAL

Входная строка обращается к неопределенной переменной оболочки, и Вы использовали флаг WRDE_UNDEF, чтобы запретить такие ссылки.

WRDE_CMDSUB

Входная строка использует замену команды, и Вы использовала флаг WRDE_ NOCMD, чтобы запретить замену команд.

WRDE_NOSPACE

Было невозможно зарезервировать память для результат. В этом случае, wordexp может сохранять часть результатов, столько, сколько она смогла зарезервировать памяти.

WRDE_SYNTAX

Имелась синтаксическая ошибка во входной строке. Например, несогласованные кавычки - синтаксическая ошибка.

void wordfree (wordexp_t *word-vector-ptr) (функция)

Освободит память, используемую для строки слов и вектора, на который указывает * word-vector-ptr. Она не освобождает структуру *word-vector-ptr непосредственно, а только другие данные, на которые она указывает.

16.4.3 Флаги для Разложения Слова

Этот раздел описывает флаги, которые Вы можете определять в аргументе flags wordexp. Выберите флаги, которые Вы хотите, и объединяете их оператором |.

WRDE_APPEND

Добавляет слова этого разложения к вектору слов, произведенных предыдущими обращениями к wordexp.

Для этого Вы не должны изменять содержимое структуры вектора слов между обращениями к wordexp. И, если Вы устанавливаете WRDE_DOOFFS в первом обращении к wordexp, Вы должны также установить его, когда Вы добавляете.

WRDE_DOOFFS

Оставляет пустое место в начале вектора слов. We_offs поле говорит, сколько места оставить. Пустое место содержит пустые указатели.

WRDE_NOCMD

Не делает замену команд; при попытке замены команды, сообщаает об ошибке.

WRDE_REUSE

Многократно использует вектор слов, сделанный предыдущим обращением к wordexp. Вместо того, чтобы зарезервировать новый вектор слов, это обращение к wordexp использует вектор, который уже существует (увеличивая его в случае необходимости).

Обратите внимание, что вектор может перемещаться в памяти, так что небезопасно хранить старый указатель и использовать его снова после вызова wordexp. Вы должны сохранять we_pathv после каждого обращения.

WRDE_SHOWERR

Покажет любые сообщения об ошибках.

WRDE_UNDEF

Если ввод относится к переменной оболочки которая не определена, выдает ошибку.

16.4.4 Пример Wordexp

Вот пример использования wordexp, чтобы рзложить отдельные строки и использования результатов чтобы выполнить команду оболочки. Он также показывает использование WRDE_APPEND, чтобы добавлять разложения и wordfree, чтобы освободить место, размещенное wordexp.

```
int
expand_and_execute(const char*program,const char*options)
{
 wordexp_t result;
```

```
pid_t pid
int status, i;
switch (wordexp (program, &result, 0))
{
        case 0:
                break;
        case WRDE_NOSPACE:
                wordfree (&result);
        default:
                return -1;
}
for (i = 0; args[i]; i++)
{
        if (wordexp (options, &result, WRDE_APPEND))
        {
                wordfree (&result);
                return -1;
        }
}
pid = fork ();
if (pid == 0)
{
        execv (result.we_wordv[0],
  result.we_wordv);
        exit (EXIT_FAILURE);
```



```
}
else if (pid < 0)
        foo1   = -1;
else
         if (waitpid (pid, &status, 0) != pid)
foo1   = -1;
wordfree (&result);
return status;
```

Практически, т. к. wordexp работает, выполняя подоболочку, было бы быстрее сделать это, связывая строки с пробелами между ними и выполняя это как команду оболочки, используюя " sh -c ".

17. Дата и время

Эта глава описывает функции для управления датой и временем, включая функции для определения текущего времени и преобразование между различными представлениями времени.

Функции времени относятся к трем категориям:

* Функции для измерения прошедшего времени CPU обсуждены в Разделе 17.1 [Время Процессора].
* Функции календарного времени обсуждены в Разделе 17.2 [Календарное Время].
* Функции для установки будильников и таймеров обсуждены в Разделе 17.3 [Установка Сигнализации].

17.1 Время Процессора

Если вы пробуете оптимизировать вашу программу или измерять эффективность, очень полезно знать, сколько времени процессора или CPU времени она использовала в любой заданной точке. Процессорное время является отличным от фактических часов, потому что оно не включает все потраченное время на ожидание ввода-вывода или когда выполняется некоторый другой процесс. Процессорное время представляется типом данных clock_t, и дано как ряд импульсов времени относительно произвольного базового времени, отмечающего начало одиночного вызова программы.

17.1.1 Запрос Основного Времени CPU

Чтобы получить прошедшее CPU время, используемое процессом, Вы можете использовать функцию clock. Это средство объявлено в заглавном файле " time.h ".

Обычно, Вы вызываете функцию clock в начале и конца интервала,

который Вы хотите измерить, вычитаете значения, и тогда делите на CLOCKS_PER_SEC (число импульсов времени clock в секунду), примерно так:

```
#include
clock_t start, end;
double elapsed;
start = clock();
. . . /* Do the work. */
end = clock();
elapsed=((double)(end-start))/CLOCKS_PER_SEC;
```

Различные компьютеры и операционные системы сильно отличаются в том, как они следят за процессорным временем. Общее для внутренних часов процессора то, что разрешающая способность где-то между тысячной и милионной долей секунды.

В системе GNU, clock _t эквивалентен long int, а CLOCKS_PER_SEC - целочисленное значение. Но в других системах, и clock _t и тип макркоманды CLOCKS_PER_SEC может быть или целое число, или с плавающей точкой. Приведением значения времени процессора к double, см. пример выше, удостоверяется, что нужные операции работают правильно и последовательно независимо от того, каково основное представление.

int CLOCKS_PER_SEC

Значение этой макркоманды - число импульсов времени в секунду, измеряемое функцией clock.

int CLK_TCK

Это - устаревшее имя для CLOCKS_PER_SEC.

clock_t (тип данных)

Это - тип значения, возвращенного функцией clock. Значения типа clock_t измеряются в единицах импульсов сигналов времени clock.

clock_t clock (void) (функция)

Эта функция возвращает прошедшее процессорное время. Базовое время произвольно, но не изменяется внутри одиночного процесса.

Если процессорное время не доступно или не может представляться, clock возвращает значение (clock_t) (-1).

17.1.2 Детализированный Запрос Времени CPU

Функция times возвращает более детализированную информацию относительно прошедшего процессорного времени в struct tmsobject. Вы должны включить заглавный файл " sys/times.h " чтобы использовать это средство.

struct tms (тип данных)

Структура tms используется, чтобы возвратить информацию относительно времени процесса. Она содержит по крайней мере следующие элементы:

clock_t tms_utime

Это - процессорное время, используемое при выполнении команд вызывающего процесса.

clock_t tms_stime

Это - процессорное время, используемое системой от имени вызываюлщего процесса.

clock_t tms_cutime

Это - сумма значений tms_utime и значений tms_cutime всех завершенных дочерних процессов данного процесса. Другими словами, она представляет общее процессорное время, используемое при выполнении команд всех завершенных дочерних процессов вызывающего процесса.

clock_t tms_cstime

Подобно tms_cutime, но представляет общее процессорное время, используемое системой от имени всех завершенных дочерних процессов.

Все времена даны в импульсах сигналов времени. Они - абсолютные значения; в новом процессе, они - все нуль. См. Раздел 23.4 [Создание Процесса].

clock_t times (struct tms *buffer) (функция)

Функция times сохраняет процессорное время для вызывающего процесса в buffer.

Возвращаемое значение - также как значение clock (): прошедшее реальное время относительно произвольной основы. Основа - константа внутри специфического процесса, и обычно представляет время начиная с запуска системы. Значение (clock_t) (-1) возвращается, чтобы указать отказ.

Примечание Переносимости: функция clock, описанная в Разделе

17.1.1 [Базисное процессорное Время], определена в соответствии с стандартом ANSI C. Функция times - возможность POSIX.1. В системе GNU, значение, возвращенное функцией clock эквивалентно сумме tms_utime и tms_stime полей, возвращенных times.

17.2 Календарное Время

Этот раздел описывает средства для слежения за датами и временем согласно Грегорианскому календарю.

Имеются три представления информации даты и времени:

* Календарное время (time_t тип данных) - компактное представление, обычно дает число секунд, истекающих начиная с некоторого основного времени.

* Имеется также представление времени с высоким разрешением (struct timeval тип данных) которое включает доли секунды. Используйте это представление времени вместо обычного календарного времени, когда нужна большая точность.

* Местное время (struct tm тип данных) представляет дату и время как набор компонентов, определяющих год, месяц, и так далее, для специфического часового пояса. Это представление обычно используется вместе с форматированием значений даты и времени.

17.2.1 Простое Календарное Время

Этот раздел описывает time_t тип данных для представления календарного времени, и функции, которые используют объекты календарного времени. Эти средства объявлены в заглавном файле " time.h ".

time_t

Это - тип данных, используемый, чтобы представить календарное время. В библиотеке GNU C и других POSIX-реализациях, time_t эквивалентен long int. Он интерпретируется как абсолютное значение времени и представляет число секунд, истекающих с 00:00:00 1 января, 1970, Координированного Универсального Времени. (Эта дата иногда упоминается как эпоха.)

В других системах, time_t может быть или целым числом или с плавающей запятой.

double difftime (time_t time1, time_t time0) (функция)

Функция difftime возвращает число секунд, между временем time1 и временем time0, как значение типа double.

В системе GNU, Вы можете просто вычитать значения time_t. Но в других системах, time_t тип данных может использовать некоторое другое кодирование, где вычитание не работает непосредственно.

time_t time (time_t *result)

Функция time возвращает текущее время как значение типа time_t. Если аргумент result - не пустой указатель, значение time, также будет сохранено в *result. Если календарный time не доступен, возвращается значение (time_t) (-1).

17.2.2 Календарь с высоким разрешением

Тип данных time_t, используемый, чтобы представить календарное вермя имеет разрешающую способность только в одну секунду.

Некоторые приложения нуждаются в большей точности.

Так, библиотека GNU C также содержит функции, которые способны представить календарь с более высокой разрешающей способностью чем одна секунда. Функции и связанные типы данных, описанные в этом разделе объявлены в " sys/time.h ".

struct timeval (тип данных)

Структура struct timeval представляет календарное время. Она имеет следующие элементы:

long int tv_sec

Этот представляет число секунд начиная с эпохи. Это эквивалентно нормальному значению time_t.

long int tv_usec

Это - дробное второе значение, представляемое как число микросекунд.

Некоторые значения struct timeval - используются для временных интервалов. Тогда tv_sec элемент - число секунд в интервале, а tv_usec - число микросекунд.

struct timezone (тип данных)

Структура struct timezone используется, чтобы содержать минимальную информацию относительно зоны местного времени. Она имеет следующие элементы:

int tz_minuteswest
Это - число минут к западу от ГРИНВИЧа.
int tz_dsttime
Если отличен от нуля, сдвинутое время применяется в течение некоторой части года.

Struct timezone устаревший тип и не должен использоваться. Вместо этого, используйте средства, описанные в Разделе 17.2.6 [Функции Часового пояса].
Часто необходимо вычесть два значения типа struct timeval. Вот самый лучший способ делать это. Он работает даже на некоторых специфических операционных системах, где tv_sec элемент имеет тип unsigned.

```
int
timeval_subtract (result, x, y)
                struct timeval *result, *x, *y;
{
        if (x->tv_usec < y->tv_usec) {
                int nsec = (y->tv_usec-x->tv_usec)/1000000+1;
                y->tv_usec -= 1000000 * nsec;
                y->tv_sec += nsec;
        }
        if (x->tv_usec - y->tv_usec > 1000000) {
                int nsec = (y->tv_usec-x->tv_usec)/1000000;
                y->tv_usec += 1000000 * nsec;
                y->tv_sec -= nsec;
        }
        result->tv_sec = x->tv_sec - y->tv_sec;
        result->tv_usec = x->tv_usec - y->tv_usec;
        return x->tv_sec < y->tv_sec;
}
```

int gettimeofday (struct timeval *tp, struct timezone *tzp) (функция)
Функция gettimeofday возвращает текущую дату и время в структуре struct timeval, обозначенной tp. Информация относительно часового пояса возвращается в структуре, указанной в tzp. Если аргумент tzp является пустым указателем, информация часового пояса, игнорируется.
Возвращаемое значение - 0 при успехе и -1 при отказе. Следующее errno условие ошибки определено для этой функции:
ENOSYS операционная система не поддерживает получение информации часового пояса, и tzp - не пустой указатель. Операционная система GNU не поддерживает использование struct timezoneto для представления информации часового пояса; это - устаревшая

возможность 4.3 BSD. Вместо этого, используйте средства, описанные в Разделе 17.2.6 [Функции Часового пояса].
int settimeofday (const struct timeval *tp, const struct timezone *tzp)
Функция settimeofday устанавливает текущую дату и время согласно аргументам. Что касается gettimeofday, информация часового пояса игнорируется, если tzp - пустой указатель.
Вы должны быть привилегированным пользователем, чтобы использовать settimeofday.
Возвращаемое значение - 0 при успехе и -1 при отказе. Следующие errno условия ошибки определены для этой функции:
EPERM Этот процесс не может устанавливать время, потому что он не привилегированный.
ENOSYS операционная система не поддерживает установку информации часового пояса, и tzp - не пустой указатель.
int adjtime (const struct timeval *delta, struct timeval *olddelta)
Эта функция ускоряет или замедляет часы системы, чтобы делать постепенные корректировки текущего времени. Она гарантирует, что время, сообщенное часами системы всегда монотонно увеличивается, чего не могло случаться, если Вы просто устанавливаете текущее время.
Аргумент delta определяет относительную корректировку, которая будет сделана относительно текущего времени. Если он отрицателен, часы системы замедляются. Если положителен, часы системы ускоряются.

Если аргумент olddelta - не пустой указатель, функция adjtime, возвращает информацию относительно любой предыдущей корректировки, которая еще не завершилась.

Эта функция обычно используется, чтобы синхронизировать часы компьютеров в местной сети.

Вы должны быть привилегированным пользователем, чтобы использовать ее. Возвращаемое значение - 0 при успехе и -1 при отказе. Следующее errno условие ошибки определено для этой функции:

EPERM Вы не имеют привилегий, чтобы установить время.

Примечание Переносимости: функции gettimeofday, settimeofday, и adjtime - из BSD.

17.2.3 Разделенное Время

Календарное время представляется как число секунд. Это удобно для вычисления, но не имеет никакого отношения к способу, которым люди обычно представляют даты и время. Нпротив, разделенное время - двоичное представление, разделенное на год, месяц, день, и так далее.

Разделенное время всегда зависит от выбора зоны местного времени, и оно также указывает, какой часовой пояс использовался.

Символы в этом разделе объявлены в заглавном файле " time.h ".

struct tm

Это - тип данных, используемый, чтобы представить разделенное время. Структура содержит по крайней мере следующие элементы, которые могут появляться в любом порядке:

int tm_sec

Это - число секунд, обычно в диапазоне от 0 до 59. (Фактическое верхнее ограничение 61, учитывая " прыгающие секунды ".)

int tm_min

Это - число минут, в диапазоне от 0 до 59.

int tm_hour

Это - число часов после полуночи, в диапазоне от 0 до 23.

int tm_mday

Это - день месяца, в диапазоне от 1 до 31.

int tm_mon

Это - число месяцев начиная с января, в диапазоне от 0 до 11.

int tm_year

Это - число лет начиная с 1900.

int tm_wday

Это - число дней начиная с воскресенья, в диапазоне от 0 до 6.

int tm_yday

Это - число дней начиная с 1 января, в диапазоне от 0 до 365.

int tm_isdst

Это - флаг, который указывает действует ли Смещение светового дня (или было, или будет) на описанное время. Значение положительно, если Смещение светового дня включено, нуль, если нет, и отрицательно, если информация не доступна.

long int tm_gmtoff

Это поле описывает часовой пояс, который использовался, чтобы

вычислить разделенное значение времени; это значение Вы должны добавить к местному времени в этой зоне, чтобы получить время ПО ГРИНВИЧУ, в секундах. Значение анологично переменной timezone (см. Раздел 17.2.6 [Функции Часового пояса]).

Tm_gmtoff поле - расширение библиотеки GNU.

const char *tm_zone

Это поле - трехсимвольное имя для часового пояса, который использовался, чтобы вычислить разделенное время. Это - расширение библиотеки GNU.

struct tm * localtime (const time_t *time) (функция)

Localtime функция преобразовывает календарное время, указываемое TIME в разделенное представление времени, выраженное относительно заданного часового пояса.

Возвращаемое значение - указатель на статическую структуру разделенного времени, которая могла бы быть записана поверх последующими обращениями к любой из функций date и time. (Но никакая другая библиотечная функция не записывает поверх содержимого этого объекта.)

Вызов localtime имеет и другой эффект: она устанавливает переменную tzname с информацией относительно текущего часового пояса. См. Раздел 17.2.6 [Функции Часового пояса].

struct tm * gmtime (const time_t *time) (функция)

Эта функция подобна localtime, за исключением того, что разделенное время выражено как Координированное Универсальное Время (UTC) то есть как Время ПО ГРИНВИЧУ а не относительно зоны местного времени.

Вспомните, что календарное время всегда выражается в координированном универсальном времени.

time_t mktime (struct tm *brokentime) (функция)

Mktime функция используется, чтобы преобразовать структуру разделенного времени в календарное время. Она также "нормализует" содержимое структуры разделенного времени, внося день недели и день года, основываясь на других компонентах даты и времени.

Mktime функция игнорирует заданное содержимое tm_wday и tm_yday элементов структуры разделенного времени. Она использует значения других компонентов, чтобы вычислить календарное время; для этих компонентов допустимо иметь ненормализованные значения вне их

нормальных диапазонов. Еще mktime корректирует компоненты структуры brokentime (включая tm_wday и tm_yday).

Если заданное разделенное время не может представляться как календарное время, mktime, возвращает значение (time_t) (-1) и не изменяет содержимое brokentime.

Вызов mktime также устанавливает переменную tzname с информацией относительно текущего часового пояса. См. Раздел 17.2.6 [Функции Часового пояса].

17.2.4 Форматирование Даты и времени

Функции, описанные в этом разделе форматирнуют значения времени как строки. Эти функции объявлены в заглавном файле " time.h ".

char * asctime (const struct tm *brokentime) (функция)

Функция asctime преобразовывает значение разделенного времени, на которое указывает brokentime в строку в стандартном формате:

```
"Tue May 21 13:46:22 1991\n"
```

Сокращения для дней недели: `Sun', `Mon', `Tue', `Wed', `Thu', `Fri', and `Sat'.

Сокращения для месяцев: `Jan', `Feb', `Mar', `Apr', `May', `Jun', `Jul', `Aug', `Sep', `Oct', `Nov', and `Dec'.

Возвращаемое значение указывает на статически размещенную строку, которая могла бы быть записана поверх последующими обращениями к любой из функций date и time. (Но никакая другая библиотечная функция не записывает поверх содержимого этой строки.)

char * ctime (const time_t *time)

Ctime функция подобна asctime, за исключением того, что значение времени определено в календарном времени (не местное время). Она эквивалентна asctime (localtime (time)) . ctime устанавливает переменную tzname, потому что так делает localtime. См. Раздел 17.2.6 [Функции Часового пояса].

size_t strftime (char *s, size_t size, const char *template, const struct tm *brokentime)

Эта функция подобна sprintf функции (см. Раздел 7.11 [Форматируемый Ввод]), но спецификации преобразования, которые могут появляться в шаблоне формата, специализированы для печати компонентов даты и времени brokentime согласно стандарту, в настоящее время заданному для преобразования времени (см. Главу 19 [Стандарты]).

Обычные символы, появляющиеся в шаблоне копируются в строку вывода s; она может включать многобайтовые символы. Спецификаторы Преобразования представляются символом ` % ', и заменяются в строке вывода следующим образом:

%a сокращенный день недели согласно текущему стандарту.

%A полный день недели согласно текущему стандарту.

%b сокращенный месяц согласно текущему стандарту.

%B полное название месяца согласно текущему стандарту.

%c привилегированное представление даты и времени для текущего стандарта.

%d день месяца как десятичное число (от 01 до 31).

%H час как десятичное число, используя 24-часовые часы (от 00 до

23).
%I час как десятичное число, используя 12-часовые часы (от 01 до 12).
%j день года как десятичное число (от 001 до 366).
%m месяц как десятичное число (от 01 до 12).
%M минуты как десятичное число.
%p Или "am" или "pm", согласно данному значению времени; или соответствующие строки для текущего стандарта.
%S секунды как десятичное число.
%U число недель текущего года как десятичное число, начинающееся с первого воскресенья как первый день первой недели.
%W число недель текущего года как десятичное число, начинающееся с первого понедельника как первый день первой недели.
%w день недели как десятичное число, воскресенье - 0.
%x привилегированное представление даты для текущего стандарта, но без времени.
%X привилегированное представление времени для текущего стандарта, но без даты.
%y год как десятичное число, но без столетия (от 00 до 99).
%Y год как десятичное число, включая столетие.
%Z часовой пояс или имя или сокращение (пусто, если часовой пояс не может быть определен).
%% литеральный символ ` % '.
Параметр size может использоваться, чтобы определить максимальное число символов, которое будет сохранено в массиве s, включая пустой символ завершения. Если форматируемое время требует

больше чем size символов, лишние символы отбрасываются. Возвращаемое значение из strftime - число символов, помещенное в массив s, не включая пустой символ завершения. Если значение равняется размеру, это означает что массив s был слишком мал; Вы должны повторить обращение, обеспечивая больший массив.
Если s - пустой указатель, strftime не делает фактической записи чего-нибудь, но взамен возвращает число символов, которое она написала бы.
Для примера strftime, см. Раздел 17.2.7 [Пример Функции Времени].

17.2.5 Определение Часового пояса с TZ
В системе GNU, пользователь может определять часовой пояс посредством TZ переменной среды.
Для уточнения инфрмации относительно того, как устанавливать переменные среды, см. Раздел 22.2 [Переменные среды]. Функции для доступа к часовому поясу объявлены в " time.h ".
Значение TZ переменной может иметь один из трех форматов. Первый формат используется, когда не имеется никакого Смещения светового дня (или в летнее время) в зоне местного времени:
std offset
Std строка определяет имя часового пояса. Эта строка должна состоять из трех или большего количества символов и не должно содержать первым символом двоеточие, и цифры, запятые, плюс или минус внутри.
Смещение определяет значение которое нужно добавить к местному времени, чтобы получить значение Координированного Универсального времени. Она имеет синтаксис подобно [+ | -] hh [: mm [: ss]]. Она положительно, если зона местного времени - к западу от Главного меридиана и отрицательно, если она восточнее. Час должен быть от 0 до 24, а минуты и секунды от 0 до 59.
Например, вот, как мы определили бы Восточное Стандартное Время, но без любых Смещений Светового дня:
EST+5
Второй формат используется, когда имеется Смещение светового дня:
std offset dst [offset],start[/time],end[/time]
Начальные std и offset определяют стандартный часовой пояс, как описано выше. Dst строка и offset определяет имя и смещение для

соответствующего Смещения Дня этого часового пояса; если смещение опущено, это значения по умолчанию равно одному часу перед стандартным временем.
Остаточный член от спецификации описывает, когда смещение светового дня действует. Поле start - то, когда смещение светового

дня входит в силу, а поле end - то, когда изменение будет сделано обратно к стандартному времени. Следующие форматы распознаваемы для этих полей:

Jn определяет Юлианский день, с n между 1 и 365. 29 февраля никогда не рассчитывается, даже в високосные годы.

N определяет Юлианский день, с n между 0 и 365. 29 февраля рассчитан в високосные годы.

Mm.w.d определяет день d недели w месяца m. день d должен быть между 0 (воскресеньем) и 6. Неделя w должна быть между 1 и 5; неделя 1 - первая неделя, в которой есть день d , а неделя 5 определяет последний d день в месяце. Месяц m должен быть между 1 и 12.

Поля time определяют, когда, по местному времени происходит изменение к другому времени. Значение по умолчанию - 02:00:00.

Например, вот, как можно определить Восточный часовой пояс в Соединенных Штатах, включая соответствующее смещение светового дня и даты применимости. Нормальное смещение ПО ГРИНВИЧУ - 5 часов; так как это - к западу от главного меридиана, знак положителен. Летний период начинается в первое воскресенье апреля в 2:00am, и кончается в последнее воскресенье октября в 2:00am.

EST+5EDT,M4.1.0/M10.5.0

План смещения светового дня в любой юрисдикции не изменяется годами.

Чтобы быть строго правильным, преобразование дат и времени должно быть основано на действующем плане. Однако, система не имеет никаких средств, чтобы допустить Вам определять, как план изменился. Наибольшее что Вы может сделать - определить один специфический план обычно план текущего дня.

Третий формат походит на:

:characters

Каждая операционная система интерпретирует этот формат по-разному; в библиотеке GNU C, characters - имя файла, который описывает часовой пояс.

Если переменная среды TZ не имеет значения, операция выбирает часовой пояс по умолчанию. Каждая операционная система имеет собственные правила для выбора заданного по умолчанию часового пояса, так что относительно этого мы можем сказать совсем немного.

17.2.6 Функции и Переменные для Часовых поясов

char * tzname [2] (переменная)

Массив tzname содержит две строки, которые являются стандартными трех-символьными именами пары часовых поясов (стандартный и смещения светового дня) которые пользователь выбрал. Tzname [0] - имя стандартного часового пояса (например, "EST"), а tzname [1] - имя для часового пояса, когда смещение светового дня находится в использовании (например, "EDT"). Они соответствуют к std и dst строкам (соответственно) из TZ переменной среды.

Tzname массив инициализируется из переменной среды TZ всякий раз, когда tzset, ctime, strftime, mktime, или localtime вызывается.

void tzset (void)

Tzset функция инициализирует переменную tzname из значения переменной среды TZ. Обычно вашей программе не нужно вызывать эту функцию, потому что она вызывается автоматически, когда Вы используете другие функции преобразования времени, которые зависят от часового пояса.

Следующие переменные определены для совместимости с System V Unix. Эти переменные устанавливаются вызоввом localtime.

long int timezone

Эта переменная содержит различие между временем ПО ГРИНВИЧУ и местным стандартным временем, в секундах. Например, в США в Восточном часовом поясе значение - 5*60*60.

int daylight

Эта переменная имеет значение отличное от нуля, если применяются стандартные американские правила смещения светового дня.

17.2.7 Пример Функции Времени

Вот пример программы, показывающий использование некоторых функций местного и календарного времени.

```
#include
#include
#define SIZE 256
int
main (void)
{
        char buffer[SIZE];
        time_t curtime;
        struct tm *loctime;
        curtime = time (NULL);
        loctime = localtime (&curtime);
        fputs (asctime (loctime), stdout);
        strftime (buffer, SIZE,
  "Today is %A, %B %d.\n", loctime);
        fputs (buffer, stdout);
        strftime (buffer, SIZE,
  "The time is %I:%M %p.\n", loctime);
        fputs (buffer, stdout);
        return 0;
}
```

Она производит примерно такой вывод:

```
Wed Jul 31 13:02:36 1991
Today is Wednesday, July  31.
The time is 01:02 PM.
```

17.3 Установка Сигнализаций

Функции alarm и setitimer обеспечивают механизм прерывания процесса, в некоторое время. Они делают это, устанавливая таймер; когда время таймер истекает, процесс получает сигнал.

Каждый процесс имеет три доступных независимых таймера интервала:

* Таймер в реальном времени, который считает время как часы. Этот таймер посылает сигнал SIGALRM процессу, когда время истекает.
* Виртуальный таймер, который считает процессорное время, используемое процессом. Этот таймер посылает сигнал SIGVTALRM процессу, когда время истекает.
* Таймер профилирования, который считает оба: процессорное время, используемое процессом, и процессорное время, потраченное в системных вызовах от имени процесса. Этот таймер посылает сигнал SIGPROF процессу, когда время истекает.

Вы можете иметь только один таймер каждого вида в любое заданное время. Если Вы устанавливаете таймер, который еще не истек, этот таймер будет сброшен в новое значение.

Вы должны установить обработчик для соответствующего сигнала alarm, используюя signal или sigaction перед обращением к setitimer или alarm. Иначе, необычная цепочка событий может заставлить таймер исчерпать время прежде, чем ваша программа установит обработчик, и в этом случае она будет завершена, так как это - заданное по умолчанию действие для сигналов alarm. См. Главу 21 [Обработка Сигнала].

Функция setitimer - первичный способ для установки будильника. Это средство объявлено в заглавном файле " sys/time.h ". Функция alarm, объявленная в " unistd.h ", обеспечивает несколько более простой интерфейс для установки таймера в реальном времени.

struct itimerval (тип данных)

Эта структура используется, чтобы определить, когда таймер должен истечь. Она содержит следующие элементы:

struct timeval it_interval

Это - интервал между последовательными прерываниями по таймеру. Значение - нуль, если сигнал будет только послан один раз.

struct timeval it_value

Это - интервал до первого прерывания по таймеру. Значение - нуль если, он заблокирован.

Тип данных Struct timeval описан в Разделе 17.2.2 [Календарь с высоким разрешением].

int setitimer (int which, struct itimerval *old, struct itimerval *new)

Функция setitimer устанавливает таймер, заданный как which согласно new. Аргумент which может иметь значение ITIMER_REAL, ITIMER_VIRTUAL, или ITIMER_PROF.

Если old - не пустой указатель, setitimer возвращает информацию относительно любого предыдущего неистекшего таймера того же самого вида в структуре, на которую он указывает.

Возвращаемое значение - 0 при успехе и -1 при отказе. Следующие errno условия ошибки определены для этой функции:

EINVAL интервал таймера был слишком большой.

int getitimer (int which, struct itimerval *old)

Getitimer функция сохраняет информацию относительно таймера, заданного which в структуре, указанной в old.

Возвращаемое значение и условия ошибки - такие же как для setitimer.

ITIMER_REAL

Эта константа может использоваться как аргумент which для setitimer и getitimer функций, чтобы определить таймер в реальном времени.

ITIMER_VIRTUAL

Эта константа может использоваться как аргумент which для setitimer и getitimer, чтобы определить виртуальный таймер.

ITIMER_PROF

Эта константа может использоваться как аргумент which для setitimer и getitimer, чтобы определить таймер профилирования.

unsigned int alarm (unsigned int seconds)

Функция alarm устанавливает таймер в реальном времени, с периодом в second секунд. Если Вы хотите отменить любой существующий таймер, Вы может сделать это, вызывая alarm с аргументом 0.

Возвращаемое значение указывает, сколько секунд оставалось прежде, чем предыдущий сигнал был бы послан. Если не было никакого предыдущего сигнала, alarm возвращает нуль.

Функция alarm могла бы быть определена в терминах setitimer примерно так:

```
unsigned int
alarm (unsigned int seconds)
{
        struct itimerval old, new;
        new.it_interval.tv_usec = 0;
        new.it_interval.tv_sec = 0;
        new.it_value.tv_usec = 0;
        new.it_value.tv_sec = (long int) seconds;
        if (setitimer (ITIMER_REAL, &new, &old) < 0)
                return 0;
        else
                return old.it_value.tv_sec;
}
```

Имеется пример, показывающий использование функции alarm в Разделе 21.4.1 [Возврат Обработчика].

Если Вы просто хотите, чтобы ваш процесс ждал данное число секунд, Вы должен использовать функцию sleep. См. Раздел 17.4 [Sleep].

Вы не должны рассчитать на сигнал, прибывающий точно, когда таймер истекает. В многопроцессорной среде имеется обычно некоторая задержка.

Примечание Переносимости: setitimer и getitimer - функции UNIX

BSD, в то время как функция alarm определена POSIX.1 стандартом. Setitimer более мощная чем alarm, но alarm более широко используется.

17.4 Sleep

Sleep дает простой способ заставить программу ждать некоторый период времени. Если ваша программа не использует сигналы (за исключением завершения), то Вы можете расчитывать, что sleep будет ждать заданное количество времени. Иначе, sleep может возвращаться, если прибывает сигнал; если Вы хотите ждать данный период независимо от сигналов, используйте select (см. Раздел 8.6 [Ждущий ввод - вывод]) и не определяйте ни каких описателей ожидания.

unsigned int sleep (unsigned int seconds)

Функция sleep ждет seconds секунд или пока не получен сигнал.

Если функция sleep возвращает значение по истечении времени, то это значение нуль. Если она возвращается после сигнала, возвращаемое значение - остающееся время ожидания sleep.

Функция sleep объявлена в " unistd.h ".

Вы можете использовать select и делать период ожидания, совершенно точным. (Конечно, загрузка системы может вызывать неизбежные дополнительные задержки, если машина не специализирована одному приложению, не имеется никакого способа, которым Вы можете избежать этого.)

17.5 Использование Ресурсов

Функция getrusage и тип данных struct rusage используется для исследования типа использования процесса. Они объявлены в " sys/resource.h ".

int getrusage (int processes, struct rusage *rusage)

Эта функция сообщает общее использование для процессов, заданных в processes, сохраняя информацию в *rusage.

В большинстве систем, processes имеет только два допустимых значения:

RUSAGE_SELF

Только текущий процесс.

RUSAGE_CHILDREN

Все дочерние процессы (прямые и косвенные) которые уже завершились.

В системе GNU, Вы можете также запрашивать относительно специфического дочернего процесса, определяя ID процесса.

Возвращаемое значение getrusage - нуль при успехе, и -1 при отказе.

Аргумент EINVAL processes не допустим.

Еще один способ получения типа использования для специфического дочернего процесса - функцией wait4, которая возвращает общие количества для дочернего процесса при его завершении. См. Раздел 23.8 [BSD Функции Ожидания].

struct rusage

Этот тип данных записывает величину использования различного рода ресурсов. Он имеет следующие элементы (возможны другие):

struct timeval ru_utime

Использованное пользовательское время.

struct timeval ru_stime

Использованное системное время.

long ru_majflt

Число страниц.

long ru_inblock

Число блокировок операций ввода.

long ru_oublock

Число блокировок операций вывода.

long ru_msgsnd

Число посланных сообщений.

long ru_msgrcv

Число полученных сообщений.

long ru_nsignals
Число полученных сигналов.
Дополнительная историческая функция для исследования типов использования, vtimes, обеспечивается но здесь не описана. Она объявлена в " sys/vtimes.h ".

17.6 Ограничение Использования Ресурсов
Вы можете определять ограничения использования ресурса для процесса. Когда процесс пробует превышать ограничение, он может терпеть неудачу, в зависимости от ограничения. Каждый процесс первоначально наследует значения ограничений от родителя, но он может впоследствии изменять их.
Символы в этом разделе определены в " sys/resource.h ".
int getrlimit (int resource, struct rlimit *rlp) (функция)
Читает текущее значение и максимальное значение ресурса resource, и сохраняет их в *rlp.
Возвращаемое значение - 0 при успехе и -1 при отказе. Единственое возможное errno условие ошибки - EFAULT.
int setrlimit (int resource, struct rlimit *rlp) (функция)
Сохраняет текущее значение и максимальное значение ресурса в

*rlp.
Возвращаемое значение - 0 при успехе и -1 при отказе. Следующее errno условие ошибки возможно:
EPERM Вы пробовали изменять максимально допустимое значение ограничения, но Вы не имеете привилегий, чтобы сделать это.
struct rlimit (тип данных)
Эта структура используется с getrlimit, чтобы получить значения ограничений, и с setrlimit, чтобы определить значения ограничений. Она имеет два поля:
Rlim_cur Текущее значение рассматриваемого ограничения.
Rlim_max Максимально допустимое значение рассматриваемого ограничения. Вы не можете устанавливать текущее значение ограничения больше чем этот максимум. Только root может изменять максимально допустимое значение.
В getrlimit, эта структура - вывод; она получает текущие значения. В setrlimit она определяет новые значения.
Вот список ресурсов, для которых Вы можете определять ограничения.
RLIMIT_CPU
Максимальное количество времени центрального процессора которое процесс может использовать. Если он выполняется дольше чем это время, он получает сигнал: SIGXCPU. Значение измеряется в секундах. См. Раздел 21.2.7 [Нестандартные Сигналы].
RLIMIT_FSIZE
Максимальный размер файла котоый процесс может создать. При попытке записать больший файл вызывается сигнал: SIGXFSZ. См. Раздел 21.2.7 [Нестандартные Сигналы].
RLIMIT_DATA
Максимальный размер памяти данных для процесса. Если процесс пробует зарезервировать память больше этого количества, функции резервирования выдает ошибку.
RLIMIT_STACK
Максимальный размер стека для процесса. Если процесс пробует расширять стек больше этого размера, он получает сигнал SIGSEGV. См. Раздел 21.2.1 [Сигналы Ошибки в программе].
RLIMIT_CORE
Максимальный размер core-файла, который этот процесс может создавать. Если процесс завершается и этот максимальный размер не

достаточен для core-файла, файл будет усечен.
RLIMIT_RSS
Максимальное количество физической памяти, которое этот процесс может получить. Этот параметр - руководство для планировщика системы и программы распределения памяти; система может давать процессу большее количество памяти, когда имеется излишек.
RLIMIT_OPEN_FILES
Максимальное число файлов, которые процесс может открывать. Если он пробует открывать большее количество файлов, он получает код ошибки EMFILE. См. Раздел 2.2 [Коды Ошибки].
RLIM_NLIMITS

Число различных ограничений ресурсов. Любой допустимый операнд ресурса должен быть меньше чем RLIM_NLIMITS.

int RLIM_INFINITY

Эта константа замещает значение "бесконечности" когда обеспечивается как значение ограничения в setrlimit.

Две исторических функции для установки ограничений ресурса, ulimit и vlimit, не зарегистрированы здесь. Они объявлены в " sys/vlimit.h " и исходят ИЗ BSD.

17.7 Приоритет Процесса

Когда отдельные процессы выполняются, их приоритеты определяют то, какую часть ресурсов CPU каждый процесс получает. Этот раздел описывает, как Вы можете читать и устанавливать приоритет процесса. Все эти функции и макркоманды объявлены в " sys/resource.h ".

Промежуток допустимых значений зависит от операционной системы, но обычно он выполняется от -20 до 20. Меньшее значение приоритета означает что процесс выполняет чаще. Эти константы описывают некоторые значения:

PRIO_MIN самое маленькое допустимое значение приоритета.

PRIO_MAX самое большое допустимое значение приоритета.

int getpriority (int class, int id) (функция)

Читает приоритет класса процессов, задаваемого class и id (см. ниже).

Возвращаемое значение - значение приоритета при успехе, и -1 при отказе. Следующее errno условие ошибки возможно для этой функции:

ESRCH комбинация class и id не соответствует никакому

существующему процессу.

EINVAL значение class не допустимо.

Когда возвращаемое значение -1, это может указывать отказ, или значение приоритета.

Единственый способ различить состоит в том, чтобы установить errno = 0 перед вызовом getpriority, и тогда использовать errno!= 0 позже как критерий для отказа.

int setpriority (int class, int id, int priority) (функция)

Устанавливает приоритет класса процессов, задаваемого class и id (см. ниже).

Возвращаемое значение - 0 при успехе и -1 при отказе. Следующее errno условие ошибки определено для этой функции:

ESRCH комбинация class и id не соответствует никакому существующему процессу.

EINVAL значение класса не допустимо.

EPERM Вы пробовали устанавливать приоритет процесса некоторого другого пользователя, и Вы не имеете привилегий для этого.

EACCES Вы пробовали понизить приоритет процесса, и Вы не имеете привилегий для этого.

Аргументы class и id вместе определяет набор процессов, которыми Вы заинтересованы. Вот возможные значения для class:

PRIO_PROCESS

Читает или устанавливает приоритет одного процесса. id аргумент - ID процесс.

PRIO_PGRP

Читает или устанавливает приоритет одной группы процесса. Аргумент id - ID группы процесса.

PRIO_USER

Читает или устанавливает приоритет процессов одного пользователя. Аргумента id - ID пользователя.

Если id аргумент - 0, то обрабатывается текущий процесс, текущая группа процесса, или текущий пользователь, согласно классу.

int nice (int increment) (функция)

Увеличьте приоритет текущего процесса приращением. Возвращаемое значение не важно.

Вот эквивалентное определение для nice:

```
int
nice (int increment)
```

```
{
        int old = getpriority (PRIO_PROCESS, 0);
        setpriority (PRIO_PROCESS,0,old+increment);
}
```

18. Расширение Символов

Ряд языков использует наборы символов, которые больше чем набор значений типа char.
Японский и Китайский - возможно наиболее близкие примеры.
Библиотека GNU C включает поддержку двух механизмов для работы с расширенными наборами символов: многобайтовые символы и широкие символы. Эта глава описывает, как использовать эти механизмы, и функции для преобразования между ними.
На поведение функций в этой главе воздействует текущий стандарт для символьной классификации LC_CTYPE класса; см. Раздел 19.3 [Категории Стандарта]. Этот стандарт указывает, который многобайтовый код используется, и также управляет значениями и характеристиками расширенных символьных кодов.

18.1 Введение в Расширение Символов
Вы можете представлять расширенные символы одним из двух способов:
* Как многобайтовые символы, которые могут быть внедрены в обычной строке, т. е. в массиве объектов char. Их преимущество - то, что много программ и операционных систем могут обрабатывать случайные многобайтовые символы, рассеянные среди обычных символов ASCII, без любого изменения.
* Как расширенные символы, которые подобны обычным символам за исключением того, что они занимают большое количество битов. Широкий символьный тип данных wchar_t, имеет достаточно места, чтобы содержать расширенные символьные коды также как традиционные коды ASCII.

Преимущество расширенных символов - то, что каждый символ является одиночным объектом данных, точно так же как обычные символы ASCII. Имеются несколько недостатков:
* Каждая существующая программа должна быть измененна и перетранслирована, чтобы использовать расширенные символы.
* Файлы расширенных символов не могут быть прочитаны программами, которые ожидают обычные символы.
Обычно, Вы используете многобайтовое символьное представление как часть внешнего интерфейса программы, типа чтения или запииси в файлы. Однако, обычно проще выполнить внутренние манипулирования на строках, содержащих расширенные символы, в массивах объектов wchar_t, так как однородное представление делает операции редактирования намного проще. Если Вы используете многобайтовые символы для файлов и расширенные символы для внутренних операций, Вы должен преобразовать их при записи и чтении данных.
Если ваша система поддерживает расширение символов, то она поддерживает их, и как многобайтовые символы и как расширенные символы. Библиотека включает функции, которые Вы можете использовать, чтобы преобразовать между двумя представлениями. Эти функции описаны в этой главе.

18.2 Стандарты и Расширенные Символы
Компьютерная система может поддерживать больше чем один многобайтовый символьный код, и больше чем один расширенный символьный код. Пользователь управляет выбором кодов через текущий стандарт для символьной классификации (см. Главу 19 [Стандарты]). Каждый стандарт определяет специфический многобайтовый символьный код и специфический расширенный символьный код. Выбор стандарта влияет на поведение функций преобразования в библиотеке.
Некоторые стандарты не поддерживают ни расширенные символы, ни нетривиальные многобайтовые символы. В этих стандартах, библиотечные функции преобразования все еще работают, даже если, что они в основном тривиальны.
Если Вы выбираете нновый стандарт для символьной классификации, внутренний параметр сдвига, поддерживаемый этими функциями может стать спутанным, так что - не стоит изменять стандарт в то время как, Вы находитесь в середине обработки строки.

18.3 Многобайтовые Символы

В обычном коде ASCII, последовательность символов - последовательность байтов, и каждый символ - один байт. Это очень просто, но учитывает только 256 различных символов.

В многобайтовом символьном коде, последовательность символов - последовательность байтов, но каждый символ может занимать одни или более последовательных байт последовательности.

Имеются много различных способов проектирования многобайтового символьного кода; различные системы используют различные коды. Специфический способ кодирования определяется обозначением базисных последовательностей байтов, которые представляют одиночный символ и какие символы они замещают. Код, который компьютер может фактически использовать, должен иметь конечное число этих базисных последовательностей, и обычно ни одна из них не длиннее чем несколько символов.

Эти последовательности не имеют одинаковую длину. Фактически, многие из их - только один байт. Потому что базисные символы ASCII в промежутке от 0 до 0177 настолько важны, что они замещают себя во всех многобайтовых символьных кодах. То есть байт, чье значение от 0 до 0177 - всегда символ сам по себе. Символы, которые больше чем один байт, должны всегда начинаться с байта в промежутке от 0200 до 0377.

Значение 0 байта может использоваться, чтобы завершить строку, точно как это часто используется в строке символов ASCII.

Определение базисных последовательностей байтов, которые представляют одиночные символы автоматически дают значения более длинным последовательностям байтов, больше чем один символ. Например, если последовательность два байта 0205 049 замещает символ греческой альфы, то 0205 049 065 должна заместить альфу, сопровождаемую " " (код ASCII 065), а 0205 049 0205 049 должна заместить две альфы в строке.

Если любая последовательность байтов может иметь больше чем одно значение как последовательность символов, то многобайтовый код неоднозначен и опасен. Коды, которые системы фактически используют все однозначны.

В большинстве кодов, имеются некоторые последовательности байтов, которые не имеют никакого значения как символ или символы.

Они называются недопустимыми.

Самый простой возможный многобайтовый код - тривиальный:

Базисные последовательности состоят из одиночных байтов.

Данный специфический код не использует многобайтовые символы вообще. Он не имеет никаких недопустимых последовательностей. Но он может обрабатывать только 256 различных символов.

Вот другой возможный код, который может обрабатывать 9376 различных символов:

Базисные последовательности состоят из

* одиночных байтов со значениями в промежутке от 0 до 0237.
* двух-байтовых последовательностей, в которых оба байта имеют значения в промежутке от 0240 до 0377.

Этот код или подобный используется на некоторых системах, чтобы представить Японские символы. Недопустимые последовательности - те, которые состоят из нечетного числа последовательных байтов в промежутке от 0240 до 0377.

Вот другой многобайтовый код, который может обрабатывать больше различных расширенных символов, фактически, почти тридцать миллионов:

Базисные последовательности состоят из

* одиночных байтов со значениями в промежутке 0 до 0177.
* последовательностей до четырех байтов, в которых первый байт находится в промежутке от 0200 до 0237, а остающиеся байты находятся в промежутке от 0240 до 0377.

В этом коде, любая последовательность, которая начинается с байта в промежутке от 0240 до 0377, недопустима.

Вот другой вариант, который имеет преимущество: при удалении последнего байта или байтов допустимого символа, никогда не может появиться другой допустимый символ.

Базисные последовательности состоят из

* одиночных байтов со значениями в промежутке от 0 до 0177.

* двух-байтовых последовательностей, в которых первый байт находится в промежутке от 0200 до 0207, а второй байт находятся в промежутке от 0240 до 0377.
* трех-байтовых последовательностей, в которых первый байт находится в промежутке от 0210 до 0217, а другие байты находятся в промежутке от 0240 до 0377.
* четырех-байтовых последовательностей, в которых первый байт

находится в промежутке от 0220 до 0227, а другие байтаы находятся в промежутке от 0240 до 0377.

Список недопустимых последовательностей для этого кода - довольно длинный и не стоит полного просмотра; примеры недопустимых последовательностей включают 0240 и 0220 0300 065.

Число возможных многобайтовых кодов очень велико. Но данная компьютерная система будет поддерживать не больше нескольких различных кодов. (Один из этих кодов может учитывать тысячи различных символов.) Другая компьютерная система может поддерживать полностью отличный код. Библиотечные средства, описанные в этой главе полезны, потому что они описывают подробности многобайтового кода специфической компьютерной системы, хотя ваши программы не должны знать.

Вы можете использовать специальные стандартные макркоманды, чтобы выяснить максимальное возможное число байтов символа в текущем многобайтовом коде ипользуйте MB_CUR_MAX, а максимум для любого многобайтового кода, обеспечиваемого на вашем компьютере содержится в MB_LEN_MAX.

int MB_LEN_MAX (макрос)

Это - максимальная длина многобайтового символа для любого обеспечиваемого стандарта. Она определена в " limits.h ".

int MB_CUR_MAX (макрос)

Эта макркоманда отображает (возможно не-константу) положительное целочисленное выражение, которое является максимальным числом байтов в многобайтовом символе в текущем стандарте. Значение никогда не большее чем MB_LEN_MAX.

MB_CUR_MAX определен в " stdlib.h ".

Что случается, если Вы пробуете передавать строку, содержащую многобайтовые символы функции, которая не знает о них? Обычно, такая функция обрабатывает строку как последовательность байтов, и интерпретирует некоторые значения особенно; все другие значения байтов "обычны". Если многобайтовый символ не содержит специальное значение байта, функция должна обработать его, как будто это были отдельные обычные символы.

18.4 Введение в Расширенные Символы

Расширенные символы намного проще чем многобайтовые символы. Они - просто символы с больше, чем восемью битами, так, чтобы они имели место для больше, чем 256 различных кодов. Расширенный символьный тип данных wchar_t, имеет достаточно большой диапазон, чтобы содержать расширенные символьные коды также как традиционные коды ASCII.

Преимущество расширенных символов - в том, что каждый символ является одиночным объектом данных, точно так же как обычные символы ASCII. Эти символы также имеют некоторые недостатки:

* Каждая существующая программа должна быть изменена и перетранслирована, чтобы использовать расширенные символы.
* Файлы расширенных символов не могут быть прочитаны программами, которые ожидают обычные символы.

Расширенные символьные значения от 0 до 0177 всегда идентичны кодам символов ASCII. Расширенный нуль часто используется, чтобы завершить строку расширенных символов, точно как, одиночный нулевой байт часто завершает строку обычных символов.

wchar_t (тип данных)

Это - тип " расширенных символов " , целочисленный тип, чей диапазон достаточно велик, чтобы представить все различные значения в любом расширенном наборе символов в обеспечиваемых стандартах. См. Главу 19 [Стандарты], для получения более подробной

информации. Этот тип определен в заглавном файле " stddef.h ".
Если ваша система поддерживает расширенные символы, то каждый расширенный символ имеет и расширенный символьный код и соответствующую многобайтовую базисную последовательность.
В этой главе, термин код используется, чтобы обратиться к одиночному расширенному символьному объекту, чтобы подчеркнуть различие от типа данных char.

18.5 Преобразование Расширенных Строк

Функция mbstowcs преобразовывает строку многобайтовых символов в массив расширенных символов. Функция wcstombs делает обратный процесс. Эти функции объявлены в заглавном файле " stdlib.h ".
В большинстве программ, эти функции - единственная Ваша потребность в преобразовании между расширенными строками и

многобайтовыми символьными строками. Но они имеют ограничения. Если ваши данные не с нулевым символом в конце или - не все в ядре сразу, Вы возможно должны использовать функции преобразования низкого уровня, чтобы преобразовать один символ за раз. См. Раздел 18.7 [Преобразование Одного Символа].

size_t mbstowcs (wchar_t *wstring, const char *string, size_t size)

mbstowcs (" многобайтовая строка в строку расширенных символов ") функция преобразовывает строку с нулевым символом в конце многобайтовых символов в массив расширенных символов, сохраняя не больше чем size расширенных символов в массиве, начинающемся в wstring. Пустой символ завершения рассчитывается в size, так что, если размер меньше, чем фактическое число расширенных символов, следующих из строки, никакой пустой символ завершения не будет сохранен.
Преобразование символов из строки начинается с начальным параметром регистра.
Если недопустимая многобайтовая символьная последовательность найдена, функция возвращает значение -1. Иначе, она возвращает число расширенных символов, сохраненных в массиве wstring. Это число не включает пустой символ завершения, который присутствует, если число меньше, чем size.
Вот пример, показывающий, как преобразовывать строку многобайтовых символов, резервируя достаточное пространство для результата.

```
wchar_t *
mbstowcs_alloc (const char *string)
{
        size_t size = strlen (string) + 1;
        wchar_t *buf = xmalloc (size * sizeof (wchar_t));
        size = mbstowcs (buf, string, size);
        if (size == (size_t) -1)
                return NULL;
        buf = xrealloc (buf, (size + 1) * sizeof (wchar_t));
        return buf;
}
```

size_t wcstombs (char *string,const wchar_t wstring,size_t size)

wcstombs преобразует массив расширенных символов с нулевым

символом в конце в строку, содержащую многобайтовые символы, сохраняя не больше чем size байт, начиная с string.
Если найден код, который не соответствует допустимому многобайтовому символу, то эта функция возвращает значение -1. Иначе, возвращаемое значение - число байтов, сохраненных в массиве. Это число не включает пустой символ завершения, который присутствует, если число - меньше чем size.

18.6 Длина Многобайтового Символа

Этот раздел описывает, как просмотреть строку, содержащую многобайтовые символы, по одному символу. Трудность в том, что нужно знать, сколько байтов каждый символ содержат. Ваша программа может использовать mblen, чтобы узнать это.

int mblen (const char *string, size_t size)

Функция mblen с непустым аргументом string возвращает число байтов, которые составляют многобайтовый символ, начинающийся с string, никогда не исследуя больше size байт. (Идея состоит в том,

чтобы обеспечить для size число байтов данных, которые Вы имеете.)

Возвращаемое значение mblen отличает три возможности: первые size байт строки начинаются с допустимого многобайтового символа, они начинаются с недопустимой последовательности байтов или включают только часть символа, или string указывает на пустую строку (пустой символ).

Для допустимого многобайтового символа, mblen возвращает число байтов в этом символе (всегда по крайней мере 1, и никогда не больше чем size). Для недопустимой последовательности байтов, mblen возвращает -1. Для пустой строки, она возвращает 0.

Если многобайтовый символьный код использует символы смены регистра, то mblen, поддерживает и модифицирует параметр регистра. Если Вы вызываете mblen с пустым указателем для строки, она инициализирует параметр регистра к стандартному начальному значению. См. Раздел 18.9 [Параметра Регистра] .

Функция mblen объявлена в " stdlib.h ".

18.7 Преобразование Расширенных Символов по Одному

Вы можете преобразовывать многобайтовые символы в расширенные символы по одному mbtowc функцией.

Wctomb функция делает обратное. Эти функции объявлены в " stdlib.h ".

int mbtowc (wchar_t *result, const char *string, size_t size)

Mbtowc преобразовывает первый многобайтовый символ в string в соответствующий расширенный символьный код. Она сохраняет результат в *result.

Mbtowc никогда не исследует больше чем size байт. (Идея состоит в том, чтобы обеспечить для size число байтов данных, которое Вы имеете.)

Mbtowc с непустой строкой Возвращаемое значение mblen отличает три возможности: первые size байт строки начинаются с допустимого многобайтового символа, они начинаются с недопустимой последовательности байтов или включают только часть символа, или string указывает на пустую строку (пустой символ).

Для допустимого многобайтового символа, mbtowc преобразовывает его в расширенный символ, сохраняет его в *result, и возвращает число байтов в том символе (всегда по крайней мере 1, и никогда не болольше чем size).

Для недопустимой последовательности байтов, mbtowc возвращает -1. Для пустой строки, она возвращает 0, также сохраняя 0 в *result.

int wctomb (char *string, wchar_t wchar)

Функция wctomb преобразует расширенный символьный код wchar в соответствующую многобайтовую символьную последовательность, и сохраняет результат в байтах, начиная с string.

Wctomb с непустой строкой отличает три возможности для wchar: допустимый расширенный символьный код (тот, который может транслироваться в многобайтовый символ), недопустимый код, и 0.

Если wchar - недопустимый расширенный символьный код, wctomb возвращает -1. Если wchar - 0, она возвращает 0, также сохраняя 0 в *string.

Вызов этой функции с нулевым wchar аргументом, когда строка - не пустой символ, имеет побочный эффект переинициализации сохраненного параметра регистра также как сохранения многобайтового символа 0 и возвращения 0.

18.8 Пример Посимвольного Преобразования

Вот пример, который читает многобайтовоый-символьный текст из дескриптора input и записывает соответствующие расширенные символы в описатель output. Мы должны преобразовать символы один за другим в этом примере, потому что mbstowcs неспособна продолжиться после пустого символа, и не может справляться с очевидно недопустимым частичным символом, читая большое количество ввода.

```
int
file_mbstowcs (int input, int output)
```

```
{
        char buffer[BUFSIZ + MB_LEN_MAX];
        int filled = 0;
        int eof = 0;
        while (!eof)
        {
                int nread;
                int nwrite;
                char *inp = buffer;
                wchar_t outbuf[BUFSIZ];
                wchar_t *outp = outbuf;
                nread = read (input, buffer +
        filled, BUFSIZ);
                if (nread < 0)
                {
                        perror ("read");
                        return 0;
                }
                if (nread == 0)
                        eof = 1;
                filled += nread;
                while (1)
                {
                        int thislen = mbtowc (outp,
                        inp, filled);
                        if (thislen == -1)
                                break;
                        if (thislen == 0) {
```



```
                                thislen = 1;
                                mbtowc (NULL, NULL, 0);
                        }
                        inp += thislen;
                        filled -= thislen;
                        outp++;
                }
                nwrite = write (output, outbuf,
                        (outp-outbuf)*sizeof(wchar_t));
                if (nwrite < 0)
                {
                        perror ("write");
                        return 0;
                }
                if ((eof && filled > 0) ||
        filled >= MB_CUR_MAX)
                {
                        error ("invalid
        multibyte character");
                        return 0;
                }
                if (filled > 0)
                        memcpy (inp, buffer, filled);
        }
        return 1;
}
```

18.9 Многобайтовые Коды, использующие Последовательности Регистров

В некоторых многобайтовых символьных кодах, значение любой специфической последовательности байтов не фиксировано; оно зависит от других последовательностей рассмотренных ранее в той же самой строке. Обычно имеются только несколько последовательностей, которые могут изменять значение других последовательностей; эти немногие называются последовательностями регистров, и мы говорим, что они устанавливают параметр регистра для других последовательностей, которые последуют.

Чтобы проиллюстрировать состояние регистра и последовательности

регистров, предположите, что мы устанавливаем, что последовательность 0200 (только один байт) вводит Японский режим, в котором пары байтов в промежутке от 0240 до 0377 являются

одиночными символами, в то время как 0201 вводит Латинский-1 режим, в котором одиночные байты в промежутке от 0240 до 0377 являются символами, и интерпретируются согласно набору символов Latin-1 Международной организации по стандартизации. Это - многобайтовый код, который имеет два альтернативных состояния регистра ("Японский режим " и " Латинский-1 режим "), и две последовательности регистров, которые определяют специфические состояния регистра.

Когда используемый многобайтовый символьный код имеет состояния регистра, то mblen, mbtowc и wctomb должны поддерживать и модифицировать текущее состояние регистра, поскольку они просматривают строку. Чтобы делать эту работу правильно, Вы должны следовать этим правилам:

* Перед стартом просмотра строки, вызовите функцию с пустым указателем для многобайтового символьного адреса например, mblen (NULL, 0). Это инициализирует состояние регистра к стандартному начальному значению.

* Просматривайте строку по одному символу. Не "возвращайтесь" и не перепросмотривайте уже просмотренные символы, и не смешивайте обработку различных строк.

Вот пример использования mblen с соблюдением этих правил:

```
void
scan_string (char *s)
{
        int length = strlen (s);
        mblen (NULL, 0);
        while (1)
        {
                int thischar = mblen (s, length);
                if (thischar == 0)
                        break;
                if (thischar == -1)
                {
                        error ("invalid multibyte
                                character");
                                                break;

              - 416 -

                }
                s += thischar;
                length -= thischar;
        }
}
```

Функции mblen, mbtowc и wctomb не используются при использовании многобайтового кода, который использует состояние регистра. Однако, никакие другие библиотечные функции не вызывают эти функции, так что Вы не должны волноваться относительно этого.

19. Национальные и Международные Стандарты

Различные страны имеют изменяющиеся стандарты. Эти стандарты содержит преобразования от очень простых, типа формата для представления дат и времени, до очень сложных, типа разговорного языка.

Межнационализация программного обеспечения означает способность к адаптированию программирования для соглашений предпочитаемых пользователем. В ANSI C, межнационализация работает посредством стандартов. Каждый стандарт определяет набор соглашений, одно соглашение для каждой цели. Пользователь выбирает набор соглашений, определяя стандарт (через переменные среды).

Все программы наследуют выбранный стандарт как часть их среды.

19.1 Какие Эффекты Стандарта Имеет Каждый Стандарт ?

Это определяет соглашения для отдельных целей, включая следующие:

* Какие многобайтовые символьные последовательности являются допустимыми, и как они интерпретируются (см. Главу 18 [Расширенные Символы]).

* Классификация того, какие символы местного набора являются буквенными, верхнего и нижнего регистра (см. Главу 4 [Обработка Символов]).

* Последовательность объединений для местного языка и набора символов (см. Раздел 5.6 [Функции Объединения]).

* Форматирование чисел (см. Раздел 19.6 [Числовое

Форматирование]).

* Форматирование дат и времени (см. Раздел 17.2.4 [Форматирование Даты и времени]).
* Какой язык использовать для вывода, включая сообщения об ошибках. (Библиотека С однако не позволяет Вам выполнить это.)
* Какой язык использовать для ответов пользователя на вопросы типа "да" или "нет".
* Какой язык использовать для более сложного ввода пользователя.

Некоторые аспекты адаптации к заданному стандарту обрабатываются автоматически библиотечными подпрограммами. Например, все ваши программы должны использовать последовательность объединений выбранного стандарта, т. е. использовать strcoll или strxfrm, чтобы сравнить строки.

Другие аспекты стандартов - вне компетенции библиотеки. Например, библиотека не может автоматически транслировать сообщения вывода вашей программы на другие языки. Единственый способ, которым Вы можете поддерживать вывод на языке пользователя, должен программироваться более или менее вручную. (В конечном счете, мы надеемся обеспечить средства, чтобы делать этот проще.)

Эта глава обсуждает механизм, которым Вы можете изменять текущий стандарт. Эффекты текущего стандарта на специфических библиотечных функциях обсуждены более подробно в описаниях самих функций.

19.2 Выбор Стандарта

Самый простой способ для пользователя, чтобы выбрать стандарт состоит в том, чтобы установить переменную среды LANG. Она определяет один стандарт для всех целей. Например, пользователь мог бы определить гипотетический стандарт, именованный " espana-castellano " чтобы использовать стандартные соглашения Испании.

Набор обеспечиваемых стандартов зависит от операционной системы, которую Вы используете. Мы не можем делать ни каких обещаний относительно того, что стандарты будут существовать, кроме одного стандартного стандарта, называемого " C " или " POSIX ".

Пользователь также имеет опцию определения различных стандартов для различных целей, выбирая смесь многих стандартов.

19.3 Категории Действий, на которые Воздействуют Стандарты

Цели, которые стандарты обслуживают, сгруппированы в категории так, чтобы пользователь или программа мог выбирать стандарт для каждого класса независимо. Вот таблица категорий: каждое имя является и переменной среды, которую пользователь может устанавливать, и именем макркоманды, которую Вы можете использовать как аргумент setlocale.

LC_COLLATE

Этот класс применяется для объединения строк (функции strcoll и strxfrm); см. Раздел 5.6 [Функции Объединения].

LC_CTYPE

Этот класс применяется для классификации и преобразований символов, и в многобайтовые и в расширенные символы; см. Главу 4 [Обработка Символов] и Главу 18 [Расширенные Символы].

LC_MONETARY

Этот класс применяется к форматированию валютных значений; см. Раздел 19.6 [Числовое форматирование].

LC_NUMERIC

Этот класс применяется к форматированию числовых значений; см. раздел 19.6 [Числовое Форматирование].

LC_TIME

Этот класс применяется для форматирования значений даты и времени; см. Раздел 17.2.4 [Форматирование Даты и времени].

LC_ALL

Это - не переменная среды, это - только макркоманда, которую Вы можете использовать с setlocale, чтобы установить одиночный стандарт для всех целей.

LANG

Если эта переменная среды определена, значение определяет

стандарт, используемый для всех целей за исключением того, как отменено переменными выше.

19.4 Как Программы Устанавлиают Стандарт

Программа на Си наследует переменные среды стандарта, когда она начинается. Это случается автоматически. Однако, эти переменные автоматически не управляют стандартом, используемым в соответствии с библиотечными функциями, потому что ANSI C говорит, что все

программы начинаются по умолчанию в стандарте " C ". Чтобы использовать стандарты, заданные средой, Вы должны вызвать setlocale. Вызовите ее следующим образом:

```
setlocale (LC_ALL, "");
```

чтобы выбрать стандарт, основанный на соответствующих переменных среды.

Вы можете также использовать setlocale, чтобы определить специфический стандарт, для общего использования или для специфического класса.

Символы в этом разделе определены в заглавном файле " locale.h".

char * setlocale (int category, const char *locale)

Функция setlocale устанавливает текущий стандарт для указанного класса.

Если класс - LC_ALL, то она определяет стандарт для всех целей. Другие возможные значения класса определяют индивидуальную цель (см. Раздел 19.3 [Категории Стандарта]).

Вы можете также использовать эту функцию, чтобы выяснить текущий стандарт, передавая пустой указатель как аргумент стандарта. В этом случае, setlocale возвращает строку, которая является именем стандарта, в настоящее время выбранного для класса class.

Строка, возвращенная setlocale может быть записана поверх последующими обращениями, так что Вы должны делать копию строки (см. Раздел 5.4 [Копирование и Конкатенация]) если Вы хотите сохранять ее после любых дальнейших обращений к setlocale. (Стандартная библиотека, как гарантируют, никогда не вызовет setlocale непосредственно.)

Вы не должны изменять строку, возвращенную setlocale. Это может быть та же самая строка, которая была передана как аргумент в предыдущем обращении к setlocale.

Когда Вы читаете текущий стандарт для класса LC_ALL, значение кодирует всю комбинацию выбранных стандартов для всех категорий. В этом случае, значение не только одиночное имя стандарта. Фактически, мы не делаем ни каких обещаний относительно того, на что это походит. Но если Вы определяете то же самое " имя стандарта" LC_ALL в последующем обращении к setlocale, она восстанавливает ту же самую комбинацию выборов стандарта.

Когда аргумент locale - не пустой указатель, строка, возвращенная setlocale отражает изменяемый стандарт.

Если Вы определяете пустую строку для стандарта, это означает, что нужно читать соответствующую переменную среды и использовать ее значение для установки стандарта указанного класса.

Если Вы определяете недопустимое имя стандарта, setlocale, возвращает пустой указатель и оставляет текущий стандарт неизмененным.

Вот пример, показывающий, как Вы могли бы использовать setlocale, чтобы временно включить к новый стандарт.

```
#include
#include
#include
#include
void
with_other_locale (char *new_locale,
                   void (*subroutine) (int),
                   int argument)
{
        char *old_locale, *saved_locale;
        old_locale = setlocale (LC_ALL, NULL);
        saved_locale = strdup (old_locale);
        if (old_locale == NULL)
                fatal ("Out of memory");
        setlocale (LC_ALL, new_locale);
```

```
                        (*subroutine) (argument);
                        setlocale (LC_ALL, saved_locale);
                        free (saved_locale);
                }
```

Примечание о переносимости: Некоторые системы ANSI C могут определять дополнительные категории стандарта. Для переносимости, запомните, что любой символ, начинающийся с " LC_ " мог бы быть определен в "locale.h".

19.5 Стандартные Стандарты

Единственые имена стандартов, которые Вы можете находить во всех операционных системах - это три стандартных:

"C" Это - стандартный стандарт Си. Атрибуты и поведение, которое он обеспечивает определены в стандарте ANSI C. Когда ваша программа начинается, она первоначально использует этот стандарт по

умолчанию.

" POSIX " Это - стандартный стандарт POSIX. В настоящее время, это - побочный результат исследования для стандартного стандарта Си.

"" Пустое имя говорит о том, что нужно выбрать стандарт, основанный на переменных среды. См. Раздел 19.3 [Категории Стандарта].

Определение и установка стандартов - обязаность вашего администратора системы (или человека, который установил библиотеку GNU C). Некоторые системы могут позволять пользователям создавать стандарты, но мы не обсуждаем это здесь.

Если ваша программа должна использовать кое-что отличное от "C" стандарт, она будет более переносимой, если Вы используете стандарт, который пользователь определяет со средой, а не пробуя определить некоторый ненормативный стандарт явно именем. Помните, что различные машины могут иметь различные наборы устанавливаемых стандартов.

19.6 Числовое Форматирование

Когда Вы хотите форматировать число или количество валюты, используя соглашения текущего стандарта, Вы можете использовать функцию localeconv, чтобы получить данные относительно того, как делать это. Функция localeconv объявлена в заглавном файле "locale.h ".

struct lconv * localeconv

Функция localeconv возвращает указатель на структуру, чьи компоненты содержат информацию относительно того, как числовые и валютные значения должны форматироваться в текущем стандарте.

Вы не должны изменять структуру или ее содержимое. Структура может быть записана поверх последующими обращениями к localeconv, или обращениями к setlocale, но никакая другая функция в библиотеке не записывает поверх этого значения.

struct lconv

Это - тип данных значения, возвращенного localeconv.

Если элемент структуры struct lconv имеет тип char, и значение - CHAR_MAX, это означает что текущий стандарт не имеет никакого значения для этого параметра.

19.6.1 Обобщенные Параметры Числового Форматирования

Это - стандартные элементы struct lconv; хотя могут иметься и другие.

char *decimal_point

char *mon_decimal_point

Это - разделители Десятичных точек, используемые в форматировании невалютных и валютных чисел, соответственно. В "C" стандарте, значение decimal_point - ".", а значение mon_decimal_point - "".

char *thousands_sep

char *mon_thousands_sep

Это - разделители, используемые, чтобы разграничить группы цифр налево от десятичной точки при форматировании невалютных и валютных чисел, соответственно. В "C" стандарте, оба элемента имеет значение "" (пустая строка).

char *grouping
char *mon_grouping
Это - строки, которые определяют, как группировать цифры налево от десятичной точки. Группировка grouping применяется к невалютным количествам, а mon_grouping применяется к валютным числам. Используются или thousands_sep или mon_thousands_sep, чтобы отделить группы цифры.
Каждая строка состоит из десятичных чисел, отделяемых точками с запятой. Последовательные числа (слева направо) дают размеры последовательных групп (справа налево, начиная с десятичной точки). Последнее число в строке используется много раз для всех оставшихся групп.
Если последним целым числом является -1, это означает, что любые остающиеся цифры формируются в одну большую группу без разделителей.
Например, при - "4;3;2", правильная группировка для числа 123456787654321 будет " 12 ", " 34 ", " 56 ", " 78 ", " 765 ", " 4321 ". С разделителем ",", число было бы напечатано как " 12,34,56,78,765,4321 ".
Значение "3" указывает повторение группы из трех цифр, как обычно используется в США.
В стандарте " C ", и grouping и mon_grouping имеют значения "".

Это значение не определяет никакой группировки вообще.
char int_frac_digits
char frac_digits
Это - целые, указывающие, сколько дробных цифр (направо от десятичной точки) должны отобразиться в валютном значении в международных и местных форматах, соответственно. (Наиболее часто, оба элемента имеют то же самое значение.)
В стандарте " C ", оба этих элемента имеют значение CHAR_MAX, означая "неопределенный".
Мы рекомендуем не печатать никаких дробных цифр. (Этот стандарт также определяет пустую строку для mon_decimal_point, так печать любых дробных цифр только путала бы!)

19.6.2 Печать Символа Валюты
Эти элементы структуры struct lconv определяют, как печатать символ, чтобы идентифицировать валютное значение международный аналог " $ ".
Каждая страна имеет два стандартных символа валюты. Местный символ валюты используется обычно внутри страны, в то время как международный символ валюты используется всемирно, чтобы обратиться к валюте этой страны, когда необходимо указать страну недвусмысленно.
Например, многие страны используют доллар в качестве денежной единицы, но для международных валют важно определить, что мы имеем дело с Канадскими долларами, вместо Американских долларов или Австралийских долларов. Но когда контекст известен, не имеется никакой потребности объявлять эти явные долларовые количества в Канадских долларах.
char *currency_symbol
Местный символ валюты для выбранного стандарта.
В стандарте " C ", этот элемент имеет значение "" (пустая строка), означая "неопределенный". Стандарт ANSI не говорит, что делать, когда Вы находите это значение; мы рекомендуем, чтобы Вы просто печатали пустую строку, как Вы печатали бы любую другую строку, найденную в соответствующем элементе.

char *int_curr_symbol
Международный символ валюты для выбранного стандарта.
Значение int_curr_symbol должно обычно состоять из трех-символьного сокращения, определенного Международной организацией по стандартизации международного эталона как один из 4217 Кодов для Представления Валюты и Фондов, сопровождаемых разделителем (часто пробел).
В стандарте " C ", этот элемент имеет значение "" (пустая

строка), означая "неопределенный". Мы рекомендуем, чтобы Вы просто печатали пустую строку, как Вы печатали бы любую другую строку, найденную в соответствующем элементе.

char p_cs_precedes
char n_cs_precedes

Эти элементы равны 1, если currency_symbol строка предшествует значению валютного количества, или 0, если строка следует за значением. P_cs_precedes лемент применяется к положительным значениям (или нулю), а n_cs_precedes элемент применяется к отрицательным значениям.

В стандарте " C ", оба этих элемента имеют значение CHAR_MAX, означая "неопределенный". Стандарт ANSI не говорит, что делать, когда Вы находите это значение, но мы рекомендуем печатать символ валюты перед числом. Это - правильно для большинства стран. Другими словами, обработайте все значения отличные от нуля подобно этим элементам.

Стандарт POSIX говорит, что эти два элемента обращаются к int_curr_symbol также как currency_symbol. Стандарт ANSI C, кажется, подразумевает, что они должны применяться только к currency_symbol_so, int_curr_symbol должен всегда предшествовать количеству.

Мы можем только предполагать, которое из этих соответствует обычному соглашению для печати международных символов валюты. Наше предположение - то, что они должны всегда предшествовать количеству. Если мы выясим точный ответ, мы поместим его здесь.

char p_sep_by_space
char n_sep_by_space

Эти элементы равны 1, если между currency_symbol строкой и количеством ставится пробел, или 0, если никакого пробела не ставится.

P_sep_by_space элемент применяется к положительным количествам (или нулю), и n_sep_by_space элемент применяется к отрицательным количествам.

В стандарте "C", оба этих элемента имеют значение CHAR_MAX, означая "неопределенный". Стандарт ANSI не говорит, что Вы должны делать, когда Вы находите это значение; мы предлагаем, чтобы Вы обрабатывали это как 1 (печатайте пробел). Другими словами, обрабатывайте все значения отличные от нуля подобно этому элементу.

Эти элементы применяются только к currency_symbol. Когда Вы используете int_curr_symbol, Вы никогда не печатаете дополнительный пробел, потому что int_curr_symbol непосредственно содержит соответствующий разделитель.

Стандарт POSIX говорит, что эти два элемента обращаются к int_curr_symbol также как currency_symbol. Но пример в стандарте ANSI C ясно подразумевает, что они должны применяться только к currency_symbol_that, int_curr_symbol содержит любой соответствующий разделитель, так что Вы никогда не должны печатать дополнительный пробел.

Мы рекомендуем, чтобы Вы игнорировали эти элементы при печати международных символов валюты и не печатали никаких дополнительных пробелов.

19.6.3 Печать Значения Количества Денег

Эти элементы структуры struct lconv определяют, как печатать знак в валютном значении.

char *positive_sign
char *negative_sign

Это - строки, используемые, чтобы указать положительное (или нуль) и отрицательное (соответственно) валютные количества.

В стандарте "C", оба этих элемента имеет значение "" (пустая строка), означая "неопределенный".

Стандарт ANSI не говорит, что делать, когда Вы находите это значение; мы рекомендуем печатать positive_sign. Для отрицательного значения, печатайте negative_sign, если Вы находите его, если оно и positive_sign пусты, тогда печатайте "-" взамен. (Не указывать знак вообще - кажется довольно неблагоразумным.)

char p_sign_posn

char n_sign_posn
Эти элементы имеют значения типа integer и указывают, как позиционировать знак для неотрицательных и отрицательных валютных количеств, соответственно. (Строка, используется знаком определенным в positive_sign или negative_sign.) возможные значения следующие:
0 Символ валюты и количество должны быть окружены круглыми скобками.
1 Печатает строку перед символом валюты и количеством.
2 Печатает строку после символа валюты и количества.
3 Печатает строку сразу перед символом валюты.
4 Печатает строку сразу за символом валюты.
CHAR_MAX "Неопределен". Оба элемента имеют это значение в стандарте " C ". Стандарт ANSI не говорит, что Вы должны делать, когда значение - CHAR_MAX. Мы рекомендуем, чтобы Вы печатали знак после символа валюты.

20. Нелокальные Выходы

Иногда, когда ваша программа обнаруживает необычную ситуацию внутри глубоко вложенного набора обращений к функциям, Вы можете захотеть немедленно возвратиться к внешнему уровню управления. Этот раздел описывает, как делать такие нелокальные выходы, используя setjmp и longjmp функции.

20.1 Введение в нелокальные Выходы

Вот пример ситуации, где нелокальный выход может быть полезен: предположим, Вы имеете интерактивную программу, которая имеет "основной цикл" который запрашивает и выполняет команды.
Предположите, что команда "read" читает ввод из файла, делая некоторый лексический анализ и анализируя ввод. Если входная ошибка низкого уровня обнаружена, то было бы полезно возвратиться немедленно в " основной цикл " вместо того, чтобы иметь необходимость делать лексический анализ, синтаксический анализ, и все фазы обработки, которые должны явно иметь дело с ситуациями

ошибки, первоначально обнаруженными вложенными обращениями.
Некоторым образом нелокальный выход подобен использованию "return", чтобы возвратиться из функции. Но в то время как "return" отказывается только от обращения, пересылая управление обратно к функции, из которой оно вызывалось, нелокальный выход может потенциально отказываться от многих уровней вложенных обращений к функциям.
Вы определяете куда возвращать управление при нелокальных выходах, вызывая функцию setjmp. Эта функция сохраняет информацию относительно среды выполнения, в которой появляется обращение к setjmp в объекте типа jmp_buf. После обращения к setjmp выполнение программы продолжается как обычно, но если позже вызывается longjmp с соответствующим объектом jmp_buf, управление передается обратно в то место, где вызывалась setjmp. Возвращаемое значение из setjmp используется, чтобы отличить обычный возврат и возврат, сделанный обращением к longjmp, так что обращения к setjmp обычно появляются в ` if '.
Вот пример программы, описанный выше:

```
#include
#include
#include
jmp_buf main_loop;
void
abort_to_main_loop (int status)
{
        longjmp (main_loop, status);
}
int
main (void)
{
        while (1)
                if (setjmp (main_loop))
                        puts ("Back at main loop....");
                else
                        do_command ();
}
```

```
                void

                                - 428 -

                        do_command (void)
                        {
                                char buffer[128];
                                if (fgets (buffer, 128, stdin) == NULL)
                                        abort_to_main_loop (-1);
                                else
                                        exit (EXIT_SUCCESS);
                        }
```

Функция abort_to_main_loop вызывает непосредственную передачу управления в main программы, независимо от того, где она вызывается.

Способ управления внутри функции main может показаться сначала немного таинственным, но это - фактически общая идиома для setjmp. Нормальное обращение к setjmp возвращает нуль, так что "else"-часть условного выражения выполнена. Если abort_to_main_loop вызывается где-нибудь внутри выполнения команды do, то это фактически действует как будто обращение к setjmp в main возвращалось со значением -1.

Так, общий шаблон для использования setjmp выглядит вроде:

```
                        if (setjmp (buffer))
                                /* Код, для выполнения после
           преждевременного возврата. */
                                . . .
                        else
                                /* Код, который будет
             выполнен после обычной установки
             возвращающей отметки. */
                                . . .
```

20.2 Подробности нелокальных Выходов

Имеются некоторые подробности относительно функций и структур данных, используемых для выполнения нелокальных выходов.

Эти средства объявлены в " setjmp.h ".

jmp_buf (тип данных)

Объекты типа jmp_buf содержат информацию о состоянии, которое будет восстановлено при нелокальном выходе.

Содержимое jmp_buf идентифицирует конкретное место возвращения.

```
                                - 429 -
```

int setjmp (jmp_buf state) (макрос)

setjmp сохраняет информацию относительно состояния выполнения программы в state и возвращает нуль. Если longjmp позже используется, чтобы выполнить нелокальный выход к этому состоянию, setjmp возвращает значение отличное от нуля.

void longjmp (jmp_buf state, int value)

Эта функция восстанавливает текущее выполнение в состояние, сохраненное в state, и продолжает выполнение от обращения к setjmp. Возвращение из setjmp посредством longjmp возвращает значение аргумента, который был передан к longjmp, а не 0. (Но если значение задано как 0, setjmp возвращает 1).

Имеется множество неизвестных, но важных ограничений на использование setjmp и longjmp. Большинство этих ограничений присутствует, потому что нелокальные выходы требуют некоторых волшебных свойств от части компилятора Си и могут взаимодействовать с другими частями языка странными способами.

setjmp - фактически макркоманда без определения функции, так что Вы не должны пробовать к " #undef " ее или брать адрес. Кроме того, обращения к setjmp безопасны в только следующих контекстах:

* Как тестовое выражение в операторе выбора или цикла (типа "if" или "while").
* Как один операнд равенства или сравнения, которое появляется как тестовое выражение оператора выбора или цикла. Другой операнд должен быть целочисленным постоянным выражением.
* Как операнд унарного оператора "!", который появляется как как тестовое выражение оператора выбора или цикла.
* Как выражение утверждения.

Пункты возврата, допустимы только в течение динамической

протяженности функции которая вызвала setjmp, установить их. Если Вы используете longjmp чтобы возвратиться отметке, которая была установлена в функции, которая уже возвратилась, могут случиться непредсказуемые и бедственные вещи.

Вы должны использовать в longjmp аргумент отличный от нуля. То что longjmp отказывается передавать обратно аргумент нуля как возвращаемое значение из setjmp, предназначено для безопасности при случайном неправильном употреблении и не является хорошим стилем программирования.

Когда Вы выполняете нелокальный выход, все доступные объекты сохраняют любые значения, которые они имели во время вызова longjmp. Исключение - значения динамических локальных переменных, локальных для функции, содержащей обращение к setjmp, будут изменены и начиная с обращения на setjmp являются неопределенными, если Вы не объявили их отдельно.

20.3 Нелокальные Выходы и Сигналы

В системах UNIX BSD, setjmp и longjmp также сохраняют и восстанавливают набор блокированных сигналов; см. Раздел 21.7 [Блокирование Сигналов]. Однако, POSIX.1 стандарт требует чтобы setjmp и longjmp не изменяли набор блокированных сигналов, и обеспечивает дополнительную пару функций (sigsetjmp и sigsetjmp) чтобы получить поведение BSD функций.

Поведение setjmp и longjmp в библиотеке GNU управляется макркомандами теста возможностей; см. Раздел 1.3.4 [Макрокоманды Проверки Возможностей]. Значение по умолчанию в системе GNU - POSIX.1 поведение, а не поведение BSD.

Средства в этом разделе объявлены в заглавном файле " setjmp.h ".

sigjmp_buf (тип данных)

Подобен jmp_buf, за исключением того, что он может также сохранять информацию о состоянии набора блокированных сигналов.

int sigsetjmp (sigjmp_buf state, int savesigs) (функция)

Подобна setjmp. Если savesigs отличен от нуля, набор блокированных сигналов сохранен в state и будет восстановлен, если siglongjmp позже будет выполнена с этим state.

void siglongjmp (sigjmp_buf state, int value) (функция)

Подобна longjmp кроме типа аргумента state. Если обращение к sigsetjmp, которое устанавило это состояние, использовало savesigs флаг отличный от нуля, siglongjmp также восстанавливает набор блокированных сигналов.

21. Обработка Сигнала

Сигнал - программное прерывание процесса. Операционная система использует сигналы, чтобы сообщить исключительные ситуации выполняемой программе. Некоторые сигналы сообщают об ошибках типа ссылок к недопустимым адресам памяти; другие сообщают асинхронные события, типа разъединения телефонной линии.

Библиотека GNU C определяет ряд типов сигналов, каждый для конкретного вида события. Некоторые виды событий делают нецелесообразным или невозможным обычное продолжение программы, и соответствующие сигналы обычно прерывают программу. Другие виды сигналов сообщают о безобидных событиях, и игнорируются по умолчанию.

Если Вы ожидаете событие, которое вызывает сигналы, Вы можете определить функцию-обработчик и сообщить операционной системе, чтобы она выполнила ее, когда придет заданный тип сигнала.

В заключение нужно сказать, что один процесс может посылать сигнал другому процессу; это позволяет родительскому процессу прерывать дочерние, или сообщаться и синхронизироваться двум связанным процессам.

21.1 Базисные Понятия Сигналов

Этот раздел объясняет базисные понятия того, как сгенерированы

сигналы, что случается после получения сигнала, и как программы могут обрабатывать сигналы.

21.1.1 Некоторые виды Сигналов

Сигнал сообщает об исключительном событии. Вот - некоторые из событий, которые могут вызывать (или генерировать) сигнал:

* Ошибка в программе типа деления на нуль или выдачой адреса вне допустимого промежутка.
* Запрос пользователя, чтобы прерывать или завершить программу. Большинство сред допускают пользователю приостанавливать программу, печатая C-z, или завершать с C-c.
* Окончание дочернего процесса.
* Окончание ожидания таймера или будильника.
* Обращение "уничтожить процесс" из другого (или из этого) процесса. Сигналы - ограниченная но полезная форма межпроцессорной связи.

Каждый из этих видов событий (за исключением явных обращений, чтобы уничтожать и вызывать) генерирует собственный вид сигнала. Различные виды сигналов перечислены и описываются подробно в Разделе 21.2 [Стандартные Сигналы].

21.1.2 Понятия Порождения Сигналов

Вообще, события, которые генерируют сигналы, относятся к трем главным категориям: ошибки, внешние события, и явные запросы.

Ошибки означают, что программа сделала кое-что недопустимое и не может продолжать выполнение. Но не все виды ошибок генерируют сигналы фактически, как раз наоборот. Например, открытие несуществующего файла - ошибка, но она не вызывает сигнал; взамен, open возвращает -1. Вообще, ошибки, которые обязательно связаны с некоторыми библиотечными функциями, сообщаются возвратом значения, которое указывает ошибку. Ошибки, которые вызывают сигналы могут случаться где-нибудь в программе, а не только в вызовах из библиотек. Они включают деление на нуль и недопустимые адреса памяти.

Явный запрос означает использование библиотечной функции типа kill, чья цель - специально генерировать сигнал.

Сигналы могут быть сгенерированы синхронно или асинхронно. Синхронный сигнал относится к специфическому действию в программе, и вызывается (если не блокирован) в течение этого действия.

Асинхронные сигналы сгенерированы снаружи при контроле над процессом, который получает их. Эти сигналы занимают непредсказуемое время в течение выполнения. Внешние события генерируют сигналы асинхронно, и так делают явные запросы, которые обращаются к некоторому другому процессу.

Данный тип сигнала является обычно синхронным или асинхронным. Например, сигналы для ошибок обычно синхронны. Но любой тип сигнала может быть сгенерирован синхронно или асинхронно с явным запросом.

21.1.3 Как Передаются Сигналы

Когда сигнал сгенерирован, он откладывается. Обычно он задерживается на короткий период времени и потом передаеся процессу. Однако, если этот вид сигнала в настоящее время блокирован, он может оставаться отложенным, пока сигнал этого вида не

откроют. Если только он откроется, он будет передан немедленно. См. Раздел 21.7 [Блокированные Сигналы].

Когда сигнал - передан, или сразу же или после задержки, выполняется заданное действие для этого сигнала. Для некоторых сигналов, типа SIGKILL и SIGSTOP, действие фиксировано, но для большинства сигналов, программа имеет выбор: игнорировать сигнал, определить функцию обработчика, или принять заданное по умолчанию действие для этого вида сигнала. Программа определяет свой выбор используя функции типа signal или sigaction (см. Раздел 21.3 [Действия Сигнала]). Мы иногда говорим, что обработчик захватывает сигнал. В то время как обработчик выполняется, этот специфический сигнал обычно блокируется.

Если заданное действие для вида сигнала - игнорировать его, то любой такой сигнал, будет отброшен немедленно. Это случается, даже если сигнал также блокирован в это время. Сигнал, отброшенный таким

образом, передан не будет никогда, даже если программа впоследствии определяет другое действие для этого вида сигнала и откроет его.

Если прибывает сигнал, который программа не обрабатывает, и не игнорирует, происходит заданное по умолчанию действие. Каждый вид сигнала имеет собственное заданное по умолчанию действие, зарегистрированное ниже (см. Раздел 21.2 [Стандартные Сигналы]). Для большинства видов сигналов, заданное по умолчанию действие должно завершить процесс. Для некоторых видов сигналов, которые представляют "безобидные" события, заданное по умолчанию действие должны не делать ничего.

Когда сигнал завершает процесс, родительский процесс может определять причину окончания, исследуя код состояния окончания, сообщенный wait или waitpid функциями. (Это обсуждено более подробно в Разделе 23.6 [Завершение Процесса].) информация, которую он может получать, включает предложение, что окончание было из-за сигнала, и вида включаемого сигнала. Если программа, которую Вы выполняете из оболочки, завершена сигналом, оболочка обычно печатает некоторое сообщение об ошибках.

Сигналы, которые обычно представляют ошибки в программе, имеют специальную особенность: когда один из этих сигналов завершает процесс, он также формирует дамп core-файла, в который записывает состояние процесса во время окончания. Вы можете исследовать core-файл отладчиком, чтобы исследовать то, что вызвало ошибку.

Если Вы вызываете сигнал "ошибки в программе" явным запросом, и он завершает процесс, он сделает core-файл, точно как если бы сигнал был непосредственно благодаря ошибке.

21.2 Стандартные Сигналы

Этот раздел перечисляет имена для различных стандартных видов сигналов и описывает какое событие, которое они означают. Каждое имя сигнала - макркоманда, которая замещает положительное целое число - число сигнала для этого вида сигнала. Ваши программы никогда не должны делать предположения относительно числового кода для специфического вида сигнала, а обращаться к ним всегда именами, определенными здесь. Потому что число для данного вида сигнала может изменяться от системы до системы, но значения имен стандартизированы и довольно однородны.

Имена сигналов определены в заглавном файле " signal.h ".

int NSIG (макрос)

Значение этой символической константы - общее число определенных сигналов. Так как номера сигналов размещены последовательно, NSIG на один больше чем самое большое определенное число сигнала.

21.2.1 Сигналы Ошибки в программе

Следующие сигналы сгенерированы, когда операционной системой или компьютером непосредственно обнаружена серьезная ошибка в программе. Вообще, все эти сигналы - индикации, что ваша программа прервана некоторым способом, и не имеется обычно никакого способа продолжить вычисление, которое столкнулось с ошибкой.

Некоторые программы обрабатывают сигналы, возникающие при ошибке в программе, чтобы закончиться логически перед завершением. Обработчик должен завершить работу, определяя заданное по умолчанию действие для сигнала, который случался; это заставит программу завершаться с этим сигналом, как будто без обработчика. (См. Раздел 21.4.2 [Окончание Обработчика].)

Окончание - результат ошибки в программе в большинстве программ. Однако, системы программирования типа Лиспа, могут затем возвратить управление к уровню команды.

Заданное по умолчанию действие для всех этих сигналов должно заставить процесс завершиться. Если Вы блокируете или игнорируете

эти сигналы или устанавливаете обработчики для них, которые просто возвращаются, ваша программа возможно пострадает, если сигналы не сгенерированы raise или kill вместо реальной ошибки.

Когда один из этих сигналов об ошибке в программе завершает процесс, он также формирует core-файл, который записывает состояние процесса во время окончания. Файл называется " core " и записан в текущий каталог процесса. (В системе GNU, Вы можете определять имя файла для дампов переменной среды COREFILE.)

Цель core-файлов - чтобы Вы могли исследовать их отладчиком, чтобы найти то, что вызвало ошибку.

int SIGFPE (макрос)

Сигнал SIGFPE сообщает фатальную арифметическую ошибку. Хотя имя происходит от " исключение с плавающей запятой ", этот сигнал фактически покрывает все арифметические ошибки, включая деление на нуль и переполнение. Если программа сохраняет целочисленные данные в стандарте, который используется операциями с плавающей запятой, это часто, вызывает исключение " недопустимая операция ", потому что процессор не может распознать данные как число с плавающей запятой.

FPE_INTOVF_TRAP

Целочисленное переполнение (невозможно в C программе, если Вы не даете возможность прерыванию переполнения в аппаратно-специфическом режиме).

FPE_INTDIV_TRAP

Целочисленное деление на нуль.

FPE_SUBRNG_TRAP

Переход нижнего индекса (кое-что, что программы C никогда не проверяют).

FPE_FLTOVF_TRAP

Плавающее переполнения.

FPE_FLTDIV_TRAP

Плавающее/десятичное деление на нуль.

FPE_FLTUND_TRAP

Плавающее антипереполнение (переполнение снизу - слишком маленькое число).

FPE_DECOVF_TRAP

Десятичное переполнения. (Только несколько машин имеют десятичную арифметику, и C никогда не использует ее.)

int SIGILL (макрос)

Имя этого сигнала происходите от " запрещенная команда "; это означает что ваша программа пробует выполнить привилегированную команду. Так как компилятор C генерирует только допустимые команды, SIGILL обычно указывает, что исполняемый файл разрушен, или что Вы пробуете выполнять данные. Некоторые общие положения входящие в последнюю ситуацию - передача недопустимого объекта там, где ожидался указатель на функцию, или запись после конца автоматического массива (или подобные проблемы с указателями на динамические локальные переменные) и разрушение других данных стека типа адреса возврата.

int SIGSEGV (макрос)

Этот сигнал сгенерирован, когда программа пробует читать или писать вне памяти, которая размещена для этого. (Фактически, сигналы происходят только, когда программа идет достаточно далеко, чтобы быть обнаруженной механизмом защиты памяти системы.) имя - сокращение " нарушение сегментации ".

int SIGBUS (макрос)

Этот сигнал сгенерирован, когда недопустимый указатель применяется. Подобно SIGSEGV, этот сигнал - обычно результат применения неинициализированного указателя. Различие между ними в том, что SIGSEGV указывает на недопустимый доступ к допустимой памяти, в то время как SIGBUS указывает на доступ к недопустимому адресу. В частности SIGBUS сигналы часто следуют из применения расположенного с нарушением границ указателя, типа целого числа (из четырех слов) с адресом, не делимым на четыре. (Каждый вид компьютера имеет собственные требования для выравнивания адреса.)

Имя этого сигнала - сокращение для " ошибка шины ".

int SIGABRT (макрос)

Этот сигнал указывает ошибку, обнаруженную программой непосредственно и сообщается, вызовом abort. См. Раздел 22.3.4 [Прерывание выполнения Программы].

21.2.2 Сигналы Завершения

Эти сигналы используются, чтобы сообщить, что процесс завершился. Они имеют различные имена, потому что они используются для немного различных целей, и программы могли бы хотеть

обрабатывать их по-разному.

Причина обработки этих сигналов - обычно то, что ваша программа должна выполнить некоторые действия перед фактическим завершением. Например, Вы могли бы хотеть сохранять информацию о состоянии, удалить временные файлы, или восстанавливать предыдущие режимы терминала. Такой обработчик должен закончиться определением заданного по умолчанию действия для сигнала, см. раздел 21.4.2 [Окончание Обработчика].)

(Очевидное) заданное по умолчанию действие для всех этих сигналов должно заставить процесс завершаться.

int SIGHUP (макрос)

SIGHUP ("зависание") сигнал используется, чтобы сообщить, что терминал пользователя разъединен, возможно, потому что сетевое или телефонное соединение было прервано. Для получения более подробной информации см. Раздел 12.4.6 [Режимы Управления].

Этот сигнал также используется, чтобы сообщить окончание процесса управления на терминале; это окончание действительно разъединяет все процессы в сеансе с терминалом управления. Для подробной информации см. раздел 22.3.5 [Внутренняя организация Окончания].

int SIGINT (макрос)

SIGINT (" прерывание программы ") сигнал посылается, когда пользователь печатает INTR символ (обычно C-c). См. Раздел 12.4.9 [Специальные Символы], для уточнения инфрмации относительно поддержки драйвера терминала для C-c.

int SIGQUIT (макрос)

Сигнал SIGQUIT подобен SIGINT, за исключением того, что он управляется другой клавишей (символом QUIT), обычно C-\ и производит core-файл, когда он завершает процесс, точно так же как сигнал ошибки в программе. Вы можете думать об этом как об условии ошибки в программе "обнаруженном" пользователем.

См. Раздел 21.2.1 [Сигналы Ошибки в программе], для уточнения инфрмации относительно core-файлов. См. Раздел 12.4.9 [Специальные Символы], для уточнения инфрмации относительно поддержки драйвера терминала.

int SIGTERM (макрос)

Сигнал SIGTERM - обобщенный сигнал, используемый, чтобы вызвать окончание программы. В отличие от SIGKILL, этот сигнал может быть

блокирован, обрабатываться, и игнорироваться.

Команда оболочки kill генерирует SIGTERM по умолчанию.

int SIGKILL (макрос)

Сигнал SIGKILL используется, чтобы вызвать непосредственное окончание программы. Он не может быть обработан или игнорироваться, и следовательно всегда фатален. Также не возможно блокировать этот сигнал.

Этот сигнал сгенерирован только явным запросом. Так как он не может быть обработан, Вы должны генерировать его только после попытки применения менее сильнодействующего лекарственного средства типа C-c или SIGTERM.

Фактически, если SIGKILL будет не в состоянии завершать процесс, то это ошибка операционной системы, которую Вы должны сообщить.

21.2.3 Сигнализация

Эти сигналы используются, чтобы указать окончание времени таймеров. См. Раздел 17.3 [Установка Сигнализаци], для уточнения информации относительно функций, которые заставляют эти сигналы быть посланными.

Заданное по умолчанию поведение для этих сигналов должно вызвать окончание программы. Это значение по умолчанию не очень полезно; но большинство способов использования этих сигналов требовало бы функций обработчика в любом случае.

int SIGALRM (макрос)

Этот сигнал обычно указывает окончание таймера, который измеряет реальное время или время часов.

Он используется функцией alarm, например.

int SIGVTALRM (макрос)

Этот сигнал обычно указывает окончание таймера, который измеряет CPU время, используемое текущим процессом. Имя - сокращение " виртуальный таймер ".

int SIGPROF (макрос)

Этот сигнал - обычно указывает окончание таймера, который измеряет оба и CPU время, используемое текущим процессом, и CPU время, израсходованное от имени процесса системой.

21.2.4 Асинхронные Сигналы ввода - вывода

Сигналы, перечисленные в этом разделе используются вместе с асинхронными средствами ввода - вывода. Вы должны явно вызвать fcntl, чтобы дать возможность специфическому описателю файла генерировать эти сигналы (см. Раздел 8.12 [Прерывания Ввода]). Заданное по умолчанию действие для этих сигналов - игнорировать их.

int SIGIO (макрос)

Этот сигнал посылается, когда дескриптор файла готов выполнить ввод или вывод.

В большинстве операционных систем, мониторы и сокеты - единственые виды файлов, которые могут генерировать SIGIO; другие виды, включая обычные файлы, никогда не генерируют SIGIO, даже если Вы спрашиваете их.

int SIGURG (макрос)

Этот сигнал послан, когда "срочные" данные или данные вне потока прибывают в этот сокет. См. Раздел 11.8.8 [Данные вне потока].

21.2.5 Сигналы Управления заданиями

Эти сигналы используются, чтобы поддерживать управление заданиями. Если ваша система не поддерживает управление заданиями, то эти макркоманды, определены, но сигналы непосредственно не могут быть вызваны или обрабатываться.

Вы должны вообще оставить эти сигналы, если Вы действительно не понимаете, как управление заданиями работает. См. Главу 24 [Управление заданиями].

int SIGCHLD (макрос)

Этот сигнал послан родительскому процессу всякий раз, когда один из дочерних процессов завершается или останавливается.

Заданное по умолчанию действие для этого сигнала - игнорировать это. Если Вы устанавливаете обработчик для этого сигнала, в то время как имеются дочерние процессы, которые завершились, но не сообщили об их состоянии через wait или waitpid (см. Раздел 23.6 [Завершение Процесса]), то применяется ли ваш обработчик к этим процессам или нет зависит от специфической операционной системы.

int SIGCONT (макрос)

Вы можете посылать сигнал SIGCONT процессу, чтобы заставить его продолжиться. Заданное по умолчанию поведение для этого сигнала

должно заставить процесс продолжиться, если он остановлен, и игнорировать его иначе.

Большинство программ не имеет никакой причины обработать SIGCONT; они просто продолжают выполнение без информации, что они когда-либо останавливались. Вы можете использовать обработчик для SIGCONT, чтобы заставить программу сделать нечто специальное, когда она остановлена и продолжиться; например, перепечатывать подсказку, когда она приостановлена при ожидании ввода.

int SIGSTOP (макрос)

Сигнал SIGSTOP останавливает процесс. Он не может быть обработан, игнорироваться, или блокироваться.

int SIGTSTP (макрос)

Сигнал SIGTSTP - интерактивный сигнал останова. В отличие от SIGSTOP, этот сигнал может быть обработан и игнорироваться.

Ваша программа должна обработать этот сигнал, если Вы имеете специальную потребность оставить файлы или таблицы системы в безопасном состоянии, при остановке процесса. Например, программы, которые выключают отображение на экране, должны обработать SIGTSTP, так чтобы они могли направлять отображение обратно на экран перед остановкой.

Этот сигнал сгенерирован, когда пользователь печатает символ SUSP (обычно C-z). Для получения более подробной информации, см. Раздел 12.4.9 [Специальные Символы].

int SIGTTIN (макрос)

Процесс не может читать с терминала пользователя, в то время как он выполняется как фоновый. Когда любой фоновый процесс пробует читать с терминала, всем процессам посылется сигнал SIGTTIN. Заданное по умолчанию действие для этого сигнала должно остановить

процесс. Для получения более подробной информации, относительно того как он взаимодействует с драйвером терминала, см. Раздел 24.4 [Доступ к Терминалу].

int SIGTTOU (макрос)

Подобен SIGTTIN, но сгенерирован, когда фоновый процесс делает попытку записи на терминал или устанавливать режимы. Снова, заданное по умолчанию действие должно остановить процесс.

В то время когда процесс приостановлен, сигналы не могут быть переданы ему, за исключением сигналов SIGKILL и (очевидно) SIGCONT. Сигнал SIGKILL всегда вызывает окончание

процесса и не может быть блокирован или игнорироваться. Вы можете блокировать или игнорировать SIGCONT, но он всегда заставляет процесс быть продолженным во всяком случае, если он остановлен. Посылка сигнала SIGCONT на процесс заставляет любые отложенные сигналы останова для этого процесса быть отброшенными. Аналогично, любая задержка SIGCONT сигнала для процесса отброшена, когда он получает сигнал останова.

21.2.6 Разнообразные Сигналы

Эти сигналы используются, чтобы сообщить некоторые другие условия. Заданное по умолчанию действие для всех из них должно заставить процесс завершиться.

int SIGPIPE (макрос)

Если Вы используете водопроводы или FIFO, Вы должны разработать ваше приложение так, чтобы один процесс открыл водопровод для чтения перед тем как другой начал запиись. Если процесс считывания никогда не начинается, или завершается неожиданно, запись в водопровод или FIFO вызывает сигнал SIGPIPE. Если SIGPIPE блокирован, обрабатывается или игнорируется, дргие обращения вызывают EPIPE взамен.

Водопроводы и FIFO обсуждены более подробно в Главе 10 [Водопроводы и FIFO].

Другая причина SIGPIPE - когда Вы пробуете выводить на сокет, который не соединен. См. Раздел 11.8.5.1 [Посылка Данных].

int SIGUSR1 (макрос)

int SIGUSR2 (макрос)

SIGUSR1 и SIGUSR2 отложены для Вас, чтобы использовать их любым способом, которым Вы хотите.

Они полезны для межпроцессорной связи. Так как эти сигналы обычно фатальны, Вы должны написать обработчик сигнала для них в программе, которая получает сигнал.

Имеется пример, показывающий использование SIGUSR1 и SIGUSR2 в Разделе 21.6.2 [Передача сигналов Другому Процессу].

21.2.7 Нестандартные Сигналы

Специфические операционные системы поддерживают дополнительные сигналы, не перечисленные выше. Стандарт ANSI C резервирует все идентификаторы, начинающиеся с " SIG " сопровождаемые символом

верхнего регистра для имен сигналов. Вам нужно смотреть документацию или заголовочные файлами для вашей конкретной операционной системы и типа процессора, чтобы выяснить какие сигналы поддерживаются.

Например, некоторые системы особо поддерживают сигналы, которые соответствуют аппаратным средствам. Некоторые другие виды сигналов используются, чтобы выполнить ограничения времени CPU или использования файловой системы, для асинхронных изменений конфигурации терминала, и т.п.. Системы могут также определять имена сигналов, которые являются побочными результатами исследований для стандартных имен сигналов.

Заданное по умолчанию действие (или действие, установленное оболочкой) для определенных реализацией сигналов обычно приемлемо. Фактически, обычно не стоит игнорировать или блокировать сигналы, о которых Вы не знаете, или пробовать устанавливать обработчик для сигналов, чьи значения Вы не понимаете.

Вот нгекоторые сигналы, которые обычно используются в операционных системах:

SIGCLD Устаревшее имя для SIGCHLD.

SIGTRAP Сгенерированный командой breakpoint машины. Используется отладчиками. Заданное по умолчанию действие должно формировать

core-файл.
SIGIOT Сгенерированный PDP-11 "iot" командой; эквивалент SIGABRT. Заданное по умолчанию действие должно формировать core-файл.
SIGEMT Ловушка эмулятора; следует из некоторых невыполненных команд. Это - сигнал ошибки в программе.
SIGSYS Ошибочный системный вызов; то есть команда системного вызова была выполнена, но код системного вызова недопустим. Это - сигнал ошибки в программе.
SIGPOLL Это - имя сигнала System V, более или менее подобное SIGIO.
SIGXCPU Превышение ограничения времени CPU. Заданное по умолчанию действие - окончание программы.
SIGXFSZ Файлом превышено ограничение размера. Заданное по умолчанию действие - окончание программы.
SIGWINCH Изменение размера окна. Он генерируется на некоторых системах, когда размер текущего окна на экране изменен. Заданное по

умолчанию действие - игнорировать это.

21.2.8 Сообщения Сигнала
Мы упомянули выше, что оболочка печатает сообщение, описывающее сигнал, который завершил дочерний процесс. Простой способ печатать сообщение, описывающее сигнал состоит в том, чтобы использовать функции strsignal и psignal. Эти функции используют номер сигнала, чтобы определить, какой вид сигнала описывать. Номер сигнала может исходить из состояния окончания дочернего процесса (см. Раздел 23.6 [Завершение Процесса]) или он может исходить из обработчика сигнала на том же самом процессе.
char * strsignal (int signum) (функция)
Эта функция возвращает указатель на статически размещенную строку, содержащую сообщение, описывающее сигнал. Вы не должны изменять содержимое этой строки; и, так как она может быть перезаписана при последующих обращениях, Вы должны сохранить его копию, если Вы должны сослаться на него позже.
Эта функция - расширение GNU, объявленное в заглавном файле "string.h".
void psignal (int signum, const char *message)
Эта функция печатает сообщение, описывающее сигнал, в стандартный поток ошибкок - stderr; см. Раздел 7.2 [Стандартные Потоки].
Если Вы вызываете psignal с сообщением, которое является или пустым указателем или пустой строкой, psignal печатает сообщение, соответствующее сигналу, добавляя конечный символ перевода строки.
Если Вы обеспечиваете непустой аргумент сообщения, то psignal предворяет вывод этой строкой. Она добавляет двоеточие и пробел, чтобы отделить сообщение от строки, соответствующей сигналу.
Эта функция - возможность BSD, объявленная в заглавном файле "stdio.h ".
Имеется также массив sys_siglist, который содержит сообщения для различных кодов сигнала. Этот массив существует в системах BSD, в отличие от strsignal.

21.3 Определение Действий Сигнала
Самый простой способ изменить действие для сигнала состоит в том, чтобы использовать функцию signal. Вы можете определять встроенное действие (типа игнорировать сигнал), или Вы можете устанавливать обработчик.
Библиотека GNU также осуществляет более универсальное средство sigaction. Этот раздел описывает эти средства и дает предложения, когда их использовать.

21.3.1 Основная Обработка сигналов
Функция signal обеспечивает простой интерфейс для установки действия для специфического сигнала.
Функция и связанные макркоманды объявлены в заголовочном файле

"signal.h ".
sighandler_t (тип данных)
Это - тип функций обработчика сигнала. Обработчики Сигнала воспринимают один целочисленный аргумент, определяющий номер сигнала, и возвращают тип void. Вы должны определять функции обработчика примерно так:

```
void handler (int signum) { . . . }
```

Имя sighandler_t для этого типа данных - расширение GNU.
sighandler_t signal (int signum, sighandler_t action) (функция)
Функция signal устанавливает action как действие для сигнала signum.
Первый аргумент, signum, идентифицирует сигнал, чьим поведением Вы хотите управлять, и должен быть номером сигнала. Соответствующий способ определять номер сигнала - одним из символических имен сигналов, описанных в Разделе 21.2 [Стандартные Сигналы] не используют явное число, потому что числовой код для данного вида сигнала может измениться от операционной системы к операционной системе.
Второй аргумент, action, определяет действие используемое для сигнала signum. Оно может быть одно из следующих:
SIG_DFL определяет заданное по умолчанию действие для специфического сигнала. Заданные по умолчанию действия для различных видов сигналов установлены в Разделе 21.2 [Стандартные Сигналы].

SIG_IGN определяет, что сигнал должен игнорироваться.
Ваша программа вообще не должна игнорировать сигналы, которые представляют серьезные события, или обычно используется, чтобы запросить окончание. Вы не можете игнорировать SIGKILL или SIGSTOP вообще. Вы можете игнорировать сигналы ошибки в программе подобно SIGSEGV, но игнорирование ошибки не будет давать возможность программе продолжить осмысленное выполнение. Игнорирование запросов пользователя типа SIGINT, SIGQUIT, и SIGTSTP некультурно!!.
Когда Вы не желаете, чтобы сигналы были передан в некоторую части программы, нужно блокировать их, а не игнорировать. См. Раздел 21.7 [Блокированные Сигналы].
handler обеспечивает адрес функции обработчика в вашей программе, выполнение этого обработчика определяется как способ передать сигнал.
Для получения более подробной информации относительно функции-обработчика сигнала, см. Раздел 21.4 [Определение Обработчиков].
Функция signal возвращает action, который был задан для указанного signum. Вы можете сохранять это значение и восстанавливать его позже, вызывая signal снова.
Если signal не может выполнить запрос, она возвращает SIG_ERR. Следующие errno условия ошибки определены для этой функции:
EINVAL Вы определили недопустимый signum; или Вы пробовали игнорировать или обеспечивать обработчик для SIGKILL или SIGSTOP.
Вот простой пример установки обработчика, чтобы удалить временные файлы, при получении некоторых фатальных сигналов:

```
#include
void
termination_handler (int signum)
{
        struct temp_file *p;
        for (p = temp_file_list; p; p = p->next)
                unlink (p->name);
}
int
main (void)
{
        . . .
```



```
        if (signal (SIGINT, termination_handler)
                == SIG_IGN)
                signal (SIGINT, SIG_IGN);
        if (signal (SIGHUP, termination_handler)
                == SIG_IGN)
                signal (SIGHUP, SIG_IGN);
        if (signal (SIGTERM, termination_handler)
```

```
                                    == SIG_IGN)
                                  signal (SIGTERM, SIG_IGN);
                        . . .
                }
```

Обратите внимание, что, если данный сигнал был предварительно установлен, чтобы игнорироваться, то код избегает изменять эту установку. Потому что оболочки часто игнорируют некоторые сигналы, при старте дочерних процессов, и эти процессы не долхны изменять это.

Мы не обрабатываем SIGQUIT или сигналы ошибки в программе в этом примере, потому что они разработаны, чтобы обеспечить информацию для отладки (core-файл), и временные файлы могут давать полезную информацию.

sighandler_t ssignal (int signum, sighandler_t action) (функция)

Функция ssignal делает ту же самую вещь что signal; она предоставляется только для совместимости с SVID.

sighandler_t SIG_ERR (макрос)

Значение этой макркоманды используется как возвращаемое значение из signal, чтобы указать ошибку.

21.3.2 Сложная Обработка Сигнала

Функция sigaction имеет тот же самый основной эффект как signal: определять, как сигнал должен быть обработан процессом. Однако, sigaction предлагает большое количество управления, за счет большего количества сложности. В частности sigaction позволяет Вам определять дополнительные флаги, чтобы управлять тем, когда генерируется сигнал и как вызывается обработчик.

Функция sigaction объявлена в "signal.h".

struct sigaction (тип данных)

Структуры типа struct sigaction используются в функции

sigaction, чтобы определить всю информацию относительно того, как обработать специфический сигнал. Эта структура содержит по крайней мере следующие элементы:

sighandler_t sa_handler

Это используется таким же образом, как аргумент action функции signal. Значение может быть SIG_DFL, SIG_IGN, или указатель на функцию. См. Раздел 21.3.1 [Общая Обработка сигнала].

sigset_t sa_mask

Этот определяет набор сигналов, которые будут блокированы, в то время как обработчик выполняется. Блокирование объясняется в Разделе 21.7.5 [Блокирование для Обработчика]. Обратите внимание, что сигнал, который был передан, автоматически блокирован по умолчанию прежде, чем обработчик начат; это - истина независимо от значения в sa_mask. Если Вы хотите, чтобы этот сигнал не был блокирован внутри обработчика, Вы должен записать в обработчике код чтобы разблокировать его.

int sa_flags

Определяет различные флаги, которые могут воздействовать на поведение сигнала. Они описаны более подробно в Разделе 21.3.5 [Флаги для Sigaction].

int sigaction (int signum, const struct sigaction *action, struct sigaction *old_action)

Аргумент action используется, чтобы установить новое действие для указанного сигнала, в то время как old_action аргумент используется, чтобы возвратить информацию относительно действия, предварительно связанного с этим сигналом. (Другими словами, Вы можете выяснить, что старое действие в действительности делало для сигнала, и восстановить его позже, если Вы хотите.)

Возвращаемое значение от sigaction - нуль, если она преуспевает, и -1 при отказе. Следующие errno условия ошибки определены для этой функции:

EINVAL аргумент signum не допустим, или Вы пробуете обрабатывать или игнорировать SIGKILL или SIGSTOP.

21.3.3 Взаимодействие signal и sigaction

Возможно использовать signal и sigaction внутри одной программы, но Вы должны быть внимательным, потому что они могут взаимодействовать немного странными способами.

Функция sigaction определяет более подробную информацию, чем функция signal, так что возвращаемое значение из signal не может выражать все возможности sigaction. Следовательно, если Вы используете signal, чтобы сохранить и позже восстанавливать действие, она может быть не способна восстановить правильно обработчик, который был установлен с sigaction.

Чтобы избежать проблем всегда используйте sigaction, чтобы сохранить и восстановить обработчик, если ваша программа использует sigaction вообще. Так как sigaction более общая, она может правильно сохранять и восстанавливать любое действие, независимо от того, было ли оно установлено первоначально с signal или sigaction.

Если Вы устанавливаете действие с signal, а потом исследуете его sigaction, адрес обработчика, который Вы получаете, может быть не такой же как тот, что Вы определили с signal. Он так же не может использоваться как аргумент action в signal. Но Вы можете полагаться на использование его как аргумента sigaction.

Так что лучше отказаться от использования того и другого механизма последовательно внутри однй программы.

Примечание о переносимости: общая функция сигнала - возможность ANSI C, в то время как sigaction - часть POSIX.1 стандарта. Если Вы беспокоитесь относительно переносимости на не-posix системы, то Вы должены использовать функцию signal.

21.3.4 Пример Функции sigaction

В Разделе 21.3.1 [Общая Обработка сигнала], мы привели пример установки простого обработчика для сигналов окончания, используя signal. Вот эквивалентный пример, использующий sigaction:

```
#include
void
termination_handler (int signum)
{
        struct temp_file *p;
        for (p = temp_file_list; p; p = p->next)
                unlink (p->name);
}
int
main (void)
{
```



```
        . . .
        struct sigaction new_action, old_action;
        new_action.sa_handler = termination_handler;
        sigemptyset (&new_action.sa_mask);
        new_action.sa_flags = 0;
        sigaction (SIGINT, NULL, &old_action);
        if (old_action.sa_handler != SIG_IGN)
                sigaction (SIGINT, &new_action, NULL);
        sigaction (SIGHUP, NULL, &old_action);
        if (old_action.sa_handler != SIG_IGN)
                sigaction (SIGHUP, &new_action, NULL);
        sigaction (SIGTERM, NULL, &old_action);
        if (old_action.sa_handler != SIG_IGN)
                sigaction (SIGTERM, &new_action, NULL);
        . . .
}
```

Программа просто загружает структуру new_action с желательными параметрами и передает ее в sigaction. Использование sigemptyset описано позже; см. Раздел 21.7 [Блокированные Сигналы].

В примере, использующем signal, мы старались избегать обрабатывать предварительно установленные сигналы. Здесь мы можем мгновенно избегать изменять обработчик сигнала, используя возможность sigaction, которая позволяет исследовать текущее действие без определения нового.

Вот другой пример. Он отыскивает информацию относительно текущего действия для SIGINT без замены этого действия.

```
struct sigaction query_action;
if (sigaction (SIGINT, NULL, &query_action) < 0)
        /* sigaction возвращает -1 в случае ошибки. */
else if (query_action.sa_handler == SIG_DFL)
        /* SIGINT обработан заданным по умолчанию,
        фатальным способом. */
else if (query_action.sa_handler == SIG_IGN)
        / * SIGINT игнорируется. * /
```

```
                else
                        /* Определенный программистом
          обработчик сигнала. */
```


21.3.5 Флаги для sigaction

элемент структуры sa_flags - sigaction определяет специальные возможности. В большинстве случаев, SA_RESTART - хорошее значение, чтобы использовать для этого поля.

Значение sa_flags интерпретируется как битовая маска. Таким образом, Вы должны выбрать флаги, которые Вы хотите установить, сделать операцию OR для этих флагов, и сохранить результат в sa_flags элементе вашей структуры sigaction.

Каждый номер сигнала имеет собственный набор флагов. Каждое обращение к sigaction воздействует на один номер сигнала, и флаги, которые Вы определяете, применяются только к этому специфическому сигналу.

В библиотеке GNU C, установка обработчика сигнала обнуляет все флаги, кроме SA_RESTART, чье значение зависит от установок, которые Вы сделали в siginterrupt. См. Раздел 21.5 [Прерванные Примитивы].

Эти макркоманды определены в заголовочном файле " signal.h ".

int SA_NOCLDSTOP (макрос)

Этот флаг имеет смысл только для сигнала SIGCHLD. Когда флаг установлен, система передает сигнал для завершенного дочернего процесса, но не для того, который остановлен. По умолчанию, SIGCHLD - передан для и завершенных дочерних процессов и для остановленных дочерних процессов.

При установке этого флага для сигнала отличного от SIGCHLD не имеет никакого эффекта.

int SA_ONSTACK (макрос)

Если этот флаг установлен для конкретного сигнала, система использует стек сигнала при сообщении этого вида сигнала. См. Раздел 21.9 [Обработка Сигнала BSD].

int SA_RESTART (макрос)

Этот флаг управляет тем, что случается, когда сигнал - передан в течение выполнения некоторых примитивов (типа open, read или write). Имеются два варианта: библиотечная функция может продолжиться, или она может возвратить отказ с кодом ошибки EINTR.

Выбор управляется SA_RESTART флагом для конкретного вида сигнала, который был передан. Если флаг установлен, по возвращении из обработчика продолжается библиотечная функция. Если флаг не установлен, происходит сбой функции. См. Раздел 21.5 [Прерванные Примитивы].

21.3.6 Начальные Действия Сигнала

После создания, новый процесс (см. Раздел 23.4 [Создание Процесса]) наследует обработку сигналов из родительского процесса. Однако, когда Вы загружаете новый образ процесса, используя функцию exec (см. Раздел 23.5 [Выполнение Файла]) обработчики любых сигналов возвращаются к SIG_DFL. Конечно, новая программа может устанавливать собственные обработчики.

Когда программа выполняется оболочкой, оболочка обычно устанавливает начальные действия для дочернего процесса как SIG_DFL или SIG_IGN, соответственно.

Вот пример того, как устанавливать обработчик для SIGHUP, но если SIGHUP в настоящее время не игнорируется:

```
. . .
struct sigaction temp;
sigaction (SIGHUP, NULL, &temp);
if (temp.sa_handler != SIG_IGN)
{
        temp.sa_handler = handle_sighup;
        sigemptyset (&temp.sa_mask);
        sigaction (SIGHUP, &temp, NULL);
}
```

21.4 Определение Обработчиков Сигнала

Этот раздел описывает, как написать функцию обработчика сигнала, которая может быть установлена функциями sigaction или signal.

Обработчик сигнала - функция, которую Вы компилируете вместе с

остальной частью программы. Вместо непосредственно вызова функции, Вы используете signal или sigaction, чтобы сообщить, чтобы операционная система вызвала ее, когда приходит сигнал. Это называется установкой обработчика. См. Раздел 21.3 [Действия Сигнала].

Имеются две стратегии, которые Вы можете использовать в функциях обработчика сигнала:

* Вы можете иметь функцию обработчика, которая отмечает, что сигнал пришел, в некоторых глобальных структурах данных, и нормально возвращается.
* Вы можете иметь функцию обработчика, которая завершают

программу или передает управление к отметке, где программа может избавиться от ситуации, которая вызвала сигнал.

Вы должны быть очень осторожны при написании функций обработчика, потому что они могут вызываться асинхронно. То есть обработчик может вызываться в любом месте программы, непредсказуемо. Если два сигнала прибывают в течение очень короткого интервала, один обработчик может выполняться внутри другого. Этот раздел описывает то, что ваш обработчик должен делать, и чего Вы должны избежать.

21.4.1 Обработчики Сигнала, которые Возвращаются

Обработчики, которые возвращаются обычно используются для сигналов типа SIGALRM и ввода-вывода и межпроцессорных сигналов связи. Но обработчик для SIGINT может также возвращаться после установки флага, который сообщает, чтобы программа завершилась в удобное время.

Небезопасно возвращаться из обработчика при сигнале ошибки в программе, потому что поведение программы, когда функция обработчика возвращается, не определено после ошибки в программе. См. Раздел 21.2.1 [Сигналы Ошибки в программе].

Обработчики, которые возвращаются, должны изменять некоторую глобальную переменную, чтобы иметь какой-нибудь эффект.

Обычно, эта переменная исследуется периодически программой. Тип данных должен быть sig_atomic_t по причинам, описанным в Разделе 21.4.7 [Быстрый Доступ к данным].

Вот простой пример такой программы. Она выполняет тело цикла, пока она не отметит, что сигнал SIGALRM прибыл.

```
#include
#include
#include
volatile sig_atomic_t keep_going = 1;
void
catch_alarm (int sig)
{
        keep_going = 0;
        signal (sig, catch_alarm);
}
```



```
void
do_stuff (void)
{
        puts ("Doing stuff while waiting
for alarm....");
}
int
main (void)
{
        signal (SIGALRM, catch_alarm);
        alarm (2);
        while (keep_going)
                do_stuff ();
        return EXIT_SUCCESS;
}
```

21.4.2 Обработчики, которые Завершают Процесс

Функции-Обработчики, которые завершают программу, обычно используются, чтобы вызвать организованную очистку от сигналов

ошибки в программе и интерактивных прерываний.

Самый чистый способ для обработчика завершить процесс состоит в том, чтобы вызвать тот же самый сигнал, который активизировал обработчик. Вот, как сделать это:

```
volatile sig_atomic_t fatal_error_in_progress = 0;
void
fatal_error_signal (int sig)
{
        if (fatal_error_in_progress)
                raise (sig);
        fatal_error_in_progress = 1;
        / * Теперь делаем очищающие действия:
                        - установка режимов терминала
                        - уничтожение дочерних процессов
                        - удаление временных файлов * /
        . . .
        raise (sig);
}
```


21.4.3 Нелокальная Передача Управления в Обработчиках

Вы можете делать нелокальную передачу управления вне обработчика сигнала, используя setjmp и longjmp средства (см. Главу 20 [Нелокальные Выходы]).

Когда обработчик делает нелокальную передачу управления, часть программы, которая выполнялась, не будет продолжаться. Если эта часть программы была в процессе изменения важной структуры данных, структура данных останется несогласованной. Так как программа не завершается, несогласованность должна быть отмечена позже.

Имеются два способа избежать этой проблемы. Первый - блокировать сигнал для частей программы, которые изменяют важные структуры данных. Блокирование сигнала задерживает выдачу, пока он не открыт, как только критическое изменение закончено. См. Раздел 21.7 [Блокированные Сигналы].

Иначе, нужно повторно инициализировать определяющие структуры данных в обработчике сигнала, или сделать их значения непротиворечивыми.

Вот довольно схематический пример, показывающий переинициализацию одной глобальной переменной.

```
#include
#include
jmp_buf return_to_top_level;
volatile sig_atomic_t waiting_for_input;
void
handle_sigint (int signum)
{
        waiting_for_input = 0;
        longjmp (return_to_top_level, 1);
}
int
main (void)
{
        . . .
        signal (SIGINT, sigint_handler);
        . . .
        while (1) {
```



```
                prepare_for_command ();
                if (setjmp (return_to_top_level) == 0)
                        read_and_execute_command ();
        }
}
char *
read_data ()
{
        if (input_from_terminal) {
                waiting_for_input = 1;
                . . .
                waiting_for_input = 0;
        }
```

```
                        else {
                                . . .
                        }
                }
```

21.4.4 Прибытие Сигналов во Время Выполнения Обработчика

Что случается, если другой сигнал приходит, когда выполняется ваша функция обработчика сигнала?

Когда вызывается обработчик для конкретного сигнала, этот сигнал обычно блокируется, пока обработчик не возвращается. Это означает что, если два сигнала того же самого вида приходят через очень короткий интервал времени, второй будет приостановлен, пока первый не будет обработан.
(Обработчик может явно открыть сигнал, используя sigprocmask, если Вы хотите позволить прибывать большему количеству сигналов этого типа; см. Раздел 21.7.3 [Маска Сигнала Процесса].)

Однако, ваш обработчик может все еще прерываться получением другого вида сигнала. Чтобы избежать этого, Вы можете использовать sa_mask - элемент структуры action, передаваемой sigaction, чтобы явно определить, какие сигналы должны быть блокированы в то время как обработчик сигнала выполняется. См. Раздел 21.7.5 [Блокирование для Обработчика].

Примечание о переносимости: Всегда используйте sigaction, чтобы установить обработчик для сигнала, который Вы ожидаете получать асинхронно, если Вы хотите, чтобы ваша программа работала правильно

на System V Unix.

21.4.5 Близкие (по времени) Сигналы Обьединяются в Один

Если несколько сигналов того же самого типа переданы вашему процессу прежде, чем ваш обработчик сигнала может быть вызван вообще, то обработчик может вызываться только один раз, как будто только одиночный сигнал прибыл. В результате сигналы обьединяются в один. Эта ситуация может возникать, когда сигнал блокирован, или в многопроцессорной среде, где система - занята выполнением некоторых других процессов, в то время как сигналы - передан. Это означает, например, что Вы не можете надежно использовать обработчик сигнала, чтобы считать сигналы. Единственное, что Вы можете сделать - узнать, прибыл ли по крайней мере один сигнал начиная с данного времени.

Вот пример обработчика для SIGCHLD, который компенсирует предложение, что число полученных сигналов не может равняться числу дочерних процессов, генерируя их. Он подразумевает, что программа следит за всеми дочерними процессами цепочкой структур следующим образом:

```
                struct process
                {
                        struct process *next;
                        int pid;
                        int input_descriptor;
                        int status;
                };
                struct process *process_list;
```

Этот пример также использует флаг, чтобы указать, прибыли ли сигналы начиная с некоторого времени в прошлом, когда программа в последний раз обнулила его.

```
                int process_status_change;
```

Вот обработчик непосредственно:

```
                void
                sigchld_handler (int signo)
                {
                        int old_errno = errno;
                        while (1) {
                                register int pid;
```



```
                                int w;
                                struct process *p;
                                do
                                        {
                                                errno = 0;
                                                pid = waitpid (WAIT_ANY,
```

```
                                        &w,
                WNOHANG | WUNTRACED);
                                }
                                while (pid <= 0 && errno == EINTR);
                                if (pid <= 0) {
                                        errno = old_errno;
                                        return;
                                }
                                for (p = process_list; p; p = p->next)
                                        if (p->pid == pid) {
                                                p->foo1   = w;
                                                p->foo1        = 1;
                                                if (WIFSIGNALED (w) ||
                                        WIFEXITED (w))
                                                if (p->input_descriptor)
                 FD_CLR (p->input_descriptor, &input_wait_mask);
                                                ++process_status_change;
                                        }
                        }
                }
```

Вот соответствующий способ проверять флаг process_status_change:

```
                if (process_status_change) {
                        struct process *p;
                        process_status_change = 0;
                        for (p = process_list; p; p = p->next)
                                if (p->have_status) {
                                        ... Исследуют p->status ...
                                }
                }
```

Важно очистить флаг перед исследованием списка; иначе, если сигнал был передан только перед очисткой флага, и после того, как соответствующий элемент списка процесса был проверен, изменение

состояния будет неотмеченным, пока следующий сигнал не прибыл, чтобы установить флаг снова. Вы могли бы, конечно, избегать этой проблемы, блокировав сигнал при просмотре списка, но более важно гарантировать правильность, делая все в правильном порядке.

Цикл, который проверяет состояние процесса избегает исследовать p->status, пока он не видит, что состояние было законно сохранено. Он должен удостовериться, что status не может изменяться в середине доступа к нему. Как только p->have_status установлен, это означает что дочерний процесс остановлен или завершен, и в любом случае он не может останавливаться или завершаться снова. См. Раздел 21.4.7.3 [Быстрое Использование], для получения более подробной информации относительно копирования с прерываниями во время доступов к переменной.

Вот, таким образом Вы можете проверять, выполнился ли обработчик начиная с последней проверки.

Эта методика использует счетчик, который никогда не изменяется снаружи обработчика. Вместо того, чтобы очищать счетчик, программа помнит предыдущее значение, и видит, изменилось ли оно начиная с предыдущей проверки. Преимущество этого метода - в том, что различные части программы могут проверять независимо, каждая часть проверяет имелся ли сигнал начиная с последней проверки в этой части.

```
                sig_atomic_t process_status_change;
                sig_atomic_t last_process_status_change;
                . . .
                {
                        sig_atomic_t prev=last_process_status_change;
                        last_process_status_change =
          process_status_change;
                        if (last_process_status_change != prev) {
                                struct process *p;
                                for (p = process_list; p; p = p->next)
                                        if (p->have_status) {
                                                ... Проверка p->status ...
                                        }
                        }
                }
```

21.4.6 Обработка Сигнала и Неповторно используемые Функции

Функции-Обработчики обычно делают не очень много. Самый лучший обработчик - который не делает ничего, но устанавливает внешнюю переменную, которую программа проверяет регулярно, и оставляет всю серьезную работу программе. Это самое лучшее, потому что обработчик может вызываться асинхронно, в непредсказуемое время, возможно в середине системного вызова, или даже между началом и концом оператора Си, который требует выполнения многократных команд. Изменяемые структуры данных могут быть в несогласованном состоянии, когда вызывается функция обработчика. Даже копирование одной int переменной в другую занимает две команды на большинстве машин.

Это означает, что Вы должны быть очень осторожын относительно того, что Вы делаете в обработчике сигнала.

* Если ваш обработчик должен обратиться к некоторым глобальным переменным вашей программы, объявляйте эти переменные независимо. Это сообщает компилятору, что значение переменной может изменяться асинхронно, и запрещает некоторые оптимизации.

* Если Вы вызываете функцию в обработчике, удостоверьтесь, что она повторно используема относительно сигналов, и еще удостоверьтесь, что сигнал не может прервать обращение к зависимой функции.

Функция не может повторно использоваться, если она использует память, которая не на стеке.

* Если функция использует статическую переменную или глобальную переменную, или динамически размещенный объект, который она устанавливает для себя, то она - неповторно используемая и любые два обращения к функции могут смешиваться.

Например, предположите, что обработчик сигнала использует gethostbyname. Эта функция возвращает значение в статическом объекте, многократно используя тот же самый объект каждый раз. Если сигнал прибывает в течение обращения в gethostbyname, или даже после него (в то время как программа все еще использует значение), то он затрет значение еще не прочитанное программой.

Однако, если программа не использует gethostbyname или любую другую функцию, которая возвращает информацию в том же самом объекте, или если она всегда блокирует сигналы вокруг каждого

использования, то Вы в безопасности.

Имеется большое количество библиотечных функций возвращающих значения в фиксированном объекте, всегда многократно используя тот же самый объект в этом режиме, и все из них вызывают ту же самую проблему. Описание функции в этом руководстве всегда упоминает это поведение.

* Если функция использует и изменяет объект, который Вы обеспечиваете, то она неповторно используема; два обращения могут смешиваться, если они используют тот же самый объект.

Этот случай возникает, когда Вы делаете ввод - вывод, используя потоки. Предположите, что обработчик сигнала печатает сообщение с fprintf. Предположите, что программа была в середине обращения к fprintf, используя некоторый поток, когда был получен сигнал. И сообщение обработчика сигнала и данные программы могут быть разрушены, потому что оба обращения функционируют на том же самом потоке непосредственно.

Однако, если Вы знаете, что поток используемый обработчиком не может использоваться программой одновременно, когда прибывают сигналы, то все хорошо. Это не проблема, если программа использует какой-нибудь другой поток.

* На большинстве систем, malloc и free неповторно используемы, потому что они используют статическую структуру данных, которая хранит сведения о свободных блоках. В результате, все библиотечные функции, которые резервируют или освобождают память, не повторно используемы, включая функции, которые резервируют место, чтобы сохранить результат.

Самый лучший способ избегать потребности зарезервировать память - зарезервировать заранее пространство для обработчиков сигнала.

Самый лучший способ избегать освобождения памяти в обработчике - отмечать или запоминать объекты, которые будут освобождены, и иметь возможность проверять в программе время от времени, ждет ли что-нибудь, чтобы его освобождили.

Но это должно быть выполнено аккуратно, т.к.
размещение объекта в цепочке - не быстрая операция, и если оно прервано
другим обработчиком сигнала, который делает ту же самую вещь, Вы можете
"потерять" один из объектов.
В системе GNU, malloc и free безопасны для использования в

обработчиках сигнала, потому что они блокируют сигналы. В
результате, библиотечные функции, которые резервируют пространство
для результата также безопасны в обработчиках сигналов. Obstack
функции резервирования безопасны, если Вы не используете тот же
самый obstack, и внутри и вне обработчика сигнала.
Функции резервирования настройки (см. Раздел 3.6 [Настройка
Программы распределения]) конечно не безопасны, для использования в
обработчике сигнала.
* Любая функция, которая изменяет errno, неповторно
используема, но Вы можете исправить это: в обработчике, сохраните
первоначальное значение errno, и восстановите его перед
возвращением. Это предотвращает смешивание ошибок, которые
происходят внутри обработчика сигнала, с ошибками из системных
вызовов в точке прерывания программы, чтобы выполнить обработчик.
Эта методика вообще применима; если Вы хотите вызвать
обработчик, который изменяет специфический объект в памяти, Вы
можете делать его безопасным, сохраняя и восстанавливая этот
объект.
* Просто чтение из объекта памяти безопасно, если Вы можете
иметь дело с любым из значений, которые могли бы появляться в
объекте одновременно, когда сигнал может быть получен. Имейте в
виду, что доступ к некоторым типам данных требует больше чем одну
команду, это означает что обработчик может выполняться
"в середине" доступа к переменной. См. Раздел 21.4.7 [Быстрый Доступ к
данным].
* Просто запиись в объект памяти безопасна, если только
внезапное изменение значения, в любое время, когда обработчик мог
бы выполняться, не будет нарушать чего-нибудь.

21.4.7 Быстрый Доступ к данным и Обработка Сигнала
Для любого вида данных в вашем приложении, Вы должны быть
внимательны к тому, что доступ к одиночному элементу
данных не обязательно быстрый. Это означает, что это может занимать
больше чем одну команду. В таких случаях, обработчик сигнала может
выполняться в середине чтения или запииси объекта.
Имеются три способа, при помощи которых Вы можете справляться с этой
проблемой. Вы можете использовать типы данных, к которым всегда
обращаются быстро; Вы можете тщательно упорядочивать их, так что

ничего неблагоприятного не случается, если доступ прерван, или Вы
можете блокировать все сигналы вокруг любого доступа, который
лучше не прерывать (см. Раздел 21.7 [Блокированные Сигналы]).

21.4.7.1 Проблемы с Немгновенным Доступом
Вот пример, который показывает то, что может случиться, если
обработчик сигнала выполняется в середине изменения переменной.
(Прерывание чтения переменной может также привести к парадоксальным
результатам, но здесь мы показываем только запиись.)

```
#include
#include
struct two_words { int a, b; } memory;
void
handler(int signum)
{
        printf ("%d,%d\n", memory.a, memory.b);
        alarm (1);
}
int
main (void)
{
        static struct two_words zeros = {0,0},
                                ones  = {1,1};
        signal (SIGALRM, handler);
        memory = zeros;
        alarm (1);
```

```
                while (1)
                {
                        memory = zeros;
                        memory = ones;
                }
        }
```

Эта программа заполняет память с нулями, еденицами, нулями, еденицами, чередуя все время; тем временем, раз в секунду, обработчик сигнала таймера печатает текущее содержимое. (Вызов printf в обработчике безопасен в этой программе, потому что она конечно не вызывается снаружи обработчика, когда случается сигнал.)

Ясно, эта программа может печатать пару нулей или пару едениц.

Но это - не все что она может делать! На большинстве машин сохранение нового значения в памяти занимает несколько команд, и значение сохраняется по одному слову. Если сигнал передан между этими командами, обработчик, может находить, что memory.a - нуль, а memory.b - один (или наоборот).

На некоторых машинах может быть возможно сохранить новое значение в памяти только одной командой, которая не может быть прервана. На этих машинах, обработчик будет всегда печатать два нуля или две еденицы.

21.4.7.2 Типы Данных, к которым Быстрый Доступ

Чтобы избегать неопределенности относительно прерывания доступа к переменной, Вы можете использовать специфический тип данных, для которого доступ является всегда быстрым: sig_atomic_t. При чтении и запииси этого типа данных, как гарантируется, выполняется одиночная команда, так что не имеется никакого способа для обработчика, чтобы выполниться "в середине " доступа.

Тип sig_atomic_t - всегда целочисленный тип данных, но сколько битов он содержит, может изменяться от машины до машины.

sig_atomic_t

Это - целочисленный тип данных. К объектам этого типа всегда обращаются быстро.

Практически, Вы можете считать, что int и другие целочисленные типы не большие, чем int - быстрые.

Вы можете также считать, что к типам pointer быстрый доступ; это очень удобно. Оба этих утверждения верны для всех машин, которые поддерживает библиотека GNU C, и на всех системах POSIX, о которых мы знаем.

21.4.7.3 Быстрое Использование Шаблонов

Некоторые шаблоны доступа избегают любой проблемы, даже если доступ прерван. Например, флаг, который установлен обработчиком, и проверяется и очищается программой main время от времени, всегда безопасен, даже если доступ к нему фактически требует двух команд. Чтобы показать, что это так, мы должны рассмотреть каждый доступ, который мог быть прерван, и показывать, что не имеется никакой проблемы, если он прерван.

Прерывание в середине тестирования флага безопасно, потому что

либо он распознан отличным от нуля, тогда точное значение не важно, либо он будет отличным от нуля, в следующий раз.

Прерывание в середине очистки флага не проблема, потому что либо программа записывает нуль, если сигнал входит прежде, чем флаг очищен, либо обрабтчик заканчивается, и последующие события происходят при флаге отличном от нуля.

Если код обрабатывает оба этих случая правильно, то он может также обрабатывать сигнал в середине очистки флага.

Иногда Вы можете обеспечивать непрерывный доступ к одному объекту, защищая его использование другим объектом, возможно тем, чей тип гарантирует быстроту. См. Раздел 21.4.5 [Обьединенные Сигналы].

21.5 Примитивы, прерванные Сигналами

Сигнал может прибывать и обрабатываться, в то время как примитив ввода - вывода типа open или read ждет устройство ввода - вывода. Если обработчик сигнала возвращается, система задает вопрос: что должно случиться затем?

POSIX определяет один подход: делайте примитивный сбой сразу же.

Код ошибки для этого вида отказа - EINTR. Это гибко, но обычно неудобно. Обычно, приложения POSIX, которые используют обработчики сигнала, должны проверить EINTR после каждой библиотечной функции, которая может возвращать его, чтобы пытаться обратиться снова. Часто программисты забывают это проверять, что является общим источником ошибок.

Библиотека GNU обеспечивает удобный способ повторить обращение после временного отказа макркомандой TEMP_FAILURE_RETRY:

TEMP_FAILURE_RETRY (expression) (макрос)

Эта макркоманда оценивает выражение один раз. Если оно терпит неудачу и код ошибки EINTR, TEMP_FAILURE_RETRY оценивает это снова, и много раз, пока результат не временный отказ.

Значение, возвращенное TEMP_FAILURE_RETRY - любое произведенное выражением значение.

BSD избегает EINTR полностью и обеспечивает более удобный подход: перезапускать прерванный примитив. Если Вы выбираете этот подход, Вы не нуждаетесь в EINTR.

Вы можете выбирать любой подход в библиотеке GNU. Если Вы используете sigaction, чтобы установить обработчик сигнала, Вы

можете определять, как этот обработчик должен вести себя. Если Вы определяете SA_RESTART флаг, после возврата из этого обработчика продолжится примитив; иначе, возврат из этого обработчика вызовет EINTR. См. Раздел 21.3.5 [Флаги для Sigaction].

Другой способ определять выбор - siginterrupt функцией. См. Раздел 21.9.1 [POSIX против BSD].

Когда Вы не определяете с sigaction или siginterrupt, что специфический обработчик должен делать, он использует заданный по умолчанию выбор. Заданный по умолчанию выбор в библиотеке GNU зависит от возможностей макркоманд, которые Вы определили. Если Вы определяете _BSD_SOURCE или _GNU_SOURCE перед вызовом сигнала, значение по умолчанию - продолжить примитивы; иначе, значение по умолчанию должно делать сбой с EINTR. (Библиотека содержит альтернативные версии функции signal, и макркоманды возможностей определяют, которую Вы действительно вызываете.) См. Раздел 1.3.4 [Макрокоманды Возможностей].

Примитивы, на которые воздействует это: close, fcntl (операция F_SETLK), open, read, recv, recvfrom, select, send, sendto, tcdrain, waitpid, wait, и write.

Имеется одна ситуация, где возобновление никогда не случается, независимо от того, какой выбор Вы делаете: когда функция преобразования типа например read или write прервана сигналом после пересылки части данных. В этом случае, функция возвращает число байтов, уже перемещенных, указывая частичный успех.

Это может вызвать ненадежное поведение на устройствах для записи (включая датаграммный сокет; см. Раздел 11.9 [Датаграммы]), при разбивании одного чтения или записи на два чтения или две записи для двух единиц. Фактически, не имеется никакой проблемы, потому что прерывание после частичной передачи не может случаться на таких устройствах; они всегда передают всю запись в одном пакете, без ожидания, если только передача данных началась.

21.6 Сигналы Производства

Кроме сигналов, которые сгенерированы в результате аппаратной проверки или прерывания, ваша программа может явно посылать сигналы себе или другому процессу.

21.6.1 Передача Сигналов Самому себе

Процесс может посылать себе сигнал функцией raise. Эта функция объявлена в "signal.h".

int raise (int signum)

Функция raise посылает сигнал процессу вызова. Она возвращает нуль, если она успешна и значение отличное от нуля, если она терпит неудачу. Единственная причина для отказа - если значение signum недопустимо.

int gsignal (int signum)

Функция gsignal функция делает то же самое, что и raise; она нужна только для совместимости с SVID.

Удобное использование raise - воспроизвести заданное по умолчанию поведение сигнала, который Вы обработали. Например, предположите, что пользователь вашей программы печатает символ SUSP (обычно C-z; см. Раздел 12.4.9 [Специальные Символы]) чтобы послать интерактивный сигнал останова (SIGTSTP), и Вы хотите очистить некоторые внутренние буфера данных перед остановкой. Вы можете сделать это примерно так:

```
#include
void
tstp_handler (int sig)
{
        signal (SIGTSTP, SIG_DFL);
        . . .
        raise (SIGTSTP);
}
void
cont_handler (int sig)
{
        signal (SIGCONT, cont_handler);
        signal (SIGTSTP, tstp_handler);
}
int
main (void)
{
        signal (SIGCONT, cont_handler);
        signal (SIGTSTP, tstp_handler);

       - 467 -

        . . .
}
```

Примечание о переносимости: raise произошла из ANSI C. Более старые системы не могут поддерживать ее, так что ее использование уничтожает возможность переноса. См. Раздел 21.6.2 [Передача сигналов Другомму Процессу].

21.6.2 Передача сигналов Другому Процессу

Функция kill может использоваться, чтобы послать сигнал другому процессу. Несмотря на имя, она может использоваться для множества вещей отличных от завершения процесса. Вот некоторые примеры ситуаций, где Вы могли бы хотеть посылать сигналы между процессами:

* Родительский процесс назначает дочернему выполнять задачу, возможно имеющую бесконечный цикл, и завершает дочерний процесс, когда задача больше ненужна.
* Процесс выполняется как часть группы, и должен завершать или информировать другие процессы в группе, когда ошибка или другое событие происходит.
* Два процесса должны синхронизироваться при работе вместе.

Функция kill объявлена в " signal.h ".

int kill (pid_t pid, int signum) (функция)

Функция kill посылает сигнал signum процессу или группе процесса, заданной pid. Кроме сигналов, перечисленных в Разделе 21.2 [Стандартные Сигналы], signum может также иметь нулевое значение, чтобы проверить правильность pid.

Pid определяет процесс или группу процесса, как получателя сигнала:

Pid > 0 Процесс, чей идентификатор - pid.

Pid == 0 Все процессы в той же самой группе что и отправитель. Отправитель непосредственно не получает сигнал.

Pid < -1 Группа процесса, чей идентификатор есть -pid.

Pid == -1 Если процесс привилегирован, посылает сигнал всем процессам, кроме некоторых специальных процессов системы. Иначе, посылает сигнал всем процессам с тем же самым эффективным ID пользователя.

Процесс может посылать сигнала себе обращением kill (getpid (), signum).

Если kill используется процессом, чтобы послать сигнал себе, и

сигнал не блокирован, то kill позволяет этому процессу принять по крайней мере один сигнал (который мог некоторым другим задержанным сигналом вместо сигнала signum) прежде чем он возвращается.

Возвращаемое значение из kill - нуль, если сигнал может быть послан успешно. Иначе, никакой сигнал не послан, и возвращается

значение -1. Если pid определяет посылку сигнала отдельным процессам, kill успешно завершается, если он может посылать сигнал по крайней мере одному из них. Не имеется никакого способа, которым Вы можете узнать, который из процессов получил сигнал или что все они получили его.

Следующие errno условия ошибки определены для этой функции:

EINVAL аргумент signum - недопустимое или неподдерживаемое число.

EPERM Вы не имеете привилегий, чтобы послать сигнал процессу или любому из процессов в группе процесса, именованной pid.

ESCRH pid аргумент не относится к существующему процессу или группе.

int killpg (int pgid, int signum) (функция)

Подобна kill, но посылает сигнала группе процесса pgid. Эта функция предусмотрена для совместимости с BSD.

Как простой пример kill, обращение kill (getpid (), sig) имеет тот же самый эффект как raise (sig).

21.6.3 Права для использования kill

Имеются ограничения, которые запрещают Вам использовать kill, чтобы послать сигнал любому процессу.

Они предназначены, чтобы предотвратить антиобщественное поведение типа произвольного уничтожения процессов, принадлежащих другому пользователю. Обычно, kill используется, чтобы передать сигналы между родителем и дочерним процессами, или процессами братьями, и в этих ситуациях Вы обычно имеете право послать сигнал. Единственное общее исключение - когда Вы выполняете программу setuid на дочернем процессе; если программа изменяет реальный UID также как эффективный UID, Вы не можете иметь право, чтобы послать сигнал. Программа su делает это.

Имеет ли процесс право, чтобы послать сигнал другому процессу, определяется пользовательскими ID этих двух процессов. Это понятие обсуждено подробно в Разделе 25.2 [Владелец Процесса].

Вообще, чтобы можно было посылать сигнал другому процессу, посылающий процесс должен принадлежать привилегированному пользователю (подобно " root "), или реальный, или эффективный пользовательский ID процесса посылки должен соответствовать реальному, или эффективному пользовательскому ID процесса получения. В некоторых реализациях, родительский процесс способен послать сигнал дочернему процессу, даже если пользовательские ID не соответствуют. Другие реализации, могут предписывать другие ограничения.

Сигнал SIGCONT - частный случай. Он может быть послан, если отправитель - часть того же самого сеанса что и получатель, независимо от пользовательских ID.

21.6.4 Использование kill для Связи

Вот более длинный пример, показывающий, как сигналы могут использоваться для межпроцессорной связи. Это то, для чего предусмотрены сигналы SIGUSR1 и SIGUSR2. Так как эти сигналы фатальны по умолчанию, процесс, который, как предполагается, получает их, должен обрабатывать их через signal или sigaction.

В этом примере, родительского процесс порождает дочерний процесс и ждет пока дочерний завершит инициализацию. Дочерний процесс сообщает родителю, когда он готов, посылая ему сигнал SIGUSR1, используя функцию kill.

```
#include
#include
#include
#include
volatile sig_atomic_t usr_interrupt = 0;
void
synch_signal (int sig)
{
        usr_interrupt = 1;
}
/* Дочерний процесс выполняет эту функцию. */
void
child_function (void)
{
        printf ("I'm here!!!  My pid is %d.\n",
```

```
                        (int) getpid ());
                kill (getppid (), SIGUSR1);
                puts ("Bye, now....");
                exit (0);
        }
        int
        main (void)
        {
                struct sigaction usr_action;
                sigset_t block_mask;
                pid_t child_id;
                sigfillset (&block_mask);
                usr_action.sa_handler = synch_signal;
                usr_action.sa_mask = block_mask;
                usr_action.sa_flags = 0;
                sigaction (SIGUSR1, &usr_action, NULL);
        /* Создание дочернего процесса. */
                child_id = fork ();
                if (child_id == 0)
                        child_function ();
                while (!usr_interrupt)
                        ;
                puts ("That's all, folks!");
                return 0;
        }
```

Этот пример использует активное ожидание, которое является плохим, потому что это потеря времени CPU, которое другие программы могли бы иначе использовать. Лучше просить, чтобы система ждала, пока сигнал не прибывает. См. пример в Разделе 21.8 [Ожидание Сигнала].

21.7 Блокированные Сигналы

Блокирование сигнала означает сообщение операционной системе, чтобы задержать его и передать его позже. Вообще, программа просто так не блокирует сигналы, она может также игнорировать их, устанавливая их действия как SIG_IGN. Но полезно блокировать сигналы ненадолго, чтобы предотвратить прерывание чувствительных операций. Например:

* Вы можете использовать функцию sigprocmask, чтобы блокировать сигналы, в то время как Вы изменяете глобальные переменные, которые также изменяются обработчиками для этих сигналов.
* Вы можете устанавливать sa_mask в вашем обращении к sigaction, чтобы блокировать некоторые сигналы, в то время как выполняется специфический обработчик сигнала. Этим способом, обработчик сигнала может выполняться без того, чтобы прерваться сигналами.

21.7.1 Почему Полезно Блокированиие Сигналов

Временное блокирование сигналов с sigprocmask дает Вам возможность предотвратить прерывания при выполнении критических частей вашего кода. Если сигналы прибывают в эту часть программы, они будут переданы позже, после того, как Вы откроете их.

Например, это полезно - для совместного использования данных обработчиком сигнала и остальной частью программы. Если тип данных - не sig_atomic_t (см. Раздел 21.4.7 [Быстрый Доступ к данным]), то обработчик сигнала может выполняться, когда остальная часть программы закончила только половину чтения или запииси данных.

Чтобы делать программу надежной, Вы можете предотвращать выполнение обработчика сигнала, в то время как остальная часть программы исследует или изменяет эти данные, блокировав соответствующий сигнал в частях программы, которые касаются данных.

Блокирование сигналов также необходимо, когда Вы хотите выполнить некоторое действие только, если сигнал не прибыл. Предположите, что обработчик для сигнала устанавливает флаг типа sig_atomic_t; Вы хотели бы проверить флаг и выполнить действие, если флаг не установлен. Это ненадежно. Предположите, что сигнал - передан немедленно после того, как Вы проверяете флаг, но перед последовательным действием: тогда программа выполнит действие, даже если сигнал прибыл.

Единственый способ проверять, надежно ли прибыл ли сигнал, состоит в

том, чтобы проверять это в то время, когда сигнал блокирован.

21.7.2 Наборы Сигналов

Все функции блокирования сигнала используют структуру данных называемую набором сигналов, чтобы определить на какие сигналы воздействовать. Таким образом, каждое действие включает две стадии:

создание набора сигналов, и передача его как аргумента библиотечной функции.

Эти средства объявлены в заглавном файле " signal.h ".

sigset_t

Тип данных sigset_t используется, чтобы представить набор сигналов. Внутренне, он может быть выполнен как целый или как структурный тип.

Для переносимости, чтобы инициализировать, изменять и читать информацию из объектов sigset_t, используйте только функции, описанные в этом разделе, не пробуйте манипулировать ими непосредственно.

Имеются два способа инициализировать набор сигналов. Вы можете первоначально определить его пустыми через sigemptyset и затем добавлять заданные сигналы индивидуально. Или Вы можете определить его полными через sigfillset и тогда удалять заданные сигналы индивидуально.

Вы должны всегда инициализировать набор сигналов одной из этих двух функций перед использованием его любым другим способом. Не пробуйте устанавливать все сигналы явно, потому что объект sigset_t может включать некоторую другую информацию (подобно полю версии) которое должно быть инициализировано также.

int sigemptyset (sigset_t *set)

Эта функция инициализирует набор наборов сигналов, чтобы исключить все определенные сигналы. Она всегда возвращает 0.

int sigfillset (sigset_t *set)

Эта функция инициализирует набор наборов сигналов, чтобы включить все определенные сигналы. Снова, возвращаемое значение - 0.

int sigaddset (sigset_t *set, int signum)

Эта функция добавляет сигнал signum к набору наборов сигналов. Все что она делает - изменяет набор; она не блокирует или не открывает никаких сигналов.

Возвращаемое значение - 0 при успехе и -1 при отказе. Следующее errno - условие ошибки определено для этой функции:

EINVAL аргумент знака не определяет допустимый сигнал.

int sigdelset (sigset_t *set, int signum)

Эта функция удаляет сигнал signum из набора сигналов. Возвращаемое значение и условия ошибки - такие же как для sigaddset.

В заключение, имеется функция, чтобы проверить то, что сигналы находятся в наборе сигналов:

int sigismember (const sigset_t *set, int signum)

Функция sigismember проверяет, является ли сигнал signum элементом набора сигналов. Она возвращает 1, если сигнал находится в наборе, 0 если нет, и -1, если имеется ошибка.

Следующее errno условие ошибки определено для этой функции:

EINVAL аргумент signum не определяет допустимый сигнал.

21.7.3 Маска Сигналов Процесса

Набор сигналов, которые в настоящее время блокированы, называется маской сигналов. Каждый процесс имеет собственную маску сигналов. Когда Вы создаете новый процесс (см. Раздел 23.4 [Создание Процесса]), он наследует маску родителя. Вы можете блокировать или открывать сигналы с большей гибкостью, изменяя маску сигналов.

Прототип для sigprocmask функции находится в " signal.h ".

int sigprocmask (int how, const sigset_t *set, sigset_t *oldset)

Функция Sigprocmask используется, чтобы исследовать или изменять маску сигналов процесса, аргумент how определяет, как изменяется маска сигналов, и должен быть одним из следующих значений:

SIG_BLOCK

Блокирует сигналы в set, и добавляет их к существующей маске. Другими словами, новая маска - объединение существующей маски и set.

```
                SIG_UNBLOCK
        Открывает сигналы в set и удаляет их из существующей маски.
                SIG_SETMASK
        Использует set для маски; игнорируя предыдущее значение маски.
        Последний аргумент, oldset, используется, чтобы возвратить
информацию относительно старой маски сигналов процесса. Если Вы
хотите только изменять маску без того, чтобы рассмотривать ее,
передавайте пустой указатель как oldset аргумент. Аналогично, если
Вы хотите знать что находится в маске, без того, чтобы заменить ее,
передайте пустой указатель для set. Oldset аргумент часто
используется, чтобы запомнить предыдущую маску сигналов, чтобы
восстановить ее позже.
        Если вызов sigprocmask открывает любые отложенные сигналы, то по

                                - 474 -

крайней мере один из этих сигналов будет передан процессу прежде,
чем sigprocmask возвртится. Порядок, в котором передаются
отложенные сигналы является не определенным, но Вы можете управлять
порядком явно, делая несколько обращений к sigprocmask, чтобы
открывать различные сигналы по одному.
        Sigprocmask функция возвращает 0, если она успешна, и -1, в противном
случае. Следующие errno условия ошибки определены для этой
функции:
        EINVAL Аргумент how недопустим.
        Вы не можете блокировать SIGKILL и  SIGSTOP, но если набор
сигналов включает их, то sigprocmask только игнорируют их вместо
того, чтобы возвратить состояние ошибки.

        21.7.4 Блокирование для Проверки Наличия Сигнала
        Вот простой пример. Предположите, что Вы устанавливаете
обработчик для сигналов SIGALRM, который устанавливает флаг всякий
раз, когда прибывает сигнал, и ваша программа main проверяет этот
флаг время от времени и сбрасывает его. Вы можете предотвращать
прибытие дополнительных сигналов SIGALRM в неподходящее время,
ограничивая критическую часть кода обращениями к sigprocmask,
примерно так:
                sig_atomic_t flag = 0;
                int
                main (void)
                {
                        sigset_t block_alarm;
                        . . .
                        sigemptyset (&block_alarm);
                        sigaddset (&block_alarm, SIGALRM);
                        while (1)
                        {
                                sigprocmask (SIG_BLOCK, &block_alarm,
                                             NULL);
                                if (flag)
                                {
                                        actions-if-not-arrived
                                        flag = 0;
                                }

                               - 475 -

                                sigprocmask (SIG_UNBLOCK, &block_alarm,
                                             NULL);
                                .       .       .
                        }
                }
```

21.7.5 Блокирование Сигналов для Обработчика

Когда обработчик сигнала вызывается, Вы обычно хотите, чтобы он закончился без прерываний другим сигналом. С момента старта обработчика до момента его окончания, Вы должны блокировать сигналы, которые могли бы запутать его или разрушить данные.

Когда функция обработчика вызывается для сигнала, этот сигнал автоматически блокируется (в дополнение к любым другим сигналам,

которые являются уже в маске сигналов процесса) пока обработчик выполняется. Если Вы устанавливаете обработчик для SIGTSTP, например, то поступление этого сигнала, вынуждает дальнейшие SIGTSTP сигналы ждать в течение выполнения обработчика.

Однако, по умолчанию, другие виды сигналов не блокированы; они могут прибывать в течение выполнения обработчика.

Надежный способ блокировать другие виды сигналов в течение выполнения обработчика состоит в том, чтобы использовать sa_mask элемент структуры sigaction.

Вот пример:

```
#include
#include
void catch_stop ();
void
install_handler (void)
{
        struct sigaction setup_action;
        sigset_t block_mask;
        sigemptyset (&block_mask);
        sigaddset (&block_mask, SIGINT);

                - 476 -

        sigaddset (&block_mask, SIGQUIT);
        setup_action.sa_handler = catch_stop;
        setup_action.sa_mask = block_mask;
        setup_action.sa_flags = 0;
        sigaction (SIGTSTP, &setup_action, NULL);
}
```

Это более надежно чем блокирование других сигналов явно в коде обработчика. Если Вы блокируете сигналы в обработчике, Вы не может избежать по крайней мере короткого интервала в начале обработчика, где они еще не блокированы.

Вы не можете удалять сигналы из текущей маски процесса, используя этот механизм. Однако, Вы можете делать обращения к sigprocmask внутри вашего обработчика, чтобы блокировать или открыть сигналы, как Вы желаете.

В любом случае, когда обработчик возвращается, система восстанавливает маску, которая была до обработчика.

21.7.6 Прверка Отложенных Сигналов

Вы можете выяснять, какие сигналы отложены в любое время, вызывая sigpending. Эта функция объявлена в " signal.h ".

int sigpending (sigset_t *set) (функция)

Sigpending функция сохраняет информацию относительно отложенных сигналов в set. Если там - отложенный сигнал, который блокирован, то этот сигнал - элемент возвращенного set. (Вы можете проверять является ли специфический сигнал элемент этого set, использующего sigismember; см. Раздел 21.7.2 [Наборы Сигналов].)

Возвращаемое значение - 0 при успехе, и -1 при отказе.

Тестирование отложен ли сигнал полезно не часто. Тестирование сигнала который не блокирован - почти всегда бессмысленно.

Вот пример.

```
#include
#include
sigset_t base_mask, waiting_mask;
sigemptyset (&base_mask);
sigaddset (&base_mask, SIGINT);
sigaddset (&base_mask, SIGTSTP);
sigprocmask (SIG_SETMASK, &base_mask, NULL);

                - 477 -

. . .
sigpending (&waiting_mask);
if (sigismember (&waiting_mask, SIGINT)) {
 /* Пользователь пробовал уничтожать процесс. */
}
else if (sigismember (&waiting_mask, SIGTSTP)) {
 /*Пользователь пробовал остановить процесс.*/
}
```

Не забудьте, что, если имеется задержка некоторого сигнала для вашего процесса, дополнительные сигналы этого же самого типа могут быть отброшены. Например, если сигнал SIGINT отложен, когда прибывает

другой сигнал SIGINT, ваша программа будет возможно видеть только один из них, когда Вы откроете этот сигнал.

Примечание Переносимости: функция sigpending новая в POSIX.1. Более старые системы не имеют никакого эквивалентного средства.

21.7.7 Запоминание Сигнала, для отложенного вызова

Вместо того, чтобы блокировать сигнал используя библиотечные средства, Вы можете получить почти те же самые результаты, делая так чтобы обработчик устанавливал флаг, который будет проверен позже, когда Вы "откроете". Вот пример:

```
volatile sig_atomic_t signal_pending;
volatile sig_atomic_t defer_signal;
void
handler (int signum)
{
        if (defer_signal)
                signal_pending = signum;
        else
         ... /*"Действительно" обрабатываем сигнал.*/
}
. . .
void
update_mumble (int frob)
{
```



```
        defer_signal++;
        mumble.a = 1;
        mumble.b = hack ();
        mumble.c = frob;
        defer_signal--;
        if (defer_signal == 0 && signal_pending != 0)
                raise (signal_pending);
}
```

Обратите внимание, как специфический сигнал сохранен в signal_pending. Этим способом, мы можем обрабатывать несколько типов неудобных сигналов.

Мы увеличиваем и уменьшаем defer_signal так, чтобы вложенные критические разделы работали правильно; таким образом, если update_mumble вызывалась с signal_pending, уже отличным от нуля, сигналы будут отсрочены не только внутри update_mumble, но также внутри вызывающего оператора. Вот почему мы не проверяем signal_pending, если defer_signal все еще отличен от нуля.

Приращение и уменьшение defer_signal требует больше чем одну команду; и возможно сигнал случиться в середине. Но это не вызывает никакой проблемы. Если сигнал случается достаточно рано чтобы увидеть значение до приращения или уменьшения, то это эквивалентно сигналу который, пришел перед началом приращения или уменьшения, что является случаем который работает правильно.

Абсолютно необходимо увеличить defer_signal перед тестированием signal_pending, потому что это позволяет избежать тонкой ошибки. Если бы мы делали это в другом порядке, примерно так,

```
if (defer_signal == 1 && signal_pending != 0)
        raise (signal_pending);
defer_signal--;
```

то сигнал, прибывающий между условным оператором и оператором уменьшения был бы эффективно "потерян" на неопределенное количество времени. Обработчик просто установил бы defer_signal, но программа, уже проверявшая эту переменную, не будет проверять переменную снова.

Ошибки подобно этим, называются ошибками синхронизации. Они - особенно опасны, потому что они случаются редко и их почти невозможны воспроизвести. Вы не сможете найти их отладчиком, как Вы нашли бы воспроизводимую ошибку. Так что надо быть особенно осторожным, чтобы избежать их.

21.8 Ожидание Сигнала

Если ваша программа управляется внешними событиями, или

используует сигналы для синхронизации, то она должена возможно ждать, пока сигнал не прибудет.

21.8.1 Использование pause

Простой способ ждать прибытия сигнала - вызвать pause.

int pause () (функция)

Функция pause приостанавливает выполнение программы, пока не прибывает сигнал, чье действие должно также выполнить функцию обработчика, или завершить процесс.

Если сигнал выполняет функцию обработчика, то pause возвращается. Это рассматривается как неудача (так как "успешное" поведение должно было бы приостановить программу навсегда), так что возвращаемое значение -1. Даже если Вы определяете, что другие примитивы должны продолжиться, когда обработчик системы возвращается (см. Раздел 21.5 [Прерванные Примитивы]), это не имеет никакого эффекта на pause; она всегда терпит неудачу, когда сигнал обработан.

Следующие errno условия ошибки определены для этой функции:

EINTR функция была прервана сигналом.

Если сигнал вызывает окончание программы, pause не возвращается (очевидно).

Функция pause объявлена в " unistd.h ".

21.8.2 Проблемы с pause

Простота pause может скрывать серьезные ошибки синхронизации, которые могут привести программу к зависанию.

Безопасно использовать pause, если реальная работа вашей программы выполняется обработчиками сигнала непосредственно, а программа не делает ничего кроме обращения к pause. Каждый сигнал будет заставлять обработчик делать следующий пакет работы, которая должна быть выполнена, и возвращаться, так чтобы цикл программы мог вызывать pause снова.

Вы не можете безопасно использовать pause, чтобы ждать, пока не

прибудет еще один сигнал, и тогда продолжить реальную работу.

Даже если Вы принимаете меры, чтобы обработчик сигнала сотрудничал, устанавливая флаг, Вы все еще не можете использовать pause надежно. Вот пример такой проблемы:

```
if (!usr_interrupt)
pause ();
/* работа, после прибытия сигнала. * /
. . .
```

Она имеет ошибку: сигнал может прибывать после того, как переменная usr_interrupt проверена, но перед обращением к pause. Если никакие дальнейшие сигналы не прибывают, процесс никогда не выполнится снова.

Вы можете изменять верхнее ограничение ожидания, используя sleep в цикле, вместо того чтобы использовать pause. (См. Раздел 17.4 [Бездействие].) Вот, на что это походит:

```
while (!usr_interrupt)
        sleep (1);
/* работа, после прибытия сигнала. */
. . .
```

Для некоторых целей это достаточно удобно. Но немного более сложно. Вы можете ждать, пока специфический обработчик сигнала не выполнен, надежно, используя sigsuspend.

21.8.3 Использование sigsuspend

Чистый и надежный способ ждать сигнал состоит в том, чтобы блокировать его и тогда использовать sigsuspend.

Используя sigsuspend в цикле, Вы можете ждать некоторые виды сигналов, разрешая другим видам сигналов обрабатываться их обработчиками.

int sigsuspend (const sigset_t *set) (функция)

Эта функция заменяет маску сигналов процесса на set и тогда приостанавливает процесс, пока не передан сигнал, чье действие должно завершать процесс или вызывать функцию обработки сигнала. Другими словами, программа действительно будет приостановлена, пока один из сигналов, который - не элемент set, не прибудет.

Если процесс пробужден сигналом, который вызывает функцию обработчика, и функция обработчика возвращается, то sigsuspend также

возвращается.
Маска остается set только, пока sigsuspend ждет. Функция sigsuspend всегда восстанавливает предыдущую маску сигналов, когда она возвращается.
Возвращаемое значение и условия ошибки - такие же как для pause.
С sigsuspend, Вы можете заменять pause или цикл sleep в предыдущем разделе кое-чем полностью надежным:

```
sigset_t mask, oldmask;
. . .
sigemptyset (&mask);
sigaddset (&mask, SIGUSR1);
. . .
/ * Ждем получения сигнала. * /
sigprocmask (SIG_BLOCK, &mask, &oldmask);
while (!usr_interrupt)
        sigsuspend (&oldmask);
sigprocmask (SIG_UNBLOCK, &mask, NULL);
```

Этот последний фрагмент кода немного сложен. Отметте, что когда sigsuspend возвращается, она сбрасывает маску сигналов процесса к первоначальному значению, в этом случае сигнал SIGUSR1 еще раз блокирован. Второе обращение к sigprocmask необходимо чтобы явно открыть этот сигнал.

21.9 BSD Обработка Сигнала

Этот раздел описывает альтернативные функции обработки сигнала, происходящие от UNIX BSD. Эти средства были современными, в их время; сегодня, они обычно устаревшие, и обеспечены в основном для совместимости с UNIX BSD.
Они обеспечивают одну возможность, которая не доступна через функции POSIX: Вы можете определять отдельный стек для использования в некоторых обработчиках сигнала. Использование стека сигнала - единственый способ, которым Вы можете обрабатывать сигнал, вызванный переполнением стека.

21.9.1 POSIX и BSD Средства Обработки Сигналов

Имеются много подобий между BSD и POSIX средствми обрабатывающими сигналы, потому что средства POSIX были вдохновлены средствами BSD. Кроме наличия различных имен для всех функций, чтобы избежать конфликтов, есть несколько основных различий:
* UNIX BSD представляет маски сигналов как int битовая маска, а не как объект sigset_t.
* Средства BSD используют отличное значение по умолчанию для проверки, должен ли прерванный примитив терпеть неудачу или нет. Средства POSIX делают сбой системных вызовов, если Вы не определяете, что они должны продолжиться. Со средством BSD, значение по умолчанию не должно делать сбой системных вызовов, если Вы не говорите, что они должны терпеть неудачу. См. Раздел 21.5 [Прерванные Примитивы].
* UNIX BSD имеет понятие стека сигналов. Это - альтернативный стек, который используется в течение выполнения функций обработчика сигнала, вместо нормального стека выполнения.
Средства BSD объявлены в " signal.h ".

21.10 Функция BSD, чтобы Установить Обработчик

struct sigvec (тип данных)
Этот тип данных - эквивалент BSD struct sigaction (см. Раздел 21.3.2 [Сложная Обработка Сигнала]); он используется, чтобы определить действия сигнала для sigvec функции. Он содержит следующие элементы:

sighandler_t sv_handler
Это - функция обработчика.

int sv_mask
Это - маска дополнительных сигналов, которые будут блокированы,

в то время как функция обработчика вызывается.

int sv_flags

Это - битовая маска, используемая, чтобы определить различные флаги, которые воздействуют на поведение сигнала. Вы можете также обратиться к этому полю как sv_onstack.

Эти символические константы могут использоваться, чтобы обеспечить значения для sv_flags поля структуры sigvec. Это поле - значение

битовой маски,следовательно Вам необходимо слить флаги, представляющие интерес для Вас вместе через OR.

int SV_ONSTACK

Если этот бит установлен в sv_flags поле структуры sigvec, это означает - использовать стек сигнала при получении сигнала.

int SV_INTERRUPT (макрос)

Если этот бит установлен в sv_flags поле структуры sigvec, это означает что, системные вызовы, прерванные этим видом сигнала не должны быть перезапущены, если обработчик возвращается; взамен, системные вызовы должны возвратиться с EINTR состоянием ошибки. См. Раздел 21.5 [Прерванные Примитивы].

int SV_RESETHAND (макрос)

Если этот бит усткновлен в sv_flags поле структуры sigvec, это означает - сбросить действие для сигнала обратно к SIG_DFL, когда сигнал получен.

int sigvec (int signum, const struct sigvec *action,struct sigvec *old_action)

Эта функция - эквивалент sigaction; она устанавливает действие для сигнала signum, возвращая информацию относительно предыдущего действия для этого сигнала в old_action.

int siginterrupt (int signum, int failflag) (функция)

Эта функция определяет, что использовать, когда некоторые примитивы прерваны обрабаткой сигнала signum. Если failflag - ложь, то примитивы рестартуют после сигнала. Если failflag - истина, обработка signum заставляет эти примитивы терпеть неудачу с кодом ошибки EINTR. См. Раздел 21.5 [Прерванные Примитивы].

21.10.1 Функции BSD для Блокирования Сигналов

int sigmask (int signum) (макрос)

Эта макркоманда возвращает маску сигналов, которая имеет бит для установки сигнала signum. Вы можете слить через OR результаты отдельных обращений к sigmask вместе, чтобы определять больше чем один сигнал. Например,

```
(sigmask (SIGTSTP) | sigmask (SIGSTOP)
        | sigmask (SIGTTIN) | sigmask (SIGTTOU))
```

определяет маску, которая включает все сигналы останова управления заданиями.

int sigblock (int mask) (функция)

Эта функция эквивалентна sigprocmask (см. Раздел 21.7.3 [Маска сигналов Процесса]) с аргументом how - SIG_BLOCK: она добавляет сигналы, заданные маской к набору блокированных сигналов процесса вызова. Возвращаемое значение - предыдущий набор блокированных сигналов.

int sigsetmask (int mask) (функция)

Это эквивалент функции sigprocmask (см. Раздел 21.7.3 [Маска сигналов Процесса]) с аргументом how - SIG_SETMASK: она устанавливает маску сигналов вызывающего процесса как mask. Возвращаемое значение - предыдущий набор блокированных сигналов.

int sigpause (int mask) (функция)

Эта функция - эквивалент sigsuspend (см. Раздел 21.8 [Ожидание Сигнала]): она устанавливает маску сигналов вызывающего процесса как mask, и ждет прибытия сигнала. Она при возвращаении восстанавливает предыдущий набор блокированных сигналов.

21.10.2 Использование Отдельного Стека Сигнала

Стек сигнала - специальная область памяти, которую нужно использовать как стек в течение выполнения обработчиков сигнала. Он должен быть довольно большим, чтобы избежать переполнения; макркоманда SIGSTKSZ определяет канонический размер для стеков сигналов. Вы можете использовать malloc, чтобы зарезервировать пространство для стека. Вызо-

вите sigaltstack или sigstack, чтобы система использовала это пространство для стека сигнала.

Вам не нужно писать обработчик сигнала по-другому чтобы использовать стек сигнала. Переключение одного стека на другой происходит автоматически. Однако, некоторые отладчики на некоторых машинах могут запутаться, если Вы исследуете след стека, в то время как обработчик, который использует стек сигнала, выполняется.

Имеются два интерфейса для сообщения системе использовать отдельный стек сигнала. Sigstack - более старый интерфейс, который исходит из 4.2 BSD. Sigaltstack - более новый интерфейс, и исходит из 4.4 BSD. Интерфейс sigaltstack имеет преимущество - не требуется, чтобы ваша программа знала в каком направлении растет стек, что зависит от специфической машины и операционной системы.

struct sigaltstack (тип данных)

Эта структура описывает стек сигнала. Она содержит следующие элементы:

void *ss_sp

Этим указываем на основание стека сигнала.

size_t ss_size

- размер (в байтах) стека сигнала, на который указывает "ss_sp ". Вы должны установить здесь - сколько места Вы зарезервировали для стека.

Есть две макркоманды, определенные в " signal.h " которые Вы должны использовать в вычислении этого размера:

SIGSTKSZ

- каноническиий размер для стека сигнала. Он должен быть достаточным для нормальных использований.

MINSIGSTKSZ

- количество пространства стека сигнала, нужное операционной системе только, чтобы выполнить сигнал. Размер стека сигнала должен быть больший чем этот.

Для большинства случаев SIGSTKSZ для ss_size достаточен. Но Вы можете захотеть использовать различный размер. В этом случае, Вы должны зарезервировать MINSIGSTKSZ дополнительных байт для стека сигнала и увеличивать ss_size.

int ss_flags

Это поле содержит поразрядное OR этих флагов:

SA_DISABLE

Сообщает системе, что она не должна использовать стек сигнала.

SA_ONSTACK

Устанавливается системой, и указывает, что стек сигнала использован в настоящее время.

int sigaltstack (const struct sigaltstack *stack, struct sigaltstack *oldstack) (функция)

Sigaltstack функция определяет альтернативный стек для использования в течение обработки сигнала.

Если oldstack - не пустой указатель, информация относительно в настоящее время установленного стека сигнала будет возвращена в расположение, на которое он указывает. Если stack - не пустой указатель, то

он будет установлен как новый стек для использования обработчиками сигнала.

Возвращаемое значение - 0 при успехе и -1 при отказе. Если sigaltstack сбоит, она устанавливает errno как одно из этих значений:

EINVAL

Вы пробовали отключать стек, который был фактически использован в настоящее время.

ENOMEM

Размер альтернативного стека был слишком мал. Он должен быть большее чем MINSIGSTKSZ.

Вот более старый интерфейс sigstack.

struct sigstack (тип данных)

Эта структура описывает стек сигнала. Она содержит следующие элементы:

void *ss_sp

- указатель вершины стека. Если стек растет вниз на вашей машине, он должен указывать на начало области, которую Вы

зарезервировали. Если стек растет вверх, он должен указывать на нижнюю часть.

int ss_onstack

Это поле истинно, если процесс в настоящее время использует этот стек.

int sigstack (const struct sigstack *stack, struct sigstack *oldstack) (функция)

Sigstack функция определяет альтернативный стек для использования в течение обработки сигнала.

Когда сигнал получен процессом, и стек сигнала используется, система переключается на в настоящее время установленный стек сигнала, в то время как выполняется обработчик для этого сигнала.

Если oldstack - не, пустой указатель, информация относительно в настоящее время установленного стека сигнала будет возвращена в расположение, на которое он указывает. Если stack - не пустой указатель, то он будет установлен как новый стек для использования обработчиками сигнала.

Возвращаемое значение - 0 при успехе и -1 при отказе.

22. Запуск и Окончание Процесса

Процессы - примитивные модули для распределения ресурсов системы. Каждый процесс имеет собственное адресное пространство. Процесс выполняет программу; Вы можете иметь многократные процессы, выполняющие ту же самую программу, но каждый процесс имеет собственную копию программы внутри собственного адресного пространства и выполняет ее независимо от других копий.

Эта глава объясняет, что ваша программа должна делать, чтобы обработать запуск процесса, завершить процесс, и получить информацию (аргументы и среду) из родительского процесса.

22.1 Аргументы Программы

Система начинает программу C, вызывая функцию main. Вы должны написать функцию, именованную main, иначе Вы не будете способны линковать вашу программу без ошибок.

Вы можете определять main без аргументов, или брать два аргумента, которые представляют аргументы командной строки программы, примерно так:

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

Аргументы командной строки - отделяемые пропуском лексемы, заданные в команде оболочки, используемой, чтобы вызвать программу; таким образом, в " cat foo bar ", аргументы - " foo " и " bar ". Программа может рассматривать аргументы командной строки единственым способом - через аргументы main.

Значение argc аргумента - число аргументов командной строки. Аргумент argv - вектор строк; элементы - индивидуальные строки аргументов командной строки. Имя файла выполняемой программы также включено в вектор как первый элемент; значение argc учитывает этот элемент. Пустой указатель всегда следует за последним элементом: argv [argc] - это пустой указатель.

Для команды " cat foo bar ", argc - 3, и argv имеет три элемента, " cat ", " foo " и " bar ".

Если синтаксис для аргументов командной строки вашей программы является достаточно простым, Вы может просто выбирать аргументы из argv вручную. Но если ваша программа берет фиксированное число

аргументов, или все аргументы интерпретируются одинаковым образом (как имена файлов, например), Вам лучше использовать getopt, чтобы делать синтаксический анализ.

22.1.1 Синтаксические Соглашения Аргументов Программы

POSIX рекомендует эти соглашения для аргументов командной строки. Getopt (см. Раздел 22.1.2 [Опции Синтаксического анализа]) облегчит их реализацию.

* Аргументы - опции, если они начинаются с разделителя дефиса ("

- ").
* За разделителем могут следовать много опций в одиночной лексеме, если опции не берут аргументов. Таким образом, " -abc " эквивалентно " -a -b -c ".
* Имена опций - одиночные алфавитно-цифровые символы (как для isalnum; см. Раздел 4.1 [Классификация Символов]).
* Некоторые опции требуют аргумента. Например, команда ` -o ' ld требует аргумент - имя выходного файла.
* Опция и ее аргумент могут не занимать отдельные лексемы. (Другими словами, пропуск, отделяющий их необязателен.) Таким образом, " -o foo " и " -ofoo " эквивалентны.
* Опции обычно предшествуют другим аргументам, не-опциям. команды
Реализация getopt в библиотеке GNU C обычно делает так, как будто все аргументы опции были определены перед всеми аргументами не-опциями для целей синтаксического анализа, даже если пользователь вашей программы смешал опции и аргументы не-опции. Она делает это, переупорядочивая элементы массива argv. Это поведение нестандартно; если Вы хотите подавлять его, определите _POSIX_OPTION_ORDER переменную среды. См. Раздел 22.2.2 [Стандартная Среда].
* Аргумент " -- " завершает все опции; все остальные аргументы обрабатываются как аргументы-не-опции, даже если они начинаются с дефиса.
* Лексема, состоящая из одиночного символа дефиса интерпретируется как обычный аргумент-не-опция. Обычно, она используется, чтобы определить ввод из или вывод в стандартный ввод и вывод.
* Опции могут быть обеспечены в любом порядке, или появляться многократно. Интерпретация оставлена до специфической прикладной

программы.
GNU добавляет длинные опции к этому соглашению. Длинные опции состоят из " -- " сопровождаемых именем, составленным из алфавитно-цифровых символов, и подчеркивания. Имена опций - обычно от одного до трех слов длинной, с дефисами, чтобы отделить слова. Пользователи могут сокращать имена опций, если только сокращения уникальны.
Пример длинной опции " --name=value ". Этот синтаксис дает возможность длинной опции принять аргумент, который является самостоятельно необязательным.
В конечном счете, система GNU будет обеспечивать длинные имена опций в оболочке.

22.1.2 Опции Программ Синтаксического анализа

Имеются подробности относительно того, как вызвать getopt функцию. Чтобы использовать это средство, ваша программа должна включить заглавный файл " unistd.h ".
int opterr (переменная)
Если значение этой переменной является отличным от нуля, то getopt, печатает сообщение об ошибках в стандартный поток ошибки, если она сталкивается с неизвестным символом опции или опцией с отсутствующим требуемым аргументом. Это - заданное по умолчанию поведение. Если Вы обнуляете эту переменную, getopt, не печатает никаких сообщений, но она все еще возвращает символ ? чтобы указывать ошибку.
int optopt (переменная)
Когда getopt сталкивается с неизвестным символом опции или опцией с отсутствующим требуемым аргументом, она сохраняет этот символ опции в этой переменной. Вы можете использовать ее для обеспечения ваших собственных диагностических сообщений.
int optind (переменная)
Эта переменная будет установлена getopt как индекс следующего элемента массива argv, который будет обработан. Если getopt нашла все аргументы-опции, Вы можете использовать эту переменную, чтобы определить, где начинаются остающиеся аргументы-не-опции. Начальное значение этой переменной 1.

char * optarg (переменная)
Эта переменная будет установлена getopt, чтобы указать число

аргументов опций, для тех опций которые принимают аргументы.

int getopt (int argc, char **argv, const char *options)
(функция)

Getopt функция получает следующий аргумент-опцию списка параметров, заданного argv и argc аргументами.

Аргумент-опция - строка, которая определяет символы опции, которые являются допустимыми для этой программы. Символ опции в этой строке может сопровождаться двоеточием (": ") чтобы указать, что она берет требуемый аргумент.

Если строка аргумента-опции начинается с дефиса (" - "), она обрабатывается особенно. Это разрешает аргументам-не-опциям, возвращаться, как будто они были связаны с последним символом опции.

Getopt функция возвращает символ опции для следующей опции командной строки. Когда нет больше аргументов-опций, она возвращает -1. Может все еще иметься большое количество аргументов-не-опций; Вы должны сравнить внешнюю переменную optind с параметром argc, чтобы проверить это.

Если опция имеет аргумент, getopt возвращает аргумент, сохраняя его в переменной optarg. Вы обычно не должны копировать optarg строку, так как это - указатель в первоначальный массив argv, а не в статическую область, которая могла бы быть перезаписана.

Если getopt находит символ опции в argv, который не был включен в опции, или отсутствующий аргумент некоторой опции, она возвращает "? ", устанавливает внешнюю переменную optopt как фактический символ опции. Если первый символ опции - двоеточие (":"), то getopt возвращает ":" вместо "? " Чтобы указать отсутствующий аргумент опции. Кроме того, если внешняя переменная opterr отлична от нуля (который является значением по умолчанию), getopt печатает сообщение об ошибках.

22.1.3 Пример Синтаксического Анализа Аргументов с getopt

Вот пример, показывающий, как getopt обычно используется:

```
#include
#include
int
main (int argc, char **argv)
{
        int aflag = 0;
        int bflag = 0;
        char *cvalue = NULL;
        int index;
        int c;
        opterr = 0;
        while ((c = getopt (argc, argv, "abc:")) != -1)
        switch (c)
        {
                case 'a':
                        aflag = 1;
                        break;
                case 'b':
                        bflag = 1;
                        break;
                case 'c':
                        cvalue = optarg;
                        break;
                case '?':
                        if (isprint (optopt))
                                fprintf (stderr, "Unknown option
                                        `-%c'.\n", optopt);
                        else
                                fprintf (stderr, "Unknown option
                        character `\\x%x'.\n", optopt);
                        return 1;
                default:
                        abort ();
```

```
                }

                                - 492 -

                printf ("aflag = %d, bflag = %d, cvalue = %s\n",
                                aflag, bflag, cvalue);
                for (index = optind; index < argc; index++)
                        printf ("Non-option argument %s\n",
                                argv[index]);
                return 0;
        }
```

Имеются некоторые примеры, показывающие, что эта программа печатает с различными комбинациями аргументов:

```
% testopt
aflag = 0, bflag = 0, cvalue = (null)

% testopt -a -b
aflag = 1, bflag = 1, cvalue = (null)

% testopt -ab
aflag = 1, bflag = 1, cvalue = (null)

% testopt -c foo
aflag = 0, bflag = 0, cvalue = foo

% testopt -cfoo
aflag = 0, bflag = 0, cvalue = foo

% testopt arg1
aflag = 0, bflag = 0, cvalue = (null)
Non-option argument arg1

% testopt -a arg1
aflag = 1, bflag = 0, cvalue = (null)
Non-option argument arg1

% testopt -c foo arg1
aflag = 0, bflag = 0, cvalue = foo
Non-option argument arg1

% testopt -a -- -b
aflag = 1, bflag = 0, cvalue = (null)

                - 493 -

Non-option argument -b

% testopt -a -
aflag = 1, bflag = 0, cvalue = (null)
Non-option argument -
```

22.1.4 Синтаксический анализ Длинных Опций

Чтобы воспринимать GNU стиль длинных опций также как одиночно-символьные опции, используйте getopt_long вместо getopt. Вы должны заставить каждую программу принимать длинные опции, если она использует опции, это занимает немного ресурсов, и помогает новичкам помнить, как использовать программу.

struct option (тип данных)
Эта структура описывает одиночное длинное имя опции для getopt_long. Аргумент longopts должен быть массивом этих структур, по одной для каждой длинной опции.
Завершите массив элементом, содержащим все нули.
Структура option имеет поля:

```
const char *name
```

Это поле - имя опции. Это - строка.

```
int has_arg
```

Это поле говорит, берет ли опция аргумент. Это - целое число, и имеются три законных значения: no_argument, required_argument и optional_argument.

```
int *flag
int val
```

Эти поля управляют, как сообщать или действовать на опцию, когда она прочитана.

Если flag - пустой указатель, то val - значение, которое идентифицирует эту опцию. Часто эти значения выбраны, чтобы однозначно идентифицировать специфические длинные опции.

Если flag - не пустой указатель, это должен быть адрес int переменной, которая является флагом для этой опции. Значение в val - значение, которое нужно сохранить во флаге, чтобы указать, что опция была замечена.

int getopt_long (int argc, char **argv, const char *shortopts, struct option *longopts, int *indexptr) (функция)

Декодирует опции из вектора argv (чья длина argc). Аргумент shortopts описывает короткие опции, принимаемые точно так же как это делается в getopt. Аргумент longopts описывает длинные опции (см. выше).

Когда getopt_long сталкивается с короткой опцией, она делает ту же самую вещь, что и getopt: она возвращает символьный код для опции, и сохраняет аргумент этой опции (если он имеется) в optarg.

Когда getopt_long сталкивается с длинной опцией, она действует, основываясь на flag и val полях определения этой опции.

Если flag - пустой указатель, то getopt_long возвращает содержимое val, чтобы указать какую опцию она нашла. Вы должны указывать различные значения в val поле для опций с различными значениями, так что Вы можете декодировать эти значения после того, как getopt_long возвращается. Если длинная опция эквивалентна короткой опции, Вы может использовать код символа короткой опции в val.

Если flag - не пустой указатель, значит эта опция должна только установить флаг в программе. Флаг - переменная типа int, что Вы и определяете. Поместите адрес флага в поле flag. Поместите в val поле значение, которое Вы хотели бы, чтобы эта опция сохранила во флаге. В этом случае, getopt_long возвращает 0.

Для любой длинной опции, getopt_long сообщает Вам индекс в массиве longopts определения опций, сохраняя его в *indexptr. Вы можете получить имя опции через longopts [*indexptr].name . Так что Вы можете различать длинные опции или значениями в их val полях или их индексами.

Когда длинная опция имеет аргумент, getopt_long помещает значение аргумента в переменную optarg перед возвращением. Когда опция не имеет никакого аргумента, значение в optarg - пустой указатель.

Когда getopt_long не имеет больше опций для обработки, она возвращает -1, и оставляет в переменной optind индекс следующего остающегося аргумента в argv.

22.1.5 Пример Синтаксического анализа Длинных Опций

```
#include
static int verbose_flag;
int
main (argc, argv)
                        int argc;
                        char **argv;
{
        int c;
        while (1)
        {
                static struct option long_options[] = {
                        {"verbose", 0, &verbose_flag, 1},
                        {"brief", 0, &verbose_flag, 0},
                        {"add", 1, 0, 0},
                        {"append", 0, 0, 0},
                        {"delete", 1, 0, 0},
                        {"create", 0, 0, 0},
                        {"file", 1, 0, 0},
                        {0, 0, 0, 0}
                };
```

```
                int option_index = 0;
                c = getopt_long (argc, argv, "abc:d:",
                long_options, &option_index);
                if (c == -1)
                        break;
                switch (c)
                {
                        case 0:
                           if (long_options[option_index].flag
                                                                != 0)
                                break;
                           printf ("option %s",
         long_options[option_index].name);
                           if (optarg)
                                printf (" with arg %s", optarg);

                       - 496 -

                                printf ("\n");
                                break;
                        case 'a':
                                puts ("option -a\n");
                                break;
                        case 'b':
                                puts ("option -b\n");
                                break;
                        case 'c':
                                printf ("option -c with value
                                        `%s'\n", optarg);
                                break;
                        case 'd':
                                printf ("option -d with value
                                        `%s'\n", optarg);
                                break;
                        case '?':
           /* getopt_long already printed an error message. */
                                break;
                        default:
                                abort ();
                }
           }
           if (verbose_flag)
                   puts ("verbose flag is set");
      /* Печатаем любые остающиеся аргументы командной строки
                                                (не опции). */
                   if (optind < argc)
                   {
                           printf ("non-option ARGV-elements: ");
                           while (optind < argc)
                                   printf ("%s ", argv[optind++]);
                           putchar ('\n');
                   }
                   exit (0);
           }
```


22.2 Переменные среды

Когда программа выполняется, она получает информацию относительно контекста, в котором она вызывалась двумя способами. Первый механизм использует argv и argc аргументы функции main, и обсужден в Разделе 22.1 [Аргументы Программы]. Второй механизм использует переменные среды и обсужден в этом разделе.

Механизм argv обычно используется, чтобы передать аргументы командной строки, специфические для специфической вызываемой программы. Среда, с другой стороны, следит за информацией, которая разделена многими программами, и к ней менее часто обращаются.

Переменные среды, обсужденные в этом разделе - те же самые переменные среды, что Вы устанавливаете используя присванивание и команду export в оболочке. Программы, выполненные из оболочки наследуют все переменные среды из оболочки.

Стандартные переменные среды используются для уточнения инфрмации относительно исходного каталога пользователя, типа терминала, текущего стандарта, и так далее; Вы можете определять дополнительные переменные для других целей. Набор всех переменных среды, которые имеют значения, общеизвестен как среда.

Имена переменных среды чувствительны к регистру и не должны содержать символ "=".

Определенные системой переменные среды неизменны относительно верхнего регистра.

Значения переменных среды могут быть чем угодно, что может представляться как строка. Значение не должно содержать внедренный пустой символ, так как им принято завершать строку.

22.2.1 Доступ к Среде

К значению переменной среды можно обращаться getenv функцией. Это объявлено в заглавном файле " stdlib.h ".

char * getenv (const char *name) (функция)

Эта функция возвращает строку, которая является значением переменной среды. Вы не должны изменять эту строку. В некоторых системах не-UNIX, не использующих библиотеку GNU, она может быть

перезаписана поверх последующими обращениями к getenv (но не к любой другой библиотечной функции). Если имя переменной среды не определено, значение - пустой указатель.

int putenv (const char *string) (функция)

Putenv функция добавляет или удаляет определения из среды. Если строка имеет форму " name=value ", определение будет добавлено к среде. Иначе, строка интерпретируется как имя переменной среды, и любое определение для этой переменной в среде будет удалено.

Библиотека GNU обеспечивает эту функцию для совместимости с SVID; она не может быть доступна в других системах.

Вы можете иметь дело непосредственно с основным представлением объектов среды, чтобы добавить большое количество переменных к среде (например, связываться с другой программой, которую Вы собираетесь выполнять; см. Раздел 23.5 [Выполнение Файла]).

char ** environ (переменная)

Среда представляется как массив строк. Каждая строка имеет формат " name=value ". Порядок, в котором строки появляются в среде не значителен, но то же самое имя не должно появиться больше чем один раз. Последний элемент массива - пустой указатель.

Эта переменная объявлена в заглавном файле " unistd.h ".

Если Вы хотите только получить значение переменной среды, использует getenv.

22.2.2 Стандартные Переменные среды

Эти переменные среды имеют стандартные значения. Это не означает, что они всегда представляются в среде; но если эти переменные присутствуют, они имеют эти значения, и Вы не должны пробовать использовать эти имена переменных среды для некоторой другой цели.

HOME

Это - строка представляет исходный каталог пользователя, или начальное значение рабочего каталога по умолчанию.

Пользователь может устанавливать HOME как любое значение. Если, Вы должны получить соответствующий исходный каталог для

специфического пользователя, Вы не должен использовать HOME; взамен, найдите имя пользователя в базе данных пользователей (см. Раздел 25.12 [База данных Пользователей]).

LOGNAME

Это - имя пользователя, используемое для входа в систему. Так как значение в среде может быть произвольно, это - не надежный способ идентифицировать пользователя, который выполняет процесс; функция getlogin (см. Раздел 25.11 [Кто Вошел В Систему]) лучше для той цели.

Для большинства целей, лучше использовать LOGNAME, потому что она позволяет пользователю определять значение.

PATH

Путь - последовательность имен каталогов, которая используется для поиска файла. Переменная PATH содержит путь, используемый для поиска программ, которые будут выполнены.

Execlp и execvp функции (см. Раздел 23.5 [Выполнение Файла]) используют эту переменную среды, как и многие оболочки и другие утилиты, которые выполнены в терминах этих функций.

Синтаксис пути - последовательность имен каталогов, отделяемых двоеточиями. Пустая строка вместо имени каталога замещает текущий каталог (см. Раздел 9.1 [Рабочий каталог]).

Типичное значение для этой переменной среды могло бы быть: :/bin:/etc:/usr/bin:/usr/new/X11:/usr/new:/usr/local/bin

Это означает что, если пользователь пробует выполнять программу, именованную foo, система будет искать файлы, именованные " foo ", " /bin/foo ", " /etc/foo ", и так далее. Первый из этих файлов, который существует - будет выполнен.

TERM

Определяет вид терминала, который получает вывод программы. Некоторые программы могут использовать эту информацию, чтобы пользоваться преимуществом специальных escape-последовательностей или режимов терминала, обеспечиваемых специфическими видами терминалов. Многие программы, которые используют termcap библиотеку (см. раздел " Поиск Описания Терминала " в Библиотечном Руководстве Termcap) использует переменную среды TERM.

TZ

Определяет часовой пояс. См. Раздел 17.2.5 [Переменная TZ], для уточнения инфрмации относительно формата этой строки и как она

используется.

LANG

Определяет заданный по умолчанию стандарт, используемый для категорий атрибутов, если ни LC_ALL ни специфическая переменная среды для этого класса не установлены. См. Главу 19 [Стандарты], для получения более подробной информации.

LC_COLLATE

Определяет какой стандарт использовать для строковой сортировки.

LC_CTYPE

Определяет какой стандарт использовать для символьных наборов и символьной классификации.

LC_MONETARY

Определяет какой стандарт использовать для форматирования валютных значений.

LC_NUMERIC

Определяет какой стандарт использовать для форматирования чисел.

LC_TIME

Определяет то, какой стандарт использовать для форматирования даты/времени.

_POSIX_OPTION_ORDER

Если эта переменная среды определена, она подавляет обычное переупорядочение аргументов командной строки getopt. См. Раздел 22.1.1 [Синтаксис Аргумента].

22.3 Завершение Программы

Обычный способ завершения программы - просто возврат функции main. Значение состояния выхода, возвращенное из функции main используется, чтобы сообщить информацию обратно родительскому процессу или оболочке.

Программа может также завершаться вызывая функцию exit.

Кроме того, программы могут быть завершены сигналами; это обсуждено более подробно в Главе 21 [Обработка Сигналов]. Функция abort вызывает сигнал, который уничтожает программу.

22.3.1 Нормальное Окончание

Процесс завершается обычно, когда программа вызывает exit. Возвращение из main эквивалентно вызову exit, и значение, которое main возвращает, используется как аргумент exit.

void exit (int status) (функция)
Функция exit завершает процесс с состоянием status. Эта функция не возвращается.
Нормальное окончание вызывает следующие действия:
1. Функции, которые были зарегистрированы с atexit или on_exit функциями, вызываются в обратном порядке их регистрации. Этот механизм позволяет вашему приложению определять собственные действия "очистки", которые нужно выполнить по окончании программы. Обычно, это используется, чтобы делать вещи подобно сохранению информации о состоянии программы в файле, или размыкании блокировок в базах общих данных.
2. Все открытые потоки будут закрыты. См. Раздел 7.4 [Закрытие Потоков]. Кроме того, временные файлы, открытые с tmpfile функцией будут удалены; см. Раздел 9.10 [Временные Файлы].
3. _exit вызывается, завершая программу. См. Раздел 22.3.5 [Внутренняя организация Окончания].

22.3.2 Состояние Выхода

Когда программа выходит, она может возвращать родительскому процессу малое количество информации относительно причины окончания, используя состояние exit. Это - значение между 0 и 255, которое выходящий процесс передает как аргумент exit.
Обычно Вы должны использовать состояние exit, чтобы сообщить очень широкую информацию относительно успеха или отказа. Вы не можете обеспечивать множество подробностей относительно причин для отказа, да и большинство родительских процессов не требуют много подробностей.
Имеются соглашения для того, что некоторые программы должны возвратить. Наиболее общее соглашение - просто 0 для успеха и 1 для отказа. Программы, которые выполняют сравнение, используют другое

соглашение: они используют состояние, 1, чтобы указать несоответствие, и состояние 2, чтобы указать неспособность сравнить. Ваша программа должна следовать за существующим соглашением, если существующее соглашение имеет смысл для нее.
Общее соглашение резервирует значения 128 состояний и более для специальных целей. В частности значение 128 используется, чтобы указать отказ выполнить другую программу в подпроцессе. Это соглашение не удовлетворяет универсальным условиям, но неплохо следовать за ним в ваших программах.
Предупреждение: Не пробуйте использовать число ошибок как состояние exit. Это фактически не очень полезно; родительский процесс вообще не должен заботиться, сколько ошибок произошло. К тому же значение состояния усекается до восьми бит. Таким образом, если программа пробовала передать 256, родитель получит 0 шибок т.е. успех.
По той же самой причине не работает использование значения errno как состояния exit.
Примечание Переносимости: Некоторые не-posix системы используют различные соглашения для значений состояния exit. Для большей переносимости, Вы можете использовать макркоманды EXIT_SUCCESS и EXIT_FAILURE для стандартного значения состояния успеха и отказа, соответственно. Они объявлены в файле " stdlib.h ".
int EXIT_SUCCESS (макрос)
Эта макркоманда может использоваться с функцией exit, чтобы указать успешное завершение программы.
На системах POSIX, значение этой макркоманды - 0. В других системах, значение может быть другим (возможно не-константа) целочисленным выражением.
int EXIT_FAILURE (макрос)
Эта макркоманда может использоваться с функцией exit, чтобы указать неудачное завершение программы в общем смысле.
На системах POSIX, значение этой макркоманды 1. На других системах, значение может быть другим.

22.3.3 Очистки на Выходе

Ваша программа может выполнить собственные функции очистки при нормальное окончании. Ненадежно вызывать функции очистки явно перед выходом. Намного лучше делать очистку невидимой для приложения, устанавливая функцию очистки используя atexit или on_exit.

int atexit (void (*function) (void)) (функция)

Atexit функция регистрирует функцию function, которую нужно вызвать при нормальном окончании программы. Функция вызывается без аргументов.

Возвращаемое значение из atexit - нуль при успехе и отличное от нуля, если функция не может быть зарегистрирована.

int on_exit (void (*function)(int status, void *arg), void *arg)

Эта функция - несколько более мощный вариант atexit. Она принимает два аргумента, функцию и произвольный указатель. При нормальном окончании программы, функция вызывается с двумя аргументами: значением состояния, переданным exit, и параметром arg.

Эта функция включена в библиотеку GNU C только для совместимости с SunOS, и может не обеспечиваться другими реализациями.

Имеется тривиальная программа, которая иллюстрирует использование exit и atexit:

```
#include
#include
void
bye (void)
{
        puts ("Goodbye, cruel world....");
}
int
main (void)
{
        atexit (bye);
        exit (EXIT_SUCCESS);
}
```

Когда эта программа выполнена, она печатает сообщение и выходит.

22.3.4 Прерывание выполнения Программы

Вы можете прервать вашу программу, используя функцию abort. Прототип для этой функции находится в " stdlib.h ".

void abort (void) (функция)

Функция abort вызывает аварийное окончание программы. Она не выполняет функции очистки, зарегистрированные с atexit или on_exit.

Эта функция фактически завершает процесс, вызывая сигнал SIGABRT, и ваша программа может включать обработчик, чтобы прервать этот сигнал; см. Главу 21 [Обработка Сигнала].

22.3.5 Внутренняя организация Окончания

Функция _exit - примитив, используемый для окончания процесса exit. Она объявлена в заглавном файле " unistd.h ".

void _exit (int status) (функция)

_exit - функция для завершения процесса с состоянием status. Вызов этой функции не выполняет функции очистки, зарегистрированные с atexit или on_exit.

Когда процесс завершается по любой причине либо явным запросом окончания, либо окончанием в результате сигнала, производятся следующие действия:

* Все описатели открытого файла в процессе будут закрыты. См. Главу 8 [Ввод - вывод низкого уровня].
* 8 битов младшего разряда возвращающегося кода состояния сохранены, для передачи родительскому процессу через wait или waitpid; см. Раздел 23.6 [Завершение Процесса].

* Любым дочерним процессам завершаемого процесса будет назначен новый родительский процесс. (Это - init процесс, с ID процесса 1.)
* Сигнал SIGCHLD послан родительскому процессу.
* Если, процесс является лидером сеанса, который контролировал терминал управления, то сигнал SIGHUP будет послан каждому процессу в приоритетной работе, и терминал управления - будет отсоединен от этого сеанса. См. Главу 24 [Управление заданиями].
* Если окончание процесса останавлвает любой элемент группы этого процесса, то сигнал SIGHUP и сигнал SIGCONT будет послан

каждому процессу в группе. См. Главу 24 [Управление заданиями].

23. Дочерние Процессы

Процессы - примитивные модули для резервирования ресурсов системы. Каждый процесс имеет собственное адресное пространство. Процесс выполняет программу; Вы можете иметь многократные процессы, выполняющие ту же самую программу, но каждый процесс имеет собственную копию программы внутри собственного адресного пространства и выполняет это независимо от других копий.

Процессы организованы иерархически. Каждый процесс имеет родительский процесс. Процессы, созданные данным родителем называются дочерними процессами. Дочерний наследует многие из атрибутов родительского процесса.

Эта глава описывает, как программа может создавать, завершать, и управлять дочерними процессами. Фактически, имеются три различных операции: создание нового дочернего процесса, назначение новому процессу выполнить программу, и координирование завершения дочернего процесса.

Функция системы обеспечивает простой механизм для выполнения другой программы; он делает все три шага автоматически. Если Вы нуждаетесь в большом количестве контроля, Вы можете использовать примитивные функции, чтобы делать каждый шаг индивидуально.

23.1 Выполнение Команды

Простой способ выполнять другую программу состоит в том, чтобы использовать функцию system. Эта функция делает всю работу выполнения подпрограммы, но она не дает Вам контроля над подробностями: Вы должны ждать, пока подпрограмма не завершится прежде, чем Вы сможете делать что-нибудь еще.

int system (const char *command) (функция)

Эта функция выполняет command как команду оболочки. В библиотеке GNU C, она всегда использует заданную по умолчанию оболочку sh, чтобы выполнить команду. В частности она ищет каталоги в PATH, чтобы найти программу для выполнения. Возвращаемое значение -1, если не возможно создать процесс оболочки, иначе - состояние

процесса оболочки. См. Раздел 23.6 [Завершение Процесса], для подробностей относительно того, как этот код состояния может интерпретироваться.

Функция system объявлена в заглавном файле " stdlib.h ".

Примечание Переносимости: Некоторые реализации C могут не иметь понятие командного процессора, который может выполнять другие программы. Вы можете определить, существует ли командный процессор, выполняя system (NULL); если возвращаемое значение отлично от нуля, командный процессор доступен.

Popen и pclose функции (см. Раздел 10.2 [Трубопровод на Подпроцесса]) близко связаны функцией system. Они позволяют родительскому процессу связываться со стандартным вводом и выводом выполняемой команды.

23.2 Понятия Создания Процесса

Этот раздел дает краткий обзор действий и шагов по созданию процесса и выполнения им другой программы.

Каждый процесс именован ID процесса. Уникальный ID процесса дан каждому процессу при создании.

Процессы создаются системным вызовом fork (так что операция создания нового процесса иногда вызывает раздваивание процесса).

Дочерний процесс, созданный fork - точный аналог первоначального родительского процесса, за исключением того, что он имеет собственный ID.

Если Вы хотите, чтобы ваша программа ждала завершения дочернего процесса, Вы должен делать это явно после операции fork, вызовом wait или waitpid (см. Раздел 23.6 [Завершение Процесса]). Эти функции дают Вам ограниченную информацию относительно того, почему завершился дочерний прцесс - например, код состояния exit.

Раздвоенный дочерний процесс продолжает выполнять ту же самую программу как родительский процесс, в точке возвращения fork. Вы можете использовать возвращаемое значение от fork, чтобы отличить, выполняется ли программа в родительском процессе или в дочернем.

Наличие нескольких процессов выполняющх ту же самую программу не очень полезно. Но дочерний может выполнять другую программу, используя одну из запускающих функций; см. Раздел 23.5 [Выполнение Файла]. Программа, которую процесс выполняет, называется образом

процесса. Начало выполнения новой программы заставляет процесс забыть все относительно предыдущего образа процесса; когда программа выходит, процесс тоже выходит, вместо того, чтобы возвратиться к предыдущему образу процесса.

23.3 Идентификация Процесса

Pid_t тип данных для ID процесса. Вы можете получить ID процесса, вызывая getpid. Функция getppid возвращает ID родителя текущего процесса (это также известно как ID родительского процесса). Ваша программа должна включить заглавные файлы " unistd.h " и " sys/types.h " чтобы использовать эти функции.

pid_t (тип данных)

Pid_t тип данных - целое число со знаком, который способен представить ID процесса. В библиотеке GNU, это - int.

pid_t getpid (void) (функция)

Getpid функция возвращает ID текущего процесса.

pid_t getppid (void) (функция)

Getppid функция возвращает ID родителя текущего процесса.

23.4 Создание Процесса

Функция fork - примитив для создания процесса. Она объявлена в заглавном файле " unistd.h ".

pid_t fork (void) (функция)

Функция fork создает новый процесс.

Если операция является успешной, то и родительский и дочерний процессы видят что fork возвращается, но с различными значениями: она возвращает значение 0 в дочернем процессе и ID порожденного процесса (ребенка) в родительском процессе.

Если создание процесса потерпело неудачу, fork возвращает значение -1 в родительском процессе. Следующие errno условия ошибки определены для fork:

EAGAIN не имеется достаточных ресурсов системы чтобы создать другой процесс, или пользователь уже имеет слишком много процессов.

ENOMEM процесс требует большего количества места чем система могла обеспечить.

Специфические атрибуты дочернего процесса, которые отличаются от родительского процесса:

* Дочерний процесс имеет собственный уникальный ID.
* ID родителя дочернего процесса - ID родительского процесса.
* Дочерний процесс получает собственные копии описателей открытых файлов родительского процесса. Впоследствии изменение атрибутов описателей файла в родительском процессе не будет воздействовать на описатели файла в дочернем, и наоборот. См. Раздел 8.7 [Операции Управления].
* Прошедшее процессорное время для дочернего процесса установлено на нуль; см. Раздел 17.1 [Процессорное время].
* Дочерний не наследует набор блокировок файла родительского процессоа. См. Раздел 8.7 [Операции Управления].
* Дочерний не наследует набор таймеров родительского процесса. См. Раздел 17.3 [Установка Сигнализации].

* Набор отложенных сигналов (см. Раздел 21.1.3 [Получение Сигналов]) для дочернего процесса, очищен. (Дочерний процесс наследует маску блокированных сигналов и действий сигналов из родительского процесса.)

pid_t vfork (void) (функция)

Vfork функция подобна fork, но более эффективна; однако, имеются ограничения, которым Вы должны следовать, чтобы использовать ее безопасно.

В то время как fork делает полную копию адресного пространства вызывающего процесса и позволяет, и родителю и дочернему выполняться независимо, vfork не делает эту копию.

Взамен, дочерний процесс, созданный с vfork совместно использует адресное пространство родителя, пока он не вызывает одну из функций exec. Тем временем, родительский процесс приостанавливает свое выполнение.

Вы должны быть очень осторожны, чтобы не позволить дочернему процессу, созданному с vfork изменять любые глобальные данные или даже локальные переменные, общедоступнные с родителем. Кроме того, дочерний процесс не может возвращаться из (или делать длинный переход) функции, которая вызвала vfork! Это спутало бы информацию управления родительского процесса. Если Вы сомневаетесь, используйте fork.

Некоторые операционные системы не выполняют vfork. Библиотека

GNU C разрешает Вам использовать vfork на всех системах, но фактически выполняет fork, если vfork не доступна. Если Вы соблюдаете соответствующие предосторожности при использовании vfork, ваша программа будет работать, даже если система использует fork взамен.

23.5 Выполнение Файла

Этот раздел описывает совокупность exec функций, для выполнения файла как образа процесса. Вы можете использовать эти функции, чтобы заставить дочерний процесс выполнить новую программу после того, как он был раздвоен.

Эти функции отличаются тем, как Вы определяете аргументы, но они все делают ту же самую вещь. Они объявлены в заглавном файле " unistd.h ".

int execv (const char *filename, char *const argv[]) (функция)

Execv функция выполняет файл, именованный filename как новый образ процесса.

Аргумент argv - массив строк с нулевым символом в конце, который используется, чтобы обеспечить значение для аргумента argv функции main программы, которая будет выполнена. Последний элемент этого массива должен быть пустой указатель. Обычно, первый элемент этого массива - имя файла программы. См. Раздел 22.1 [Аргументы Программы] , для подробностей относительно того, как программы могут обращаться к этим аргументам.

Среда для нового образа процесса берется из переменной environ текущего образа процесса; см. Раздел 22.2 [Переменные среды], для уточнения инфрмации относительно сред.

int execl (const char *filename, const char *arg0, . . .) (функция)

Подобна execv, но строки argv определены индивидуально, а не как массив. Пустой указатель должен быть передан как последний такой аргумент.

int execve (const char *filename, char *const argv[], char *const env[])

Подобна execv, но разрешает Вам определять среду для новой программы явно как env аргумент. Это должен быть массив строк в том же самом формате как переменная environ; см. Раздел 22.2.1 [Доступ

Среды].

int execle (const char *filename, const char *arg0, char *const env[], . . .)

Подобна execl, но разрешает Вам определять среду для новой программы явно. Аргумент среды передан после пустого указателя, который отмечает последний аргумент argv, и должен быть массивом строк в том же самом формате как переменная environ.

int execvp (const char *filename, char *const argv[]) (функция)

Execvp функция подобна execv, за исключением того, что она ищет каталоги, перечисленные в переменной среды PATH (см. Раздел 22.2.2 [Стандартная Среда]) чтобы найти полное имя файла filename, если filename не содержит наклонную черту вправо.

Эта функция полезна для выполняющихся утилит системы, потому что она ищет их в местах, которые пользователь выбрал. Оболочки используют ее, чтобы выполнить команды написанные пользователем.

int execlp (const char *filename, const char *arg0, . . .) (функция)

Эта функция - подобна execl, за исключением того, что она выполняет тот же поиск имени файла как в execvp.

Размер списка параметров и списка среды, вместе не должен быть больше чем ARG_MAX байт. См. Раздел 27.1 [Общие Ограничения]. В системе GNU, размер (который сравнивается с ARG_MAX) включает, для каждой строки, число символов в строке, плюс размер char*, плюс один, округленный вверх после умножения на размер char*. Другие системы могут иметь несколько отличные правила для подсчета.

Эти функции обычно не возвращаются, так как выполнение новой программы заставляет завершиться программу выполнения в настоящее время. Значение -1 возвращено в случае отказа. В дополнение к обычным синтаксическим ошибкам имени файла (см. Раздел 6.2.3 [Ошибки Имени файла]), следующие errno условия ошибки определены для этих функций:

E2BIG объединенный размер списка параметров новой программы и списка среды больше чем ARG_MAX байт. Система GNU не имеет никакого специфического ограничения размера списка параметров, так что этот код ошибки не может получиться, но Вы можете получать ENOMEM взамен, если аргументы слишком большие для доступной памяти.

ENOEXEC заданный файл не может быть выполнен, потому что он не находится в правильном формате.

ENOMEM Выполнение заданного файла требует большего количества памяти чем было доступно.

Если выполнение нового файла преуспевает, это модифицирует поле времени доступа файла, как будто файл был прочитан. См. Раздел 9.8.9 [Времена Файла].

Выполнение нового образа процесса полностью не изменяет содержимое памяти, копируются только аргументы и строки среды. Но много других атрибутов процесса неизменяемы:

* ID процесса и ID родительского процесса. См. Раздел 23.2 [Понятия Создания Процесса].
* Групповая принадлежность сеанса и процесса. См. Раздел 24.1 [Понятия Управления заданиями].
* Реальный пользовательский ID, ID группы, и дополнительный ID группы. См. Раздел 25.2 [Владелец Процесса].
* Отложенные таймеры. См. Раздел 17.3 [Установка Сигнализации].
* Текущий рабочий каталог и корневой каталог. См. Раздел 9.1 [Рабочий каталог].
* Маска режима создаваемого файла. См. Раздел 9.8.7 [Установка Прав].
* Маска сигналов Процесса; см. Раздел 21.7.3 [Маска сигналов Процесса].
* Отложенные сигналы; см. Раздел 21.7 [Блокированные Сигналы].
* Прошедшее процессорное время, связанное с процессом; см. Раздел 17.1 [Процессорное время]. Если set-user-ID и set-group-ID биты режима файла образа процесса установлены, это воздействует на эффективный ID пользователя и эффективный ID группы (соответственно) процесса. Эти понятия обсуждены подробно в Разделе 25.2 [Влвделец Процесса].

Сигналы которые игнорируются в существующем образе процесса, также будут установлены, чтобы игнорироваться в новом образе процесса. Все другие сигналы будт установлены по умолчанию в новом образе процесса. См. Главу 21 [Обработка Сигнала].

Описатели Файла, открытые в существующем образе процесса остаются открытыми в новом образе процесса, если они не имеют FD_CLOEXEC флага. Файлы, которые остаются открытыми, наследуют все атрибуты описания открытого файла из существующего образа процесса,

включая блокировки файла. Описатели Файла обсуждены в Главе 8 [Ввод

- вывод низкого уровня].

Новый образ процесса не имеет никаких потоков за исключением тех, что он создает заново.

Каждый из потоков в предыдущем образе процесса имеет описатель внутри него, и эти описатели остаются после exec (если они не имеют FD_CLOEXEC). Новый образ процесса может повторно соединять их с новыми потоками, используя fdopen (см. Раздел 8.4 [Описатели и Потоки]).

23.6 Завершение Процесса

Функции, описанные в этом разделе используются, чтобы ждать завершения или останова дочернего процесса и определять его состояние. Эти функции объявлены в заглавном файле " sys/wait.h ".

pid_t waitpid (pid_t pid, int *status_ptr, int options) (функция)

Waitpid функция используется, чтобы запросить информацию состояния дочернего процесса, чей ID является pid. Обычно, вызывающий процесс приостановлен, пока дочерний процесс не делает информацию состояния доступной, завершаясь.

Другие значения для pid аргумента имеют специальные интерпретации. Значение -1 или WAIT_ANY информация состояния для любого дочернего процесса; значение 0 или WAIT_MYPGRP запрашивает информацию для любого дочернего процесса в той же самой группе процесса как вызывающий процесс; и любое другое отрицательное значение - pgid запрашивает информацию для любого дочернего процесса, чей ID группы - pgid.

Если информация состояния дочернего процесса доступна немедленно, эта функция возвращается немедленно без ожидания. Если доступна информация состояния больше чем одного готового продолжиться дочернего процесса, один из них будет выбран беспорядочно, и его состояние возвращено немедленно.

Чтобы получить состояние других готовых продолжиться дочерних процессов, Вы должны вызвать waitpid снова.

Аргумент options - битовая маска. Значение должно быть поразрядным ИЛИ (то есть `|') нуля или большого количества WNOHANG и WUNTRACED флагов. Вы можете использовать WNOHANG флаг, чтобы

указать, что родительский процесс не должен ждать; и WUNTRACED флаг, чтобы запросить информацию состояния остановленных процессов также как процессов, которые завершились.

Информация состояния дочернего процесса сохранена в объекте, на который указывает status_ptr, если status_ptr не пустой указатель.

Возвращаемое значение - обычно ID дочернего процесса, о чьем состояние сообщено. Если WNOHANG опция была определена и никакой дочерний процесс, не ждет, чтобы быть отмеченным, то значение - нуль. Значение -1 возвращено в случае ошибки. Следующие errno ошибки определены для этой функции:

EINTR

Функция была прервана получением сигнала. См. Раздел 21.5 [Прерванные Примитивы].

ECHILD

Не имеется никаких дочерних процессов, или заданный pid не дочерний для вызывающего процесса.

EINVAL

Недопустимое значение аргумента options.

Эти символические константы определены как значения для pid аргумента waitpid функции.

WAIT_ANY

Эта макркоманда (чье значение -1) определяет, что waitpid должен возвратить информацию состояния относительно любого дочернего процесса.

WAIT_MYPGRP

Эта константа (со значением 0) определяет, что waitpid должен возвратить информацию состояния относительно любого дочернего процесса в той же самой группе процесса что и вызывающий процесс.

Эти символические константы определены как флаги для аргумента options функции waitpid.

Вы можете сделать OR флагов вместе, чтобы получить значение, и использовать его как аргумент.

WNOHANG

Этот флаг определяет, что waitpid должна возвратиться немедленно вместо ожидания, если не имеется никакого дочернего процесса,

готового быть отмеченным.
WUNTRACED
Этот флаг определяет, что waitpid должна сообщить состояние

любых дочерних процессов, которые были остановлены также как тех, которые завершились.
pid_t wait (int *status_ptr) (функция)
Это - упрощенная версия waitpid; используется, чтобы ждать пока не завершится любой дочерний процесс. Обращение:

```
wait (&status)
```

эквивалентно:

```
waitpid (-1, &status, 0)
```

Имеется пример того, как использовать waitpid, чтобы получить состояние всех дочерних процессов, которые завершились, без какого-либо ожидания. Эта функция разработана, чтобы быть обработчиком для сигнала SIGCHLD, который указывает, что по крайней мере один дочерний процесс завершился.

```
void
sigchld_handler (int signum)
{
        int pid;
        int status;
        while (1)
        {
                pid = waitpid (WAIT_ANY, &status,
                               WNOHANG);
                if (pid < 0)
                {
                        perror ("waitpid");
                        break;
                }
                if (pid == 0)
                break;
                notice_termination (pid, status);
        }
}
```

23.7 Состояние Завершения Процесса

Если значение состояния выхода (см. Раздел 22.3 [Завершение Программы]) дочернего процесса - нуль, то значение состояния, сообщенное waitpid или wait - также нуль. Вы можете проверять

другие виды информации, закодированные в возвращенном значении состояния, используя следующие макркоманды. Эти макркоманды определены в заглавном файле " sys/wait.h ".
int WIFEXITED (int status)
Эта макркоманда возвращает значение отличное от нуля если дочерний процесс завершон exit или _exit.
int WEXITSTATUS (int status)
Если WIFEXITED - истина, эта макркоманда возвращает 8 битов младшего разряда значения состояния выхода из дочернего процесса. См. Раздел 22.3.2 [Состояние Выхода].
int WIFSIGNALED (int status)
Эта макркоманда возвращает значение отличное от нуля, если дочерний процесс завершен потому что он получил сигнал который не был обработан. См. Главу 21 [Обработка Сигнала].
int WTERMSIG (int status)
Если WIFSIGNALED - истина, эта макркоманда возвращает номер сигнала, который завершил дочерний процесс.
int WCOREDUMP (int status)
Эта макркоманда возвращает значение отличное от нуля, если дочерний процесс завершен и произведен core-файл.
int WIFSTOPPED (int status)
Эта макркоманда возвращает значение отличное от нуля, если дочерний процесс остановлен.
int WSTOPSIG (int status)
Если WIFSTOPPED - истина, эта макркоманда возвращает номер сигнала, который заставил дочерний процесс остановиться.

23.8 BSD Функции Ожидания Процесса

Библиотека GNU также обеспечивает эти средства для совместимости с UNIX BSD. BSD использует тип данных union, чтобы представить значения состояния, а не int. Два представления фактически взаимозаменяемы; они описывают те же самые битовые шаблоны. Библиотека GNU C определяет макркоманды типа WEXITSTATUS так, чтобы они работали на любом виде объекта, и функция wait определена, чтобы принять любой тип указателя как аргумент status_ptr.
Эти функции объявлены в " sys/wait.h ".

union wait (тип данных)
Этот тип данных представляет значения состояния окончания программы. Он имеет следующие элементы:
int w_termsig
Значение этого элемента - то же что результат WTERMSIG макркоманды.
int w_coredump
Значение этого элемента - результат WCOREDUMP макркоманды.
int w_retcode
Значение этого элемента - результат WEXITSTATUS макркоманды.
int w_stopsig
Значение этого элемента - результат WSTOPSIG макркоманды.
Вместо того, чтобы обращаться к этим элементам непосредственно, Вы должны использовать эквивалентные макркоманды.
pid_t wait3 (union wait *status_ptr, int options, struct rusage *usage)
Если usage - пустой указатель, wait3 эквивалентна waitpid (-1, status_ptr, options).
Если usage - не пустой символ, wait3 сохраняет тип использования для дочернего процесса в *rusage (но только, если дочерний завершился, а не остановился). См. Раздел 17.5 [Использование Ресурсов].
pid_t wait4 (pid_t pid, union wait *status_ptr, int options, struct rusage *usage)
Если usage - пустой указатель, wait4 эквивалентна waitpid (pid, status_ptr, options).
Если usage - не пустой символ, wait4 сохраняет тип использования для дочернего процесса в *rusage (но только, если дочерний завершился, а не остановился). См. Раздел 17.5 [Использование Ресурсов].

23.9 Пример Создания Процесса

Вот пример программы, показывающий, как Вы могли бы написать функцию, подобную встроенной системе. Она выполняет аргумент command, используя " sh -c command ".

```
#include
#include
#include
#include
#include
#define SHELL "/bin/sh"
int
my_system (const char *command)
{
        int status;
        pid_t pid;
        pid = fork ();
        if (pid == 0)
        {
                execl (SHELL, SHELL, "-c", command,
```

```
                                                NULL);
                                _exit (EXIT_FAILURE);
                        }
                        else if (pid < 0)
                                foo1   = -1;
                        else
                        if (waitpid (pid, &status, 0) != pid)
                                foo1   = -1;
                        return status;
                }
```

Имеется две вещей, на которые Вы должны обратить внимание в этом примере.

Не забудьте, что первый аргумент argv, представляет имя выполняемой программы. Именно поэтому, в обращении к execl, SHELL обеспечена один раз, чтобы назвать выполняемую программу, и второй раз, чтобы обеспечить значение для argv [0].

Вызов execl в дочернем процессе не возвращается, если он

успешен. Если он терпит неудачу, Вы должен делать кое-что, чтобы заставить дочерний процесс завершиться. Правильное поведение для дочернего процесса - сообщить отказ родительскому процессу.

Вызовите _exit, чтобы выполнить это. Причина для использования _exit вместо exit состоит в том, чтобы избежать flush полностью буферизированных потоков типа stdout. Буфера этих потоков возможно содержат данные, которые были скопированы из родительского процесса функцией fork, эти данные будут выводиться в конечном счете родительским процессом. Вызов exit в дочернем вывел бы данные дважды. См. Раздел 22.3.5 [Внутренняя организация Окончания].

24. Управление заданиями

Управление заданиями относится к протоколу для разрешения пользователю двигаться между многими группами процессов (или работ) внутри одиночного сеанса входа в систему. Средства управления заданиями установлены так, чтобы соответствующее поведение для большинства программ устанавливалось автоматически и они не должны делать что-нибудь специальное относительно управления заданиями. Так что Вы можете возможно игнорировать материал этой главы, если Вы не пишите программу входа в систему или оболочку.

Вы должны быть знакомы с понятиями создания процесса (см. Раздел 23.2 [Понятия Создания Процесса]) и обработки сигналов (см. Главу 21 [Обработка Сигналов]) чтобы понять материал этой главы.

24.1 Понятия Управления заданиями

Фундаментальная цель интерактивной оболочки читать команды из терминала пользователя и создавать процессы, чтобы выполнить программы, заданные этими командами. Это можно делать использованием fork (см. Раздел 23.4 [Создание Процесса]) и exec (см. Раздел 23.5 [Выполнение Файла]) функций.

Одиночная команда может выполнять только один процесс, но часто одна команда использует отдельные процессы.

Если Вы используете оператор `|' в команде оболочки, Вы явно, запрашиваете несколько программ в их собственных процессах. Но даже если Вы выполняете только одну программу, она может использовать

многократные процессы внутренне. Например, одиночная команда трансляции типа " cc -c foo .c " обычно использует четыре процесса. Если Вы выполняете make, ее работа - выполнить другие программы в отдельных процессах.

Процессы, принадлежащие одной команде называются группой процессов или работой. Для того, чтобы Вы могли функционировать на всех сразу. Например, печать C-c посылает сигнал SIGINT, чтобы завершить все процессы в приоритетной группе процессов.

Сеанс - большая группа процессов. Обычно все процессы одиночного входа в систему принадлежат тому же самому сеансу.

Каждый процесс принадлежит группе процессов. Когда процесс создан, он становится элементом той же самой группы процессов и сеанса как и родительский процесс. Вы можете помещать его в другую группу процессов, используя setpgid функцию, если группа процессов

принадлежит тому же самому сеансу.

Единственый способ поместить процесс в другой сеанс состоит в том, чтобы сделать его начальным процессом нового сеанса или лидером сеанса, используя setsid функцию. Она также помещает лидера сеанса в новую группу процессов, и Вы не можете перемещать его вне этой группы процессов снова.

Обычно, новые сеансы создаются программой входа в систему, и лидер сеанса - процесс, выполняющий оболочку входа в систему пользователя.

Оболочка, которая поддерживает управление заданиями, должна знать, которая работа может использовать терминал в любое время. Иначе имелось бы сразу много работ, пробующих читать из терминала, и беспорядок относительно того, который процесс должен получить ввод, печатаемый пользователем. Чтобы предотвращать это, оболочка должна сотрудничать с драйвером терминала, используя протокол, описанный в этой главе.

Оболочка может давать безграничный доступ к терминалу управления только одной группы процессов одновременно. Она называется приоритетной работой на этом терминале. Другие группы процессов, управляемые оболочкой, которые выполняются без такого доступа к терминалу, называются фоновыми работами.

Если фоновая работа нуждается в чтении из или записи на терминал управления, она остановлена драйвером терминала. Пользователь может останавливать приоритетную работу, печатая символ SUSP (см. Раздел

12.4.9 [Специальные Символы]) и программа может останавливать любую работу, посылая ей сигнал SIGSTOP. Ответственность оболочки - обратить внимание на останов работ, сообщать пользователю относительно них, и обеспечивать механизмы для разрешения пользователю в интерактивном режиме продолжить остановленные работы и переключать работы между проритетной и фоновыми.

См. Раздел 24.4 [Доступ к Терминалу], для получения более подробной информации о терминале управления,

24.2 Управление Заданиями Необязательно

Не все операционные системы поддерживают управление заданиями. Система GNU поддерживает управление заданиями, но если Вы используете библиотеку GNU на некоторой другой системе, эта система может не поддерживать управление заданиями непосредственно.

Вы можете использовать _POSIX_JOB_CONTROL макркоманду, чтобы проверить в во время компиляции, поддерживает ли система управление заданиями. См. Раздел 27.2 [Опции Системы].

Если управление заданиями не обеспечивается, то может быть только одна группа процессов на сеанс, который ведет себя, как будто он всегда приоритетный. Функции для создания дополнительных групп процессов просто терпят неудачу с кодом ошибки ENOSYS.

Макркоманды, именующие различные сигналы управления заданиями (см. Раздел 21.2.5 [Сигналы Управления заданиями]) определены, даже если управление заданиями не обеспечивается.

24.3 Управление Терминалом Процесса

Один из атрибутов процесса - терминал управления. Дочерние процессы, созданные с fork наследуют терминал управления из их родительского процесса. Таким образом,, все процессы в сеансе наследуют терминал управления от лидера сеанса. Лидер сеанса, который имеет контроль над терминалом, называется процессом управления этого терминала.

Вы вообще не должны волноваться относительно точного механизма, используемого, чтобы зарезервировать терминал управления сеанса, так как это выполнено для Вас системой когда Вы входите в систему.

Индивидуальный процесс отделяется от терминала управления,

когда это вызывает setsid, чтобы стать лидером нового сеанса. См. Раздел 24.7.2 [Функции Группы процессов].

24.4 Доступ к Терминалу Управления

Процессы в приоритетной работе терминала управления имеют неограниченный доступ к этому терминалу; а фоновые процессы - нет.

Этот раздел описывает более подробно, что случается, когда процесс в фоновой работе пробует обращаться к терминалу управления.

Когда процесс в фоновой работе пробует читать из терминала управления, группе процессов обычно послан сигнал SIGTTIN. Это обычно заставляет все процессы в той группе останавливаться (если они не обрабатывают сигнал и не останавливают себя). Однако, если процесс считывания игнорирует или блокирует этот сигнал, то происходит сбои read с EIO ошибкой.

Аналогично, когда процесс в фоновой работе пробует писать на терминал управления, заданное по умолчанию поведение - послать сигнал SIGTTOU группе процессов. Однако, поведение изменяется TOSTOP битом флагов автономных режимов (см. Раздел 12.4.7 [Автономные режимы]). Если этот бит не установлен (по умолчанию), то запись на терминал управления всегда разрешается без сигнала. Запись также разрешается, если сигнал SIGTTOU игнорируется или блокируется процессом записи.

Большинство других операций терминала, которые программа может делать, обрабатываются как чтение или как запиись. (Описание каждой операции должно говорить как.)

24.5 Свободные Группы процессов

Когда процесс управления завершается, терминал становится свободным, и на нем может быть установлен новый сеанс. (Фактически, другой пользователь мог бы войти в систему на терминале.) Это может вызывать проблему, если любые процессы из старого сеанса все еще пробуют использовать этот терминал.

Чтобы предотвратить проблемы, группы процессов, которые продолжают выполняться даже после завершения лидера сеанса, отмечены как свободные группы процессов. Процессы в свободной группе процессов не могут читать из или писать на терминал

управления. Это вызовет EIO ошибку.

Когда группа процессов становится свободной, процессам послан сигнал SIGHUP. Обычно, это заставляет процессы завершиться. Однако, если программа игнорирует этот сигнал или устанавливает обработчик для него (см. Главу 21 [Обработка Сигнала]), она может продолжать выполнять свободную группу процессов даже после того, как процесс управления завершается.

24.6 Выполнение Оболочки Управления заданиями

Этот раздел описывает то, что оболочка должна делать, чтобы выполнить управление заданиями, обеспечивая протяженную типовую программу, чтобы проиллюстрировать включаемые понятия.

* Раздел 24.6.1 [Структуры Данных], представляет пример и первичные структуры данных.
* Раздел 24.6.2 [Инициализация Оболочки], обсуждает действия, которые оболочка должна выполнить, чтобы реализовать управление заданиями.
* Раздел 24.6.3 [Запуск Работ], включает информацию относительно того, как создать работы, чтобы выполнить команды.
* Раздел 24.6.4 [Приоритетный и Фоновые], обсуждает то, что оболочка должна делать при запуске приоритетной работы в противоположность фоновой работе.
* Раздел 24.6.5 [Остановленные и Завершенные Работы], обсуждает сообщения состояния работы для оболочки.
* Раздел 24.6.6 [Продолжение Остановленных Работ], сообщает Вам, как продолжить работы, которые были остановлены.
* Раздел 24.6.7 [Отсутствующие Части], обсуждает другие части оболочки.

24.6.1 Структуры Данных для Оболочки

Все примеры программы, включенные в эту главу - часть простой программы оболочки. Этот раздел представляет структуры данных и сервисные функции, которые используются в примере.

Типовая оболочка имеет дело в основном с двумя структурами данных. Тип job содержит информацию относительно работы, которая является набором подпроцессов, связанных вместе трубопроводами. Тип

process содержит информацию относительно одиночного подпроцесса.
Имеются релевантные объявления структуры данных:

```
typedef struct process
{
        struct process *next;
        char **argv;
        pid_t pid;
        char completed;
        char stopped;
        int status;
} process;
typedef struct job
{
        struct job *next;
        char *command;
        process *first_process;
        pid_t pgid;
        char notified;
        struct termios tmodes;
        int stdin, stdout, stderr;
} job;
job *first_job = NULL;
```

Имеются некоторые сервисные функции, которые используются для оперирования объектами job.

```
job *
find_job (pid_t pgid)
{
        job *j;
        for (j = first_job; j; j = j->next)
                if (j->pgid == pgid)
                        return j;
        return NULL;
}
int
job_is_stopped (job *j)
{
        process *p;
        for (p = j->first_process; p; p = p->next)

               - 524 -

                if (!p->completed && !p->stopped)
                        return 0;
        return 1;
}
int
job_is_completed (job *j)
{
        process *p;
        for (p = j->first_process; p; p = p->next)
                if (!p->completed)
                        return 0;
        return 1;
}
```

24.6.2 Инициализация Оболочки

Когда программа оболочки, которая обычно выполняет управление заданиями, начата, она должна быть внимательна в случае, если она вызвалось из другой оболочки, которая уже делает собственное управление заданиями.

Подоболочка, которая выполняется в интерактивном режиме, должна убедиться, что она была помещена на передний план родительской оболочкой прежде, чем она может давать возможность управлению заданиями непосредственно. Она делает это, получая начальную ID группы процессов с getpgrp функцией, и сравнивая его с ID группы процессов текущей приоритетной работы, связанной с терминалом управления (который может быть восстановлен, использованием функции tcgetpgrp).

Если подоболочка не выполняется как приоритетная работа, она должна остановить себя, посылая сигнал SIGTTIN собственной группе процессов. Она не может произвольно помещаться на передний план; она должна ждать пока пользователь сообщит родительской оболочке сделала это. Если подоболочка продолжена снова, она должна повторить проверку и останов непосредственно снова, если она все

еще не на переднем плане.

Если только подоболочка была помещена на передний план родительской оболочкой, она может давать возможность собственному управлению заданиями. Она делает это, вызывая setpgid, чтобы

поместить себя в собственную группу процессов, и вызывая tcsetpgrp, чтобы поместить эту группу процессов на передний план.

Когда оболочка дает возможность управлению заданиями, она должна установить себя, чтобы игнорировать все сигналы останова управления заданиями так, чтобы она случайно не остановила себя. Вы можете сделать это, устанавливая действие для всех сигналов останова как SIG_IGN.

Подоболочка, которая выполняется не-в интерактивном режиме, не может и не должна поддерживать управление заданиями. Она должна оставить все процессы, которые она создает в той же самой группе процессов как оболочка непосредственно; это позволяет родительской оболочке обрабатывать не-интерактивную оболочку и дочерние процессы как одиночную работу. Это просто сделать, только не используйте любой из примитивов управления заданиями, но Вы не должны забыть заставить оболочку делать это.

Вот код инициализации для типовой оболочки, который показывает, как сделать все это.

```
#include
#include
#include
pid_t shell_pgid;
struct termios shell_tmodes;
int shell_terminal;
int shell_is_interactive;
void
init_shell ()
{
        shell_terminal = STDIN_FILENO;
        shell_is_interactive=isatty(shell_terminal);
        if (shell_is_interactive)
        {
                while (tcgetpgrp (shell_terminal) !=
                        (shell_pgid = getpgrp ()))
                kill (- shell_pgid, SIGTTIN);
                signal (SIGINT, SIG_IGN);
                signal (SIGQUIT, SIG_IGN);
                signal (SIGTSTP, SIG_IGN);
                signal (SIGTTIN, SIG_IGN);
```



```
                signal (SIGTTOU, SIG_IGN);
                signal (SIGCHLD, SIG_IGN);
                shell_pgid = getpid ();
                if (setpgid(shell_pgid,shell_pgid) < 0)
                {
                        perror ("Couldn't put the shell
                        in its own process group");
                        exit (1);
                }
                tcsetpgrp (shell_terminal, shell_pgid);
                tcgetattr(shell_terminal,&shell_tmodes);
        }
}
```

24.6.3 Запуск Работ

Если только оболочка приняла ответственность за выполнение управления заданиями на терминале управления, она может начинать работы в ответ на команды, печатаемые пользователем.

Чтобы создавать процессы в группе процессов, Вы используете те же самые fork и exec, описанные в Разделе 23.2 [Понятия Создания Процесса]. Так как имеются многократные дочерние включаемые процессы, Вы должны быть внимательны, чтобы делать дела в правильном порядке.

Вы имеете два выбора для того, как структурировать дерево родитель - дочерних связей среди процессов. Вы можете либо делать все процессы в группе процессов дочерними процесса оболочки, либо

Вы можете делать один процесс в группе предком всех других процессов в той группе. Типовая программа оболочки, обеспеченная в этой главе использует первый подход, потому что это кажется несколько более простым.

При раздвоении процесса, он должен помещаться в новую группу процессов, вызовом setpgid; см. Раздел 24.7.2 [Функции Группы процессов]. Первый процесс в новой группе становится лидером группы процессов, и его ID, становится ID группы процессов.

Оболочка должна также вызвать setpgid, чтобы поместить каждый из дочерних процессов в новую группу процессов.

Имеется потенциальная проблема синхронизации: каждый дочерний

процесс должен быть помещен в группу процессов прежде, чем он начинает выполнять новую программу, и оболочка зависит от наличия всех дочерних процессов в группе прежде, чем она продолжает выполняться. Если и дочерние процессы и оболочка вызывает setpgid, это гарантирует, что все будет правильно независимо от того, который процесс принимается за это раньше.

Если работа начинается как приоритетная работа, новая группа процессов также, должна быть помещена на передний план на терминале управления, используя tcsetpgrp. Снова, это должно быть выполнено оболочкой также как и каждым из дочерних процессов, чтобы избежать условий состязания.

Следующая вещь, которую каждый дочерний процесс должен делать, - сбросить действия сигналов.

В течение инициализации, процесс оболочки устанавливает себя, чтобы игнорировать сигналы управления заданиями; см. Раздел 24.6.2 [Инициализация Оболочки]. В результате, любые дочерние процессы, которые он создает, игнорируют эти сигналы наследованием. Это нежелательно, так что каждый дочерний процесс должен явно установить действия для этих сигналов обратно к SIG_DFL после того, как он раздвоено.

Так как оболочки следуют за этим соглашением, приложения могут принимать, что они наследуют правильную обработку этих сигналов из родительского процесса. Но каждое приложение не должно изменять обработку сигналов останова. Приложения, которые отключают нормальную интерпретацию символа SUSP, должны обеспечить некоторый другой механизм для пользователя, чтобы остановить работу. Когда пользователь вызывает этот механизм, программа должна послать сигнал SIGTSTP группе процессов, а не только на процесс непосредственно. См. Раздел 21.6.2 [Передача сигналов Другому Процессу].

В заключение, каждый дочерний процесс должен вызвать exec нормальным способом. Это - также точка, в которой должна быть обработана переадресация стандартного ввода и каналов вывода. См. Раздел 8.8 [Дублирование Описателей], для объяснения того, как делать это.

Вот функция из типовой программы оболочки, которая ответственна за запуск программы. Функция выполняется каждым дочерним процессом немедленно после того, как он был раздвоен оболочкой, и никогда не

возвращается.

```
void
launch_process (process *p, pid_t pgid,
                int infile, int outfile, int errfile,
                int foreground)
{
        pid_t pid;
        if (shell_is_interactive)
        {
                pid = getpid ();
                if (pgid == 0) pgid = pid;
                        setpgid (pid, pgid);
                if (foreground)
                        tcsetpgrp (shell_terminal, pgid);
                signal (SIGINT, SIG_DFL);
                signal (SIGQUIT, SIG_DFL);
                signal (SIGTSTP, SIG_DFL);
                signal (SIGTTIN, SIG_DFL);
                signal (SIGTTOU, SIG_DFL);
                signal (SIGCHLD, SIG_DFL);
```

```
                        }
                        if (infile != STDIN_FILENO)
                        {
                                dup2 (infile, STDIN_FILENO);
                                close (infile);
                        }
                        if (outfile != STDOUT_FILENO)
                        {
                                dup2 (outfile, STDOUT_FILENO);
                                close (outfile);
                        }
                        if (errfile != STDERR_FILENO)
                        {
                                dup2 (errfile, STDERR_FILENO);
                                close (errfile);
                        }
                        execvp (p->argv[0], p->argv);
                        perror ("execvp");

                               - 529 -

                        exit (1);
                }
```

Если оболочка не выполняется в интерактивном режиме, эта функция, не делает ничего с группами процессов или сигналами. Не забудьте, что оболочка, не выполняющая управление заданиями должна хранить все подпроцессы в той же самой группе процессов что и оболочка непосредственно.

Вот функция, что фактически начинает полную работу. После создания дочерних процессов, эта функция вызывает некоторые другие функции, чтобы поместить недавно созданную работу на передний план (или как фон); они обсуждены в Разделе 24.6.4 [Приоритетный и Фоновые].

```
                void
                launch_job (job *j, int foreground)
                {
                        process *p;
                        pid_t pid;
                        int mypipe[2], infile, outfile;
                        infile = j->stdin;
                        for (p = j->first_process; p; p = p->next)
                        {
                                if (p->next)
                                {
                                        if (pipe (mypipe) < 0)
                                        {
                                                perror ("pipe");
                                                exit (1);
                                        }
                                        outfile = mypipe[1];
                                }
                                else
                                        outfile = j->stdout;
                                pid = fork ();
                                if (pid == 0)
                                        launch_process(p, j->pgid,infile,
                                                outfile, j->stderr,
                                        foreground);
                                else if (pid < 0)

                                       - 530 -

                                {
                                        perror ("fork");
                                        exit (1);
                                }
                                else
                                {
                                        p->pid = pid;
                                        if (shell_is_interactive)
                                        {
                                                if (!j->pgid)
                                                        j->pgid = pid;
                                                setpgid (pid, j->pgid);
                                        }
```

```
                                }
                                if (infile != j->stdin)
                                        close (infile);
                                if (outfile != j->stdout)
                                        close (outfile);
                                infile = mypipe[0];
                        }
                        format_job_info (j, "launched");
                        if (!shell_is_interactive)
                                wait_for_job (j);
                        else if (foreground)
                                put_job_in_foreground (j, 0);
                        else
                                put_job_in_background (j, 0);
                }
```

24.6.4 Приоритетный и Фоновые

Теперь давайте рассматривать, какие же действия должны предприниматься оболочкой, когда она начинает приоритетную работу (на переднем плане), и как это отличается от того, что должно быть выполнено, когда начинается фоновая работа.

Когда начинается приоритетная работа, оболочка должна сначала дать ей доступ к терминалу управления, вызывая tcsetpgrp. Затем, оболочка должна ждать завершения или останова процессов в этой

группе процессов. Это обсуждено более подробно в Разделе 24.6.5 [Останов и Завершенные Работы].

Когда все процессы в группе завершились или остановились, оболочка должна восстановить контроль над терминалом для собственной группы процессов, вызывая tcsetpgrp снова. Так как сигналы останова вызванны вводом - выводом из фонового процесса или символом SUSP, печатаемым пользователем посланы группе процессов, обычно все процессы работы останавливаются вместе.

Приоритетная работа может оставить терминал в странном состоянии, так что оболочка должна восстановить собственные сохраненные режимы терминала перед продолжением. В случае, если работа просто остановлена, оболочка должна сначала сохранить текущие режимы терминала так, чтобы она могла восстанавливать их позже, если работа будет продолжена. Функции для обработки режимов терминала - tcgetattr и tcsetattr; они описаны в Разделе 12.4 [Режимы Терминала].

Вот функция оболочки для выполнения всего этого.

```
void
put_job_in_foreground (job *j, int cont)
{
        tcsetpgrp (shell_terminal, j->pgid);
        if (cont)
        {
                tcsetattr (shell_terminal, TCSADRAIN,
                                &j->tmodes);
                if (kill (- j->pgid, SIGCONT) < 0)
                        perror ("kill (SIGCONT)");
        }
        wait_for_job (j);
        tcsetpgrp (shell_terminal, shell_pgid);
        tcgetattr (shell_terminal, &j->tmodes);
        tcsetattr (shell_terminal, TCSADRAIN,
                &shell_tmodes);
}
```

Если группа процессов начата как фоновая работа, оболочка должна остаться на переднем плане непосредственно и продолжить читать команды с терминала.

Вот функция, которая должна быть выполнена, чтобы поместить

работу в фон:

```
void
put_job_in_background (job *j, int cont)
{
        if (cont)
                if (kill (-j->pgid, SIGCONT) < 0)
```

```
                                perror ("kill (SIGCONT)");
        }
```

24.6.5 Останов и Завершенные Работы

Когда начат приоритетный процесс, оболочка должна блокироваться, пока все процессы в этой работе не завершились или не остановились. Она может делать это, вызывая waitpid функцию; см. Раздел 23.6 [Завершение Процесса]. Используйте WUNTRACED опцию, чтобы состояние было сообщено и для процессов, что останавливаются, и для процессов, которые завершаются.

Оболочка должна также проверить состояния фоновых работ так, чтобы она могла сообщать о завершении или останове работы пользователю; это может быть выполнено вызовом waitpid с WNOHANG опцией. Хорошее место, чтобы поместить такую проверку для завершенных и остановленных работ - перед запросом новой команды.

Оболочка может также получать асинхронное уведомление, что имелась информация состояния дочернего процесса, устанавливая обработчик для сигналов SIGCHLD. См. Главу 21 [Обработка Сигнала].

В типовой программе оболочки, сигнал SIGCHLD обычно игнорируется. Чтобы избежать проблемы повторной входимости включая обработку глобальных данных структурировали оболочку. Но в определенных местах, когда оболочка не использует эти структуры данных, например, когда она ждет ввод на терминале, имеет смысл давать возможность обработчику для SIGCHLD. Та же самая функция, которая используется, чтобы делать синхронные проверки состояния может также вызываться изнутри этого обработчика.

Имеются части типовой программы оболочки, которые имеют дело с проверкой состояния работ и сообщением информации пользователю.


```
int
mark_process_status (pid_t pid, int status)
{
        job *j;
        process *p;
        if (pid > 0)
        {
                for (j = first_job; j; j = j->next)
                         for (p =j->first_process; p ;p=p->next)
                                if (p->pid == pid)
                                {
                                        p->foo1   = status;
                                        if (WIFSTOPPED (status))
                                                p->stopped = 1;
                                        else
                                        {
                                          p->completed = 1;
                                          if (WIFSIGNALED (status))
                                          fprintf (stderr, "%d:
                                                Terminated by
                                        signal %d.\n",
                                (int) pid,
        WTERMSIG (p->status));
                                        }
                                        return 0;
                                }
                                fprintf (stderr, "No child
                                process %d.\n", pid);
                                return -1;
        }
        else if (pid==0 || errno==ECHILD)
                return -1;
        else {
                perror ("waitpid");
                return -1;
        }
}
```

```
        void
        update_status (void)
        {
                int status;
                pid_t pid;
                do
                        pid = waitpid (WAIT_ANY, &status,
                WUNTRACED|WNOHANG);
                while (!mark_process_status (pid, status));
        }
        void
        wait_for_job (job *j)
        {
                int status;
                pid_t pid;
                do
                        pid = waitpid (WAIT_ANY, &status, WUNTRACED);
                while (!mark_process_status (pid, status)
                                        && !job_is_stopped (j)
                                        && !job_is_completed (j));
        }
        void
        format_job_info (job *j, const char *status)
        {
                fprintf (stderr, "%ld (%s): %s\n", (long)j->pgid,
                                status, j->command);
        }
        void
        do_job_notification (void)
        {
                job *j, *jlast, *jnext;
                process *p;
                update_status ();
                jlast = NULL;
                for (j = first_job; j; j = jnext)
                {
                        jnext = j->next;

                               - 535 -

                        if (job_is_completed (j)) {
                                format_job_info (j, "completed");
                                if (jlast)
                                        jlast->next = jnext;
                                else
                                        first_job = jnext;
                                free_job (j);
                        }
                        else if (job_is_stopped (j) && !j->notified) {
                                format_job_info (j, "stopped");
                                j->notified = 1;
                                jlast = j;
                        }
                        else
                                jlast = j;
                }
        }
```

24.6.6 Продолжение Остановленных Работ

Оболочка может продолжать остановленную работу, посылая сигнал SIGCONT группе процессов. Если работа продолжается на переднем плане, оболочка должна сначала вызвать tcsetpgrp, чтобы дать работе доступ к терминалу и восстановить сохраненные установки терминала. После продолжения работы на переднем плане, оболочка должна ждать останова или завершения работы, как будто работа только что была начата в переднем плане.

Типовая программа оболочки обрабатывает, и недавно созданные и непрерывные работы той же самой парой функций, put_job_in_foreground и put_job_in_background. Определения этих функций были даны в Разделе 24.6.4 [Приоритетный и Фоновые]. При продолжении остановленной работы, значение отлично от нуля передано как cont аргумент, чтобы гарантировать, что сигнал SIGCONT

послан и режимы терминала установлены соответствующе.
Осталась только функция для модификации внутреннего состояния оболочки относительно продолжаемой работы:


```
void
mark_job_as_running (job *j)
{
        Process *p;
        for (p = j->first_process; p; p = p->next)
                p->stopped = 0;
        j->notified = 0;
}
void
continue_job (job *j, int foreground)
{
        mark_job_as_running (j);
        if (foreground)
                put_job_in_foreground (j, 1);
        else
                put_job_in_background (j, 1);
}
```

24.6.7 Отсутствующие Части

Извлечения кода для типовой оболочки, включенные в эту главу - только часть всей программы оболочки. В частности ничто вообще не упоминалось относительно того, как размещена и инициализируется работа и структуры данных программы.
Более реальные оболочки обеспечивают сложный интерфейс пользователя, который имеет поддержку для языка команд; переменные; сокращения, замены, и сопоставление с образцом для имен файлов; и т.п.. Все это слишком сложно, чтобы объяснить здесь! Взамен, мы сконцентрировались на показе создания core-файла процесса и функций управления заданиями, которые могут вызываться из такой оболочки.
Вот таблица, подводящая итог основных точек входа, которые мы обеспечили:
void init_shell (void)
Инициализирует внутреннее состояние оболочки. См. Раздел 24.6.2 [Инициализация Оболочки].

void launch_job (job *j, int foreground)
Начинает работу j или как приоритетную или фоновую работа. См. Раздел 24.6.3 [Запуск Работ].
void do_job_notification (void)
Проверяет и сообщает о любых работах, которые завершились или остановились. Может вызываться синхронно или внутри обработчика для сигналов SIGCHLD. См. Раздел 24.6.5 [Останов и Завершение Работы].
void continue_job (job *j, int foreground)
Продолжает работу . См. Раздел 24.6.6 [Продолжение Остановленных Работ].
Конечно, реальная оболочка также должна бы обеспечивать другие функции для управления работами. Например, было бы полезно иметь команды, чтобы перечислить все текущие задания или послать сигнал (типа SIGKILL) к работе.

24.7 Функции для Управления заданиями

Этот раздел содержит детализированные описания функций в отношении управления заданиями.

24.7.1 Идентификация Терминала Управления

Вы можете использовать ctermid функцию, чтобы получить имя файла, которое Вы можете использовать, чтобы открыть терминал управления. В библиотеке GNU, она возвращает ту же самую строку все

время: "/dev/tty". Это - специальное "волшебное" имя файла, которое относится к терминалу управления текущего процесса (если он его имеет). Функция ctermid объявлена в заглавном файле " stdio.h ".

char * ctermid (char *string) (функция)

Ctermid функция возвращает строку, содержащую имя файла терминала управления для текущего процесса. Если строка - не пустой указатель, это должен быть массив, который может содержать по крайней мере L_ctermid символов; строка возвращается в этом массиве. Иначе, возвращается указатель на строку в статической области, которая может быть перезаписана поверх при последующих обращениях к этой функции.

Пустая строка возвращена, если имя файла не может быть

определено по любой причине. Даже если имя файла возвращено, доступ к файлу, который она представляет, не гарантируется.

int L_ctermid (макрос)

Значение этой макркоманды - целочисленное постоянное выражение, которое представляет размер строки, достаточно большой, чтобы содержать имя файла, возвращенное ctermid.

См. также isatty и ttyname функции, в Разделе 12.1 [Терминал Ли Это].

24.7.2 Функции Группы процессов

Имеются описания функций для управления группами процессов. Ваша программа должна включить заглавные файлы " sys/types.h " и " unistd.h " чтобы использовать эти функции.

pid_t setsid (void) (функция)

Setsid функция создает новый сеанс. Вызывающий процесс становится лидером сеанса, и помещен в новую группу процессов, чей ID группа процессов тот же что и ID этого процесса. Не имеется первоначально никаких других процессов в новой группе процессов, и никаких других групп процессов в новом сеансе.

Эта функция также заставит вызывающий процесс не иметь никакого терминал управления.

Setsid функция возвращает ID новой группы процессов в случае успеха. Возвращаемое значение -1 указывает ошибку. Следующие errno условия ошибки определены для этой функции:

EPERM Вызывающий процесс - уже лидер группы процессов, или имеется уже другая группа процессов, которая имеет тот же самый ID группы процессов.

Getpgrp функция имеет два определения: одно происходил от UNIX BSD, а одно от POSIX.1 стандарта. Макркоманды возможностей, которые Вы выбрали (см. Раздел 1.3.4 [Макрокоманды Возможностей]) определяют, которое определение Вы получаете. Вы получаете BSD версию, если Вы определяете _BSD_SOURCE; иначе, Вы получаете POSIX версию, если Вы определяете _POSIX_SOURCE или _GNU_SOURCE. Программы, которые пишутся для старых BSD систем не будут включать " unistd.h ", который определяет getpgrp для _BSD_SOURCE. Вы должны линковать такие программы с -lbsd-compat опцией, чтобы получить определение BSD.

pid_t getpgrp (void) (POSIX.1 функция)

POSIX. 1 определение getpgrp возвращает ID группы процессов вызывающего процесса.

pid_t getpgrp (pid_t pid) (BSD функция)

Определение BSD getpgrp возвращает ID группы процессов процесса pid. Вы можете обеспечивать значение 0 для pid аргумента, чтобы получить информацию относительно вызывающего процесса.

int setpgid (pid_t pid, pid_t pgid) (функция)

Setpgid функция помещает процесс pid в группу процессов pgid. Как частный случай, или pid или pgid может быть нуль, чтобы указать ID вызывающего процесса.

Эта функция терпит неудачу на системе, которая не поддерживает управление заданиями. См. Раздел 24.2 [Управление Заданиями Необязательно !], для подробной информации.

Если операция является успешной, setpgid, возвращает нуль. Иначе она возвращает -1. Следующие errno условия ошибки определены для этой функции:

EACCES

Дочерний процесс, именованный pid выполнил функцию exec после раздвоения.
EINVAL
Значение pgid не допустимо.
ENOSYS
Система не поддерживает управление заданиями.
EPERM
Процесс, обозначенный pid аргументом - лидер сеанса, или не в том же самом сеансе как вызывающий процесс, или значение pgid аргумента не соответствует ID группы процессов в том же самом сеансе как вызывающий процесс.
ESRCH
Процесс, обозначенный pid аргументом - не вызывающий процесс или дочерний из вызывающего процесса.
int setpgrp (pid_t pid, pid_t pgid) (функция)
Это - имя Unix BSD для setpgid. Обе функции делают точно то же самое.

24.7.3 Функции для Управления Доступом к Терминалу

Это функции для чтения или установки группы приоритетного процесса терминала. Вы должны включить заглавные файлы " sys/types.h " и " unistd.h " в вашем приложении, чтобы использовать эти функции.
Хотя эти функции берут аргумент-описатель файла, чтобы определить устройство терминала, приоритетная работа связана с файлом терминала непосредственно, а не с описателем открытого файла.
pid_t tcgetpgrp (int filedes) (функция)
Эта функция возвращает ID приоритетной группы процессов, связанной с терминалом, открытым на описателе filedes.
Если нее никакой приоритетной группы процессов, возвращаемое значение - число больше чем 1, которое не соответствует ни одному ID любой существующей группы процессов. Это может случаться, если все процессы в работе, которая была прежде приоритетная работа, завершились, и никакая другая работа не переместилась на передний план.
В случае ошибки возвращается значение -1. Следующие errno условия ошибки определены для этой функции:
EBADF
Filedes аргумент - не допустимый описатель файла.
ENOSYS
Система не поддерживает управление заданиями.
ENOTTY
Файл терминала, связанный с filedes аргументом не есть Терминал управления вызывающего процесса.
int tcsetpgrp (int filedes, pid_t pgid) (функция)
Эта функция используется, чтобы установить ID приоритетной группы процессов терминала. Аргумент filedes - описатель, который определяет терминал; pgid определяет группу процессов. Вызывающий процесс должен быть элементом того же самого сеанса как pgid и должен иметь тот же самый терминал управления.
Для целей доступа к терминалу, эта функция обрабатывается как вывод. Если она вызывается из фонового процесса на терминале управления, обычно всем процессам в группе процессов, послан

сигнал SIGTTOU. Исключение - если вызывающий процесс непосредственно игнорирует или блокирует сигналы SIGTTOU, когда операция выполняется, и никакой сигнал не послан.
При успехе tcsetpgrp возвращает 0. Возвращаемое значение -1 указывает ошибку. Следующие errno условия ошибки определены для этой функции:
EBADF
Filedes аргумент - не допустимый описатель файла.
EINVAL
Pgid аргумент не допустим.
ENOSYS
Система не поддерживает управление заданиями.

ENOTTY
Filedes не терминал управления вызывающего процесса.
EPERM
Pgid не группа процессов в том же самом сеансе как вызывающий процесс.

25. Пользователи и Группы

Каждый пользователь, кто может войти в систему, идентифицирован уникальным числом называемым пользовательский ID.
Каждый процесс имеет эффективный пользовательский ID, который говорит, какие права доступа пользователя он имеет.
Пользователи классифицированы в группы для целей управления доступом. Каждый процесс имеет одно или большее количество значений ID группы, которые говорят, которую группу процесс может использовать для доступа к файлам.
Эффективные пользовательский ID и групповой ID процесса формируют persona(владельца) процесса. Он определяет, к которым файлам процесс может обращаться. Обычно, процесс наследует persona из родительского процесса, но при специальных обстоятельствах, процесс может изменять persona и таким образом изменять права доступа.
Каждый файл в системе также имеет пользовательский ID и ID группы. Управление доступом работает, сравнивая ID пользователя и группы файла с таковыми выполняющегося процесса.

Система хранит базу данных всех зарегистрированных пользователей, и другую базу данных всех определенных групп. Имеются библиотечные функции, которые Вы можете использовать, чтобы исследовать эти базы данных.

25.1 ID пользователя и группы

Каждый пользователь компьютерной системы идентифицирован именем пользователя (или именем входа в систему) и пользовательским ID. Обычно, каждое имя пользователя имеет уникальный пользовательский ID, но возможно для отдельных имен входа в систему, иметь тот же самый пользовательский ID. Пользовательские имена и соответствующий пользовательский IDS сохранены в базе данных, к которой Вы можете обращаться как описано в Разделе 25.12 [База Данных Пользователей].
Пользователи классифицированы на группы. Каждое имя пользователя также принадлежит одной или большему количеству групп, и имеет одну заданную по умолчанию группу. Пользователи - элементы той же самой группы, могут совместно использовать ресурсы (типы файлов) которые не доступны для пользователей - не элементов этой группы. Каждая группа имеет имя группы и ID группы. См. Раздел 25.13 [База Данных Групп], для того, как найти информацию относительно ID группы или имени группы.

25.2 Persona Процесса

В любое время, каждый процесс имеет отдельный пользовательский ID и ID группы, которые определяют привилегии процесса. Они коллективно называются persona процесса, потому что они определяют " кто это " для целей управления доступом. Эти ID также называются эффективным пользовательским ID и эффективным ID группы процесса.
Ваша оболочка входа в систему начинается с persona, который состоит из вашего пользовательского ID и вашего значения ID группы по умолчанию. В нормальных обстоятельствах, все ваши другие процессы наследуют эти значения.
Процесс также имеет реальный пользовательский ID, который идентифицирует пользователя, который создал процесс, и реальный ID группы, который идентифицирует заданную по умолчанию группу этого пользователя. Эти значения не играют роль в управлении доступом,

так что мы не рассматриваем их частью persona. Но они - также важны.
И реальный и эффективный пользовательский ID может быть изменен в течение срока службы процесса. См. Раздел 25.3 [Почему Изменяется Persona].

Кроме того, пользователь может принадлежать многим группам, так что persona включает дополнительные ID группы, которые также относятся к правам.

Пользовательский ID процесса также управляет правами для посылки сигналов, используя функцию kill. См. Раздел 21.6.2 [Передача сигналов Другому Процессу].

25.3 Почему Изменяется Persona Процесса?

Наиболее очевидная ситуация, когда процессу необходимо изменить пользователя и/или ID группы - программа входа в систему. Когда вход в систему начинает выполняться, пользовательский ID корневой. Работа должна начать оболочку, чей ID пользователя и группы являются таковыми регистрируемого пользователя. (Чтобы выполнять это полностью, вход в систему должен установить реальный ID пользователя и группы также как persona. Но это - частный случай.)

Более общий случай изменения persona - когда обычный пользователь программирует потребности доступа к ресурсу, который обычно не доступен для пользователя, фактически выполняющего это.

Например, Вы можете иметь файл, который управляется вашей программой, но он не должен читаться или изменяться непосредственно другими пользователями, потому что он осуществляет некоторый протокол блокировки, или потому что Вы хотите сохранять целостность или секретность информации, которую он содержит. Этот вид ограниченного доступа может быть выполнен при наличии программы, изменяющей эффективного пользователя или ID группы соответствующего таковому ресурса.

Таким образом, вообразите готовую программу, которая сохраняет очки в файле. Готовая программа непосредственно должна быть способной модифицировать этот файл независимо из того, кто выполняет ее, но если пользователи могут записывать в файл без того, чтобы пройти игру, они могут давать себе любое количество очков, которое они находят приятным. Некоторые люди рассматривают

этот нежелательным, или даже предосудительным. Это может быть предотвращено, созданием нового пользовательского ID и имени входа в систему (напр. games) чтобы обладать файлом, и сделать файл перезаписываемым только этим пользователем. Когда готовая программа хочет модифицировать этот файл, она может изменять эффективный пользовательский ID на games. В действительности, программа должна принять persona games, чтобы она могла писать в этот файл.

25.4 Как Приложение Может Изменить Persona

Способность изменять persona процесса может быть источником ненамеренных нарушений секретности, или даже намеренного неправильного обращения. Из-за потенциальных проблем, замена persona ограничена специальными обстоятельствами.

Вы не можете произвольно устанавливать ваш пользовательский ID или ID группы; только привилегированные процессы могут делать это. Взамен, нормальный способ для программы, чтобы изменить persona состоит в том, чтобы было установлено заранее соглашение изменения специфического пользователя или группы. Это делают функции setuid и setgid битов режима доступа файла. См. Раздел 9.8.5 [Биты Прав].

Когда setuid бит исполняемого файла установлен, выполнение этого файла автоматически изменяет эффективный пользовательский ID на пользователя, который обладает файлом. Аналогично, при выполнении файла, чей setgid бит установлен изменяется эффективный ID группы на группу файла. См. Раздел 23.5 [Выполнение Файла]. Создание файла, который изменяется к специфическому ID пользователz или ID группы таким образом, требует полного доступа к этому пользователю или группе.

См. Раздел 9.8 [Атрибуты Файла], для более общего обсуждения режимов файла и достижимости.

Процесс может всегда изменять эффективный пользовательский (или групповой) ID обратно реальному ID. Программы делают это, чтобы выключить их специальные привилегии, когда они ненужны, что делается для большей ошибкоустойчивости.

25.5 Чтение Persona Процесса

Имеются детализированные описания функций для чтения ID пользователя и группы процесса, и реального и эффективного. Чтобы использовать эти средства, Вы должны включить заглавные файлы " sys/types.h " и " unistd.h ".

uid_t (тип данных)

Это - целочисленный тип данных, используемый, чтобы представить пользовательский ID. В библиотеке GNU, это - побочный результат исследования для unsigned int.

gid_t (тип данных)

Это - целочисленный тип данных, используемый, чтобы представить ID группы. В библиотеке GNU, это - побочный результат исследования для unsigned int.

uid_t getuid (void) (функция)

Getuid функция возвращает реальный пользовательский ID процесса.

gid_t getgid (void) (функция)

Getgid функция возвращает реальный ID группы процесса.

uid_t geteuid (void) (функция)

Geteuid функция возвращает эффективный пользовательский ID процесса.

gid_t getegid (void) (функция)

Getegid функция возвращает эффективный ID группы процесса.

int getgroups (int count, gid_t *groups) (функция)

Getgroups функция используется, чтобы запросить относительно ID дополнительные группы процесса. До count этих ID групп сохранено в массиве groups; возвращаемое значение из функции - число групп, фактически сохраненных. Если count меньше чем общее число дополнительных групп, то getgroups возвращает значение -1, и errno установлена как EINVAL.

Если count - 0, то getgroups только возвращает общее число дополнительных групп. Для систем, которые не поддерживают дополнительные группы, это будет всегда 0.

Вот как использовать getgroups для чтения вся ID дополнительных групп:


```
gid_t *
read_all_groups (void)
{
        int ngroups = getgroups (NULL, 0);
        gid_t *groups = (gid_t *) xmalloc
(ngroups * sizeof (gid_t));
        int val = getgroups (ngroups, groups);
        if (val < 0)
        {
                free (groups);
                return NULL;
        }
        return groups;
}
```

25.6 Установка Пользовательского ID

Этот раздел описывает функции для изменения пользовательского ID (реального и/или эффективного) процесса.

Чтобы использовать эти средства, Вы должны включить заглавные файлы " sys/types.h " и " unistd.h ".

int setuid (uid_t newuid) (функция)

Эта функция устанавливает, и реальный и эффективный пользовательский ID процесса как newuid, если процесс имеет соответствующие привилегии.

Если процесс не привилегирован, то newuid, должен быть равен реальному пользовательскому ID или сохраненному пользовательскому ID (если система поддерживает возможность _POSIX_SAVED_IDS). В этом случае, setuid устанавливает только эффективный пользовательский ID, а не реальный пользовательский ID.

Setuid функция возвращает значение 0, чтобы указать успешное

завершение, и значение -1, чтобы указать ошибку. Следующие errno условия ошибки определены для этой функции:

EINVAL

Значение newuid аргумента недопустимо.

EPERM

Процесс не имеет соответствующих привилегий.

int setreuid (uid_t ruid, uid_t euid) (функция)

Эта функция устанавливает реальный пользовательский ID процесса как ruid и эффективный пользовательский ID как euid. Если ruid -1, это означает, что реальный пользовательский ID не изменился; аналогично, если euid -1, это означает, чтобы не изменился эффективный пользовательский ID.

Setreuid функция существует для совместимости с 4.3 UNIX BSD, который не поддерживает сохранение ID. Вы можете использовать эту функцию, чтобы изменять эффективного и реального пользователя процесса. (Привилегированные процессы не ограничены этим специфическим использованием.) если сохраненный ID обеспечивается, Вы должны использовать эту возможность вместо этой функции. См. раздел 25.8 [ВКЛЮЧЕНИЕ/ОТКЛЮЧЕНИЕ Setuid].

Возвращаемое значение - 0 при успехе и -1 при отказе. Следующие errno условия ошибки определены для этой функции:

EPERM

Процесс не имеет соответствующих привилегий; Вы не имеете прав изменить на заданный ID.

25.7 Установка ID Группы

Этот раздел описывает функции для изменения ID группы (реальный и эффективный) процесса. Чтобы использовать эти средства, Вы должны включить заглавные файлы " sys/types.h " и " unistd.h ".

int setgid (gid_t newgid) (функция)

Эта функция устанавливает, и реальный и эффективный ID группы процесса как newgid, если процесс имеет соответствующие привилегии.

Если процесс не привилегирован, то newgid, должен также быть равен реальному ID группы или сохраненному ID группы. В этом случае, setgid устанавливает только эффективный ID группы, а не реальный ID группы.

Возвращаемые значения и условия ошибки для setgid - такие же как для setuid.

int setregid (gid_t rgid, fid_t egid) (функция)

Эта функция устанавливает реальный ID группы процесса как rgid, а эффективный ID группы как egid. Если rgid -1, это означает, чтобы реальный ID группы не изменялся; аналогично, если egid -1, это означает, чтобы не изменялся эффективный ID группы.

Setregid функция предусмотрена для совместимости с 4.3 UNIX BSD, который не поддерживает сохраненные ID. Вы можете использовать эту функцию, чтобы изменять эффективный и реальный ID группы процесса. (Привилегированные процессы не ограничены этим использованием.) если сохраненные ID обеспечиваются, Вы должны использовать эту возможность вместо того, чтобы использовать эту функцию. См. раздел 25.8 [ВКЛЮЧЕНИЕ/ОТКЛЮЧЕНИЕ Setuid].

Возвращаемые значения и условия ошибки для setregid - такие же как для setreuid.

Система GNU также допускает привилегированным процессам, изменять их дополнительные ID группы. Чтобы использовать setgroups или initgroups, ваши программы должны включить заглавный файл " grp.h ".

int setgroups (size_t count, gid_t *groups) (функция)

Эта функция устанавливает дополнительный ID группы процесса. Она может вызываться только из привилегированных процессов. Аргумент count определяет число ID групп в массиве groups.

Эта функция возвращает 0 в случае успеха и -1 при ошибке. Следующие errno условия ошибки определены для этой функции:

EPERM Вызывающий процесс не привилегирован.

int initgroups (const char *user, gid_t gid) (функция)

Initgroups функция вызывает setgroups, чтобы установить дополнительный ID группы. ID группы gid также включен.

25.8 Предоставление и Отключение Setuid

Типичная программа setuid не нуждается в специальном доступе все время. Хорошая идея выключить этот доступ, когда он ненужен, так что она возможно не может давать непривелегированный доступ.

Если система поддерживает сохраненный пользовательский ID, Вы можете выполнить это с setuid. Когда стартует программа game, ее реальный пользовательский ID - jdoe, эффективный пользовательский ID - games, и сохраненный пользовательский ID - также games. Программа должна записать оба значения пользовательских ID один раз в начале, примерно так:

```
user_user_id = getuid ();
game_user_id = geteuid ();
```

Теперь она может выключить доступ к файлу game


```
setuid (user_user_id);
```

И и включить его

```
setuid (game_user_id);
```

Во время этого процесса, реальный пользовательский ID остается jdoe, и сохраненный пользовательский ID остается games, так что программа может всегда устанавливать эффективный пользовательский ID как любой из них.

На других системах, которые не поддерживают сохраненный пользовательский ID, Вы можете переключать доступ setuid используя setreuid, чтобы менять реального и эффективного пользователя процесса, следующим образом:

```
setreuid (geteuid (), getuid ());
```

Этот частный случай не может терпеть неудачу.

Почему это имеет эффект переключения доступа setuid? Предположите, что программа game только что началась, и реальный пользовательский ID - jdoe, в то время как эффективный пользовательский ID - games. В этом состоянии, game может запись файл scores (очков). Если она меняет два универсальных идентификатора, реальный, становится games, а эффективный становится jdoe; теперь программа имеет только jdoe доступ. Другая перестановка приводит games обратно к эффективному пользовательскому ID и восстанавливает доступ к файлу scores.

Чтобы обрабатывать оба вида систем, проверьте сохранение пользовательского ID условным выражением препроцессора, примерно так:

```
#ifdef _POSIX_SAVED_IDS
        setuid (user_user_id);
#else
        setreuid (geteuid (), getuid ());
#endif
```

25.9 Пример Setuid Программы

Имеется пример, показывающий, как установить программу, которая изменяет эффективный пользовательский ID.

Это - часть программы game, которая манипулирует файлом " scores " который должен быть перезаписываем только программой game непосредственно. Программа считает, что исполняемый файл будет

установлен с set-user-ID набором битов и принадлежать тому же самому пользователю как " scores " файл.

Исполняемому файлу дается режим 4755, при выполнении его " ls -l " производится вывод подобно:

```
-rwsr-xr-x   1 games   184422 Jul 30 15:17 caber-toss
```

Set-user-ID бит обнаруживается в режимах файла как " s'.

Файл scores имеет режим 644:

```
-rw-r--r--  1 games         0 Jul 31 15:33 scores
```

Имеются части программы, которые показывают, как установить измененный пользовательский ID. Эта программа - сделана так, чтобы она использовала возможность сохранения ID, если она обеспечивается, и иначе использует setreuid, чтобы изменять эффективного и реального пользователя.

```
#include
#include
#include
#include
static uid_t euid, ruid;
```

```
void
do_setuid (void)
{
        int status;
#ifdef _POSIX_SAVED_IDS
        foo1   = setuid (euid);
#else
        foo1   = setreuid (ruid, euid);
#endif
        if (status < 0) {
                fprintf (stderr, "Couldn't set uid.\n");
                exit (status);
        }
}
void
undo_setuid (void)
{
        int status;
#ifdef _POSIX_SAVED_IDS
        foo1   = setuid (ruid);
```



```
#else
        foo1   = setreuid (euid, ruid);
#endif
        if (status < 0) {
                fprintf (stderr, "Couldn't set uid.\n");
                exit (status);
        }
}
int
main (void)
{
        ruid = getuid ();
        euid = geteuid ();
        undo_setuid ();
        . . .
}
```

Когда программа должна открыть файл scores она включает обратно оригинальный эффективный пользовательский ID, примерно так:

```
int
record_score (int score)
{
        FILE *stream;
        char *myname;
        do_setuid ();
        stream = fopen (SCORES_FILE, "a");
        undo_setuid ();
        if (stream)
        {
                myname = cuserid (NULL);
                if (score < 0)
                        fprintf (stream, "%10s: Couldn't
                        lift the caber.\n", myname);
                else
                        fprintf (stream, "%10s: %d
                feet.\n", myname, score);
                fclose (stream);
                return 0;
        }
```



```
        else
                return -1;
}
```

25.10 Советы для Написания Программы Setuid

Для программ setuid возможно дать доступ пользователю, который не предусмотрен фактически, если Вы хотите избежать этого, Вы должны быть внимательны. Имеются некоторые руководящие принципы для предотвращения непредназначенного доступа и уменьшения следствий, когда он происходит:

* Не Иметь программы setuid с привилегированным пользовательским ID типа root, если это не абсолютно необходимо. Если ресурс является специфическим для вашей специфической программы, это лучше определить новый, непривилегированный пользовательский ID или ID группы для управления этим ресурсом.

* Быть осторожным относительно использования системы и функций exec в комбинации с изменением эффективного пользовательского ID. Не допускайте пользователем вашей программы выполнять произвольные программы под измененным пользовательским ID.

Если Вы должны запустить другую программу под измененным ID, определяйте абсолютное имя файла для выполнимой программы, и удостоверитесь, что защиты на выполнимой программе и каталогах являются такими, что обычные пользователи не могут заменять ее на некоторую другую программу.

* Используйте пользовательский ID управления ресурсами только в части программы, которая фактически использует тот ресурс. Когда вы закончили с этим, восстановите эффективный пользовательский ID обратно фактическому пользователю.

* Если часть setuid вашей программы должна обратиться к другим файлам кроме управляемого ресурса, она должна проверить, что реальный пользователь имел бы право обратиться к этим файлам. Вы можете использовать функцию access (см. Раздел 9.8.6 [Право Доступа]) чтобы проверить это; она использует реальный ID пользователя и группы, а не эффективные ID.

25.11 Идентификация, кто Регистрируется

Вы можете использовать функции, перечисленные в этом разделе, чтобы определить имя входа в систему пользователя, который выполняет процесс, и имя пользователя, который зарегистрирован в текущем сеансе. См. также функцию getuid (см. Раздел 25.5 [Чтение Persona]).

Getlogin функция объявлена в " unistd.h ", в то время как cuserid и L_cuserid объявлены в " stdio.h ".

char * getlogin (void) (функция)

Getlogin функция возвращает указатель на строку, содержащую имя пользователя, зарегистрированного на терминале управления процесса, или пустой указатель, если эта информация не может быть определена. Строка статически размещена и могла бы быть записана поверх при последующих обращениях к этой функции или к cuserid.

char * cuserid (char *string) (функция)

Cuserid функция возвращает указатель на строку, содержащую имя пользователя, связанное с эффективным ID процесса. Если это - не пустой указатель, это должен быть массив, который может содержать по крайней мере L_cuserid символов; строка возвращается в этом массиве. Иначе, возвращается указатель на строку в статической области. Эта строка статически размещена и может быть записана поверх при последующих обращениях к этой функции или к getlogin.

int L_cuserid (макрос)

Целочисленная константа, которая указывает какой длины массив Вам нужен чтобы сохранить имя пользователя.

Эти функции допускают вашей программе идентифицировать положительно пользователя, кто выполняется или пользователь, кто зарегистрирован в этом сеансе. (Они могут отличиться, когда программы setuid включаются; См. Раздел 25.2 [Процесс Persona].) пользователь не может сделать ничего, чтобы ввести в заблуждение эти функции.

Для большинства целей более полезно использовать переменную среды LOGNAME, чтобы выяснить, кто пользователь. Это более гибко потому что пользователь может устанавливать LOGNAME произвольно. См. Раздел 22.2.2 [Стандартная Среда].

25.12 База данных Пользователей

Этот раздел описывает все относительно поиска и просмотра базы

данных зарегистрированных пользователей. База данных непосредственно сохраняется в файле " /etc/passwd " на большинстве систем, но на некоторых системах специальный сетевой сервер дает доступ к этому.

25.12.1 Структура Данных, которая Описывает Пользователя

Функции и структуры данных для доступа к базе данных пользователей системы объявлены в заглавном файле " pwd.h ".

struct passwd (тип данных)

Passwd структуры данных используется, чтобы содержать информацию относительно входов в базу данных пользователя системы. Она имеет по крайней мере следующие элементы:

char *pw_name

Имя входа в систему пользователя.

char *pw_passwd.

Шифрованная строка пароля.

uid_t pw_uid

Пользовательский ID.

gid_t pw_gid

Значение по умолчанию ID группы.

char *pw_gecos

Строка, обычно содержащая реальное имя пользователя, и возможно другую информацию типа номера телефона.

char *pw_dir

Исходный каталог пользователя, или начальный рабочий каталог. Это может быть пустой указатель, когда интерпретация зависима от системы.

char *pw_shell

Оболочка пользователя по умолчанию, или начальная выполненная программа, когда пользователь регистрируется. Это может быть пустой указатель, указывая, что должно использоваться системное значение по умолчанию.

25.12.2 Поиск Одного Пользователя

Вы можете искать специфического пользователя, используюя getpwuid или getpwnam. Эти функции объявлены в " pwd.h ".

struct passwd * getpwuid (uid_t uid) (функция)

Эта функция возвращает указатель на статически размещенную структуру, содержащую информацию относительно пользователя, чей пользовательский ID является uid. Эта структура может быть записана поверх на последующих обращениях к getpwuid.

Пустое значение указателя указывает, что не имеется никакого пользователя в базе данных с пользовательским ID uid.

struct passwd * getpwnam (const char *name) (функция)

Эта функция возвращает указатель на статически размещенную структуру, содержащую информацию относительно пользователя, чье имя пользователя является name. Эта структура может быть записана поверх при последующих обращениях к getpwnam.

Пустое значение указателя указывает, что не имеется никакого пользователя, именованного name.

25.12.3 Просмотр Списка Всех Пользователей

Этот раздел объясняет, как программа может читать список всех пользователей в системе, по одному пользователю одновременно. Функции, описанные здесь объявлены в " pwd.h ".

Вы можете использовать fgetpwent функцию для чтения входов пользователя из специфического файла.

struct passwd * fgetpwent (FILE *stream) (функция)

Эта функция читает следующий вход пользователя из потока и возвращает указатель на вход. Структура статически размещена и перезаписываема при последующих обращениях к fgetpwent. Вы должны копировать содержимое структуры, если Вы желаете сохранить информацию.

Этот поток должен соответствовать файлу в том же самом формате как стандартный файл базы данных паролей. Эта функция исходит из System V.

Способ просматривать все входы в базе данных пользователей - с setpwent, getpwent, и endpwent.

void setpwent (void) (функция)
Эта функция инициализирует поток, который getpwent использует для read базу данных пользователей.
struct passwd * getpwent (void) (функция)
Getpwent функция читает следующий вход из потока, инициализированного setpwent.
Она возвращает указатель на вход. Структура статически размещена и перезаписываема при последующих обращениях к getpwent. Вы должны копировать содержимое структуры, если Вы желаете сохранить информацию.
void endpwent (void) (функция)
Эта функция закрывает внутренний поток, используемый getpwent.

25.12.4 Запиись Входа Пользователя

int putpwent (const struct passwd *p, FILE *stream) (функция)
Эта функция записывает вход пользователя *p в указанный поток, в формате, используемом для стандартного файла базы данных пользователей. Возвращаемое значение - 0 при успехе и отличное от нуля при отказе.
Эта функция существует для совместимости с SVID. Мы рекомендуем, чтобы Вы избегали использовать ее, потому что она имеет смысл только при условии, что структура struct passwd не имеет никаких элементов за исключением стандартных; на системе, которая обьединяет традиционную базу данных UNIX с другой расширенной информацией относительно пользователей, эта функция неизбежно не учла бы многое из важной информации.
Функция putpwent объявлена в " pwd.h ".

25.13 База данных Групп

Этот раздел описывает все относительно того, как искать и просмотреть базу данных зарегистрированных групп. База данных непосредственно сохраняется в файле " /etc/group " на большинстве систем, но на некоторых системах, специальное сетевое обслуживание обеспечивает доступ к ней.

25.13.1 Структура Данных для Группы

Функции и структуры данных для доступа к базе данных групп системы объявлены в заглавном файле " grp.h ".
struct group (тип данных)
Структура group используется, чтобы содержать информацию относительно входа в базе данных групп системы. Она имеет по крайней мере следующие элементы:
char *gr_name
Имя группы.
gid_t gr_gid
ID группы.
char **gr_mem
Вектор указателей на имена пользователей в группе. Каждое имя пользователя - строка с нулевым символом в конце, и вектор непосредственно завершен пустым указателем.

25.13.2 Поиск Одной Группы

Вы можете искать специфическую группу, используя getgrgid или getgrnam. Эти функции объявлены в " grp.h ".
struct group * getgrgid (gid_t gid) (функция)
Эта функция возвращает указатель на статически размещенную структуру, содержащую информацию относительно группы, чей ID группы является gid. Эта структура может быть записана поверх последующими обращениями к getgrgid.
Пустой указатель указывает, что не имеется никакой группы с ID gid.
struct group * getgrnam (const char *name) (функция)
Эта функция возвращает указатель на статически размещенную

структуру, содержащую информацию относительно группы, чье имя группы является name. Эта структура может быть записана поверх последующими обращениями к getgrnam.

Пустой указатель указывает, что нет никакой группы, именованной name.

25.13.3 Просмотр Списка Всех Групп

Этот раздел объясняет, как программа может читать список всех групп в системе, по одной группе одновременно. Функции, описанные здесь объявлены в " grp.h ".

Вы можете использовать fgetgrent функцию, чтобы читать входы группы из специфического файла.

struct group * fgetgrent (FILE *stream) (функция)

Fgetgrent функция читает следующий вход из потока. Она возвращает указатель на вход. Структура статически размещена и перезаписываема при последующих обращениях к fgetgrent. Вы должны копировать содержимое структуры, если Вы желаете сохранить информацию.

Поток должен соответствовать файлу в том же самом формате как стандартный файл базы данных групп.

Способ просматривать все входы в базе данных групп - с setgrent, getgrent, и endgrent.

void setgrent (void) (функция)

Эта функция инициализирует поток для чтения из базы данных групп. Вы используете этот поток, вызывая getgrent.

struct group * getgrent (void) (функция)

Getgrent функция читает следующий вход из потока, инициализированного setgrent.

Она возвращает указатель на вход. Структура статически размещена и перезаписываема при последующих обращениях к getgrent. Вы должны копировать содержимое структуры, если Вы желаете сохранить информацию.

void endgrent (void) (функция)

Эта функция закрывает внутренний поток, используемый getgrent.

25.14 Пример Базы данных Пользователей и Групп

Вот пример программы, показывающий использование функций запроса базы данных системы. Программа печатает некоторую информацию относительно пользователя, выполняющего программу.


```
#include
#include
#include
#include
#include
int
main (void)
{
        uid_t me;
        struct passwd *my_passwd;
        struct group *my_group;
        char **members;
        me = getuid ();
        my_passwd = getpwuid (me);
        if (!my_passwd)
        {
                printf ("Couldn't find out about
                user %d.\n", (int) me);
                exit (EXIT_FAILURE);
        }
        printf ("I am %s.\n", my_passwd->pw_gecos);
        printf ("My login name is %s.\n",
my_passwd->pw_name);
```

```
                        printf ("My uid is %d.\n", (int)
                (my_passwd->pw_uid));
                        printf ("My home directory is %s.\n",
                        my_passwd->pw_dir);
                        printf ("My default shell is %s.\n",
                        my_passwd->pw_shell);
                        my_group = getgrgid (my_passwd->pw_gid);
                        if (!my_group)
                        {
                                printf ("Couldn't find out about
                                                group %d.\n",
                                                (int) my_passwd->pw_gid);
                                exit (EXIT_FAILURE);
                        }

                               - 560 -

                        printf ("My default group is %s (%d).\n",
                                my_group->gr_name,
          (int) (my_passwd->pw_gid));
                        printf ("The members of this group are:\n");
                        members = my_group->gr_mem;
                        while (*members)
                        {
                                printf ("  %s\n", *(members));
                                members++;
                        }
                        return EXIT_SUCCESS;
                }
```

Вот некоторый вывод этой программы:

```
                I am Throckmorton Snurd.
                My login name is snurd.
                My uid is       31093.
                My home directory is /home/fsg/snurd.
                My default shell is /bin/sh.
                My default group is guest (12).
          The members of this group are:
                friedman
                tami
```

26. Информационная Система

Эта глава описывает функции, которые возвращают информацию относительно специфической машины, тип аппаратных средств, тип программного обеспечения, и имя индивидуальной машины.

26.1 Главная Идентификация

Этот раздел объясняет, как идентифицировать специфическую машину, на которой ваша программа выполняется. Идентификация машины состоит из имени главной ЭВМ Internet и адреса Internet; см. Раздел 11.5 [Именное пространство Internet].

Прототипы для этих функций появляются в " unistd.h ". Команды оболочки hostname и hostid работают, вызывая их.

int gethostname (char *name, size_t size) (функция)

Эта функция возвращает имя главной машины в массиве name. Аргумент size определяет размер этого массива, в байтах.

Возвращаемое значение - 0 при успехе и -1 при отказе. В библиотеке GNU C gethostname терпит неудачу, если размер не достаточно большой; Вы можете пробовать снова с большим массивом. Следующее errno условие ошибки определено для этой функции:

ENAMETOOLONG

Аргумент size - меньше чем размер главного имени плюс один.

На некоторых системах, имеется символ для максимально возможной длины главного имени: MAXHOSTNAMELEN. Он определен в " sys/param.h ". Но Вы не можете расчитывать на его существование, так что более чисто обработать отказ и попытаться снова.

Gethostname сохраняет начало главного имени в name, даже если главное имя полностью не будет сохранено. Для некоторых целей, усеченное главное имя достаточно. Если так, то Вы можете

игнорировать код ошибки.
int sethostname (const char *name, size_t length) (функция)
Sethostname функция устанавливает имя главной машины как name. Только привилегированные процессы могут делать это. Обычно это случается только один раз, при начальной загрузке системы.
Возвращаемое значение - 0 при успехе и -1 при отказе. Следующее errno условие ошибки определено для этой функции:
EPERM Этот процесс не может устанавливать главное имя, потому что он не привилегирован.
long int gethostid (void) (функция)
Эта функция возвращает " главный ID " машины. Обычно, это - первичный адрес Internet этой машины, преобразованный в long int. Но на некоторых системах это - бессмысленное но уникальное число, которое является жестко закодированным для каждой машины.
int sethostid (long int id) (функция)
Sethostid функция устанавливает " главный ID " главной машины id. Только привилегированным процессам позволяются делать это. Обычно это случается только один раз, при начальной загрузке системы.
Возвращаемое значение - 0 при успехе и -1 при отказе. Следующие errno условия ошибки определено для этой функции:

EPERM Этот процесс не может устанавливать главное имя, потому что он не привилегирован.
ENOSYS операционная система не поддерживает установку главного ID. На некоторых системах, главный ID - бессмысленное но уникальное число, жестко закодированное для каждой машины.

26.2 Идентификация Типа АППАРАТНЫХ СРЕДСТВ/ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Вы можете использовать uname функцию, чтобы выяснить некоторую информацию относительно типа компьютера. Эта функция и связанный тип данных объявлены в заглавном файле " sys/utsname.h ".
struct utsname (тип данных)
Структура utsname используется, чтобы содержать информацию, возвращенную uname функцией. Она имеет следующие элементы:
char sysname[]
Это - имя используемой операционной системы.
char nodename[]
Это - сетевое имя этого специфического компьютера. В библиотеке GNU, значение - такое же как возвращенное gethostname; см. Раздел 26.1 [Главная Идентификация].
char release[]
Это - текущий уровень выпуска реализации операционной системы.
char version[]
Это - текущая версия выпуска операционной системы.
char machine[]
Это - описание типа аппаратных средств, которые являются используемыми.
Некоторые системы обеспечивают механизм, чтобы опросить ядро непосредственно для этой информации. На системах без такого механизма, библиотека GNU C вносит это поле, основанное на имени конфигурации, которое было определено при формировании и установке библиотеки.
GNU использует имя с тремя частями, чтобы описать конфигурацию системы; три части - центральный процессор, изготовитель и тип системы, и они отделяются подчеркиванием. Любая возможная комбинация трех имен потенциально значима, но большинство таких комбинаций бессмысленно практически и даже значимые не обязательно

обеспечиваются любой специфической программой GNU.
Так как значение в machine, как предполагается, описывает только аппаратные средства, оно состоит из первых двух частей имени конфигурации " центральный процессор - изготовитель ". Например:
"sparc-sun", "i386-anything", "m68k-hp", "m68k-sony", "m68k-sun", "mips-dec"
int uname (struct utsname *info) (функция)
Uname функция вносит в структуру, указанную info информацию относительно операционной системы и главной машины. Неотрицательное значение указывает, что данные были успешно сохранены.

-1 указывает ошибку. Единственая возможная ошибка - EFAULT, которую мы обычно не упоминаем, поскольку она - всегда возможна.

27. Параметры Конфигурации Системы

Функции и макркоманды, перечисленные в этой главе дают информацию относительно параметров конфигурации операционной системы, например, ограничений пропускной способности, присутствие необязательных POSIX возможностей, и заданный по умолчанию путь для исполняемых файлов (см. Раздел 27.12 [Строковые Параметры]).

27.1 Общие Ограничения Пропускной способности

POSIX.1 и POSIX.2 стандарты определяют ряд параметров, которые описывают ограничения пропускной способности системы. Эти ограничения могут быть фиксированные константы для данной операционной системы, или они могут изменяться от машины к машине.

Каждый из следующих параметров ограничения имеет макркоманду, которая определена в " limits.h " только, если система имеет фиксированное, однородное ограничение для рассматриваемого параметра. Если система позволяет различным файловым системам или файлам иметь различные ограничения, то макркоманда неопределена; используйте sysconf, чтобы выяснить ограничение, которое применяется в специфическое время на специфической машине. См. Раздел 27.4 [Sysconf].

Каждый из этих параметров также имеет другую макрокоманду, с именем, начинающимся с " _POSIX ", которая дает самое низкое

значение, которое ограничению позволяется иметь на любой системе POSIX. См. Раздел 27.5 [Минимумы].

int ARG_MAX

Если этот макрос определен, то это неизменяющийся максимум объединенной длины argv и environ аргументов которые могут быть переданы функции exec.

int CHILD_MAX

Если этот макрос определен, то это неизменяющееся число максимума процессов, которые могут существовать с тем же самым реальным пользовательским ID одновременно.

int OPEN_MAX

Если этот макрос определен, то это неизменяющееся число максимума файлов, которые одиночный процесс может иметь открытым одновременно.

int STREAM_MAX

Если этот макрос определен, то это неизменяющееся число максимума потоков, которые одиночный процесс может иметь открытым одновременно. См. Раздел 7.3 [Открытие Потоков].

int TZNAME_MAX

Если этот макрос определен, то это неизменяющаяся максимальная длина имени часового пояса. См. Раздел 17.2.6 [Функции Часового пояса].

Эти макркоманды ограничений всегда определяются в " limits.h ".

int NGROUPS_MAX

Максимальное число ID дополнительных групп, которые один процесс может иметь.

int SSIZE_MAX

Самое большое значение, которое может содержаться в объекте типа ssize_t. Действительно, это - ограничение числа байтов, которые могут читаться или записываться в одиночной операции.

Эта макркоманда определена во всех системах POSIX, т. к. это ограничение никогда не переконфигурируется.

int RE_DUP_MAX (макрос)

Самое большое число повторений, которое Вам позволяется в конструкции " \{min,max\} " в регулярном выражении.

Эта макркоманда определена во всех POSIX.2 системах, т. к. POSIX.2 говорит, что она должна всегда определяться, даже если не имеется никакого специфического наложенного ограничения.

27.2 Полные Опции Системы

POSIX определяет некоторые системно-специфические опции, которые не все системы POSIX поддерживают. Так как эти опции обеспечиваются в ядре, а не в библиотеке, просто использование библиотек GNU C не гарантирует любой из этих возможностей; это зависит от системы, которую Вы используете.

Вы можете проверять доступность данной опции, используя макркоманды в этом разделе, вместе с функцией sysconf. Макркоманды определены только, если Вы включаете " unistd.h ".

Для следующих макркоманд, если макркоманда определена в " unistd.h ", то опция обеспечивается.

Иначе, опция может или не может обеспечена; используйте sysconf для выяснения. См. Раздел 27.4 [Sysconf].

int _POSIX_JOB_CONTROL (макрос)

Если этот символ определен, это указывает что система поддерживает управление заданиями. Иначе, реализация ведет себя, как будто все процессы внутри сеанса принадлежат одиночной группе процессов. См. Главу 24 [Управление заданиями].

int _POSIX_SAVED_IDS (макрос)

Если этот символ определен, это указывает, что система запоминает эффективный ID пользователя и группы процесса прежде, чем она выполняет исполняемый файл с установкой set-user-ID или set-group-ID битов, и что явное изменение эффективного пользователя или группы обратно к этим значениям разрешается. Если эта опция не определена, то, если непривилегированный процесс изменяет эффективного пользователя или группу на реального пользователя или группу процесса, то он не может изменять их обратно снова. См. раздел 25.8 [ВКЛЮЧЕНИЕ/ОТКЛЮЧЕНИЕ Setuid].

Для следующих макркоманд, если макркоманда определена в " unistd.h ", то значение указывает, обеспечивается ли опция. Значение -1 означает нет, а любое другое значение означает да. Если макркоманда не определена, то опция может или не может обеспечиваться; используйте sysconf для выяснения. См. Раздел 27.4 [Sysconf].

int _POSIX2_C_DEV (макрос)

Если этот символ определен, это указывает, что система имеет

POSIX.2 команду компилятора C, c89. Библиотека GNU C всегда определяет его как 1.

int _POSIX2_FORT_DEV (макрос)

Если этот символ определен, это указывает, что система имеет POSIX.2 команду компилятора ФОРТРАНа, fort77. Библиотека GNU C никогда не определяет его, т. к. мы не знаем то, что система имеет.

int _POSIX2_FORT_RUN

Если этот символ определен, это указывает, что система имеет POSIX.2 команду asa для интерпретирования управление каретки ФОРТРАНа. Библиотека GNU C никогда не определяет его, т. к. мы не знаем что имеет система.

int _POSIX2_LOCALEDEF (макрос)

Если этот символ определен, это указывает, что система имеет POSIX.2 команду localedef. Библиотека GNU C никогда не определяет его.

int _POSIX2_SW_DEV (макрос)

Если этот символ определен, это указывает, что система имеет POSIX.2 команды ar, make, и strip. Библиотека GNU C всегда определяет его как 1.

27.3 Которая Версия POSIX Обеспечивается

long int _POSIX_VERSION (макрос)

Эта константа представляет версию POSIX.1 стандарта который соответствует реализация. Для реализации, соответсвующей 1990 POSIX.1 стандарту, значение - целое число 199009L.

_POSIX_VERSION всегда определяется (в " unistd.h ") в любой системе POSIX.

Примечание Использования: Не пробуйте тестировать поддерживает ли система POSIX, включая " unistd.h " и тогда проверяя, определен ли _POSIX_VERSION. На не - posix системах, это возможно выдаст ошибку, т. к. там нет никакого " unistd.h ". Мы не знаем любого способа, которым Вы можете надежно проверять при трансляции, поддерживает ли ваша целевая система POSIX или существует ли " unistd.h ".

Компилятор GNU C предопределяет символ __ POSIX __, если целевая система - система POSIX. Если Вы не используете любой, другие компиляторы на системах POSIX, проверяя определенный (__ POSIX __)

надежно обнаружят такие системы.

long int _POSIX2_C_VERSION (макрос)
Эта константа представляет версию POSIX.2 стандарта, который поддерживает библиотека и ядро системы. Мы не знаем какое значение это будет для первой версии POSIX.2 стандарта, т. к. значение включает год и месяц, в котором стандарт официально принят.
Значение этого символа не говорит ничто относительно утилит, установленных на системе.
Примечание Использования: Вы можете использовать эту макркоманду, чтобы проверить что POSIX.1 библиотека систем поддерживает POSIX.2 также. Любая POSIX.1 система содержит " unistd.h ", так что включайте этот файл и тогда проверяйте defined (_POSIX2_C_VERSION).

27.4 Использование sysconf

Когда ваша система имеет ограничения системы с перестраиваемой конфигурацией, Вы можете использовать функцию sysconf для выяснения значений, которые применяются для любой специфической машины. Функция и связанные константы параметров объявлены в заглавном файле " unistd.h ".

27.4.1 Определение sysconf

long int sysconf (int parameter) (функция)
Эта функция используется, для запроса относительно параметров системы. Аргумент parameter должен быть один из " _SC_ " символов, перечисленных ниже.
Нормальное возвращаемое значение из sysconf - значение, которое Вы запросили. Значение -1 возвращается, если реализация не налагает ограничения, и в случае ошибки.
Следующие errno условия ошибки определены для этой функции:
EINVAL значение параметра недопустимо.

27.4.2 Константы для Sysconf Параметров

Имеются символические константы которые используют как аргументы в sysconf. Значения - все целочисленные константы (более специально, значения перечислимого типа).

_SC_ARG_MAX
Запрос относительно параметра, соответствующего ARG_MAX.
_SC_CHILD_MAX
Запрос относительно параметра, соответствующего CHILD_MAX.
_SC_OPEN_MAX
Запрос относительно параметра, соответствующего OPEN_MAX.
_SC_STREAM_MAX
Запрос относительно параметра, соответствующего STREAM_MAX.
_SC_TZNAME_MAX
Запрос относительно параметра, соответствующего TZNAME_MAX.
_SC_NGROUPS_MAX
Запрос относительно параметра, соответствующего NGROUPS_MAX.
_SC_JOB_CONTROL
Запрос относительно параметра, соответствующего _POSIX_JOB_CONTROL.
_SC_SAVED_IDS
Запрос относительно параметра, соответствующего _POSIX_SAVED_IDS.
_SC_VERSION
Запрос относительно параметра, соответствующего _POSIX_VERSION.
_SC_CLK_TCK
Запрос относительно параметра, соответствующего CLOCKS_PER_SEC; см. Раздел 17.1.1 [Основное Время CPU].
_SC_2_C_DEV
Запрос относительно того, имеет ли система POSIX.2 команду компилятора C, c89.
_SC_2_FORT_DEV
Запрос относительно того, имеет ли система POSIX.2 команду компилятора ФОРТРАНа, fort77.

_SC_2_FORT_RUN
Запрос относительно того, ли система имеет POSIX.2 команду asa для интерпретирования управления каретки ФОРТРАНа.
_SC_2_LOCALEDEF
Запросите относительно того, имеет ли система POSIX.2 команду localedef.
_SC_2_SW_DEV
Запросите относительно того, имеет ли система POSIX.2 команды

ar, make, и strip.
_SC_BC_BASE_MAX
Запрос относительно максимального значения obase в утилите bc.
_SC_BC_DIM_MAX
Запрос относительно максимального размера массива в утилите bc.
_SC_BC_SCALE_MAX
Запрос относительно максимального значения масштаба в утилите bc.
_SC_BC_STRING_MAX
Запрос относительно максимального размера строковой константы в утилите bc.
_SC_COLL_WEIGHTS_MAX
Запрос относительно максимального числа весов, которые могут обязательно использоваться в определении последовательности объединений для стандарта.
_SC_EXPR_NEST_MAX
Запрос относительно максимального числа выражений, вложенных внутри круглых скобок при использовании утилиты expr.
_SC_LINE_MAX
Запрос относительно максимального размера текстовой строки который POSIX.2 текстовые утилиты могут обрабатывать.
_SC_EQUIV_CLASS_MAX
Запрос относительно максимального числа весов, которые могут быть назначены на вход LC_COLLATE класса (" порядок " ключевое слово в определении стандарта). Библиотека GNU C не поддерживает определения стандарта.
_SC_VERSION
Запрос относительно номера версии POSIX.1 поддерживаемой ядром.
_SC_2_VERSION
Запрос относительно номера версии POSIX.2, поддерживаемой утилитами системы.
_SC_PAGESIZE
Запрос относительно виртуального размера страницы памяти машины. Getpagesize возвращает то же самое значение. См. <неопределенный> [XXX getpagesize].

27.4.3 Примеры sysconf

Мы рекомендуем, чтобы Вы сначала проверили макроопределение для параметра в котором Вы заинтересованы, и вызывали sysconf только, если макркоманда не определена. Например, вот как проверяют обеспечивается ли управление заданиями:

```
int
have_job_control (void)
{
#ifdef _POSIX_JOB_CONTROL
        return 1;
#else
        int value = sysconf (_SC_JOB_CONTROL);
        if (value < 0)
                fatal (strerror (errno));
        return value;
#endif
}
```

Here is how to get the value of a numeric limit:

```
int
get_child_max ()
{
```

```
#ifdef CHILD_MAX
        return CHILD_MAX;
#else
        int value = sysconf (_SC_CHILD_MAX);
        if (value < 0)
                fatal (strerror (errno));
        return value;
#endif
}
```


27.5 Минимальные Значения для Общих Ограничений Емкости

Имеются названия для POSIX минимальных верхних пределов для параметров ограничений системы.

Смысл этих значений состоит в том, что Вы можете безопасно помещать в эти ограничения без проверки, может ли специфическая система, которую Вы используете принять их.

_POSIX_ARG_MAX

Значение этой макркоманды - наименьшее ограничение, разрешенное POSIX для максимума объединенной длины argv и других аргументов, которые могут быть переданы запускаемой функции. Значение - 4096.

_POSIX_CHILD_MAX

Значение этой макркоманды наименьшее ограничение разрешенное POSIX для максимального числа одновременных процессов на реального пользователя. Значение - 6.

_POSIX_NGROUPS_MAX

Значение этой макркоманды - наименьшее значение ограничения, разрешенное POSIX для максимального числа дополнительных групп процесса. Значение - 0.

_POSIX_OPEN_MAX

Значение этой макркоманды - наименьшее значение ограничения, разрешенное POSIX для максимального числа файлов, которые одиночный процесс может иметь открытым одновременно. Значение - 16.

_POSIX_SSIZE_MAX

Значение этой макркоманды - наименьшее значение ограничения, разрешенное POSIX для максимального значения, которое может быть сохранено в объекте типа ssize_t. Значение - 32767.

_POSIX_STREAM_MAX

Значение этой макркоманды - наименьшее значение ограничения, разрешенное POSIX для максимального числа потоков, которые одиночный процесс может иметь открытым одновременно. Значение - 8.

_POSIX_TZNAME_MAX

Значение этой макркоманды - наименьшее значение ограничения, разрешенное POSIX для максимальной длины имени часового пояса. Значение - 3.

_POSIX2_RE_DUP_MAX

Значение этой макркоманды - наименьшее значение ограничения, разрешенное POSIX для чисел, используемых в " \ {min, max\} " конструкциях в

регулярном выражении. Значение - 255.

27.6 Ограничения Емкости Файловой системы

POSIX.1 стандарт определяет ряд параметров, которые описывают ограничения файловой системы. Система может иметь фиксированное, однородное ограничение для параметра, но это не обычный случай. На большинстве систем, для различных файловых систем (и, для некоторых параметров, даже для различных файлов) имеются различные максимальные ограничения. Например, если Вы используете NFS для устанавливки некоторых файловых систем из других машин.

Каждая из следующих макркоманд определена в " limits.h " только, если система имеет фиксированное, однородное ограничение для рассматриваемого параметра. Если система позволяет различным файловым системам или файлам иметь различные ограничения, то макркоманда неопределена; используйте pathconf или fpathconf, чтобы выяснить ограничение, которое применяется к

специфическому файлу. См. Раздел 27.9 [Pathconf].

Каждый параметр также имеет другую макркоманду, с именем, начинающимся с " _POSIX ", которая дает самое низкое значение, которое ограничению позволяется иметь на любой POSIX системе. См. Раздел 27.8 [Минимумы Файла].

int LINK_MAX (макрос)

Однородное ограничение системы для числа имен для данного файла. См. Раздел 9.3 [Жесткие связи].

int MAX_CANON (макрос)

Однородное ограничение системы для количества текста в строке ввода, когда допускается входное редактирование. См. Раздел 12.3 [Канонический или Нет].

int MAX_INPUT (макрос)

Однородное ограничение системы для общего числа символов, печатаемых вперед как ввод. См. Раздел 12.2 [Очереди ввода - вывода].

int NAME_MAX (макрос)

Однородное ограничение системы для длины компонента имени файла.

int PATH_MAX (макрос)

Однородное ограничение системы для длины всего имени файла (то есть аргумент, данный системным вызовам типа open).

int PIPE_BUF (макрос)

Однородное ограничение системы для числа байтов, которые можно

написать в трубопровод быстро. См. Главу 10 [Трубопроводы и FIFOs].

Это - альтернативные имена макркоманд.

int MAXNAMLEN (макрос)

Это - имя BSD для NAME_MAX. Она определена в " dirent.h ".

int FILENAME_MAX (макрос)

Значение этой макркоманды - целочисленное постоянное выражение, которое представляет максимальную длину строки имени файла. Она определена в " stdio.h ".

В отличие от PATH_MAX, эта макркоманда определена, даже если не имеется никакого фактического наложенного ограничения. В таком случае, значение - обычно очень большое количество. Так всегда в системе GNU.

Примечание Использования: Не используйте FILENAME_MAX как размер массива, чтобы сохранить имя файла! Вы не можете возможно делать массив таким большим! Используйте динамическое резервирование (см. Главу 3 [Распределение памяти]).

27.7 Необязательные Возможности в Поддержке Файлов

POSIX определяет некоторые системно-специфические опции в системных вызовах для файлов. Некоторые системы поддерживают эти опции, а другие - нет. Так как эти опции обеспечиваются в ядре, а не в библиотеке, просто использование библиотеки GNU C не гарантирует, что любая из этих возможностей обеспечивается; это зависит от системы, которую Вы используете. Они могут также изменяться между файловыми системами на одиночной машине.

Этот раздел описывает макркоманды, которые Вы можете проверять, чтобы определить, обеспечивается ли специфическая опция на вашей машине. Если данная макркоманда определена в " unistd.h ", то значение, говорит обеспечивается ли соответствующая возможность. (Значение -1 указывает нет; любое другое значение указывает да.) если макркоманда неопределена, это означает специфические файлы могут и не поддерживать эту возможность.

Так как все машины, которые поддерживают библиотеку GNU C также поддерживают NFS, можно делать общее утверждение относительно того, все ли файловые системы поддерживают возможности _POSIX_NO_TRUNC и _POSIX_CHOWN_RESTRICTED. Так что эти имена никогда не определены как макркоманды в библиотеке GNU C.

int _POSIX_CHOWN_RESTRICTED (макрос)

Если эта опция есть, chown функция ограничена так, что единственые

изменения, разрешенные непривилегированным процессам: изменить владельца группы файла либо на эффективный ID группы процесса, либо на один из ID дополнительных групп. См. Раздел 9.8.4 [Владелец Файла].

int _POSIX_NO_TRUNC (макрос)

Если эта опция обеспечивается, когда компоненты имени файла длиннее чем NAME_MAX, генерируется ENAMETOOLONG ошибка. Иначе, компоненты имени файла, которые являются слишком длинными будут тихо усечены.

unsigned char _POSIX_VDISABLE (макрос)

Эта опция значима только для файлов, которые являются устройствами

терминала. Если она допускается, то обработчик для специальных управляющих символов, может быть заблокирован индивидуально. См. Раздел 12.4.9 [Специальные Символы].

Если одна из этих макркоманд является неопределенной, это означает что эта опция может быть определена в действительности для некоторых файлов и не определена для других. Чтобы запрашивать относительно специфического файла, вызовите pathconf или fpathconf. См. Раздел 27.9 [Pathconf].

27.8 Минимальные Значения для Ограничений Файловой системы

Имеются имена для POSIX минимальных верхних пределов для некоторых из вышеупомянутых параметров.

Смысл этих значений состоит в том, что Вы можете безопасно помещать в эти ограничения без проверки, может ли специфическая система, которую Вы используете принять их.

_POSIX_LINK_MAX

Наименьшее значение предела, разрешенное POSIX для максимального значения числа связей файла. Значение этой константы - 8; таким образом, Вы можете всегда создавать до восьми имен для файла без столкновений с ограничением системы.

_POSIX_MAX_CANON

Наименьшее значение предела, разрешенное POSIX для максимального числа байтов в канонической входной строке из устройства терминала. Значение этой константы - 255.

_POSIX_MAX_INPUT

Наименьшее значение предела, разрешенное POSIX для максимального числа байтов во входной очереди устройства терминала (или буфере клавиатуры). См. Раздел 12.4.4 [Режимы Ввода]. Значение этой константы - 255.

_POSIX_NAME_MAX

Наименьшее значение предела, разрешенное POSIX для максимального числа байтов в компоненте имени файла. Значение этой константы - 14.

_POSIX_PATH_MAX

Наименьшее значение предела, разрешенное POSIX для максимального числа байтов в имени файла. Значение этой константы - 255.

_POSIX_PIPE_BUF

Наименьшее значение предела, разрешенное POSIX для максимального числа байтов, которые можно написать в трубопровод быстро. Значение этой константы - 512.

27.9 Использование pathconf

Когда ваша машина позволяет различным файлам иметь различные значения для параметра файловой системы, Вы можете использовать функции в этом разделе, чтобы выяснить значение, которое применяется к любому специфическому файлу.

Эти функции и связанные константы для аргумента parameter объявлены в заглавном файле " unistd.h ".

long int pathconf (const char *filename, int parameter) (функция)

Эта функция используется, чтобы запросить относительно ограничений, которые применяются к файлу, именованному filename.

Аргумент parameter должен быть одной из " _PC_ " констант, перечисленных ниже.

Нормальное возвращаемое значение из pathconf - значение, которое Вы запросили. Значение -1 будет возвращено, если реализация не налагает ограничений, и в случае ошибки.

В вышеупомянутом случае, errno не установлена, в то время как в последнем случае, errno установлена, чтобы указать причину проблемы. Так что единственый способ использовать эту функцию robustly состоит в том, чтобы сохранить 0 в errno перед ее вызовом.

Кроме обычных синтаксических ошибок имени файла (см. Раздел 6.2.3 [Ошибки Имени файла]), следующие условия ошибки определены для этой функции:

EINVAL значение параметра недопустимо, или реализация не поддерживает параметр для специфического файла.

long int fpathconf (int filedes, int parameter) (функция)

Точно такая же как pathconf за исключением того, что вместо имени файла используется описатель открытого файла, чтобы определить файл, для которого запрошена информация.

Следующие errno условия ошибки определены для этой функции:

EBADF filedes аргумент - не допустимый описатель файла.

EINVAL значение параметра недопустимо, или реализация не поддерживает параметр для специфического файла.

Имеются символические константы, которые Вы можете использовать как аргумент parameter в pathconf и fpathconf. Значения - целочисленные константы.

_PC_LINK_MAX
Запрос относительно значения LINK_MAX.

_PC_MAX_CANON
Запрос относительно значения MAX_CANON.

_PC_MAX_INPUT
Запрос относительно значения MAX_INPUT.

_PC_NAME_MAX
Запрос относительно значения NAME_MAX.

_PC_PATH_MAX
Запрос относительно значения PATH _MAX.

_PC_PIPE_BUF
Запрос относительно значения PIPE_BUF.

_PC_CHOWN_RESTRICTED
Запрос относительно значения _POSIX_CHOWN_RESTRICTED.

_PC_NO_TRUNC
Запрос относительно значения _POSIX_NO_TRUNC.

_PC_VDISABLE
Запрос относительно значения _POSIX_VDISABLE.

27.10 Ограничения для Утилит

POSIX.2 стандарт определяет некоторые ограничения системы, к которым Вы можете обращаться через sysconf, которые относятся к поведению некоторых утилит а не к поведению библиотеки или операционной системы.

Библиотека GNU C определяет макркоманды для этих ограничений, и sysconf возвращает значения для них, если Вы спрашиваете; но эти значения не передают никакую значимую информацию. Они - просто самые маленькие

значения которые POSIX. 2 разрешает.

int BC_BASE_MAX (макрос)
Самое большое значение obase, который утилита bc, как гарантируют, будет поддерживать.

int BC_SCALE_MAX (макрос)
Самое большое значение масштаба, который утилита bc, как гарантируют, будет поддерживать.

int BC_DIM_MAX (макрос)
Самое большое число элементов в одном массиве, который утилита bc, как гарантируют, будет поддерживать.

int BC_STRING_MAX (макрос)
Самое большое число символов в одной строковой константе, которую утилита bc, как гарантируют, будет поддерживать.

int BC_DIM_MAX (макрос)
Самое большое число элементов в одном массиве, который утилита bc, как гарантируют, будет поддерживать.

int COLL_WEIGHTS_MAX (макрос)
Самое большое число весов, которые могут обязательно использоваться в определении последовательности объединений для стандарта.

int EXPR_NEST_MAX (макрос)
Максимальное число выражений, которые могут быть вложены круглыми скобками для утилиты expr.

int LINE_MAX (макрос)
Самая большая текстовая строка, которую POSIX.2 текстовые утилиты могут поддерживать. (Если, Вы используете GNU версии этих утилит, то нет никакого фактического ограничения за исключением тех, что наложены доступной виртуальной памятью, но не никакого способа, которым библиотека может сообщать Вам это.)

27.11 Минимальные Значения для Пределов Утилит

_POSIX2_BC_BASE_MAX
Наименьшее значение предела, разрешенное POSIX.2 для максимального значения obase в утилите bc. Значение - 99.

_POSIX2_BC_DIM_MAX
Наименьшее значение предела, разрешенное POSIX.2 для максимального

размера массива в утилите bc. Значение - 2048.

_POSIX2_BC_SCALE_MAX
Наименьшее значение предела, разрешенное POSIX.2 для максимального значения масштаба в утилите bc. Значение - 99.
_POSIX2_BC_STRING_MAX
Наименьшее значение предела, разрешенное POSIX. 2 для максимального размера строковой константы в утилите bc. Значение - 1000.
_POSIX2_COLL_WEIGHTS_MAX
Наименьшее значение предела, разрешенное POSIX. 2 для максимального числа весов, которые могут использоваться в определении последовательности объединений для стандарта. Значение - 2.
_POSIX2_EXPR_NEST_MAX
Наименьшее значение предела, разрешенное POSIX. 2 для максимального числа выражений, вложенных внутри круглых скобок при использовании утилиты expr. Значение - 32.
_POSIX2_LINE_MAX
Наименьшее значение предела, разрешенное POSIX. 2 для максимального размера текстовой строки, которую текстовые утилиты могут обрабатывать. Значение - 2048.

27.12 Строковые Параметры

POSIX. 2 определяет способ получить строковые параметры из операционной системы - функцией confstr:
size_t confstr (int parameter, char *buf, size_t len) (функция)
Эта функция читает значение строкового параметра системы, сохраняя строку в len байты пространства памяти, начинающегося в buf. Аргумент parameter должен быть один из " _CS_ " символов, перечисленных ниже.
Нормальное возвращаемое значение из confstr - длина строкового значения, о котором Вы просили. Если Вы обеспечиваете пустой указатель для buf, то confstr, не пробует сохранять строку; она только возвращает длину. Значение 0 указывает ошибку.
Если строка, о которой Вы просили, слишком длинная для буфера (то есть длиньше чем len - 1), то confstr сохраняет только то что помещается (оставляя участок памяти для пустого символа завершения). Вы можете понять, что это случилось, потому что confstr возвращает значение больше чем или равный len.
Следующие errno условия ошибки определены для этой функции:
EINVAL значение параметра недопустимо.

В настоящее время имеется только один параметр, который Вы можете читать с confstr:
_CS_PATH значение этого параметра - рекомендуемый заданный по умолчанию путь для поиска исполняемых файлов. Это - путь, который пользователь имеет по умолчанию только после регистрации.
Способ использовать confstr без любого произвольного ограничения строкового размера состоит в том, чтобы вызвать ее дважды: сначала вызвать ее, чтобы получить длину, зарезервировать буфер соответственно, и тогда вызывать confstr снова, чтобы заполнить буфер, примерно так:

```
char *
get_default_path (void)
{
        size_t len = confstr (_CS_PATH, NULL, 0);
        char *buffer = (char *) xmalloc (len);
        if (confstr (_CS_PATH, buf, len + 1) == 0) {
                free (buffer);
                return NULL;
        }
        return buffer;
}
```

Приложение A: Средства Языка C в Библиотеке

О некоторых средствах, выполненных библиотекой C действительно нужно думать как о части Языка C непосредственно. Эти средства должны быть зарегистрированы в Руководстве Языка C, а не в библиотечном руководстве;

но так как мы пока не имеем руководства языка, и документация для этих возможностей пишется, мы издаем его здесь.

A.1 Явная Проверка Внутренней Непротиворечивости

Когда вы - пишите программу, часто неплохо поместить проверки в стратегических местах для "невозможных" ошибок или нарушений базисных предположений. Эти проверки полезны при отладке проблем из-за непониманий между различными частями программы.

Макркоманда assert, определенная в заглавном файле " assert.h ",

обеспечивает удобный способ прервать программу при печати сообщения относительно того, где в программе была обнаружена ошибка.

Если только Вы думаете, что ваша программа отлажена, Вы можете отключать проверки ошибки, выполняемые макркомандой assert, перетранслируя с определенной макркомандой NDEBUG. Это означает что Вы, фактически не должны изменить исходный текст программы, чтобы отключить эти проверки.

Но отключение этих проверок непротиворечивости нежелательно, если они не делают программу значительно медленнее. Большое количество проверок ошибок - хорошо независимо от того, кто выполняет программу.

Знающий пользователь хотел бы иметь аварийный отказ программы, явно, чем возврат ерунды без указания, что что-то неправильно.

void assert (int expression) (макрос)

Проверяет убеждение программиста, что выражение должно быть отлично от нуля в этом месте программы.

Если NDEBUG не определен, assert проверяет значение выражения. Если оно является ложным (нулем), assert прерывает программу (см. Раздел 22.3.4 [Прерывание выполнения Программы]) после печати сообщения вида:

`file':linenum: Assertion `expression' failed.

в стандартный поток ошибки stderr (см. Раздел 7.2 [Стандартные Потоки]). Filename и номер строки берутся из макркоманд препроцессора C __FILE__ и __LINE__.

Если макркоманда препроцессора NDEBUG определена в отметке, где "assert.h " включен, макркоманда assert не будет делать абсолютно ничего.

Предупреждение: Даже выражение аргумента не будет оценено, если NDEBUG - определен. Так что никогда не используйте assert с аргументами, которые включают побочные эффекты. Например, assert (++ i> 0); является плохой идеей, потому что i не будет увеличена, если NDEBUG определен.

Примечание Использования: средство assert разработано для обнаружения внутренней несогласованности; оно не подходит для сообщения недопустимого ввода или неподходящего использования пользователем программы.

Информация в диагностических сообщениях, напечатанных макркомандой assert предназначена, чтобы помочь Вам, программисту, прослеживать причину ошибки, но не полезна для сообщения пользователю вашей программы, почему его или ее ввод был недопустим или почему команда не могла быть проведена. Так что Вы не можете использовать assert, чтобы печатать сообщения об ошибках для этих целей.

Более того, ваша программа не должна прерываться когда дан

недопустимый ввод, а assert сделала бы это выйдя с состоянием отличным от нуля (см. Раздел 22.3.2 [Состояние Выхода]) после печати сообщений об ошибках.

A.2 Variadic Функции

ANSI C определяет синтаксис для объявления функции, берущей переменное число или тип аргументов. (Такие функции упоминаются как varargs функции или variadic функции.) Однако, язык непосредственно не обеспечивает никакого механизма для таких функций, чтобы обратиться к их не-требуемым аргументам; взамен, Вы используете переменные макркоманды аргументов, определенные в " stdarg.h ".

Этот раздел описывает, как объявить variadic функции, как написать их, и как вызвать их правильно.

Примечание Совместимости: Намного более старые диалекты C обеспечивают подобный, но несовместимый, механизм для определения функций с переменным числом аргументов, используя " varargs.h ".

A.2.1 Зачем Используются Variadic Функции

Обычные функции C берут фиксированное число аргументов. Когда Вы определяете функцию, Вы определяете тип данных для каждого аргумента. Каждое обращение к функции должно обеспечить ожидаемое число аргументов, с типами, которые могут быть преобразованы в заданные. Таким образом, если функция " foo " объявлена: int foo (int, char*); то Вы должны вызвать ее с

двумя аргументами: числом и строковым указателем.

Но некоторые функции выполняют операции, которые могут принимать неограниченное число аргументов.

В некоторых случаях функция может обрабатывать любое число значений, действуя на все из них, как на блок. Например, рассмотрите функцию, которая резервирует одномерный массив через malloc, чтобы содержать заданный набор значений. Эта операция имеет смысл для любого числа значений, если только длина массива соответствует к заданному числу. Без средств для переменных аргументов, Вы были бы должны определять отдельную функцию для каждого возможного размера массива.

Библиотечная функция printf (см. Раздел 7.9 [Форматируемый Вывод]) - пример другого класса функции, где переменные аргументы полезны. Эта функция печатает аргументы (которые могут изменяться в типе также как в числе) при контроле над строкой шаблона формата.

Это причины определить variadic функцию, которая может обрабатывать так много аргументов, как вызывающий оператор выбирает.

Некоторые функции типа open берут фиксированный набор аргументов, но иногда игнорируют несколько последних. Строгая приверженность ANSI C требует, чтобы эти функции были определены как variadic; практически, однако, компилятор GNU C и большинство других компиляторов C допускает Вам определять такую функцию, чтобы брать фиксированный набор аргументов больше чем она может когда-либо использовать и тогда только объявлять функцию как variadic (или не объявлять аргументы вообще!).

A.2.2 Как Variadic Функции определяются и Используются

Определение и использование variadic функций включает три шага:

* Определить функцию как variadic, используя ". . ." в списке параметров, и использовать специальные макркоманды, чтобы обратиться к переменным аргументам. См. Раздел A. 2.2.2 [Получение Аргументов].
* Объявить функцию как variadic, используя прототип с ". . .", во всех файлах, которые ее вызывают. См. Раздел A. 2.2.1 [Variadic Прототипы].
* Вызвать функцию записью с фиксированными аргументами, сопровождаемыми дополнительными переменными аргументами. См. Раздел A. 2.2.4 [Вызов Variadic].

A.2.2.1 Синтаксис для Переменных Аргументов

Функция, которая принимает переменное число аргументов, должна быть объявлена с прототипом, который говорит это. Вы записываете фиксированные аргументы как обычные, и тогда указываете ". . ." чтобы указать возможность дополнительных аргументов. Синтаксис ANSI C требует по крайней мере одного фиксированного аргумента перед ". . .". Например,

```
int
func (const char *a, int b, . . .)
{
        . . .
}
```

выделяет определение функции func, которая возвращает int и берет два требуемых аргумента, const char * и int. Они сопровождаются любым числом анонимных аргументов.

A.2.2.2 Получение Значения Аргумента

Обычные фиксированные аргументы имеют индивидуальные имена, и Вы можете использовать эти имена, чтобы обратиться к их значениям. Но необязательные аргументы не имеют никаких имен коме ". . . ". Как Вы может обращаться к ним ?

Единственый способ обращаться к ним - последовательно, в порядке, в котором они написаны, и Вы должны использовать специальные макркоманды из " stdarg.h " в следующем процессе:

1. Вы инициализируете переменную указателя аргумента типа va_list, используя va_start. Указатель аргумента, после инициализации, указывает на первый необязательный аргумент.
2. Вы обращаетесь к необязательным аргументам последовательными обращениями к va_arg. Первое обращение к va_arg дает Вам, первый необязательный аргумент, следующее обращение дает Вам второй, и так

далее.
3. Вы указываете, что Вы закончили с переменной указателя аргумента, вызывая va_end.
См. Раздел A. 2.2.5 [Макросы Аргумента], для полных определений va_start, va_arg и va_end.
Шаги 1 и 3 должны выполниться в функции, которая принимает необязательные аргументы. Однако, Вы можете передавать va_list переменную как аргумент другой функции и выполнять весь или часть шага 2 там.
Вы можете выполнять всю последовательность из трех шагов несколько раз внутри одного вызова функции. Если Вы хотите игнорировать необязательные аргументы, Вы можете не делать этого.
Вы можете иметь больше чем одну переменную указателя аргумента, если Вы находите приятным. Вы можете инициализировать каждую переменную с va_start, когда Вы пожелаете, и т. д.

A.2.2.3 Сколько Аргументов Обеспечивается

Не имеется никакого общего способа для функции, чтобы определить число и тип необязательных аргументов, с которыми она вызывалась.
Один вид соглашения о вызовах должен передать число необязательных аргументов как один из фиксированных аргументов. Это соглашение работает, принимая что все необязательные аргументы имеют тот же самый тип.
Подобный вариант должен иметь один из требуемых аргументов как битовую маску, с битом для каждой возможной цели, для которой

необязательный аргумент мог бы быть обеспечен. Вы проверили бы биты в предопределенной последовательности; если бит установлен, выборка значения следующего аргумента, иначе используется значение по умолчанию.
Требуемый аргумент может использоваться как шаблон, чтобы определить и число и типы необязательных аргументов. Аргумент строки формата printf - один пример этого (см. Раздел 7.9.7 [Функции Форматированного Вывода]).
Другая возможность состоит в том, чтобы передать значение " end marker " как последний необязательный аргумент. Например, для функции, которая манипулирует, произвольным числом аргументов указателей, пустой указатель мог бы указывать конец списка параметров. (Она принимает, что пустой указатель иначе не значим для функции.) execl функция работает только этим способом; см. Раздел 23.5 [Выполнение Файла].

A.2.2.4 Вызов Variadic Функции

Вы не должны писать что-нибудь специальное, когда Вы вызываете variadic функцию. Только запишите аргументы (требуемые аргументы, сопровождаемые необязательными) внутри круглых скобок, отделенные запятыми, как обычно.
В принципе, функции, которые определены, чтобы быть variadic, должны также быть объявлены, чтобы быть variadic использованием прототипа функции всякий раз, когда Вы вызываете их. (См. Раздел A. 2.2.1 [Variadic Прототипы], для того какэто сделать.) Это - потому что некоторые компиляторы C используют различное соглашение о вызовах, чтобы передать тот же самый набор значений аргумента к функции в зависимости от того, берет ли та функция переменные аргументы или фиксированне аргументы.
Но имеются несколько функций, которые чрезвычайно удобно не объявлять как variadic, например open и printf.
Так как прототип не определяет типы для необязательных аргументов, в обращении к variadic функции, заданные по умолчанию поддержки аргумента выполняются на необязательных значениях аргумента. Это означает объекты типа char или short int (или signed, или нет) преобразуются или в int или в unsigned int, соответственно; а объекты float типа - в double тип. Так, если вызывающий оператор передает char как необязательный аргумент, получится int, и функция должна получить его с va_arg (ap, int).
Преобразование требуемых аргументов управляется прототипом функции обычным способом: выражение аргумента преобразовано в объявленный тип аргумента.

A.2.2.5 Макросы Доступа к Аргументу

Имеются описания макркоманд, используемых, чтобы отыскать переменные аргументы. Эти макркоманды определены в заглавном файле " stdarg.h ".
va_list (тип данных)
Тип va_list используется для переменной указателя аргумента.
void va_start (va_list ap, last_required) (макрос)
Эта макркоманда инициализирует переменную указателя аргумента ap,

чтобы указать на первый из необязательных аргументов текущей функции; last_required должен быть последний требуемый аргумент функции.

См. Раздел A.2.3.1 [Старый Varargs], для альтернативного определения va_start в заглавном файле " varargs.h ".

type va_arg (va_list ap, type) (макрос)

Va_arg макркоманда возвращает значение следующего необязательного аргумента, и изменяет значение ap, чтобы указsdать на последующий аргумент. Таким образом, последовательные использования va_arg возвращают последовательные необязательные аргументы.

Тип значения, возвращенного va_arg - type как определено в обращении. type должен быть само-поддерживающийся тип (не char или short int или float) который соответствует типу фактического аргумента.

void va_end (va_list ap) (макрос)

Он заканчивает использование ap. После обращения va_end, дальнейшие va_arg обращения с тем же самым ap не могут работать. Вы должны вызвать va_end перед возвращением из функции, в которой va_start вызывался с тем же самым аргументом ap.

В библиотеке GNU C, va_end не делает ничего, и Вы не нуждаетесь в его использовании, кроме причин переносимости.

A.2.3 Пример Variadic Функции

Вот полная типовая функция, которая принимает переменное число аргументов. Первый аргумент функции - число остающихся аргументов, которые складываются и возвращается результат.

```
#include
#include
int
add_em_up (int count,...)
{
        va_list ap;

                - 586 -

        int i, sum;
        va_start (ap, count);
        sum = 0;
        for (i = 0; i < count; i++)
                sum += va_arg (ap, int);
        va_end (ap);
        return sum;
}
int
main (void)
{
        printf ("%d\n", add_em_up (3, 5, 5, 6));
        printf ("%d\n", add_em_up (10, 1, 2, 3, 4, 5,
        6, 7, 8, 9, 10));
        return 0;
}
```

A.2.3.1 Variadic Функции Старого стиля

Раньше ANSI C программисты использовали немного отличное средство для написания variadic функции. Компилятор GNU C все еще поддерживает его; в настоящее время оно более переносимое чем средство ANSI C, так как поддержка для ANSI C все еще не универсальна. Заглавный файл, который определяет традиционное variadic средство называется " varargs.h ".

Использование " varargs.h " почти такое же как использование " stdarg.h ". Не имеется никакого различия в том, как Вы вызываете variadic функцию; См.Раздел A. 2.2.4 [Вызов Variadics]. Единственое различие находится в том, как Вы определяете их. Прежде всего Вы должны использовать синтаксис не-прототипа старого стиля, примерно так:

```
tree
build (va_alist)
                va_dcl
{
```

Во-вторых, Вы должны дать va_start только один аргумент, примерно так:

```
        va_list p;
        va_start (p);
```

Вот специальные макркоманды, используемые для определения variadic функций старого стиля:

va_alist (макрос)

Эта макркоманда замещает список имени аргумента, требуемый в variadic

функции.

va_dcl (макрос)

Эта макркоманда объявляет неявный аргумент или аргументы для variadic функции.

void va_start (va_list ap) (макрос)

Эта макркоманда, как определено в " varargs.h ", инициализирует ap переменную указателя аргумента, чтобы указывать на первый аргумент текущей функции.

Другие макркоманды аргумента, va_arg и va_end, те же самые в " varargs.h " как и в " stdarg.h "; см. Раздел A. 2.2.5 [Макросы Аргумента].

Это не работает, чтобы включить, и " varargs.h " и " stdarg.h " в той же самой трансляции; они определяют va_start противоречии способами.

A.3 Константа - Нулевой Указатель

Пустой указатель, как гарантируют, не укажет на любой реальный объект. Вы можете назначать ее для любой переменной указателя, так как она имеет тип void *. Обычный способ записи пустого указателя - NULL.

void * NULL (макрос)

Это - пустой указатель.

Вы можете также использовать 0 или (void *) 0 как нулевые указатели, но использование NULL более чистое, потому что это делает цель константы более очевидной.

Если Вы используете NULL как аргумент функции, то для полной переносимости, Вы должен удостовериться, что функция имеет объявление прототипа. Иначе, если целевая машина имеет два различных представления указателя, компилятор не будет знать которое представление использовать для этого аргумента. Вы можете избежать этой проблемы, явно приводя константу к соответствующему типу указателя, но мы рекомендуем добавить прототип для функции, которую Вы вызываете.

A.4 Важные Типы Данных

Результат вычитания двух указателей на C - всегда целое число,но точный тип данных изменяется от компилятора к компилятору. Аналогично, тип данных результата sizeof также изменяется между компиляторами. ANSI определяет стандартные побочные результаты исследования для этих двух типов, так что Вы можете обратиться к ним в переносимом режиме. Они определены в заглавном файле " stddef.h ".

ptrdiff_t (тип данных)

Это - целочисленный со знаком тип результата вычитания двух указателей. Например, с объявлением char *p1*, p2;, выражение p2 - p1 имеет тип ptrdiff_t.

Это будет возможно один из стандартных целочисленных со знаком типов (short int, int или long int), но мог бы быть нестандартный тип, который существует только для этой цели.

size_t (тип данных)

Это - целочисленный беззнаковый тип, используемый, чтобы представить размеры объектов. Результат оператора sizeof имеет этот тип, и функции типа malloc (см. Раздел 3.3 [Беспрепятственное Резервирование]) и memcpy (см. Раздел 5.4 [Копирование и конкатенация]) принимают аргументы этого типа, чтобы определить объектные размеры.

Примечание Использования: size_t - привилегированный способ объявить любые аргументы или переменные, которые содержат размер объекта.

В системе GNU size_t эквивалентен или unsigned int или unsigned long int. Эти типы имеют идентичные свойства в системе GNU, и для большинства целей, Вы можете использовать их неизменяя. Однако, они различны как типы данных, что дает различие в некоторых контекстах.

Примечание Совместимости: Реализации C перед появлением ANSI C вообще использовали unsigned int для представления объектных размеров и int для результатов вычитания указателей. Они не обязательно определяли или size_t или ptrdiff_t. Системы UNIX определяли size_t, в " sys/types.h ", но определение было обычно знаковым типом.

A.5 Размеры Типов Данных

Большинство времени, если Вы выбираете соответствующий тип данных для каждого объекта в вашей программе, Вы не должны иметь отношение к тому, как они представляются или сколько битов они используют. Когда Вы нуждаетесь в такой информации, Язык C непосредственно не обеспечивает способ получить это. Заглавные файлы " limits.h " и " float .h " содержит макркоманды, которые дают Вам эту информацию.

A.5.1 Вычисление Ширины Целого

Наиболее общая причина, по которой программа должна знать, сколько битов находятся в целочисленном типе - для использования массива long int как битового вектора. Вы можете обращаться к биту с индексом n как vector[n / LONGBITS] & (1 << (n % LONGBITS)), если Вы определяете LONGBITS как число битов в long int.

Не имеется никакого оператора в Языке C, который может давать Вам число битов в целочисленном типе данных. Но Вы можете вычислять его из макркоманды CHAR_BIT, определенный в заглавном файле " limits.h ".

CHAR_BIT Это - число битов в char; восемь, на большинстве систем. Значение имеет тип int.

Вы можете вычислять число битов в любом типе данных примерно так:

```
sizeof (type) * CHAR_BIT
```

A.5.2 Промежуток Целого Типа

Предположите, что Вы должны сохранить целочисленное значение, которое может быть в промежутке от нуля до одного миллиона. Который тип является самым маленьким типом, который Вы можете использовать? Не имеется никакого общего правила; это зависит от компилятора C и целевой машины. Вы можете использовать " MIN " и " MAX " макркоманды в " limits.h " чтобы определить, который тип будет работать.

Каждый целочисленный со знаком тип имеет пару макркоманд, которые дают самые маленькие и самые большие значения, которые он может содержать. Каждый целочисленный беззнаковый тип имеет одну такую макркоманду, для максимального значения; минимальное значение, конечно - нуль.

Значения этих макркоманд - все целочисленные постоянные выражения. " MAX " и " MIN " макркоманды для char и short int имеют значения типа int. " MAX " и " MIN " макркоманды для других типов имеет значения того же самого типа, описанного макркомандой таким образом, ULONG_MAX имеет unsigned long int тип.

SCHAR_MIN

Это - минимальное значение, которое может представляться signed char.

SCHAR_MAX

UCHAR_MAX

Это - максимальные значения, которые могут представлять signed char и char без знака, соответственно.

CHAR_MIN

Это - минимальное значение, которое может представлять char. Оно равно SCHAR_MIN, если char - со знаком, или нуль иначе.

CHAR_MAX

Это - максимальное значение, которое может представлять char. Оно равно SCHAR_MAX, если char - со знаком, или UCHAR_MAX иначе.

SHRT_MIN

Это - минимальное значение, которое может представлять signed short int. На большинстве машин, на которых библиотека GNU C выполняется, короткие целые - 16 битовые.

SHRT_MAX

USHRT_MAX

Это - максимум, который может представляться signed short int и unsigned short int, соответственно.

INT_MIN

Это - минимальное значение, которое может представляться signed int. На большинстве машин, на которых система GNU C выполняется, int - 32 битовый.

INT_MAX

UINT_MAX

Это - максимум, который может представляться, соответственно, signed int типом и unsigned int типом.

LONG_MIN

Это - минимальное значение, которое может представляться signed long int. На большинстве машин, на которых система GNU C выполняется, длинные целые числа - 32 битовые, того же самого размера как int.

LONG_MAX

ULONG_MAX

Это - максимум, который может представляться signed long int и unsigned long int, соответственно.

LONG_LONG_MIN

Это - минимальное значение, которое может представляться signed long long int. На большинстве машин, на которых система GNU C выполняется, длинные длинные целые числа - 64 битовые.

LONG_LONG_MAX

ULONG_LONG_MAX

Это - максимум, который может представляться signed long long int и unsigned long long int, соответственно.

- 591 -

WCHAR_MAX

Это - максимальное значение, которое может представляться wchar_t. См. Раздел 18.4 [Введение в Расширенные Символы].

Заглавный файл " limits.h " также определяет некоторые дополнительные константы, которые операционная система и файловая система ограничивает. Эти константы описаны в Главе 27 [Конфигурация Системы].

A.5.3 Плавающий Тип

Специфическое представление чисел с плавающей запятой изменяется от машины к машине. Потому что числа с плавающей запятой представляются внутренне как приблизительные количества, алгоритмы для управления данными с плавающей запятой часто должны принять во внимание точные характеристики конкретной машины.

Заглавный файл " float .h " описывает формат, используемый вашей машиной.

A.5.3.1 Концепции Представления Чисел С Плавающей Запятой

Этот раздел представляет терминологию для описания представлений с плавающей запятой.

Вы возможно уже знакомы с большинством этих понятий в терминах научного или экспоненциального представления чисел для чисел с плавающей запятой. Например, число 123456.0 могло бы быть выражено в экспоненциальном представлении как 1.23456e + 05, укорченная запись, указывающая, что мантисса 1.23456 умножена на основание 10 в степени 5.

Более формально, внутреннее представление числа са плавающей запятой может характеризоваться в терминах следующих параметров:

* Знак является или -1 или 1.
* Основа или основание системы счисления для возведения в степень, целое число больше чем 1. Это - константа для специфического представления.
* Экспонента, в которую основание возводится. Верхние и нижние пределы значения экспоненты - константы для специфического представления.
* Мантисса - целочисленное беззнаковое, которое является частью каждого числа с плавающей запятой.
* Точность мантиссы. Если основа представления - b, то точность - число из b цифр в мантиссе. Это - константа для специфического представления.

Снова, библиотека GNU не обеспечивает никаких средств для работы с такими аспектами представления низкого уровня.

- 592 -

Мантисса числа с плавающей запятой фактически представляет неявную дробь, чей знаменатель является основой в степени точности. Так как самая большая представимая мантисса на один меньше чем этот знаменатель, значение дроби - всегда строго меньше чем 1. Математическое значение числа с плавающей запятой - произведние этой дроби, знака, и основы в стпени экспоненты.

Мы говорим, что число с плавающей запятой нормализовано, если дробь - по крайней мере 1/b, где b - основа. Другими словами, мантисса была слишком большая для представления, если она была бы умножена на основу. Не-нормализованные числа иногда называются нестандартными; они содержат меньшее количество точности чем представление, обычно может содержать.

Если число не нормализовано, то Вы может вычитать 1 из экспоненты при умножении мантиссы на основу, и получать другое число с плавающей запятой с тем же самым значением. Нормализация состоит из выполнения этого неоднократно, пока число не нормализовано. Два различных нормализованных числа с плавающей запятой не могут быть равны в значении.

A.5.3.2 Параметры с плвающей точкой

К этим макроопределениям можно обращаться, включая заглавный файл " кfloat.h " в вашей программе.

Имена макрокоманд, начинающиеся с " FLT_ " относятся к float типу,

в то время как имена, начинающиеся с " DBL_ " обращаются к double типу, а имена, начинающиеся " LDBL_ " относит к long double типу. (В настоящее время GCC не поддерживает long double как отдельный тип данных, так что значения для "LDBL_ " константы равны соответствующим константам для double типа.)

Из этих макркоманд, только FLT_RADIX, как гарантируют, будет постоянным выражением. Другие макркоманды, перечисленные здесь не могут надежно использоваться в местах, которые требуют постоянных выражений, типа " #if " директив предварительной обработки или в размерах статических массивов.

Хотя стандарт ANSI C определяет минимальные и максимальные значения для большинства этих параметров, реализация GNU C использует любые значения, описывающие представление с плавающей запятой целевой машины. Так в принципе GNU C фактически удовлетворяет требования ANSI C только, если целевая машина подходящая. Практически, все машины, в настоящее время обеспечиваемые, подходящие.

FLT_ROUNDS

Это значение характеризует режим округления для сложения с плавающей запятой. Следующие значения указывают режимы округления:

-1 Режим - неопределенный

0 Округление к нулю.

1 Округление к самому близкому числу.

2 Округление к положительной бесконечности.

3 Округление к отрицательной бесконечности.

Любое другое значение представляет машинно-зависимый нестандартный режим округления.

На большинстве машин, значение 1, в соответствии с стандартом ИИЭРа для чисел с плавающей запятой.

Вот таблица, показывающая, как некоторые значения округляются для каждого возможного значения FLT_ROUNDS.

0	1	2	3
1.00000003	1.0	1.0	1.00000012	1.0
1.00000007	1.0	1.00000012	1.00000012	1.0
-1.00000003	-1.0	-1.0	-1.0	-1.00000012
-1.00000007	-1.0	-1.00000012	-1.0	-1.00000012

FLT_RADIX

Это - значение основы или основания системы счисления. Оно, как гарантируют, будет постоянным выражением, в отличие от других макркоманд, описанных в этом разделе. Значение - 2 на всех машинах, о которых мы знаем за исключением IBM 360 и производных.

FLT_MANT_DIG

Это - число (FLT_RADIX-ичных) цифр в мантиссе с плавающей запятой для типа данных float. Следующее выражение производит 1.0 (даже если математически этого не должно быть) из-за ограниченного числа цифр мантиссы:

```
float radix = FLT_RADIX;
1.0f + 1.0f / radix / radix / . . . / radix
```

где основание системы счисления появляется FLT_MANT_DIG раз.

DBL_MANT_DIG

LDBL_MANT_DIG

Это - число (FLT_RADIX-ичных) цифр в мантиссе с плавающей запятой для double и long double, соответственно.

FLT_DIG

Это - число десятичных цифр точности для типа данных float. Технически, если p и b - точность и основа (соответственно) для представления, то десятичная точность q - максимальное число десятичных цифр таких, что любое число с плавающей запятой с q-основанием из 10 цифр может быть округлено к числу с плавающей запятой с p-основанием из b цифр и обратно снова, без изменения q десятичных цифр.

Значение этой макркоманды, как предполагается, является по крайней мере 6, для удовлетворения ANSI C.

DBL_DIG

LDBL_DIG

Они подобны FLT_DIG, но для double и long double, соответственно. Значения этих макркоманд, как предполагается, являются по крайней мере 10.

FLT_MIN_EXP

Это - самое маленькое возможное значение экспоненты для float типа.

Более точно - минимальное отрицательное целое число такое, что значение FLT_RADIX в этой степени минус 1, может представляться как нормализованное число с плавающей запятой float типа.

DBL_MIN_EXP

LDBL_MIN_EXP

Они подобны FLT_MIN_EXP, но для double и long double, соответственно.

FLT_MIN_10_EXP

Это - минимальное отрицательное целое число такое, что 10 в этой степени минус 1, может представляться как нормализованное число с плавающей запятой float типа. Оно, как предполагается, является -37 или даже меньше.

DBL_MIN_10_EXP

LDBL_MIN_10_EXP

Они подобны FLT_MIN_10_EXP, но для double и long double, соответственно.

FLT_MAX_EXP

Это - самое большое возможное значение экспоненты для float типа. Более точно, это - максимальное положительное целое число такое, что значение FLT_RADIX в этой степени минус 1, может представляться как число с плавающей запятой float типа.

DBL_MAX_EXP

LDBL_MAX_EXP

Они подобны FLT_MAX_EXP, но для double и long double, соответственно.

FLT_MAX_10_EXP

Это - максимальное положительное целое число такое, что 10 в этой степени минус 1, может представляться как нормализованное число с

плавающей запятой float типа. Оно, как предполагается, является по крайней мере 37.

DBL_MAX_10_EXP

LDBL_MAX_10_EXP

Они подобны FLT_MAX_10_EXP, но для double и long double, соответственно.

FLT_MAX

Значение этой макркоманды - максимальное число, представимое в float типе. Предполагается что оно по крайней мере 1E + 37. Значение имеет float тип.

Самое маленькое представимое число - -FLT_MAX.

DBL_MAX

LDBL_MAX

Они подобны FLT_MAX, но для double и long double, соответственно. Тип значения макркоманды - такой же как тип, который она описывает.

FLT_MIN

Значение этой макркоманды - минимальное нормализованное положительное число с плавающей запятой, которое является представимым в float типе. Предполагается быть не больше, чем 1E-37.

DBL_MIN

LDBL_MIN

Они подобны FLT_MIN, но для double и long double, соответственно. Тип значения макркоманды - такой же как тип, который она описывает.

FLT_EPSILON

Это - минимальное положительное число с плавающей запятой float типа такое, что 1.0 + FLT_EPSILON != 1.0 является истинным. Предполагается быть не больше чем 1E-5.

DBL_EPSILON

LDBL_EPSILON

Они подобны FLT_EPSILON, но для double и long double, соответственно. Тип значения макркоманды - такой же как тип, который она описывает. Значения, как предполагается, не больше чем 1E-9.

A.5.3.3 ИИЭР(IEEE) числа с плавающей запятой

Вот пример, показывающий, как размеры типа с плавающей запятой выступают в поддержку наиболее общего представления с плавающей запятой, заданного ИИЭР Стандартом для Двоичной Арифметики С плавающей запятой (ANSI/IEEE Std 754-1985). Почти все компьютеры, разработанные начиная с 1980-ого используют этот формат.

ИИЭР представление float с одинарной точностью использует основание 2. Имеется знаковый разряд, мантисса с 23 битами плюс один скрытый бит (так что общая точность - 24 2-ичные цифры), и экспонента с 8 битами, которая может представлять значения в промежутке от -125 до 128, включительно.

Так, для реализации, которая использует, это представление для типа данных float, соответствующие значения для соответствующих параметров:

FLT_RADIX 2

```
                FLT_MANT_DIG     24
                FLT_DIG          6
                FLT_MIN_EXP     -125
                FLT_MIN_10_EXP  -37
                FLT_MAX_EXP      128
                FLT_MAX_10_EXP  +38
                FLT_MIN          1.17549435E-38F
                FLT_MAX          3.40282347E+38F
                FLT_EPSILON      1.19209290E-07F
      Имеются значения для типа данных double:
                DBL_MANT_DIG     53
                DBL_DIG          15
                DBL_MIN_EXP     -1021
                DBL_MIN_10_EXP  -307
                DBL_MAX_EXP      1024
                DBL_MAX_10_EXP   308
                DBL_MAX          1.7976931348623157E+308
                DBL_MIN          2.2250738585072014E-308
                DBL_EPSILON      2.2204460492503131E-016
```

A.5.4 Величина Смещения Поля Структуры

Вы можете использовать offsetof, чтобы измерить расположение внутри структурного типа специфического элемента структуры.

size_t offsetof (type, member) (макрос)

Он расширяется до целочисленного постоянного выражения, которое является смещением элемента структуры, именованного member в структурном типе type. Например, offsetof (struct s, elem) - смещение, в байтах, элемента elem в struct s.

Эта макркоманда не будет работать, если элемент - битовое поле; Вы получаете ошибку из компилятора C в этом случае.

Приложение B: Резюме Библиотечных Средств

Это приложение - полный список средств, объявленных внутри заглавных файлов, обеспеченных библиотекой GNU C. Каждый вход также перечисляет стандарт или другой источник, из которого каждое средство получено, и сообщает Вам, где в руководстве Вы можете найти более подробную информацию.

```
void abort (void)
`stdlib.h' (ANSI): Раздел 22.3.4 [Прерывание выполнения Программы].
int abs (int number)
`stdlib.h' (ANSI): Раздел 14.3 [Абсолютное Значение].
int accept (int socket, struct sockaddr *addr, size_t *length`ptr)
`sys/socket.h' (BSD): Раздел 11.8.3 [Принятие Соединений].
int access (const char *filename, int how)
`unistd.h' (POSIX.1): Раздел 9.8.8 [Прверка Прав Файла].
double acosh (double x)
`math.h' (BSD): Раздел 13.5 [Гиперболические функции].
double acos (double x)
`math.h' (ANSI): Раздел 13.3 [Обратные Тригонометрические Функции].
int adjtime (const struct timeval *delta, struct timeval *olddelta)
`sys/time.h' (BSD): Раздел 17.2.2 [Точный Календарь].
AF_FILE
`sys/socket.h' (GNU): Раздел 11.3.1 [Форматы Адреса].
AF_INET
`sys/socket.h' (BSD): Раздел 11.3.1 [Форматы Адреса].
AF_UNIX
`sys/socket.h' (BSD): Раздел 11.3.1 [Форматы Адреса].
AF_UNSPEC
`sys/socket.h' (BSD): Раздел 11.3.1 [Форматы Адреса].
unsigned int alarm (unsigned int seconds)
`unistd.h' (POSIX.1): Раздел 17.3 [Установка Сигнализации].
void * alloca (size_t size);
`stdlib.h' (GNU, BSD): Раздел 3.5 [Автоматический Размер Переменной].
ALTWERASE
`termios.h' (BSD): Раздел 12.4.7 [Автономные режимы].
int ARG_MAX
`limits.h' (POSIX.1): Раздел 27.1 [Основные Ограничения].
```

`sys/types.h' (POSIX.1): Раздел 9.8.1 [Значения Атрибутов].
double difftime (time_t time1, time_t time0)
`time.h' (ANSI): Раздел 17.2.1 [Простое Календарное Время].
DIR
`dirent.h' (POSIX.1): Раздел 9.2.2 [Открытие Каталога].
div_t div (int numerator, int denominator)
`stdlib.h' (ANSI): Раздел 14.6 [Целое Деление].
div_t
`stdlib.h' (ANSI): Раздел 14.6 [Целое Деление].
double drem (double numerator, double denominator)
`math.h' (BSD): Раздел 14.5 [Округление и Остаточные члены].
int dup2 (int old, int new)
`unistd.h' (POSIX.1): Раздел 8.8 [Двойные Описатели].
int dup (int old)
`unistd.h' (POSIX.1): Раздел 8.8 [Двойные Описатели].
int E2BIG
`errno.h' (POSIX.1: Argument list too long): Раздел 2.2 [Коды Ошибки].
int EACCES

`errno.h' (POSIX.1: Permission denied): Раздел 2.2 [Коды Ошибки].
int EADDRINUSE
`errno.h' (BSD: Address already in use): Раздел 2.2 [Коды Ошибки].
int EADDRNOTAVAIL
`errno.h' (BSD: Can't assign requested address):
Раздел 2.2[Коды Ошибки].
int EAFNOSUPPORT
`errno.h' (BSD: Address family not supported by protocol family):
Раздел 2.2 [Коды Ошибки].
int EAGAIN
`errno.h' (POSIX.1: Resource temporarily unavailable):
Раздел 2.2 [Коды Ошибки].
int EALREADY
`errno.h' (BSD: Operation already in progress):Раздел 2.2 [Коды Ошибки].
int EBACKGROUND
`errno.h' (GNU: Inappropriate operation for background process):
Раздел 2.2 [Коды Ошибки].
int EBADF
`errno.h' (POSIX.1: Bad file descriptor): Раздел 2.2 [Коды Ошибки].
int EBUSY
`errno.h' (POSIX.1: Device busy): Раздел 2.2 [Коды Ошибки].
int ECHILD
`errno.h' (POSIX.1: No child processes): Раздел 2.2 [Коды Ошибки].
ECHOCTL
`termios.h' (BSD): Раздел 12.4.7 [Автономные режимы].
ECHOE
`termios.h' (POSIX.1): Раздел 12.4.7 [Автономные режимы].
ECHO
`termios.h' (POSIX.1): Раздел 12.4.7 [Автономные режимы].
ECHOKE
`termios.h' (BSD): Раздел 12.4.7 [Автономные режимы].
ECHOK
`termios.h' (POSIX.1): Раздел 12.4.7 [Автономные режимы].
ECHONL
`termios.h' (POSIX.1): Раздел 12.4.7 [Автономные режимы].
ECHOPRT
`termios.h' (BSD): Раздел 12.4.7 [Автономные режимы].
int ECONNABORTED

`errno.h' (BSD: Software caused connection abort):
Раздел 2.2 [Коды Ошибки].
int ECONNREFUSED
`errno.h' (BSD: Connection refused): Раздел 2.2 [Коды Ошибки].
int ECONNRESET
`errno.h' (BSD: Connection reset by peer): Раздел 2.2 [Коды Ошибки].
int EDEADLK
`errno.h' (POSIX.1: Resource deadlock avoided):Раздел 2.2 [Коды Ошибки].
int EDESTADDRREQ
`errno.h' (BSD: Destination address required): Раздел 2.2 [Коды Ошибки].
int ED
`errno.h' (GNU: ?): Раздел 2.2 [Коды Ошибки].
int EDOM

`errno.h' (ANSI: Numerical argument out of domain):
Раздел 2.2 [Коды Ошибки].
int EDQUOT
`errno.h' (BSD: Disc quota exceeded): Раздел 2.2 [Коды Ошибки].
int EEXIST
`errno.h' (POSIX.1: File exists): Раздел 2.2 [Коды Ошибки].
int EFAULT
`errno.h' (POSIX.1: Bad address): Раздел 2.2 [Коды Ошибки].
int EFBIG
`errno.h' (POSIX.1: File too large): Раздел 2.2 [Коды Ошибки].
int EGRATUITOUS
`errno.h' (GNU: Gratuitous error): Раздел 2.2 [Коды Ошибки].
int EGREGIOUS
`errno.h' (GNU: You really blew it this time): Раздел 2.2 [Коды Ошибки].
int EHOSTDOWN
`errno.h' (BSD: Host is down): Раздел 2.2 [Коды Ошибки].
int EHOSTUNREACH
`errno.h' (BSD: No route to host): Раздел 2.2 [Коды Ошибки].
int EIEIO
`errno.h' (GNU: Computer bought the farm): Раздел 2.2 [Коды Ошибки].
int EINPROGRESS
`errno.h' (BSD: Operation now in progress): Раздел 2.2 [Коды Ошибки].
int EINTR
`errno.h' (POSIX.1: Interrupted system call): Раздел 2.2 [Коды Ошибки].
int EINVAL

`errno.h' (POSIX.1: Invalid argument): Раздел 2.2 [Коды Ошибки].
int EIO
`errno.h' (POSIX.1: Input/output error): Раздел 2.2 [Коды Ошибки].
int EISCONN
`errno.h' (BSD: Socket is already connected): Раздел 2.2 [Коды Ошибки].
int EISDIR
`errno.h' (POSIX.1: Is a directory): Раздел 2.2 [Коды Ошибки].
int ELOOP
`errno.h' (BSD: Too many levels of symbolic links):
Раздел 2.2 [Коды Ошибки].
int EMFILE
`errno.h' (POSIX.1: Too many open files): Раздел 2.2 [Коды Ошибки].
int EMLINK
`errno.h' (POSIX.1: Too many links): Раздел 2.2 [Коды Ошибки].
int EMSGSIZE
`errno.h' (BSD: Message too long): Раздел 2.2 [Коды Ошибки].
int ENAMETOOLONG
`errno.h' (POSIX.1: File name too long): Раздел 2.2 [Коды Ошибки].
void endgrent (void)
`grp.h' (SVID, BSD): Раздел 25.13.3 [Просмотр Всех Групп].
void endhostent ()
`netdb.h' (BSD): Раздел 11.5.2.4 [Главные Имена].
void endnetent (void)
`netdb.h' (BSD): Раздел 11.12 [База данных Сетей].
void endprotoent (void)
`netdb.h' (BSD): Раздел 11.5.6 [База данных Протоколов].
void endpwent (void)
`pwd.h' (SVID, BSD): Раздел 25.12.3 [Просмотр Всех Пользователей].
void endservent (void)
`netdb.h' (BSD): Раздел 11.5.4 [База Данных Услуг].
int ENETDOWN
`errno.h' (BSD: Network is down): Раздел 2.2 [Коды Ошибки].
int ENETRESET
`errno.h' (BSD: Network dropped connection on reset):
Раздел 2.2 [Коды Ошибки].
int ENETUNREACH
`errno.h' (BSD: Network is unreachable): Раздел 2.2 [Коды Ошибки].
int ENFILE

`errno.h' (POSIX.1: Too many open files in system):
Раздел 2.2 [Коды Ошибки].
int ENOBUFS
`errno.h' (BSD: No buffer space available): Раздел 2.2 [Коды Ошибки].
int ENODEV
`errno.h' (POSIX.1: Operation not supported by device):

Раздел 2.2 [Коды Ошибки].
int ENOENT
`errno.h' (POSIX.1: No such file or directory):
Раздел 2.2 [Коды Ошибки].
int ENOEXEC
`errno.h' (POSIX.1: Exec format error): Раздел 2.2 [Коды Ошибки].
int ENOLCK
`errno.h' (POSIX.1: No locks available): Раздел 2.2 [Коды Ошибки].
int ENOMEM
`errno.h' (POSIX.1: Cannot allocate memory): Раздел 2.2 [Коды Ошибки].
int ENOPROTOOPT
`errno.h' (BSD: Protocol not available): Раздел 2.2 [Коды Ошибки].
int ENOSPC
`errno.h' (POSIX.1: No space left on device): Раздел 2.2 [Коды Ошибки].
int ENOSYS
`errno.h' (POSIX.1: Function not implemented): Раздел 2.2 [Коды Ошибки].
int ENOTBLK
`errno.h' (BSD: Block device required): Раздел 2.2 [Коды Ошибки].
int ENOTCONN
`errno.h' (BSD: Socket is not connected): Раздел 2.2 [Коды Ошибки].
int ENOTDIR
`errno.h' (POSIX.1: Not a directory): Раздел 2.2 [Коды Ошибки].
int ENOTEMPTY
`errno.h' (POSIX.1: Directory not empty): Раздел 2.2 [Коды Ошибки].
int ENOTSOCK
`errno.h' (BSD: Socket operation on non-socket):
Раздел 2.2 [Коды Ошибки].
int ENOTTY
`errno.h' (POSIX.1: Inappropriate ioctl for device):
Раздел 2.2 [Коды Ошибки].
char ** environ
`unistd.h' (POSIX.1): Раздел 22.2.1 [Доступ Среды].

int ENXIO
`errno.h' (POSIX.1: Device not configured): Раздел 2.2 [Коды Ошибки].
int EOF
`stdio.h' (ANSI): Раздел 7.13 [EOF и Ошибки].
int EOPNOTSUPP
`errno.h' (BSD: Operation not supported): Раздел 2.2 [Коды Ошибки].
int EPERM
`errno.h' (POSIX.1: Operation not permitted): Раздел 2.2 [Коды Ошибки].
int EPFNOSUPPORT
`errno.h' (BSD: Protocol family not supported):
Раздел 2.2 [Коды Ошибки].
int EPIPE
`errno.h' (POSIX.1: Broken pipe): Раздел 2.2 [Коды Ошибки].
int EPROTONOSUPPORT
`errno.h' (BSD: Protocol not supported): Раздел 2.2 [Коды Ошибки].
int EPROTOTYPE
`errno.h' (BSD: Protocol wrong type for socket):
Раздел 2.2 [Коды Ошибки].
int EQUIV_CLASS_MAX
`limits.h' (POSIX.2): Раздел 27.10 [Пределы Утилит].
int ERANGE
`errno.h' (ANSI: Numerical result out of range): Раздел 2.2 [Коды Ошибки].
int EREMOTE
`errno.h' (BSD: Too many levels of remote in path):
Раздел 2.2 [Коды Ошибки].
int EROFS
`errno.h' (POSIX.1: Read-only file system): Раздел 2.2 [Коды Ошибки].
volatile int errno
`errno.h' (ANSI): Раздел 2.1 [Проверка Ошибок].
int ESHUTDOWN
`errno.h' (BSD: Can't send after socket shutdown):
Раздел 2.2 [Коды Ошибки].
int ESOCKTNOSUPPORT
`errno.h' (BSD: Socket type not supported): Раздел 2.2 [Коды Ошибки].
int ESPIPE
`errno.h' (POSIX.1: Illegal seek): Раздел 2.2 [Коды Ошибки].
int ESRCH
`errno.h' (POSIX.1: No such process): Раздел 2.2 [Коды Ошибки].

- 613 -

- 614 -

`fcntl.h' (POSIX.1): Раздел 8.10 [Флаги Состояния Файла].
int F_SETLK
`fcntl.h' (POSIX.1): Раздел 8.11 [Блокировки Файла].
int F_SETLKW
`fcntl.h' (POSIX.1): Раздел 8.11 [Блокировки Файла].
int F_SETOWN
`fcntl.h' (BSD): Раздел 8.12 [Прерванный Ввод].
int fsetpos (FILE *stream, const fpos_t position)
`stdio.h' (ANSI): Раздел 7.16 [Переносное Позиционирование].
int fstat (int filedes, struct stat *buf)
`sys/stat.h' (POSIX.1): Раздел 9.8.2 [Чтение Атрибутов].
long int ftell (FILE *stream)
`stdio.h' (ANSI): Раздел 7.15 [Позиционирование Файла].
F_UNLCK
`fcntl.h' (POSIX.1): Раздел 8.11 [Блокировки Файла].
size_t fwrite (const void *data, size_t size, size_t count,
FILE *stream)
`stdio.h' (ANSI): Раздел 7.12 [Блокирование Вывода/Вывода].
F_WRLCK

`fcntl.h' (POSIX.1): Раздел 8.11 [Блокировки Файла].
int getchar (void)
`stdio.h' (ANSI): Раздел 7.6 [Символьный Ввод].
int getc (FILE *stream)
`stdio.h' (ANSI): Раздел 7.6 [Символьный Ввод].
char * getcwd (char *buffer, size_t size)
`unistd.h' (POSIX.1): Раздел 9.1 [Рабочий Каталог].
ssize_t getdelim (char **lineptr, size_t *n, int delimiter,
FILE *stream)
`stdio.h' (GNU): Раздел 7.7 [Строчный Ввод].
gid_t getegid (void)
`unistd.h' (POSIX.1): Раздел 25.5 [Чтение Persona].
char * getenv (const char *name)
`stdlib.h' (ANSI): Раздел 22.2.1 [Доступ Среды].
uid_t geteuid (void)
`unistd.h' (POSIX.1): Раздел 25.5 [Чтение Persona].
gid_t getgid (void)
`unistd.h' (POSIX.1): Раздел 25.5 [Чтение Persona].
struct group * getgrent (void)
`grp.h' (SVID, BSD): Раздел 25.13.3 [Просмотр Всех Групп].
struct group * getgrgid (gid_t gid)
`grp.h' (POSIX.1): Раздел 25.13.2 [Поиск Группы].
struct group * getgrnam (const char *name)
`grp.h' (SVID, BSD): Раздел 25.13.2 [Поиск Группы]].
int getgroups (int count, gid_t *groups)
`unistd.h' (POSIX.1): Раздел 25.5 [Чтение Persona].
struct hostent * gethostbyaddr (const char *addr, int length,
int format)
`netdb.h' (BSD): Раздел 11.5.2.4 [Главные Имена].
struct hostent * gethostbyname (const char *name)
`netdb.h' (BSD): Раздел 11.5.2.4 [Главные Имена].
struct hostent * gethostent ()
`netdb.h' (BSD): Раздел 11.5.2.4 [Главные Имена].
long int gethostid (void)
`unistd.h' (BSD): Раздел 26.1 [Главная Идентификация].
int gethostname (char *name, size_t size)
`unistd.h' (BSD): Раздел 26.1 [Главная Идентификация].
int getitimer (int which, struct itimerval *old)

`sys/time.h' (BSD): Раздел 17.3 [Установка Сигнализации].
ssize_t getline (char **lineptr, size_t *n, FILE *stream)
`stdio.h' (GNU): Раздел 7.7 [Строчный Ввод].
char * getlogin (void)
`unistd.h' (POSIX.1): Раздел 25.11 [Кто вошел в систему].
struct netent * getnetbyaddr (long net, int type)
`netdb.h' (BSD): Раздел 11.12 [База данных Сетей].
struct netent * getnetbyname (const char *name)
`netdb.h' (BSD): Раздел 11.12 [База данных Сетей].
struct netent * getnetent (void)
`netdb.h' (BSD): Раздел 11.12 [База данных Сетей].
int getopt (int argc, char **argv, const char *options)

time_t mktime (struct tm *brokentime)
`time.h' (ANSI): Раздел 17.2.3 [Сброшенное Время].
mode_t
`sys/types.h' (POSIX.1): Раздел 9.8.1 [Значения Атрибутов].
double modf (double value, double *integer`part)
`math.h' (ANSI): Раздел 14.5 [Округление и Остаточные члены].
int MSG_DONTROUTE
`sys/socket.h' (BSD): Раздел 11.8.5.3 [Опции Данных Гнезда].
int MSG_OOB
`sys/socket.h' (BSD): Раздел 11.8.5.3 [Опции Данных Гнезда].
int MSG_PEEK
`sys/socket.h' (BSD): Раздел 11.8.5.3 [Опции Данных Гнезда].
struct mstats mstats (void)
`malloc.h' (GNU): Раздел 3.3.10 [Статистика Malloc].
int NAME_MAX
`limits.h' (POSIX.1): Раздел 27.6 [Ограничения для Файлов].
double NAN
`math.h' (GNU): Раздел 14.1 [Не число (Not a Number, NaN)].
int NCCS
`termios.h' (POSIX.1): Раздел 12.4.1 [Тип Данных Режима].

int NGROUPS_MAX
`limits.h' (POSIX.1): Раздел 27.1 [Основные Ограничения].
int nice (int increment)
`dunno.h' (dunno.h): Раздел 17.7 [Приоритет].
nlink_t
`sys/types.h' (POSIX.1): Раздел 9.8.1 [Значения Атрибутов].
NO_ADDRESS
`netdb.h' (BSD): Раздел 11.5.2.4 [Главные Имена].
NOFLSH
`termios.h' (POSIX.1): Раздел 12.4.7 [Автономные режимы].
NOKERNINFO
`termios.h' (BSD): Раздел 12.4.7 [Автономные режимы].
NO_RECOVERY
`netdb.h' (BSD): Раздел 11.5.2.4 [Главные Имена].
int NSIG
`signal.h' (BSD): Раздел 21.2 [Стандартные Сигналы].
unsigned long int ntohl (unsigned long int netlong)
`netinet/in.h' (BSD): Раздел 11.5.5 [Байтовый Порядок].
unsigned short int ntohs (unsigned short int netshort)
`netinet/in.h' (BSD): Раздел 11.5.5 [Байтовый Порядок].
void * NULL
`stddef.h' (ANSI): Раздел A.3 [Константа - Нулевой Указатель].
int O_ACCMODE
`fcntl.h' (POSIX.1): Раздел 8.10 [Флаги Состояния Файла].
O_APPEND
`fcntl.h' (POSIX.1): Раздел 8.10 [Флаги Состояния Файла].
O_APPEND
`fcntl.h' (POSIX.1): Раздел 8.1 [Открытие и Закрытие Файлов].
void obstack_1grow_fast (struct obstack *obstack`ptr, char c)
`obstack.h' (GNU): Раздел 3.4.7 [Сверх Быстрый Рост].
void obstack_1grow (struct obstack *obstack`ptr, char c)
`obstack.h' (GNU): Раздел 3.4.6 [Возрастающие Объекты].
int obstack_alignment_mask (struct obstack *obstack`ptr)
`obstack.h' (GNU): Раздел 3.4.9 [Obstacks Выравнивание Данных].
void * obstack_alloc (struct obstack *obstack`ptr, size_t size)
`obstack.h' (GNU): Раздел 3.4.3 [Резервирование в Obstack].
void * obstack_base (struct obstack *obstack`ptr)
`obstack.h' (GNU): Раздел 3.4.8 [Состофние Obstack].

void obstack_blank_fast (struct obstack *obstack`ptr, size_t size)
`obstack.h' (GNU): Раздел 3.4.7 [Сверх Быстрый Рост].
void obstack_blank (struct obstack *obstack`ptr, size_t size)
`obstack.h' (GNU): Раздел 3.4.6 [Возрастающие Объекты].
size_t obstack_chunk_size (struct obstack *obstack`ptr)
`obstack.h' (GNU): Раздел 3.4.10 [Куски в Obstack].
void * obstack_copy0 (struct obstack *obstack`ptr, void *address,
size_t size)
`obstack.h' (GNU): Раздел 3.4.3 [Резервирование в Obstack].
void * obstack_copy (struct obstack *obstack`ptr, void *address,

- 633 -

- 634 -

`regex.h' (POSIX.2):
Раздел 16.3.1 [POSIX Компиляция Регулярных выражений].
int regexec (regex_t *compiled, char *string, size_t nmatch,
regmatch_t matchptr [], int eflags)
`regex.h' (POSIX.2):
Раздел 16.3.3 [POSIX Соответствие Регулярных выражений].
REG_EXTENDED
`regex.h' (POSIX.2): Раздел 16.3.2 [Флаги для POSIX Regexps].
regex_t
`regex.h' (POSIX.2):
Раздел 16.3.1 [POSIX Компиляция Регулярных выражений].
void regfree (regex_t *compiled)
`regex.h' (POSIX.2): Раздел 16.3.6 [Очистка Regexp].
REG_ICASE
`regex.h' (POSIX.2): Раздел 16.3.2 [Флаги для POSIX Regexps].
int register_printf_function (int spec, printf_function handler
`function, printf_arginfo_function arginfo`function)
`printf.h' (GNU): Раздел 7.10.1 [Регистрация Новых Преобразований].
regmatch_t
`regex.h' (POSIX.2): Раздел 16.3.4 [Подвыражения].
REG_NEWLINE

`regex.h' (POSIX.2): Раздел 16.3.2 [Флаги для POSIX Regexps].
REG_NOMATCH
`regex.h' (POSIX.2):
Раздел 16.3.3 [POSIX Соответствие Регулярных выражений].
REG_NOSUB
`regex.h' (POSIX.2): Раздел 16.3.2 [Флаги для POSIX Regexps].
REG_NOTBOL
`regex.h' (POSIX.2):
Раздел 16.3.3 [POSIX Соответствие Регулярных выражений].
REG_NOTEOL
`regex.h' (POSIX.2):
Раздел 16.3.3 [POSIX Соответствие Регулярных выражений].
regoff_t
`regex.h' (POSIX.2): Раздел 16.3.4 [Подвыражения].
int remove (const char *filename)
`stdio.h' (ANSI): Раздел 9.5 [Удаление Файлов].
int rename (const char *oldname, const char *newname)
`stdio.h' (ANSI): Раздел 9.6 [Переименование Файлов].
void rewinddir (DIR *dirstream)
`dirent.h' (POSIX.1): Раздел 9.2.5 [Каталоги неопределенного доступа].
void rewind (FILE *stream)
`stdio.h' (ANSI): Раздел 7.15 [Позиционирование Файла].
char * rindex (const char *string, int c)
`string.h' (BSD): Раздел 5.7 [Функции Поиска].
double rint (double x)
`math.h' (BSD): Раздел 14.5 [Округление и Остаточные члены].
RLIMIT_CORE
`sys/resource.h' (BSD): Раздел 17.6 [Ограничения Ресурсов].
RLIMIT_CPU
`sys/resource.h' (BSD): Раздел 17.6 [Ограничения Ресурсов].
RLIMIT_DATA
`sys/resource.h' (BSD): Раздел 17.6 [Ограничения Ресурсов].
RLIMIT_FSIZE
`sys/resource.h' (BSD): Раздел 17.6 [Ограничения Ресурсов].
RLIMIT_OPEN_FILES
`sys/resource.h' (BSD): Раздел 17.6 [Ограничения Ресурсов].
RLIMIT_RSS
`sys/resource.h' (BSD): Раздел 17.6 [Ограничения Ресурсов].

RLIMIT_STACK
`sys/resource.h' (BSD): Раздел 17.6 [Ограничения Ресурсов].
RLIM_NLIMITS
`sys/resource.h' (BSD): Раздел 17.6 [Ограничения Ресурсов].
int rmdir (const char *filename)
`unistd.h' (POSIX.1): Раздел 9.5 [Удаление Файлов].
int R_OK
`unistd.h' (POSIX.1): Раздел 9.8.8 [Прверка Прав Файла].
void * r_re_alloc (void **handleptr, size_t size)

`malloc.h' (GNU): Раздел 3.6.2 [Распределение Переместимых Блоков].
RUSAGE_CHILDREN
`sys/resource.h' (BSD): Раздел 17.5 [Использование Ресурсов].
RUSAGE_SELF
`sys/resource.h' (BSD): Раздел 17.5 [Использование Ресурсов].
int SA_NOCLDSTOP
`signal.h' (POSIX.1): Раздел 21.3.5 [Флаги для Sigaction].
int SA_ONSTACK
`signal.h' (BSD): Раздел 21.3.5 [Флаги для Sigaction].
int SA_RESTART
`signal.h' (BSD): Раздел 21.3.5 [Флаги для Sigaction].
_SC_2_C_DEV
`unistd.h' (POSIX.2): Раздел 27.4.2 [Константы для Sysconf].
_SC_2_FORT_DEV
`unistd.h' (POSIX.2): Раздел 27.4.2 [Константы для Sysconf].
_SC_2_FORT_RUN
`unistd.h' (POSIX.2): Раздел 27.4.2 [Константы для Sysconf].
_SC_2_LOCALEDEF
`unistd.h' (POSIX.2): Раздел 27.4.2 [Константы для Sysconf].
_SC_2_SW_DEV
`unistd.h' (POSIX.2): Раздел 27.4.2 [Константы для Sysconf].
_SC_2_VERSION
`unistd.h' (POSIX.2): Раздел 27.4.2 [Константы для Sysconf].
double scalb (double value, int exponent)
`math.h' (BSD): Раздел 14.4 [Функции Нормализации].
int scanf (const char *template, . . .)
`stdio.h' (ANSI): Раздел 7.11.8 [Функции Форматированного Ввода].
_SC_ARG_MAX

`unistd.h' (POSIX.1): Раздел 27.4.2 [Константы для Sysconf].
_SC_BC_BASE_MAX
`unistd.h' (POSIX.2): Раздел 27.4.2 [Константы для Sysconf].
_SC_BC_DIM_MAX
`unistd.h' (POSIX.2): Раздел 27.4.2 [Константы для Sysconf].
_SC_BC_SCALE_MAX
`unistd.h' (POSIX.2): Раздел 27.4.2 [Константы для Sysconf].
_SC_BC_STRING_MAX
`unistd.h' (POSIX.2): Раздел 27.4.2 [Константы для Sysconf].
_SC_CHILD_MAX
`unistd.h' (POSIX.1): Раздел 27.4.2 [Константы для Sysconf].
_SC_CLK_TCK
`unistd.h' (POSIX.1): Раздел 27.4.2 [Константы для Sysconf].
_SC_COLL_WEIGHTS_MAX
`unistd.h' (POSIX.2): Раздел 27.4.2 [Константы для Sysconf].
_SC_EQUIV_CLASS_MAX
`unistd.h' (POSIX.2): Раздел 27.4.2 [Константы для Sysconf].
_SC_EXPR_NEST_MAX
`unistd.h' (POSIX.2): Раздел 27.4.2 [Константы для Sysconf].
SCHAR_MAX
`limits.h' (ANSI): Раздел A.5.2 [Диапазон Типа].
SCHAR_MIN
`limits.h' (ANSI): Раздел A.5.2 [Диапазон Типа].
_SC_JOB_CONTROL
`unistd.h' (POSIX.1): Раздел 27.4.2 [Константы для Sysconf].
_SC_LINE_MAX
`unistd.h' (POSIX.2): Раздел 27.4.2 [Константы для Sysconf].
_SC_NGROUPS_MAX
`unistd.h' (POSIX.1): Раздел 27.4.2 [Константы для Sysconf].
_SC_OPEN_MAX
`unistd.h' (POSIX.1): Раздел 27.4.2 [Константы для Sysconf].
_SC_PAGESIZE
`unistd.h' (GNU): Раздел 27.4.2 [Константы для Sysconf].
_SC_SAVED_IDS
`unistd.h' (POSIX.1): Раздел 27.4.2 [Константы для Sysconf].
_SC_STREAM_MAX
`unistd.h' (POSIX.1): Раздел 27.4.2 [Константы для Sysconf].
_SC_TZNAME_MAX

`unistd.h' (POSIX.1): Раздел 27.4.2 [Константы для Sysconf].
_SC_VERSION
`unistd.h' (POSIX.1): Раздел 27.4.2 [Константы для Sysconf].

_SC_VERSION
`unistd.h' (POSIX.2): Раздел 27.4.2 [Константы для Sysconf].
int SEEK_CUR
`stdio.h' (ANSI): Раздел 7.15 [Позиционирование Файла].
void seekdir (DIR *dirstream, off_t pos)
`dirent.h' (BSD): Раздел 9.2.5 [Доступ к Каталогам].
int SEEK_END
`stdio.h' (ANSI): Раздел 7.15 [Позиционирование Файла].
int SEEK_SET
`stdio.h' (ANSI): Раздел 7.15 [Позиционирование Файла].
int select (int nfds, fd_set *read`fds, fd_set *write`fds,
fd_set *except`fds, struct timeval *timeout)
`sys/types.h' (BSD): Раздел 8.6 [Ожидание Ввода/Вывода].
int send (int socket, void *buffer, size_t size, int flags)
`sys/socket.h' (BSD): Раздел 11.8.5.1 [Посылка Данных].
int sendmsg (int socket, const struct msghdr *message, int flags)
`sys/socket.h' (BSD): Раздел 11.9.2 [Получение Датаграмм].
int sendto (int socket, void *buffer. size_t size, int flags,
struct sockaddr *addr, size_t length)
`sys/socket.h' (BSD): Раздел 11.9.1 [Посылка Datagrams].
void setbuffer (FILE *stream, char *buf, size_t size)
`stdio.h' (BSD): Раздел 7.17.3 [Буферизация Управления].
void setbuf (FILE *stream, char *buf)
`stdio.h' (ANSI): Раздел 7.17.3 [Буферизация Управления].
int setgid (gid_t newgid)
`unistd.h' (POSIX.1): Раздел 25.7 [Установка Группы].
void setgrent (void)
`grp.h' (SVID, BSD): Раздел 25.13.3 [Просмотр Всех Групп].
int setgroups (size_t count, gid_t *groups)
`grp.h' (BSD): Раздел 25.7 [Установка Группы].
void sethostent (int stayopen)
`netdb.h' (BSD): Раздел 11.5.2.4 [Главные Имена].
int sethostid (long int id)
`unistd.h' (BSD): Раздел 26.1 [Главная Идентификация].
int sethostname (const char *name, size_t length)

`unistd.h' (BSD): Раздел 26.1 [Главная Идентификация].
int setitimer (int which, struct itimerval *old, struct itimerval *new)
`sys/time.h' (BSD): Раздел 17.3 [Установка Сигнализации].
int setjmp (jmp_buf state)
`setjmp.h' (ANSI): Раздел 20.2 [Не-локальные Подробности].
void setlinebuf (FILE *stream)
`stdio.h' (BSD): Раздел 7.17.3 [Буферизация Управления].
char * setlocale (int category, const char *locale)
`locale.h' (ANSI): Раздел 19.4 [Установка Стандарта].
void setnetent (int stayopen)
`netdb.h' (BSD): Раздел 11.12 [База данных Сетей].
int setpgid (pid_t pid, pid_t pgid)
`unistd.h' (POSIX.1): Раздел 24.7.2 [Функции Группы Процессов].
int setpgrp (pid_t pid, pid_t pgid)
`unistd.h' (BSD): Раздел 24.7.2 [Функции Группы Процессов].
int setpriority (int class, int id, int priority)
`sys/resource.h' (BSD): Раздел 17.7 [Приоритет].
void setprotoent (int stayopen)
`netdb.h' (BSD): Раздел 11.5.6 [База данных Протоколов].
void setpwent (void)
`pwd.h' (SVID, BSD): Раздел 25.12.3 [Просмотр Всех Пользователей].
int setregid (gid_t rgid, fid_t egid)
`unistd.h' (BSD): Раздел 25.7 [Установка Группы].
int setreuid (uid_t ruid, uid_t euid)
`unistd.h' (BSD): Раздел 25.6 [Утановка Пользовательского ID].
int setrlimit (int resource, struct rlimit *rlp)
`sys/resource.h' (BSD): Раздел 17.6 [Ограничения Ресурсов].
void setservent (int stayopen)
`netdb.h' (BSD): Раздел 11.5.4 [База Данных Услуг].
pid_t setsid (void)
`unistd.h' (POSIX.1): Раздел 24.7.2 [Функции Группы Процессов].
int setsockopt (int socket, int level, int optname, void *optval,
size_t optlen)
`sys/socket.h' (BSD): Раздел 11.11.1 [Функции для Опций Гнезда].
void * setstate (void *state)
`stdlib.h' (BSD): Раздел 13.6.2 [BSD Random].
int settimeofday (const struct timeval *tp, const struct timezone *tzp) `sys/time.h' (BSD):

Раздел 17.2.2 [Точный Календарь].

int setuid (uid_t newuid)
`unistd.h' (POSIX.1): Раздел 25.6 [Утановка Пользовательского ID].
int setvbuf (FILE *stream, char *buf, int mode, size_t size)
`stdio.h' (ANSI): Раздел 7.17.3 [Буферизация Управления].
SHRT_MAX
`limits.h' (ANSI): Раздел A.5.2 [Диапазон Типа].
SHRT_MIN
`limits.h' (ANSI): Раздел A.5.2 [Диапазон Типа].
int shutdown (int socket, int how)
`sys/socket.h' (BSD): Раздел 11.7.2 [Закрытие Гнезда].
S_IEXEC
`sys/stat.h' (BSD): Раздел 9.8.5 [Биты Прав].
S_IFBLK
`sys/stat.h' (BSD): Раздел 9.8.3 [Проверка Типа Файла].
S_IFCHR
`sys/stat.h' (BSD): Раздел 9.8.3 [Проверка Типа Файла].
S_IFDIR
`sys/stat.h' (BSD): Раздел 9.8.3 [Проверка Типа Файла].
S_IFIFO
`sys/stat.h' (BSD): Раздел 9.8.3 [Проверка Типа Файла].
S_IFLNK
`sys/stat.h' (BSD): Раздел 9.8.3 [Проверка Типа Файла].
int S_IFMT
`sys/stat.h' (BSD): Раздел 9.8.3 [Проверка Типа Файла].
S_IFREG
`sys/stat.h' (BSD): Раздел 9.8.3 [Проверка Типа Файла].
S_IFSOCK
`sys/stat.h' (BSD): Раздел 9.8.3 [Проверка Типа Файла].
int SIGABRT
`signal.h' (ANSI): Раздел 21.2.1 [Сигналы Ошибки в Программе].
int sigaction (int signum, const struct sigaction *action,
struct sigaction *old`action)
`signal.h' (POSIX.1): Раздел 21.3.2 [Соглашение Обработки Сигналов].
int sigaddset (sigset_t *set, int signum)
`signal.h' (POSIX.1): Раздел 21.7.2 [Множества Сигналов].
int SIGALRM
`signal.h' (POSIX.1): Раздел 21.2.3 [Сигналы таймера].

int sigaltstack (const struct sigaltstack *stack, struct sigaltstack
*oldstack)
`signal.h' (BSD): Раздел 21.10.2 [Стек Сигнала].
sig_atomic_t
`signal.h' (ANSI): Раздел 21.4.7.2 [Быстрые Типы].
int sigblock (int mask)
`signal.h' (BSD): Раздел 21.10.1 [Блокирование BSD].
SIG_BLOCK
`signal.h' (POSIX.1): Раздел 21.7.3 [Маска Сигналов Процесса].
int SIGBUS
`signal.h' (BSD): Раздел 21.2.1 [Сигналы Ошибки в Программе].
int SIGCHLD
`signal.h' (POSIX.1): Раздел 21.2.5 [Сигналы Контроля Заданий].
int SIGCONT
`signal.h' (POSIX.1): Раздел 21.2.5 [Сигналы Контроля Заданий].
int sigdelset (sigset_t *set, int signum)
`signal.h' (POSIX.1): Раздел 21.7.2 [Множества Сигналов].
int sigemptyset (sigset_t *set)
`signal.h' (POSIX.1): Раздел 21.7.2 [Множества Сигналов].
sighandler_t SIG_ERR
`signal.h' (ANSI): Раздел 21.3.1 [Основная Обработка Сигнала].
int sigfillset (sigset_t *set)
`signal.h' (POSIX.1): Раздел 21.7.2 [Множества Сигналов].
int SIGFPE
`signal.h' (ANSI): Раздел 21.2.1 [Сигналы Ошибки в Программе].
sighandler_t
`signal.h' (GNU): Раздел 21.3.1 [Основная Обработка Сигнала].
int SIGHUP
`signal.h' (POSIX.1): Раздел 21.2.2 [Сигналы Завершения].

- 644 -

- 645 -

`sys/socket.h' (BSD): Раздел 11.2 [Стили Связи].
int SOCK_RDM
`sys/socket.h' (BSD): Раздел 11.2 [Стили Связи].
int SOCK_SEQPACKET
`sys/socket.h' (BSD): Раздел 11.2 [Стили Связи].
int SOCK_STREAM
`sys/socket.h' (BSD): Раздел 11.2 [Стили Связи].
SO_DEBUG
`sys/socket.h' (BSD): Раздел 11.11.2 [Опции Гнезда].
SO_DONTROUTE
`sys/socket.h' (BSD): Раздел 11.11.2 [Опции Гнезда].
SO_ERROR
`sys/socket.h' (BSD): Раздел 11.11.2 [Опции Гнезда].
SO_KEEPALIVE
`sys/socket.h' (BSD): Раздел 11.11.2 [Опции Гнезда].
SO_LINGER
`sys/socket.h' (BSD): Раздел 11.11.2 [Опции Гнезда].
int SOL_SOCKET
`sys/socket.h' (BSD): Раздел 11.11.2 [Опции Гнезда].
SO_OOBINLINE
`sys/socket.h' (BSD): Раздел 11.11.2 [Опции Гнезда].
SO_RCVBUF

`sys/socket.h' (BSD): Раздел 11.11.2 [Опции Гнезда].
SO_REUSEADDR
`sys/socket.h' (BSD): Раздел 11.11.2 [Опции Гнезда].
SO_SNDBUF
`sys/socket.h' (BSD): Раздел 11.11.2 [Опции Гнезда].
SO_STYLE
`sys/socket.h' (GNU): Раздел 11.11.2 [Опции Гнезда].
SO_TYPE
`sys/socket.h' (BSD): Раздел 11.11.2 [Опции Гнезда].
speed_t
`termios.h' (POSIX.1): Раздел 12.4.8 [Скорость Строки].
int sprintf (char *s, const char *template, . . .)
`stdio.h' (ANSI): Раздел 7.9.7 [Функции Форматированного Вывода].
double sqrt (double x)
`math.h' (ANSI): Раздел 13.4 [Экспоненты и Логарифмы].
void srand (unsigned int seed)
`stdlib.h' (ANSI): Раздел 13.6.1 [ANSI Random].
void srandom (unsigned int seed)
`stdlib.h' (BSD): Раздел 13.6.2 [BSD Random].
int sscanf (const char *s, const char *template, . . .)
`stdio.h' (ANSI): Раздел 7.11.8 [Функции Форматированного Ввода].
sighandler_t ssignal (int signum, sighandler_t action)
`signal.h' (SVID): Раздел 21.3.1 [Основная Обработка Сигнала].
int SSIZE_MAX
`limits.h' (POSIX.1): Раздел 27.1 [Основные Ограничения].
ssize_t
`unistd.h' (POSIX.1): Раздел 8.2 [Прмитивы Ввода/Вывода].
int stat (const char *filename, struct stat *buf)
`sys/stat.h' (POSIX.1): Раздел 9.8.2 [Чтение Атрибутов].
STDERR_FILENO
`unistd.h' (POSIX.1): Раздел 8.4 [Описатели и Потоки].
FILE * stderr
`stdio.h' (ANSI): Раздел 7.2 [Стандартные Потоки].
STDIN_FILENO
`unistd.h' (POSIX.1): Раздел 8.4 [Описатели и Потоки].
FILE * stdin
`stdio.h' (ANSI): Раздел 7.2 [Стандартные Потоки].
STDOUT_FILENO

`unistd.h' (POSIX.1): Раздел 8.4 [Описатели и Потоки].
FILE * stdout
`stdio.h' (ANSI): Раздел 7.2 [Стандартные Потоки].
char * stpcpy (char *to, const char *from)
`string.h' (Unknown origin): Раздел 5.4 [Копирование и Конкатенация].
int strcasecmp (const char *s1, const char *s2)
`string.h' (BSD): Раздел 5.5 [Сравнение Строки/Массива].
char * strcat (char *to, const char *from)
`string.h' (ANSI): Раздел 5.4 [Копирование и Конкатенация].

- 653 -

- 654 -

`stdio.h' (ANSI): Раздел 7.9.9 [Вывод с Переменными Аргументами].
int VQUIT
`termios.h' (POSIX.1): Раздел 12.4.9.3 [Символы Сигнала].
int VREPRINT
`termios.h' (BSD): Раздел 12.4.9.2 [BSD Редактирование].
int vscanf (const char *template, va_list ap)
`stdio.h' (GNU): Раздел 7.11.9 [Ввод с Переменными Аргументами].
int vsnprintf (char *s, size_t size, const char *template, va_list ap)
`stdio.h' (GNU): Раздел 7.9.9 [Вывод с Переменными Аргументами].
int vsprintf (char *s, const char *template, va_list ap)
`stdio.h' (ANSI): Раздел 7.9.9 [Вывод с Переменными Аргументами].
int vsscanf (const char *s, const char *template, va_list ap)
`stdio.h' (GNU): Раздел 7.11.9 [Ввод с Переменными Аргументами].
int VSTART
`termios.h' (POSIX.1): Раздел 12.4.9.4 [Символы Начала/Останова].
int VSTATUS

`termios.h' (BSD): Раздел 12.4.9.5 [Другой Выбор].
int VSTOP
`termios.h' (POSIX.1): Раздел 12.4.9.4 [Символы Начала/Останова].
int VSUSP
`termios.h' (POSIX.1): Раздел 12.4.9.3 [Символы Сигнала].
int VTIME
`termios.h' (POSIX.1): Раздел 12.4.10 [Неканонический Ввод].
int VWERASE
`termios.h' (BSD): Раздел 12.4.9.2 [BSD Редактирование].
pid_t wait3 (union wait *status`ptr, int options, struct rusage *usage)
`sys/wait.h' (BSD): Раздел 23.8 [BSD Функции Ожидания].
pid_t wait4 (pid_t pid, union wait *status`ptr, int options,
struct rusage *usage)
`sys/wait.h' (BSD): Раздел 23.8 [BSD Функции Ожидания].
pid_t wait (int *status`ptr)
`sys/wait.h' (POSIX.1): Раздел 23.6 [Завершение Процесса].
pid_t waitpid (pid_t pid, int *status`ptr, int options)
`sys/wait.h' (POSIX.1): Раздел 23.6 [Завершение Процесса].
WCHAR_MAX
`limits.h' (GNU): Раздел A.5.2 [Диапазон Типа].
wchar_t
`stddef.h' (ANSI): Раздел 18.4 [Введение в Расширенные Символы].
int WCOREDUMP (int status)
`sys/wait.h' (BSD): Раздел 23.7 [Статус Завершения Процесса].
size_t wcstombs (char *string, const wchar_t wstring, size_t size)
`stdlib.h' (ANSI): Раздел 18.5 [Расширенное Строковое Преобразование].
int wctomb (char *string, wchar_t wchar)
`stdlib.h' (ANSI): Раздел 18.7 [Преобразование Одного Символа].
int WEXITSTATUS (int status)
`sys/wait.h' (POSIX.1): Раздел 23.7 [Статус Завершения Процесса].
int WIFEXITED (int status)
`sys/wait.h' (POSIX.1): Раздел 23.7 [Статус Завершения Процесса].
int WIFSIGNALED (int status)
`sys/wait.h' (POSIX.1): Раздел 23.7 [Статус Завершения Процесса].
int WIFSTOPPED (int status)
`sys/wait.h' (POSIX.1): Раздел 23.7 [Статус Завершения Процесса].
int W_OK
`unistd.h' (POSIX.1): Раздел 9.8.8 [Прверка Прав Файла].

int wordexp (const char *words, wordexp_t *word-vector-ptr, int flags)
`wordexp.h' (POSIX.2): Раздел 16.4.2 [Вызов Wordexp].
wordexp_t
`wordexp.h' (POSIX.2): Раздел 16.4.2 [Вызов Wordexp].
void wordfree (wordexp_t *word-vector-ptr)
`wordexp.h' (POSIX.2): Раздел 16.4.2 [Вызов Wordexp].
WRDE_APPEND
`wordexp.h' (POSIX.2): Раздел 16.4.3 [Флаги для Wordexp].
WRDE_BADCHAR
`wordexp.h' (POSIX.2): Раздел 16.4.2 [Вызов Wordexp].
WRDE_BADVAL
`wordexp.h' (POSIX.2): Раздел 16.4.2 [Вызов Wordexp].
WRDE_CMDSUB
`wordexp.h' (POSIX.2): Раздел 16.4.2 [Вызов Wordexp].

WRDE_DOOFFS
`wordexp.h' (POSIX.2): Раздел 16.4.3 [Флаги для Wordexp].
WRDE_NOCMD
`wordexp.h' (POSIX.2): Раздел 16.4.3 [Флаги для Wordexp].
WRDE_NOSPACE
`wordexp.h' (POSIX.2): Раздел 16.4.2 [Вызов Wordexp].
WRDE_REUSE
`wordexp.h' (POSIX.2): Раздел 16.4.3 [Флаги для Wordexp].
WRDE_SHOWERR
`wordexp.h' (POSIX.2): Раздел 16.4.3 [Флаги для Wordexp].
WRDE_SYNTAX
`wordexp.h' (POSIX.2): Раздел 16.4.2 [Вызов Wordexp].
WRDE_UNDEF
`wordexp.h' (POSIX.2): Раздел 16.4.3 [Флаги для Wordexp].
ssize_t write (int filedes, const void *buffer, size_t size)
`unistd.h' (POSIX.1): Раздел 8.2 [Прмитивы Ввода/Вывода].
int WSTOPSIG (int status)
`sys/wait.h' (POSIX.1): Раздел 23.7 [Статус Завершения Процесса].
int WTERMSIG (int status)
`sys/wait.h' (POSIX.1): Раздел 23.7 [Статус Завершения Процесса].
int X_OK
`unistd.h' (POSIX.1): Раздел 9.8.8 [Прверка Прав Файла].

Приложение C: Сопровождение Библиотеки

C.1 Как Установить Библиотеку GNU C

Установка библиотеки GNU C относительно проста.

Вы нуждаетесь в последней версии GNU MAKE. Изменение Библиотеки GNU C, чтобы работать с другими программами make было бы настолько трудно, что мы рекомендуем Вам взамен перенести GNU MAKE.

Чтобы конфигурировать библиотеку GNU C для вашей системы, выполните команду оболочки "configure" с sh. Используйте аргумент, который является стандартным именем GNU для конфигурации вашей системы например "sparc-sun-sunos4.1 " для Sun 4 выполняющего Sunos 4.1. См. раздел " Установка GNU CC " в Руководстве по Использованию и Перенесению GNU CC, для полного описания стандартных имен конфигурации GNU. Если Вы опускаете имя конфигурации,"cofigure" пробует предположить его, анализируя систему, на которой она выполняется. Она может и не быть способна придумать предположение, и предположение может быть неправильно. "configure" сообщит Вам канонической имя относительно выбранной конфигурации перед продолжением.

Библиотека GNU C в настоящее время поддерживает конфигурации, которые соответствуют следующим шаблонам:

```
alpha-dec-osf1
i386-anything-bsd4.3
i386-anything-gnu
i386-anything-sco3.2
i386-anything-sco3.2v4
i386-anything-sysv
i386-anything-sysv4
i386-sequent-bsd
m68k-hp-bsd4.3
m68k-sony-newsos
m68k-sun-sunos4.n
mips-dec-ultrix4.n
sparc-sun-solaris2.n
sparc-sun-sunos4.n
```

В то время как никакие другие конфигурации не обеспечиваются, имеются удобные побочные результаты исследования для этих немногих. (Эти побочные результаты исследования работают в другом программном обеспечении GNU

также.)

```
decstation
hp320-bsd4.3 hp300bsd
i386-sco
i386-sco3.2v4
i386-sequent-dynix
i386-svr4
news
```

```
                sun3-sunos4.n sun3
                sun4-solaris2.n sun4-sunos5.n
                sun4-sunos4.n sun4
```

Имеются некоторые опции, которые Вы должны определить когда Вы выполняете выбор конфигурации:

`--with-gnu-ld'

Используйте эту опцию, если Вы планируете использовать GNU ld, чтобы линковать программы с Библиотекой GNU C. (Мы строго рекомендуем, чтобы Вы ее использовали.)

`--with-gnu-as'

Используйте эту опцию, если Вы планируете использовать GNU ассемблер, gas, при формировании Библиотеки GNU C. На некоторых системах, библиотека не может формироваться правильно, если Вы не используете gas.

`--nfp'

Используйте эту опцию, если ваш компьютер испытывает недостаток аппаратной поддержки с плавающей запятой.

`--prefix=directory'

Устанавливает машинно-независимые файлы данных в подкаталоге "directory". (Вы можете также установить это в " configparms'; см. ниже.)

`--exec-prefix=directory'

Устанавливает библиотеку и другие машинно-зависимые файлы в подкаталогах "directory". (Вы можете также установить это в " configparms'; см. ниже.)

Самый простой способ выполнять выбор конфигурации состоит в том, чтобы делать это в каталоге, который содержит библиотечные источники (исходники).

Вы можете формировать библиотеку в некотором другом каталоге, перейдя в этот другой каталог, чтобы выполнить configure. В общем, чтобы выполнить выбор конфигурации, Вы должны будете определить каталог для этого, примерно так:


```
mkdir   ../hp320
cd ../hp320
../src/configure hp320-bsd4.3
```

configure ищет источники в любом каталоге, который Вы определили для configure непосредственно.

Эта возможность позволяет Вам, хранить исходники и bin-файлы в различных каталогах, и это облегчает формирование библиотеки для отдельных различных машин из того же самого набора исходников. Просто создайте каталог формирования для каждой целевой машины, и выполните configure в этом каталоге, определяя имя конфигурации целевой машины.

Библиотека имеет ряд параметров конфигурации специального назначения. Они определены в файле " Makeconfig "; см. комментарии в этом файле для подробностей.

Но не редактируйте файл " Makeconfig " непосредственно, взамен, создайте файл " configparms' в каталоге, где Вы формируете библиотеку, и определяете в этом файле параметры, которые Вы хотите определить. "Configparms" не должен быть отредактированной копией " Makeconfig "; определите только параметры, которые Вы хотите отменить!

Некоторые из машинно-зависимых программ (часть их кода) для некоторых машин используют расширения в компиляторе GNU C, так что Вы возможно будете должны компилировать библиотеку с GCC. (Фактически, все существующие полные перенесенные версии требуют GCC.)

Текущая реализация библиотеки C содержит некоторые заглавные файлы, которые компилятор обычно обеспечивает: " stddef.h ", " stdarg.h ", и отдельные файлы с именами вида " va-machine.h ". Версии этих файлов, которые пришли с более старыми выпусками GCC, не работают правильно с библиотекой GNU C. "Stddef.h " файл в выпуске 2.2 и позже GCC правилен. Если Вы имеете выпуск 2.2 или позднее GCC, используйте версию " stddef.h " вместо версии библиотеки C. Чтобы сделать это, поместите строку " override stddef.h = " в " configparms ". Другие файлы исправлены в выпуске 2.3 и позже. "Configure" автоматически обнаружит совместимы ли установленные " stdarg.h " и " va-machine.h " файлы с библиотекой C, и использует собственные иначе.

Имеется потенциальная проблема с size_t типом и версиями GCC до выпуска 2.4. ANSI C требует, чтобы size_t всегда был тип без знака. Для совместимости с заглавными файлами существующих систем GCC определяет size_t в " stddef.h " так чтобы он мог быть любым типом, который определит " sys/types.h ".

Большинство систем UNIX, которые определяют size_t в " sys/types.h ",

определяют его как тип со знаком. Некоторый код в библиотеке зависит от того что size_t, - тип без знака, и не будет работать правильно, если он - со знаком.

Код библиотеки GNU C, который ожидает, что size_t будет без знака, правилен. Определение size_t как типа со знаком неправильно. Версии GCC 2.4 и позже всегда определяют size_t как тип без знака.

Тем временем, мы работаем вокруг этой проблемы, сообщая GCC явно, чтобы использовать тип без знака для size_t при компилировании библиотеки GNU C. " Configure " автоматически обнаружит то, какой тип GCC использует для size_t, чтобы отменить это в случае необходимости.

Чтобы сформировать библиотеку напечатайте make lib. Это произведет множество вывода, похожего на ошибки make (но не ошибки). Ищите сообщения об ошибках make, содержащие " *** ". Они указывают, что кое-что - действительно неправильно.

Чтобы сформировать и выполнить некоторые тестовые программы, которые осуществляют некоторые из библиотечных средств, напечатайте make test. Это произведет несколько файлов с именами подобно "program.out".

Чтобы форматировать Справочное Описание Библиотеки GNU C для печати, наберите make dvi. Для форматирования Инфо версии руководства для чтении в C-hi в Emacs или программой info, наберите make info.

Чтобы установить библиотеку, заглавные файлы, и файлы Информации руководства, наберите make install, после установки каталогов в "configparms".

C.2 Как Сообщить об Ошибках

Возможно в библиотеке GNU C имеются ошибки. Имеются конечно ошибки и пропуски в этом руководстве. Если Вы сообщите их, они станут фиксированными. Если Вы не сделаете этого, никто не будет когда-либо знать относительно них, и они останутся нефиксированными на всю вечность, если не дольше.

Чтобы сообщить ошибку, сначала Вы должны найти ее. Обнадеживает то, что это будет самая трудная часть. Если только вы нашли ошибку, убедитесь что это - действительно ошибка. Хороший способ сделать это состоит в том, чтобы посмотреть, ведет ли библиотека GNU C себя так же, как некоторая другая библиотека C. Если так, возможно Вы неправы, а библиотека права (но не обязательно). Если нет, одна из библиотек - возможно неправа.

Если Вы уверены, что Вы нашли ошибку, попробуете свести ее к самому маленькому тесту, который воспроизводит проблему.

Заключительный шаг, когда Вы имеете простой тест, - сообщить ошибку.

При сообщении ошибки, пошлите ваш тест, результаты, которые Вы получили, результаты, которые Вы ожидали, что Вы думаете об этой проблеме (если вы думали о чем - нибудь), тип вашей системы, и версию библиотеки GNU C, которую Вы используете. Также включите файлы "config.status" и "config.make" которые созданы, при выполнении "configure"; они будут в том каталоге, который был текущий, когда Вы выполнили "configure".

Если Вы думаете, что Вы нашли некоторый способ, которым библиотека GNU C не соответствует ANSI и POSIX стандартам (см. Раздел 1.2 [Стандарты и Переносимость]. Сообщите это!

Пошлите отчет об ошибке по адресу Internet "bug-glibc@prep.ai.mit.edu" или через UUCP "mit-eddie!prep.ai.mit.edu!bug-glibc". Если Вы имеете другие проблемы с установкой или использованием, пожалуйста, сообщайте их также.

Если Вы не уверены, как должна вести себя функция, и это руководство не сообщает Вам, это - ошибка в руководстве. Сообщите это! Если поведение функции не сходится с руководством, то либо библиотека либо руководство имеет ошибку. Если Вы нашли любые ошибки или пропуски в этом руководстве, пожалуйста, сообщите их по адресу Internet "bug-glibc-manual@prep.ai.mit.edu
" или через UUCP "mit-eddie!prep.ai.mit.edu!bug-glibc-manual ".

C.3 Добавление Новых Функций

Процесс формирования библиотеки управляется make-файлами, которые используют специальные возможности GNU MAKE. Make-файлы очень сложны, и Вы возможно не хотите попробовать понять их. Но то, что они делают довольно просто, и требует только, чтобы Вы определили несколько переменных в нужных местах.

Библиотечные исходники разделены на подкаталоги, сгруппированные по темам. Подкаталог ` string ' содержит все функции строкового манипулирования, " stdio " - все стандартные функции ввода - вывода, и т.д.

Каждый подкаталог содержит простой make-файл, называемый "Makefile", который определяет несколько переменных и тогда включает глобальный make-файл "Rules" строкой подобно:

```
include ../Rules
```

Базисные переменные, которые определяет make-файл подкаталога:

subdir

Имя подкаталога, например " stdio ". Эта переменная должна быть определена.

headers

Имена заглавных файлов в этом разделе библиотеки, типа " stdio.h ".

routines

aux

Имена модулей (исходных файлов) в этом разделе библиотеки. Это должны быть простые имена, типа " strlen " (а не полные имена файлов, типа " strlen.c "). Используйте routines для модулей, которые определяют функции в библиотеке, и aux для дополнительных модулей, содержащих вещи подобно определениям данных. Но значения routines и aux только что конкатенируются, так что действительно не имеется никакого практического различия.

tests

Имена тестовых программ для этого раздела библиотеки.

others

Имена "других" программ, связанных с этим разделом библиотеки. Это - программы, которые - не тесты по существу, но - другие малые программы, включенные в библиотеку. Они формируются " make others'.

install-lib

install-data

install

Файлы, которые будут установлены " make install ". Файлы, перечисленные в " install-lib " устанавливаются в каталог, заданный " libdir " в "configparms" или "Makeconfig" (см. Раздел C.1 [Установка]). Файлы, перечисленные в install-data будут установлены в каталог, заданный " datadir " в "configparms" или "Makeconfig". Файлы, перечисленные в install будут установлены в каталог, заданный " bindir " в "configparms" или "Makeconfig".

distribute

Другие файлы из этого подкаталога, которые должны быть помещены в дистрибутивный файл tar. Вы не должны перечислить здесь make-файл непосредственно или исходник и заглавные файлы, перечисленные в других стандартных переменных. Определите distribute только, если имеются файлы, используемые необычным способом, которые должны войти в распределение.

generated

Файлы, которые сгенерированы " Makefile " в этом подкаталоге. Эти файлы будут удалены " make clean ", и они никогда не будут входить в распределение.

extra-objs

Дополнительные объектные файлы, которые сформированы " Makefile " в этом подкаталоге. Это должен быть список имен файлов подобно " foo .o "; файлы будут фактически найдены в любых объектных файлах каталога. Эти

файлы будут удалены " make clean ". Эта переменная используется для вторичных объектных файлов, нужных для формирования других или тестов.

C.4 Перенесение библиотеки GNU C

Библиотека GNU C написана так чтобы быть легко переносимой на ряд машин и операционных систем. Машинно- и системо- зависимые функции отделены, чтобы было проще добавить реализации для новых машин или операционных систем. Этот раздел описывает размещение библиотечного исходного дерева и объясняет механизмы, используемые для выбора используемого машинно-зависимого кода.

Все машинно-зависимые и системо-зависимые файлы в библиотеке находятся в подкаталоге " sysdeps ". Этот каталог содержит иерархию подкаталогов (см. Раздел C.4.1 [Соглашения Иерархии]).

Каждый подкаталог " sysdeps " содержит исходные файлы для специфической машины или операционной системы, или для класса машин или операционных систем (например, все машины, которые используют ИИЭР 754 формат с плавающей запятой). Конфигурация определяет упорядоченный список этих подкаталогов. Каждый подкаталог неявно конкатенирует директорию предыдущего уровня к списку. Например, определение списка "unix/bsd/vax" является эквивалентным определению списка "unix/bsd/vax unix/bsd unix". Подкаталог может также определять, что он подразумевает другие подкаталоги, которые - непосредственно не выше него в иерархии каталогов. Если файл "Implies" существует в подкаталоге, он перечисляет другие подкаталоги "sysdeps", которые конкатенированы к списку, появляясь после подкаталога, содержащего "Implies" файл. Строки в "Implies" файле, которые начинаются с символа `#', игнорируются как комментарии. Например, "unix/bsd/Implies" содержит:

```
# BSD has Internet-related things.
```

unix/inet
и "unix/Implies" содержит:
posix
Так что конечный список - " unix/bsd/vax unix/bsd unix/inet unix posix ".
"Sysdeps" имеет два "специальных" подкаталога, называемые "generic" и "stub". Эти два всегда неявно конкатенируются к списку подкаталогов (в этом порядке), так что Вы не должны помещать их в " Implies' файл, и Вы не должны создвать никаких подкаталогов под ними. "Generic" нужен для вещей, которые могут быть выполнены на Машинно-независимом C, используя только другие машинно-независимые функции в библиотеке C. "Stub" нужен для stub-версий

функций, которые не могут быть выполнены на специфической машине или операционной системе. Функции stub всегда возвращают ошибку, и устанавливают errno как ENOSYS (Функция, не выполнена). См. Главу 2 [Сообщения об Ошибках].
Исходный файл, как известно, является зависимым от системы наличием версии в " generic " или " stub "; каждая зависимая от системы функция должна иметь или generic или stub реализацию (нет никакого смысла в наличии обеих).
Если Вы натолкнетесь на файл который находится в одном из основных исходных каталогов (" string ", " stdio ", и т.д.), и Вы хотите к написать машинно- или системо- зависимую его версию, переместите файл в " sysdeps/generic " и запишите вашу новую реализацию в соответствующем системно-специфическом подкаталоге. Обратите внимание, что, если файл должен быть зависимым от системы, он не должен появляться в одном из основных каталогов.
Имеются несколько специальных файлов, которые могут существовать в каждом подкаталоге " sysdeps ":
`Makefile'
Make-файл для этой машины или операционной системы, или класса машины или операционных систем. Этот файл включен библиотечным make-файлом "Makerules", который используется make-файлом верхнего уровня и make-файлами подкаталога.
Каждый make-файл в подкаталоге в упорядоченном списке подкаталогов. Так как могут быть включены несколько зависимых от системы make-файлов, каждый должен быть конкатенирован к "sysdep-routines" а не просто установлен:
sysdep-routines := $(sysdep-routines) foo bar
`Subdirs'
Этот файл содержит имена новых подкаталогов под библиотечным исходным деревом верхнего уровня, которые должны быть включены для этой системы. Эти подкаталоги обрабатываются точно так же как независимые от системы подкаталоги в библиотечном исходном дереве, типа " stdio " и " math ".
Используйте его, когда имеются полностью новые наборы функций и заглавных файлов, которые должны войти в библиотеку. Например, "sysdeps/unix/inet/Subdirs" содержит "inet"; "inet" каталог содержит различные ориентируемые сетью операции, которые имеет смысл помещать в библиотеку на системах, которые поддерживают Internet.
`Dist'
Этот файл содержит имена файлов (относительно подкаталога " sysdeps "

в котором он появляется) которые должны быть включены в распределение. Перечислите любые новые файлы, используемые в " Makefile " в том же самом каталоге, или заглавные файлы, используемые исходными файлами в этом каталоге. Вы не должны перечислять файлы, чьи имена даны в машинно-независимых make-файлах в основном исходном дереве.
`configure'
Этот файл - часть команды оболочки, которая будет выполнена во время конфигурации. " configure " команда использует команду оболочки, чтобы читать " configure " файл в каждом зависимом от системы выбранном каталоге. " Configure" файлы часто сгенерирован из " configure.in " файлов, используюя Autoconf.
Для опции " - with-package=value " " configure " устанавливает переменную оболочки " with_package "; если опция - только " - with-package' (никакого аргумента), то она устанавливает " with_package" как "yes".
`configure.in'
Этот файл - фрагмент ввода Autoconf, который будет обработан в файл "configure" в этом подкаталоге. См. раздел "Введение" в Autoconf: Производство Автоматической Конфигурации, для описания Autoconf. Вы должны записать либо " configure " либо " configure .in ", но не обе. Первая строка "configure .in" должна вызвать m4 макркоманду "GLIBC_PROVIDES". Эта макркоманда делает несколько AC_PROVIDE запросов Autoconf макркоманд, которые используются

командой верхнего уровня " configure "; без этого, эти макркоманды могли бы вызываться снова необязательно Autoconf.

Это - общая схема того, как изолированы зависимости системы. Следующий раздел объясняет, как решить какие каталоги в " sysdeps " использовать. Раздел C.4.2 [Перенос на UNIX], имеет некоторые советы относительно переноса библиотеки на варианты UNIX.

C.4.1 Иерархия " sysdeps " Размещения Каталогов

Имя конфигурации GNU имеет три части: CPU тип, имя изготовителя, и операционная система. " Configure " использует их, чтобы выбрать список зависимых от системы каталогов. Если " - nfp " опция не передана к "configure", каталог " machine/fpu " также используется. Операционная система часто имеет основную операционную систему; например, если операционная система - "sunos4.1", основная операционная система - "unix/bsd". Алгоритм, используемый, чтобы выбрать список каталогов прост: " configure " делает список из основной операционной системы, изготовителя, типа CPU, и операционной системой, в этом порядке. И затем конкатенирует их все вместе с наклонными чертами

вправо между ними, чтобы произвести имя каталога; например, конфигурация " sparc-sun-sunos4.1" соответствует " unix/bsd/sun/sparc/sunos4.1 ". Затем " configure " пробует удалять каждый элемент списка по очереди, так что " unix/bsd/sparc " и " sun/sparc " также пробуются, среди других. Так как точный номер версии операционной системы - часто не важен, и было бы очень неудобно, для примера, иметь идентичные "sunos4.1.1" и "sunos4.1.2" каталоги, ", configure " пробует менее специфические имена операционной системы, удаляя конечные суфиксы, начинающиеся с точки.

Например, вот полный список каталогов, которых бы добивалась конфигурация "sparc-sun-sunos4.1" (без опции "- nfp"):

```
sparc/fpu
unix/bsd/sun/sunos4.1/sparc
unix/bsd/sun/sunos4.1
unix/bsd/sun/sunos4/sparc
unix/bsd/sun/sunos4
unix/bsd/sun/sunos/sparc
unix/bsd/sun/sunos
unix/bsd/sun/sparc
unix/bsd/sun
unix/bsd/sunos4.1/sparc
unix/bsd/sunos4.1
unix/bsd/sunos4/sparc
unix/bsd/sunos4
unix/bsd/sunos/sparc
unix/bsd/sunos
unix/bsd/sparc
unix/bsd
unix/sun/sunos4.1/sparc
unix/sun/sunos4.1
unix/sun/sunos4/sparc
unix/sun/sunos4
unix/sun/sunos/sparc
unix/sun/sunos
unix/sun/sparc
unix/sun
unix/sunos4.1/sparc
unix/sunos4.1
unix/sunos4/sparc
```



```
unix/sunos4
unix/sunos/sparc
unix/sunos
unix/sparc
unix
sun/sunos4.1/sparc
sun/sunos4.1
sun/sunos4/sparc
sun/sunos4
sun/sunos/sparc
sun/sunos
sun/sparc
sun
sunos4.1/sparc
```

```
                sunos4.1
                sunos4/sparc
                sunos4
                sunos/sparc
                sunos
                sparc
```

Различные архитектуры - традиционно подкаталоги верхнего уровеня "sysdeps" дерева каталогов. Например, " sysdeps/sparc" и " sysdeps/m68k ". Они содержат файлы, специфические для этой машинной архитектуры.

Имеются несколько каталогов на верхнем уровне " sysdeps " иерархии, которые не архитектуры.

" generic "

" stub "

Как описано выше (см. Раздел C.4 [Перенесение]), это - два подкаталога, которые каждая конфигурация неявно использует после всех остальных.

" ieee754 "

Этот каталог - для кода, использующего ИИЭР 754 формат с плавающей запятой, где float тип является ИИЭР 754 форматом с одинарной точностью, и double - ИИЭР 754 форматом двойной точности. Обычно этот каталог упоминается в " Implies" файле в архитектуро-определенном каталоге, типа "m68k/Implies".

" posix "

Этот каталог содержит реализации некоторых вещей в библиотеке в терминах POSIX.1 функций. Он включает некоторые из POSIX.1 функций,

непосредственно. Конечно, POSIX.1 не может быть полностью выполнен в терминах себя, так что конфигурация, использующая только " posix " не может быть полна.

" UNIX "

Это - каталог для unix-подобных вещей. См. Раздел C. 4.2 [Перенос на UNIX].

" UNIX " подразумевает " posix ". Имеются некоторые подкаталоги специального назначения " UNIX ":

" pnix/common "

Этот каталог - для вещей, общих для System V (4) и BSD.

И " unix/bsd " и " unix/sysv/sysv4 " подразумевают " unix/common ".

" pnix/inet "

Этот каталог - для функций гнезда и зависимых в системах UNIX.

" inet " подкаталог верхнего уровня допускается "unix/inet/subdirs" . "unix/common" подразумевает "unix/inet".

" mach "

Это - каталог для вещей, основанных на микроядре Mach из CMU (включая операционную систему GNU). Другие базисные операционные системы (VMS, например) имели бы собственные каталоги верхнего уровня " sysdeps " иерархии, параллельные " UNIX " и " mach ".

C.4.2 Перенесение Библиотеки GNU C в Системы UNIX

Большинство систем UNIX существенно похожи. Имеются небоьшие различия между различными машинами, и средствами обеспечиваемыми ядром. Но интерфейс для средств операционной системы, довольно однородный и простой.

Код для систем UNIX находится в каталоге " unix ", на верхнем уровне "sysdeps" иерархии. Этот каталог содержит подкаталоги (и деревья подкаталогов) для различных вариантов UNIX.

Функции, которые являются системными вызовами в большинстве систем UNIX, выполнены в ассемблерном коде в файлах в " sysdeps/unix ". Эти файлы именованы с суффиксом " .S "; например, " __ open.S".

Эти файлы используют набор макркоманд, которые должны быть определены в " sysdep.h ". " Sysdep.h " файл в " sysdeps/unix " частично определяет их; " sysdep.h " файл в другом каталоге должен закончить их определение для специфической машины и варианта операционной системы. См. " sysdeps/unix/sysdep.h " и машинно-специфическую " sysdep.h " реализацию.

Системно-специфический make-файл для каталога ` unix ' (то есть файл "

sysdeps/unix/Makefile ") содержит правила генерации отдельных файлов в системе UNIX, на которой Вы формируете библиотеку (которая принята как целевая система). Все сгенерированные файлы будут помещены в каталог, где сохраняются объектные файлы; они не должны воздействовать на исходное дерево непосредственно. Будут сгенерированы файлы: "ioctls.h", "errnos.h", "sys/param.h", и "errlist.c" (для " stdio " раздела библиотеки).

C. 5 Исследователи и Создатели Библиотеки GNU C

Почти вся библиотека была написана Роландом Мак-Гратом (Roland McGrath), за исключением некоторых дополнений, которе мы не будем здесь обсуждать.

С О Д Е Р Ж А Н И Е

- 675 -

- 676 -