Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
Пермский национальный исследовательский политехнический университет

Электротехнический факультет
Кафедра « Информационные технологии и автоматизированные системы »

Взаимодействие процессов в многозадачной среде (по вариантам)

ЗАДАНИЯ ПО КУРСОВОМУ ПРОЕКТИРОВАНИЮ
по дисциплине

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Пермь, 2016

Оглавление

Общие положения....4
Цель курсовой работы....4
Задачи:....4
Задание на курсовую работу....4
Если вы хотите выбрать свою тему по курсовой работе....5
Варианты....5
Запросы....6
Оформление отчета....6
Введение....6
Заключение....6
Список используемой литературы или используемых источников....7
Приложения....7
Правила оформления курсовой работы....7
Рекомендации по выполнению....7
Ориентировочная последовательность действий....7
Факторы, влияющие на повышение балла....8
Теоретический материал....9
1. Семафоры....9
a. Операции над семаформами....10
2. Реализация взаимодействия процессов....12
3. Элементарные средства межпроцессорного взаимодействия....14
a. Сигналы....14
b. Каналы....17
c. Именованные каналы (FIFO)....22
4. Реализация нитей....23
a. Представление атрибутов нити....24
b. Создание нити....25
c. Завершение нити. Ожидание завершения нити....25
5. Сокеты....27
a. Типы сокетов....28
b. Коммуникационный домен....30
c. Создание и конфигурация сокета....31
i. Создание сокета....31
ii. Связывание сокета с адресом....31
d. Предварительное установление соединения....32
i. Сокеты с установлением соединения. Запрос на соединение....32

ii. Сервер: прослушивание сокета и подтверждение соединения.....33
e. Прием и передача данных.....35
f. Завершение работы с сокетом.....36
g. Общая схема работы с сокетами.....36

Общие положения

Курсовая работа – заключительный этап изучения дисциплины.

Курсовая работа основывается на обобщении выполненных студентом практических/лабораторных работ или представляет собой индивидуальное задание по изучаемой дисциплине и подготавливается к защите в завершающий период теоретического обучения.

Цель курсовой работы

Систематизация и закрепление теоретических знаний, полученных за время обучения, а также приобретение и закрепление навыков самостоятельной работы.

Задачи:

- углубление теоретических знаний в соответствии с заданной темой;
- применение студентами теоретических знаний и практических умений, полученных при изучении учебной дисциплины «Системное программное обеспечение»;
- закрепление у студентов навыков ведения самостоятельной работы, освоение методики теоретического, экспериментального и научно-практического исследования;
- развитие умений студентов работать с различными литературными источниками, анализировать, обобщать, делать выводы, составлять рекомендации, предложения;
- поощрение творческой инициативы, самостоятельности, ответственности и организованности студента;

Задание на курсовую работу

Необходимо разработать программу на Си из двух частей (клиент и сервер). Обмен между клиентом и сервером должен происходить с использованием одного из способов взаимодействия процессов (по варианту). При этом на каждое обращение клиента сервер должен обрабатывать в отдельном потоке (способ организации потока по вариантам).

Если вы хотите выбрать свою тему по курсовой работе

По согласованию с руководителем курсовой работы вы можете сами предложить тему курсовой работы. В работе должны рассматриваться взаимодействия процессов на системном уровне.

Варианты

Таблица 1 - Варианты заданий

Вариант	**ОС**	**Способ взаимодействия**	**Способ организации многозадачности** (потоков)	**Запрос** (расшифровка приведена в п. "Запросы")
1	linux	неименованные каналы	процессы	1
2	"	"	нити	1
3	"	именованные каналы	процессы	1
4	"	"	нити	1
5	"	сигналы	процессы	3
6	"	"	нити	3
7	"	сокеты TCP	процессы	1
8	"	"	нити	1
9	"	сокеты UDP	процессы	1
10	"	"	нити	1
11	"	неименованные каналы	процессы	2
12	"	"	нити	2
13	"	именованные каналы	процессы	2
14	"	"	нити	2
15	"	неименованные каналы	процессы	4
16	"	"	нити	4
17	"	сокеты TCP	процессы	1
18	"	"	нити	1
19	"	IPC механизм общей памяти	процессы	1
20	"	"	нити	1
21	win	неименованные каналы	процессы	1
22	"	"	нити	1
23	"	именованные каналы	процессы	1
24	"	"	нити	1
25	"	сокеты TCP	процессы	1

26	"	"	нити	1
27	"	сокеты UDP	процессы	1
28	"	"	нити	1
29	"	неименованные каналы	процессы	2
30	"	"	нити	2
31	"	именованные каналы	процессы	2
32	"	"	нити	2
33	"	неименованные каналы	процессы	4
34	"	"	нити	4
35	"	сокеты TCP	процессы	1
36	"	"	нити	1

Запросы

1) клиент должен передать серверу число, сервер должен передать клиенту сумму чисел, поступивших серверу, начиная от старта сервера;
2) сервер должен передать клиенту номер запроса клиента, начиная от старта сервера ;
3) сервер должен передать клиенту четность номера запроса клиента, начиная от старта сервера;
4) клиент должен передать серверу строку, сервер должен передать клиенту конкатенацию строк, поступивших серверу, начиная от старта сервера;

Оформление отчета

- титульный лист;
- оглавление (содержание);
- введение;
- основная часть (обычно состоит из двух разделов):
- первый раздел — теоретические основы разрабатываемой темы;
- второй раздел — практическая часть, представленная расчетами, графиками, таблицами, схемами, скринами и т.п.;
- заключение;
- список используемой литературы или используемых источников;
- приложения;

Введение

Необходимо обоснование актуальности темы работы, оно ограничено по объему несколькими абзацами и характеризует потенциальную пользу выбранной темы. Обязательны задачи и цели, выделяют одну-две цели и несколько задач. Цель - более глобальная категория, чем задача. В рамках одной цели, как правило, решается несколько задач, что удобно представлять в виде вложенного списка, задачи – шаги, по которым выполняется достижение цели. Приветствуется краткое содержание, что именно и как необходимо сделать для выполнения поставленных задач.

Заключение

Заключение отражает итог работы, выводы по вопросам, исследуемым в курсовой работе, содержит авторское мнение, преимущества и проблемы, раскрываемые в исследовании. В нём указываются основные мероприятия, проведённые в практической части работы. Также обязательно изложение проблем и хорошо обдуманные пути их решения. В заключение входят поставленные цели и анализируемые задачи из введения, а основные результаты вписываются в него из основной части работы.

Список используемой литературы или используемых источников

Оформление ссылок на печатные издания выглядит так:

Автор Название книги. Издательство, год. Количество стрвниц.

Пример: Орлов С. А. Программная инженерия. Технологии разработки программного обеспечения. СПб.: Питер, 2016. 640 с.

Оформление ссылок на интернет ресурсы:

Основное заглавие: расшифровка заглавия [Электронный ресурс]. URL: полная ссылка на документ (дата обращения: ДД.ММ.ГГ)

Пример: Язык программирования Perl: Лекция 16: Взаимодействие процессов [Электронный ресурс] URL: http://www.intuit.ru/studies/courses/2157/19/lecture/619 (дата обращения: 12.11.16)

Приложения

В этом разделе находятся: исходный код программ, большие таблицы и схемы.

Правила оформления курсовой работы

- Работа оформляется по ГОСТ 7.32;
- Объём курсовой работы – 15-25 страниц печатного текста;
- Формат А4;
- Шрифт Times New Roman;
- Шрифт для кода программ: Courier New;
- Размер 14, интервал 1,5;

Рекомендации по выполнению

Ориентировочная последовательность действий

Сервер

- принимает соединение от клиента (по варианту);
- запускает процесс обработки запроса (по варианту);
- процесс обработки запроса обрабатывает запрос;
- процесс обработки запроса передает клиенту ответ;

Клиент

- запрашивает у пользователя данные для запроса (по варианту);
- посылает запрос серверу;
- принимает ответ от сервера;
- ответ выдает ответ пользователю;

Факторы, влияющие на повышение балла

- блокировка критических секций семафором;
- проверка принимаемых/передаваемых значений;
- разработка структурного кода и сценария компиляции;
- обнаружение противоречий в задании, пруф прилагать;

Теоретический материал

1. Семафоры

Семафор представляет собой переменную целого типа, над которой определены две операции: down(P) и up(V). (P и V являются сокращениями голландских слов proberen – проверить и verhogen – увеличить. В англоязычной литературе общепринятыми являются также следующие названия операций над семафором: wait, test или lock (аналоги down) и post, unlock или signal (аналоги up)).

Операция down проверяет значение семафора, и если оно больше нуля, то уменьшает его на 1. Если же это не так, процесс блокируется, причем операция down считается незавершенной. Важно отметить, что вся операция является неделимой, т.е. проверка значения, его уменьшение и, возможно, блокирование процесса производятся как одно атомарное действие, которое не может быть прервано. Операция up увеличивает значение семафора на 1. При этом, если в системе присутствуют процессы, блокированные ранее при выполнении down на этом семафоре, ОС разблокирует один из них с тем, чтобы он завершил выполнение операции down, т.е. вновь уменьшил значение семафора. При этом также принято, что увеличение значения семафора и, возможно, разблокирование одного из процессов и уменьшение значения являются атомарной неделимой операцией.

Также семафоры представляют собой одну из форм IPC и, как правило, используются для синхронизации доступа нескольких процессов к разделяемым ресурсам, так как сами по себе другие средства IPC не предоставляют механизма синхронизации.

Как правило, использование семафоров в качестве средства синхронизации доступа к другим разделяемым объектам предполагает следующую схему:

- с каждым разделяемым ресурсом связывается один семафор из набора;
- положительное значение семафора означает возможность доступа к ресурсу (ресурс свободен), неположительное – отказ в доступе (ресурс занят);
- перед тем как обратиться к ресурсу, процесс уменьшает значение соответствующего ему семафора, при этом, если значение семафора после уменьшения должно оказаться отрицательным, то процесс будет заблокирован до тех пор, пока семафор не примет такое значение, чтобы при уменьшении его значение оставалось неотрицательным;
- закончив работу с ресурсом, процесс увеличивает значение семафора (при этом разблокируется один из ранее заблокированных процессов, ожидающих увеличения значения семафора, если таковые имеются);

- в случае реализации взаимного исключения используется двоичный семафор, т.е. такой, что он может принимать только значения 0 и 1: такой семафор всегда разрешает доступ к ресурсу не более чем одному процессу одновременно.

Для получения доступа к массиву семафоров (или его создания) используется системный вызов:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget (key_t key, int nsems, int semflag);
```

Первый параметр функции semget() – ключ для доступа к разделяемому ресурсу, второй - количество семафоров в создаваемом наборе (длина массива семафоров) и третий параметр – флаги, управляющие поведением вызова.

Отметим семантику прав доступа к такому типу разделяемых ресурсов, как семафоры: процесс, имеющий право доступа к массиву семафоров по чтению, может проверять значение семафоров; процесс, имеющий право доступа по записи, может как проверять, так и изменять значения семафоров.

В случае, если среди флагов указан IPC_CREAT, аргумент nsems должен представлять собой положительное число, если же этот флаг не указан, значение nsems игнорируется. Отметим, что в заголовочном файле <sys/sem.h> определена константа SEMMSL, задающая максимально возможное число семафоров в наборе. Если значение аргумента nsems больше этого значения, вызов semget() завершится неудачно.

В случае успеха вызов semget() возвращает положительный дескриптор созданного разделяемого ресурса, в случае неудачи -1.

a. Операции над семаформами

Используя полученный дескриптор, можно производить изменять значения одного или нескольких семафоров в наборе, а также проверять их значения на равенство нулю, для чего используется системный вызов semop():

```
#include <sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
int semop (int semid, struct sembuf *semop, size_t nops);
```

Этому вызову передаются следующие аргументы:

semid – дескриптор массива семафоров;

semop – массив из объектов типа struct sembuf, каждый из которых задает одну операцию над семафором;

nops – длина массива semop. Количество семафоров, над которыми процесс может одновременно производить операцию в одном вызове semop(),

ограничено константой SEMOPM, описанной в файле <sys/sem.h>. Если процесс попытается вызвать semop() с параметром nops, большим этого значения, этот вызов вернет неуспех.

Структура имеет sembuf вид:

```
struct sembuf {
short sem_num; /* номер семафора в векторе */
short sem_op; /* производимая операция */
short sem_flg; /* флаги операции */
}
```

Общий принцип обработки этой структуры приведен на Рисунке 2.

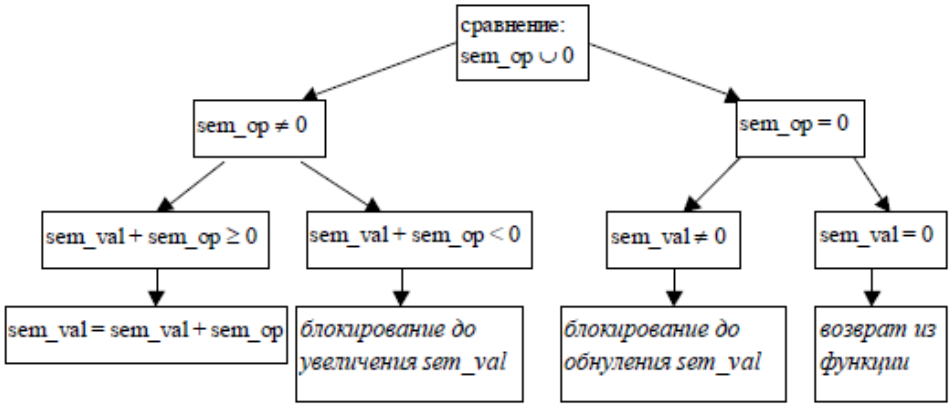

Рисунок 2 – Схема работы системного вызова semop()

Пусть значение семафора с номером sem_num равно sem_val.

1. если значение операции не равно нулю:

- оценивается значение суммы sem_val + sem_op;
- если эта сумма больше либо равна нулю, то значение данного семафора устанавливается равным этой сумме: sem_val = sem_val + sem_op ;
- если же эта сумма меньше нуля, то действие процесса будет приостановлено до тех пор, пока значение суммы sem_val + sem_op не станет больше либо равно нулю, после чего значение семафора устанавливается равным этой сумме: sem_val = sem_val + sem_op;

2. Если код операции sem_op равен нулю:

- если при этом значение семафора (sem_val) равно нулю, происходит немедленный возврат из вызова ;
- иначе происходит блокирование процесса до тех пор, пока значение семафора не обнулится, после чего происходит возврат из вызова;

Таким образом, ненулевое значение поля sem_op обозначает необходимость прибавить к текущему значению семафора значение sem_op, а нулевое – дождаться обнуления семафора.

Поле sem_flg в структуре sembuf содержит комбинацию флагов, влияющих на выполнение операции с семафором. В этом поле может быть

установлен флаг IPC_NOWAIT, который предписывает соответствующей операции над семафором не блокировать процесс, а сразу возвращать управление из вызова semop(). Вызов semop() в такой ситуации вернет –1. Кроме того, в этом поле может быть установлен флаг SEM_UNDO, в этом случае система запомнит изменение значения семафора, произведенные данным вызовом, и по завершении процесса автоматически ликвидирует это изменение. Это предохраняет от ситуации, когда процесс уменьшил значение семафора, начав работать с ресурсом, а потом, не увеличив значение семафора обратно, по какой-либо причине завершился. В этом случае остальные процессы, ждущие доступа к ресурсу, оказались бы заблокированы навечно.

2. Реализация взаимодействия процессов

Средства межпроцессного взаимодействия ОС UNIX позволяют строить прикладные системы различной топологии, функционирующие как в пределах одной локальной ЭВМ, так и в пределах сетей ЭВМ.

При рассмотрении любых средств межпроцессного взаимодействия возникает необходимость решения двух проблем, связанных с организацией взаимодействия процессов: проблемы именования взаимодействующих процессов и проблемы синхронизации процессов при организации взаимодействия. Рассмотрим их подробнее.

При организации любого взаимодействия мы сталкиваемся с необходимостью выбора пространства имен процессов-отправителей и получателей или имен некоторых объектов, через которые осуществляется взаимодействие. Эта проблема решается по-разному в зависимости от конкретного механизма взаимодействия. В системах взаимодействия процессов, функционирующих на различных компьютерах в рамках сети, используется адресация, принятая в конкретной сети ЭВМ (примером могут служить аппарат сокетов и MPI). При использовании средств взаимодействия процессов, локализованных в пределах одной ЭВМ, способ именования (а следовательно, и способ доступа к среде взаимодействия) зависит от конкретного механизма взаимодействия. В частности, для ОС UNIX механизмы взаимодействия процессов можно разделить на те средства, доступ к которым получают лишь родственные процессы, и средства, доступные произвольным процессам.

При взаимодействии родственных процессов проблема именования решается за счет наследования потомками некоторых свойств своих прародителей. Например, в случае неименованных каналов процесс-родитель для организации взаимодействия создает канал. Дескрипторы, ассоциированные с этим каналом, наследуются сыновними процессами, тем самым создается возможность организации симметричного (ибо все процессы изначально равноправны) взаимодействия родственных процессов. Другой пример – взаимодействие процессов по схеме главный-подчиненный

(трассировка процесса). Данный тип взаимодействия асимметричный: один из взаимодействующих процессов получает статус и права «главного», второй – «подчиненного». Главный – это родительский процесс, подчиненный – сыновний. В данном случае проблема именования снимается, так как идентификаторы процесса-сына и процесса-отца всегда доступны им обоим и однозначно определены.

При взаимодействии произвольных процессов использовать наследование тех свойств процессов, которые могут использоваться для именования, разумеется, невозможно. Поэтому в данном случае обычно используются две схемы: первая – использование для именования идентификаторов взаимодействующих процессов (например, при передаче сигналов); вторая – использование некоторого системного ресурса, обладающего уникальным именем. Примером последнего могут являться именованные каналы, использующие для организации взаимодействия процессов файлы специального типа (FIFO-файлы).

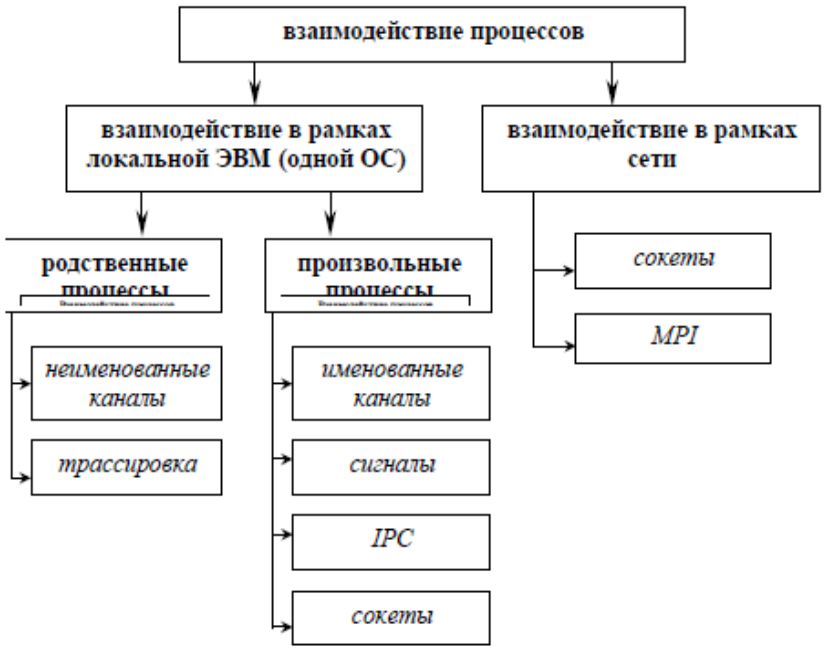

Рисунок 3 – Классификация средств взаимодействия процессов ОС UNIX

Другая проблема организации взаимодействия – это проблема синхронизации взаимодействующих процессов. Любое взаимодействие процессов представимо либо в виде оказания одним процессом воздействия на другой процесс, либо в виде использования некоторых разделяемых ресурсов, через которые возможна организация обмена данными. Первое требование к средствам взаимодействия процессов – это атомарность (неразделяемость) базовых операций. Синхронизация должна обеспечить атомарность операций взаимодействий или обмена данными с разделяемыми ресурсами. К примеру, система должна блокировать начало чтения данных из некоторого разделяемого ресурса до того, пока начавшаяся к этому моменту операция записи по этому ресурсу не завершится.

Второе требование – это обеспечение определенного порядка в операциях взаимодействия, или семантическая синхронизация. Например, некорректной является попытка чтения данных, которых еще нет (и операция записи которых еще не начиналась). В зависимости от конкретного механизма взаимодействия, уровней семантической синхронизации может быть достаточно много.

Комплексное решение проблемы синхронизации зависит от свойств используемых средств взаимодействия процессов. В некоторых случаях операционная система обеспечивает некоторые уровни синхронизации (например, при передаче сигналов, использовании каналов). В других случаях участие операционной системы в решении проблемы синхронизации минимально (например, при использовании разделяемой памяти IPC). В любом случае, конкретная прикладная система должна это учитывать и при необходимости обеспечивать семантическую синхронизацию процессов.

3. Элементарные средства межпроцессорного взаимодействия

a. Сигналы

Сигналы представляют собой средство уведомления процесса о наступлении некоторого события в системе. Инициатором посылки сигнала может выступать как другой процесс, так и сама ОС. Сигналы, посылаемые ОС, уведомляют о наступлении некоторых строго предопределенных ситуаций (как, например, завершение порожденного процесса, прерывание работы процесса нажатием комбинации Ctrl-C, попытка выполнить недопустимую машинную инструкцию, попытка недопустимой записи в канал и т.п.), при этом каждой такой ситуации сопоставлен свой сигнал. Кроме того, зарезервирован один или несколько номеров сигналов, семантика которых определяется пользовательскими процессами по своему усмотрению (например, процессы могут посылать друг другу сигналы с целью синхронизации).

Сигналы являются механизмом асинхронного взаимодействия, т.е. момент прихода сигнала процессу заранее неизвестен. Однако процесс может предвидеть возможность получения того или иного сигнала и установить определенную реакцию на его приход. В этом плане сигналы можно рассматривать как программный аналог аппаратных прерываний.

При получении сигнала процессом возможны три варианта реакции на полученный сигнал:

- процесс реагирует на сигнал стандартным образом, установленным по умолчанию (для большинства сигналов действие по умолчанию – это завершение процесса);
- процесс может заранее установить специальный способ обработки конкретного сигнала, в этом случае по приходу этого сигнала

вызывается функция-обработчик, определенная процессом (при этом говорят, что сигнал перехватывается);
- процесс может проигнорировать сигнал.

Для каждого сигнала процесс может устанавливать свой вариант реакции, например, некоторые сигналы он может игнорировать, некоторые перехватывать, а на остальные установить реакцию по умолчанию. При этом по ходу своей работы процесс может изменять вариант реакции на тот или иной сигнал. Однако, необходимо отметить, что некоторые сигналы невозможно ни перехватить, ни игнорировать. Они используются ядром ОС для управления работой процессов (например, SIGKILL, SIGSTOP).

Если в процесс одновременно доставляется несколько различных сигналов, то порядок их обработки не определен. Если же обработки ждут несколько экземпляров одного и того же сигнала, то ответ на вопрос, сколько экземпляров будет доставлено в процесс – все или один – зависит от конкретной реализации ОС.

В любом случае, в момент прихода сигнала, для которого задана пользовательская функция-обработчик, нормальное выполнение процесса прерывается, и управление переходит на точку входа обработчика сигнала. По выходу из функции-обработчика выполнение процесса возобновляется с той точки, на которой оно было прервано.

Для отправки сигнала служит системный вызов **kill()**:

```
#include <sys/types.h>
#include <signal.h>
int kill (pid_t pid, int sig)
```

Первым параметром вызова служит идентификатор процесса, которому посылается сигнал (в частности, процесс может послать сигнал самому себе). Существует также возможность одновременно послать сигнал нескольким процессам, например, если значение первого параметра есть 0, сигнал будет передан всем процессам, которые принадлежат той же группе, что и процесс, посылающий сигнал, за исключением процессов с идентификаторами 0 и 1.

Во втором параметре передается номер посылаемого сигнала. Если этот параметр равен 0, то будет выполнена проверка корректности обращения к kill() (в частности, существование процесса с идентификатором pid), но никакой сигнал в действительности посылаться не будет.

Если процесс-отправитель не обладает правами привилегированного пользователя, то он может отправить сигнал только тем процессам, у которых реальный или эффективный идентификатор владельца процесса совпадает с реальным или эффективным идентификатором владельца процесса-отправителя.

Для определения реакции на получение того или иного сигнала в процессе служит системный вызов **signal()**:

```
#include <signal.h>
void (*signal ( int sig, void (*disp) (int))) (int)
```

где аргумент sig — номер сигнала, для которого устанавливается реакция, а disp — либо определенная пользователем функция-обработчик сигнала, либо одна из констант: SIG_DFL и SIG_IGN. Первая из них указывает, что необходимо установить для данного сигнала обработку по умолчанию, т.е. стандартную реакцию системы, а вторая — что данный сигнал необходимо игнорировать. При успешном завершении функция возвращает указатель на предыдущий обработчик данного сигнала (он может использоваться процессом, например, для последующего восстановления прежней реакции на сигнал).

Как видно из прототипа вызова signal(), определенная пользователем функция-обработчик сигнала должна принимать один целочисленный аргумент (в нем будет передан номер обрабатываемого сигнала), и не возвращать никаких значений.

Механизм сигналов является достаточно ресурсоемким, ибо отправка сигнала представляет собой системный вызов, а доставка сигнала влечет за собой прерывание нормального порядка выполнения процесса-получателя. Вызов функции-обработчика и возврат требует операций со стеком. Сигналы также несут весьма ограниченную информацию.

Пример 1. Обработка сигнала.

В данном примере при получении сигнала SIGINT четырежды вызывается специальный обработчик, а в пятый раз происходит обработка по умолчанию.

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
int count = 1;
void SigHndlr (int s) /* обработчик сигнала */
{
printf("\n I got SIGINT %d time(s) \n",
++ count);
if (count == 5) signal (SIGINT, SIG_DFL);
/* ставим обработчик сигнала по умолчанию */
else signal (SIGINT, SigHndlr);
/* восстанавливаем обработчик сигнала */
}
int main(int argc, char **argv)
{
signal (SIGINT, SigHndlr); /* установка реакции на сигнал */
while (1); /*”тело программы” */
return 0;
   }
```

b. Каналы

Одним из простейших средств взаимодействия процессов в операционной системе UNIX является механизм каналов. Неименованный канал есть некая сущность, в которую можно помещать и извлекать данные, для чего служат два файловых дескриптора, ассоциированных с каналом: один для записи в канал, другой — для чтения. Для создания канала служит системный вызов **pipe()**:

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

Данный системный вызов выделяет в оперативной памяти некоторый буфер ограниченного размера и возвращает через параметр fd массив из двух файловых дескрипторов: один для записи в канал — fd[1], другой для чтения — fd[0].

Эти дескрипторы являются дескрипторами открытых файлов, с которыми можно работать, используя такие системные вызовы как read(), write(), dup() и так далее. Более того, эти дескрипторы, как и прочие дескрипторы открытых файлов, наследуются при порождении сыновнего процесса (что и позволяет использовать каналы как средство общения между процессами). Однако следует четко понимать различия между обычным файлом и каналом.

Основные отличительные свойства канала следующие:

- в отличие от файла, к неименованному каналу невозможен доступ по имени, т.е. единственная возможность использовать канал – это те файловые дескрипторы, которые с ним ассоциированы;
- канал не существует вне процесса, т.е. для существования канала необходим процесс, который его создаст и в котором он будет существовать, а после того, как будут закрыты все дескрипторы, ассоциированные с этим каналом, ОС автоматически освободит занимаемый им буфер. Для файла это, разумеется, не так;
- канал реализует модель последовательного доступа к данным (FIFO), т.е. данные из канала можно прочитать только в той же последовательности, в какой они были записаны. Это означает, что для файловых дескрипторов, ассоциированных с каналом, не определена операция позиционирования lseek() (при попытке обратиться к этому вызову произойдет ошибка).

Кроме того, существует ряд отличий в поведении операций чтения и записи в канал, а именно:

При чтении из канала:

- если прочитано меньше байтов, чем находится в канале, оставшиеся данные сохраняются в канале;
- если делается попытка прочитать больше данных, чем имеется в канале, и при этом существуют открытые дескрипторы записи,

ассоциированные с каналом, будет прочитано (т.е. изъято из канала) доступное количество данных, после чего читающий процесс блокируется до тех пор, пока в канале не появится достаточное количество данных для завершения операции чтения;

- процесс может избежать такого блокирования, изменив для канала режим блокировки с использованием системного вызова fcntl(). В неблокирующем режиме в ситуации, описанной выше, будет прочитано доступное количество данных, и управление будет сразу возвращено процессу;
- при закрытии записывающей стороны канала (т.е. закрытии всех дескрипторов записи, связанных с данным каналом), для него устанавливается признак конца файла (это можно представить себе так, что в канал «помещается символ EOF»(Надо понимать, что на самом деле EOF не является символом, а представляет собой константу, отличную от представления какого-либо символа в какой-либо кодировке, которая служит лишь для удобства возврата значений из читающих операций при достижении признака конца файла. Поэтому фактически, конечно, никакой «символ» EOF в канал (как и в обычный файл) никогда не помещается)). После этого процесс, осуществляющий чтение, может выбрать из канала все оставшиеся данные и получить признак конца файла, благодаря чему блокирования при чтении в этом случае не происходит.

При записи в канал:

- если процесс пытается записать большее число байтов, чем помещается в канал (но не превышающее предельный размер канала) записывается возможное количество данных, после чего процесс, осуществляющий запись, блокируется до тех пор, пока в канале не появится достаточное количество места для завершения операции записи;
- процесс может избежать такого блокирования, изменив для канала режим блокировки с использованием системного вызова **`fcntl()`**. В неблокирующем режиме в ситуации, описанной выше, запись в канал не производится (т.к. при частичной записи у ядра нет возможности обеспечить ее атомарность), и вызов **`write()`** возращает ошибку, устанавливая в переменной **`errno`** значение **`EAGAIN`**;
- если же процесс пытается записать в канал порцию данных, превышающую предельный размер канала, то будет записано доступное количество данных, после чего процесс заблокируется до появления в канале свободного места любого размера (пусть даже и всего в 1 байт), затем процесс разблокируется, вновь производит запись на доступное место в канале, и если данные для записи еще не исчерпаны, вновь блокируется до появления свободного места и

т.д., пока не будут записаны все данные, после чего происходит возврат из вызова **`write();`**

- если процесс пытается осуществить запись в канал, с которым не ассоциирован ни один дескриптор чтения, то он получает сигнал **`SIGPIPE`** (тем самым ОС уведомляет его о недопустимости такой операции)**.**

В стандартной ситуации (при отсутствии переполнения) система гарантирует атомарность операции записи, т. е. при одновременной записи нескольких процессов в канал их данные не перемешиваются.

Пример 2. Использование канала.

Пример использования канала в рамках одного процесса – копирование строк. Фактически осуществляется посылка данных самому себе.

```
#include <unistd.h>
#include <stdio.h>
int main(int argc, char **argv)
{
char *s = ”chanel”;
char buf[80];
int pipes[2];
pipe(pipes);
write(pipes[1], s, strlen(s) + 1);
read(pipes[0], buf, strlen(s) + 1);
close(pipes[0]);
close(pipes[1]);
printf(“%s\n”, buf);
return 0;
}
```

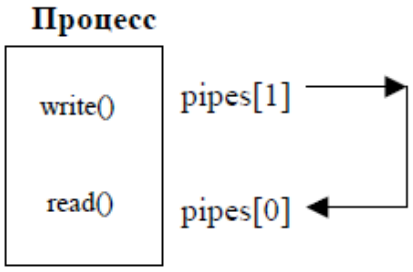

Рисунок 4 – Обмен через канал в рамках одного процесса

Чаще всего, однако, канал используется для обмена данными между несколькими процессами. При организации такого обмена используется тот факт, что при порождении сыновнего процесса посредством системного вызова fork() наследуется таблица файловых дескрипторов процесса-отца, т.е. все файловые дескрипторы, доступные процессу-отцу, будут доступны и процессу-сыну. Таким образом, если перед порождением потомка был создан канал, файловые дескрипторы для доступа к каналу будут унаследованы и сыном. В итоге обоим процессам оказываются доступны дескрипторы,

связанные с каналом, и они могут использовать канал для обмена данными (см. Рис. 5 и Пример 3).

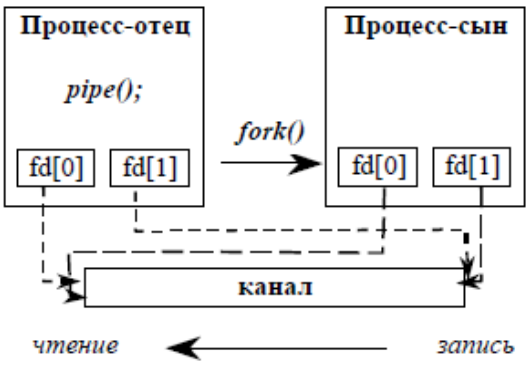

Рисунок 5 – Пример обмена данными между процессами через канал

Пример 3. Схема взаимодействия процессов с использованием канала.

```
#include <sys/types.h>
#include <unistd.h>
int main(int argc, char **argv)
{
int fd[2];
pipe(fd);
if (fork())
{/*процесс-родитель*/
close(fd[0]); /* закрываем ненужный дескриптор */
write (fd[1], …);
…
close(fd[1]);
…
}
else
{/*процесс-потомок*/
close(fd[1]); /* закрываем ненужный дескриптор */
while(read (fd[0], …))
{
…
}
…
}
}
```

Аналогичным образом может быть организован обмен через канал между, например, двумя потомками одного порождающего процесса и вообще между любыми родственными процессами − единственным требованием здесь, как уже говорилось, является необходимость создавать канал в порождающем процессе прежде, чем его дескрипторы будут унаследованы порожденными процессами.

Как правило, канал используется как однонаправленное средство передачи данных, т.е. только один из двух взаимодействующих процессов

осуществляет запись в него, а другой процесс осуществляет чтение, при этом каждый из процессов закрывает не используемый им дескриптор. Это особенно важно для неиспользуемого дескриптора записи в канал, так как именно при закрытии пишущей стороны канала в него помещается символ конца файла. Если, к примеру, в рассмотренном Пример 3 процесс-потомок не закроет свой дескриптор записи в канал, то при последующем чтении из канала, исчерпав все данные из него, он будет заблокирован, так как записывающая сторона канала будет открыта, и следовательно, читающий процесс будет ожидать очередной порции данных.

Пример 4. Реализация конвейера.

Пример реализации конвейера print|wc – две программы будут работать параллельно, причем вывод программы print будет подаваться на вход программы wc. Программа print печатает некоторый текст. Программа wc считает количество прочитанных строк, слов и символов в своем стандартном вводе.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(int argc, char **argv)
{
int fd[2];
pipe(fd); /*организован канал*/
if (fork())
{
/*процесс-родитель*/
dup2(fd[1], 1); /* отождествили стандартный вывод с файловым
дескриптором канала, предназначенным для записи */
close(fd[1]); /* закрыли файловый дескриптор канала,
предназначенный для записи */
close(fd[0]); /* закрыли файловый дескриптор канала,
предназначенный для чтения */
exelp(“print”, ”print”, 0); /* запустили программу print */
}
/*процесс-потомок*/
dup2(fd[0], 0); /* отождествили стандартный ввод с файловым
дескриптором канала, предназначенным для чтения*/
close(fd[0]); /* закрыли файловый дескриптор канала,
предназначенный для чтения */
close(fd[1]); /* закрыли файловый дескриптор канала,
предназначенный для записи */
execl(“/usr/bin/wc”, ”wc”, 0); /* запустили программу wc */
}
```

В приведенном выше тексте программы используется системный вызов dup2():

```
#include <unistd.h>
int dup2(int oldfd, int newfd);
```

Параметрами этого вызова являются два файловых дескриптора. Вызов создает копию дескриптора oldfd в дексрипторе newfd, при этом, если ранее дескриптор newfd был ассоциирован с каким-то другим открытым файлом, то перед переназначением он освобождается.

c. Именованные каналы (FIFO)

Рассмотренные выше программные каналы имеют важное ограничение: так как доступ к ним возможен только посредством дескрипторов, возвращаемых при порождении канала, необходимым условием взаимодействия процессов через канал является передача этих дескрипторов по наследству при порождении процесса. Именованные каналы (FIFO-файлы) расширяют свою область применения за счет того, что подключиться к ним может любой процесс в любое время, в том числе и после создания канала. Это возможно благодаря наличию у них имен.

FIFO-файл представляет собой отдельный тип файла в файловой системе UNIX, который обладает всеми атрибутами файла, такими как имя владельца, права доступа и размер. Для его создания используется системный вызов mknod() :

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
int mknod (char *pathname, mode_t mode, dev);
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo (char *pathname, mode_t mode);
```

В обоих вызовах первый аргумент представляет собой имя создаваемого канала, во втором указываются права доступа к нему для владельца, группы и прочих пользователей, и кроме того, устанавливается флаг, указывающий на то, что создаваемый объект является именно FIFO-файлом (в разных версиях ОС он может иметь разное символьное обозначение – S_IFIFO или I_FIFO). Третий аргумент вызова mknod() игнорируется.

После создания именованного канала любой процесс может установить с ним связь посредством системного вызова open(). При этом действуют следующие правила:

- если процесс открывает FIFO-файл для чтения, он блокируется до тех пор, пока какой-либо процесс не откроет тот же канал на запись;
- если процесс открывает FIFO-файл на запись, он будет заблокирован до тех пор, пока какой-либо процесс не откроет тот же канал на чтение;
- процесс может избежать такого блокирования, указав в вызове open() специальный флаг (в разных версиях ОС он может иметь

разное символьное обозначение – O_NONBLOCK или O_NDELAY). В этом случае в ситуациях, описанных выше, вызов open() сразу же вернет управление процессу, однако результат операций будет разным: попытка открытия по чтению при закрытой записывающей стороне будет успешной, а попытка открытия по записи при закрытой читающей стороне – вернет неуспех.

Правила работы с именованными каналами, в частности, особенности операций чтения-записи, полностью аналогичны неименованным каналам.

4. Реализация нитей

Рассмотрим модель многопоточного (многонитевого) программирования, описанную в стандарте POSIX – она называется pthreads.

Библиотека pthreads реализует прикладные нити, для которых практически не требуется поддержка ядра ОС. Функционирование таких нитей (в частности, переключение выполнения между ними) практически не требует обращения к ОС и вследствие этого является очень быстрым и несет минимум накладных расходов.

Для каждой нити поддерживаются следующие элементы:

- идентификатор нити;
- динамический стек;
- набор регистров (счетчик команд, регистр стека);
- сигнальная маска;
- значение приоритета;
- специальная память;

Каждой создаваемой нити присваивается идентификатор, уникальный среди всех нитей данного процесса – он характеризует ее подобно тому, как PID характеризует процесс. Наличие собственного динамического стека и регистров позволяет нити выполняться независимо от других нитей. В момент создания нить наследует сигнальную маску породившего ее процесса, а также получает от него по наследству значения приоритета, содержимое стека и регистров. В дальнейшем нить может изменить свою сигнальную маску и значение приоритета.

Подобно тому, как выполнение процесса ассоциируется с некой основной функцией реализующей его программы (в языке Си такой функцией является main()), каждой нити при создании ставится в соответствие функция, выполнением которой она занимается на протяжении всего своего существования. Нить завершается, когда происходит выход из соответствующей ей функции, либо когда происходит обращение к специальной функции выхода (pthread_exit()).

До того, как процесс явно породит хотя бы одну нить, в нем уже существует одна нить выполнения, ассоциированная непосредственно с

самим процессом. Таким образом, при явном порождении первой нити, фактически, в процессе будет насчитываться уже две нити: одна (вновь порожденная) будет заниматься выполнением назначенной ей функции, в то время как вторая – основная нить процесса – продолжит выполнение самого процесса.

Как и системные вызовы, все функции библиотеки pthreads (если явно не указано иное) в случае успешного завершения возвращают 0, в противном случае возвращается ненулевой код ошибки. Они не устанавливают переменную errno, но используют константы, описанные в <errno.h>, при непосредственном возврате значений, сигнализирующих об ошибках.

a. Представление атрибутов нити

Для описания атрибутов и управления ими в библиотеке pthreads вводятся понятие набора атрибутов нити и комплект функций для создания, удаления, запроса и изменения значений атрибутов.

Объект «набор атрибутов» может существовать независимо ни от какой нити, и один и тот же набор атрибутов может быть связан с несколькими нитями, что сокращает объем кода в том случае, если нескольким нитям следует придать одни и те же значения атрибутов.

Для создания набора атрибутов нити служит функция:

```
# include <pthread.h>
int pthread_attr_init(pthread_attr_t* attr_p);
```

Функция создает новый набор атрибутов нити, придает им значения по умолчанию и размещает в области памяти, на которую указывает переданный ей параметр. Тип данных pthread_attr_t служит для представления набора атрибутов нити.

Набор атрибутов существует до тех пор, пока не будет явно уничтожен функцией

```
# include <pthread.h>
int pthread_attr_destroy(pthread_attr_t* attr_p);
```

Эта функция удаляет набор атрибутов, на который указывает переданный ей параметр.

b. Создание нити

Для создания новой нити используется библиотечная функция:

```
#include <pthread.h>
```

```
int pthread_create(pthread_t* tid_p, const pthread_attr_t*
attr, void *(*funcp) (void *), void *argp);
```

Эта функция создает новую нить для выполнния функции, указатель на которую задан аргументом funcp. Как видно из прототипа, такая функция должна принимать один параметр типа void * и возвращать значение такого же типа. Фактический аргумент для функции, связанной с создаваемой нитью, передается в параметре argp. Практически, такие типы параметра и возвращаемого значения позволяют функции нити принимать и возвращать «указатель на что угодно», что дает необходимую гибкость. Если же в функцию нужно передать несколько параметров, следует определить для них структуру и передавать ее адрес в качестве единственного аргумента.

Для представления идентификатора нити служит тип данных pthread_t. Идентификатор создаваемой нити будет возвращен через параметр tid_p. Если приложение не интересуется идентификатором создаваемой нити, вместо этого параметра можно передать NULL.

Нить может узнать свой собственный идентификатор, обратившись к функции

```
#include <pthread.h>
pthread_t pthread_self(void);
```

Аргумент attr задает набор атрибутов создаваемой нити (он должен быть создан предварительно с помощью рассмотренной выше функции pthread_attr_init). Если нити следует использовать значения по умолчанию для всех своих атрибутов, вместо этого аргумента также можно передать NULL.

При последующем изменении значений атрибутов из набора, эти изменения не распространяются на те нити, которые были созданы ранее с использованием этого набора. Новые значения атрибутов получат лишь нити, созданные после изменения атрибутов. Для того, чтобы изменить значение атрибута уже существующей нити, необходимо явно обратиться к соответствующей функции, устанавливающей значение того или иного атрибута.

c. Завершение нити. Ожидание завершения нити

Нить может завершить свое выполнение, обратившись к функции

```
#include <pthread.h>
void pthread_exit(void *retval);
```

Этой функции в качестве аргумента можно передать указатель на статическую область памяти, содержащую код возврата (каждой нити выделяется свой собственный динамический стек, в котором размещаются

фактический параметр и локальные переменные связанной с ней функции и т.п. Этот стек, как и другие ресурсы нити, может быть разрушен при ее завершении, поэтому код возврата там размещать нельзя). Кроме того, нить может завершиться и неявно – если произойдет возврат из функции нити. Напомним, что возвращаемое значение функции нити имеет тип void * − в этом случае значение, переданное оператору return, и будет адресом кода возврата.

Подобно тому, как процесс может ожидать завершения порожденных им процессов-потомков при помощи системного вызова wait(), любая нить может приостановить свое выполнение в ожидании завершения другой нити того ж процесса, используя функцию:

```
#include <pthread.h>
int pthread_join(pthread_t tid, void ** ret);
```

Нить, вызвавшая эту функцию, приостанавливается до тех пор, пока не завершится выполнение нити с идентификатором tid. Как и в случае с системным вызовом wait(), ожидающая нить может получить код завершения ожидаемой нити, который будет записан по адресу, указанному во втором параметре. Если ожидающая нить не интересуется кодом завершения, в качестве этого параметра можно передать NULL.

Однако, существует важное отличие механизма ожидания завершения нитей от такового же для процессов. Любая нить может находиться в одном из двух состояний: она может быть ожидаемой (joinable) либо обособленной (detached). При завершении ожидаемой нити освобождаются все ресурсы, связанные с ней, за исключением ее идентификатора, а также тех структур данных, что необходимы для хранения ее кода возврата и информации о причинах ее завершения. Для того, чтобы освободить эти ресурсы, необходимо, чтобы другая нить явно обратилась к pthread_join().Ни библиотека, ни ОС не предпринимют никаких действий для освобождения указанных ресурсов, если после завершения ожидаемой нити никакая другая нить не обращается к pthread_join(), чтобы узнать результат завершения этой нити.

При завершении же обособленной нити все ресурсы, связанные с ней, освобождаются немедленно. Никакая другая нить не может ожидать ее завершения посредством вызова pthread_join(). Идентификатор, принадлежавший завершившейся обособленной нити, может быть сразу же присвоен новой создаваемой нити.

Статус нити – будет ли она ожидаемой или обособленной – определяется атрибутом detachstate, который может быть задан в наборе атрибутов, используемом при создании нити, при помощи функции:

```
#include <pthread.h>
int pthread_attr_setdetachstate(pthread_attr_t *attr, int
detach_state);
```

Первым аргументом ее является набор атрибутов, использованный для создания нити. Значениями второго аргумента могут быть константы PTHREAD_CREATE_JOINABLE (создать ожидаемую нить) либо PTHREAD_CREATE_DETACHED (создать обособленную нить).

Узнать текущее значение этого атрибута можно, обратившись к функции:

```
#include <pthread.h>
int pthread_attr_getdetachstate(pthread_attr_t *attr, int
*detach_state);
```

Текущее значение атрибута будет записано во второй параметр функции.

Кроме того, нить, созданная как ожидаемая, впоследствии может изменить свой статус на обособленную, обратившись к функции:

```
#include <pthread.h>
int pthread_detach(pthread_t tid);
```

В качестве аргумента ей указывается идентификатор нити.

5. Сокеты

Сокеты представляют собой в определенном смысле обобщение механизма каналов, но с учетом возможных особенностей, возникающих при работе в сети. Кроме того, они предоставляют больше возможностей по передаче сообщений, например, могут поддерживать передачу экстренных сообщений вне общего потока данных. Общая схема работы с сокетами любого типа такова: каждый из взаимодействующих процессов должен на своей стороне создать и отконфигурировать сокет, после чего процессы могут осуществить соединение с использованием этой пары сокетов. По окончании взаимодействия сокеты уничтожаются.

Механизм сокетов чрезвычайно удобен при разработке взаимодействующих приложений, образующих систему «клиент-сервер». Клиент посылает серверу запросы на предоставление услуги, а сервер отвечает на эти запросы.

Схема использования механизма сокетов для взаимодействия в рамках модели «клиент-сервер» такова. Процесс-сервер запрашивает у ОС сокет и, получив его, присваивает ему некоторое имя (адрес), которое предполагается заранее известным всем клиентам, которые захотят общаться с данным сервером. После этого сервер переходит в режим ожидания и обработки запросов от клиентов. Клиент, со своей стороны, тоже создает сокет и запрашивает соединение своего сокета с сокетом сервера, имеющим известное ему имя (адрес). После того, как соединение будет установлено, клиент и сервер могут обмениваться данными через соединенную пару сокетов.

а. Типы сокетов

Сокеты подразделяются на несколько типов в зависимости от типа коммуникационного соединения, который они используют. Два основных типа коммуникационных соединений (и, соответственно, использующих их сокетов) представляет собой соединение с использованием виртуального канала и датаграммное соединение.

Соединение с использованием виртуального канала – это последовательный поток байтов, гарантирующий надежную доставку сообщений с сохранением порядка их следования. Данные начинают передаваться только после того, как виртуальный канал установлен, и канал не разрывается, пока все данные не будут переданы.

Границы сообщений при таком виде соединений не сохраняются, т.е. приложение, получающее данные, должно само определять, где заканчивается одно сообщение и начинается следующее. Такой тип соединения может также поддерживать передачу экстренных сообщений вне основного потока данных, если это возможно при использовании конкретного выбранного протокола.

Платой за все положительные стороны установления виртуального канала является то, что сокет, участвующий в таком соединении, будет «занят» (аналогично телефонному аппарату) на протяжении всего сеанса связи, пока соединение не будет разорвано. Это означает, что данный сокет в это время не может параллельно использоваться в других соединениях.

Датаграммное соединение используется для передачи отдельных пакетов, содержащих порции данных – датаграмм. Для датаграмм не гарантируется доставка в том же порядке, в каком они были посланы. Для них не гарантируется доставка вообще, надежность соединения в этом случае ниже, чем при установлении виртуального канала. Однако датаграммные соединения, как правило, более быстрые. Примером датаграммного соединения из реальной жизни может служить обычная почта: письма и посылки могут приходить адресату не в том порядке, в каком они были посланы, а некоторые из них могут и совсем пропадать. Другим примером датаграммного соединения является передача сообщений посредством SMS: в отличие от телефонного разговора («соединения с установлением виртуального канала»), при этом оба телефонных аппарата («сокета») почти все время остаются незаняты и параллельно могут использоваться для приема и передачи сообщений, адресованных другим абонентам. Однако при передаче каждого сообщения необходимо заново указывать телефонный номер адресата («адрес сокета»), кроме того, абонент («вызвающая программа») сам должен сортировать сообщения, принятые от разных адресатов.

При использовании сокетов с установлением виртуального соединения для создания программ с архитектурой «клиент-сервер» возникает один очень важный вопрос. Как можно обеспечить параллельную обработку

сервером клиентских запросов, если серверный сокет на все время сеанса связи с конкретным клиентом будет занят этим соединением и недоступен для поступления запросов от других клиентов? Для решения этой проблемы следует использовать для серверного сокета режим прослушивания сокета. В этом режиме всякий раз при поступлении на серверный сокет запроса на соединение, порождается новый уникальный сокет-дубликат, который и участвует в этом соединении, в то время как исходный серверный сокет остается свободным и продолжает ожидать поступления новых запросов.

Продолжая аналогию с телефонными звонками, можно сказать, что сервер представляет в этом случае подобие call-центра с единым многоканальным телефоном. Всякий раз при поступлении звонка (запроса на соединение) он переводится на конкретного оператора (новый уникальный сокет), который и занимается его обработкой, а входящая линия снова становится свободной и может принимать новые звонки от других клиентов.

Общая схема работы с сокетами двух основных типов показана на Рисунке 6, 7.

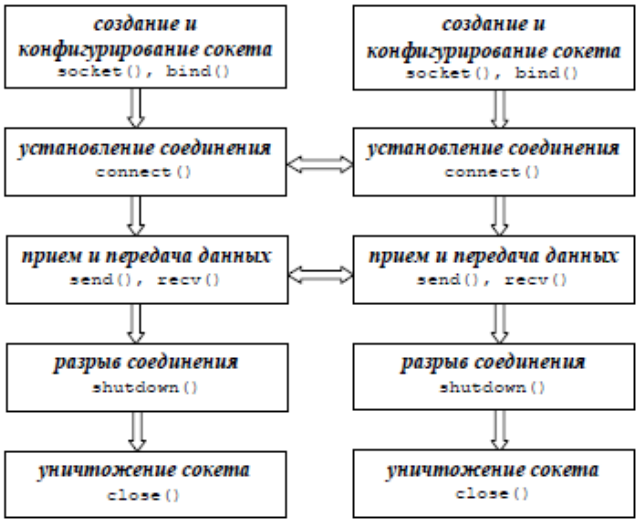

Рисунок 6 – Схема работы с сокетами с установлением виртуального канала

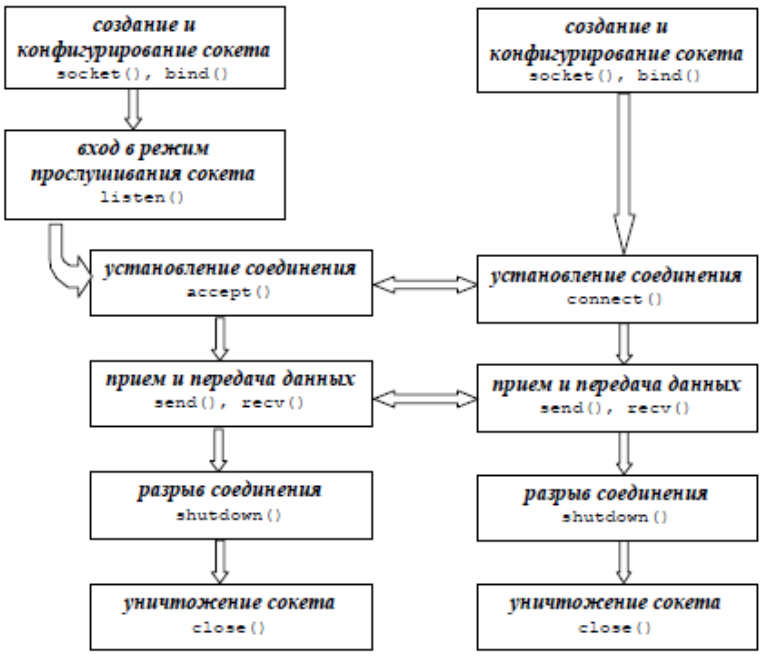

Рисунок 7 – Схема работы с сокетами с установлением виртуального канала. Клиент – сервер

Рисунок 8 – Схема работ с сокетами без установления виртуального канала

b. Коммуникационный домен

Поскольку сокеты могут использоваться как для локального, так и для удаленного взаимодействия, встает вопрос о пространстве адресов сокетов. При создании сокета указывается так называемый коммуникационный домен, к которому данный сокет будет принадлежать. Коммуникационный домен определяет форматы адресов и правила их интерпретации. Рассматриваются два основных домена: для локального взаимодействия – домен AF_UNIX и для взаимодействия в рамках сети – домен AF_INET (префикс AF обозначает сокращение от «address family» – семейство адресов). В домене AF_UNIX формат адреса – это допустимое имя файла, в домене AF_INET адрес образуют имя хоста + номер порта.

c. Создание и конфигурация сокета

i. Создание сокета

```
#include <sys/types.h>
#include <sys/socket.h>
int socket (int domain, int type, int protocol);
```

Функция создания сокета так и называется – socket(). У нее имеется три аргумента. Первый аргумент – domain – обозначает коммуникационный домен, к которому должен принадлежать создаваемый сокет. Для двух рассмотренных доменов соответствующие константы будут равны, как уже говорилось, AF_UNIX и AF_INET. Второй аргумент – type – определяет тип соединения, которым будет пользоваться сокет (и, соответственно, тип сокета). Для двух основных рассматриваемых типов сокетов это будут константы SOCK_STREAM для соединения с установлением виртуального канала и SOCK_DGRAM для датаграмм. Третий аргумент – protocol – задает конкретный протокол, который будет использоваться в рамках данного коммуникационного домена для создания соединения. Если установить значение данного аргумента в 0, система автоматически выберет подходящий протокол. В наших примерах мы так и будем поступать. Однако здесь для справки приведем константы для протоколов, используемых в домене AF_INET:

IPPROTO_TCP – обозначает протокол TCP (корректно при создании сокета типа SOCK_STREAM);

IPPROTO_UDP – обозначает протокол UDP (корректно при создании сокета типа SOCK_DGRAM).

Функция socket() возвращает в случае успеха положительное целое число – дескриптор сокета, которое может быть использовано в дальнейших вызовах при работе с данным сокетом. Заметим, что дескриптор сокета фактически представляет собой файловый дескриптор, а именно, он является индексом в таблице файловых дескрипторов процесса, и может использоваться в дальнейшем для операций чтения и записи в сокет, которые осуществляются подобно операциям чтения и записи в файл.

В случае если создание сокета с указанными параметрами невозможно (например, при некорректном сочетании коммуникационного домена, типа сокета и протокола), функция возвращает –1.

ii. Связывание сокета с адресом

Для того чтобы к созданному сокету мог обратиться какой-либо процесс извне, необходимо присвоить ему адрес. Формат адреса зависит от коммуникационного домена, в рамках которого действует сокет, и может представлять собой либо путь к файлу, либо сочетание IP-адреса и номера

порта. Но в любом случае связывание сокета с конкретным адресом осуществляется одной и той же функцией bind:

```
#include <sys/types.h>
#include <sys/socket.h>
int bind (int sockfd, struct sockaddr *myaddr, int addrlen);
```

Первый аргумент функции – дескриптор сокета, возвращенный функцией socket(); второй аргумент – указатель на структуру, содержащую адрес сокета. Для домена AF_UNIX формат структуры описан в <sys/un.h> и выглядит следующим образом:

```
#include <sys/un.h>
struct sockaddr_un {
short sun_family; /* == AF_UNIX */
char sun_path[108];
};
```

Для домена AF_INET формат структуры описан в <netinet/in.h> и выглядит следующим образом:

```
#include <netinet/in.h>
struct sockaddr_in {
short sin_family; /* == AF_INET */
u_short sin_port; /* port number */
struct in_addr sin_addr; /* host IP address */
char sin_zero[8]; /* not used */
};
```

Последний аргумент функции задает реальный размер структуры, на которую указывает myaddr.

d. Предварительное установление соединения

i. Сокеты с установлением соединения. Запрос на соединение

Различают сокеты с предварительным установлением соединения, когда до начала передачи данных устанавливаются адреса сокетов отправителя и получателя данных – такие сокеты соединяются друг с другом и остаются соединенными до окончания обмена данными; и сокеты без установления соединения, когда соединение до начала передачи данных не устанавливается, а адреса сокетов отправителя и получателя передаются с каждым сообщением. Если тип сокета – виртуальный канал, то сокет должен устанавливать соединение, если же тип сокета – датаграмма, то, как правило, это сокет без установления соединения (хотя в отдельных случаях для удобства программист может установить соединение – тогда он сможет

использовать менее громоздкие вызовы приема-передачи, не указывая в них всякий раз структуры адреса). Для установления соединения служит следующая функция:

```
#include <sys/types.h>
#include <sys/socket.h>
int connect (int sockfd, struct sockaddr *serv_addr, int
addrlen);
```

Здесь первый аргумент – дескриптор сокета, второй аргумент – указатель на структуру, содержащую адрес сокета, с которым производится соединение, в формате, который мы обсуждали выше, и третий аргумент содержит реальную длину этой структуры. Функция возвращает 0 в случае успеха и –1 в случае неудачи, при этом код ошибки можно посмотреть в переменной errno.

Заметим, что в рамках модели «клиент-сервер» клиенту, вообще говоря, не важно, какой адрес будет назначен его сокету, так как никакой процесс не будет пытаться непосредственно установить соединение с сокетом клиента. Поэтому клиент может не вызывать предварительно функцию bind(), в этом случае при вызове connect() система автоматически выберет приемлемые значения для локального адреса клиента. Однако сказанное справедливо только для взаимодействия в рамках домена AF_INET, в домене AF_UNIX клиентское приложение должно явно позаботиться о связывании сокета.

ii. Сервер: прослушивание сокета и подтверждение соединения

Следующие два вызова используются сервером только в том случае, если используются сокеты с предварительным установлением соединения.

```
#include <sys/types.h>
#include <sys/socket.h>
int listen (int sockfd, int backlog);
```

Этот вызов используется процессом-сервером для того, чтобы сообщить системе о том, что он готов к обработке запросов на соединение, поступающих на данный сокет. Тем самым сервер входит в режим прослушивания сокета. До тех пор, пока процесс – владелец сокета не вызовет listen(), все запросы на соединение с данным сокетом будут возвращать ошибку.

Первый аргумент функции – дескриптор сокета. Второй аргумент, backlog, содержит максимальный размер очереди запросов на соединение. ОС буферизует приходящие запросы на соединение, выстраивая их в очередь до тех пор, пока процесс не сможет их обработать. В случае если очередь запросов на соединение переполняется, поведение ОС зависит от того, какой

протокол используется для соединения. Если конкретный протокол соединения не поддерживает возможность перепосылки (retransmission) данных, то соответствующий вызов connect() вернет ошибку ECONNREFUSED. Если же перепосылка поддерживается (как, например, при использовании TCP), ОС просто выбрасывает пакет, содержащий запрос на соединение, как если бы она его не получала вовсе. При этом пакет будет присылаться повторно до тех пор, пока очередь запросов не уменьшится и попытка соединения не увенчается успехом, либо пока не произойдет тайм-аут, определенный для протокола. В последнем случае вызов connect() завершится с ошибкой ETIMEDOUT. Это позволит клиенту отличить, был ли процесс-сервер слишком занят, либо он не функционировал. В большинстве систем максимальный допустимый размер очереди равен 5.

Конкретное соединение устанавливается при помощи вызова accept():

```
#include <sys/types.h>
#include <sys/socket.h>
int accept (int sockfd, struct sockaddr *addr, int *addrlen);
```

Этот вызов применяется сервером для удовлетворения поступившего клиентского запроса на соединение с сокетом, который сервер к тому моменту уже прослушивает (т.е. предварительно была вызвана функция listen()). Вызов accept() извлекает первый запрос из очереди запросов, ожидающих соединения, и устанавливает с ним соединение. Если к моменту вызова accept() очередь запросов на соединение пуста, процесс, вызвавший accept(), блокируется до поступления запросов.

Когда запрос поступает и соединение устанавливается, accept() создает новый сокет, который будет использоваться для работы с данным соединением, и возвращает дескриптор этого нового сокета, соединенного с сокетом клиентского процесса. При этом первоначальный сокет продолжает оставаться в состоянии прослушивания. Через новый сокет осуществляется обмен данными, в то время как старый сокет продолжает обрабатывать другие поступающие запросы на соединение (именно первоначально созданный сокет связан с адресом, известным клиентам, поэтому все клиенты могут слать запросы только на соединение с этим сокетом). Это позволяет процессу-серверу поддерживать несколько соединений одновременно.

Во втором параметре передается указатель на структуру, в которой возвращается адрес клиентского сокета, с которым установлено соединение, а в третьем параметре возвращается реальная длина этой структуры. Благодаря этому сервер всегда знает, куда ему в случае надобности следует послать ответное сообщение. Если адрес клиента не интересен, то в качестве второго аргумента можно передать NULL.

e. Прием и передача данных

Собственно для приема и передачи данных через сокет используются три пары функций.

```
#include <sys/types.h>
#include <sys/socket.h>
int send(int sockfd, const void *msg, int len, unsigned int
flags);
int recv(int sockfd, void *buf, int len, unsigned int flags);
```

Эти функции используются для обмена только через сокет с предварительно установленным соединением. Аргументы функции send(): sockfd – дескриптор сокета, через который передаются данные, msg и len – сообщение и его длина. Если сообщение слишком длинное для того протокола, который используется при соединении, оно не передается и вызов возвращает ошибку EMSGSIZE. Если же сокет окажется переполнен, т.е. в его буфере не хватит места, чтобы поместить туда сообщение, выполнение процесса блокируется до появления возможности поместить сообщение. Функция send() возвращает количество переданных байт в случае успеха и -1 в случае неудачи. Код ошибки при этом устанавливается в errno. Аргументы функции recv() аналогичны: sockfd – дескриптор сокета, buf и len – указатель на буфер для приема данных и его первоначальная длина. В случае успеха функция возвращает количество считанных байт, в случае неудачи -1.

Последний аргумент обеих функций – flags – может содержать комбинацию специальных опций. Нас будут интересовать две из них:

MSG_OOB – этот флаг сообщает ОС, что процесс хочет осуществить прием/передачу экстренных сообщений;

MSG_PEEK – данный флаг может устанавливаться при вызове recv(). При этом процесс получает возможность прочитать порцию данных, не удаляя ее из сокета, таким образом, что последующий вызов recv() вновь вернет те же самые данные.

Другая пара функций, которые могут использоваться при работе с сокетами с предварительно установленным соединением – это обычные read() и write(), в качестве дескриптора которым передается дескриптор сокета.

И, наконец, пара функций, которая может быть использована как с сокетами с установлением соединения, так и с сокетами без установления соединения:

```
#include <sys/types.h>
#include <sys/socket.h>
int sendto(int sockfd, const void *msg, int len, unsigned int
flags, const struct sockaddr *to, int tolen);
int recvfrom(int sockfd, void *buf, int len, unsigned int
flags, struct sockaddr *from, int *fromlen);
```

В последних двух аргументах в функцию sendto() должны быть переданы указатель на структуру, содержащую адрес получателя, и ее размер, а функция recvfrom() в них возвращает соответственно указатель на структуру с адресом отправителя и ее реальный размер.

Перед вызовом recvfrom() параметр fromlen должен быть установлен равным первоначальному размеру структуры from. Здесь, как и в функции accept, если не интересен адрес отправителя, в качестве from можно передать NULL.

f. Завершение работы с сокетом

Если процесс закончил прием либо передачу данных, ему следует закрыть соединение. Это можно сделать с помощью функции shutdown():

```
# include <sys/types.h>
# include <sys/socket.h>
int shutdown (int sockfd, int mode);
```

Помимо дескриптора сокета, ей передается целое число, которое определяет режим закрытия соединения. Если mode=0, то сокет закрывается для чтения, при этом все дальнейшие попытки чтения будут возвращать EOF. Если mode=1, то сокет закрывается для записи, и при осуществлении в дальнейшем попытки передать данные будет выдан кода неудачного завершения (-1). Если mode=2, то сокет закрывается и для чтения, и для записи.

Аналогично файловому дескриптору, дескриптор сокета освобождается системным вызовом close(). При этом даже если до этого не был вызван shutdown(), соединение будет закрыто. Таким образом, если по окончании работы с сокетом вы собираетесь закрыть соединение и по чтению, и по записи, можно было бы сразу вызвать close() для дескриптора данного сокета, опустив вызов shutdown(). Однако, есть небольшое различие с тем случаем, когда предварительно был вызван shutdown(). Если используемый для соединения протокол гарантирует доставку данных (т.е. тип сокета – виртуальный канал), то вызов close() будет блокирован до тех пор, пока система будет пытаться доставить все данные, находящиеся «в пути» (если таковые имеются), в то время как вызов shutdown() извещает систему о том, что эти данные уже не нужны и можно не предпринимать попыток их доставить, и соединение закрывается немедленно. Таким образом, вызов shutdown() важен в первую очередь для закрытия соединения сокета с использованием виртуального канала.

g. Общая схема работы с сокетами

Можно изобразить типичную последовательность системных вызовов для работы с сокетами с установлением соединения в следующем виде:

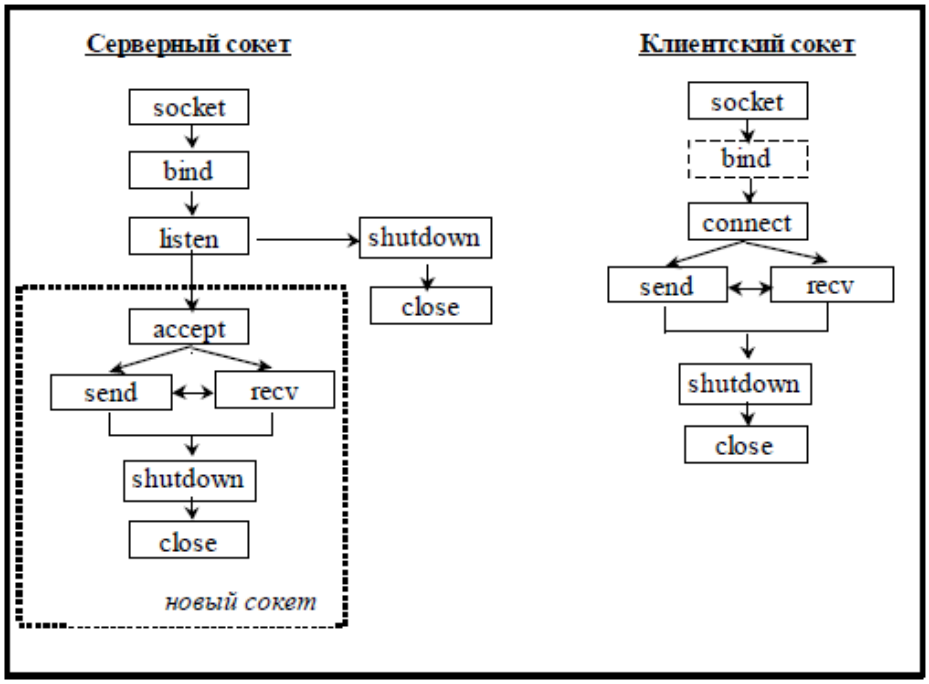

Рисунок 9 – Последовательность вызовов при работе с сокетами с установлением соединения

Пример 1. Работа с локальными сокетами

Пример иллюстрирует работу с сокетами в рамках локального домена (AF_UNIX). Программа в зависимости от параметра командной строки исполняет роль клиента или сервера. Клиент и сервер устанавливают соединение с использованием датаграммных сокетов. Клиент читает строку со стандартного ввода и пересылает серверу; сервер посылает ответ в зависимости от того, какова была строка. При введении строки «quit» и клиент, и сервер завершаются.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <stdio.h>
#include <string.h>
#define SADDRESS "mysocket"
#define CADDRESS "clientsocket"
#define BUFLEN 40
int main(int argc, char **argv)
{
struct sockaddr_un party_addr, own_addr;
int sockfd;
int is_server;
char buf[BUFLEN];
int party_len;
int quitting;
```

```
if (argc != 2) {
printf("Usage: %s client|server.\n", argv[0]);
return 0;
}
quitting = 1;
/* определяем, кто мы: клиент или сервер*/
is_server = !strcmp(argv[1], "server");
memset(&own_addr, 0, sizeof(own_addr)); /* предварительно
очищаем структуру */
own_addr.sun_family = AF_UNIX; /* локальный домен */
strcpy(own_addr.sun_path, is_server ? SADDRESS : CADDRESS);
/* создаем сокет, в sockfd помещается дескриптор создаваемого
сокета*/
if ((sockfd = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0)
{
printf("can't create socket\n");
return 0;
}
/* связываем сокет */
unlink(own_addr.sun_path); /* удаляем файл, иначе, если такой
файл уже существует, то считается, что такое имя уже занято и
связывание завершается неуспешно */
if (bind(sockfd, (struct sockaddr *) &own_addr,
sizeof(own_addr.sun_family)+ strlen(own_addr.sun_path) +1) < 0)
/* к размеру структуры прибавляем реальную длину строки с
адресом, включая завершающий ‘\0’ */
{
printf("can't bind socket!");
return 0;
}
if (!is_server)
{
/* это – клиент */
/* аналогично заполняем адресную структуру с именем файла
сервера */
memset(&party_addr, 0, sizeof(party_addr));
party_addr.sun_family = AF_UNIX;
strcpy(party_addr.sun_path, SADDRESS);
printf("type the string: ");
/* читаем строки с командами от пользователя */
while (gets(buf)) {
/* не пора ли выходить? */
quitting = (!strcmp(buf, "quit"));
/* считали строку и передаем ее серверу, размер строки
указываем с учетом завершающего ‘\0’ */
if (sendto(sockfd, buf, strlen(buf) + 1, 0, (struct sockaddr *)
&party_addr, sizeof(party_addr.sun_family) +
strlen(SADDRESS)) != strlen(buf) + 1)
{
/* Строка не отправилась */
printf("client: error writing socket!\n");
unlink(own_addr.sun_path); /*удаляем файл при выходе */
return 0;
}
```

```
/*получаем ответ и выводим его на печать; NULL указывает на то,
что нам не нужна адресная структура отправителя сообщения */
if (recvfrom(sockfd, buf, BUFLEN, 0, NULL, 0) < 0)
{ // ошибка при приеме данных
printf("client: error reading socket!\n");
unlink(own_addr.sun_path); /*удаляем файл при выходе */
return 0;
}
printf("client: server answered: %s\n", buf);
if (quitting) break;
printf("type the string: ");
} // while
close(sockfd);
unlink(own_addr.sun_path); /*удаляем файл при выходе */
return 0;
} // if (!is_server)
/* это – сервер */
while (1)
{
/* получаем строку от клиента и выводим на печать; в последних
двух параметрах получаем адресную структуру отправителя и ее
размер */
party_len = sizeof(party_addr);
if (recvfrom(sockfd, buf, BUFLEN, 0,(struct sockaddr *)
&party_addr, &party_len) < 0)
{ // ошибка при приеме данных
printf("server: error reading socket!");
unlink(own_addr.sun_path); /*удаляем файл при выходе */
return 0;
}
printf("server: received from client: %s \n", buf);
/* не пора ли выходить? */
quitting = (!strcmp(buf, "quit"));
if (quitting)
strcpy(buf, "quitting now!");
else
/* в зависимости от запроса готовим ответ */
if (!strcmp(buf, "ping!"))
strcpy(buf, "pong!");
else
strcpy(buf, "wrong string!");
/* посылаем ответ */
if (sendto(sockfd, buf, strlen(buf) + 1, 0, (struct sockaddr *)
&party_addr, party_len) != strlen(buf)+1)
{ // ошибка при передаче данных
printf("server: error writing socket!\n");
unlink(own_addr.sun_path); /*удаляем файл при выходе */
return 0;
}
if (quitting) break;
} // while
close(sockfd);
unlink(own_addr.sun_path); /*удаляем файл, который был создан в
файловой системе при связывании сокета*/
return 0;
```

```
}
```