Московский Государственный Университет им. М.В. Ломоносова
Факультет вычислительной математики и кибернетики

Н.В.Вдовикина, А.В.Казунин, И.В.Машечкин, А.Н.Терехин

Системное программное обеспечение: взаимодействие процессов.

(учебно-методическое пособие)

Москва
2002

УДК 681.3.06
ББК 32.973-018.2
С40

В пособии рассматриваются основные аспекты управления процессами в операционной системе и организации межпроцессного взаимодействия на примере операционной системы UNIX. Изложение проиллюстрировано большим количеством программных примеров. Пособие рекомендуется для студентов, аспирантов и преподавателей в поддержку курсов лекций «Системное программное обеспечение» и «Операционные системы».

Авторы выражают благодарность Е.М.Шляховой, Ю.О.Нестеровой, А.Н.Розинкину, О.И.Вдовикину за помощь в подготовке пособия.

УДК 681.3.06
ББК 32.973-018.2

Рецензенты:
чл.-корр. РАН Л.Н.Королев
доцент Е.А.Кузьменкова

Вдовикина Н.В., Казунин А.В., Машечкин И.В., Терехин А.Н.

С40 **Системное программное обеспечение: взаимодействие процессов: учебно-методическое пособие**.

Издательский отдел факультета ВМиК МГУ
(лицензия ИД № 05899 от 24.09.2001), 2002, - 183 с.

Печатается по решению Редакционно-издательского Совета факультета вычислительной математики и кибернетики МГУ им. М.В. Ломоносова

ISBN 5-89407-139-9

ОГЛАВЛЕНИЕ

ЧАСТЬ I. ТЕОРЕТИЧЕСКИЕ ОСНОВЫ. 5

1 Введение. 5

2 Понятие процесса. 5

2.1 Некоторые типы процессов. 6

2.1.1 «Полновесные процессы» 6

2.1.2 «Легковесные процессы» 7

2.2 Жизненный цикл процесса. 8

3 Синхронизация параллельных процессов. 12

3.1 Способы реализации взаимного исключения. 16

3.1.1 Запрещение прерываний и специальные инструкции. 16

3.1.2 Алгоритм Петерсона. 17

3.1.3 Активное ожидание. 18

3.1.4 Семафоры. 19

3.1.5 Мониторы. 20

3.1.6 Обмен сообщениями. 22

3.2 Классические задачи синхронизации процессов. 25

3.2.1 «Обедающие философы» 25

3.2.2 Задача «читателей и писателей» 28

3.2.3 Задача о «спящем парикмахере» 31

ЧАСТЬ II. РЕАЛИЗАЦИЯ ПРОЦЕССОВ. 34

4 Реализация процессов в ОС UNIX 34

4.1 Понятие процесса в UNIX. 34

4.1.1 Контекст процесса. 34

4.1.2 Тело процесса. 35

4.1.3 Аппаратный контекст. 36

4.1.4 Системный контекст. 37

4.2 Аппарат системных вызов в ОС UNIX. 38

4.3 Порождение новых процессов. 41

4.4 Механизм замены тела процесса. 45

4.5 Завершение процесса 50

4.6 Жизненный цикл процесса в ОС UNIX. 55

4.7 Начальная загрузка. Формирование 0 и 1 процессов. 56

4.8 Планирование процессов в ОС UNIX. 58

4.9 Принципы организация свопинга. 60

ЧАСТЬ III. РЕАЛИЗАЦИЯ ВЗАИМОДЕЙСТВИЯ ПРОЦЕССОВ. ..62

5 Элементарные средства межпроцессного взаимодействия. 65

5.1 Сигналы. 65

5.2 Надежные сигналы. 73

5.3 Программные каналы 79

5.4 Именованные каналы (FIFO) 87

5.5 Нелокальные переходы. 90

5.6 Трассировка процессов. 93

6 Средства межпроцессного взаимодействия System V. 99
6.1 Организация доступа и именования в разделяемых ресурсах. ..99
6.1.1 Именование разделяемых объектов. 99
6.1.2 Генерация ключей: функция ftok(). 100
6.1.3 Общие принципы работы с разделяемыми ресурсами. 101
6.2 Очередь сообщений. 103
6.2.1 Доступ к очереди сообщений. 104
6.2.2 Отправка сообщения. 104
6.2.3 Получение сообщения. 105
6.2.4 Управление очередью сообщений. 106
6.3 Разделяемая память 112
6.3.1 Создание общей памяти. 113
6.3.2 Доступ к разделяемой памяти. 113
6.3.3 Открепление разделяемой памяти. 114
6.3.4 Управление разделяемой памятью. 115
6.4 Семафоры. 116
6.4.1 Доступ к семафору 117
6.4.2 Операции над семафором 118
6.4.3 Управление массивом семафоров. 120

7 Взаимодействие процессов в сети. 126
7.1 Механизм сокетов. 126
7.1.1 Типы сокетов. Коммуникационный домен. 127
7.1.2 Создание и конфигурирование сокета. 128
7.1.3 Предварительное установление соединения. 131
7.1.4 Прием и передача данных. 133
7.1.5 Завершение работы с сокетом. 135
7.1.6 Резюме: общая схема работы с сокетами. 136
7.2 Среда параллельного программирования MPI 145
7.2.1 Краткий обзор параллельных архитектур. 145
7.2.2 Модель программирования MPI. 150
7.2.3 Функции общего назначения. Общая структура программы. 152
7.2.4 Прием и передача данных. Общие замечания. 156
7.2.5 Коммуникации «точка-точка». Блокирующий режим. 158
7.2.6 Коммуникации «точка-точка». Неблокирующий режим. 164
7.2.7 Коллективные коммуникации. 171

8 Алфавитный указатель упоминаемых библиотечных функций и системных вызовов. 181

9 Список литературы 183

ЧАСТЬ I. ТЕОРЕТИЧЕСКИЕ ОСНОВЫ.

1 ВВЕДЕНИЕ.

Вычислительная система (ВС) есть совокупность аппаратных и программных средств, функционирующих как единое целое и предназначенных для решения задач определенного класса. Любая вычислительная система обладает некоторым набором **ресурсов**. Эти ресурсы включают в себя как реально существующие физические ресурсы (устройства) с их реальными характеристиками, так и устройства, эксплутационные характеристики которых полностью или частично реализованы программным образом – виртуальные (логические) ресурсы (устройства). Под **операционной системой** (ОС) понимают комплекс программ осуществляющий управление, распределение и контроль за использованием ресурсов вычислительной системы.

Любая операционная система оперирует некоторыми сущностями (понятиями), которые во многом характеризуют свойства этой операционной системы. К таким сущностям могут относиться понятия – задача, процесс, объект, файл, набор данных и т.д.. Одним из важнейших таких понятий является понятие **процесса**. В общем случае процесс можно определить как совокупность данных и машинных команд, исполняющуюся в рамках ВС и обладающую правами на владение некоторым набором ресурсов. Эти права могут носить эксклюзивный характер, когда ресурс принадлежит только одному процессу, либо ресурс может одновременно принадлежать нескольким процессам – в этом случае ресурс называется **разделяемым**.

Во всех современных ВС поддерживается режим мультипрограммирования, поэтому одной из основных функций ОС является задача управления процессами в ВС. Эта задача включает в себя:

- обеспечение жизненного цикла процессов (порождение, выполнение и уничтожение процессов);
- распределение ресурсов ВС;
- синхронизацию процессов;
- организацию межпроцессного взаимодействия.

2 Понятие процесса.

Нетрудно найти целый ряд “синонимов” понятия типа “процесс” – процесс, нить, задача, задание или программа, причем в

различных ОС могут присутствовать только часть понятий из этого набора, и их интерпретация во многом будет зависеть от конкретной вычислительной среды, где они используются. Таким образом, разные операционные системы оперируют с разными понятиями относительно базовой сущности, с которой работает ОС . Более того эти понятия могут переплетаться, т.е. задача может состоять из нескольких процессов, а процесс имеет многонитевую структуру.

Для каждого процесса определена структура данных, фиксирующая текущее состояние процесса. Такую структуру называют **контекстом процесса**. Она существует с момента создания процесса и до момента завершения его работы. В зависимости от конкретной операционной системы, данная информационная структура может включать в себя содержимое пользовательского адресного пространства (т.е. содержимое сегментов программного кода, данных, стека, разделяемых сегментов и сегментов файлов, отображаемых в виртуальную память) – **пользовательский контекст**, содержимое аппаратных регистров (таких, как регистр счетчика команд, регистр состояния процессора, регистр указателя стека и регистров общего назначения) – **регистровый контекст**, а также структуры данных ядра ОС, связанные с этим процессом (**контекст системного уровня**). В общем случае данная информация может дополняться или меняться по ходу исполнения процесса в зависимости от того, чем в данный момент процесс занимался.

Одной из целей создания и поддержания такой структуры является возможность продолжения корректной работы процесса после приостановки его функционирования на процессоре. В момент выделения следующему процессу вычислительных ресурсов происходит так называемое переключение контекста, в результате которого происходит сохранение контекста текущего процесса и передача управления новому.

2.1 Некоторые типы процессов.

2.1.1 «Полновесные процессы»

Существует понятие «полновесные процессы» - это процессы, выполняющиеся внутри защищенных участков памяти операционной системы, то есть имеющие собственные виртуальные адресные пространства для статических и динамических данных. Для «полновесных процессов» можно сказать, что операционная система поддерживает их обособленность: у каждого процесса

имеется свое виртуальное адресное пространство, каждому процессу назначаются свои ресурсы - файлы, окна, семафоры и т.д. Такая обособленность нужна для того, чтобы защитить один процесс от другого, поскольку они, совместно используя все ресурсы ВС, конкурируют с друг другом. В общем случае процессы принадлежат разным пользователям, разделяющим один компьютер, и ОС берет на себя все функции, связанные с распределением ресурсов между конкурирующими процессами. В операционных системах управление такими процессами тесно связанно с управлением и защитой памяти, поэтому переключение процессора с выполнения одного процесса на выполнение другого является достаточно дорогой операцией по времени.

2.1.2 «Легковесные процессы»

Наряду с «полновесными процессами» существуют и «легковесные процессы», они же нити, которые в той или иной степени присутствуют в различных операционных системах. При мультипрограммировании повышается пропускная способность системы, но отдельный процесс никогда не может быть выполнен быстрее, чем если бы он выполнялся в однопрограммном режиме. Однако задача, решаемая в рамках одного процесса, может обладать внутренним параллелизмом, который позволяет ускорить ее выполнение. Например, в ходе выполнения задачи происходит обращение к внешнему устройству, и на время этой операции можно не блокировать полностью выполнение процесса, а продолжить вычисления по другой "ветви" процесса. Для этих целей современные ОС предлагают использовать механизм *многонитевой обработки (multithreading)*. При этом вводится новое понятие "нить" (thread). Нити, относящиеся к одному процессу, не настолько изолированы друг от друга, как процессы в традиционной многозадачной системе, между ними легко организовать тесное взаимодействие. Нити, или «легковесные процессы», во многих отношениях схожи с процессами. Каждая нить выполняется строго последовательно и имеет свой собственный программный счетчик и стек. Нити, как и процессы, могут, например, порождать нити-потомки, могут переходить из состояния в состояние. Подобно традиционным процессам (то есть процессам, состоящим из одной нити), нити могут находится в одном из следующих состояний: ВЫПОЛНЕНИЕ, ОЖИДАНИЕ и ГОТОВНОСТЬ. Пока одна нить заблокирована, другая нить того же процесса может выполняться. Нити разделяют процессор так, как это делают процессы, в соответствии с различными вариантами планирования.

Однако различные нити в рамках одного процесса не настолько независимы, как отдельные процессы. Все такие нити имеют одно и то же адресное пространство. Это означает, что они разделяют одни и те же глобальные переменные. Поскольку каждая нить может иметь доступ к каждому виртуальному адресу, одна нить может использовать стек другой нити. Все нити одного процесса всегда решают общую задачу одного пользователя, и аппарат нитей используется для более быстрого решения задачи путем ее распараллеливания. При этом программисту очень важно получить в свое распоряжения удобные средства организации взаимодействия частей одной задачи. Кроме разделения адресного пространства, все нити разделяют также набор открытых файлов, таймеров, сигналов и т.п.

Итак, нити имеют собственные: программный счетчик, стек, регистры, нити-потомки, состояние. В то же время все нити одного процесса разделяют: адресное пространство, глобальные переменные, открытые файлы, таймеры, статистическую информацию.

В рамках одной ОС понятия полновесного и легковесного процесса (нити) могут как взаимозаменяться (например, в случае ОС с однонитевой организацией процесса, когда каждый процесс может иметь лишь одну исполняемую нить), так и дополнять друг друга (в случае многонитевой организации процесса, когда каждый процесс представляет собой совокупность исполняемых нитей).

Обобщая сказанное, отметим, что понятие процесса в любой ОС включает в себя:

- исполняемый код;
- собственное виртуальное адресное пространство;
- совокупность ресурсов, выделенных данному процессу операционной системой;
- хотя бы одну исполняемую нить.

2.2 Жизненный цикл процесса.

С момента запуска и до завершения выполнения процесс может находиться в различных активных или пассивных состояниях, которые в совокупности описывают жизненный цикл процесса в вычислительной системе. Количество и характеристики таких состояний может меняться в зависимости от конкретной вычислительной системы. Можно выделить несколько основных состояний процесса:

1. ПОРОЖДЕНИЕ – состояние процесса, когда он уже создан, но не готов к запуску, при этом создаются информационные структуры, описывающие данный процесс; загружается кодовый сегмент процесса в оперативную память или в область свопинга.
2. ВЫПОЛНЕНИЕ - активное состояние процесса, во время которого процесс обладает всеми необходимыми ресурсами и непосредственно выполняется процессором;
3. ОЖИДАНИЕ - пассивное состояние процесса, процесс заблокирован, он не может выполняться по своим внутренним причинам, т.е. он ждет осуществления некоторого события, например, завершения операции ввода-вывода, получения сообщения от другого процесса, освобождения какого-либо необходимого ему ресурса;
4. ГОТОВНОСТЬ - также пассивное состояние процесса: процесс имеет все требуемые для него ресурсы, он готов выполняться, однако процессор занят выполнением другого процесса.
5. ЗАВЕРШЕНИЕ – конечное состояние в жизненном цикле процесса, процесс выгружается из памяти и разрушаются все структуры данных, связанные с ним.

Жизненный цикл процесса

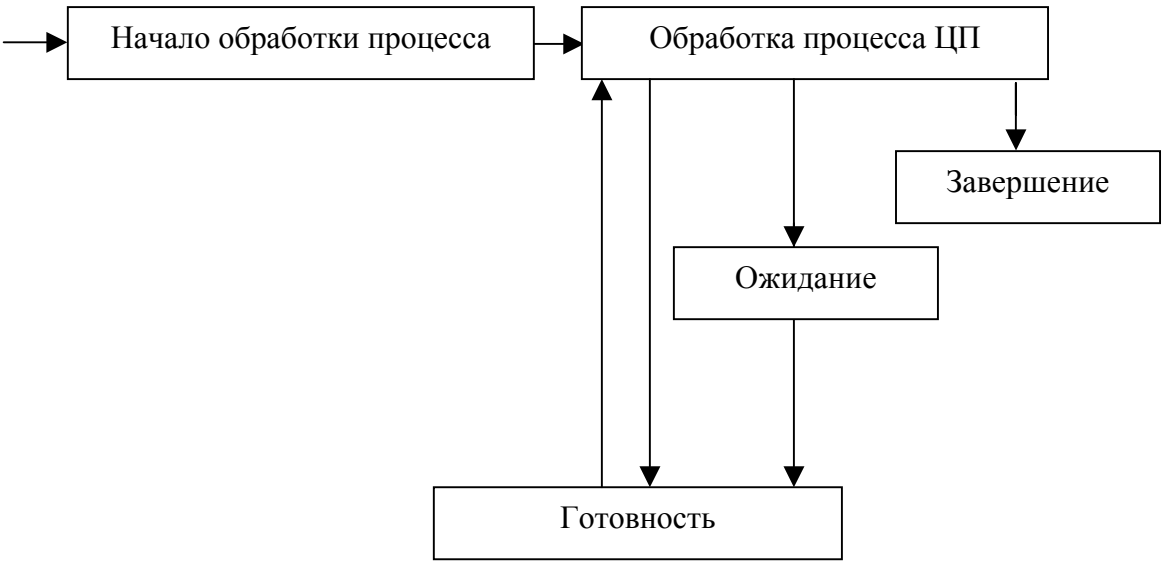

Рис. 1 Общая схема состояний процесса.

В общем случае жизненный цикл процесса представлен на **Рис. 1**. Жизненный цикл любого процесса начинается с момента его создания, когда он попадает в очередь готовых к выполнению процессов. Жизненный цикл процесса во всех состояниях, кроме

собственно выполнения, всегда связан с той или иной очередью, количество которых также зависит от операционной системы (см. **Рис. 1**). Основной причиной попадания процесса в ту или иную очередь является невозможность взаимодействия с тем или иным устройством (это может быть как центральный процессор, так и любое устройство ввода-вывода) в данный момент времени. Поэтому основные состояния процесса описываются набором очередей в операционной системе: очередь на начало обработки, очередь готовых процессов, очередь заблокированных процессов. Последняя есть совокупность очередей, связанных с ожиданием использования конкретных ресурсов в вычислительной системе. Количество таких очередей меняется в зависимости от конкретной архитектуры или операционной системы. Это множество очередей может состоять из очередей распределения процессов по функциональным устройствам, времени ЦП, доступа к внешним или внутренним устройствам ввода вывода, памяти. Итак, жизненный цикл процесса есть совокупность состояний, которые в основном характеризуются либо работой процесса, либо ожиданием в какой-либо очереди.

Как видно из **Рис. 1**, начальным этапом обработки процесса в операционной системе является очередь на запуск. Существование и длина такой очереди зависит от конкретной операционной системы. Если это однозадачная ОС, то длина такой очереди равна нулю, или попросту говоря ее не существует. Если ОС является мультизадачной, то длина такой очереди определяется конкретной ОС. Движение в этой очереди может быть организовано как с помощью элементарных алгоритмов типа FIFO, так и с помощью более сложных алгоритмов с использованием понятия приоритета и динамического планирования.

Из очереди на запуск, если такая существует, процесс переходит в стадию выполнения. Время его обработки ЦП также определяется алгоритмами планирования для конкретной ОС, т.е. процесс может работать, пока он не завершится, а может возникнуть переключение на другой процесс спустя некоторый квант времени или при появлении готового на выполнение более приоритетного процесса.

Процесс также может остановить свое выполнение, если он ожидает некоторого события, например, ввода с внешнего устройства. Тогда он попадает в очередь процессов, ожидающих ввод/вывод, и будет находится там до тех пор, пока ожидаемое событие не произойдет. Если процесс больше не ожидает никакого события, и он готов к выполнению, он попадает в очередь готовых процессов и ждет, пока ему будет выделено процессорное время.

Алгоритмы планирования очереди готовых процессов, как впрочем и всех других очередей, также зависят от конкретной операционной системы и во многом определяют ее тип, о чем будет подробнее рассказано ниже.

Описанная схема для различных ОС может отличаться как отсутствием некоторых состояний, так и присутствием некоторых дополнительных.

Еще раз подчеркнем, что для любой многозадачной операционной системы существует проблема управления различными очередями и временем центрального процессора. Операционная система должна обладать четкими критериями для определения того, какому готовому к выполнению процессу и когда предоставить ресурс процессора, какой процесс взять из очереди и какой поставить. Некоторые задачи планирования решаются программными средствами, а некоторые аппаратно. Существует множество различных алгоритмов планирования, преследующих различные цели и обеспечивающих различное качество мультипрограммирования[1].

[1] Отметим, что детальное рассмотрение многих конкретных алгоритмов планирования выходит за рамки данного пособия.

3 Синхронизация параллельных процессов.

В однопроцессорных системах имеет место так называемый **псевдопараллелизм** – хотя в каждый конкретный момент времени процессор занят обработкой одной конкретной задачи, благодаря постоянному переключению с исполнения одной задачи на другую, достигается иллюзия параллельного исполнения нескольких задач. Во многопроцессорных системах задача максимально эффективного использования каждого конкретного процессора также решается путем переключения между процессами, однако тут, наряду с псевдопараллелизмом, имеет место и **действительный параллелизм**, когда на разных процессорах в один и тот же момент времени исполняются разные процессы.

Процессы, выполнение которых хотя бы частично перекрывается по времени, называются параллельными. Они могут быть независимыми и взаимодействующими. Независимые процессы – процессы, использующие независимое множество ресурсов; на результат работы такого процесса не должна влиять работа другого независимого процесса. Наоборот – взаимодействующие процессы совместно используют ресурсы и выполнение одного процесса может оказывать влияние на результат другого. Совместное использование ресурса двумя процессами, т.е. когда каждый из процессов полностью владеет ресурсом некоторое время, называют **разделением ресурса**. Разделению подлежат как аппаратные, так программные ресурсы. Разделяемые ресурсы, которые должны быть доступны в текущий момент времени только одному процессу – это так называемые **критические ресурсы**. Таковыми ресурсами могут быть как внешнее устройство, так и некая переменная, значение которой может изменяться разными процессами.

Процессы могут быть связаны некоторыми соотношениями (например, когда один процесс является прямым потомком другого), а могут быть не связанными друг с другом. Кроме того, процессы могут выполняться в разных узлах сети. Эти обстоятельства влияют на способ их взаимодействия, а именно – на возможность совместно использовать ресурсы, обмениваться информацией, оповещать друг друга о наступлении некоторых событий, а также определяют возможность одного процесса влиять на выполнение другого.

Таким образом, необходимо уметь решать две важнейшие задачи:

1. Распределение ресурсов между процессами.
2. Организация защиты ресурсов, выделенных определенному процессу, от неконтролируемого доступа со стороны других процессов.

Важнейшим требованием мультипрограммирования с точки зрения распределения ресурсов является следующее: результат выполнения процессов не должен зависеть от порядка переключения выполнения между процессами, т.е. от соотношения скорости выполнения данного процесса со скоростями выполнения других процессов.

В качестве примера ситуации, когда это правило нарушается, рассмотрим следующую. Пусть имеется некоторая простая функция, которая считывает символ, введенный с клавиатуры, и выводит его на экран:

```
void echo()
     {
          char in;
          input(in);
          output(in)
     }
```

В данном примере мы используем некоторые условные функции **`input()`** и **`output()`**, так как в данный момент для нас неважно, как конкретно реализован ввод/вывод в данной системе. Поскольку такой кусок кода будет использоваться практически в любой программе, его удобно сделать разделяемым, когда ОС загружает в некоторую область памяти, доступную всем процессам, одну-единственную копию данной программы, и все процессы используют эту копию совместно. Заметим, что в этом случае переменная **`in`** является разделяемой. Представим теперь ситуацию, изображенную на Рис. 2:

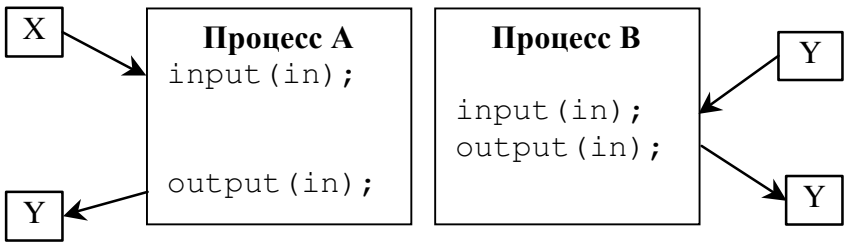

Рис. 2 Конкуренция процессов за ресурс.

1. Процесс **A** вызывает функцию **`echo(),`** однако в тот момент, когда входной символ был считан в переменную **`in`**, но до того, как он был выведен на экран, выполнение процесса прерывается и на выполнение загружается процесс **B.**
2. Процесс **B** тоже вызывает функцию **`echo().`** После выполнения функции **`echo()`** переменная **`in`** содержит уж новое значение, считанное с клавиатуры.
3. Процесс **A** возобновляет свою работу в той точке, в которой он был прерван, и выводит на экран символ, находящийся в переменной **`in.`**

В рассмотренном случае символ, считанный процессом **A**, был потерян, а символ, считанный процессом B, был выведен дважды. Результат выполнения процессов здесь зависит от того, в какой момент осуществляется переключение процессов, и от того, какой конкретно процесс будет выбран для выполнения следующим. Такие ситуации называются **гонками** (в англоязычной литературе - **race conditions**) между процессами, а процессы – конкурирующими. Единственный способ избежать гонок при использовании разделяемых ресурсов – контролировать доступ к любым разделяемым ресурсам в системе. При этом необходимо организовать **взаимное исключение** – т.е. такой способ работы с разделяемым ресурсом, при котором постулируется, что в тот момент, когда один из процессов работает с разделяемым ресурсом, все остальные процессы не могут иметь к нему доступ.

Проблему организации взаимного исключения можно сформулировать в более общем виде. Часть программы (фактически набор операций), в которой осуществляется работа с критическим ресурсом, называется **критической секцией**, или **критическим интервалом**. Задача взаимного исключения в этом случае сводится к тому, чтобы не допускать ситуации, когда два процесса одновременно находятся в критических секциях, связанных с одним и тем же ресурсом.

Заметим, что вопрос организации взаимного исключения актуален не только для взаимосвязанных процессов, совместно использующих определенные ресурсы для обмена информацией. Выше отмечалось, что возможна ситуация, когда процессы, не подозревающие о существовании друг друга, используют глобальные ресурсы системы, такие как, например, устройства ввода/вывода. В с этом случае имеет место конкуренция за ресурсы,

доступ к которым также должен быть организован по принципу взаимного исключения.

Важно отметить, что при организации взаимного исключения могут возникнуть две неприятные проблемы:

1. Возникновение так называемых **тупиков (deadlocks**). Рассмотрим следующую ситуацию (см. Рис. 3): имеются процессы **A** и **B**, каждому из которых в некоторый момент требуется иметь доступ к двум ресурсам $\mathbf{R_1}$ и $\mathbf{R_2}$**.** Процесс **A** получил доступ к ресурсу $\mathbf{R_1}$, и следовательно, никакой другой процесс не может иметь к нему доступ, пока процесс **A** не закончит с ним работать. Одновременно процесс **B** завладел ресурсом $\mathbf{R_2}$**.** В этой ситуации каждый из процессов ожидает освобождения недостающего ресурса, но оба ресурса никогда не будут освобождены, и процессы никогда не смогут выполнить необходимые действия.

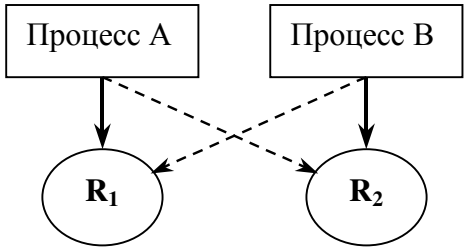

Рис. 3 Возникновение тупиковой ситуации.

2. Ситуация **блокирования** (дискриминации) одного из процессов, когда один из процессов будет бесконечно находиться в ожидании доступа к разделяемому ресурсу, в то время как каждый раз при его освобождении доступ к нему получает какой-то другой процесс.

Таким образом, любые средства организации взаимного исключения должны обеспечивать разрешение этих двух проблем. Помимо этого, к организации взаимного исключения выдвигаются следующие требования:

1. Не должно возникать ситуации, при которой процесс, находящийся *вне* своей критической секции, блокирует исполнение другого процесса.
2. Не должно делаться никаких предположений относительно взаимных скоростей исполнения

процессов, а также относительно количества и скоростей работы процессоров в системе.

Далее мы рассмотрим различные механизмы организации взаимного исключения для синхронизации доступа к разделяемым ресурсам и обсудим достоинства, недостатки и области применения этих подходов.

3.1 Способы реализации взаимного исключения.

3.1.1 Запрещение прерываний и специальные инструкции.

Простейшее умозаключение, которое, на первый взгляд, решает проблему синхронизации доступа к критическим ресурсам, заключается в следующем: некорректная работа с разделяемыми ресурсами возникает в том случае, если переключение процессов происходит в тот момент, когда выполняющийся процесс начал работу с разделяемым ресурсом, но не закончил ее, т.е. находится в своей критической секции. Чтобы этого не происходило, процесс должен запретить все прерывания непосредственно после входа в критическую секцию и разрешить их перед самым выходом из нее. Если прерывания запрещены, то переключения процессов не происходит, так как передача управления планировщику может быть реализована только с использованием прерываний (это может быть как прерывание по таймеру, так и другие прерывания, например, при заказе обмена). Таким образом, запрещение прерываний гарантирует процессу, что никакой другой процесс не обратится к разделяемому ресурсу, пока прерывания не будут разрешены.

Этот подход, однако, имеет ряд существенных недостатков. Первым из них является снижение общей производительности системы вследствие ограничения возможности переключения процессов для оптимальной загрузки процессора. Другая проблема заключается в том, что нет никаких гарантий, что процесс, запретивший прерывания, не зациклится в своей критической секции, тем самым приведя систему в полностью неработоспособное состояние. Цена программной ошибки здесь становится слишком высокой. Кроме всего прочего, для многопроцессорной системы запрещение прерываний не гарантирует даже взаимного исключения, так как запрещение прерываний на одном из процессоров никак не влияет на исполнение процессов на других процессорах ВС, и эти процессы по-прежнему имеют доступ к разделяемому ресурсу, например, ячейке общей памяти.

Для организации взаимного исключения на многопроцессорной системе с общей памятью могут использоваться

специальные машинные инструкции. Примером такой инструкции может служить **`TSL – test and set lock`**, реализующая чтение ячейки памяти и запись нового значения в ту же ячейку как единую операцию. При этом гарантируется, что совокупность операций чтения и записи ячейки памяти является неделимой, т.е. доступ к ячейке памяти со стороны других процессоров блокируется на все время исполнения инструкции. Вариацией этой идеи является специальная инструкция **`exchange`**, которая меняет местами содержимое регистра и ячейки памяти. Здесь опять гарантируется, что на все время выполнения инструкции доступ к ячейке памяти блокируется.

Используя эти инструкции и дополнительную разделяемую переменную, значение которой говорит о том, можно ли в данный момент войти в критическую секцию, процессы могут синхронизировать свою работу. Возможность считывать старое значение и записывать новое за одну операцию позволяет избежать ситуации, когда процесс, считав значение переменной, говорящее ему о том, что он может войти в свою критическую секцию, не успеет изменить ее значение на «занято» и будет в этот момент прерван; другой же процесс, получивший управление, считает неизмененное значение переменной, которое позволит и ему войти в критическую секцию, в результате чего два процесса одновременно окажутся в критических секциях.

3.1.2 Алгоритм Петерсона.

Другим подходом к решению задачи организации взаимного исключения является изобретение различных алгоритмов, позволяющих процессам синхронизировать свою работу программно. Одним из первых программных решений является алгоритм Петерсона.

Алгоритм Петерсона для случая 2х процессов представлен на Рис. 4.

```
int first_process() {
  for(;;) {
    flag[0] = 1;
    turn = 1;
    while (flag[1] && turn) {
        /* waiting */
    };
    /* critical section */
    flag[0] = 0;
  }
}
```

```
int second_process() {
  for(;;) {
    flag[1] = 1;
    turn = 1;
    while (flag[0] && (!turn)){
        /* waiting */
    };
    /* critical section */
    flag[1] = 0;
  }
}
```

Рис. 4 Алгоритм Петерсона

Элементы массива **`flag`** символизируют собой желание соответствующего процесса попасть в критическую секцию. Каждый из процессов видит все элементы массива **`flag`**, но модифицирует только «свой» элемент. Переменная **`turn`** устанавливает очередность прохождения критической секции при обоюдном желании процессов вступить в нее. Благодаря тому, что каждый из процессов прежде всего устанавливает флаг очередности в пользу другого процесса, исключается ситуация «монополизации» ресурса одним из процессов и вечного блокирования второго.

3.1.3 Активное ожидание.

Общим недостатком рассмотренных выше решений является то, что процесс, желающий получить доступ к занятому ресурсу, блокируется в состоянии так называемого **активного ожидания** (в англоязычной литературе – **busy waiting**), т.е. в цикле постоянно проверяет, не наступило ли условие, при котором он сможет войти в критическую секцию. Использование активного ожидания приводит к бесполезному расходованию процессорного времени и как следствие, снижению общей производительности системы. Кроме того, при некоторых стратегиях планирования времени ЦП в целом корректный алгоритм с активным ожиданием может привести к тупику. Примером может служить стратегия планирования с приоритетами, когда процесс, имеющий больший приоритет, загружается на выполнение и переходит в состояние активного ожидания, в то время как процесс с меньшим приоритетом захватил необходимый ресурс, но был выгружен до того, как освободил его. Поскольку процесс с большим приоритетом не блокирован, а готов к продолжению выполнения, переключение процессов не происходит, и процесс, владеющий ресурсом, никогда не сможет его освободить.

Далее мы рассмотрим ряд подходов, которые позволяют реализовать взаимное исключение без использования активного ожидания. Основная идея всех этих подходов в том, чтобы блокировать ожидающий процесс, вместо того, чтобы оставлять его в состоянии активного ожидания, а затем, при наступлении возможности использования нужного ресурса, разблокировать один из ранее заблокированных процессов. Для этого, однако, требуется определенная поддержка со стороны ОС, так как именно в ее ведении находится переключение состояний процессов.

3.1.4 Семафоры.

Первый из таких подходов был предложен Дейкстрой в 1965 г. Дейкстра предложил новый тип данных, именуемый **семафором**. Семафор представляет собой переменную целого типа, над которой определены две операции: **`down(P)`** и **`up(V)`**.[2] 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 постулируется, что увеличение значения семафора и, возможно, разблокирование одного из процессов и уменьшение значения являются атомарной неделимой операцией.

Чтобы прояснить смысл использования семафоров для синхронизации, можно привести простую аналогию из повседневной жизни. Представим себе супермаркет, посетители которого, прежде чем войти в торговый зал, должны обязательно взять себе инвентарную тележку. В момент открытия магазина на входе имеется **N** свободных тележек – это начальное значение семафора. Каждый посетитель забирает одну из тележек (уменьшая тем самым количество оставшихся на **1**) и проходит в торговый зал – это аналог операции **`down`**. При выходе посетитель возвращает тележку на

---

[2] Оригинальные обозначения P и V, данные Дейкстрой и получившие широкое распространение в литературе, являются сокращениями голландских слов *proberen* – проверить и *verhogen* – увеличить

место, увеличивая тележек на **1** – это аналог операции **up**. Теперь представим себе, что очередной посетитель обнаруживает, что свободных тележек нет – он вынужден *блокироваться* на входе в ожидании появления тележки. Когда один из посетителей, находящихся в торговом зале, покидает его, посетитель, ожидающий тележку, *разблокируется***,** забирает тележку и проходит в зал. Таким образом, наш семафор в виде тележек позволяет находиться в торговом зале (*аналоге критической секции*) не более чем **N** посетителям одновременно. Положив **N = 1**, получим реализацию взаимного исключения. Семафор, начальное (и максимальное) значение которого равно **1**, называется **двоичным семафором** (так как имеет только **2** состояния: **0** и **1**). Использование двоичного семафора для организации взаимного исключения проиллюстрировано на Рис. 5.

процесс 1

```
int semaphore;
…
down(semaphore);
/*    критическая секция
      процесса 1 */
…
up(semaphore);
…
```

процесс 2

```
int semaphore;
…
down(semaphore);
/*    критическая секция
      процесса 2 */
…
up(semaphore);
…
```

Рис. 5 Взаимное исключение с использованием семафора

Семафоры представляют собой мощное средство синхронизации, однако программирование с использованием семафоров является достаточно тяжелой задачей, причем незаметная на первый взгляд логическая ошибка может привести к образованию тупиковых ситуаций или нарушению условий синхронизации.

С целью облегчить написание корректных программ были предложены более высокоуровневые средства синхронизации, которые мы рассмотрим далее.

3.1.5 Мониторы.

Идея **монитора** была впервые сформулирована в 1974 г. Хоаром. В отличие от других средств, монитор представляет собой *языковую* конструкцию, т.е. некоторое средство, предоставляемое языком программирования и поддерживаемое компилятором. Монитор представляет собой совокупность процедур и структур данных, объединенных в программный модуль специального типа. Постулируются три основных свойства монитора:

1. Структуры данных, входящие в монитор, могут быть доступны только для процедур, входящих в этот монитор (таким образом, монитор представляет собой некоторый аналог объекта в объектно-ориентированных языках и реализует инкапсуляцию данных)
2. Процесс «входит» в монитор путем вызова одной из его процедур
3. В любой момент времени внутри монитора может находиться не более одного процесса. Если процесс пытается попасть в монитор, в котором уже находится другой процесс, он блокируется. Таким образом, чтобы защитить разделяемые структуры данных, их достаточно поместить внутрь монитора вместе с процедурами, представляющими критические секции для их обработки.

Подчеркнем, что монитор представляет собой конструкцию языка программирования, и следовательно, компилятору известно о том, что входящие в него процедуры и данные имеют особую семантику, поэтому первое условие может проверяться еще на этапе компиляции. Кроме того, код для процедур монитора тоже может генерироваться особым образом, чтобы удовлетворялось третье условие. Поскольку организация взаимного исключения в данном случае возлагается на компилятор, количество программных ошибок, связанных с организацией взаимного исключения, сводится к минимуму.

Дополнительная синхронизация: переменные-условия.

Помимо обычных структур данных, мониторы могут включать в себя специальные **переменные-условия**, на которых определены операции **`wait`** и **`signal`.** Они используются для синхронизации. Если процесс, находящийся внутри монитора (т.е. исполняющий одну из его процедур), обнаруживает, что логически он не может продолжать выполнение, пока не выполнится определенное условие (например, буфер для записи данных переполнился), он вызывает операцию **`wait`** над определенной переменной-условием. При этом его дальнейшее выполнение блокируется, и это позволяет другому процессу, ожидающему входа в монитор, попасть в него. В дальнейшем, если этот другой процесс произведет некоторые действия, которые приведут к изменению обстоятельств (в нашем примере – считает часть данных из буфера), он должен вызвать для соответствующей переменной-условия операцию **`signal`**, что позволит разблокировать ожидающий процесс. Тонкость

заключается в том, что разблокированный процесс, как и тот, кто его разблокировал, должен оказаться внутри монитора, но нахождение двух процессов внутри монитора одновременно невозможно по определению. Хоар постулировал, что в этом случае процесс, вызвавший **`signal`**, приостанавливается. Хансен в своей модификации мониторов в 1975 г. предложил более простое дополнительное условие: вызов **`signal`** должен быть самым последним внутри процедуры монитора, чтобы процесс немедленно после его выполнения покинул монитор. Заметим, что переменные-условия используются в мониторах не для организации взаимного исключения (оно постулируется самим определением монитора), а для дополнительной синхронизации процессов. В нашем примере разделяемый ресурс – буфер для чтения/записи охраняется от одновременного доступа по чтению и по записи самим монитором, а переменная-условие предохраняет пишущий процесс от затирания ранее записанных данных.

Несомненным достоинством мониторов является то, что взаимное исключение здесь организуется автоматически, что существенно упрощает программирование и снижает вероятность ошибок. Недостатком же является то, что, как уже говорилось, монитор – это языковая конструкция. Следовательно, если язык программирования не содержит таких конструкций (а для большинства распространенных языком это так и есть), программист не может ею воспользоваться. В то же время семафоры, например, являются средством ОС, и если соответствующая ОС поддерживает семафоры, программист может их использовать независимо от того, на каком языке он пишет программы. Мониторы реализованы в некоторых языках программирования, таких как Concurrent Euclid, Concurrent Pascal, Modula-2, Modula-3, однако эти языки не слишком распространены.

3.1.6 Обмен сообщениями.

Общей проблемой и для мониторов, и для семафоров является то, что их реализация существенно опирается на предположение, что мы имеем дело либо с однопроцессорной системой, либо с многопроцессорной системой, где все процессоры имеют доступ к общей памяти. Однако в случае распределенной системы, где каждый процессор имеет прямой доступ только к своей памяти, такие средства не подходят. Более общим средством, решающим проблему синхронизации как для однопроцессорных систем и систем с общей памятью, так и для распределенных, является **обмен сообщениями**.

Обмен сообщениями представляет собой средство, которое может быть использовано как для синхронизации, в частности для организации взаимного исключения, так и для обмена информацией между взаимосвязанными процессами, выполняющими общую работу. Рассмотрим общую концепцию обмена сообщениями. Основная функциональность реализуется двумя примитивами, реализующими, соответственно, посылку и прием сообщения:

```
send(destination, message)
receive(source, message)
```

Как и семафоры, и в отличие от мониторов, эти примитивы являются системными вызовами, а не конструкциями языка.

Рассмотрим основные особенности, которыми может обладать та или иная система обмена сообщениями.

Синхронизация.

Сам смысл обмена сообщениями предполагает определенную синхронизацию между процессом-отправителем и процессом-получателем, так как сообщение не может быть получено до того, как оно послано. Возникает вопрос, что происходит, если один процесс хочет получить сообщение, а другой его не отослал, и наоборот, если один процесс отсылает сообщение, а другой не собирается его получать. Здесь есть две возможности. Как операция посылки сообщения, так операция приема могут быть **блокирующими** и **неблокирующими.** Для операции **`send`** это означает, что либо процесс-отправитель может блокироваться до тех пор, пока получатель не вызовет **`receive`**, либо выполнение процесса может продолжаться далее независимо от наличия получателя. Для операции **`receive`** подобная ситуация возникает, когда эта операция вызвана раньше, чем сообщение было послано – в этом случае она может либо блокироваться до получения сообщения, либо возвращать управление сразу же.

В зависимости от целей использования механизма сообщений могут быть полезны различные комбинации этих условий:

- Блокирующий **`send`** и блокирующий **`receive`** – эта схема известна под названием «схемы рандеву». Она не требует буферизации сообщений и часто используется для синхронизации процессов
- Неблокирующий **`send`** и блокирующий **`receive`** – такая схема очень распространена в системах клиент/сервер: серверный процесс блокируется в ожидании очередного запроса для обработки, в то время как клиент,

пославший запрос серверу, может продолжать выполняться, не ожидая окончания обработки своего запроса

- Также весьма распространена схема, когда обе операции являются неблокирующими – в этом случае оба процесса могут продолжать выполнение, не дожидаясь окончания коммуникации

Важно понимать, что в случае, если **`send`** является неблокирующим, процесс-отправитель не может знать, получено ли его сообщение. В этом случае, если требуется организовать гарантированную доставку сообщений, необходимо, чтобы процессы обменивались сообщениями-подтверждениями. Проблема потери сообщений встает также, если используется блокирующий **`receive`** – в этом случае процесс-получатель может оказаться заблокированным навечно. Поэтому в такую схему часто добавляется дополнительный примитив, позволяющий процессу-получателю проверить, есть ли для него сообщение, но не блокироваться, если его нет.

Адресация.

Другая важная проблема – организовать адресацию сообщений. Одно из решений – так называемая **прямая адресация**, при которой каждому из процессов присваивается некоторый идентификатор, и сообщения адресуются этим идентификаторам. При этом процесс-получатель может указать явно идентификатор отправителя, от которого он желает получить сообщение, либо получать сообщения от любого отправителя.

Иное решение заключается в том, чтобы предоставить специальную структуру данных – **почтовый ящик,** или **очередь сообщений**, которая по сути своей является буфером, рассчитанным на определенное количество сообщений. В этом случае сообщения адресуются не процессам, а почтовым ящикам, при этом один и тот же ящик может использоваться и несколькими отправителями, и несколькими получателями. Такая схема, называемая **косвенной адресацией**, обеспечивает дополнительную гибкость. Заметим, что связь между процессом-получателем или отправителем и почтовым ящиком может быть не только статической (т.е. раз навсегда заданной при создании ящика), но и динамической; в последнем случае для установления и разрыва связи используются дополнительные примитивы (**`connect/disconnect`**). Кроме того, поскольку почтовый ящик является самостоятельным объектом,

необходимо наличие примитивов создания и удаления ящика (**`create/destroy`**).

Длина сообщения.

Немаловажным аспектом является формат сообщений. В той или иной системе могут допускаться как сообщения фиксированной длины, так и переменной. В последнем случае в заголовке сообщения, помимо отправителя и получателя, должна указываться длина сообщения. Выбор того или иного варианта зависит от целей, которые преследует система обмена сообщениями, и от предполагаемой архитектуры ВС. Так, если предполагается возможность передачи больших объемов данных, то сообщения с переменной длиной будут более гибким решением и позволят сократить накладные расходы на большое количество коротких сообщений, где значительную часть занимает сам заголовок. С другой стороны, если обмен происходит между процессами на одной машине, немаловажную роль играет эффективность. Здесь, возможно, было бы уместно ограничить длину сообщения, с тем, чтобы использовать для их передачи системные буфера с быстрым доступом.

В заключение отметим еще раз, что механизм обмена сообщениями является мощным и гибким средством синхронизации, пригодным для использования как на однопроцессорных системах и системах с общей памятью, так и в распределенных ВС. Однако, по сравнению с семафорами и мониторами, он, как правило, является менее быстрым.

3.2 Классические задачи синхронизации процессов.

3.2.1 «Обедающие философы»

Рассмотрим одну из классических задач, демонстрирующих проблему разделения доступа к критическим ресурсам – «задачу об обедающих философах» [2]. Данная задача иллюстрирует ситуацию, когда процессы конкурируют за право исключительного доступа к ограниченному числу ресурсов. Пять философов собираются за круглым столом, перед каждым из них стоит блюдо со спагетти, и между каждыми двумя соседями лежит вилка. Каждый из философов некоторое время размышляет, затем берет две вилки (одну в правую руку, другую в левую) и ест спагетти, затем кладет вилки обратно на стол и опять размышляет и так далее. Каждый из них ведет себя независимо от других, однако вилок запасено ровно столько, сколько философов, хотя для еды каждому из них нужно две. Таким образом, философы должны совместно использовать

имеющиеся у них вилки (ресурсы). Задача состоит в том, чтобы найти алгоритм, который позволит философам организовать доступ к вилкам таким образом, чтобы каждый имел возможность насытиться, и никто не умер с голоду.

Рассмотрим простейшее решение, использующее семафоры. Когда один из философов хочет есть, он берет вилку слева от себя, если она в наличии, а затем - вилку справа от себя. Закончив есть, он возвращает обе вилки на свои места. Данный алгоритм может быть представлен следующим способом:

```
#define N 5                     /* число философов*/
void philosopher (int i)        /* i – номер философа от 0
до 4*/
{
     while (TRUE)
     {
          think();                  /*философ думает*/
          take_fork(i);             /*берет левую вилку*/
          take_fork((i+1)%N);       /*берет правую вилку*/
          eat();                    /*ест*/
          put_fork(i);              /*кладет обратно левую
     вилку*/
          put_fork((i+1)%N);        /*    кладет    обратно
     правую вилку */
     }
}
```

Функция **take_fork()** описывает поведение философа по захвату вилки: он ждет, пока указанная вилка не освободится, и забирает ее.

На первый взгляд, все просто, однако, данное решение может привести к тупиковой ситуации. Что произойдет, если все философы захотят есть в одно и то же время? Каждый из них получит доступ к своей левой вилке и будет находиться в состоянии ожидания второй вилки до бесконечности.

Другим решением может быть алгоритм, который обеспечивает доступ к вилкам только четырем из пяти философов. Тогда всегда среди четырех философов по крайней мере один будет иметь доступ к двум вилкам. Данное решение не имеет тупиковой ситуации. Алгоритм решения может быть представлен следующим образом:

```
# define N 5                    /* количество философов */
# define LEFT (i-1)%N           /* номер легого соседа для i-ого
                                 философа */
# define RIGHT (i+1)%N          /* номер правого соседа для i-
                                 ого философа*/
# define THINKING 0                  /* философ думает */
# define HUNGRY 1                    /* философ голоден */
# define EATING 2                    /* философ ест */

typedef int semaphore;          /* тип данных «семафор» */
int state[N];                   /* массив состояний философов */
semaphore mutex=1;  /* семафор для критической секции */
semaphore s[N];     /* по одному семафору на философа */

void philosopher (int i)
          /* i : номер философа от 0 до N-1 */
{
     while (TRUE)                   /* бесконечный цикл */
     {
       think();                     /* философ думает */
       take_forks(i);               /*философ берет обе вилки
                                    или блокируется */
       eat();                       /* философ ест */
       put_forks(i);                /* философ освобожает обе
                                    вилки */
     }
}

void take_forks(int i)
          /* i : номер философа от 0 до N-1 */
{
     down(&mutex);              /* вход в критическую секцию */
     state[i] = HUNGRY;         /*записываем, что i-ый  философ
                                голоден */
     test(i);           /* попытка взять обе вилки */
     up(&mutex);        /* выход из критической секции */
     down(&s[i]);       /* блокируемся, если вилок нет */
```

```
}

void put_forks(i)
          /* i : номер философа от 0 до N-1 */
{
     down(&mutex);        /* вход в критическую секцию */
     state[i] = THINKING;      /* философ закончил есть */
     test(LEFT);
     /* проверить может ли левый сосед сейчас есть */
     test(RIGHT);
     /* проверить может ли правый сосед сейчас есть*/
     up(&mutex);     /* выход из критической секции */
}

void test(i)
          /* i : номер философа от 0 до N-1 */
{
     if (state[i] == HUNGRY && state[LEFT] != EATING &&
state[RIGHT] != EATING)
     {
        state[i] = EATING;
        up (&s[i]);
     }
}
```

3.2.2 Задача «читателей и писателей»

Другой классической задачей синхронизации доступа к ресурсам является задача «читателей и писателей» [2], иллюстрирующая широко распространенную модель совместного доступа к данным. Представьте себе ситуацию, например, в системе резервирования билетов, когда множество конкурирующих процессов хотят читать и обновлять одни и те же данные. Несколько процессов могут читать данные одновременно, но когда один процесс начинает записывать данные (обновлять базу данных проданных билетов), ни один другой процесс не должен иметь доступ к данным, даже для чтения. Возникает вопрос, как спланировать работу такой системы? Одно из решений представлено ниже:

```
typedef int semaphore;    /* тип данных «семафор» */
semaphore mutex = 1;      /* контроль за доступом к «rc» (разделямый ресурс) */
semaphore db = 1;         /* контроль за доступом к базе данных */
int rc = 0;               /* кол-во процессов читающих или пишущих */

void reader (void)
{
     while (TRUE)                 /* бесконечный цикл */
     {
          down(&mutex);           /* получить эксклюзивный доступ к «rc»*/
          rc = rc + 1;            /* еще одним читателем больше */
          if (rc == 1) down(&db); /* если это первый читатель, нужно заблокировать эксклюзивный доступ к базе */
          up(&mutex);             /*освободить ресурс rc */
          read_data_base();       /* доступ к данным */
          down(&mutex);           /*получить эксклюзивный доступ к «rc»*/
          rc = rc - 1:            /* теперь одним читателем меньше */
          if (rc == 0) up(&db);   /*если это был последний читатель, разблокировать эксклюзивный доступ к базе данных */
          up(&mutex);             /*освободить разделяемый ресурс rc */
          use_data_read();        /* некритическая секция */
     }
}

void writer (void)
{
```

```
    while(TRUE)                 /* бесконечный цикл */
    {
        think_up_data();        /* некритическая секция */
        down(&db);              /* получить эксклюзивный
                                   доступ к данным*/
        write_data_base();      /* записать данные */
        up(&db);                /* отдать эксклюзивный
                                   доступ */
    }
}
```

В этом примере, первый процесс, обратившийся к базе данных по чтению, осуществляет операцию **down()** над семафором **db**, тем самым блокируя эксклюзивный доступ к базе, который нужен для записи. Число процессов, осуществляющих чтение в данный момент, определяется переменной **rc** (обратите внимание! Так как переменная **rc** является разделяемым ресурсом – ее изменяют все процессы, обращающиеся к базе данных по чтению – то доступ к ней охраняется семафором **mutex**). Когда читающий процесс заканчивает свою работу, он уменьшает значение **rc** на единицу. Если он является последним читателем, он также совершает операцию **up** над семафором **db**, тем самым разрешая заблокированному писателю, если таковой имелся, получить эксклюзивный доступ к базе для записи.

Надо заметить, что приведенный алгоритм дает преимущество при доступе к базе данных процессам-читателям, так как процесс, ожидающий доступа по записи, будет ждать до тех пор, пока все читающие процессы не окончат работу, и если в это время появляется новый читающий процесс, он тоже беспрепятственно получит доступ. Это может привести к неприятной ситуации в том случае, если в фазе, когда ресурс доступен по чтению, и имеется ожидающий процесс-писатель, будут появляться новые и новые читающие процессы. К примеру, представим себе, что новый читающий процесс появляется каждую секунду и чтение длится в среднем 2 секунды. Количество читающих процессов в этом случае будет приблизительно константным, и процесс-писатель никогда не получит доступ к данным.

Чтобы этого избежать, можно модифицировать алгоритм таким образом, чтобы в случае, если имеется хотя бы один ожидающий процесс-писатель, новые процессы-читатели не получали доступа к ресурсу, а ожидали, когда процесс-писатель

обновит данные. В этой ситуации процесс-писатель должен будет ожидать окончания работы с ресурсом только тех читателей, которые получили доступ раньше, чем он его запросил. Однако, обратная сторона данного решения в том, что оно несколько снижает производительность процессов-читателей, так как вынуждает их ждать в тот момент, когда ресурс не занят в эксклюзивном режиме.

3.2.3 Задача о «спящем парикмахере»

Еще одна классическая задача на синхронизацию процессов – это так называемая «задача о спящем парикмахере» [2]. Рассмотрим парикмахерскую, в которой работает один парикмахер, имеется одно кресло для стрижки и несколько кресел в приемной для посетителей, ожидающих своей очереди. Если в парикмахерской нет посетителей, парикмахер засыпает прямо на своем рабочем месте. Появившийся посетитель должен его разбудить, в результате чего парикмахер приступает к работе. Если в процессе стрижки появляются новые посетители, они должны либо подождать своей очереди, либо покинуть парикмахерскую, если в приемной нет свободного кресла для ожидания. Задача состоит в том, чтобы корректно запрограммировать поведение парикмахера и посетителей.

Одно из возможных решений этой задачи представлено ниже. Процедура **`barber()`** описывает поведение парикмахера (она включает в себя бесконечный цикл – ожидание клиентов и стрижку). Процедура **`customer()`** описывает поведение посетителя. Несмотря на кажущуюся простоту задачи, понадобится целых 3 семафора: **`customers`** – подсчитывает количество посетителей, ожидающих в очереди, **`barbers`** – обозначает количество свободных парикмахеров (в случае одного парикмахера его значения либо 0, либо 1) и **`mutex`** – используется для синхронизации доступа к разделяемой переменной **`waiting`**. Переменная **`waiting`**, как и семафор **`customers`**, содержит количество посетителей, ожидающих в очереди, она используется в программе для того, чтобы иметь возможность проверить, имеется ли свободное кресло для ожидания, и при этом не заблокировать процесс, если кресла не окажется. Заметим, что как и в предыдущем примере, эта переменная является разделяемым ресурсом, и доступ к ней охраняется семафором **`mutex`**. Это необходимо, так как для обычной переменной, в отличие от семафора, чтение и последующее изменение не являются неделимой операцией.

```
#define CHAIRS 5
typedef int semaphore;   /* тип данных «семафор» */
```

```
semaphore customers = 0;        /*          посетители,
                                ожидающие в очереди */
semaphore barbers = 0;          /*          парикмахеры,
                            ожидающие посетителей */
semaphore mutex = 1;        /* контроль за доступом к
                            переменной waiting */
int waiting = 0;
void barber()
{
while (true) {
     down(customers);     /* если customers == 0,
                т.е.      посетителей      нет,      то
                заблокируемся до появления посетителя
                */
     down(mutex);         /*    получаем    доступ    к
                            waiting */
     waiting = wating - 1;     /*   уменьшаем   кол-во
                ожидающих клиентов */
     up(barbers);         /*    парикмахер    готов    к
                        работе */
     up(mutex);           /*     освобождаем     ресурс
                        waiting */
     cut_hair();          /* процесс стрижки */
}
void customer()
{
     down(mutex);    /* получаем доступ к waiting */
     if (waiting < CHAIRS)     /*   есть   место   для
                                ожидания */
     {
          waiting = waiting + 1; /* увеличиваем кол-
                            во ожидающих клиентов */
          up(customers);       /*   если   парикмахер
                            спит, это его разбудит */
          up(mutex);           /* освобождаем ресурс
                                waiting */
          down(barbers); /* если парикмахер занят,
                            переходим     в     состояние
                            ожидания, иначе - занимаем
                            парикмахера*/
          get_haircut();       /* процесс стрижки */
```

```
        }
        else
        {
          up(mutex);          /* нет свободного кресла
                                 для ожидания – придется
                                 уйти */
        }
    }
```

ЧАСТЬ II. РЕАЛИЗАЦИЯ ПРОЦЕССОВ.

В этой части пособия мы подробно рассмотрим практическое применение понятия процесса в ОС, а также реализацию механизма управления процессами на примере ОС UNIX.

4 Реализация процессов в ОС UNIX

4.1 Понятие процесса в UNIX.

Выше уже говорилось, что в каждой конкретной ОС существует свое системно-ориентированное определение понятия процесса. В ОС UNIX процесс можно определить, с одной стороны, как единицу управления и потребления ресурсов, с другой стороны – как объект, зарегистрированный в **таблице процессов** ядра UNIX. Каждому процессу в UNIX сопоставлено некое уникальное целое число, называемое **идентификатором процесса – PID**. Это число находится в диапазоне от нуля до некоторого предельного номера, характеризующего максимально возможное количество одновременно существующих процессов в данной ОС. Некоторые значения идентификаторов являются зарезервированными и назначаются специальным процессам ОС, например, процесс с PID=0 ассоциируется с работой ядра ОС, а процесс с PID=1 – это процесс init, работа которого будет подробно рассмотрена ниже.

4.1.1 Контекст процесса.

С точки зрения организации данных ядра ОС, идентификатор процесса фактически представляет собой номер записи в таблице процессов, соответствующей данному процессу. Содержимое записи таблицы процессов позволяет получить доступ к **контексту процесса** (а именно, часть информации, составляющей контекст процесса, хранится непосредственно в таблице процессов, а на структуры данных, содержащие оставшуюся часть контекста, в записи таблицы процессов имеются прямые или косвенные ссылки). Таблица процессов поддерживается ядром UNIX и находится в адресном пространстве ядра.

С точки зрения логической структуры контекст процесса в UNIX состоит из:

- **пользовательской** составляющей или **тела процесса** (иногда используется термин «пользовательский контекст»)
- **аппаратной** составляющей (иногда используется термин «аппаратный контекст»)

- **системной** составляющей ОС UNIX (иногда называемой «системным контекстом» или «контекстом системного уровня»)

Иногда при рассмотрении контекста процесса два последних компонента объединяют, в этом случае используется термин **общесистемная** составляющая контекста.

Рассмотрим подробнее каждую из составляющих контекста процесса.

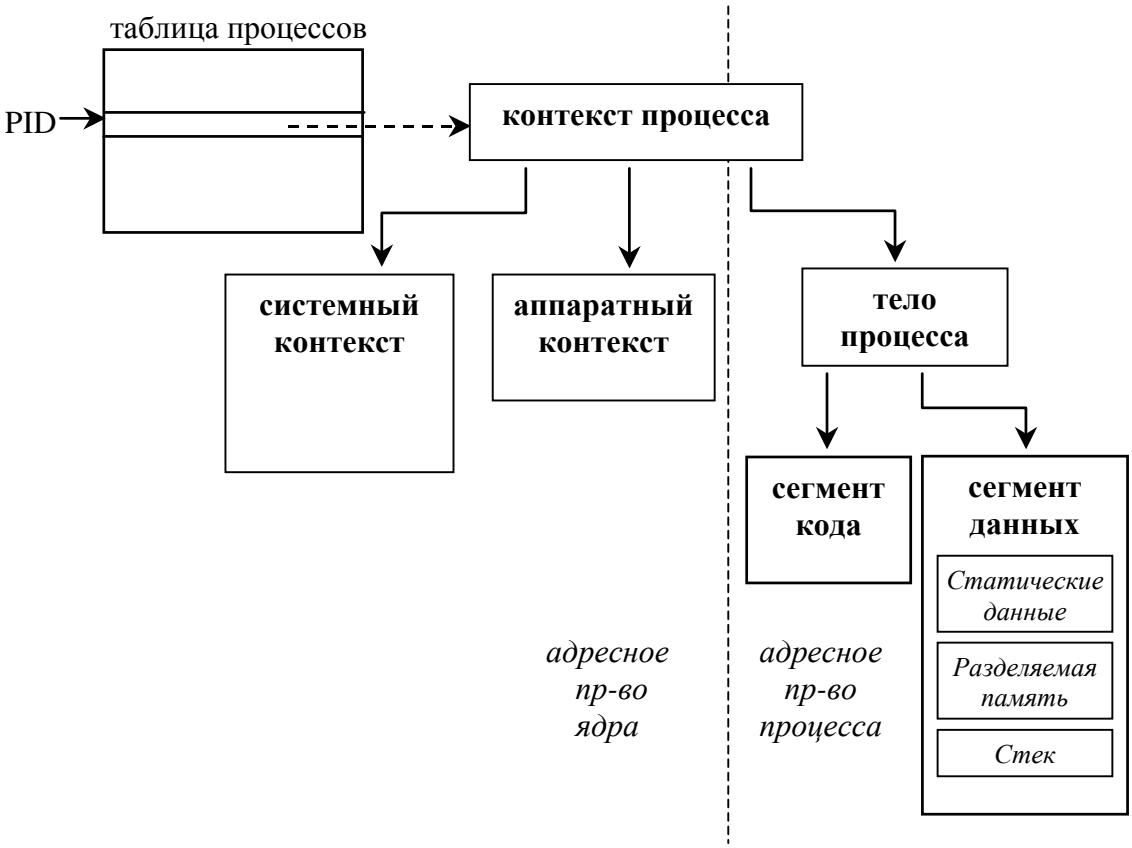

Рис. 6 Контекст процесса

4.1.2 Тело процесса.

Тело процесса состоит из сегмента кода и сегмента данных[3].

Сегмент кода содержит машинные команды и неизменяемые константы соответствующей процессу программы. Данные в этом сегменте не подлежат изменению.

Сегмент данных содержит данные, динамически изменяемые в ходе выполнения процесса. Сегмент данных содержит область

[3] Под сегментом здесь мы понимаем область памяти, которой система управляет как единым целым

статических переменных, область разделяемой с другими процессами памяти, а также область стека (обычно эта область служит основой для организации автоматических переменных, передачи параметров в функции, организацию динамической памяти).

Некоторые современные ОС имеют возможность разделения единого сегмента кода между разными процессами. Тем самым достигается экономия памяти в случаях одновременного выполнения идентичных процессов. Например, при функционировании терминального класса одновременно могут быть сформированы несколько копий текстового редактора. В этом случае сегмент кода у всех процессов, соответствующих редакторам, будет единый, а сегменты данных будут у каждого процесса свои.

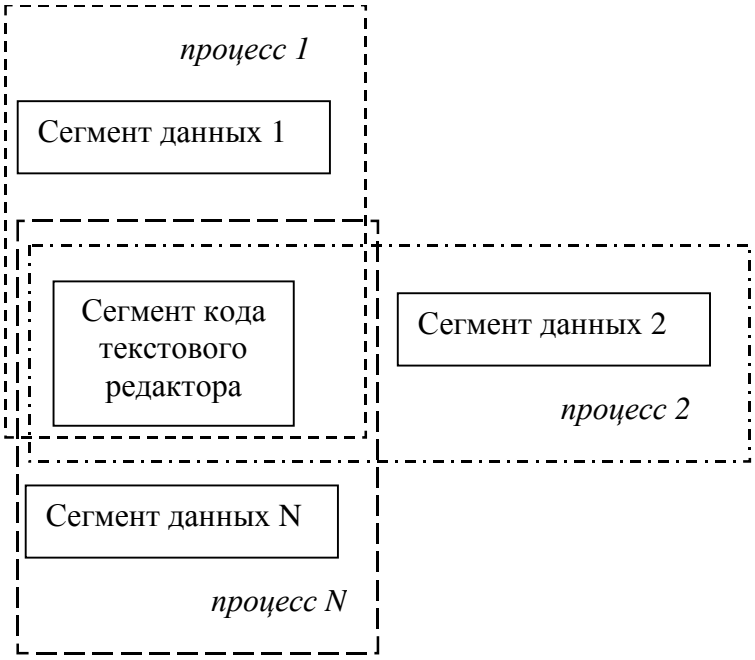

Рис. 7 Разделение сегмента кода разными экземплярами программы

Следует отметить, что при использовании динамически загружаемых библиотек возможно разделение сегмента кода на неизменную часть, которая может разделяться между процессами и часть, соответствующую изменяемому в динамике коду подгружаемых программ.

4.1.3 Аппаратный контекст.

Аппаратная составляющая содержит все регистры и аппаратные таблицы ЦП, используемые активным или исполняемым процессом (счетчик команд, регистр состояния процессора, аппарат виртуальной памяти, регистры общего назначения и т. д.). Аппаратная составляющая контекста доступна только в тот момент,

когда процесс находится в состоянии выполнения. В тот момент, когда процесс переходит из состояния выполнения в какое-либо другое состояние, необходимые данные из аппаратной составляющей его контекста сохраняются в область системной составляющей контекста, и до тех пор, пока процесс снова не перейдет в состояние выполнения, в его контексте аппаратной составляющей не будет.

4.1.4 Системный контекст.

В системной составляющей контекста процесса содержатся различные атрибуты процесса, такие как:

- идентификатор родительского процесса
- текущее состояние процесса
- приоритет процесса
- реальный и эффективный идентификатор пользователя-владельца
- реальный и эффективный идентификатор группы, к которой принадлежит владелец
- список областей памяти
- таблица открытых файлов процесса
- диспозиция сигналов, т.е. информация о том, какая реакция установлена на тот или иной сигнал
- информация о сигналах, ожидающих доставки в данный процесс
- сохраненные значения аппаратной составляющей (когда выполнение процесса приостановлено)

Поясним сказанное выше.

Реальные и эффективные идентификаторы пользователя и группы. Как правило при формировании процесса эти идентификаторы совпадают и равны реальному идентификатору пользователя и реальному идентификатору группы, т.е. они определяются персонификацией пользователя, сформировавшего данный процесс. При этом права процесса по доступу к файловой системе определяются правами сформировавшего процесс пользователя и его группы. Этого бывает недостаточно. Примером может служить ситуация, когда пользователь желает запустить некоторый процесс, изменяющий содержимое файлов, которые не принадлежать этому пользователю (например, изменение пароля на

доступ пользователя в систему). Для разрешения данной ситуации имеется возможность установить специальный признак в исполняемом файле, наличие которого позволяет установить в процессе, сформированном при запуске данного файла в качестве эффективных идентификаторов, идентификатор владельца и группы владельца этого файла.

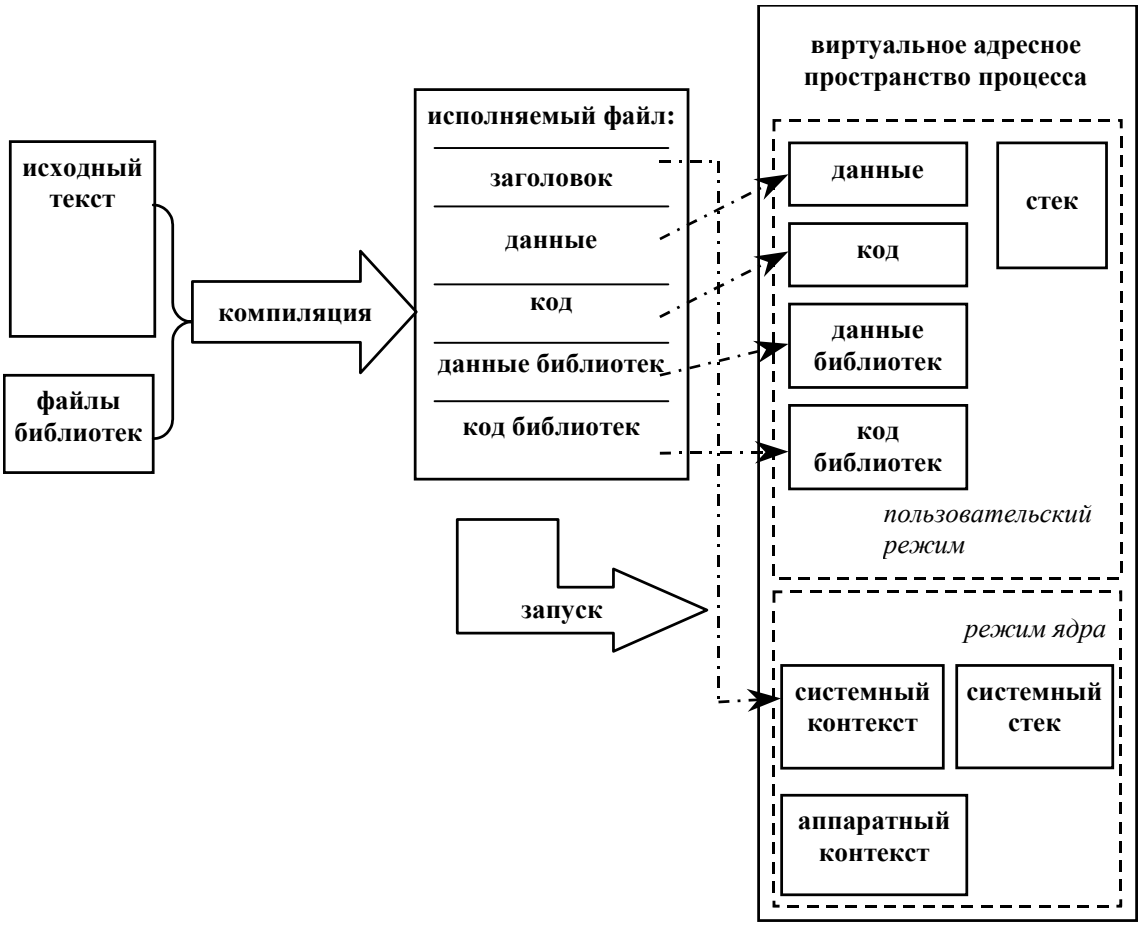

Рис. 8 Формирование контекста процесса.

4.2 Аппарат системных вызов в ОС UNIX.

Как известно, одной из основных функций любой ОС является управление ресурсами. Вынесение непосредственного доступа к ресурсам в зону ответственности ядра необходимо для того, чтобы обеспечить надежность и работоспособность всей вычислительной системы, так как невозможно гарантировать, что пользовательский процесс, получив непосредственный доступ к ресурсам вычислительной системы, будет работать с ними корректно. Кроме того, в многозадачной системе имеет место конкуренция процессов за ресурсы, и ОС должна здесь выполнять также функцию планирования доступа к ресурсам и защиты ресурсов, выделенных

конкретному процессу, от несанкционированного доступа со стороны других процессов.

Чтобы обеспечить гарантии того, что определенные действия, такие как операции с ресурсами, планирование процессов и т.п., может выполнять только ОС, вычислительная система должна обладать определенными свойствами, и в частности, иметь **привилегированный режим выполнения**. Это означает, что в ВС имеется два режима выполнения: обычный (пользовательский) и привилегированный (иногда называемый также режимом ядра, или защищенным режимом). Существует набор операций (инструкций), которые не могут быть выполнены процессом, работающим в пользовательском режиме. Они доступны только в привилегированном режиме, в котором работает ядро ОС. Кроме того, процессу, работающему в пользовательском режиме, недоступно адресное пространство других процессов и адресное пространство ядра.

Итак, обычные процессы выполняются в пользовательском режиме, и им недоступны те операции, которые может выполнять ядро ОС, работающее в привилегированном режиме, в частности, непосредственный доступ к ресурсам. Каким же образом обычный процесс, работающий в пользовательском режиме, может все же получить возможность работать с ресурсами ВС, например, записывать данные в файл или выводить их на печать? Для обеспечения такой возможности вводится аппарат **системных вызовов,** посредством которых ядро предоставляет процессам определенный набор услуг.

С точки зрения пользовательского процесса, системные вызовы оформлены аналогично библиотечным функциям, и обращение к ним при программировании ничем не отличается от вызова обычной функции. Однако в действительности при обращении к системному вызову выполнение переключается в привилегированный режим, благодаря чему во время выполнения системного вызова процессу доступны все инструкции, в том числе и привилегированные, а также системные структуры данных. По завершении выполнения системного вызова выполнение процесса снова переключается в пользовательский режим. Таким образом, механизм системных вызовов, код которых является частью ядра, является для обычного пользовательского процесса единственной возможностью получить права для выполнения привилегированных операций, и тем самым обеспечивается безопасность системы в целом.

Так как любой процесс может в различные моменты своего выполнения находиться как в привилегированном режиме, так и в пользовательском режиме, то и виртуальное адресное пространство процесса состоит из двух частей: одна из них используется, когда процесс находится в пользовательском режиме, а другая – в привилегированном. Причем процессу, находящемуся в пользовательском режиме, недоступна та часть его виртуального адресного пространства, которая соответствует режиму ядра. На Рис. 8 показано отображение исполняемого файла на виртуальное адресное пространство процесса, которое производит ОС при запуске процесса.

Далее нами будут рассмотрены некоторые системные вызовы, предоставляемые ОС UNIX. К интересующим нас вызовам относятся вызовы

- для создания процесса;
- для организации ввода вывода;
- для решения задач управления;
- для операции координации процессов;
- для установки параметров системы.

Отметим некоторые общие моменты, связанные с работой системных вызовов.

Большая часть системных вызовов определены как функции, возвращающие целое значение, при этом при нормальном завершении системный вызов возвращает 0, а при неудачном завершении -1[4]. При этом код ошибки можно выяснить, анализируя значение внешней переменной **`errno`**, определенной в заголовочном файле **`<errno.h>`**.

В случае, если выполнение системного вызова прервано сигналом, поведение ОС зависит от конкретной реализации. Например, в BSD UNIX ядро автоматически перезапускает системный вызов после его прерывания сигналом, и таким образом, внешне никакого различия с нормальным выполнением системного вызова нет. Стандарт POSIX допускает и вариант, когда системный вызов не перезапускается, при этом системный вызов вернет –1, а в переменной **`errno`** устанавливается значение **`EINTR`**, сигнализирующее о данной ситуации.

[4] Существуют и исключения из этого правила. Далее в этом пособии, рассказывая о системных вызовах, мы будем оговаривать такие исключения особо.

4.3 Порождение новых процессов.

Для порождения новых процессов в UNIX существует единая схема, с помощью которой создаются все процессы, существующие в работающем экземпляре ОС UNIX, за исключением процессов с PID=0 и PID=1[5].

Для создания нового процесса в операционной системе UNIX используется системный вызов **`fork().`**

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

При этом в таблицу процессов заносится новая запись, и порожденный процесс получает свой уникальный идентификатор. Для нового процесса создается контекст, большая часть содержимого которого идентична контексту родительского процесса, в частности, тело порожденного процесса содержит копии сегментов кода и данных его родителя. Кроме того, в порожденном процессе наследуется (т.е. является копией родительской):

- окружение - при формировании процесса ему передается некоторый набор параметров-переменных, используя которые, процесс может взаимодействовать с операционным окружением (интерпретатором команд и т.д.);
- файлы, открытые в процессе-отце, за исключением тех, которым было запрещено передаваться процессам-потомкам с помощью задания специального параметра при открытии. (Речь идет о том, что в системе при открытии файла с файлом ассоциируется некоторый атрибут, который определяет правила передачи этого открытого файла сыновним процессам. По умолчанию открытые в «отце» файлы можно передавать «потомкам», но можно изменить значение этого параметра и блокировать передачу открытых в процессе-отце файлов.);
- способы обработки сигналов;
- разрешение переустановки эффективного идентификатора пользователя;

[5] Эти процессы создаются во время начальной загрузки системы, механизм которой будет подробно рассмотрен ниже

- разделяемые ресурсы процесса-отца;
- текущий рабочий каталог и домашний каталоги
- и т.д.

Не наследуются порожденным процессом следующие атрибуты родительского процесса:

- идентификатор процесса (PID)
- идентификатор родительского процесса (PPID)
- сигналы, ждущие доставки в родительский процесс
- время посылки предупреждающего сигнала, установленное системным вызовом **`alarm()`** (в порожденном процессе оно сбрасывается в нуль)
- блокировки файлов, установленные родительским процессом

По завершении системного вызова **`fork()`** каждый из процессов – родительский и порожденный – получив управление, продолжат выполнение с одной и той же инструкции одной и той же программы, а именно с той точки, где происходит возврат из системного вызова **`fork().`** Вызов **`fork()`** в случае удачного завершения возвращает сыновнему процессу значение **0**, а родительскому **PID** порожденного процесса. Это принципиально важно для различения сыновнего и родительского процессов, так как сегменты кода у них идентичны. Таким образом, у программиста имеется возможность разделить путь выполнения инструкций в этих процессах.

В случае неудачного завершения, т.е. если сыновний процесс не был порожден, системный вызов **`fork()`** возвращает –1, код ошибки устанавливается в переменной **`errno`**.

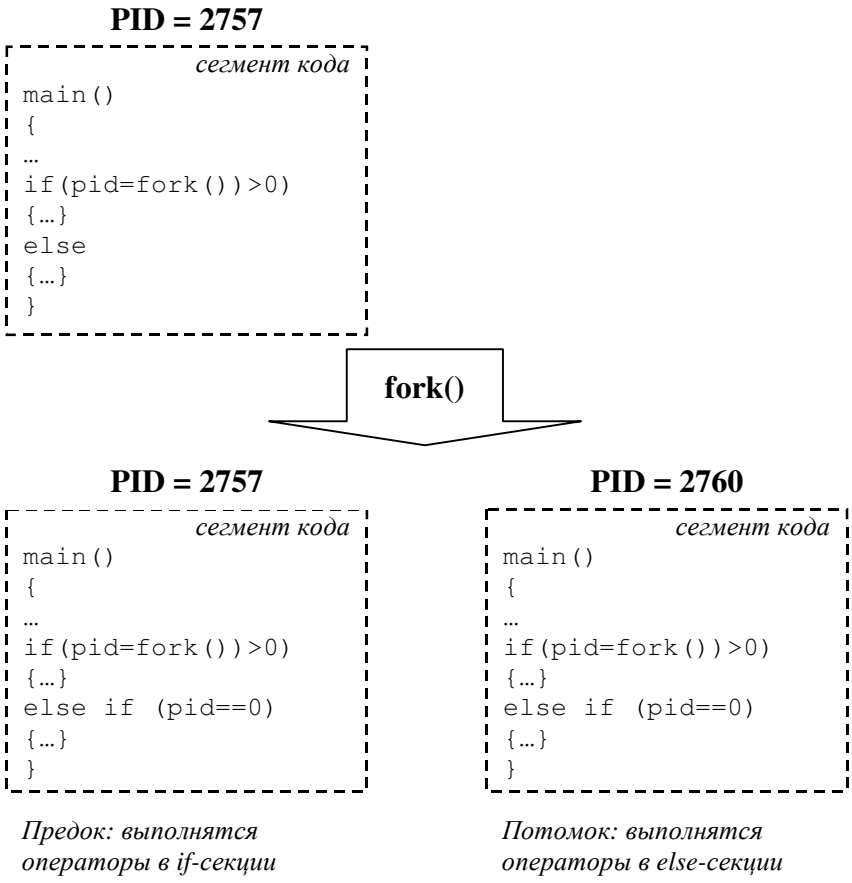

Рис. 9 Выполнение системного вызова fork()

Пример 1. Порождение сыновнего процесса. Идентификаторы процессов.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(int argc, char **argv)
{
     printf("PID=%d; PPID=%d \n",getpid(),
    getppid());
     /*печать PID  текущего процесса и PID процесса-
    предка */
     fork();
     /*создание нового процесса, с этого момента два
    процесса функционируют параллельно и независимо*/
     printf("PID=%d; PPID=%d \n",getpid(),
    getppid());
    /*оба процесса печатают PID  текущего процесса и PID
процесса-предка*/
```

```
        return 0;
}
```

В этом примере оба процесса узнают свой собственный идентификатор процесса с помощью вызова **getpid()**, а идентификатор родительского процесса – с помощью вызова **getppid()**.

Следует отметить два момента, связанных с функционированием двух процессов. Во-первых, нельзя определенно сказать, в каком порядке будет происходить печать с момента появления двух процессов – это будет определяться планировщиком процессов. Во-вторых, ответ на вопрос - какой идентификатор родительского процесса распечатает вновь созданный процесс, если процесс-предок завершит свою работу раньше, будет приведен несколько ниже.

Пример 2. Порождение сыновнего процесса. Одновременное выполнение.

Программа создает два процесса – процесс-предок распечатывает заглавные буквы, а процесс-потомок строчные.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(int argc, char **argv)
{
        char ch, first, last;
        int pid;
        if((pid=fork())>0)
        {
                /*процесс-предок*/
                first =’A’;
                last =’Z’;
        }
        else
        {
                /*процесс-потомок*/
                first =’a’;
                last =’z’;
        }
```

```
        for (ch = first; ch <= last; ch++)
        {
             write(1,&ch,1);
        }
        return 0;
}
```

Оба процесса распечатывают буквы одним и тем же оператором **for.** Оба процесса имеют возможность получить управление, таким образом любой из них может начать исполнение первым.

4.4 Механизм замены тела процесса.

Семейство системных вызовов **exec()**[6] производит замену тела вызывающего процесса, после чего данный процесс начинает выполнять другую программу, передавая управление на точку ее входа. Возврат к первоначальной программе происходит только в случае ошибки при обращении к **exec()**, т.е. если фактической замены тела процесса не произошло.

Заметим, что выполнение "нового" тела происходит в рамках уже существующего процесса, т.е. после вызова **exec()** сохраняется идентификатор процесса, и идентификатор родительского процесса, таблица дескрипторов файлов, приоритет, и большая часть других атрибутов процесса. Фактически происходит замена сегмента кода и сегмента данных. Изменяются следующие атрибуты процесса:

- диспозиция сигналов: для сигналов, которые перехватывались, после замены тела процесса будет установлена обработка по умолчанию, так как в новой программе могут отсутствовать указанные функции-обработчики сигналов;
- эффективные идентификаторы владельца и группы могут измениться, если для новой выполняемой программы установлен s-бит
- перед началом выполнения новой программы могут быть закрыты некоторые файлы, ранее открытые в процессе. Это касается тех файлов, для которых при помощи системного

[6] Существует шесть различных системных вызовов exec(), отличающихся параметрами, которые будут описаны ниже.

вызова **fcntl()** был установлен флаг **close-on-exec**. Соответствующие файловые дескрипторы будут помечены как свободные.

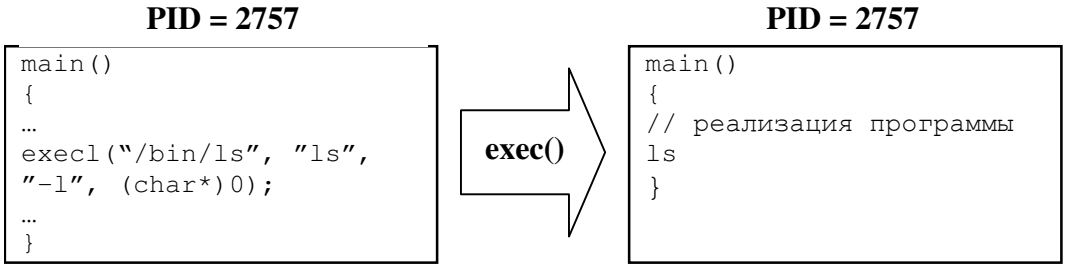

Рис. 10 Выполнение системного вызова exec()

Ниже представлены прототипы функций семейства **exec()**:

```
#include <unistd.h>
int execl(const char *path, char *arg0,…);
int execlp(const char *file, char *arg0,…);
int execle(const char *path, char *arg0,…, const char **env);
int execv(const char *path, const char **arg);
int execvp(const char *file, const char **arg);
int execve(const char *path, const char **arg, const char **env);
```

Первый параметр во всех вызовах задает имя (краткое или полное путевое) файла программы, подлежащей исполнению. Этот файл должен быть исполняемым файлом (в UNIX-системах это может быть также командный файл (сценарий) интерпретатора shell, но стандарт POSIX этого не допускает), и пользователь-владелец процесса должен иметь право на исполнение данного файла. Для функций с суффиксом «p» в названии имя файла может быть кратким, при этом при поиске нужного файла будет использоваться переменная окружения PATH.

Далее передаются аргументы командной строки для вновь запускаемой программы, которые отобразятся в ее массив **argv** – в виде списка аргументов переменной длины для функций с суффиксом «l» либо в виде вектора строк для функций с суффиксом «v». В любом случае, в списке аргументов должно присутствовать как минимум 2 аргумента: имя программы, которое отобразится в элемент **argv[0]**, и значение NULL, завершающее список.

В функциях с суффиксом «e» имеется также дополнительный аргумент, описывающий переменные окружения для вновь

запускаемой программы – это массив строк вида `name=value`, завершенный значением NULL.

Пример 3. Запуск на выполнение команды ls.

```
#include <unistd.h>
#include <stdio.h>
int main(int argc, char **argv)
{
     …
     /*тело программы*/
     …
     execl(“/bin/ls”,”ls”,”-l”,(char*)0);
     /*  или execlp(“ls”,”ls”, ”-l”,(char*)0);*/
     printf(“это напечатается в случае неудачного обращения к предыдущей функции, к примеру, если не был найден файл ls \n”);
     …
}
```

В данном случае второй параметр – вектор из указателей на параметры строки, которые будут переданы в вызываемую программу. Как и ранее первый указатель – имя программы, последний – нулевой указатель. Эти вызовы удобны, когда заранее неизвестно число аргументов вызываемой программы.

Пример 4. Вызов программы компиляции.

```
#include <unistd.h>
int main(int argc, char **argv)
{
     char *pv[]={
                    “cc”,
                    “-o”,
                    “ter”,
                    “ter.c”,
                     (char*)0
                     };
     …
     /*тело программы*/
     …
```

```
        execv ("/bin/cc",pv);
        …
}
```

Наиболее интересным является использование **fork()** совместно с системным вызовом **exec().** Как отмечалось выше системный вызов **exec()** используется для запуска исполняемого файла в рамках существующего процесса. Ниже приведена общая схема использования связки **fork() - exec().**

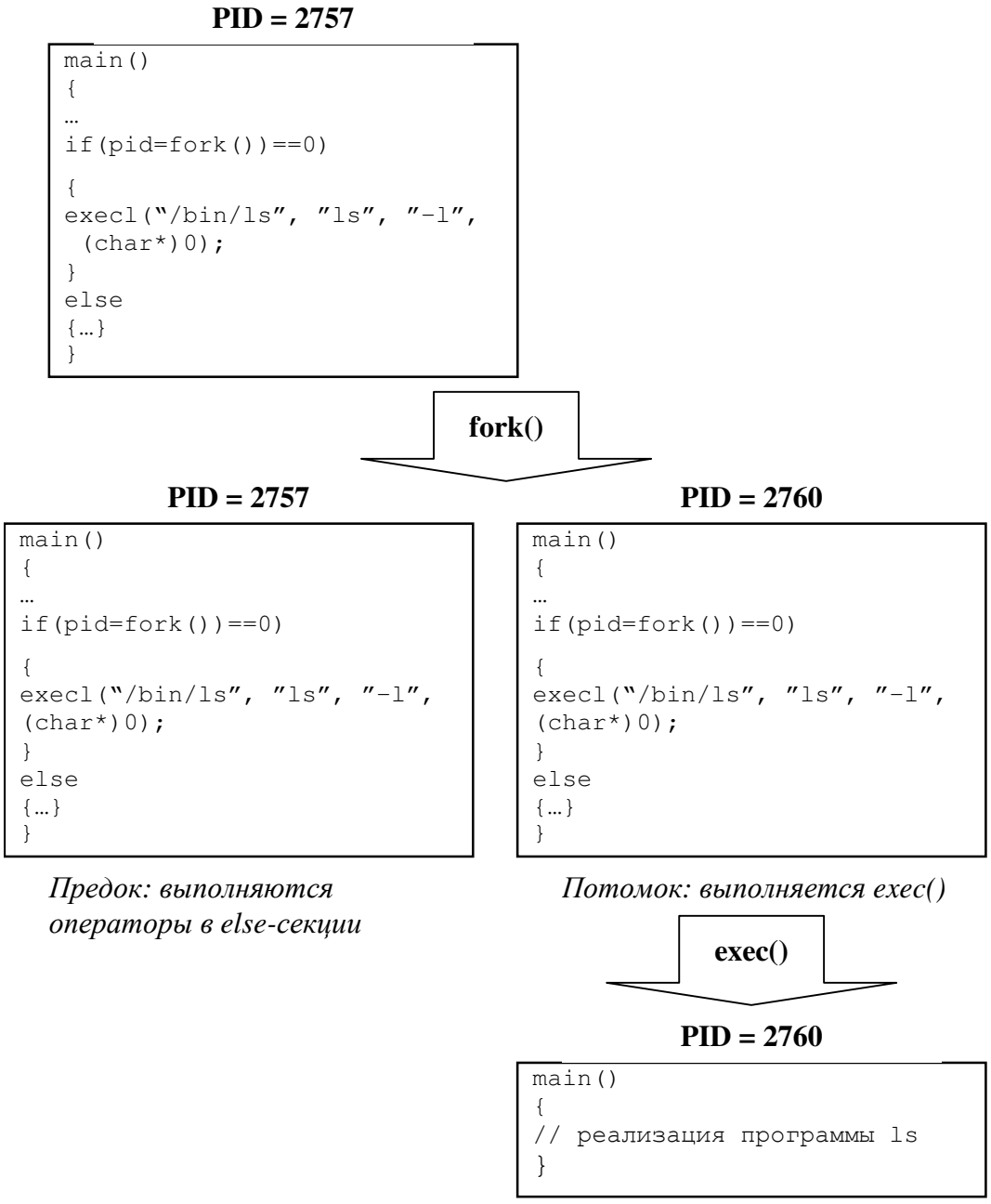

Рис. 11 Использование схемы fork()-exec()

Пример 5. Схема использования fork-exec

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

```
#include <unistd.h>
int main(int argc, char **argv)
{
     int pid;
     if ((pid=fork())!=0){
          if(pid>0)
          {
               /* процесс-предок */
          }
          else
          {
               /* ошибка */
          }

     }
     else
     {
          /* процесс-потомок */
     }
}
```

Пример 6. Использование схемы fork-exec

Программа порождает три процесса, каждый из которых запускает программу **echo** посредством системного вызова **exec().** Данный пример демонстрирует важность проверки успешного завершения системного вызова **exec(),** в противном случае возможно исполнение нескольких копий исходной программы. В нашем случае если все вызовы **exec()** проработают неуспешно, то копий программ будет восемь. Если все вызовы **exec()** будут успешными, то после последнего вызова **fork()** будет существовать четыре копии процесса. В любом случае, порядок, в котором они будут выполняться, не определен.

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(int argc, char **argv)
{
```

```
        if(fork()==0)
        {
             execl("/bin/echo",      "echo",      "это",
             "сообщение один", NULL);
             printf("ошибка\n");
        }
        if(fork()==0)
        {
             execl("/bin/echo",      "echo",      "это",
             "сообщение два", NULL);
             printf("ошибка\n");
        }
        if(fork()==0)
        {
             execl("/bin/echo",      "echo",      "это",
             "сообщение три", NULL);
             printf("ошибка\n ");
        }
        printf("процесс-предок закончился\n");
        return 0;
}
```

Результат работы может быть следующим.

процесс-предок закончился

это сообщение три

это сообщение два

это сообщение один

4.5 Завершение процесса.

Для завершения выполнения процесса предназначен системный вызов **`_exit()`**

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

Этот вызов никогда не завершается неудачно, поэтому для него не предусмотрено возвращающего значения. С помощью параметра **`exit_code`** процесс может передать породившему его процессу информацию о статусе своего завершения. Принято, хотя и не является обязательным правилом, чтобы процесс возвращал

нулевое значение при нормальном завершении, и ненулевое – в случае какой-либо ошибки или нештатной ситуации.

В стандартной библиотеке Си имеется сервисная функция **exit()**, описанная в заголовочном файле **stdlib.h**, которая, помимо обращения к системному вызову **_exit()**, осуществляет ряд дополнительных действий, таких как, например, очистка стандартных буферов ввода-вывода.

Кроме обращения к вызову **_exit()**, другими причинами завершения процесса могут быть:

- выполнение оператора **return**, входящего в состав функции **main()**
- получение некоторых сигналов (об этом речь пойдет чуть ниже)

В любом из этих случаев происходит следующее:

- освобождаются сегмент кода и сегмент данных процесса
- закрываются все открытые дескрипторы файлов
- если у процесса имеются потомки, их предком назначается процесс с идентификатором 1
- освобождается большая часть контекста процесса, однако сохраняется запись в таблице процессов и та часть контекста, в которой хранится статус завершения процесса и статистика его выполнения
- процессу-предку завершаемого процесса посылается сигнал **SIGCHLD**

Состояние, в которое при этом переходит завершаемый процесс, в литературе часто называют состоянием “зомби”.

Процесс-предок имеет возможность получить информацию о завершении своего потомка. Для этого служит системный вызов **wait()**:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
```

При обращении к этому вызову выполнение родительского процесса приостанавливается до тех пор, пока один из его потомков не завершится либо не будет остановлен. Если у процесса имеется несколько потомков, процесс будет ожидать завершения любого из них (т.е., если процесс хочет получить информацию о завершении

каждого из своих потомков, он должен несколько раз обратиться к вызову **wait()**).

Возвращаемым значением **wait()** будет идентификатор завершенного процесса, а через параметр **status** будет возвращена информация о причине завершения процесса (завершен путем вызова **_exit()**, либо прерван сигналом) и коде возврата. Если процесс не интересуется это информацией, он может передать в качестве аргумента вызову **wait()** NULL-указатель.

Конкретный формат данных, записываемых в параметр **status**, может различаться в разных реализациях ОС. Во всех современных версиях UNIX определены специальные макросы для извлечения этой информации, например:

макрос **WIFEXITED(*status)** возвращает ненулевое значение, если процесс был завершен путем вызова **_exit()**, при этом макрос **WEXITSTATUS(*status)** возвращает статус завершения, переданный через **_exit()**;

макрос **WIFSIGNALED(*status)** возвращает ненулевое значение, если процесс был прерван сигналом, при этом макрос **WTERMSIG(*status)** возвращает номер этого сигнала;

макрос **WIFSTOPPED(*status)** возвращает ненулевое значение, если процесс был приостановлен системой управления заданиями, при этом макрос **WSTOPSIG(*status)** возвращает номер сигнала, с помощью которого он был приостановлен.

Если к моменту вызова **wait()** один из потомков данного процесса уже завершился, перейдя в состояние зомби, то выполнение родительского процесса не блокируется, и **wait()** сразу же возвращает информацию об этом завершенном процессе. Если же к моменту вызова **wait()** у процесса нет потомков, системный вызов сразу же вернет –1. Также возможен аналогичный возврат из этого вызова, если его выполнение будет прервано поступившим сигналом.

После того, как информация о статусе завершения процесса-зомби будет доставлена его предку посредством вызова **wait()**, все оставшиеся структуры, связанные с данным процессом-зомби, освобождаются, и запись о нем удаляется из таблицы процессов. Таким образом, переход в состояние зомби необходим именно для того, чтобы процесс-предок мог получить информацию о судьбе своего завершившегося потомка, независимо от того, вызвал он **wait()** до или после фактического его завершения.

Что происходит с процессом-потомком, если его предок вообще не обращался к **wait()** и/или завершился раньше потомка? Как уже говорилось, при завершении процесса отцом для всех его

потомков становится процесс с идентификатором 1. Он и осуществляет системный вызов **wait(),** тем самым освобождая все структуры, связанные с потомками-зомби.

Часто используется сочетание функций **fork()-wait(),** если процесс-сын предназначен для выполнения некоторой программы, вызываемой посредством функции **exec().** Фактически этим предоставляется процессу-родителю возможность контролировать окончание выполнения процессов-потомков.

Пример 7. Использование системного вызова wait()

Пример программы, последовательно запускающей программы, имена которых указаны при вызове.

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
int main(int argc, char **argv)
{
     int i;
     for (i=1; i<argc; i++)
     {
          int status;
          if(fork()>0)
          {
          /*процесс-предок ожидает сообщения
          от процесса-потомка о завершении */
               wait(&status);
               printf("process-father\n");
               continue;
          }
          execlp(argv[i], argv[i], 0);
          return -1;
          /*попадем сюда при неуспехе exec()*/
     }
     return 0;
}
```

Пусть существуют три исполняемых файла **print1**, **print2**, **print3,** каждый из которых только печатает текст **first**, **second**, **third** соответственно, а код вышеприведенного примера находится в исполняемом файле с именем **file**. Тогда результатом работы команды **file print1 print2 print3** будет

first

process-father

second

process-father

third

process-father

Пример 8. Использование системного вызова wait()

В данном примере процесс-предок порождает два процесса, каждый из которых запускает команду **echo**. Далее процесс-предок ждет завершения своих потомков, после чего продолжает выполнение.

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
int main(int argc, char **argv)
{
     if ((fork()) == 0) /*первый процесс-потомок*/
     {
          execl("/bin/echo",  "echo",  "this  is",
     "string 1", 0);
          return -1;
     }
     if ((fork()) == 0) /*второй процесс-потомок*/
     {
          execl("/bin/echo",  "echo",  "this  is",
     "string 2", 0);
          return -1;
     }
     /*процесс-предок*/
```

```
        printf("process-father    is    waiting    for
children\n");
        while(wait(NULL) != -1);
        printf("all children terminated\n");
        return 0;
}
```

В данном случае **wait()** вызывается в цикле три раза – первые два ожидают завершения процессов-потомков, последний вызов вернет неуспех, ибо ждать более некого.

4.6 Жизненный цикл процесса в ОС UNIX.

Подведем короткие итоги. Итак, процесс в UNIX представляет собой исполняемую программу вместе с необходимым ей окружением. Окружение состоит из информации о процессе, которая содержится в различных системных структурах данных, информации о содержимом регистров, программ операционной системы, стеке процесса, информации об открытых файлах, обработке сигналов и так далее. Процесс представляет собой изменяющийся во времени динамический объект. Программа представляет собой часть процесса. Процесс может создавать процессы-потомки посредством системного вызова **fork(),** может изменять свою программу через системный вызов **exec().** Процесс может приостановить свое исполнение, используя вызов **wait()**, а также завершить свое исполнение посредством функции **exit().**

С учетом вышеизложенного, рассмотрим подробнее состояния, в которых может находится процесс:

1. Процесс только что создан посредством вызова **fork()**.
2. Процесс находится в очереди готовых на выполнение процессов.
3. Процесс выполняется в режиме задачи, т.е. реализуется алгоритм, заложенный в программу. Выход из этого состояния может произойти через системный вызов, прерывание или завершение процесса.
4. Процесс может выполняться в режиме ядра ОС, когда по требованию процесса через системный вызов выполняются определенные инструкции ядра ОС или произошло прерывание.
5. Процесс в ходе выполнения не имеет возможность получить требуемый ресурс и переходит в состояние блокирования.
6. Процесс осуществил вызов **_exit()** или получил сигнал на завершение. Ядро освобождает ресурсы, связанные с процессом,

кроме кода возврата и статистики выполнения. Далее процесс переходит в состоянии *зомби,* а затем уничтожается.

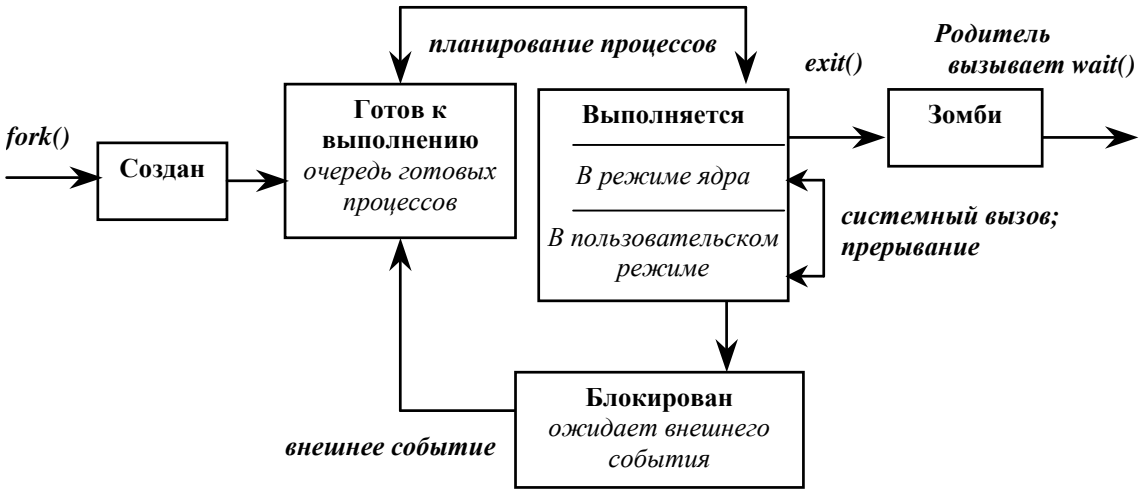

Рис. 12 Жизненный цикл процесса в ОС UNIX.

4.7 Начальная загрузка. Формирование 0 и 1 процессов.

Рассмотрим подробнее, что происходит в момент начальной загрузки ОС UNIX. Начальная загрузка – это загрузка ядра системы в основную память и ее запуск. Нулевой блок каждой файловой системы предназначен для записи короткой программы, выполняющей начальную загрузку. Начальная загрузка выполняется в несколько этапов.

1.Аппаратный загрузчик читает нулевой блок системного устройства.

2.После чтения этой программы она выполняется, т.е. ищется и считывается в память файл **`/unix`**, расположенный в корневом каталоге и который содержит код ядра системы.

3.Запускается на исполнение этот файл.

В самом начале ядром выполняются определенные действия по инициализации системы, а именно, устанавливаются системные часы (для генерации прерываний), формируется диспетчер памяти, формируются значения некоторых структур данных (наборы буферов блоков, буфера индексных дескрипторов) и ряд других. По окончании этих действий происходит инициализация процесса с номером **"0".** По понятным причинам для этого невозможно использовать методы порождения процессов, изложенные выше, т.е. с использованием функций **`fork()`** и **`exec().`** При инициализации

этого процесса резервируется память под его контекст и формируется нулевая запись в таблице процессов. Основными отличиями нулевого процесса являются следующие моменты:

- Данный процесс не имеет кодового сегмента , это просто структура данных, используемая ядром и процессом его называют потому, что он каталогизирован в таблице процессов.
- Он является чисто системным процессом, т.е. существует в течении всего времени работы системы и считается, что он активен, когда работает ядро ОС.

Далее ядро копирует **"0"** процесс и создает **"1"** процесс. Алгоритм создания этого процесса напоминает стандартную процедуру, хотя и носит упрощенный характер. Сначала процесс **"1"** представляет собой полную копию процесса **"0"** , т.е. у него нет области кода. Далее происходит увеличение его размера и во вновь созданную кодовую область копируется программа, реализующая системный вызов **exec()**, необходимый для выполнения программы **/etc/init.** На этом завершается подготовка первых двух процессов. Первый из них представляет собой структуру данных, при помощи которой ядро организует мультипрограммный режим и управление процессами. Второй – это уже подобие реального процесса. Далее ОС переходит к выполнению программ диспетчера. Диспетчер наделен обычными функциями и на первом этапе у него нет выбора – он запускает **exec(),** который заменит команды процесса **"1"** кодом, содержащимся в файле **/etc/init.** Получившийся процесс, называемый **init**, призван настраивать структуры процессов системы. Далее он подключает интерпретатор команд к системной консоли. Так возникает однопользовательский режим, так как консоль регистрируется с корневыми привилегиями и доступ по каким-либо другим линиям связи невозможен. При выходе из однопользовательского режима **init** создает многопользовательскую среду. С этой целью для каждого активного канала связи, т.е. каждого терминала, **init** создает отдельный процесс, выполняющий команду **getty**. Эта программа ожидает входа в систему кого-либо по каналу связи. Далее, используя системный вызов **exec()**, **getty** передает управление программе **login**, проверяющей пароль. Во время работы ОС процесс **init** ожидает завершения одного из порожденных им процессов (т.е. окончания сессии работы с системой), после чего он активизируется и создает для соответствующего терминала новый процесс **getty** взамен завершившегося . Таким образом, процесс **init**

поддерживает многопользовательскую структуру во время функционирования системы. Схема описанного “круговорота” представлена на Рис. 13

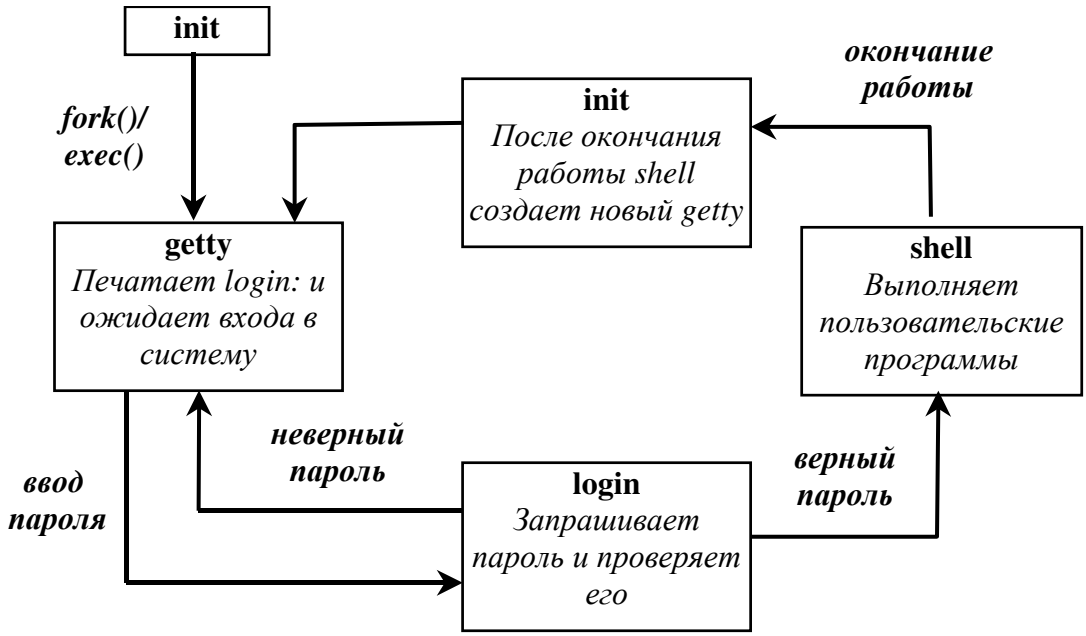

Рис. 13 Поддержка многопользовательской работы в ОС UNIX.

4.8 Планирование процессов в ОС UNIX.

Планирование процесса, которому предстоит занять время центрального процессора, основывается на понятии приоритета. Каждому процессу сопоставляется некоторое целое числовое значение его приоритета (в т.ч., возможно, и отрицательное). Общее правило таково: чем больше числовое значение приоритета процесса, тем меньше его приоритет, т.е. наибольшие шансы занять время ЦП будут у того процесса, у которого числовое значение приоритета минимально.

Итак, числовое значение приоритета, или просто приоритет процесса - это параметр, который размещен в таблице процессов, и по значению этого параметра осуществляется выбор очередного процесса для продолжения работы и принимается решение о приостановке работающего процесса. Приоритеты системных и пользовательских процессов вычисляются по-разному. Рассмотрим, как это происходит для пользовательского процесса.

В общем случае, значение приоритета есть некоторая функция

```
P_PRI  = F(P_NICE, P_CPU)
```

Т.е. в вычислении приоритета **P_PRI** используются две изменяемые составляющие - **P_NICE** и **P_CPU** (в простейшем случае эти составляющие просто суммируются). **P_NICE** - это

пользовательская составляющая приоритета. Его начальное значение полагается равным системной константе **NZERO**, в процессе выполнения процесса **P_NICE** может модифицироваться системным вызовом **nice().** Аргументом этого системного вызова является добавка к текущему значению (для обычного – непривилегированного - процесса эти добавки представляют собой неотрицательные числа). Значение **P_NICE** наследуется при порождении процессов, и таким образом, значение приоритета не может быть понижено при наследовании. Заметим, что изменяться **P_NICE** может только в сторону увеличения значения (до некоторого предельного значения), таким образом пользователь может снижать приоритет своих процессов.

P_CPU - это системная составляющая. Она формируется системой следующим образом: при прерывании по таймеру через предопределенные периоды времени для процесса, занимающего процессор в текущий момент, **P_CPU** увеличивается на единицу. Так же, как и **P_NICE**, **P_CPU** имеет некоторое предельное значение. Если процесс будет находиться в состоянии выполнения так долго, что составляющая **P_CPU** достигнет своего верхнего предела, то значение **P_CPU** будет сброшено в нуль, а затем снова начнет расти. Отметим, однако, что такая ситуация весьма маловероятна, так как скорее всего, этот процесс будет выгружен и заменен другим еще до того момента, как **P_CPU** достигнет максимума.

Упрощенная формула вычисления приоритета такова

```
P_PRI  = P_USER + P_NICE + P_CPU
```

Константа **P_USER** представляет собой нижний порог приоритета для пользовательских процессов. Пользовательская составляющая, как правило, учитывается в виде разности **P_NICE – N_ZERO**, что позволяет принимать в расчет только добавку, введенную посредством системного вызова **nice().** Системная составляющая учитывается с некоторым коэффициентом. Поскольку неизвестно, проработал ли до момента прерывания по таймеру процесс на процессоре полный интервал между прерываниями, то берется некоторое усреднение. Суммарно получается следующая формула для вычисления приоритета

```
P_PRI  = P_USER + P_NICE – NZERO + P_CPU/a
```

Заметим, что, если приоритет процесса не изменялся при помощи **nice()**, то единственной изменяемой составляющей приоритета будет **P_CPU**, причем эта составляющая растет только для того процесса, который находится в состоянии выполнения. В

тот момент, когда значение ее станет таково, что в очереди готовых к выполнению процессов найдется процесс с меньшим значением приоритета, выполняемый процесс будет приостановлен и заменен процессом с меньшим значением приоритета. При этом значение составляющей **P_CPU** для выгруженного процесса сбрасывается в нуль.

Пример 9. Планирование процессов.

Рассмотрим два активных процесса, разделяющих процессор, причем таких, что ни их процессы-предки, ни они сами не меняли составляющую **P_NICE** системным вызовом **nice()**. Тогда **P_NICE=NZERO** и оба процесса имеют начальное значение приоритета **P_PRI=P_USER**, так как для них обоих в начальный момент **P_CPU=0**. Пусть прерывание по таймеру происходит через **N** единиц времени, и каждый раз значение **P_CPU** увеличивается на единицу, а в вычисление приоритета составляющая **P_CPU** входит с коэффициентом **1/A**. Таким образом, дополнительная единица в приоритете процесса, занимающего процессор, «набежит» через **A** таймерных интервалов. Значение **P_CPU** второго процесса остается неизменным, и его приоритет остается постоянным. Через **NA** единиц времени разница приоритетов составит единицу в пользу второго процесса и произойдет смена процессов на процессоре.

4.9 Принципы организация свопинга.

В системе определенным образом выделяется пространство для области свопинга. Есть пространство оперативной памяти, в котором находятся процессы, обрабатываемые системой в режиме мультипрограммирования. Есть область на ВЗУ, предназначенная для откачки этих процессов по мере необходимости. Упрощенная схема планирования подкачки основывается на использовании некоторого приоритета, который называется **P_TIME** и также находится в контексте процесса. В этом параметре аккумулируется время пребывания процесса в состоянии мультипрограммной обработки, или в области свопинга. В поле **P_TIME** существует счётчик выгрузки (**outage**) и счётчик загрузки (**inage**).

При перемещении процесса из оперативной памяти в область свопинга или обратно система обнуляет значение параметра **P_TIME**. Для загрузки процесса в память из области свопинга выбирается процесс с максимальным значением **P_TIME**. Если для загрузки этого процесса нет свободного пространства оперативной памяти, то система ищет среди процессов в оперативной памяти процесс, ожидающий ввода/вывода (сравнительно медленных операций,

процессы у которых приоритет выше значения **P_ZERO**) и имеющий максимальное значение **P_TIME** (т.е. тот, который находился в оперативной памяти дольше всех). Если такого процесса нет, то выбирается просто процесс с максимальным значением **P_TIME**.

ЧАСТЬ III. РЕАЛИЗАЦИЯ ВЗАИМОДЕЙСТВИЯ ПРОЦЕССОВ.

В этой части будет рассмотрен весь спектр средств межпроцессного взаимодействия, которые предоставляет программисту ОС UNIX. Отметим, что многие из этих средств (такие как сигналы, средства IPC, сокеты, MPI) или их аналоги имеются и в других операционных системах. Таким образом, данное рассмотрение будет полезно в целом для формирования представления о многообразии механизмов взаимодействия процессов, теоретические основы которых были рассмотрены ранее.

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.

Первая проблема представляет собой необходимость 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, обеспечивающих взаимодействие процессов, функционирующих на различных компьютерах в сети используется адресация, принятая в конкретной сети ЭВМ (примером могут служить аппарат сокетов и MPI). В средствах взаимодействия процессов, локализованных в пределах одной ЭВМ, способ именования зависит от конкретного механизма взаимодействия. В частности, для ОС UNIX механизмы взаимодействия процессов можно разделить на те средства, которые доступны исключительно родственным процессам, и средства, доступные произвольным процессам[7] (см. также Рис. 14).

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, наследуется

[7] при условии, разумеется, что у данного процесса имеются достаточные права для доступа к конкретному ресурсу, через который осуществляется взаимодействие

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 не происходит наследования некоторых свойств процессов, которые могут использоваться для именования. Поэтому в данном случае обычно используются две схемы: первая – использование для именования идентификаторов взаимодействующих процессов (к примеру, аппарат передачи 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).

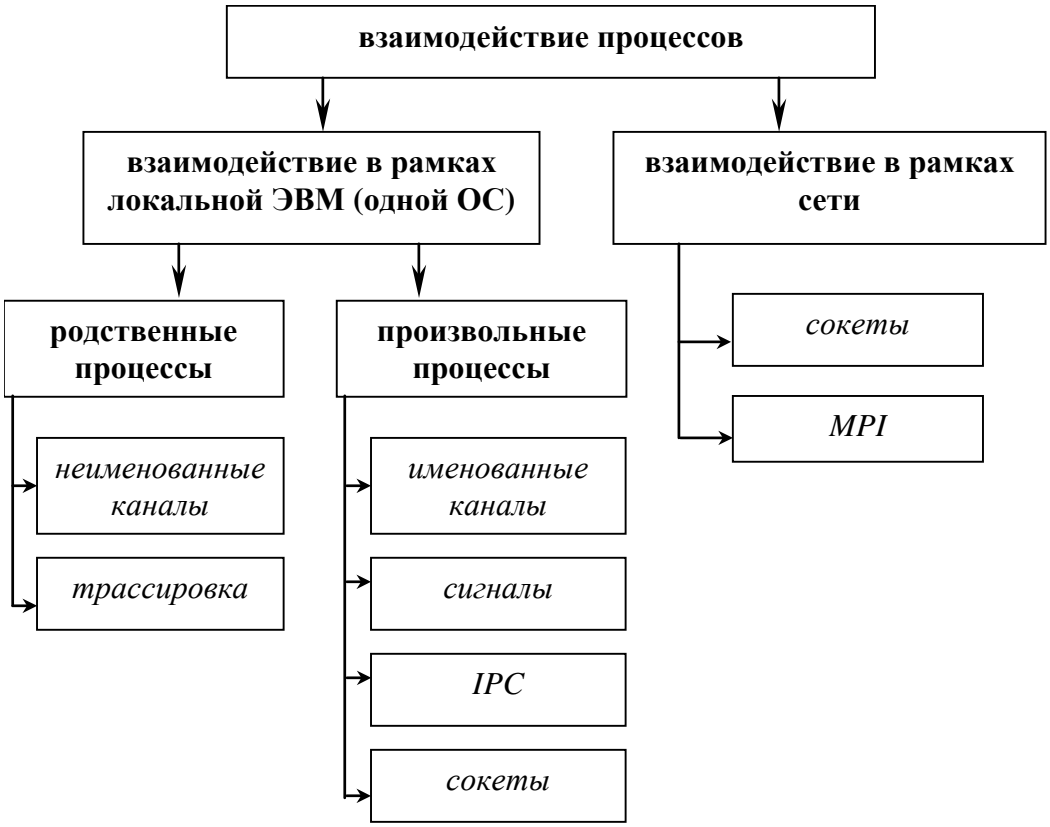

Рис. 14 Классификация средств взаимодействия процессов ОС UNIX

Другая проблема организации взаимодействия – это проблема синхронизации взаимодействующих процессов. Любое взаимодействие процессов представимо в виде оказания одним процессом воздействия на другой процесс либо 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 учитывать, и при необходимости обеспечивать семантическую синхронизацию процессов.

Ниже будут рассмотрены конкретные средства взаимодействия процессов, предоставляемые ОС UNIX.

5 Элементарные средства межпроцессного взаимодействия.

5.1 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 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, зарезервировано один или несколько номеров сигналов, семантика которых определяется пользовательскими процессами по своему усмотрению (например, процессы могут посылать друг другу сигналы с целью синхронизации).

Количество различных сигналов в современных версиях UNIX около 30, каждый из них имеет уникальное имя и номер. Описания представлены в файле **<signal.h>**. В таблице приведено несколько примеров сигналов[8]:

Числовое значение	Константа	Значение сигнала
2	SIGINT	Прерывание выполнения по нажатию Ctrl-C
3	SIGQUIT	Аварийное завершение работы
9	SIGKILL	Уничтожение процесса
14	SIGALRM	Прерывание от программного таймера
18	SIGCHLD	Завершился процесс-потомок

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.

[8] Следует заметить, что в разных версиях UNIX имена сигналов могут различаться.

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 установить специальную обработку сигнала, в этом случае по приходу 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 в процессе свое 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.

Отдельного рассмотрения заслуживает ситуация, когда сигнал приходит в момент выполнения системного вызова. Обработка такой ситуации в разных версиях UNIX реализована по-разному, например, обработка сигнала может быть отложена до завершения системного вызова; либо системный вызов автоматически перезапускается после его прерывания сигналом; либо системный вызов вернет –1, а в переменной **errno** будет установлено значение **EINTR**

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

```
#include <sys/types.h>
#include <signal.h>
int kill (pit_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

этого 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 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.

Отметим одну особенность реализации сигналов в ранних версиях UNIX: каждый раз при получении сигнала его диспозиция (т.е. действие при получении сигнала) сбрасывается на действие по умолчанию, т.о. если процесс желает многократно обрабатывать сигнал своим собственным обработчиком, он должен каждый раз при обработке сигнала заново устанавливать реакцию на него (см. пример 9)

В заключении отметим, что механизм сигналов является достаточно ресурсоемким, ибо отправка сигнала представляет собой

системный вызов, а доставка сигнала - прерывание выполнения процесса-получателя. Вызов функции-обработчика и возврат требует операций со стеком. Сигналы также несут весьма ограниченную информацию.

Пример 10. 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 = 0;
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;
}
```

Пример 11. Удаление временных файлов при завершении программы.

При разработке программ нередко приходится создавать временные файлы , которые позже удаляются. Если произошло непредвиденное событие, такие файлы могут остаться не удаленными. Ниже приведено решение этой задачи.

```
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
const char * tempfile = “abc”;

void SigHndlr (int s)
{
     unlink(tempfile);
     /* уничтожение временного файла в случае
прихода сигнала SIGINT. В случае, если такой файл не
существует (еще не создан или уже удален), вызов
вернет –1 */
}

int main(int argc, char **argv)
{
     signal (SIGINT, SigHndlr); /*установка реакции
на сигнал */
     …
     creat(tempfile, 0666); /*создание временного
файла*/
     …
     unlink(tempfile);
     /*уничтожение временного файла в случае
     нормального функционирования процесса */
     return 0;
}
```

В данном примере для создания временного файла используется системный вызов **`creat()`**:

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

А системный вызов **unlink()** удаляет имя и файл, на который оно ссылается.

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

Пример 12. Программа “Будильник”.

Программа “Будильник”. Существуют задачи, в которых необходимо прервать выполнение процесса по истечении некоторого количества времени. Средствами ОС “заводится” будильник, который будет поторапливать ввести некоторое имя. Системный вызов **alarm():**

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

инициализирует отложенное появление сигнала **SIGALRM** - процесс запрашивает ядро отправить ему самому сигнал по прошествии определенного времени**.**

```
#include <unistd.h>
#include <signal.h>
#include <stdio.h>

void alrm(int s) /*обработчик сигнала SIG_ALRM */
{
     printf(“\n жду имя \n”);
     alarm(5); /* заводим будильник */
     signal(SIGALRM, alrm); /* переустанавливаем
     реакцию на сигнал */
}

int main(int argc, char **argv)
{
     char s[80];
     signal(SIGALRM, alrm);
     /* установка обработчика alrm на приход сигнала
SIG_ALRM */
     alarm(5); /* заводим будильник */
     printf(“Введите имя \n”);
     for (;;)
```

```
    {
        printf("имя:");
        if (gets(s) != NULL) break; /* ожидаем
        ввода имени */
    };
    printf("OK! \n");
    return 0;
}
```

В начале программы мы устанавливаем реакцию на сигнал **SIGALRM** - функцию **alarm()**, далее мы заводим будильник, запрашиваем "*Введите имя*" и ожидаем ввода строки символов. Если ввод строки задерживается, то будет вызвана функция **alarm()**, которая напомнит, что программа "ждет имя", опять заведет будильник и поставит себя на обработку сигнала **SIGALRM** еще раз. И так будет до тех пор, пока не будет введена строка. Здесь имеется один нюанс: если в момент выполнения системного вызова возникает событие, связанное с сигналом, то система прерывает выполнение системного вызова и возвращает код ответа, равный «-1».

Пример 13. Двухпроцессный вариант программы "Будильник".

```
#include <signal.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

void alr(int s)
{
    printf("\n Быстрее!!! \n");
    signal(SIGALRM, alr);
    /* переустановка обработчика alr на приход
сигнала SIGALRM */
}

int main(int argc, char **argv)
{
    char s[80];
```

```
        int pid;


        signal(SIGALRM, alr);
        /* установка обработчика alr на приход сигнала
    SIGALRM */
        if  (pid = fork()) {
             for (;;)
             {
                  sleep(5);       /*приостанавливаем
                       процесс на 5 секунд */
                  kill(pid, SIGALRM);
                  /*отправляем сигнал SIGALRM процессу-
                  сыну */
             }
        }
        else {
             printf("Введите имя \n");
             for (;;)
             {
                  printf("имя:");
                  if (gets(s) != NULL) break; /*ожидаем
                  ввода имени*/
             }
             printf("OK!\n");
             kill(getppid(), SIGKILL);
             /* убиваем зациклившегося отца */
        }
        return 0;
    }
```

В данном случае программа реализуется в двух процессах. Как и в предыдущем примере, имеется функция реакции на сигнал **alr()**, которая выводит на экран сообщение и переустанавливает функцию реакции на сигнал, опять же на себя. В основной программе мы также указываем **alr()** как реакцию на **SIGALRM**. После этого мы запускаем сыновний процесс, и отцовский процесс (бесконечный цикл) "засыпает" на 5 единиц времени, после чего сыновнему процессу будет отправлен сигнал **SIGALRM.** Все, что ниже цикла, будет выполняться в процессе-сыне: мы ожидаем ввода

строки, если ввод осуществлен, то происходит уничтожение отца (**SIGKILL**).

5.2 Надежные сигналы.

Вышеописанная реализация механизма сигналов имела место в ранних версиях UNIX (UNIX System V.2 и раньше). Позднее эта реализация подверглась критике за недостаточную надежность. В частности, это касалось сброса диспозиции перехваченного сигнала в реакцию по умолчанию всякий раз перед вызовом функции-обработчика. Хотя и существует возможность заново установить реакцию на сигнал в функции-обработчике, возможна ситуация, когда между моментом вызова пользовательского обработчика некоторого сигнала и моментом, когда он восстановит нужную реакцию на этот сигнал, будет получен еще один экземпляр того же сигнала. В этом случае второй экземпляр не будет перехвачен, так как на момент его прихода для данного сигнала действует реакция по умолчанию.

Поэтому в новых версиях UNIX (BSD UNIX 4.2 и System V.4) была реализована альтернативная модель так называемых надежных сигналов, которая вошла и в стандарт POSIX. В этой модели при перехватывании сигнала ядро не меняет его диспозицию, тем самым появляется гарантия перехвата всех экземпляров сигнала. Кроме того, чтобы избежать нежелательных эффектов при рекурсивном вызове обработчика для множества экземпляров одного и того же сигнала, ядро блокирует доставку других экземпляров того же сигнала в процесс до тех пор, пока функция-обработчик не завершит свое выполнение.

В модели надежных сигналов также появилась возможность на время блокировать доставку того или иного вида сигналов в процесс. Отличие блокировки сигнала от игнорирования в том, что пришедшие экземпляры сигнала не будут потеряны, а произойдет лишь откладывание их обработки на тот период времени, пока процесс не разблокирует данный сигнал. Таким образом процесс может оградить себя от прерывания сигналом на тот период, когда он выполняет какие-либо критические операции. Для реализации механизма блокирования вводится понятие **сигнальной маски**, которая описывает, какие сигналы из посылаемых процессу блокируются. Процесс наследует свою сигнальную маску от

родителя при порождении[9], и имеет возможность изменять ее в процессе своего выполнения.

Рассмотрим системные вызовы для работы с сигнальной маской процесса. Сигнальная маска описывается битовым полем типа **sigset_t**. Для управления сигнальной маской процесса служит системный вызов:

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set,
sigset_t *old_set);
```

Значения аргумента **how** влияют на характер изменения маски сигналов:

SIG_BLOCK – к текущей маске добавляются сигналы, указанные в наборе **set**;

SIG_UNBLOCK – из текущей маски удаляются сигналы, указанные в наборе **set**;

SIG_SETMASK – текущая маска заменяется на набор **set**.

Если в качестве аргумента set передается NULL-указатель, то сигнальная маска не изменяется, значение аргумента **how** при этом игнорируется. В последнем аргументе возвращается прежнее значение сигнальной маски до изменения ее вызовом **sigprocmask()**. Если процесс не интересуется прежним значением маски, он может передать в качестве этого аргумента NULL-указатель.

Если один или несколько заблокированных сигналов будут разблокированы посредством вызова **sigprocmask()**, то для их обработки будет использована диспозиция сигналов, действовавшая до вызова **sigprocmask()**. Если за время блокирования процессу пришло несколько экземпляров одного и того же сигнала, то ответ на вопрос о том, сколько экземпляров сигнала будет доставлено – все или один – зависит от реализации конкретной ОС.

Существует ряд вспомогательных функций, используемых для того, чтобы сформировать битовое поле типа **sigset_t** нужного вида:

- Инициализация битового набора - очищение всех битов:

[9] Несмотря на это, как уже говорилось, сами сигналы, ожидающие своей обработки родительским процессом на момент порождения потомка, в том числе и блокированные, не наследуются потомком

```
#include <signal.h>
int sigemptyset(sigset_t *set);
```

- Противоположная предыдущей функция устанавливает все биты в наборе:

```
#include <signal.h>
int sigfillset(sigset_t *set);
```

- Две следующие функции позволяют добавить или удалить флаг, соответствующий сигналу, в наборе:

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

В качестве второго аргумента этим функциям передается номер сигнала

- Приведенная ниже функция проверяет, установлен ли в наборе флаг, соответствующий сигналу, указанному в качестве параметра:

```
#include <signal.h>
int sigismember(sigset_t *set, int signo);
```

Этот вызов возвращает 1, если в маске **set** установлен флаг, соответствующий сигналу **signo**, и 0 в противном случае.

Чтобы узнать, какие из заблокированных сигналов ожидают доставки, используется функция **sigpending():**

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

Через аргумент этого вызова возвращается набор сигналов, ожидающих доставки.

Пример 14. Работа с сигнальной маской.

В данном примере анализируется сигнальная маска процесса, и выдается сообщение о том, заблокирован ли сигнал **SIGINT**, и ожидает ли такой сигнал доставки в процесс. Для того, чтобы легче было увидеть в действии результаты данных операций, предусмотрена возможность добавить этот сигнал к сигнальной маске процесса и послать этот сигнал самому себе.

```
#include <signal.h>
#include <sys/types.h>
#include <stdlib.h>
```

```
#include <stdio.h>
int main(int argc, char **argv)
{
     sigset_t sigset;
     int fl;
     sigemptyset(&sigset);
     printf("Добавить SIGINT к текущей маске? (yes –
     1, no – 0) \n");
     scanf("%d", &fl);
     if (fl)
     {
          sigaddset(&sigset, SIGINT);
          sigprocmask(SIG_BLOCK, &sigset, NULL);
     }
     printf("Послать SIGINT? (yes – 1, no – 0)\n");
     scanf("%d", &fl);
     if (fl)
          kill(getpid(), SIGINT);
     if (sigprocmask(SIG_BLOCK, NULL, &sigset) == –
1)
     /* получаем сигнальную маску процесса. Так как
второй аргумент NULL, то первый аргумент
игнорируется */
     {
          printf("Ошибка при вызове
          sigprocmask()\n");
          return –1;
     }
     else if (sigismember(&sigset, SIGINT))
     /*проверяем наличие сигнала SIGINT в маске*/
     {
          printf("Сигнал SIGINT заблокирован! \n");
          sigemptyset(&sigset);
          if (sigpending(&sigset) == –1)
          /*узнаем сигналы, ожидающие доставки */
          {
```

```
                printf("Ошибка при вызове
                sigpending()\n");
                return -1;
            }
            printf("Сигнал SIGINT %s\n",
            sigismember(&sigset, SIGINT) ? "ожидает
            доставки" : "не ожидает доставки");
            /*проверяем наличие сигнала SIGINT в
        маске*/
        }
        else printf("Сигнал SIGINT не заблокирован.
        \n");
        return 0;
    }
```

Для управления работой сигналов используется функция, аналогичная функции **signal()** в реализации обычных сигналов, но более мощная, позволяющая установить обработку сигнала, узнать ее текущее значение, приостановить получение сигналов:

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

Аргументами данного вызова являются: номер сигнала, структура, описывающая новую реакцию на сигнал, и структура, через которую возвращается прежний метод обработки сигнала. Если процесс не интересуется прежней обработкой сигнала, он может передать в качестве последнего параметра NULL-указатель.

Структура **sigaction** содержит информацию, необходимую для управления сигналами. Ее полями являются :

```
void (*sa_handler) (int),
void (sa_sigaction) (int, siginfo_t*, void*),
sigset_t sa_mask,
int sa_flags
```

Здесь поле **sa_handler** — функция-обработчик сигнала, либо константы **SIG_IGN** или **SIG_DFL**, говорящие соответственно о том, что необходимо игнорировать сигнал или установить обработчик по умолчанию. В поле **sa_mask** указывается набор сигналов, которые будут добавлены к маске сигналов на время работы функции-обработчика. Сигнал, для которого устанавливается функция-обработчик, также будет заблокирован на время ее работы. При

возврате из функции-обработчика маска сигналов возвращается в первоначальное состояние. В последнем поле указываются флаги, модифицирующие доставку сигнала. Одним из них может быть флаг **SA_SIGINFO**. Если он установлен, то при получении этого сигнала будет вызван обработчик **sa_sigaction**, ему помимо номера сигнала также будет передана дополнительная информация о причинах получения сигнала и указатель на контекст процесса.

Итак, «надежные» сигналы являются более мощным средством межпроцессного взаимодействия нежели обычные сигналы. В частности, здесь ликвидированы такие недостатки, как необходимость восстанавливать функцию-обработчик после получения сигнала, имеется возможность отложить получение сигнала на время выполнения критического участка программы, имеются большие возможности получения информации о причине отправления сигнала.

Пример 15. Использование надежных сигналов.

При получении сигнала **SIGINT** четырежды вызывается установленный обработчик, а в пятый раз происходит обработка по умолчанию.

```
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>

int count = 1;
struct sigaction action, sa;

void SigHandler(int s)
{
     printf("\nI got SIGINT %d time(s)\n", count++);
     if (count == 5)
     {
          action.sa_handler = SIG_DFL;
          /* изменяем указатель на функцию-
          обработчик сигнала */
          sigaction(SIGINT, &action, &sa);
          /* изменяем обработчик для сигнала SIGINT
          */
     }
```

```
}

int main(int argc, char **argv)
{
     sigset_t sigset;
     sigemptyset(&sigset);  /*  инициализируем  набор
     сигналов */
     sigaddset(&sigset, SIGINT); /*добавляем в набор
     сигналов бит, соответствующий сигналу SIGINT*/
     if  (sigprocmask(SIG_UNBLOCK,  &sigset,  NULL)  ==
     -1)
     /*устанавливаем новую сигнальную маску*/
     {
          printf("sigprocmask() error\n");
          return -1;
     }
     action.sa_handler = SigHandler;
     /* инициализируем указатель на функцию-
     обработчик сигнала*/
     sigaction(SIGINT, &action, &sa);
     /* изменяем обработчик по умолчанию для сигнала
     SIGINT */
     while(1); /* бесконечный цикл */
     return 0;
}
```

5.3 Программные 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 некоторое ограниченное пространство и возвращает чебрез 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**.** 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 существование канала необходим процесс, который его создаст и в котором он будет существовать, для файла это не так.
- Канал реализует модель последовательного доступа к данным (FIFO), т.е. данные из канала можно прочитать только в той же последовательности, в каком они были записаны. Это означает, что для файловых дескрипторов, ассоциированных с каналом, не определена операция **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 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, в него помещается символ EOF. После этого процесс, осуществляющий чтение, может выбрать из канала все оставшиеся данные и признак конца файла, благодаря которому 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, будет записано возможное количество данных, и управление будет сразу возвращено процессу.
- 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 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 16. 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;
}
```

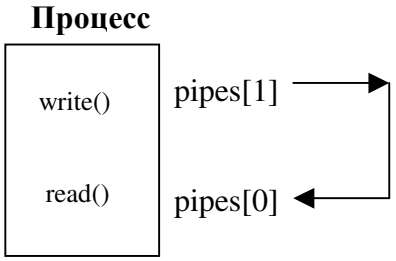

Рис. 15 Обмен через канал в рамках одного процесса.

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. 16 и Пример 17).

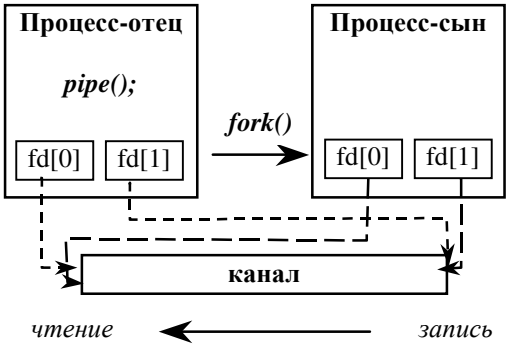

Рис. 16 Пример обмена данными между процессами через канал.

Пример 17. 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 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[10], 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 17 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 18. Реализация конвейера.

Пример реализации конвейера **`print|wc –`** вывод программы **`print`** будет подаваться на вход программы **`wc`**. Программа **`print`** печатает некоторый текст. Программа **`wc`** считает количество прочитанных строк, слов и символов.

```
#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())
```

[10] это правило не является обязательным, но для корректной организации двустороннего обмена через один канал требуется дополнительная синхронизация

```
    {
        /*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 */
}
```

Пример 19. Совместное использование сигналов и каналов – «пинг-понг».

Пример программы с использованием каналов и сигналов для осуществления связи между процессами – весьма типичной ситуации в системе. При этом на канал возлагается роль среды двусторонней передачи информации, а на сигналы – роль системы синхронизации при передаче информации. Процессы посылают друг другу целое число, всякий раз увеличивая его на 1. Когда число достигнет некоего максимума, оба процесса завершаются.

```
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
```

```
#include <stdio.h>

#define MAX_CNT 100
int target_pid, cnt;
int fd[2];
int status;

void SigHndlr(int s)
{
     /* в обработчике сигнала происходит и чтение, и
     запись */
     signal(SIGUSR1, SigHndlr);

     if (cnt < MAX_CNT)
     {
          read(fd[0], &cnt, sizeof(int));
          printf("%d \n", cnt);
          cnt++;
          write(fd[1], &cnt, sizeof(int));
          /* посылаем сигнал второму: пора читать из
          канала */
          kill(target_pid, SIGUSR1);
     }
     else
          if (target_pid == getppid())
          {
               /* условие окончания игры проверяется
               потомком */
               printf("Child is going to be
               terminated\n");
               close(fd[1]); close(fd[0]);
               /* завершается потомок */
               exit(0);
          } else
               kill(target_pid, SIGUSR1);
}
```

```
int main(int argc, char **argv)
{
     pipe(fd); /* организован канал */
     signal (SIGUSR1, SigHndlr);
     /* установлен обработчик сигнала для обоих
     процессов */
     cnt = 0;

     if (target_pid = fork())
     {
          /* Предку остается только ждать завершения
          потомка */
          wait(&status);
          printf("Parent is going to be
          terminated\n");
          close(fd[1]); close(fd[0]);
          return 0;
     }
     else
     {
          /* процесс-потомок узнает PID родителя */
          target_pid = getppid();
          /* потомок начинает пинг-понг */
          write(fd[1], &cnt, sizeof(int));
          kill(target_pid, SIGUSR1);
          for(;;); /* бесконечный цикл */
     }
}
```

5.4 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 в UNIX System V.3 и ранее используется системный вызов **mknod()**, а в BSD UNIX и System V.4 – вызов **mkfifo()** (этот вызов поддерживается и стандартом POSIX):

```
#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 установит 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

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

Ниже рассматривается пример, где один из процессов является сервером, предоставляющим некоторую услугу, другой же процесс, который хочет воспользоваться этой услугой, является клиентом. Клиент посылает серверу запросы на предоставление услуги, а сервер отвечает на эти запросы.

Пример 20. Модель «клиент-сервер».

Процесс-сервер запускается на выполнение первым, создает именованный канал, открывает его на чтение в неблокирующем режиме и входит в цикл, пытаясь прочесть что-либо. Затем запускается процесс-клиент, подключается к каналу с известным ему именем и записывает в него свой идентификатор. Сервер выходит из цикла, прочитав идентификатор клиента, и печатает его.

```
/* процесс-сервер*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char **argv)
{
     int fd;
     int pid;
     mkfifo("fifo", S_IFIFO | 0666);
     /*создали специальный файл FIFO с открытыми для
     всех правами доступа на чтение и запись*/

     fd = open("fifo", O_RDONLY | O_NONBLOCK);
     /* открыли канал на чтение*/
     while  (read (fd, &pid, sizeof(int)) == -1) ;
     printf("Server %d  got message from %d !\n",
     getpid(), pid);
     close(fd);
```

```
        unlink("fifo");/*уничтожили именованный канал*/
        return 0;
}

/* процесс-клиент*/
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
int main(int argc, char **argv)
{
        int fd;
        int pid = getpid( );
        fd = open("fifo", O_RDWR);
        write(fd, &pid, sizeof(int));
        close(fd);
        return 0;
}
```

5.5 Нелокальные переходы.

Рассмотрим некоторые дополнительные возможности по организации управления ходом процесса в UNIX, а именно возможность передачи управления в точку, расположенную вне данной функции.

Как известно, оператор **`goto`** позволяет осуществлять безусловный переход только внутри одной функции. Это ограничение связано с необходимостью сохранения целостности стека: в момент входа в функцию в стеке отводится место, называемое стековым кадром, где записываются адрес возврата, фактические параметры, отводится место под автоматические переменные. Стековый кадр освобождается при выходе из функции. Соответственно, если при выполнении безусловного перехода процесс минует тот фрагмент кода, где происходит освобождение стекового кадра, и управление непосредственно перейдет в другую часть программы (например, в объемлющую функцию), то фактическое состояние стека не будет соответствовать текущему участку кода, и тем самым стек подвергнется разрушению.

Однако, такое ограничение в некоторых случаях создает большое неудобство: например, в случае возникновения ошибки в рекурсивной функции, после обработки ошибки имеет смысл перейти в основную функцию, которая может находиться на несколько уровней вложенности выше текущей. Поскольку такой переход невозможно осуществить ни оператором **return**, ни оператором **goto**, программист будет вынужден создавать какие-то громоздкие структуры для обработки ошибок на каждом уровне вложенности.

Возможность передавать управление в точку, находящуюся в одной из вызывающих функций, предоставляется двумя системными вызовами, реализующими механизм нелокальных переходов:

```
#include <setjmp.h>
int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

Вызов **setjmp()** используется для регистрации некоторой точки кода, которая в дальнейшем будет использоваться в качестве пункта назначения для нелокального перехода, а вызов **longjmp()** – для перехода в одну из ранее зарегистрированных конечных точек.

При обращении к вызову **setjmp()**, происходит сохранение параметров текущей точки кода (значения счетчика адреса, позиции стека, регистров процессора и реакций на сигналы). Все эти значения сохраняются в структуре типа **jmp_buf**, которая передается вызову **setjmp()** в качестве параметра. При этом вызов **setjmp()** возвращает 0.

После того, как нужная точка кода зарегистрирована с помощью вызова **setjmp()**, управление в нее может быть передано при помощи вызова **longjmp()**. При этом в качестве первого параметра ему указывается та структура, в которой были зафиксированы атрибуты нужной нам точки назначения. После осуществления вызова **longjmp()** процесс продолжит выполнение с зафиксированной точки кода, т.е. с того места, где происходит возврат из функции **setjmp()**, но в отличие от первого обращения к **setjmp()**, возвращающим значением **setjmp()** станет не 0, а значение параметра **val** в вызове **longjmp()**, который произвел переход.

Отметим, что если программист желает определить в программе несколько точек назначения для нелокальных переходов, каждая из них должна быть зарегистрирована в своей структуре типа **jmp_buf**. С другой стороны, разумеется, на одну и ту же точку

назначения можно переходить из разных мест программы, при этом, чтобы различить, из какой точки был произведен нелокальный переход, следует указывать при переходах разные значения параметра **val**. В любом случае, при вызове **longjmp()** значение параметра **val** не должно быть нулевым (даже если оно есть 0, то возвращаемое значение **setjmp()** будет установлено в 1). Кроме того, переход должен производиться только на такие точки, которые находятся в коде одной из вызывающих функций для той функции, откуда осуществляется переход (в том числе, переход может быть произведен из функции-обработчика сигнала). При этом в момент перехода все содержимое стека, используемое текущей функцией и всеми вызывающими, вплоть до необходимой, освобождается.

Пример 21. Использование нелокальных переходов.

```
#include <signal.h>
#include <setjmp.h>
jmp_buf env;
void abc(int s)
{
     …
     longjmp(env,1);    /*переход  – в точку *** */
}

int main(int argc, char **argv)
{
     …
     if (setjmp(env) == 0)
     /* запоминается данная точка процесса – *** */
     {
          signal(SIGINT,abc);    /* установка реакции
          на сигнал */
          …
          /* цикл обработки данных после вызова
          функции setjmp() */
     }
     else
     {
          …
```

```
            /* цикл обработки данных после возврата из
            обработчика сигнала */
        }
        ...
}
```

5.6 Трассировка процессов.

Обзор форм межпроцессного взаимодействия в UNIX был бы не полон, если бы мы не рассмотрели простейшую форму взаимодействия, используемую для отладки — трассировку процессов. Принципиальное отличие трассировки от остальных видов межпроцессного взаимодействия в том, что она реализует модель «главный-подчиненный»: один процесс получает возможность управлять ходом выполнения, а также данными и кодом другого.

В UNIX трассировка возможна только между родственными процессами: процесс-родитель может вести трассировку только непосредственно порожденных им потомков, при этом трассировка начинается только после того, как процесс-потомок дает разрешение на это.

Далее схема взаимодействия процессов путем трассировки такова: выполнение отлаживаемого процесса-потомка приостанавливается всякий раз при получении им какого-либо сигнала, а также при выполнении вызова **exec()**. Если в это время отлаживающий процесс осуществляет системный вызов **wait()**, этот вызов немедленно возвращает управление. В то время, как трассируемый процесс находится в приостановленном состоянии, процесс-отладчик имеет возможность анализировать и изменять данные в адресном пространстве отлаживаемого процесса и в пользовательской составляющей его контекста. Далее, процесс-отладчик возобновляет выполнение трассируемого процесса до следующей приостановки (либо, при пошаговом выполнении, для выполнения одной инструкции).

Основной системный вызов, используемый при трассировке,– это **ptrace()**, прототип которого выглядит следующим образом:

```
#include <sys/ptrace.h>
int ptrace(int cmd, pid, addr, data);
```

где **cmd** – код выполняемой команды, **pid –** идентификатор процесса-потомка, **addr** – некоторый адрес в адресном пространстве процесса-потомка, **data** – слово информации.

Чтобы оценить уровень предоставляемых возможностей, рассмотрим основные коды - **cmd** операций этой функции.

cmd = PTRACE_TRACEME — ptrace() с таким кодом операции сыновний процесс вызывает в самом начале своей работы, позволяя тем самым трассировать себя**.** Все остальные обращения к вызову **ptrace()** осуществляет процесс-отладчик.

cmd = PTRACE_PEEKDATA — чтение слова из адресного пространства отлаживаемого процесса по адресу **addr**, **ptrace()** возвращает значение этого слова.

cmd = PTRACE_PEEKUSER — чтение слова из контекста процесса**.** Речь идет о доступе к пользовательской составляющей контекста данного процесса, сгруппированной в некоторую структуру, описанную в заголовочном файле **<sys/user.h>**. В этом случае параметр **addr** указывает смещение относительно начала этой структуры. В этой структуре размещена такая информация, как регистры, текущее состояние процесса, счетчик адреса и так далее. **ptrace()** возвращает значение считанного слова.

cmd = PTRACE_POKEDATA — запись данных, размещенных в параметре **data**, по адресу **addr** в адресном пространстве процесса-потомка.

cmd = PTRACE_POKEUSER — запись слова из **data** в контекст трассируемого процесса со смещением **addr**. Таким образом можно, например, изменить счетчик адреса трассируемого процесса, и при последующем возобновлении трассируемого процесса его выполнение начнется с инструкции, находящейся по заданному адресу.

cmd = PTRACE_GETREGS, PTRACE_GETFREGS — чтение регистров общего назначения (в т.ч. с плавающей точкой) трассируемого процесса и запись их значения по адресу **data**.

cmd = PTRACE_SETREGS, PTRACE_SETFREGS — запись в регистры общего назначения (в т.ч. с плавающей точкой) трассируемого процесса данных, расположенных по адресу **data** в трассирующем процессе.

cmd = PTRACE_CONT — возобновление выполнения трассируемого процесса**.** Отлаживаемый процесс будет выполняться до тех пор, пока не получит какой-либо сигнал, либо пока не завершится.

cmd = PTRACE_SYSCALL, PTRACE_SINGLESTEP — эта команда, аналогично **PTRACE_CONT**, возобновляет выполнение трассируемой

программы, но при этом произойдет ее остановка после того, как выполнится одна инструкция. Таким образом, используя **PTRACE_SINGLESTEP**, можно организовать пошаговую отладку. С помощью команды **PTRACE_SYSCALL** возобновляется выполнение трассируемой программы вплоть до ближайшего входа или выхода из системного вызова. Идея использования **PTRACE_SYSCALL** в том, чтобы иметь возможность контролировать значения аргументов, переданных в системный вызов трассируемым процессом, и возвращаемое значение, переданное ему из системного вызова.

cmd = PTRACE_KILL — завершение выполнения трассируемого процесса.

Пример 22. Общая схема использования механизма трассировки.

Рассмотрим некоторый модельный пример, демонстрирующий общую схему построения отладочной программы (см. также Рис. 17):

```
...
if ((pid = fork()) == 0)
{
     ptrace(PTRACE_TRACEME, 0, 0, 0);
     /* сыновний процесс разрешает трассировать себя
     */
     exec("трассируемый процесс", 0);
     /*   замещается   телом   процесса,   который
     необходимо трассировать */
}
else
{
     /* это процесс, управляющий трассировкой */
     wait((int ) 0);
     /* процесс приостанавливается до тех пор, пока
     от трассируемого процесса не придет сообщение о
     том, что он приостановился */

     for(;;)
     {
          ptrace(PTRACE_SINGLESTEP, pid, 0, 0);
```

```
            /* возобновляем выполнение трассируемой
            программы */
            wait((int ) 0);
            /* процесс приостанавливается до тех пор,
            пока от трассируемого процесса не придет
            сообщение о том, что он приостановился */
            …
            ptrace(cmd, pid, addr, data);
            /* теперь выполняются любые действия над
            трассируемым процессом */
            …
        }
    }
```

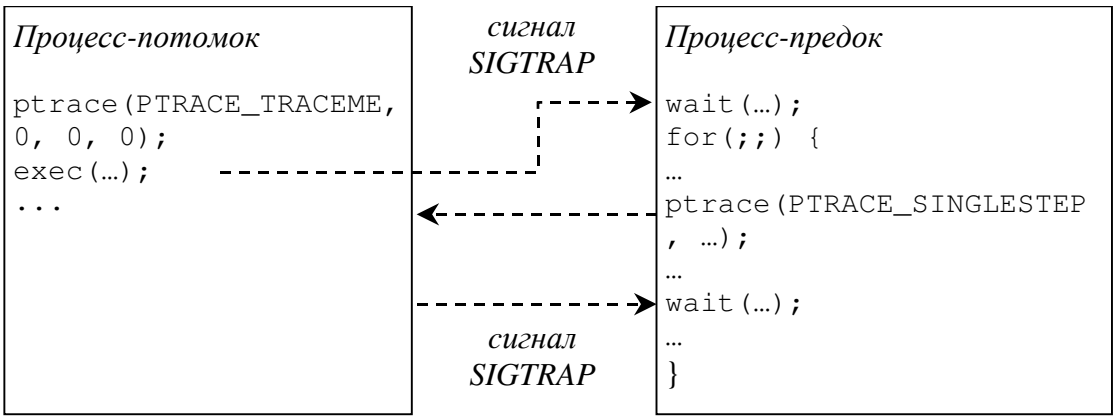

Рис. 17 Общая схема трассировки процессов

Предназначение процесса-потомка — разрешить трассировку себя. После вызова **ptrace(PTRACE_TRACEME, 0, 0, 0)** ядро устанавливает для этого процесса бит трассировки. Сразу же после этого можно заместить код процесса-потомка кодом программы, которую необходимо отладить. Отметим, что при выполнении системного вызова **exec()**, если для данного процесса ранее был установлен бит трассировки, ядро перед передачей управления в новую программу посылает процессу сигнал **SIGTRAP**. При получении данного сигнала трассируемый процесс приостанавливается, и ядро передает управление процессу-отладчику, выводя его из ожидания в вызове **wait()**.

Процесс-родитель вызывает **wait()** и переходит в состояние ожидания до того момента, пока потомок не перейдет в состояние трассировки. Проснувшись, управляющий процесс, выполняя функцию **ptrace(cmd, pid, addr, data)** с различными кодами операций, может производить любое действие с трассируемой

программой, в частности, читать и записывать данные в адресном пространстве трассируемого процесса, а также разрешать дальнейшее выполнение трассируемого процесса или производить его пошаговое выполнение. Схема пошаговой отладки показана в примере выше и на рисунке: на каждом шаге процесс-отладчик разрешает выполнение очередной инструкции отлаживаемого процесса и затем вызывает **wait()** и погружается в состояние ожидания, а ядро возобновляет выполнение трассируемого потомка, исполняет трассируемую команду и вновь передает управление отладчику, выводя его из ожидания .

Пример 23. Трассировка процессов.

```
/* Процесс-сын: */
int main(int argc, char **argv)
{
     /* деление на ноль - здесь процессу будет
     послан сигнал SIGFPE - floating point exception
     */
     return argc/0;
}

Процесс-родитель:
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>
#include <sys/ptrace.h>
#include <sys/user.h>
#include <sys/wait.h>
int main(int argc, char *argv[])
{
     pid_t pid;
     int status;
     struct user_regs_struct REG;
     if  ((pid = fork()) == 0) {
          /*находимся в процессе-потомке, разрешаем
          трассировку */
          ptrace(PTRACE_TRACEME, 0, 0, 0);
```

```
            execl("son", "son", 0);   /* замещаем тело
            процесса */
            /* здесь процесс-потомок будет остановлен
            с сигналом SIG_TRAP, ожидая команды
            продолжения выполнения от управляющего
            процесса*/
        }
        /* в процессе-родителе */
        while (1) {
            /* ждем, когда отлаживаемый процесс
            приостановится */
            wait(&status);
            /*читаем содержимое регистров
            отлаживаемого процесса */
            ptrace(PTRACE_GETREGS, pid, &REG, &REG);
            /* выводим статус отлаживаемого процесса,
            номер сигнала, который его остановил и
            значения прочитанных регистров  */
            printf("signal = %d, status = %#x, EIP=%#x
            ESP=%#x\n",    WSTOPSIG(status),    status,
            REG.eip, REG.esp);
            if (WSTOPSIG(status) != SIGTRAP) {
                if (!WIFEXITED(status)) {
                    /*      завершаем      выполнение
                    трассируемого процесса */
                    ptrace (PTRACE_KILL, pid, 0, 0);
                }
                break;
            }
            /*   разрешаем   выполнение   трассируемому
            процессу */
            ptrace (PTRACE_CONT, pid, 0, 0);
        }
}
```

6 Средства межпроцессного взаимодействия System V.

Все рассмотренные выше средства взаимодействия процессов не обладают достаточной универсальностью и имеют те или иные недостатки: так, сигналы несут в себе слишком мало информации и не могут использоваться для передачи сколь либо значительных объемов данных; неименованный канал должен быть создан до того, как порождается процесс, который будет осуществлять коммуникацию; именованный канал, хотя и лишен этого недостатка, требует одновременной работы с ним обоих процессов, участвующих в коммуникации.

Поэтому практически все UNIX-системы поддерживают более мощные и развитые средства межпроцессного взаимодействия, хотя появление и развитие этих средств в разных версиях UNIX шло разными путями.. Рассматриваемую в этой главе группу средств межпроцессного взаимодействия называют System V IPC[11], так как изначально эти средства были реализованы именно в UNIX System V, однако теперь она реализована во всех версиях UNIX. Она включает в себя **очереди сообщений**, **семафоры, разделяемую память**.

Следует отметить, что объекты и методы IPC поддерживаются стандартом POSIX (хотя они определены не в самом стандарте POSIX.1, а в стандарте POSIX.1b, описывающем переносимую ОС реального времени), однако, синтаксис их отличается от предложенного в System V, в частности, используется другое пространство имен. Далее в этой главе мы рассмотрим методы IPC так, как они реализованы в System V.

6.1 *Организация доступа и именования в разделяемых ресурсах.*

6.1.1 Именование разделяемых объектов.

Для всех средств IPC приняты общие правила именования объектов, позволяющие процессу получить доступ к такому объекту. Для именования объекта IPC используется ключ, представляющий собой целое число. Ключи являются уникальными во всей UNIX-системе идентификаторами объектов IPC, и зная ключ для некоторого объекта, процесс может получить к нему доступ. При этом процессу возвращается дескриптор объекта, который в

[11] аббревиатура IPC – это сокращение от английского interprocess communication

дальнейшем используется для всех операций с ним. Проведя аналогию с файловой системой, можно сказать, что ключ аналогичен имени файла, а получаемый по ключу дескриптор – файловому дескриптору, получаемому во время операции открытия файла. Ключ для каждого объекта IPC задается в момент его создания тем процессом, который его порождает, а все процессы, желающие получить в дальнейшем доступ к этому объекту, должны указывать тот же самый ключ.

Итак, все процессы, которые хотят работать с одним и тем же IPC-ресурсом, должны знать некий целочисленный ключ, по которому можно получить к нему доступ. В принципе, программист, пишущий программы для работы с разделяемым ресурсом, может просто жестко указать в программе некоторое константное значение ключа для именования разделяемого ресурса. Однако, возможна ситуация, когда к моменту запуска такой программы в системе уже существует разделяемый ресурс с таким значением ключа, и в виду того, что ключи должны быть уникальными во всей системе, попытка породить второй ресурс с таким же ключом закончится неудачей (подробнее этот момент будет рассмотрен ниже).

6.1.2 Генерация ключей: функция ftok().

Как видно, встает проблема именования разделяемого ресурса: необходим некий механизм получения заведомо уникального ключа для именования ресурса, но вместе с тем нужно, чтобы этот механизм позволял всем процессам, желающим работать с одним ресурсом, получить одно и то же значение ключа.

Для решения этой задачи служит функция **`ftok()`**:

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(char *filename, char proj);
```

Эта функция генерирует значение ключа по некоторой строке символов и добавочному символу, передаваемым в качестве параметров. Гарантируется, что полученное таким образом значение будет отличаться от всех других значений, сгенерированных функцией **`ftok()`** с другими значениями параметров, и в то же время, при повторном запуске **`ftok()`** с теми же параметрами, будет получено то же самое значение ключа.

Смысл второго аргумента функции **`ftok()`** – добавочного символа – в том, что он позволяет генерировать разные значения ключа по одному и тому же значению первого параметра – строки.

Это позволяет программисту поддерживать несколько версий своей программы, которые будут использовать одну и ту же строку, но разные добавочные символы для генерации ключа, и тем самым получат возможность в рамках одной системы работать с разными разделяемыми ресурсами.

Следует заметить, что функция **`ftok()`** не является системным вызовом, а предоставляется библиотекой.

6.1.3 Общие принципы работы с разделяемыми ресурсами.

Рассмотрим некоторые моменты, общие для работы со всеми разделяемыми ресурсами IPC. Как уже говорилось, общим для всех ресурсов является механизм именования. Кроме того, для каждого IPC-ресурса поддерживается идентификатор его владельца и структура, описывающая права доступа к нему. Подобно файлам, права доступа задаются отдельно для владельца, его группы и всех остальных пользователей; однако, в отличие от файлов, для разделяемых ресурсов поддерживается только две категории доступа: по чтению и записи. Априори считается, что возможность изменять свойства ресурса и удалять его имеется только у процесса, эффективный идентификатор пользователя которого совпадает с идентификатором владельца ресурса. Владельцем ресурса назначается пользователь, от имени которого выполнялся процесс, создавший ресурс, однако создатель может передать права владельца другому пользователю. В заголовочном файле **`<sys/ipc.h>`** определен тип **`struct ipc_perm`**, который описывает права доступа к любому IPC-ресурсу. Поля в этой структуре содержат информацию о создателе и владельце ресурса и их группах, правах доступа к ресурсу и его ключе.

Для создания разделяемого ресурса с заданным ключом, либо подключения к уже существующему ресурсу с таким ключом используются ряд системных вызовов, имеющих общий суффикс **`get`**. Общими параметрами для всех этих вызовов являются ключ и флаги. В качестве значения ключа при создании любого IPC-объекта может быть указано значение **`IPC_PRIVATE`**. При этом создается ресурс, который будет доступен только породившему его процессу. Такие ресурсы обычно порождаются родительским процессом, который затем сохраняет полученный дескриптор в некоторой переменной и порождает своих потомков. Так как потомкам доступен уже готовый дескриптор созданного объекта, они могут непосредственно работать с ним, не обращаясь предварительно к «**`get`**»-методу. Таким образом, созданный ресурс может совместно

использоваться родительским и порожденными процессами. Однако, важно понимать, что если один из этих процессов повторно вызовет «**get»-**метод с ключом **IPC_PRIVATE**, в результате будет получен другой, совершенно новый разделяемый ресурс, так как при обращении к «**get»-**методу с ключом **IPC_PRIVATE** всякий раз создается новый объект нужного типа.

Если при обращении к «**get»-**методу указан ключ, отличный от **IPC_PRIVATE**, происходит следующее:

- Происходит поиск объекта с заданным ключом среди уже существующих объектов нужного типа. Если объект с указанным ключом не найден, и среди флагов указан флаг **IPC_CREAT,** будет создан новый объект. При этом значение параметра флагов должно содержать побитовое сложение флага **IPC_CREAT** и константы, указывающей права доступа для вновь создаваемого объекта.
- Если объект с заданным ключом не найден, и среди переданных флагов отсутствует флаг **IPC_CREAT**, «**get»-**метод вернет –1, а в переменной **errno** будет установлено значение **ENOENT**
- Если объект с заданным ключом найден среди существующих, «**get»-**метод вернет дескриптор для этого существующего объекта, т.е. фактически, в этом случае происходит подключение к уже существующему объекту по заданному ключу. Если процесс ожидал создания нового объекта по указанному ключу, то для него такое поведение может оказаться нежелательным, так как это будет означать, что в результате случайного совпадения ключей (например, если процесс не использовал функцию **ftok()**) он подключился к чужому ресурсу. Чтобы избежать такой ситуации, следует указать в параметре флагов наряду с флагом **IPC_CREAT** и правами доступа еще и флаг **IPC_EXCL –** в этом случае «**get»-**метод вернет -1, если объект с таким ключом уже существует (переменная **errno** будет установлена в значение **EEXIST**)
- Следует отметить, что при подключении к уже существующему объекту дополнительно проверяются права доступа к нему. В случае, если процесс, запросивший доступ к объекту, не имеет на то прав, «**get»-**метод вернет –1, а в переменной **errno** будет установлено значение **EACCESS**

Нужно иметь в виду, что для каждого типа объектов IPC существует некое ограничение на максимально возможное количество одновременно существующих в системе объектов данного типа. Если при попытке создания нового объекта окажется, что указанное ограничение превышено, «**get»-**метод, совершавший попытку создания объекта, вернет -1, а в переменной **errno** будет указано значение **ENOSPC**.

Отметим, что даже если ни один процесс не подключен к разделяемому ресурсу, система не удаляет его автоматически. Удаление объектов IPC является обязанностью одного из работающих с ним процессов и для этого определена специальная функция. Для этого системой предоставляются соответствующие функции по управлению объектами System V IPC.

6.2 Очередь сообщений.

Итак, одним из типов объектов System V IPC являются очереди сообщений. Очередь сообщений представляет собой некое хранилище типизированных сообщений, организованное по принципу FIFO. Любой процесс может помещать новые сообщения в очередь и извлекать из очереди имеющиеся там сообщения. Каждое сообщение имеет тип, представляющий собой некоторое целое число. Благодаря наличию типов сообщений, очередь можно интерпретировать двояко — рассматривать ее либо как сквозную очередь неразличимых по типу сообщений, либо как некоторое объединение подочередей, каждая из которых содержит элементы определенного типа. Извлечение сообщений из очереди происходит согласно принципу FIFO – в порядке их записи, однако процесс-получатель может указать, из какой подочереди он хочет извлечь сообщение, или, иначе говоря, сообщение какого типа он желает получить – в этом случае из очереди будет извлечено самое «старое» сообщение нужного типа (см. Рис. 18).

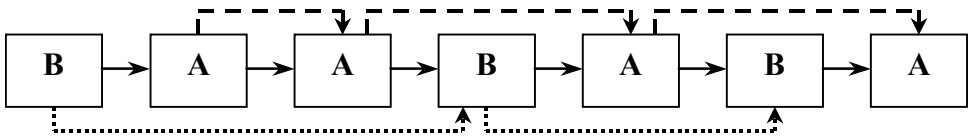

Рис. 18 Типизированные очереди сообщений

Рассмотрим набор системных вызовов, поддерживающий работу с очередями сообщений.

6.2.1 Доступ к очереди сообщений.

Для создания новой или для доступа к существующей используется системный вызов:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/message.h>
int msgget (key_t key, int msgflag)
```

В случае успеха вызов возвращает положительный дескриптор очереди, который может в дальнейшем использоваться для операций с ней, в случае неудачи -1. Первым аргументом вызова является ключ, вторым – флаги, управляющие поведением вызова. Подробнее детали процесса создания/подключения к ресурсу описаны выше.

6.2.2 Отправка сообщения.

Для отправки сообщения используется функция **msgsnd():**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd (int msqid, const void *msgp, size_t
msgsz, int msgflg)
```

Ее первый аргумент — идентификатор очереди, полученный в результате вызова **msgget()**. Второй аргумент — указатель на буфер, содержащий реальные данные и тип сообщения, подлежащего посылке в очередь, в третьем аргументе указывается размер буфера.

В качестве буфера необходимо указывать структуру, содержащую следующие поля (в указанном порядке):

long msgtype — тип сообщения

char msgtext[ ] — данные (тело сообщения)

В заголовочном файле **<sys/msg.h>** определена константа **MSGMAX**, описывающая максимальный размер тела сообщения. При попытке отправить сообщение, у которого число элементов в массиве **msgtext** превышает это значение, системный вызов вернет –1.

Четвертый аргумент данного вызова может принимать значения 0 или **IPC_NOWAIT**. В случае отсутствия флага **IPC_NOWAIT** вызывающий процесс будет блокирован (т.е. приостановит работу), если для посылки сообщения недостаточно системных ресурсов, т.е.

если полная длина сообщений в очереди будет больше максимально допустимого. Если же флаг **IPC_NOWAIT** будет установлен, то в такой ситуации выход из вызова произойдет немедленно, и возвращаемое значение будет равно –1.

В случае удачной записи возвращаемое значение вызова равно 0.

6.2.3 Получение сообщения.

Для получения сообщения имеется функция **msgrcv():**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgrcv (int msqid, void *msgp, size_t msgsz,
long msgtyp, int msgflg)
```

Первые три аргумента аналогичны аргументам предыдущего вызова: это дескриптор очереди, указатель на буфер, куда следует поместить данные, и максимальный размер (в байтах) тела сообщения, которое можно туда поместить. Буфер, используемый для приема сообщения, должен иметь структуру, описанную выше.

Четвертый аргумент указывает тип сообщения, которое процесс желает получить. Если значение этого аргумента есть 0, то будет получено сообщение любого типа. Если значение аргумента **msgtyp** больше 0, из очереди будет извлечено сообщение указанного типа. Если же значение аргумента **msgtyp** отрицательно, то тип принимаемого сообщения определяется как наименьшее значение среди типов, которые меньше модуля **msgtyp**. В любом случае, как уже говорилось, из подочереди с заданным типом (или из общей очереди, если тип не задан) будет выбрано самое старое сообщение.

Последним аргументом является комбинация (побитовое сложение) флагов. Если среди флагов не указан **IPC_NOWAIT**, и в очереди не найдено ни одного сообщения, удовлетворяющего критериям выбора, процесс будет заблокирован до появления такого сообщения. (Однако, если такое сообщение существует, но его длина превышает указанную в аргументе **msgsz**, то процесс заблокирован не будет, и вызов сразу вернет –1. Сообщение при этом останется в очереди). Если же флаг **IPC_NOWAIT** указан, то вызов сразу вернет –1.

Процесс может также указать флаг **MSG_NOERROR** – в этом случае он может прочитать сообщение, даже если его длина превышает указанную емкость буфера. В этом случае в буфер будет

записано первые **msgsz** байт из тела сообщения, а остальные данные отбрасываются.

В случае удачного чтения возвращаемое значение вызова равно 0.

6.2.4 Управление очередью сообщений.

Функция управления очередью сообщений выглядит следующим образом:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl(int msqid, int cmd, struct msgid_ds *buf)
```

Данный вызов используется для получения или изменения процессом управляющих параметров, связанных с очередью и уничтожения очереди. Ее аргументы — идентификатор ресурса, команда, которую необходимо выполнить, и структура, описывающая управляющие параметры очереди. Тип **msgid_ds** описан в заголовочном файле **<sys/message.h>**, и представляет собой структуру, в полях которой хранятся права доступа к очереди, статистика обращений к очереди, ее размер и т.п.

Возможные значения аргумента **cmd**:

IPC_STAT – скопировать структуру, описывающую управляющие параметры очереди по адресу, указанному в параметре **buf**

IPC_SET – заменить структуру, описывающую управляющие параметры очереди, на структуру, находящуюся по адресу, указанному в параметре **buf**

IPC_RMID – удалить очередь. Как уже говорилось, удалить очередь может только процесс, у которого эффективный идентификатор пользователя совпадает с владельцем или создателем очереди, либо процесс с правами привилегированного пользователя.

Пример 24. Использование очереди сообщений.

Пример программы, где основной процесс читает некоторую текстовую строку из стандартного ввода, и в случае, если строка начинается с буквы **'a'**, эта строка в качестве сообщения будет передана процессу **A**, если **'b'** - процессу **B,** если **'q'** - то процессам **A** и **B**, затем будет осуществлен выход. Процессы **A** и **B** распечатывают полученные строки на стандартный вывод.

6.2.4.1.1.1.1.1.1 Основной процесс.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <unistd.h>
#include <stdio.h>

struct {
     long mtype;    /* тип сообщения */
     char Data[256];     /* сообщение */
} Message;

int main(int argc, char **argv)
{
     key_t key;  int msgid;  char str[256];

     key = ftok("/usr/mash",'s');
     /*получаем    уникальный    ключ,    однозначно
     определяющий доступ к ресурсу */
     msgid=msgget(key, 0666 | IPC_CREAT);
     /*создаем очередь сообщений , 0666 определяет
     права доступа */

     for(;;) {
          /* запускаем вечный цикл */
          gets(str); /* читаем из стандартного ввода
          строку */
          strcpy(Message.Data, str);
          /* и копируем ее в буфер сообщения */
          switch(str[0]){
               case 'a':
               case 'A':
                    Message.mtype = 1;
                    /* устанавливаем тип */
                    msgsnd(msgid,  (struct  msgbuf*)
                    (&Message), strlen(str) + 1, 0);
```

```
                    /* посылаем сообщение в очередь
                    */
                    break;
               case 'b':
               case 'B':
                    Message.mtype = 2;
                    msgsnd(msgid, (struct msgbuf*)
                    (&Message), strlen(str) + 1, 0);
                    break;
               case 'q':
               case 'Q':
                    Message.mtype = 1;
                    msgsnd(msgid, (struct msgbuf*)
                    (&Message), strlen(str) + 1, 0);
                    Message.mtype = 2;
                    msgsnd(msgid, (struct msgbuf*)
                    (&Message), strlen(str) + 1, 0);
                    sleep(10);
                    /* ждем получения сообщений
                    процессами A и B */
                    msgctl(msgid, IPC_RMID, NULL);
                    /* уничтожаем очередь*/
                    return 0;
               default:
                    break;
          }
     }
}
```

Процесс–приемник A

```
/* процесс B аналогичен с точностью до четвертого
параметра в msgrcv */

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
```

```
struct {
     long mtype;
     char Data[256];
} Message;

int main(int argc, char **argv)
{
     key_t key;  int msgid;

     key = ftok("/usr/mash",'s');
     /* получаем ключ по тем же параметрам */
     msgid = msgget(key, 0666 | IPC_CREAT);
     /*подключаемся к очереди сообщений */
     for(;;) {
          /* запускаем вечный цикл */
          msgrcv(msgid, (struct msgbuf*) (&Message),
          256, 1, 0);
          /* читаем сообщение с типом 1*/
          if         (Message.Data[0]=='q'         ||
          Message.Data[0]=='Q') break;
          printf("\nПроцесс-приемник     A:     %s",
          Message.Data);
     }
     return 0;
}
```

Благодаря наличию типизации сообщений, очередь сообщений предоставляет возможность мультиплексировать сообщения от различных процессов, при этом каждая пара взаимодействующих через очередь процессов может использовать свой тип сообщений, и таким образом, их данные не будут смешиваться.

В качестве иллюстрации приведем следующий стандартный пример взаимодействия. Рассмотрим еще один пример - пусть существует **процесс-сервер** и несколько **процессов-клиентов**. Все они могут обмениваться данными, используя одну очередь сообщений. Для этого сообщениям, направляемым от клиента к серверу, присваиваем значение типа **1**. При этом процесс, отправивший сообщение, в его теле передает некоторую

информацию, позволяющую его однозначно идентифицировать. Тогда сервер, отправляя сообщение конкретному процессу, в качестве его типа указывает эту информацию (например, **PID** процесса). Таким образом, сервер будет читать из очереди только сообщения типа **1**, а клиенты — сообщения с типами, равными идентификаторам их процессов.

Пример 25. Очередь сообщений. Модель «клиент-сервер»

server

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>

int main(int argc, char **argv)
{
     struct {
          long mestype;
          char mes [100];
          } messageto;
     struct {
          long mestype;
          long mes;
          } messagefrom;

     key_t key;
     int mesid;

     key = ftok("example",'r');
     mesid = msgget (key, 0666 | IPC_CREAT);

     while(1)
     {
          if (msgrcv(mesid, &messagefrom,
          sizeof(messagefrom), 1, 0) <= 0) continue;
          messageto.mestype = messagefrom.mes;
```

```
            strcpy( messageto.mes, "Message for
            client");
            msgsnd (mesid, &messageto,
            sizeof(messageto), 0);
      }
      msgctl (mesid, IPC_RMID, 0);
      return 0;
}
```

client

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv)
{
      struct {
            long mestype;   /*описание структуры
            сообщения*/
            long mes;
            } messageto;
      struct {
            long mestype;   /*описание структуры
            сообщения*/
            char mes[100];
            } messagefrom;

      key_t key;
      int mesid;

      long pid = getpid();
      key = ftok("example", 'r');
      mesid = msgget(key, 0);   /*присоединение к
      очереди сообщений*/
```

```
    messageto.mestype = 1;
    messageto.mes = pid;

    msgsnd (mesid, &messageto, sizeof(messageto),
    0); /* отправка */

    while ( msgrcv (mesid, &messagefrom,
    sizeof(messagefrom), pid, 0) <= 0);
    /*прием сообщения */

    printf("%s\n", messagefrom.mes);
    return 0;
}
```

6.3 Разделяемая память

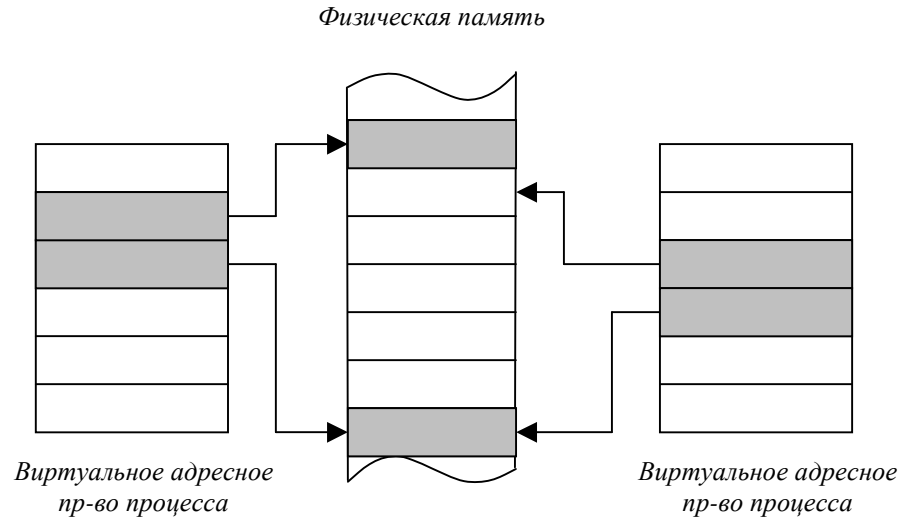

Рис. 19 Разделяемая память

Механизм разделяемой памяти позволяет нескольким процессам получить отображение некоторых страниц из своей виртуальной памяти на общую область физической памяти. Благодаря этому, данные, находящиеся в этой области памяти, будут доступны для чтения и модификации всем процессам, подключившимся к данной области памяти.

Процесс, подключившийся к разделяемой памяти, может затем получить указатель на некоторый адрес в своем виртуальном адресном пространстве, соответствующий данной области разделяемой памяти. После этого он может работать с этой областью

памяти аналогично тому, как если бы она была выделена динамически (например, путем обращения к **malloc()**), однако, как уже говорилось, сама по себе разделяемая область памяти не уничтожается автоматически даже после того, как процесс, создавший или использовавший ее, перестанет с ней работать.

Рассмотрим набор системных вызовов для работы с разделяемой памятью.

6.3.1 **Создание общей памяти.**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget (key_t key, int size, int shmemflg)
```

Аргументы этого вызова: **key -** ключ для доступа к разделяемой памяти; **size** задает размер области памяти, к которой процесс желает получить доступ. Если в результате вызова **shmget()** будет создана новая область разделяемой памяти, то ее размер будет соответствовать значению **size**. Если же процесс подключается к существующей области разделяемой памяти, то значение **size** должно быть не более ее размера, иначе вызов вернет –1. Заметим, что если процесс при подключении к существующей области разделяемой памяти указал в аргументе **size** значение, меньшее ее фактического размера, то впоследствии он сможет получить доступ только к первым **size** байтам этой области.

Отметим, что в заголовочном файле **<sys/shm.h>** определены константы **SHMMIN** и **SHMMAX**, задающий минимально возможный и максимально возможный размер области разделяемой памяти. Если процесс пытается создать область разделяемой памяти, размер которой не удовлетворяет этим границам, системный вызов **shmget()** окончится неудачей.

Третий параметр определяет флаги, управляющие поведением вызова. Подробнее алгоритм создания/подключения разделяемого ресурса был описан выше.

В случае успешного завершения вызов возвращает положительное число – дескриптор области памяти, в случае неудачи - -1.

6.3.2 **Доступ к разделяемой памяти.**

```
#include <sys/types.h>
#include <sys/ipc.h>
```

```
#include <sys/shm.h>
char *shmat(int shmid, char *shmaddr, int shmflg)
```

При помощи этого вызова процесс подсоединяет область разделяемой памяти, дескриптор которой указан в **shmid**, к своему виртуальному адресному пространству. После выполнения этой операции процесс сможет читать и модифицировать данные, находящиеся в области разделяемой памяти, адресуя ее как любую другую область в своем собственном виртуальном адресном пространстве.

В качестве второго аргумента процесс может указать виртуальный адрес в своем адресном пространстве, начиная с которого необходимо подсоединить разделяемую память. Чаще всего, однако, в качестве значения этого аргумента передается 0, что означает, что система сама может выбрать адрес начала разделяемой памяти. Передача конкретного адреса в этом параметре имеет смысл в том случае, если, к примеру, в разделяемую память записываются указатели на нее же (например, в ней хранится связанный список) – в этой ситуации для того, чтобы использование этих указателей имело смысл и было корректным для всех процессов, подключенных к памяти, важно, чтобы во всех процессах адрес начала области разделяемой памяти совпадал.

Третий аргумент представляет собой комбинацию флагов. В качестве значения этого аргумента может быть указан флаг **SHM_RDONLY**, который указывает на то, что подсоединяемая область будет использоваться только для чтения.

Эта функция возвращает адрес, начиная с которого будет отображаться присоединяемая разделяемая память. В случае неудачи вызов возвращает **-1**.

6.3.3 **Открепление разделяемой памяти.**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmdt(char *shmaddr)
```

Данный вызов позволяет отсоединить разделяемую память, ранее присоединенную посредством вызова **shmat()**.

Параметр **shmaddr** - адрес прикрепленной к процессу памяти, который был получен при вызове **shmat()**.

В случае успешного выполнения функция возвращает 0, в случае неудачи -1

6.3.4 Управление разделяемой памятью.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf)
```

Данный вызов используется для получения или изменения процессом управляющих параметров, связанных с областью разделяемой памяти, наложения и снятия блокировки на нее и ее уничтожения. Аргументы вызова — дескриптор области памяти, команда, которую необходимо выполнить, и структура, описывающая управляющие параметры области памяти. Тип **shmid_ds** описан в заголовочном файле **<sys/shm.h>**, и представляет собой структуру, в полях которой хранятся права доступа к области памяти, ее размер, число процессов, подсоединенных к ней в данный момент, и статистика обращений к области памяти.

Возможные значения аргумента **cmd**:

IPC_STAT – скопировать структуру, описывающую управляющие параметры области памяти по адресу, указанному в параметре **buf**

IPC_SET – заменить структуру, описывающую управляющие параметры области памяти, на структуру, находящуюся по адресу, указанному в параметре **buf**. Выполнить эту операцию может процесс, у которого эффективный идентификатор пользователя совпадает с владельцем или создателем очереди, либо процесс с правами привилегированного пользователя, при этом процесс может изменить только владельца области памяти и права доступа к ней.

IPC_RMID – удалить очередь. Как уже говорилось, удалить очередь может только процесс, у которого эффективный идентификатор пользователя совпадает с владельцем или создателем очереди, либо процесс с правами привилегированного пользователя.

SHM_LOCK, SHM_UNLOCK – блокировать или разблокировать область памяти. Выполнить эту операцию может только процесс с правами привилегированного пользователя.

Пример 26. Общая схема работы с общей памятью в рамках одного процесса.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int putm(char *);
int waitprocess(void);

int main(int argc, char **argv)
{
     key_t key;
     int shmid;
     char *shmaddr;

     key = ftok("/tmp/ter",'S');
     shmid = shmget(key, 100, 0666|IPC_CREAT);
     shmaddr = shmat(shmid, NULL, 0); /* подключение
     к памяти */

     putm(shmaddr); /* работа с ресурсом */
     waitprocess();

     shmctl(shmid,IPC_RMID,NULL);   /*   уничтожение
     ресурса */
     return 0;
}
```

В данном примере считается, что **`putm()`** и **`waitprocess()`** – некие пользовательские функции, определенные в другом месте

6.4 Семафоры.

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 уже говорилось, семафор представляет собой особый вид числовой переменной, над которой определены две неделимые операции: уменьшение ее значения с возможным блокированием процесса и увеличение значения с возможным разблокированием одного из ранее заблокированных процессов. Объект System V IPC представляет собой набор семафоров. 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

Рассмотрим набор вызовов для оперирования с семафорами в UNIX System V.

6.4.1 Доступ к семафору

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. Подробнее процесс создания разделяемого ресурса описан выше. 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.

6.4.2 Операции над семафором

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 */
}
```

Поле операции в структуре интерпретируется следующим образом:

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.

6.4.3 Управление массивом семафоров.

```
#include <sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
int semctl (int semid, int num, int cmd, union semun
arg)
```

С помощью этого системного вызова можно запрашивать и изменять управляющие параметры разделяемого ресурса, а также удалять его.

Первый параметр вызова – дескриптор массива семафоров. Параметр **num** представляет собой индекс семафора в массиве, параметр **cmd** задает операцию, которая должна быть выполнена над данным семафором. Последний аргумент имеет тип **union semun** и используется для считывания или задания управляющих параметров одного семафора или всего массива, в зависимости от значения аргумента **cmd**. Тип данных **union semun** определен в файле **<sys/sem.h>** и выглядит следующим образом:

```
union semun {
     int val;  // значение одного семафора
     struct semid_ds *buf;     /* параметры массива
семафоров в целом */
     ushort    *array;         /* массив значений
семафоров */
}
```

где **struct semid_ds** – структура, описанная в том же файле, в полях которой хранится информация о всем наборе семафоров в целом, а именно, количество семафоров в наборе, права доступа к нему и статистика доступа к массиву семафоров.

Приведем некоторые наиболее часто используемые значения аргумента **cmd**:

IPC_STAT – скопировать управляющие параметры набора семафоров по адресу **arg.buf**

IPC_SET – заменить управляющие параметры набора семафоров на те, которые указаны в **arg.buf**. Чтобы выполнить эту операцию, процесс должен быть владельцем или создателем массива семафоров, либо обладать правами привилегированного пользователя, при этом процесс может изменить только владельца массива семафоров и права доступа к нему.

IPC_RMID – удалить массив семафоров. Чтобы выполнить эту операцию, процесс должен быть владельцем или создателем массива семафоров, либо обладать правами привилегированного пользователя

GETALL, SETALL – считать / установить значения всех семафоров в массив, на который указывает **arg.array**

GETVAL – возвратить значение семафора с номером **num**. Последний аргумент вызова игнорируется.

SETVAL – установить значение семафора с номером num равным **arg.val**

В случае успешного завершения вызов возвращает значение, соответствующее конкретной выполнявшейся операции (0, если не оговорено иное), в случае неудачи – -1.

Пример 27. Работа с разделяемой памятью с синхронизацией семафорами.

Программа будет оперировать с разделяемой памятью.

1 процесс – создает ресурсы “разделяемая память” и “семафоры”, далее он начинает принимать строки со стандартного ввода и записывает их в разделяемую память.

2 процесс – читает строки из разделяемой памяти.

Таким образом мы имеем критический участок в момент, когда один процесс еще не дописал строку, а другой ее уже читает. Поэтому следует установить некоторые синхронизации и задержки.

1й процесс:

```
#include <stdio.h>
#include <sys/types.h>
```

```
#include <sys/ipc.h>
#include <sys/sem.h>
#include <string.h>
#define NMAX   256
int main(int argc, char **argv)
{
     key_t key;
     int semid, shmid;
     struct sembuf sops;
     char *shmaddr;
     char str[NMAX];

     key = ftok("/usr/ter/exmpl", 'S');
     /* создаем уникальный ключ */
     semid = semget(key, 1, 0666 | IPC_CREAT);
     /* создаем один семафор с определенными правами
     доступа */
     shmid = shmget(key, NMAX, 0666 | IPC_CREAT);
     /* создаем разделяемую память на 256 элементов
     */
     shmaddr = shmat(shmid, NULL, 0);
     /* подключаемся к разделу памяти, в shaddr -
     указатель на буфер с разделяемой памятью */
     semctl(semid,0,SETVAL, (int) 0);
     /* инициализируем семафор значением 0 */
     sops.sem_num = 0;
     sops.sem_flg = 0;
     do { /* запуск цикла */
          printf("Введите строку:");
          if (fgets(str, NMAX, stdin) == NULL)
          {
               /* окончание ввода */
               /* пишем признак завершения - строку
               "Q" */
               strcpy(str, "Q");
          }
```

```
            /* в текущий момент семафор открыт для
            этого процесса */
            strcpy(shmaddr, str); /* копируем строку в
            разд. память */
            /* предоставляем второму процессу
            возможность войти */
            sops.sem_op = 3; /* увеличение семафора на
            3 */
            semop(semid, &sops, 1);
            /* ждем, пока семафор будет открыт для 1го
            процесса -  для следующей итерации цикла
            */
            sops.sem_op = 0; /* ожидание обнуления
            семафора */
            semop(semid, &sops, 1);
      } while (str[0] != 'Q');
      /* в данный момент второй процесс уже дочитал
      из разделяемой памяти и отключился от нее -
      можно ее удалять*/
      shmdt(shmaddr) ; /* отключаемся от разделяемой
      памяти */
      shmctl(shmid, IPC_RMID, NULL);
      /* уничтожаем разделяемую память */
      semctl(semid, 0, IPC_RMID, (int) 0);
      /* уничтожаем семафор */
      return 0;
}
```

2й процесс:

```
/* необходимо корректно определить существование
ресурса, если он есть - подключиться */
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <string.h>
#define NMAX   256
```

```
int main(int argc, char **argv)
{
     key_t key;
     int semid, shmid;
     struct sembuf sops;
     char *shmaddr;
     char str[NMAX];

     key = ftok("/usr/ter/exmpl",'S');
     /* создаем тот же самый ключ */
     semid = semget(key, 1, 0666 | IPC_CREAT);
     shmid = shmget(key, NMAX, 0666 | IPC_CREAT);
     /* аналогично предыдущему процессу  -
     инициализации ресурсов */
     shmaddr = shmat(shmid, NULL, 0);
     sops.sem_num = 0;
     sops.sem_flg = 0;
     /* запускаем цикл */
     do {
          printf("Waiting… \n"); /* ожидание на
          семафоре */
          sops.sem_op = -2;
          /* будем ожидать, пока "значение семафора"
          + "значение sem_op" не станет
          положительным, т.е. пока значение семафора
          не станет как минимум 3 (3-2=1 > 0) */
          semop(semid, &sops, 1);
          /* теперь значение семафора равно 1 */
          strcpy(str, shmaddr); /* копируем строку
          из разд.памяти */
          /*критическая секция - работа с
          разделяемой памятью - в этот момент
          первый процесс к разделяемой памяти
          доступа не имеет*/
          if (str[0] == 'Q')
          {
                 /*завершение работы - освобождаем
                 разделяемую память */
```

```
                shmdt(shmaddr);
            }
            /*после работы – обнулим семафор*/
            sops.sem_op=-1;
            semop(semid, &sops, 1);
            printf("Read from shared memory: %s\n",
            str);
        } while (str[0] != 'Q');
        return 0;
    }
```

Отметим, что данный пример демонстрирует два разных приема использования семафоров для синхронизации: первый процесс блокируется в ожидании обнуления семафора, т.е. для того, чтобы он мог войти в критическую секцию, значение семафора должно стать нулевым; второй процесс блокируется при попытке уменьшить значение семафора до отрицательной величины, для того, чтобы этот процесс мог войти в критическую секцию, значение семафора должно быть не менее 3. Обратите внимание, что в данном примере, помимо взаимного исключения процессов, достигается строгая последовательность действий двух процессов: они получают доступ к критической секции строго по очереди.

7 Взаимодействие процессов в сети.

7.1 Механизм сокетов.

Средства межпроцессного взаимодействия ОС UNIX, представленные в системе IPC, решают проблему взаимодействия двух процессов, выполняющихся в рамках одной операционной системы. Однако, очевидно, их невозможно использовать, когда требуется организовать взаимодействие процессов в рамках сети. Это связано как с принятой системой именования, которая обеспечивает уникальность только в рамках данной системы, так и вообще с реализацией механизмов разделяемой памяти, очереди сообщений и семафоров, – очевидно, что для удаленного взаимодействия они не годятся. Следовательно, возникает необходимость в каком-то дополнительном механизме, позволяющем общаться двум процессам в рамках сети. Однако если разработчики программ будут иметь два абсолютно разных подхода к реализации взаимодействия процессов, в зависимости от того, на одной машине они выполняются или на разных узлах сети, им, очевидно, придется во многих случаях создавать два принципиально разных куска кода, отвечающих за это взаимодействие. Понятно, что это неудобно и хотелось бы в связи с этим иметь некоторый унифицированный механизм, который в определенной степени позволял бы абстрагироваться от расположения процессов и давал бы возможность использования одних и тех же подходов для локального и нелокального взаимодействия. Кроме того, как только мы обращаемся к сетевому взаимодействию, встает проблема многообразия сетевых протоколов и их использования. Понятно, что было бы удобно иметь какой-нибудь общий интерфейс, позволяющий пользоваться услугами различных протоколов по выбору пользователя.

Обозначенные проблемы был призван решить механизм, впервые появившийся в UNIX – BSD (4.2) и названный сокетами (**sockets**).

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

должны 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. Ниже мы подробно рассмотрим функции, выполняющие все необходимые действия с сокетами, и напишем пример небольшой серверной и клиентской программы, использующих сокеты.

7.1.1 **Типы сокетов. Коммуникационный домен**.

Сокеты подразделяются на несколько типов в зависимости от типа коммуникационного соединения, который они используют. Два основных типа коммуникационных соединений и, соответственно, 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. Примером соединения с установлением виртуального канала является механизм каналов в UNIX, аналогом такого соединения из реальной жизни также является телефонный разговор. Заметим, что 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.

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. Вообще говоря, 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.

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. Мы будем рассматривать 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.

Заметим, что фактически коммуникационный домен определяет также используемые семейства протоколов. Так, для домена **AF_UNIX** это будут внутренние протоколы ОС, для домена **AF_INET** – протоколы семейства TCP/IP. Современные системы поддерживают и другие коммуникационные домены, например BSD UNIX поддерживает также третий домен – **AF_NS**, использующий протоколы удаленного взаимодействия Xerox NS.

Ниже приведен набор функций для работы с сокетами.

7.1.2 Создание и конфигурирование сокета.

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 нами доменов

соответствующие константы будут равны, как мы уже говорили, **AF_UNIX** и **AF_INET**. Второй аргумент – **type** – определяет тип соединения, которым будет пользоваться сокет (и, соответственно, тип сокета). Для двух основных рассматриваемых нами типов сокетов это будут константы **SOCK_STREAM** для соединения с установлением виртуального канала и **SOCK_DGRAM** для датаграмм[12]. 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 (подробно эти операции будут рассмотрены ниже).

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

Связывание.

Для того чтобы к созданному сокету мог обратиться какой-либо процесс извне, необходимо присвоить ему адрес. Как мы уже говорили, формат адреса зависит от коммуникационного домена, в рамках которого действует сокет, и может представлять собой либо

[12] Заметим, что данный аргумент может принимать не только указанные два значения, например, тип сокета **SOCK_SEQPACKET** обозначает соединение с установлением виртуального канала со всеми вытекающими отсюда свойствами, но при этом сохраняются границы сообщений; однако данный тип сокетов не поддерживается ни в домене **AF_UNIX**, ни в домене **AF_INET**, поэтому мы его здесь рассматривать не будем

путь к файлу, либо сочетание IP-адреса и номера порта. Но в любом случае присвоение 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`**.

Важно отметить, что если мы имеем дело с доменом **`AF_UNIX`** и адрес сокета представляет собой имя файла, то при выполнении функции **`bind()`** система в качестве побочного эффекта создает файл с таким именем. Поэтому для успешного выполнения **`bind()`** необходимо, чтобы такого файла не существовало к данному моменту. Это следует учитывать, если мы «зашиваем» в программу определенное имя и намерены запускать нашу программу несколько раз на одной и той же машине – в этом случае для успешной работы **`bind()`** необходимо удалять файл с этим именем перед связыванием. Кроме того, в процессе создания файла, естественно, проверяются права доступа пользователя, от имени которого производится вызов,

ко всем директориям, фигурирующим в полном путевом имени файла, что тоже необходимо учитывать при задании имени. Если права доступа к одной из директорий недостаточны, вызов **bind()** завершится неуспешно.

В случае успешного связывания **bind()** возвращает 0, в случае ошибки – -1**.**

7.1.3 Предварительное установление соединения.

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 последнее не является требованием. 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 само должно позаботиться о связывании сокета.

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. 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 (напомним, что 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. Обычно это реализуется путем порождения для каждого установленного соединения отдельного процесса-потомка, который занимается собственно обменом данными только с этим конкретным клиентом, в то время как процесс-родитель продолжает прослушивать первоначальный сокет и порождать новые соединения (см. Пример 28).

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 нас не интересует, в качестве второго аргумента можно передать **NULL**.

7.1.4 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[13].

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>
```

[13] Отметим, что, как уже говорилось, при использовании сокетов с установлением виртуального соединения границы сообщений не сохраняются, поэтому приложение, принимающее сообщения, может принимать данные совсем не теми же порциями, какими они были посланы. Вся работа по интерпретации сообщений возлагается на приложение.

```
#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);
```

Первые 4 аргумента у них такие же, как и у рассмотренных выше. В последних двух 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. Отметим, что перед вызовом **recvfrom()** параметр **fromlen** должен быть установлен равным первоначальному размеру структуры **from.** Здесь, как и в функции **accept**, если нас не интересует адрес отправителя, в качестве **from** можно передать **NULL.**

7.1.5 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, разумеется, даже если до этого не был вызван **shutdown()**, соединение будет закрыто. Таким образом, в принципе, если по окончании работы с сокетом мы собираемся 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.

7.1.6 Резюме: общая схема работы с сокетами.

Мы рассмотрели все основные функции работы с сокетами. Обобщая изложенное, можно изобразить общую схему работы с сокетами с установлением соединения в следующем виде:

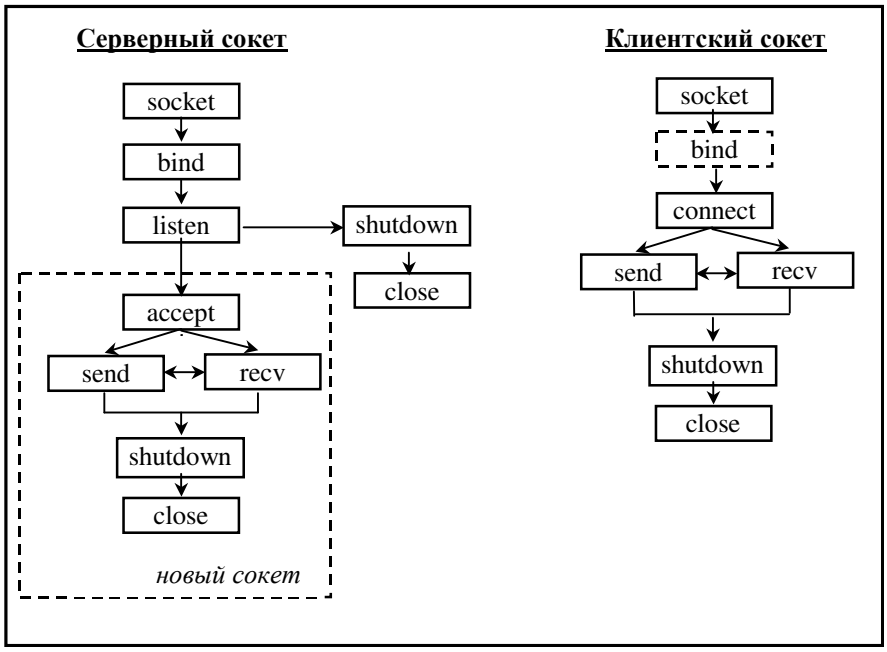

Рис. 20 Схема работы с сокетами с установлением соединения

Общая схема работы с сокетами без предварительного установления соединения проще, она такова:

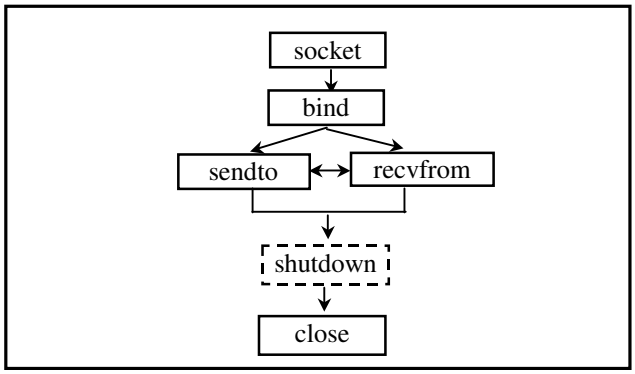

Рис. 21 Схема работы с сокетами без установления соединения

Пример 28. Работа с локальными сокетами.

Рассмотрим небольшой пример, иллюстрирующий работу с сокетами в рамках локального домена (**AF_UNIX**). Ниже приведена небольшая программа, которая 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));
own_addr.sun_family = AF_UNIX;
strcpy(own_addr.sun_path, is_server ? SADDRESS
: CADDRESS);

/* создаем сокет */
if ((sockfd = socket(AF_UNIX, SOCK_DGRAM, 0)) <
0)
{
  printf("can't create socket\n");
  return 0;

}

/* связываем сокет */
unlink(own_addr.sun_path);
if (bind(sockfd, (struct sockaddr *) &own_addr,
sizeof(own_addr.sun_family)+
strlen(own_addr.sun_path)) < 0)
{
      printf("can't bind socket!");
      return 0;
}

if (!is_server)
{
     /* это - клиент */
     memset(&party_addr,                    0,
     sizeof(party_addr));
     party_addr.sun_family = AF_UNIX;
     strcpy(party_addr.sun_path, SADDRESS);
     printf("type the string: ");

     while (gets(buf)) {
          /* не пора ли выходить? */
          quitting = (!strcmp(buf, "quit"));
```

```
            /* считали строку и передаем ее
            серверу */
            if (sendto(sockfd, buf, strlen(buf) +
            1, 0, (struct sockaddr *)
            &party_addr,
            sizeof(party_addr.sun_family) +
            strlen(SADDRESS)) != strlen(buf) + 1)
            {
                 printf("client:  error  writing
                 socket!\n");
                 return 0;
            }

            /*получаем ответ и выводим его на
            печать*/
            if (recvfrom(sockfd, buf, BUFLEN, 0,
            NULL, 0) < 0)
            {
                 printf("client:  error  reading
                 socket!\n");
                 return 0;
            }
            printf("client:   server   answered:
            %s\n", buf);
            if (quitting) break;
            printf("type the string: ");
       } // while
       close(sockfd);
       return 0;
} // if (!is_server)

/* это - сервер */
while (1)
{
       /* получаем строку от клиента и выводим на
       печать */
       party_len = sizeof(party_addr);
```

```
        if    (recvfrom(sockfd,    buf,    BUFLEN,
        0,(struct    sockaddr    *)    &party_addr,
        &party_len) < 0)
        {
             printf("server:     error     reading
             socket!");
             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
             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)
        {
             printf("server:     error     writing
             socket!\n");
             return 0;
        }
        if (quitting) break;
    } // while
    close(sockfd);
    return 0;
}
```

Пример 29. Пример работы с сокетами в рамках сети.

В качестве примера работы с сокетами в домене **AF_INET** напишем простенький web-сервер, который будет понимать только одну команду :

```
GET /<имя файла>
```

Сервер запрашивает у системы сокет, связывает его с адресом, считающимся известным, и начинает принимать клиентские запросы. Для обработки каждого запроса порождается отдельный потомок, в то время как родительский процесс продолжает прослушивать сокет. Потомок разбирает текст запроса и отсылает клиенту либо содержимое требуемого файла, либо диагностику (“плохой запрос” или “файл не найден”).

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <netinet/in.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>

#define PORTNUM 8080
#define BACKLOG 5
#define BUFLEN 80

#define FNFSTR "404 Error File Not Found "
#define BRSTR "Bad Request "

int main(int argc, char **argv)
{
     struct sockaddr_in own_addr, party_addr;
     int sockfd, newsockfd, filefd;
     int party_len;
     char buf[BUFLEN];
     int len;
     int i;
```

```
/* создаем сокет */
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0))
< 0)
{
     printf("can't create socket\n");
     return 0;
}
/* связываем сокет */
memset(&own_addr, 0, sizeof(own_addr));
own_addr.sin_family = AF_INET;
own_addr.sin_addr.s_addr = INADDR_ANY;
own_addr.sin_port = htons(PORTNUM);
if (bind(sockfd, (struct sockaddr *) &own_addr,
sizeof(own_addr)) < 0)
{
     printf("can't bind socket!");
     return 0;
}

/* начинаем обработку запросов на соединение */
if (listen(sockfd, BACKLOG) < 0)
{
     printf("can't listen socket!");
     return 0;
}

while (1) {
     memset(&party_addr, 0,
     sizeof(party_addr));
     party_len = sizeof(party_addr);
     /* создаем соединение */
     if ((newsockfd = accept(sockfd, (struct
     sockaddr *)&party_addr, &party_len)) < 0)
     {
          printf("error accepting
          connection!");
          return 0;
```

```
}

if (!fork())
{
     /*это - сын, он обрабатывает запрос и
     посылает ответ*/
     close(sockfd);   /*  этот сокет сыну
     не нужен */
     if ((len = recv(newsockfd, &buf,
     BUFLEN, 0)) < 0)
     {
          printf("error reading socket!");
          return 0;
     }
     /* разбираем текст запроса */
     printf("received: %s \n", buf);
     if (strncmp(buf, "GET /", 5))
     { /*плохой запрос!*/
          if (send(newsockfd, BRSTR,
          strlen(BRSTR) + 1, 0) !=
          strlen(BRSTR) + 1)
          {
               printf("error writing
               socket!");
               return 0;
          }

          shutdown(newsockfd, 1);
          close(newsockfd);
          return 0;
     }

     for (i=5; buf[i] && (buf[i] > ' ');
     i++);
     buf[i] = 0;
     /* открываем файл */
     if ((filefd = open(buf+5, O_RDONLY))
     < 0)
```

```
                    {
                         /* нет файла! */
                         if (send(newsockfd, FNFSTR,
                         strlen(FNFSTR) + 1, 0) !=
                         strlen(FNFSTR) + 1)
                         {
                              printf("error writing
                              socket!");
                              return 0;
                         }
                         shutdown(newsockfd, 1);
                         close(newsockfd);
                         return 0;
                    }

                    /* читаем из файла порции данных и
                    посылаем их клиенту */
                    while (len = read(filefd, &buf,
                    BUFLEN))
                    if (send(newsockfd, buf, len, 0) < 0)
                    {
                         printf("error writing socket!");
                         return 0;
                    }
                    close(filefd);
                    shutdown(newsockfd, 1);
                    close(newsockfd);
                    return 0;
               }

          /* процесс – отец. Он закрывает новый сокет и
          продолжает прослушивать старый */
          close(newsockfd);
     }
}
```

7.2 Среда параллельного программирования MPI

Еще одним программным механизмом, позволяющим организовать взаимодействие параллельных процессов вне зависимости от их взаимного расположения (т.е. исполняются ли они на одной машине (процессоре) или на разных), является библиотека MPI.

Как следует из ее названия[14], библиотека MPI представляет собой набор функций, позволяющих программисту организовать обмен данными между процессами в терминах передачи сообщений. При этом ответственность за то, чтобы этот обмен был реализован оптимальным образом, в зависимости от конкретной вычислительной системы, на которой исполняется приложение, и от взаимного расположения обменивающихся процессов, ложится на саму библиотеку. Например, очевидно, что для процессов, исполняющихся на одном процессоре, наиболее эффективной будет реализация с использованием разделяемой памяти, в то время как в случае исполнения процессов на разных процессорах могут использоваться сокеты или другие механизмы (подробнее об особенностях различных архитектур мы поговорим чуть ниже).

Таким образом, MPI представляет собой более высокоуровневое средство программирования, нежели рассмотренные выше механизмы, такие как сокеты и разделяемая память, и в то же время в реализации библиотеки MPI могут использоваться все эти средства.

Кроме того, важно отметить, что в отличие от всех рассмотренных выше средств, MPI предоставляет возможность создавать эффективные программы, не только для работы в условиях псевдопараллелизма, когда параллельные процессы в реальности исполняются на единственном процессоре, но и для выполнения на многопроцессорных системах с реальным параллелизмом.

7.2.1 Краткий обзор параллельных архитектур.

Прежде чем вдаваться в детали программирования для параллельных архитектур, рассмотрим на примерах некоторые особенности, общие для всего многообразия этих систем, и дадим некоторую их классификацию.

---

[14] MPI – сокращение от Message Passing Interface, т.е. «Механизм (или интерфейс) передачи сообщений»

Очевидно, что главное отличие любой параллельной вычислительной системы от классической фон-неймановской архитектуры заключается в наличии нескольких процессоров, на которых одновременно могут выполняться независимые или взаимодействующие между собой задачи. При этом параллельно работающие процессы конкурируют между собой за остальные ресурсы системы, а также могут осуществлять взаимодействие аналогично тому, как это происходит в однопроцессорной системе. Рассматривая подробнее особенности различных многопроцессорных систем, мы обратим особое внимание на те механизмы, которые позволяют организовать на них взаимодействие параллельных процессов.

Основным параметром, позволяющим разбить все множество многопроцессорных архитектур на некоторые классы, является доступ к памяти. Существуют системы **с общей памятью**, обладающие определенным массивом блоков памяти, которые одинаково доступны всем процессорам, и системы **с распределенной памятью**, в которых у каждого процессора имеется в наличии только локальная память, доступ в которую других процессоров напрямую невозможен. Существует и промежуточный вариант между этими двумя классами – это так называемые системы **с неоднородным доступом к памяти**, или NUMA-системы. Далее мы подробнее остановимся на каждом из этих классов архитектур.

Системы с распределенной памятью – MPP.

Примером системы с распределенной памятью может служить **массивно-параллельная архитектура – MPP**[15]. Массивно-параллельные системы состоят из однородных вычислительных узлов, каждый из которых включает в себя:

- один или несколько процессоров
- локальную память, прямой доступ к которой с других узлов невозможен
- коммуникационный процессор или сетевой адаптер
- устройства ввода/вывода

Схема MPP системы, где каждый вычислительный узел (ВУ) имеет один процессорный элемент (например, RISC-процессор,

[15] Аббревиатура MPP представляет собой сокращение от «Massive Parallel Processing»

одинаковый для всех ВУ), память и коммуникационное устройство, изображена на рисунке.

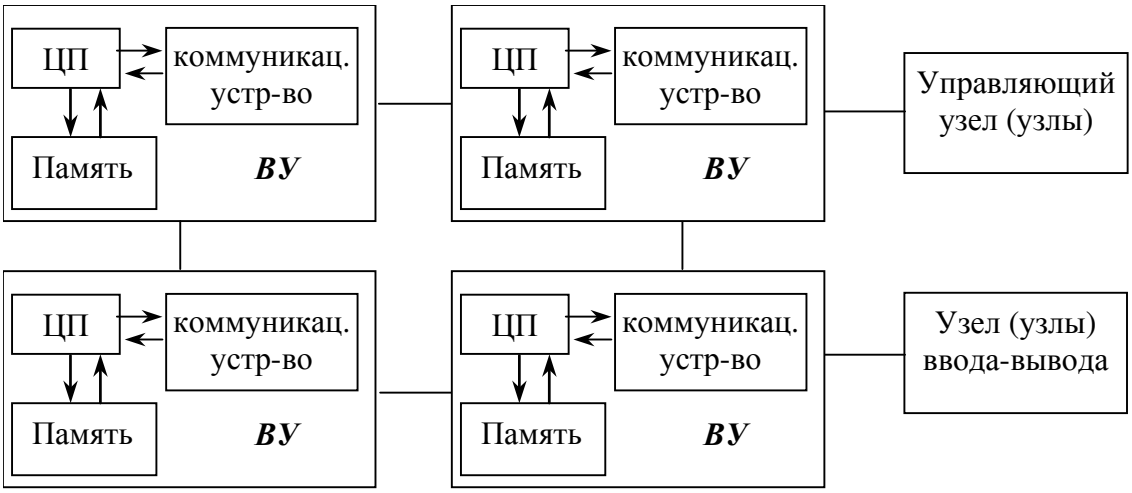

Рис. 22 Архитектура MPP.

Помимо вычислительных узлов, в систему могут входить специальные узлы ввода-вывода и управляющие узлы. Узлы связаны между собой посредством высокоскоростной среды передачи данных определенной топологии. Число процессоров в MPP-системах может достигать нескольких тысяч.

Поскольку в MPP-системе каждый узел представляет собой относительно самостоятельную единицу, то, как правило, управление массивно-параллельной системой в целом осуществляется одним из двух способов:

1. На каждом узле может работать полноценная операционная система, функционирующая отдельно от других узлов. При этом, разумеется, такая ОС должна поддерживать возможность коммуникации с другими узлами в соответствии с особенностями данной архитектуры.
2. «Полноценная» ОС работает только на управляющей машине, а на каждом из узлов MPP-системы работает некоторый сильно «урезанный» вариант ОС, обеспечивающий работу задач на данном узле.

В массивно-параллельной архитектуре отсутствует возможность осуществлять обмен данными между ВУ напрямую через память, поэтому взаимодействие между процессорами реализуется с помощью аппаратно и программно поддерживаемого механизма передачи сообщений между ВУ. Соответственно, и программы для MPP-систем обычно создаются в рамках модели передачи сообщений.

Системы с общей памятью – SMP.

В качестве наиболее распространенного примера систем с общей памятью рассмотрим архитектуру **SMP**[16] – **симметричную мультипроцессорную систему**. SMP-системы состоят из нескольких **однородных** процессоров и массива общей памяти, который обычно состоит из нескольких независимых блоков. Слово «симметричный» в названии данной архитектуры указывает на то, что все процессоры имеют доступ напрямую (т.е. возможность адресации) к любой точке памяти, причем доступ любого процессора ко всем ячейкам памяти осуществляется с одинаковой скоростью. Общая схема SMP-архитектуры изображена на Рис. 23.

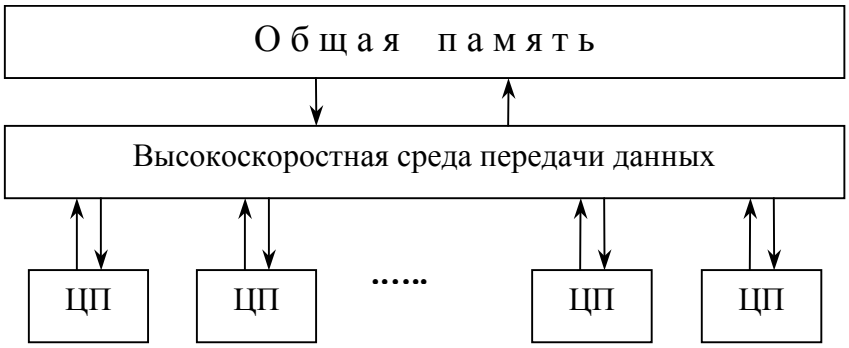

Рис. 23 Архитектура SMP

Процессоры подключены к памяти либо с помощью общей шины, либо с помощью коммутатора. Отметим, что в любой системе с общей памятью возникает проблема кэширования: так как к некоторой ячейке общей памяти имеет возможность обратиться каждый из процессоров, то вполне возможна ситуация, когда некоторое значение из этой ячейки памяти находится в кэше одного или нескольких процессоров, в то время как другой процессор изменяет значение по данному адресу. В этом случае, очевидно, значения, находящиеся в кэшах других процессоров, больше не могут быть использованы и должны быть обновлены. В SMP-архитектурах обычно согласованность данных в кэшах поддерживается аппаратно.

Очевидно, что наличие общей памяти в SMP-архитектурах позволяет эффективно организовать обмен данными между задачами, выполняющимися на разных процессорах, с использованием механизма разделяемой памяти. Однако сложность организации симметричного доступа к памяти и поддержания согласованности кэшей накладывает существенное ограничение на количество процессоров в таких системах – в реальности их число

[16] Аббревиатура SMP является сокращением фразы «Symmetric Multi Processing»

обычно не превышает 32 – в то время, как стоимость таких машин весьма велика. Некоторым компромиссом между масштабируемостью и однородностью доступа к памяти являются NUMA-архитектуры, которые мы рассмотрим далее.

Системы с неоднородным доступом к памяти – NUMA.

Системы с неоднородным доступом к памяти (NUMA[17]) представляют собой промежуточный класс между системами с общей и распределенной памятью. Память в NUMA-системах является физически распределенной, но логически общедоступной. Это означает, что каждый процессор может адресовать как свою локальную память, так и память, находящуюся на других узлах, однако время доступа к удаленным ячейкам памяти будет в несколько раз больше, нежели время доступа к локальной памяти. Заметим, что единой адресное пространство и доступ к удаленной памяти поддерживаются аппаратно. Обычно аппаратно поддерживается и когерентность (согласованность) кэшей во всей системе

Системы с неоднородным доступом к памяти строятся из однородных базовых модулей, каждый из которых содержит небольшое число процессоров и блок памяти. Модули объединены между собой с помощью высокоскоростного коммутатора. Обычно вся система работает под управлением единой ОС. Поскольку логически программисту предоставляется абстракция общей памяти, то модель программирования, используемая в системах NUMA, обычно в известной степени аналогична той, что используется на симметричных мультипроцессорных системах, и организация межпроцессного взаимодействия опирается на использование разделяемой памяти.

Масштабируемость NUMA-систем ограничивается объемом адресного пространства, возможностями аппаратуры поддержки когерентности кэшей и возможностями операционной системы по управлению большим числом процессоров.

Кластерные системы.

Отдельным подклассом систем с распределенной памятью являются **кластерные системы,** которые представляют собой некоторый аналог массивно-параллельных систем, в котором в

[17] Аббревиатура NUMA расшифровывается как «Non-Unified Memory Access», что в буквальном переводе и означает «неоднородный доступ к памяти». Часто используется также обозначение «ccNUMA», что означает «cache-coherent NUMA», или система с неоднородным доступом к памяти с поддержкой когерентности (т.е. согласованности) кэшей

качестве ВУ выступают обычные рабочие станции общего назначения, причем иногда узлы кластера могут даже одновременно использоваться в качестве пользовательских рабочих станций. Кластер, объединяющий компьютеры разной мощности или разной архитектуры, называют **гетерогенным** (неоднородным). Для связи узлов используется одна из стандартных сетевых технологий, например, Fast Ethernet.

Главными преимуществами кластерных систем, благодаря которым они приобретают все большую популярность, являются их относительная дешевизна, возможность масштабирования и возможность использования при построении кластера тех вычислительных мощностей, которые уже имеются в распоряжении той или иной организации.

При программировании для кластерных систем, как и для других систем с распределенной памятью, используется модель передачи сообщений.

7.2.2 Модель программирования MPI.

Как мы видим, при написании программ для параллельных архитектур выбор модели программирования сильно зависит от конкретной архитектуры, на которой предполагается выполнять программу. Например, если целевой архитектурой является система с общей памятью, то для обмена данными между процессами целесообразно использовать механизм разделяемой памяти, если же программа пишется для работы на системе с распределенной памятью, то необходимо организовывать обмен с помощью сообщений. Таким образом, если программист имеет возможность доступа к системе с общей памятью и с распределенной памятью, ему придется создавать отдельную версии своей программы для работы на каждой из этих систем, осваивая при этом различные модели программирования.

В то же время, хотелось бы иметь некоторый единый механизм взаимодействия, который был бы реализован, и притом эффективно, для большинства или хотя бы для многих конкретных параллельных систем. В таком случае для перенесения программы с одной архитектуры на другую было бы достаточно простой перекомпиляции, а программист, освоивший данное средство, получил бы возможность создавать эффективные программы для широкого класса параллельных архитектур. Одним из таких широко распространенных средств параллельного программирования является MPI.

MPI представляет собой стандарт, описывающий некоторое множество функций для обмена сообщениями между параллельными процессами. Существует множество реализаций MPI для различных параллельных архитектур, как с распределенной, так и с общей памятью. Как правило, эти реализации оформлены в виде набора библиотечных функций, которые можно использовать при программировании на языках Фортран и Си.

В модели программирования MPI приложение представляет собой совокупность процессов или нитей (иначе называемых **ветвями**), общающихся друг с другом путем передачи сообщений. При этом для организации обмена не является важным, будут ли процессы исполняться на одном процессоре или вычислительном узле или на разных – механизм обмена данными в обоих случаях одинаков. Во всем, что не касается передачи и приема сообщений, ветви являются независимыми и изолированными друг от друга. Отметим, что ветви приложения могут обмениваться сообщениями в среде MPI только между собой, но не с процессами других приложений, исполняющимися в той же вычислительной системе. Помимо функций для обмена сообщениями, MPI предоставляет возможности для взаимной синхронизации процессов и для решения ряда других вспомогательных задач.

Количество ветвей в данном приложении задается в момент его запуска, т.е. не существует возможности порождать ветви динамически во время исполнения приложения[18]. Запуск MPI-приложения осуществляется с помощью специальной программы (чаще всего она называется **`mpirun`**), которой обычно указывается количество ветвей, которые необходимо породить, имя исполняемого файла, а также входные параметры приложения.

При написании программы с использованием MPI ее исходный текст должен содержать код для всех ветвей сразу, однако во время исполнения у каждой ветви имеется возможность определить свой собственный порядковый номер и общее количество ветвей и в зависимости от этого исполнять ту или иную часть алгоритма (данный подход в чем-то аналогичен использованию системного вызова **`fork()`**)

[18] Заметим, что в версии стандарта MPI-2 описывается возможность динамического порождения ветвей, однако на момент написания данного пособия нам неизвестно ни об одной реализации MPI, в которой поддерживалось бы динамическое порождение ветвей, поэтому далее мы будем рассматривать только модель со статическим порождением ветвей в момент запуска приложения.

7.2.3 Функции общего назначения. Общая структура программы.

Все без исключения функции Си-библиотеки MPI возвращают целое значение, описывающее код завершения операции. Существует специальная константа **`MPI_SUCCESS`**, соответствующая успешному завершению функции. В случае неудачного завершения возвращаемое значение будет отлично от **`MPI_SUCCESS`**, и равно одному из кодов ошибок, определенных в MPI и описывающих различные исключительные ситуации.

Все типы данных, константы и функции, относящиеся к Си-библиотеке MPI, описаны в заголовочном файле **`<mpi.h>`**.

Коммуникаторы и группы.

Важными понятиями MPI являются группы ветвей и коммуникаторы. **Группа ветвей** представляет собой упорядоченное множество ветвей. Каждой ветви в группе назначается уникальный порядковый номер – целое число в диапазоне [0, N-1], где N – количество ветвей в группе.

При запуске приложения все его порожденные ветви образуют начальную группу. Библиотека MPI предоставляет функции, позволяющие создавать новые группы на основе существующих путем их разбиения, объединения, исключения ветвей из групп и т.п. операций.

С каждой группой ветвей MPI связан так называемый «коммуникационный контекст», или «коммуникационное поле», задающее множество всех возможных участников операций обмена данными и уникальным образом описывающее каждого участника. Кроме того, коммуникационный контекст может использоваться для хранения некоторых общих для всех ветвей данной группы данных. Для описания коммуникационного контекста в MPI служат объекты специального типа – **коммуникаторы**. Коммуникатор, или описатель коммуникационного контекста, присутствует в качестве параметра во всех вызовах MPI, связанных с обменом данными. Подчеркнем, что любая операция обмена данными может быть осуществлена только внутри определенного коммуникационного контекста, поэтому, например, сообщение, отправленное с помощью какого-либо коммуникатора, может быть получено только с помощью этого же самого коммуникатора.

Коммуникаторы MPI описываются специальным типом данных **`MPI_Comm`**. При запуске приложения сразу после инициализации среды выполнения MPI каждой ветви становятся

доступны два предопределенных коммуникатора, обозначаемых константами **MPI_COMM_WORLD** и **MPI_COMM_SELF**. **MPI_COMM_WORLD** представляет собой коммуникатор, описывающий коммуникационный контекст начальной группы ветвей, т.е. всех ветвей, порожденных при запуске приложения. **MPI_COMM_SELF** – это коммуникатор, включающий одну-единственную ветвь – текущую, и соответственно, для каждой ветви он будет иметь свое значение.

Каждая ветвь приложения имеет возможность узнать общее количество ветвей в своей группе и свой собственный уникальный номер в ней. Для этого служат функции **MPI_Comm_size()** и **MPI_Comm_rank()**:

```
#include <mpi.h>
int MPI_Comm_size(MPI_Comm comm, int *size);
int MPI_Comm_rank(MPI_Comm comm, int *rank);
```

В первом параметре каждой из этих функций передается коммуникатор, описывающий коммуникационный контекст интересующей группы ветвей, во втором – указатель на целочисленную переменную, в которую будет записан результат: для функции **MPI_Comm_size()** – количество ветвей в группе; для функции **MPI_Comm_rank()** – уникальный номер текущей ветви в группе.

Отметим, что в MPI предусмотрена также возможность обмена сообщениями между ветвями, принадлежащими разным группам. Для этого существует возможность создать так называемый интер-коммуникатор, объединяющий коммуникационные контексты, заданные двумя различными коммуникаторами, каждый из которых описывает коммуникационный контекст некоторой группы. После создания такого коммуникатора его можно использовать для организации обмена данными между ветвями этих групп.

Обрамляющие функции. Инициализация и завершение.

Самым первым вызовом MPI в любой программе, использующей эту библиотеку, должен быть вызов функции **MPI_Init()** – инициализация среды выполнения MPI.

```
#include <mpi.h>
int MPI_Init(int *argc, char ***argv);
```

Этой функции передаются в качестве параметров указатели на параметры, полученные функцией **main()**, описывающие аргументы командной строки. Смысл этого заключается в том, что при загрузке MPI-приложения **mpirun** может добавлять к его аргументам

командной строки некоторые дополнительные параметры, необходимые для инициализации среды выполнения MPI. В этом случае вызов **MPI_Init()** также изменяет массив аргументов командной строки и их количество, удаляя эти дополнительные аргументы.

Еще раз отметим, что функция **MPI_Init()** должна быть вызвана до любой другой функции MPI, кроме того, в любой программе она должна вызываться не более одного раза.

По завершении работы с MPI в каждой ветви необходимо вызвать функцию закрытия библиотеки – **MPI_Finalize()**:

```
#include <mpi.h>
int MPI_Finalise(void);
```

После вызова этой функции невозможен вызов ни одной функции библиотеки MPI (в том числе и повторный вызов **MPI_Init()**)

Для аварийного завершения работы с библиотекой MPI служит функция **MPI_Abort()**.

```
#include <mpi.h>
int MPI_Abort(MPI_Comm comm, int errorcode);
```

Эта функция немедленно завершает все ветви приложения, входящие в коммуникационный контекст, который описывает коммуникатор **comm**. Код завершения приложения передается в параметре **errorcode**. Отметим, что **MPI_Abort()** вызывает завершения всех ветвей данного коммуникатора даже в том случае, если она была вызвана только в какой-либо одной ветви.

Синхронизация: барьеры.

Для непосредственной синхронизации ветвей в MPI предусмотрена одна специальная функция – **MPI_Barrier()**.

```
#include <mpi.h>
int MPI_Barrier(MPI_Comm comm);
```

С помощью этой функции все ветви, входящие в определенный коммуникационный контекст, могут осуществить так называемую **барьерную синхронизацию**. Суть ее в следующем: в программе выделяется некоторая точка синхронизации, в которой каждая из ветвей вызывает **MPI_Barrier()**. Эта функция блокирует вызвавшую ее ветвь до тех пор, пока все остальные ветви из данного коммуникационного контекста также не вызовут **MPI_Barrier()**. После этого все ветви одновременно разблокируются. Таким

образом, возврат управления из **MPI_Barrier()** осуществляется во всех ветвях одновременно.

Барьерная синхронизация используется обычно в тех случаях, когда необходимо гарантировать завершение работы над некоторой подзадачей во всех ветвях перед переходом к выполнению следующей подзадачи.

Важно понимать, что функция **MPI_Barrier()**, как и другие коллективные функции, которые мы подробно рассмотрим ниже, должна быть обязательно вызвана во всех ветвях, входящих в данный коммуникационный контекст. Из семантики функции **MPI_Barrier()** следует, что если хотя бы в одной ветви из заданного коммуникационного контекста **MPI_Barrier()** вызвана не будет, это приведет к «зависанию» всех ветвей, вызвавших **MPI_Barrier()**.

Пример 30. Использование барьерной синхронизации.

В данном примере иллюстрируется общая схема построения программы с использованием библиотеки MPI, а также использование функций общего назначения и барьерной синхронизации.

На экран выводится общее количество ветвей в приложении, затем каждая ветвь выводит свой уникальный номер. После этого ветвь с номером 0 печатает на экран аргументы командной строки. Барьерная синхронизация необходима для того, чтобы сообщения о порядковых номерах ветвей не смешивались с сообщением об общем количестве ветвей и с выводом аргументов командной строки.

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char **argv)
{
     int size, rank, i;

     MPI_Init(&argc, &argv); /* Инициализируем
     библиотеку */
     MPI_Comm_size(MPI_COMM_WORLD, &size);
     /* Узнаем количество задач в запущенном
     приложении... */
```

```
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    /* ...и свой собственный номер: от 0 до (size-
    1) */


    /* задача с номером 0 сообщает пользователю
    размер группы,коммуникационный контекст которой
    описывает коммуникатор MPI_COMM_WORLD, т.е.
    число ветвей в приложении */
    if (rank == 0) {
         printf("Total processes count = %d\n",
         size );
    }
    /* Осуществляется барьерная синхронизация  */
    MPI_Barrier(MPI_COMM_WORLD);


    /* Теперь каждая задача выводит на экран свой
    номер */
    printf("Hello! My rank in MPI_COMM_WORLD =
    %d\n", rank);


    /* Осуществляется барьерная синхронизация  */
    MPI_Barrier(MPI_COMM_WORLD);
    /* затем ветвь с номером 0 печатает аргументы
    командной строки. */
    if (rank == 0) {
         printf("Command line of process 0:\n");
         for(i = 0; i < argc; i++) {
              printf("%d: \"%s\"\n", i, argv[i]);
         }
    }
    /* Все задачи завершают выполнение */
    MPI_Finalize();
    return 0;
}
```

7.2.4 Прием и передача данных. Общие замечания.

Прежде чем перейти к рассмотрению функций, непосредственно относящихся к организации обмена сообщениями между ветвями, остановимся на некоторых важных общих

моментах, таких как атрибуты сообщения и поддержка типизации данных в MPI.

Сообщения и их атрибуты.

Каждое сообщение MPI обладает следующим набором атрибутов:

- Номер ветви-отправителя
- Номер или номера ветвей-получателей
- Коммуникатор, т.е. описатель коммуникационного контекста, в котором данное сообщение передается
- Тип данных, образующих тело сообщения
- Количество элементов указанного типа, составляющих тело сообщения
- Тэг, или бирка сообщения

В MPI не существует специальной структуры данных, описывающей сообщение. Вместо этого, атрибуты сообщения передаются в соответствующие функции и получаются из них как отдельные параметры.

Тэг сообщения представляет собой целое число, аналог типа сообщения в механизме очередей сообщений IPC. Интерпретация этого атрибута целиком возлагается на саму программу.

Поддержка типов данных в MPI.

В отличие от других средств межпроцессного взаимодействия, рассмотренных нами ранее, в MPI требуется явное указание типа данных, образующих тело сообщения. Причина этого заключается в том, что библиотека MPI берет на себя заботу о корректности передачи данных в случае использования ее в гетерогенной (т.е. состоящей из разнородных узлов) среде. Одни и те же типы данных на машинах разных архитектур могут иметь различное представление (к примеру, в архитектуре Intel представление целых чисел характеризуется расположением байт от младшего к старшему, в то время как в процессорах Motorola принято обратное расположение; кроме того, на разных машинах один и тот же тип может иметь разную разрядность и выравнивание). Очевидно, что в такой ситуации передача данных как последовательности байт без соответствующих преобразований приведет к их неверной интерпретации. Реализация MPI производит необходимые преобразования данных в теле сообщения перед его посылкой и

после получения, однако для этого необходима информация о типе данных, образующих тело сообщения.

Для описания типов данных в MPI введен специальный тип – **`MPI_Datatype`**. Для каждого стандартного типа данных Си в заголовочном файле библиотеки описана константа типа **`MPI_Datatype`**; ее имя состоит из префикса MPI_ и имени типа, написанного большими буквами, например, **`MPI_INT`**, **`MPI_DOUBLE`** и т.д. Кроме этого, MPI предоставляет широкие возможности по конструированию описателей производных типов, однако их рассмотрение выходит за рамки данного пособия.

Параметр типа **`MPI_Datatype`**, описывающий тип данных, образующих тело сообщения, присутствует во всех функциях приема и передачи сообщений MPI.

Непосредственно с поддержкой типов данных MPI связана еще одна особенность функций приема-передачи данных: в качестве размера тела сообщения всюду фигурирует не количество байт, а количество элементов указанного типа.

7.2.5 Коммуникации «точка-точка». Блокирующий режим.

Библиотека MPI предоставляет возможности для организации как индивидуального обмена сообщениями между парой ветвей – в этом случае у каждого сообщения имеется единственный получатель – так и коллективных коммуникаций, в которых участвуют все ветви, входящие в определенный коммуникационный контекст.

Рассмотрим сначала функции MPI, используемые для отправки и получения сообщений между двумя ветвями – такой способ обмена в литературе часто носит название **коммуникации «точка-точка»**.

Библиотека MPI предоставляет отдельные пары функций для реализации блокирующих и неблокирующих операций приема и посылки данных. Заметим, что они совместимы друг с другом в любых комбинациях, например, для отправки сообщения может использоваться блокирующая операция, а для приема этого же сообщения – неблокирующая, и наоборот.

Отправка сообщений в блокирующем режиме.

Для отправки сообщений в блокирующем режиме служит функция **`MPI_Send()`**:

```
#include <mpi.h>
```

```
int MPI_Send(void* buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm);
```

Аргументы у этой функции следующие:

buf – указатель на буфер, в котором расположены данные для передачи

count – количество элементов заданного типа в буфере

datatype – тип элементов в буфере

dest – уникальный номер ветви-получателя. Представляет собой неотрицательное целое число в диапазоне [0, N-1], где N – число ветвей в коммуникационном контексте, описываемом коммуникатором **comm**

tag – тэг (бирка) сообщения. Представляет собой неотрицательное целое число от 0 до некоторого максимума, значение которого зависит от реализации (однако стандарт MPI гарантирует, что этот максимум имеет значение не менее 32767). Тэги сообщений могут использоваться ветвями для того, чтобы отличать разные по смыслу сообщения друг от друга.

comm – коммуникатор, описывающий коммуникационный контекст, в котором будет передаваться сообщение

Еще раз отметим, что в этой функции, как и во всех остальных функциях приема-передачи MPI, в параметре **count** указывается не длина буфера в байтах, а количество элементов типа **datatype**, образующих буфер.

При вызове **MPI_Send()** управление возвращается процессу только тогда, когда вызвавший процесс может повторно использовать буфер **buf** (т.е. записывать туда другие данные) без риска испортить еще не переданное сообщение. Однако, возврат из функции **MPI_Send()** не обязательно означает, что сообщение было доставлено адресату – в реализации данной функции может использоваться буферизация, т.е. тело сообщения может быть скопировано в системный буфер, после чего происходит возврат из **MPI_Send()**, а сама передача будет происходить позднее в асинхронном режиме.

С одной стороны, буферизация при отправке позволяет ветви-отправителю сообщения продолжить выполнение, не дожидаясь момента, когда ветвь-адресат сообщения инициирует его прием; с другой стороны – буферизация, очевидно, увеличивает накладные

расходы как по памяти, так и по времени, так как требует добавочного копирования. При использовании вызова **MPI_Send()** решение о том, будет ли применяться буферизация в каждом конкретном случае, принимается самой библиотекой MPI. Это решение принимается из соображений наилучшей производительности, и может зависеть от размера сообщения, а также от наличия свободного места в системном буфере. Если буферизация не будет применяться, то возврат управления из **MPI_Send()** произойдет только тогда, когда ветвь-адресат инициирует прием сообщения, и его тело будет скопировано в буфер-приемник ветви-адресата.

Режимы буферизации.

MPI позволяет программисту самому управлять режимом буферизации при отправке сообщений в блокирующем режиме. Для этого в MPI существуют три дополнительные модификации функции **MPI_Send()**: **MPI_Bsend()**, **MPI_Ssend()** и **MPI_Rsend()**:

```
#include <mpi.h>

int MPI_Bsend(void* buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm);

int MPI_Ssend(void* buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm);

int MPI_Rsend(void* buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm);
```

Их параметры полностью аналогичны **MPI_Send()**, различие заключается только в режимах буферизации.

MPI_Bsend() предполагает отсылку сообщения с буферизацией. Это означает, что если к моменту вызова **MPI_Bsend()** ветвь-адресат не инициировала прием сообщения, и т.о. сообщение не может быть доставлено немедленно, оно должно быть буферизовано. Если в буфере окажется недостаточно свободного места, **MPI_Bsend()** вернет ошибку. MPI предоставляет специальные функции, с помощью которых ветвь может выделить в своем адресном пространстве некоторый буфер, который будет использоваться MPI для буферизации сообщений в этом режиме, таким образом, программист может сам управлять размером доступного буферного пространства. Заметим, что посылка сообщения с помощью функции **MPI_Bsend()** является, в отличие от всех остальных операций отсылки, **локальной** операцией, т.е. ее завершение не зависит от поведения ветви-адресата.

Функция **MPI_Ssend()**, наоборот, представляет собой отсылку сообщения без буферизации. Возврат управления из функции **MPI_Ssend()** осуществляется только тогда, когда адресат инициировал прием сообщения, и оно скопировано в буфер-приемник адресата. Если на принимающей стороне используется блокирующая функция приема сообщения, то такая коммуникация представляет собой реализацию **схемы рандеву**: операция обмена, инициированная на любой из сторон, не завершится до тех пор, пока вторая сторона также не притупит к обмену.

Вызов функции **MPI_Rsend()** сообщает MPI о том, что ветвь-адресат уже инициировала запрос на прием сообщения. В некоторых случаях знание этого факта позволяет MPI использовать более короткий протокол для установления соединения и тем самым повысить производительность передачи данных. Однако, использование этой функции ограничено теми ситуациями, когда ветви-отправителю известно о том, что ветвь-адресат уже ожидает сообщение. В случае, если на самом деле ветвь-адресат не инициировала прием сообщения, **MPI_Rsend()** завершается с ошибкой.

Прием сообщений в блокирующем режиме.

Прием сообщений в блокирующем режиме осуществляется функцией **MPI_Recv()**:

```
#include <mpi.h>
int MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status);
```

Она имеет следующие аргументы:

buf – указатель на буфер, в котором нужно разместить тело принимаемого сообщения

count – максимальное количество элементов заданного типа, которое может уместиться в буфере

datatype – тип элементов в теле получаемого сообщения

dest – уникальный номер ветви-отправителя. С помощью этого параметра процесс получатель может указать конкретного отправителя, от которого он желает получить сообщение, либо сообщить о том, что он хочет получить сообщение от любой ветви данного коммуникатора – в этом случае в качестве значения параметра указывается константа **MPI_ANY_SOURCE**

tag – тэг (бирка) сообщения. Ветвь-получатель может инициировать прием сообщения с конкретным значением тэга, либо указать в этом параметре константу **MPI_ANY_TAG** для получения сообщения с любым значением тэга.

comm – коммуникатор, описывающий коммуникационный контекст, в котором принимается сообщение

status – указатель на структуру типа **MPI_Status**, поля которой будут заполнены функцией **MPI_Recv()**. Используя эту структуру, ветвь-получатель может узнать дополнительную информацию о сообщении, такую как, например, его фактический размер, а также фактические значения тэга и отправителя в том случае, если использовались константы **MPI_ANY_SOURCE** и **MPI_ANY_TAG**.

Если в момент вызова **MPI_Recv()** нет ни одного сообщения, удовлетворяющего заданным критериям (т.е. посланного через заданный коммуникатор и имеющего нужного отправителя и тэг, в случае, если были заданы их конкретные значения), то выполнение ветви блокируется до момента поступления такого сообщения. В любом случае, управление возвращается процессу лишь тогда, когда сообщение будет получено, и его данные будут записаны в буфер **buf** и структуру **status**.

Отметим, что фактический размер принятого сообщения может быть меньше, чем указанная в параметре **count** максимальная емкость буфера. В этом случае сообщение будет успешно получено, его тело записано в буфер, а остаток буфера останется нетронутым. Для того, чтобы узнать фактический размер принятого сообщения, служит функция **MPI_Get_count()**:

```
#include <mpi.h>

int MPI_Get_count(MPI_Status *status, MPI_Datatype
datatype, int *count);
```

Этой функции передаются в качестве параметров указатель на структуру типа **MPI_Status**, которая была заполнена в момент вызова функции приема сообщения, а также тип данных, образующих тело принятого сообщения. Параметр **count** представляет собой указатель на переменную целого типа, в которую будет записано количество элементов типа **datatype**, образующих тело принятого сообщения.

Если же при приеме сообщения его размер окажется больше указанной максимальной емкости буфера, функция **MPI_Recv()**

вернет соответствующую ошибку. Для того чтобы избежать такой ситуации, можно сначала вызвать функцию **MPI_Probe()**, которая позволяет получить информацию о сообщении, не принимая его:

```
#include <mpi.h>

int MPI_Probe(int source, int tag, MPI_Comm comm,
MPI_Status *status);
```

Параметры этой функции аналогичны последним 4-м параметрам функции **MPI_Recv()**. Функция **MPI_Probe()**, как и **MPI_Recv()**, возвращает управление только при появлении сообщения, удовлетворяющего переданным параметрам, и заполняет информацией о сообщении структуру типа **MPI_Status**, указатель на которую передается в последнем параметре. Однако, в отличие от **MPI_Recv()**, вызов **MPI_Probe()** не осуществляет собственно прием сообщения. После возврата из **MPI_Probe()** сообщение остается во входящей очереди и может быть впоследствии получено одним из вызовов приема данных, в частности, **MPI_Recv()**.

Пример 31. MPI: прием сообщения, размер которого неизвестен заранее.

В данном примере приведена схема работы с сообщением, размер которого заранее неизвестен. Ветвь с номером 0 посылает ветви с номером 1 сообщение, содержащее имя исполняемой программы (т.е. значение нулевого элемента в массиве аргументов командной строки). В ветви с номером 1 вызов **MPI_Probe()** позволяет получить доступ к описывающей сообщение структуре типа **MPI_Status**. Затем с помощью вызова **MPI_Get_count()** можно узнать размер буфера, который необходимо выделить для приема сообщения.

```
#include <mpi.h>
#include <stdio.h>


int main(int argc, char **argv)
{
     int size, rank;


     MPI_Init(&argc,    &argv);    /*    Инициализируем
     библиотеку */
     MPI_Comm_size(MPI_COMM_WORLD, &size);
```

```
    /* Узнаем количество задач в запущенном
    приложении... */
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    /* ...и свой собственный номер: от 0 до (size-
    1) */

    if ((size > 1) && (rank == 0)) {
        /* задача с номером 0 отправляет
        сообщение*/
        MPI_Send(argv[0],          strlen(argv[0]),
        MPI_CHAR, 1, 1, MPI_COMM_WORLD);
        printf("Sent to process 1: \"%s\"\n",
        argv[0]);
    } else if ((size > 1) && (rank == 1)) {
        /* задача с номером 1 получает сообщение*/
        int count;
        MPI_Status status;
        char *buf;

        MPI_Probe(0, 1, MPI_COMM_WORLD, &status);
        MPI_Get_count(&status, MPI_CHAR, &count);
        buf    =    (char    *)    malloc(count    *
        sizeof(char));
        MPI_Recv(buf,  count,  MPI_CHAR,  0,  1,
        MPI_COMM_WORLD, &status);
        printf("Received    from    process    0:
        \"%s\"\n", buf);
     }

    /* Все задачи завершают выполнение */
    MPI_Finalize();
    return 0;
}
```

7.2.6 Коммуникации «точка-точка». Неблокирующий режим.

Во многих приложениях для повышения производительности программы бывает выгодно совместить отправку или прием сообщений с основной вычислительной работой, что удается при

использовании неблокирующего режим приема и отправки сообщений. В этом режиме вызов функций приема либо отправки сообщения инициирует начало операции приема/передачи и сразу после этого возвращает управление вызвавшей ветви, а сам процесс передачи данных происходит асинхронно.

Однако, для того, чтобы инициировавшая операцию приема/передачи ветвь имела возможность узнать о том, что операция завершена, необходимо каким-то образом идентифицировать каждую операцию обмена. Для этого в MPI существует механизм так называемых **«квитанций»** (requests), для описания которых служит тип данных **`MPI_Request`**. В момент вызова функции, инициирующей операцию приема/передачи, создается квитанция, соответствующая данной операции, и через параметр-указатель возвращается вызвавшей ветви. Впоследствии, используя эту квитанцию, ветвь может узнать о завершении ранее начатой операции.

Отсылка и прием сообщений в неблокирующем режиме.

Для отсылки и приема сообщений в неблокирующем режиме служат следующие базовые функции:

```
#include <mpi.h>

int MPI_Isend(void* buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm,
MPI_Request *request);

int MPI_Irecv(void* buf, int count, MPI_Datatype
datatype, int source, int tag, MPI_Comm comm,
MPI_Request *request);
```

Все их параметры, кроме последнего, аналогичны параметрам функций **`MPI_Send()`** и **`MPI_Recv()`**. Последним параметром служит указатель на переменную типа **`MPI_Request`**, которая после возврата из данного вызова будет содержать квитанцию для инициированной операции отсылки или приема данных.

Возврат из функции **`MPI_Isend()`** происходит сразу же, как только система инициирует операцию отсылки сообщения, и будет готова начать копировать данные из буфера, переданного в параметре **`buf`**. Важно понимать, что возврат из функции **`MPI_Isend()`** не означает, что этот буфер можно использовать по другому назначению! Вызвавший процесс не должен производить никаких операций над этим буфером до тех пор, пока не убедится в том, что операция отсылки данных завершена.

Возврат из функции **MPI_Irecv()** происходит сразу же, как только система инициирует операцию приема данных и будет готова начать принимать данные в буфер, предоставленный в параметре **buf**, вне зависимости от того, существует ли в этот момент сообщение, удовлетворяющее остальным переданным параметрам (**source**, **tag**, **comm**). При этом вызвавший процесс не может полагать, что в буфере **buf** находятся принятые данные до тех пор, пока не убедится, что операция приема завершена. Отметим также, что у функции **MPI_Irecv()** отсутствует параметр **status**, так как на момент возврата из этой функции информация о принимаемом сообщении, которая должна содержаться в **status**, может быть еще неизвестна.

В неблокирующем режиме, также как и в блокирующем, доступны различные варианты буферизации при отправке сообщения. Для явного задания режима буферизации, отличного от стандартного, служат следующие функции:

```
#include <mpi.h>

int MPI_Issend(void* buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm,
MPI_Request *request);

int MPI_Ibsend(void* buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm,
MPI_Request *request);

int MPI_Irsend(void* buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm,
MPI_Request *request);
```

Отметим, что поскольку **MPI_Ibsend()** предполагает обязательную буферизацию сообщения, а **MPI_rsend()** используется лишь тогда, когда на принимающей стороне уже гарантированно инициирован прием сообщения, их использование лишь в некоторых случаях может дать сколь либо заметный выигрыш по сравнению с аналогичными блокирующими вариантами, так как разницу во времени их выполнения составляет лишь время, необходимое для копирования данных.

Работа с квитанциями.

Библиотека MPI предоставляет целое семейство функций, с помощью которых процесс, инициировавший одну или несколько асинхронных операций приема/передачи сообщений, может узнать о статусе их выполнения. Самыми простыми из этих функций являются **MPI_Wait()** и **MPI_Test()**:

```
#include <mpi.h>
```

```
int MPI_Wait(MPI_Request *request, MPI_Status
*status);

int MPI_Test(MPI_Request *request, int *flag,
MPI_Status *status);
```

Функция **MPI_Wait()** вернет управление лишь тогда, когда операция приема/передачи сообщения, соответствующая квитанции, переданной в параметре **request**, будет завершена (для операции отсылки, инициированной функцией **MPI_Ibsend()** это означает, что данные были скопированы в специально отведенный буфер). При этом для операции приема сообщения в структуре, на которую указывает параметр **status**, возвращается информация о полученном сообщении, аналогично тому, как это делается в **MPI_Recv()**. После возврата из функции **MPI_Wait()** для операции отсылки сообщения процесс может повторно использовать буфер, содержавший тело отосланного сообщения, а для операции приема сообщения гарантируется, что после возврата из **MPI_Wait()** в буфере для приема находится тело принятого сообщения.

Для того, чтобы узнать, завершена ли та или иная операция приема/передачи сообщения, но избежать блокирования в том случае, если операция еще не завершилась, служит функция **MPI_Test()**. В параметре **flag** ей передается адрес целочисленной переменной, в которой будет возвращено ненулевое значение, если операция завершилась, и нулевое в противном случае. В случае, если квитанция соответствовала операции приема сообщения, и данная операция завершилась, **MPI_Test()** заполняет структуру, на которую указывает параметр **status**, информацией о принятом сообщении.

Для того, чтобы узнать статус сразу нескольких асинхронных операций приема/передачи сообщений, служит ряд функций работы с массивом квитанций:

```
#include <mpi.h>

int MPI_Waitany(int count, MPI_Request
*array_of_requests, int *index, MPI_Status *status);

int MPI_Testany(int count, MPI_Request
*array_of_requests, int *index, int *flag,
MPI_Status *status);

int MPI_Waitsome(int count, MPI_Request
*array_of_requests, int *outcount, int
*array_of_indices, MPI_Status *array_of_statuses);
```

```
int MPI_Testsome(int count, MPI_Request
*array_of_requests, int *outcount, int
*array_of_indices, MPI_Status *array_of_statuses);

int MPI_Waitall(int count, MPI_Request
*array_of_requests, MPI_Status *array_of_statuses);

int MPI_Testall(int count, MPI_Request
*array_of_requests, int *flag, MPI_Status
*array_of_statuses);
```

Все эти функции получают массив квитанций в параметре **array_of_requests** и его длину в параметре **count**. Функция **MPI_Waitany()** блокируется до тех пор, пока не завершится хотя бы одна из операций, описываемых переданными квитанциями. Она возвращает в параметре **index** индекс квитанции для завершившейся операции в массиве **array_of_requests** и, в случае, если завершилась операция приема сообщения, заполняет структуру, на которую указывает параметр **status**, информацией о полученном сообщении. **MPI_Testany()** не блокируется, а возвращает в переменной, на которую указывает параметр **flag**, ненулевое значение, если одна из операций завершилась. В этом случае она возвращает в параметрах **index** и **status** то же самое, что и **MPI_Waitany()**.

Отметим, что если несколько операций из интересующего множества завершаются одновременно, то и **MPI_Waitany()**, и **MPI_Testany()** возвращают индекс и статус лишь одной из них, выбранной случайным образом. Более эффективным вариантом в таком случае является использование соответственно **MPI_Waitsome()** и **MPI_Testsome()**: их поведение аналогично **MPI_Waitany()** и **MPI_Testany()** за тем исключением, что в случае, если одновременно завершается несколько операций, они возвращают статус для всех завершившихся операций. При этом в параметре **outcount** возвращается количество завершившихся операций, в параметре **array_of_indices** – индексы квитанций завершившихся операций в исходном массиве квитанций, а массив **array_of_statuses** содержит структуры типа **MPI_Status**, описывающие принятые сообщения (значения в массиве **array_of_statuses** имеют смысл только для операций приема сообщения). Отметим, что у функции **MPI_Testany()** нет параметра **flag** – вместо этого, она возвращает 0 в параметре **outcount**, если ни одна из операций не завершилась.

Функция **MPI_Waitall()** блокируется до тех пор, пока не завершатся все операции, квитанции для которых были ей переданы, и заполняет информацию о принятых сообщениях для операций приема сообщения в элементах массива **array_of_statuses** (при этом i-й элемент массива **array_of_statuses** соответствует операции, квитанция для которой была передана в i-м элементе массива **array_of_requests**). Функция **MPI_Testall()** возвращает ненулевое значение в переменной, адрес которой указан в параметре **flag**, если завершились все операции квитанции для которых были ей переданы, и нулевое значение в противном случае. При этом, если все операции завершились, она заполняет элементы массива **array_of_statuses** аналогично тому, как это делает **MPI_Waitall()**.

Пример 32. MPI: коммуникации «точка-точка». «Пинг-понг».

В этом примере рассматривается работа в неблокирующем режиме. Процессы посылают друг другу целое число, всякий раз увеличивая его на 1. Когда число достигнет некоего максимума, переданного в качестве параметра командной строки, все ветви завершаются. Для того, чтобы указать всем последующим ветвям на то, что пора завершаться, процесс, обнаруживший достижение максимума, обнуляет число.

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char **argv)
{
     int size, rank, max, i = 1, j, prev, next;
     MPI_Request recv_request, send_request;
     MPI_Status status;

     MPI_Init(&argc, &argv); /* Инициализируем
     библиотеку */
     MPI_Comm_size(MPI_COMM_WORLD, &size);
     /* Узнаем количество задач в запущенном
     приложении... */
     MPI_Comm_rank (MPI_COMM_WORLD, &rank);
     /* ...и свой собственный номер: от 0 до (size-
     1) */
```

```
/* определяем максимум */
if ((argc < 2) || ((max = atoi(argv[1])) <= 0))
{
     printf("Bad arguments!\n");
     MPI_Finalize();
     return 1;
}

/* каждая ветвь определяет, кому посылать и от
кого принимать сообщения */
next = (rank + 1) % size;
prev = (rank == 0) ? (size - 1) : (rank - 1);

if (rank == 0) {
     /* ветвь с номером 0 начинает «пинг-понг»
     */
     MPI_Isend(&i,   1,   MPI_INT,   1,   next,
     MPI_COMM_WORLD, &send_request);
     printf("process  %d  sent  value  \"%d\"  to
     process %d\n", rank, i, next);
     MPI_Wait(&send_request, NULL);
}

while ((i > 0) && (i < max)) {
     MPI_Irecv(&i,     1,     MPI_INT,     prev,
     MPI_ANY_TAG,               MPI_COMM_WORLD,
     &recv_request);
     MPI_Wait(&recv_request, &status);
     if (i > 0) {
          printf("process   %d   received   value
          \"%d\" " + "from process %d\n", rank,
          i++, prev);
          if (i == max) {
               i = 0;
          }
          MPI_Isend(&i,  1,  MPI_INT,  next,  0,
          MPI_COMM_WORLD, &send_request);
```

```
                printf("process %d sent value \"%d\"
                " + " to process %d\n", rank, i,
                next);
                MPI_Wait(&send_request, NULL);
            } else if (i == 0) {
                MPI_Isend(&i, 1, MPI_INT, next, 0,
                MPI_COMM_WORLD, &send_request);
                MPI_Wait(&send_request, NULL);
                break;
            }
        }

        MPI_Finalize();
        return 0;
    }
```

7.2.7 Коллективные коммуникации.

Как уже говорилось, помимо коммуникаций «точка-точка», MPI предоставляет различные возможности для осуществления **коллективных операций**, в которых участвуют все ветви приложения, входящие в определенный коммуникационный контекст. К таким операциям относятся рассылка данных от выделенной ветви всем остальным, сбор данных от всех ветвей на одной выделенной ветви, рассылка данных от всех ветвей ко всем, а также коллективная пересылка данных, совмещенная с их одновременной обработкой.

Отметим, что хотя все те же самые действия можно запрограммировать с использованием коммуникаций «точка-точка», однако более целесообразным является применение коллективных функций, и тому есть несколько причин. Во-первых, во многих случаях использование коллективной функции может быть эффективнее, нежели моделирование ее поведения при помощи функций обмена «точка-точка», так как реализация коллективных функций может содержать некоторую оптимизацию (например, при рассылке всем ветвям одинаковых данных может использоваться не последовательная посылка всем ветвям сообщений от одной выделенной ветви, а распространение сообщения между ветвями по принципу двоичного дерева). Кроме того, при использовании коллективных функций сводится к нулю риск программной ошибки, связанной с тем, что сообщение из коллективного потока данных будет перепутано с другим, индивидуальным сообщением

(например, получено вместо него при использовании констант **MPI_ANY_SOURCE**, **MPI_ANY_TAG** в качестве параметров функции приема сообщения). При передаче сообщений, формируемых коллективными функциями, используется временный дубликат заданного коммуникатора, который неявно создается библиотекой на время выполнения данной функции и уничтожается автоматически после ее завершения. Таким образом, сообщения, переданные коллективной функцией, могут быть получены в других или в той же самой ветви только при помощи той же самой коллективной функции.

Важной особенностью любой коллективной функции является то, что ее поведение будет корректным только в том случае, если она была вызвана во всех ветвях, входящих в заданный коммуникационный контекст. В некоторых (или во всех) ветвях при помощи этой функции данные будут приниматься, в некоторых (или во всех) ветвях – отсылаться. Иногда с помощью одного вызова происходит одновременно и отправка, и прием данных. Как правило, все ветви должны вызвать коллективную функцию с одними и теми же значениями фактических параметров (за исключением параметров, содержащих адреса буферов – эти значения, разумеется, в каждой ветви могут быть своими). Все коллективные функции имеют параметр-коммуникатор, описывающий коммуникационный контекст, в котором происходит коллективный обмен.

Семантика буферизации для коллективных функций аналогична стандартному режиму буферизации для одиночных коммуникаций в блокирующем режиме: возврат управления из коллективной функции означает, что вызвавшая ветвь может повторно использовать буфер, в котором содержались данные для отправки (либо гарантирует, что в буфере для приема находятся принятые данные). При этом, отнюдь не гарантируется, что в остальных ветвях к этому моменту данная коллективная операция также завершилась (и даже не гарантируется, что она в них уже началась). Таким образом, хотя в некоторых случаях возврат из коллективных функций действительно происходит во всех ветвях одновременно, нельзя полагаться на этот факт и использовать его для целей синхронизации ветвей. Для явной синхронизации ветвей может использоваться лишь одна специально предназначенная коллективная функция – это описанная выше функция **MPI_Barrier()**.

Коллективный обмен данными.

Для отправки сообщений от одной выделенной ветви всем ветвям, входящим в данный коммуникационный контекст, служат функции **MPI_Bcast()** и **MPI_Scatter()**. Функция **MPI_Bcast()** служит для отправки всем ветвям одних и тех же данных:

```
#include <mpi.h>

int MPI_Bcast(void* buffer, int count, MPI_Datatype
datatype, int root, MPI_Comm comm);
```

Параметр **comm** задает коммуникатор, в рамках которого осуществляется коллективная операция. Параметр **root** задает номер ветви, которая выступает в роли отправителя данных. Получателями выступают все ветви из данного коммуникационного контекста (включая и ветвь-отправитель). При этом параметр **buffer** во всех ветвях задает адрес буфера, но ветвь-отправитель передает в этом буфере данные для отправки, а в ветвях-получателях данный буфер используется для приема данных. После возврата из данной функции в буфере каждой ветви будут записаны данные, переданные ветвью с номером **root**.

Параметры **datatype** и **count** задают соответственно тип данных, составляющих тело сообщения, и количество элементов этого типа в теле сообщения. Их значения должны совпадать во всех ветвях. Разумеется, для корректного выполнения функции необходимо, чтобы размер буфера, переданного каждой ветвью в качестве параметра **buffer**, был достаточен для приема необходимого количества данных.

Как уже говорилось, функция **MPI_Bcast()** осуществляет пересылку всем ветвям одних и тех же данных. Однако, часто бывает необходимо разослать каждой ветви свою порцию данных (например, в случае распараллеливания обработки большого массива данных, когда одна выделенная ветвь осуществляет ввод или считывание всего массива, а затем отсылает каждой из ветвей ее часть массива для обработки). Для этого предназначена функция **MPI_Scatter()**:

```
#include <mpi.h>

int MPI_Scatter(void* sendbuf, int sendcount,
MPI_Datatype sendtype, void* recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm);
```

Параметр **root** здесь опять задает номер выделенной ветви, являющейся отправителем данных, параметр **comm** – коммуникатор, в рамках которого осуществляется обмен. Параметры **sendbuf**, **sendtype**, **sendcount** имеют смысл только для ветви-отправителя и задают соответственно адрес буфера с данными для рассылки, их тип и количество элементов заданного типа, которое нужно отправить каждой из ветвей. Для остальных ветвей эти параметры игнорируются. В результате действия функции массив данных в **sendbuf** делится на N равных частей (где N – количество ветвей в коммуникаторе), и каждой ветви посылается i-я часть этого массива, где i – уникальный номер данной ветви в этом коммуникаторе. Отметим, что для того, чтобы вызов был корректным, буфер **sendbuf** должен, очевидно, содержать **N*sendcount** элементов (ответственность за это возлагается на программиста).

Параметры **recvbuf**, **recvtype**, **recvcount** имеют значение для всех ветвей (в том числе и ветви-отправителя) и задают адрес буфера для приема данных, тип принимаемых данных и их количество. Формально типы отправляемых и принимаемых данных могут не совпадать, однако жестко задается ограничение, в соответствии с которым общий размер данных, отправляемых ветви, должен точно совпадать с размером данных, которые ею принимаются.

Операцию, обратную **MPI_Scatter()**, – сбор порций данных от всех ветвей на одной выделенной ветви – осуществляет функция **MPI_Gather()**:

```
#include <mpi.h>
int MPI_Gather(void* sendbuf, int sendcount,
MPI_Datatype sendtype, void* recvbuf, int recvcount,
MPI_Datatype recvtype, int root, MPI_Comm comm);
```

Параметр **root** задает номер ветви-получателя данных; отправителями являются все ветви из данного коммуникационного контекста (включая и ветвь с номером **root**). В результате выполнения этой функции в буфере **recvbuf** у ветви с номером **root** формируется массив, составленный из N равных частей, где i-я часть представляет собой содержимое буфера **sendbuf** у ветви с номером i (т.е. порции данных от всех ветвей располагаются в порядке их номеров). Параметры **sendbuf**, **sendtype**, **sendcount** должны задаваться всеми ветвями и описывают отправляемые данные; параметры **recvbuf**, **recvtype**, **recvcount** описывают буфер для приема данных, а также количество и тип принимаемых данных и

имеют значение только для ветви с номером **root**, а у остальных ветвей игнорируются. Для корректной работы функции необходимо, чтобы буфер **recvbuf** имел достаточную емкость, чтобы вместить данные от всех ветвей.

Работу функций **MPI_Scatter()** и **MPI_Gather()** для случая 3х ветвей и **root=0** наглядно иллюстрирует Рис. 24.

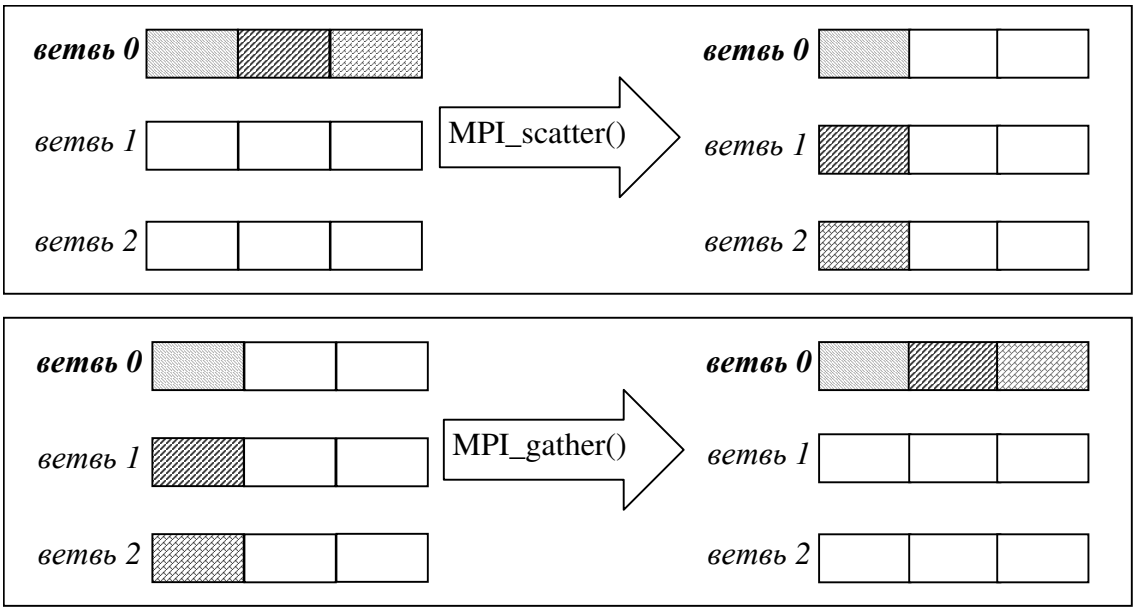

Рис. 24 Работа MPI_scatter() и MPI_gather()

Существует также возможность осуществить сбор данных от всех ветвей в единый массив, но так, чтобы доступ к этому результирующему массиву имела не одна выделенная ветвь, а все ветви, принимающие участие в операции. Для этого служит функция **MPI_Allgather()**:

```
#include <mpi.h>

int MPI_Allgather(void* sendbuf, int sendcount,
MPI_Datatype sendtype, void* recvbuf, int recvcount,
MPI_Datatype recvtype, MPI_Comm comm);
```

Работа этой функции проиллюстрирована на Рис. 25. Эта функция отличается от предыдущей лишь тем, что у нее отсутствует параметр **root**, а параметры **recvbuf**, **recvtype**, **recvcount** имеют смысл для всех ветвей. В результате работы этой функции на каждой из ветвей в буфере **recvbuf** формируется результирующий массив, аналогично тому, как описано для **MPI_Gather()**. Ответственность за то, чтобы приемные буфера имели достаточную емкость, возлагается на программиста.

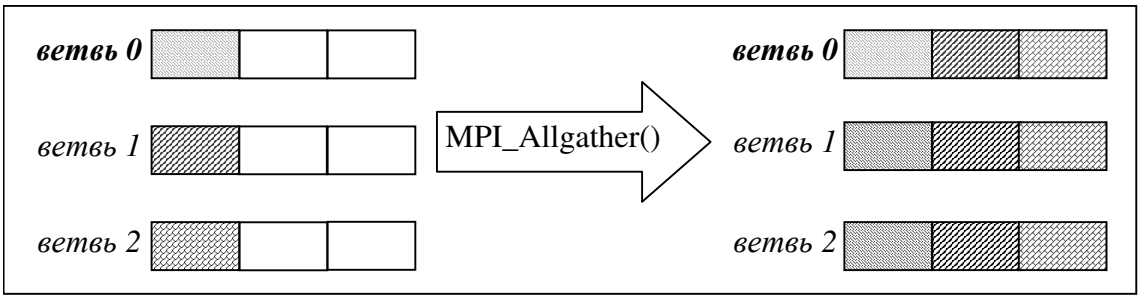

Рис. 25 Работа MPI_Allgather()

Функция **MPI_Alltoall()** представляет собой расширение **MPI_Allgather()**, заключающееся в том, что каждая ветвь-отправитель посылает каждой конкретной ветви-получателю свою отдельную порцию данных, подобно тому, как это происходит в **MPI_Scatter()**. Другими словами, i-я часть данных, посланных ветвью с номером j, будет получена ветвью с номером i и размещена в j-м блоке ее результирующего буфера.

```
#include <mpi.h>

int MPI_Alltoall(void* sendbuf, int sendcount,
MPI_Datatype sendtype, void* recvbuf, int recvcount,
MPI_Datatype recvtype, MPI_Comm comm);
```

Параметры этой функции аналогичны параметрам **MPI_Allgather()**.

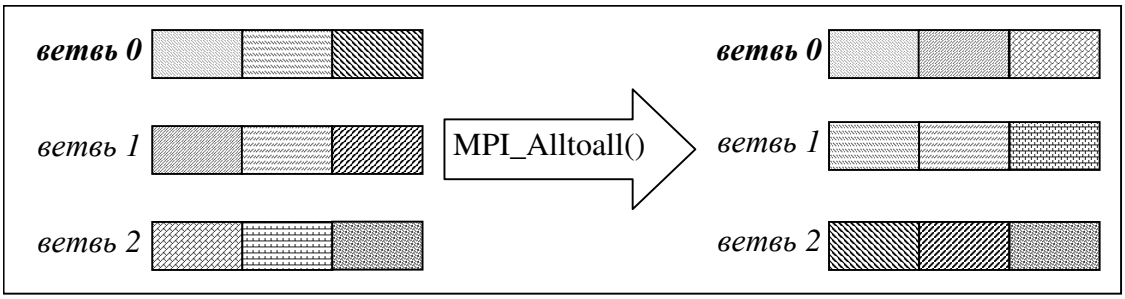

Рис. 26 Работа функции MPI_Alltoall()

Помимо рассмотренных выше функций, библиотека MPI предоставляет так называемые «векторные» аналоги функций **MPI_Scatter()**, **MPI_Gather()**, **MPI_Allgather()** и **MPI_Alltoall()**, позволяющие разным ветвям пересылать или получать части массива разных длин, однако их подробное рассмотрение выходит за рамки данного пособия.

Коллективный обмен, совмещенный с обработкой данных.

Во многих случаях после сбора данных от всех ветвей непосредственно следует некоторая обработка полученных данных, например, их суммирование, или нахождение максимума и т.п. Библиотека MPI предоставляет несколько коллективных функций, которые позволяют совместить прием данных от всех ветвей и их обработку. Простейшей из этих функций является **MPI_Reduce()**:

```
#include <mpi.h>

int MPI_Reduce(void* sendbuf, void* recvbuf, int
count, MPI_Datatype datatype, MPI_Op op, int root,
MPI_Comm comm);
```

Функция **MPI_Reduce()** является своего рода расширением функции **MPI_Gather()**: ветвь с номером **root** является получателем данных, отправителями являются все ветви. Параметры **sendbuf**, **datatype**, **count** имеют смысл для всех ветвей и задают адрес буфера с данными для отправки, тип отправляемых данных и их количество. Тип и количество данных должны совпадать во всех ветвях. Однако в процессе получения данных над ними поэлементно производится операция, задаваемая параметром **op**, результат которой заносится в соответствующий элемент буфера **recvbuf** ветви с номером **root**. Так, например, если параметр **op** равен константе **MPI_SUM**, что означает суммирование элементов, то в i-й элемент буфера **recvbuf** ветви с номером **root** будет записана сумма i-х элементов буферов **sendbuf** всех ветвей. Параметр **recvbuf** имеет смысл лишь для ветви с номером **root**, для всех остальных ветвей он игнорируется.

Для описания операций в MPI введен специальный тип **MPI_Op**. MPI предоставляет ряд констант этого типа, описывающих стандартные операции, такие как суммирование (**MPI_SUM**), умножение (**MPI_PROD**), вычисление максимума и минимума (**MPI_MAX** и **MPI_MIN**) и т.д. Кроме того, у программиста существует возможность описывать новые операции.

Функция **MPI_Allreduce()** представляет собой аналог **MPI_Reduce()** с той лишь разницей, что результирующий массив формируется на всех ветвях (и, соответственно, параметр **recvbuf** также должен быть задан на всех ветвях):

```
#include <mpi.h>
```

```
int MPI_Allreduce(void* sendbuf, void* recvbuf, int
count, MPI_Datatype datatype, MPI_Op op, MPI_Comm
comm);
```

Функция **MPI_Reduce_scatter()** сочетает в себе возможности **MPI_Reduce()** и **MPI_Scatter()**:

```
#include <mpi.h>

int MPI_Reduce_scatter(void* sendbuf, void* recvbuf,
int *recvcounts, MPI_Datatype datatype, MPI_Op op,
MPI_Comm comm);
```

В процессе ее выполнения сначала над массивами **sendbuf** поэлементно производится операция, задаваемая параметром **op**, а затем результирующий массив разделяется на N частей, где N – количество ветвей в коммуникационном контексте, и каждая из ветвей получает в буфере **recvbuf** свою часть в соответствии со своим номером. Элементы массива **recvcounts** задают количество элементов, которое должно быть отправлено каждой ветви, что позволяет каждой ветви получать порции данных разной длины.

Наконец, функция **MPI_scan()** представляет собой некоторое сужение функции **MPI_AllReduce()**:

```
#include <mpi.h>

int MPI_Scan(void* sendbuf, void* recvbuf, int
count, MPI_Datatype datatype, MPI_Op op, MPI_Comm
comm);
```

В буфере **recvbuf** ветви с номером i после возврата из этой функции будет находиться результат поэлементного применения операции **op** к буферам **sendbuf** ветвей с номерами от 0 до i включительно. Очевидно, что в буфере **recvbuf** ветви с номером N-1 будет получен тот же результат, что и для аналогичного вызова **MPI_Allreduce()**.

Пример 33. MPI: применение коллективных коммуникаций.

В данном примере производится приближенное вычисление числа π путем численного интегрирования на отрезке методом прямоугольников. Количество точек в разбиении определяет пользователь. Ветвь с номером 0 осуществляет ввод данных от пользователя, а затем, по окончании вычислений, - сбор данных от всех ветвей и вывод результата.

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char **argv)
{
     int size, rank, N, i;
     double h, sum, x, global_pi;

     MPI_Init(&argc,   &argv);   /*   Инициализируем
     библиотеку */
     MPI_Comm_size(MPI_COMM_WORLD, &size);
     /*   Узнаем   количество   задач   в   запущенном
     приложении... */
     MPI_Comm_rank (MPI_COMM_WORLD, &rank);
     /* ...и свой собственный номер: от 0 до (size-
     1) */

     if (rank == 0) {
          /* ветвь с номером 0 определяет количество
          итераций... */
          printf("Please, enter the total iteration
          count:\n");
          scanf("%d", &N);
     }
     /*... и рассылает это число всем ветвям */
     MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
     h = 1.0 / N;
     sum = 0.0;

     /* вычисление частичной суммы */
     for (i = rank + 1; i <= N; i += size) {
          x = h * (i - 0.5);
          sum += f(x);
     }
     sum = h * sum;
```

```
    /* суммирование результата на ветви с номером 0
    */
    MPI_Reduce(&sum,  &global_pi,  1,  MPI_DOUBLE,
    MPI_SUM, 0, MPI_COMM_WORLD);
    if (rank == 0) {
        printf("The    result    is    %.10f\n",
        global_pi);
    }
    MPI_Finalize();
    return 0;
}
```

8 Алфавитный указатель упоминаемых библиотечных функций и системных вызовов.

_exit() 51
accept() 133
alarm() 43
bind() 131
connect() 132
creat() 70
exec(), семейство 47
exit(), функция 52
fcntl() 47, 81
fork() 42
ftok(), функция 101
getpid() 45
getppid() 45
kill() 67
listen() 133
longjmp() 92
mkfifo() 89
mknod() 89
MPI_Abort() 155
MPI_Allgather() 176
MPI_Allreduce() 179
MPI_Alltoall() 177
MPI_Barrier() 155
MPI_Bcast() 174
MPI_Bsend() 161
MPI_Comm_rank() 154
MPI_Comm_size() 154
MPI_Finalise() 155
MPI_Gather() 175
MPI_Get_count() 163
MPI_Ibsend() 167
MPI_Init() 154
MPI_Irecv() 166
MPI_Irsend() 167
MPI_Isend() 166
MPI_Issend() 167
MPI_Probe() 164
MPI_Recv() 162
MPI_Reduce() 178
MPI_Reduce_scatter() 179
MPI_Rsend() 161
MPI_Scan() 179
MPI_Scatter() 174
MPI_Send() 160
MPI_Ssend() 161
MPI_Test() 168
MPI_Testall() 169
MPI_Testany() 168
MPI_Testsome() 169
MPI_Wait() 168
MPI_Waitall() 169
MPI_Waitany() 168
MPI_Waitsome() 168
msgctl() 107
msgget() 105
msgrcv() 106

msgsnd()....105
nice()....60
pipe()....80
ptrace()....94
recv()....135
recvfrom()....136
semctl()....121
semget()....118
semop()....119
send()....134
sendto()....136
setjmp()....92
shmat()....115
shmctl()....116
shmdt()....115
shmget()....114
shutdown()....136
sigaction()....78
sigaddset()....76
sigdelset()....76
sigemptyset()....76
sigfillset()....76
sigismember()....76
signal()....68
sigpending()....76
sigprocmask()....75
socket()....129
unlink()....71
wait()....52

9 Список литературы

1. Д.Цикритзис, Ф.Бернстайн. Операционные системы. Москва, Мир, 1977.
2. Э.С.Таненбаум. Современные операционные системы, 2-е издание. Санкт-Петербург, Издательский дом Питер, 2002.
3. П.Кейлингер. Элементы операционных систем. Москва, Мир, 1985.
4. Ч.Хоар. Взаимодействующие последовательные процессы. Москва , Мир, 1989.
5. Pate S.D. UNIX Internals. A Practical Approach. Addison-Wesley Longman, 1997.
6. К.Кристиан. Введение в операционную систему UNIX. Москва, Финансы и статистика. 1985.
7. Б.К.Керниган, Р.Пайк. UNIX – универсальная среда программирования. Москва, Финансы и статистика, 1992.
8. М.Дансмур, Г.Дейвис. Операционная система UNIX и программирование на языке Си. Москва, Радио и связь, 1989.
9. Т.Чан. Системное программирование на C++ для UNIX. Киев, Издательская группа BHV, 1997.
10. М.Устюгов. Введение в TCP/IP. Москва, .МГУ, 1997.
11. Б.Керниган, Д.Ритчи. Язык программирования Си. Издание 3-е, исправленное. Санкт-Петербург, Невский Диалект, 2001.
12. И.В.Машечкин, М.И.Петровский, П.Д.Скулачев, А.Н.Терехин. Системное Программное Обеспечение: файловые системы ОС Unix и Windows NT. Москва, Диалог-МГУ,1997.
13. А.Робачевский Операционная система UNIX. Санкт-Петербург, BHV-Санкт-Петербург, 1998
14. К.Хэвиленд, Д.Грей, Б.Салама. Системное программирование в UNIX. Москва, ДМК, 2000.
15. Н.Д.Васюкова, И.В.Машечкин, В.В.Тюляева, Е.М.Шляховая. Краткий конспект семинарских занятий по языку Си. Москва, МГУ, 1999.
16. В.Столлингс. Операционные системы. 4-ое издание. Москва, Издательский дом Вильямс, 2002.