ГЛАВА 3

ОСОБЕННОСТИ АРХИТЕКТУРЫ РАСПРЕДЕЛЕННЫХ ОПЕРАЦИОННЫХ СИСТЕМ

3.1. ОПРЕДЕЛЕНИЕ РАСПРЕДЕЛЕННОЙ СИСТЕМЫ ОБРАБОТКИ ДАННЫХ

Распределенная система обработки данных (РСОД) или распределенная вычислительная система (РВС) — это среда, в которой компоненты системы или ресурсы: процессоры, память, принтеры, графические станции, программы, данные и т.д., связанны вместе посредством сети, которая позволяет пользователям представлять ВС как единую вычислительную среду и иметь доступ к ее ресурсам [52]. Распределенная ВС имеет некоторые особенности, характерные только для нее:

* О*граниченность*. (failure). Она связана с тем, что количество узлов в сети ограничено и они являются независимыми компонентами сети.
* *Идентификация.* (naming). Каждый из ресурсов сети должен однозначно идентифицироваться.
* *Распределенное управление* (distributing control). Каждый из узлов должен иметь программно-аппаратные возможности выполнения функций управления сетью. Однако повышенные требования к надежности распределенной системы вызывают необходимость избегать применения алгоритмов управления, выделяющих привилегированные по сравнению с другими узлы в сети.
* *Гетерогенность* (heterogeneity). Узлы сети могут быть разнородны, с различной длиной слов и байтовой организацией. Этот аспект может касаться и программного обеспечения, как прикладного так и системного, применяемого для каждого из узлов.

При написании прикладных программ пользователям не требуется знать особенности аппаратного обеспечения сети. Пользователь распределенной вычислительной системы не обязательно должен иметь представление о деталях системы, с которой он работает. Все необходимые пользователю действия при взаимодействии с системными ресурсами сети обеспечиваются для него посредством операционной системы.

Рассмотрим более подробно тот раздел программного обеспечения, который, собственно, и решает задачи, связанные с обеспечением сокрытия вышеупомянутых особенностей системы и облегчением работы пользователя в распределенной вычислительной среде, а также обеспечением эффективного функционирования РСОД. В дальнейшем для определения этого системного программного обеспечения используется термин "распределенная операционная система" (РОС) (distributed operating system).

3.2. КОНЦЕПЦИИ ПОСТРОЕНИЯ РОС

Существуют 3 различные отправные точки, которые могут быть учтены при проектировании РОС .

Оригинальность (from scratch). Так как концептуально по архитектуре и принципам функционирования РОС значительно отличаются от обычных ОС, то имеются принципиальные аргументы в пользу построения совершенно нового типа ОС с новыми характеристиками и использования новых принципов решения задач организации вычислительного процесса в системе и взаимодействия компонентов. Однако, это неизбежно приведет к необходимости пересмотра существующего *системного*, *математического* и *программного* обеспечения, а также модификации прикладного, т.к. новые предоставляемые возможности требуют глубокого пересмотра принципов программирования, а возможно, и аппаратного обеспечения (рис.3.1.а).

Модифицируемость существующих систем (modify an existing system). Обеспечивая совместимость со многими существующими системами, можно уменьшить объем программного обеспечения, который все же необходимо будет написать заново. Однако, в этом случае поиск компромисса для поддержания совместимости может, естественно, вызвать ряд затруднений при проектировании таких систем и ухудшить их параметры, что наблюдается при разработке программного обеспечения РС (рис. 3.1.б).

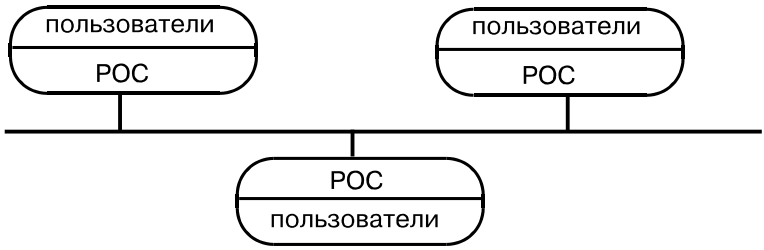

Рис. 3.1.а. *Подход 1*

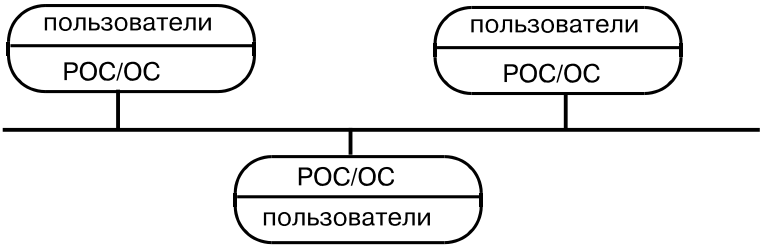

Рис. 3.1.б. *Подход 2*

3. *Расширяемость (Открытость)* (the layered approach). Не всегда возможно заново спроектировать или модифицировать существующую ОС, поэтому этот подход состоит в добавлении к существующему программному обеспечению нового раздела (операционного слоя) и, таким образом, обеспечении новых свойств системы. Лучшим примером здесь могут служить так называемые сетевые ОС. Однако, при этом возникают практически те же проблемы, что и в предыдущем подходе (рис.3.1.в) [52].

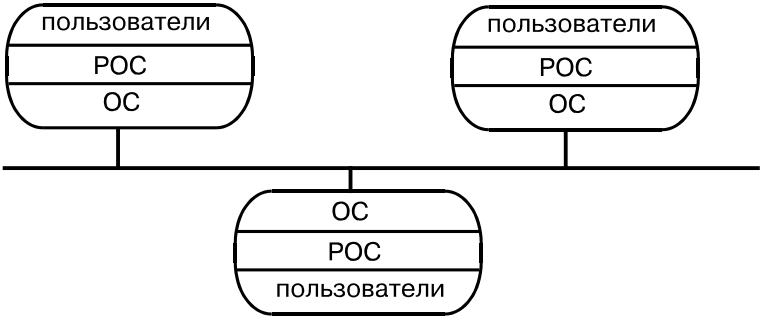

Рис. 3.1.в. *Подход 3*

3.3. КЛАССИФИКАЦИЯ РАСПРЕДЕЛЕННЫХ СИСТЕМ ОБРАБОТКИ ДАННЫХ

Существуют две различные концепции построения архитектуры распределенных систем (РС):

1. *На основе сервера* (server model). Стандартные функции РОС для управления ресурсами системы и планирования процессов выполнены не на уровне каждого узла системы, а на уровне *особого серверного узла*. Пользовательские узлы обеспечены только лишь необходимым минимумом программ для доступа к

серверу. Возможную опасность здесь таит в себе незащищенность отдельных пользовательских узлов от несанкционированного доступа или порчи программ при вызове отдельных обслуживающих функций, кроме этого, имеются трудности совместного использования общих ресурсов (рис.3.2)

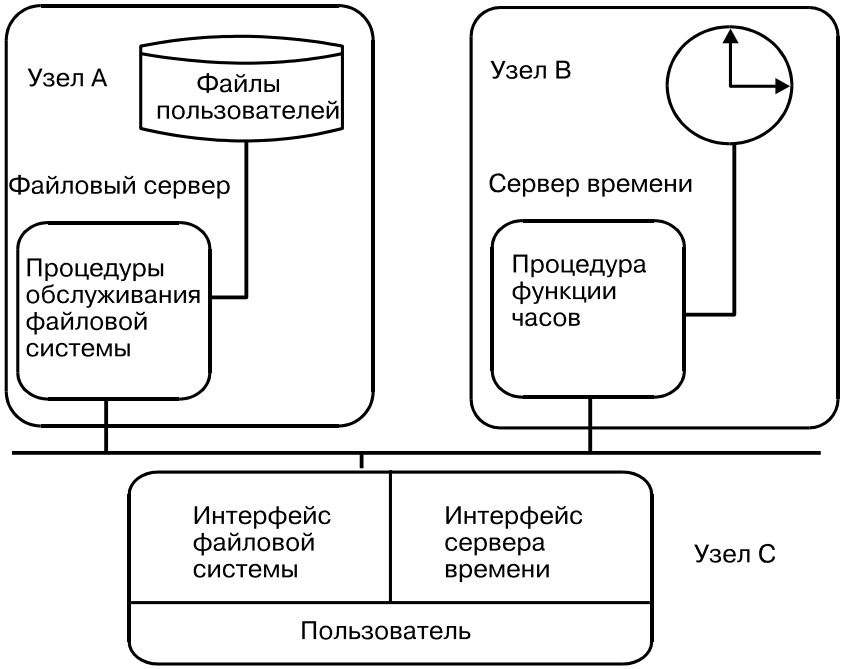

Рис. 3.2. Модель РС с сервером

2. *Интегрированная модель* (integrated model). В этой альтернативной предыдущему подходу концепции каждый узел инсталирован "полной" (интегрированной) (complete-integrated) версией ОС. В этом случае не существует особых ограничений для каждого узла при выполнении функций, процедур и т.д., при этом проблемы серверной архитектуры могут возникнуть только в особых ситуациях. Недостаток данного подхода — большие затраты в оборудовании и программном обеспечении. (рис.3.3).

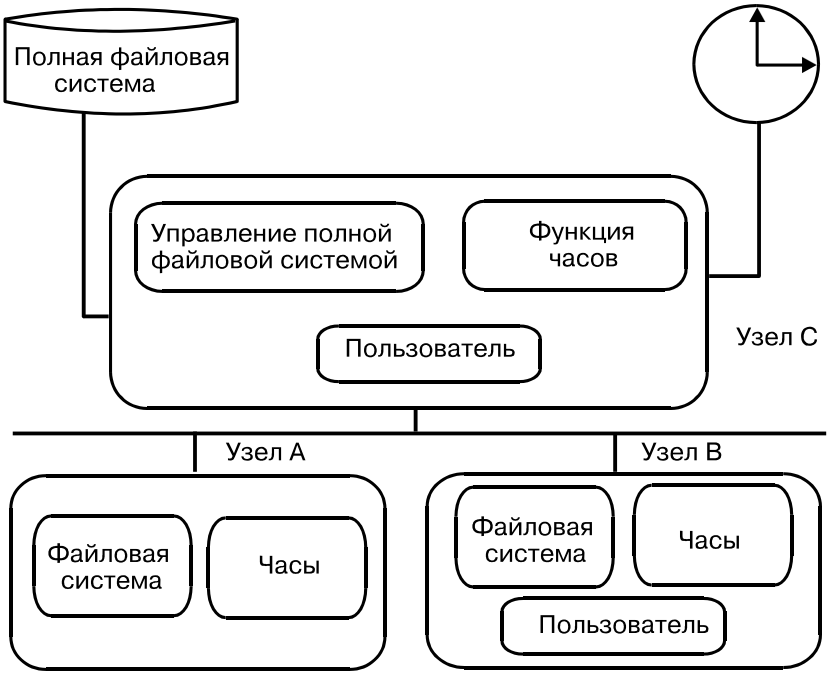

Рис. 3.3. Интегрированная модель

Можно выделить еще один тип архитектуры — комбинированный, который соединяет в себе признаки вышеупомянутых двух. Здесь некоторые узлы будут иметь лишь некоторую часть программного обеспечения для доступа к серверу (например, это касается низкоприоритетных клиентов — пользователей), а другие привилегированные пользователи (это может касаться также обслуживающего персонала распределенной системы) — полностью установленную ОС и дополнительный набор программного обеспечения для выхода на сервер. При таком подходе ВС, особо не проигрывая в количественных показателях, значительно выигрывает в качественных параметрах всей системы.

3.4. ПРОБЛЕМЫ УПРАВЛЕНИЯ ДАННЫМИ В РОС

Основой РОС является распределенная файловая система. При ее проектировании решаются следующие проблемы:

* Какую модель файловой системы выбрать?

* Какой функциональный набор системных функций необходим для работы такой системы?

Как правило структура такой файловой системы незначительно отличается от структуры файловой системы в однопроцессорных системах. Базовыми понятиями распределенной файловой системы являются система директорий и файловых обозначений. При этом возникает проблема *прозрачности* (transparency):

* В какой степени имя файла связано с его расположением?
* Может ли система перемещать файл без файлового имени?

В различных ОС эти вопросы решаются по разному.

Файловые различия в распределенных системах — это целый комплекс злободневных и актуальных проблем. Выделяют различные семантические модели: UNIX — семантику, сессионную семантику, "неизменные файлы" и деловую семантику. Каждая имеет свои преимущества и недостатки. UNIX — семантика интуитивна по сути и хорошо знакома большинству программистов, но дорога в исполнении. Сессионная семантика менее детерменирована, но более эффективна. "Неизменные файлы" неизвестны большинству, и к тому же в них усложнена изменяемость файлов.

Исполнение распределенной файловой системы предполагает решение многих проблем связанных с:

- полнотой системы для полноценного существования и работы;
- выполнением операций кеширования;
- управляемостью файловых перемещений, копированием и т.д.

Эффективное решение этих проблем имеет важное значение для проектировщиков и пользователей. Решение этих задач постепенно выходит на качественно новый уровень и касается изменений не только программного обеспечения, математического аппарата и организации вычислительного процесса, но и технологии аппаратного обеспечения, размеров системы со всеми вытекающими отсюда последствиями, связанными с легкостью пользования, переносимостью ошибок и т.д.

3.5. ОСНОВНЫЕ ПРОБЛЕМЫ В РАЗРАБОТКЕ ПЕРСПЕКТИВНЫХ РОС

Защищенность (protection). РОС должна обладать средствами защиты не только самой себя, но и программ, и данных, обрабатываемых распределенной вычислительной средой. Следует различать глобальные и локальные средства защиты, отличающиеся сферой действия. С точки зрения функционирования всей системы в целом для РОС более важными являются средства глобальной защиты, т.к. нарушение защиты операционной системы может привести к катастрофическим последствиям.

Общность ресурсов (raplication, availability). Принцип общности ресурсов, иногда его называют принципом запрашиваемости, может значительно улучшить характеристики распределенной вычислительной системы за счет

перераспределения критических ресурсов в особых ситуациях. Причем любой ресурс системы может быть таковым. На РОС возложены функции управления распределением запросов, слежения за совпадением запросов доступа к запрашиваемым ресурсам и их занятостью, балансным планированием доступа к ограниченным ресурсам системы. Эти функции РОС должна выполнять, учитывая интересы как самой РСОД, так и интересы пользователей.

Соглашения (протоколизм) (transactions). Файловая система РОС должна обеспечивать протокольный механизм (механизм соглашений) как на общем уровне, так и в более локальной среде. С этой точки зрения здесь интересна гибкость этого механизма.

Рассмотрим данные проблемы более детально.

3.5.1. Идентификация (именование)

Все объекты в любой ВС системе (файлы, процессы устройства и т.д.) должны быть поименованы (идентифицированы). Каждый объект должен иметь как минимум два имени. Символический идентификатор, задаваемый пользователем, и внутреннее имя, используемое системой. Последнее может быть связано с размещением объекта, т.е. с его адресом, или с возможностью поиска по нему адреса объекта. Его можно также использовать при планировании, в определении значимости самого объекта в системе и т.д. Обычно пользовательское имя преобразуется во внутреннее посредством особого рода карт в которых отображается пространственное (имя узла) расположение соответствующих файлов, путь доступа к ним и механизм доступа. Особое значение идентификация объектов приобретает при решении задач пространственного статического или динамического планирования параллельных процессов в РОС.

РОС должны обеспечивать функционирование механизма имен. Составление карты имен здесь усложняется, т.к. имена используются для ссылки на ресурсы различных узлов. Попытка решить эту проблему была предпринята в DECnet [64]. Узлы в DECnet используют стандартные операционные системы, такие как VMS, RSX, ULTRIX и один из подходов доступа к файлам из любого узла сети.

Практически все системы имеют иерархическую структуру и VMS — имена [65] поддерживают именно эту структуру. Полное VMS — имя состоит из имени устройства, предшествующего любому количеству имен директорий, следующего за ним имени файла и расширения. (рис.3.4.)

В больших сетях идентификация осуществляется с помощью расширения иерархии на один уровень вверх, включающий имя узла. (рис.3.5.)

Файлы отдаленных узлов становятся доступными при указании перед полным именем файла в разделе имени узла.

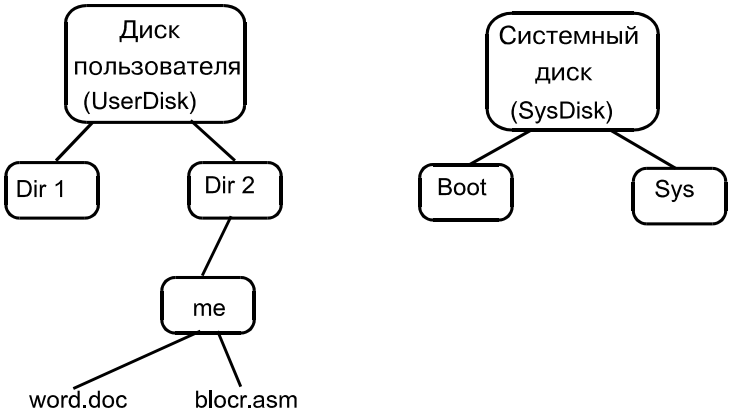

Рис.3.4. VMS директории в одном узле

Прим: на рис. 3.4 показан тип структуры VMS с директориями на двух устройствах — UserDisk и SysDisk. Директория "Dir2" содержит одну поддиректорию "me", в которой находятся два файла: Word.doc и Block.asm. Полный путь к файлу Word.doc имеет вид UserDisk:[dir2.me]Word.doc

При таком подходе имя узла должно быть уникальным в своем роде, и каждый узел должен знать имена всех других узлов в сети. Эта схема относительно легко реализуема и понятна для пользователя.

Эта концепция простая для технической реализации, но, тем не менее, обладает рядом недостатков. Недостатки такой реализации обусловлены приобретением РОС новых свойств, связанных с обеспечением реконфигурации, статическим, динамическим и балансовым планированием. Так как местоположение файла — это часть его имени, то при перемещении файла из одного узла в другой, должно меняться его имя. Однако такие изменения должны фиксироваться системой. Таким образом, вся информация о файле, так или иначе, подлежит изменению. Если существуют две копии файла, то они должны иметь различные имена и обрабатываться индивидуально.

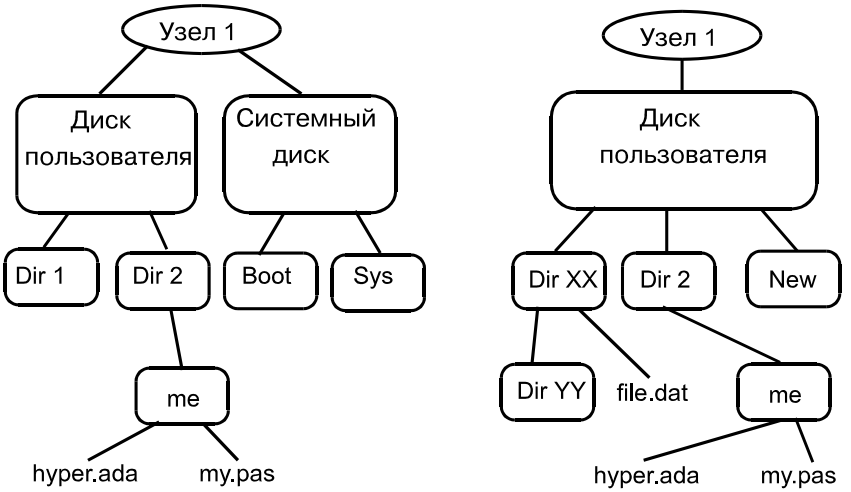

Рис.3.5. DECnet именование с несколькими узлами

Прим: Пользователи могут запрашивать ресурсы по всей сети с указанием имени узла. Полный путь к файлу My.pas на узле с названием Узел 1 (Node1) имеет вид Node1::UserDisk:[dir2.me]my.pas

Проблема такого механизма идентификации состоит в том, что некоторые скрытые системные детали сети становятся видимыми пользователю, т.к. в этом случае местоположение файла в форме имени его узла является неотъемлемой его частью. Для обеспечения скрытости сети от пользователя необходимо предварительно отказаться от такого подхода. Это понятие известно как *прозрачность* сети.

3.5.2. Основные характеристики РОС

* *Прозрачность сети (transparency)*

Прозрачность сети требует, чтобы детали сети были скрыты от конечных пользователей. Так, пользователь истинно прозрачной распределенной системы должен находиться под впечатлением, что он использует одиночную мини — (супер) — ЭВМ. При этом перемещение файлов должно осуществляться без изменения их имен. Если это требование реализовано, то написание программ для распределенной системы не должно быть сложнее создания программ для

сосредоточенной системы. Определим некоторые аспекты решения этой задачи [107]:

* имя любого ресурса системы не должно быть связано с его местоположением;
* имя должно быть уникальным в глобальном смысле, и не имеет значения в каком месте системы оно будет использовано;
* имя ресурса желательно связать с соответствующим применением этого ресурса, независимо от места его применения (рис.3.6.).

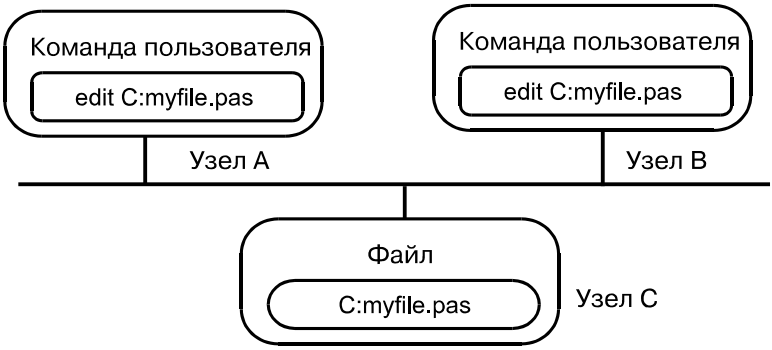

Рис.3.6. Прозрачность сети

Таким образом, прозрачность сети действительно может эффективно решить проблему скрытости местоположения ресурсов от пользователя, однако для диспетчеров распределенных систем ее реализация сталкивается со многими конфликтными ситуациями.

* *Локальная автономность (local autonomy)*

Это естественная норма для пользователя (администратора) узла в распределенной системе заключается в возможности контроля и управления ресурсами. Однако здесь разработчики сталкиваются с проблемой обеспечения прозрачности всей сети и возникающими при этом конфликтными проблемами — все команды должны иметь одинаковый эффект во всех узлах сети, но в этом случае пользователь другого узла уже не может так же использовать данную команду (ресурс), а может вообще не иметь доступа к нему. Поэтому в данной ситуации от локальной автономности прийдется отказаться.

* *Оптимизация управления ресурсами*

В некоторых случаях пользователь может иметь желание точно знать о местоположении ресурсов (а иногда и управлять ими). Это может потребовать оптимизации всей системы, путем размещения часто используемых ресурсов в те

места, где они были бы легко доступны. При этом значительно усложняются функции управления всей системой, а вместе с этим и РОС. Тогда необходимо будет либо вообще отказаться от механизма прозрачности, либо найти другое средство для реализации оптимизации в сети, либо совсем отказаться от такой реконфигурации системы.

* *Разнородность (heterogeneity)*

Трудно добиться полной прозрачности сети. Проблема возникает сразу в двух аспектах. Во-первых, скрытность аппаратного обеспечения не позволяет определить, какие ресурсы могут быть использованы в данном узле, особенно это касается неоднородной структуры, где узлы сильно разнятся между собой по своим аппаратным ресурсам, программному обеспечению и структурам данных. Во-вторых, более сложная проблема возникает в том случае, если узлы используют различные ОС [107]. Очень сложно выявить и учесть все это, учитывая требования к прозрачности сети. В DECnet, например, количество операций, используемых между узлами с гетерогенными свойствами, — это подмножество операций, которое употребляется в гомогенной структуре [64].

3.6. УПРАВЛЕНИЕ И КОНТРОЛЬ

3.6.1. Общие проблемы управления РСОД

РОС планирует, управляет и контролирует доступ и использование ресурсов, находящихся в ее подчинении. Ярким примером тому служит планирование процессов, обслуживание очередей к ресурсам, закрытие и открытие файлов и т.д. В сосредоточенной системе все алгоритмы, которые использует ОС, централизованы, т.е. конечное решение принимается только в одном месте. В распределенных системах в формировании конечного решения могут принимать участие и узлы. Алгоритмы, решения которых децентрализованы, называются *распределенными* (distributed), т.к. контроль и управление ресурсами в этом случае может производиться как на уровне одного узла, так и на уровне всей сети.

Распределенная система обеспечивает возможность одновременного развития разных типов процессов, принимающих решения. Естественно, главным преимуществом здесь является возможность применения данного ресурса не только в одном конкретном узле, а и в других узлах и без особых ограничений со стороны особого управляющего узла — диспетчера, как если бы это было в централизованной системе. С другой стороны, сети, использующие принцип централизованного управления, не являются надежными из-за очень больших последствий в случае выхода из строя управляющего узла. Конечно, такая ситуация, в случае использования алгоритмов распределенного контроля и управления, может возникнуть с гораздо меньшей вероятностью. Например, система CSMA/CD (Cambridge Model Distributed System) [52, p.183-189] не

парализуется в случае выхода из строя некоторого количества узлов, а продолжает свое функционирование с естественной деградацией эффективности и производительности.

Безусловно, повышение надежности управления и контроля в распределенных системах является главной проблемой. С нашей точки зрения в перспективе необходимо двигаться к гибким алгоритмам смешанного плана, где, в случае возникновения вышеуказанной ситуации, система могла бы перестраиваться, самовосстанавливаться, изменять стратегию управления, принимать другую форму, если это будет выгодно с точки зрения *самой* системы и ее функционирования.

3.6.2. Коммуникационные аспекты управления РСОД

При выполнении функций управления распределенные системы основываются на том факте, что разделенные в пространстве узлы должны обмениваться информацией между собой. Поэтому далее более подробно рассмотрены проблемы, возникающие при проектировании коммуникационных механизмов для распределенных систем.

* *Уровень сообщений (message passing)*

На низшем уровне все процессы "общаются" между собой при помощи обмена сообщениями — некоторой порции информации определенных размеров. Эти сообщения рассылаются по сети в виде пакетов. Процесс, желающий передать информацию, посылает сообщение. Процесс — приемник получает сообщение, выполняет прием и через некоторое время посылает ответ процессу, передавшему ему информацию. Таким образом, распределенная система на этом уровне предполагает наличие некоторого количества примитивных операций: посылка, прием, отзыв (ответ, отклик).

Существует два основных типа операторов приема-посылки с многочисленными вариантами — это *синхронные и асинхронные* сообщения. Иногда они называются блочными и безблочными примитивами. При использовании синхронных операций посылки источник сообщения приостанавливает свое выполнение до тех пор, пока не получит ответа от получателя (рис.3.7).

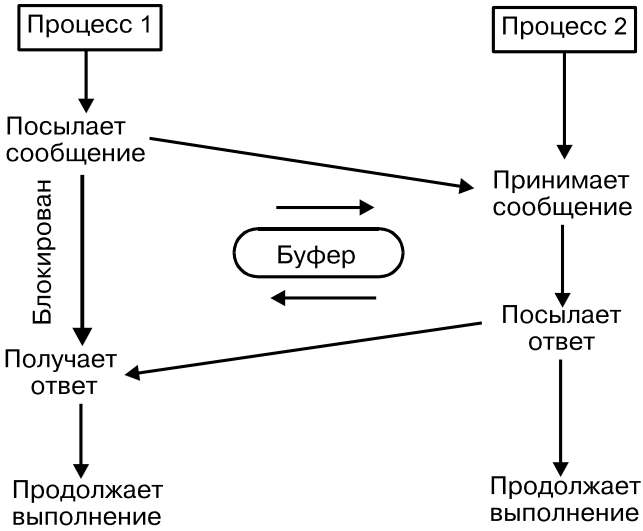

Рис.3.7. Синхронная операция пересылки

Пример: Процесс 1 блокирован после посылки сообщения Процессу 2 до получения ответа от Процесса 2.

При использовании асинхронных операций источник не прерывает свое выполнение, ожидая ответа от получателя, но, получая ответ на более поздней стадии, выполняет при этом специальную операцию приема (рис. 3.8).

Сообщения пересылаются и хранятся с помощью системного буфера. При использовании асинхронного метода каждый процесс может иметь дело со многими сообщениями в одно и то же время. Однако здесь могут возникнуть сложности в связи с ограниченностью системного буфера как ресурса, и, в случае его полной занятости, он не сможет применяться в данный момент времени. Кроме этого, при таком способе буферизации в коде обменов в системе могут возникнуть тупиковые ситуации. Поэтому при асинхронном методе в системе необходимо выполнять нетривиальный алгоритм управления буфером. Такой алгоритм выглядит гораздо понятнее и проще в синхронном методе, где подобной ситуации в принципе быть не может.

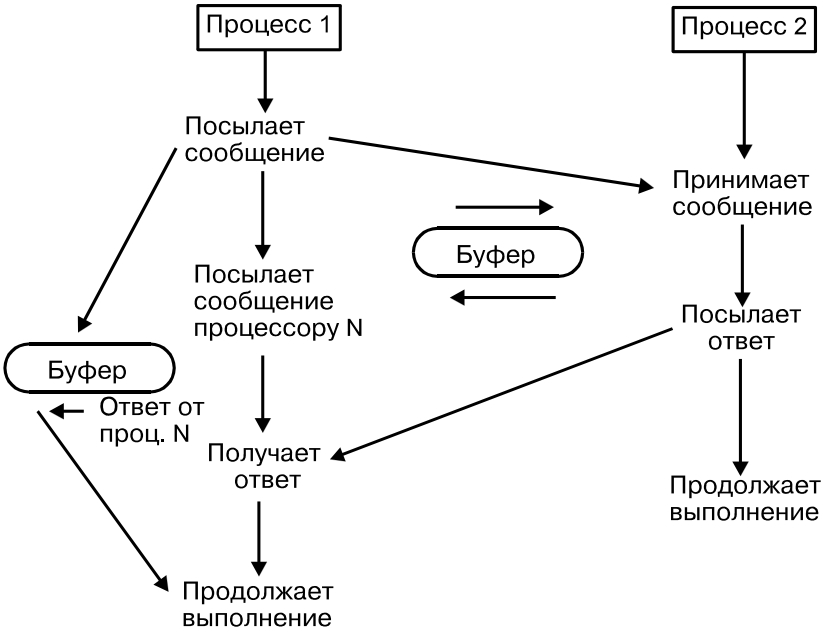

Рис.3.8. Асинхронная операция пересылки

Пример: Процесс 1 продолжает выполняться после посылки сообщения Процессу 2, а ответ от Процесса N получает позже.

Синхронный метод особенно хорошо подходит для модели "клиент — сервер" (client/server). Клиент, которому необходим доступ к серверу для выполнения некоторых действий, посылает ему сообщение. После этого сервер имеет дело только с данным пользователем, и в течение времени, необходимого этому пользователю для взаимодействия с сервером, другие клиенты не могут к нему обратиться. Таким образом, сервер можно описать как бесконечный цикл обращений к нему, состоящий из ожидания запроса, обработки запроса и посылки результата (ответа). При получении обслуживания сервером клиент блокирует свою дальнейшую работу, а сервер может приостановить свою деятельность в случае отсутствия запросов к нему (рис.3.9).

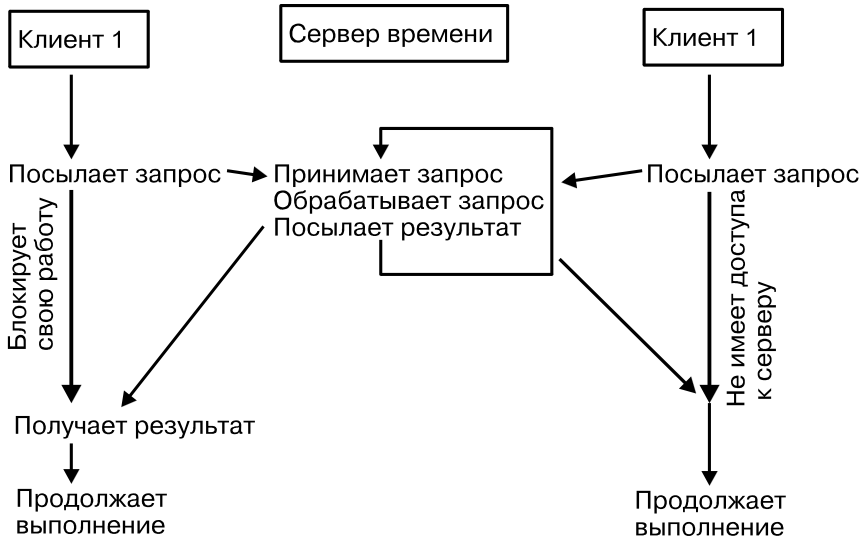

Рис.3.9. Модель "Клиент — сервер"

Пример: Клиент1 блокируется до тех пор, пока сервер не ответит ему. Сервер времени находится в цикле ожидания до получения запроса на чтение часов.

Синхронные примитивы создают дополнительную возможность — возможность обмениваться информацией. Когда процесс вызывает такой примитив, то сам процесс приостанавливается и остается в таком состоянии до тех пор, пока примитив не закончит свое выполнение. Это блокирующее свойство синхронных примитивов может быть использовано для синхронизации подобно семафорам (рис.3.10) [108].

Другой примитив приема/посылки — оператор селективного приема, позволяет пользователю выбирать процесс или группу процессов, с которыми необходимо связаться. Оператор условной посылки будет завершен немедленно — без посылки сообщения, если получатель не блокирован в данный момент времени для приема сообщения. С этой точки зрения связь между двумя процессами может быть вообще не установлена. Именно для предотвращения этого, а также ради большей надежности распределенные системы часто используют уровень синхронных примитивов.

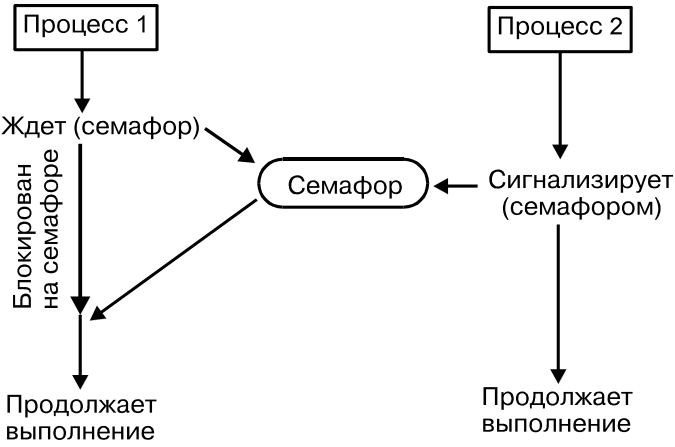

Рис.3.10. Процесс синхронизации с использованием семафора

Хотя модель "клиент — сервер" и обеспечивает удобный способ структуризации распределенной системы, однако она обладает трудноисправимым недостатком — для операций ввода/вывода особенно важны коммуникационные аспекты передачи/приема информации через узлы ВС.

* *Технология функционирования модели клиент — сервер*

Клиент—сервер — это модель взаимодействия компьютеров в сети.

Как правило, компьютеры не являются равноправными. Каждый из них имеет свое, отличное от других, назначение, играет свою роль. Некоторые компьютеры в сети владеют и распоряжаются информационно-вычислительными ресурсами, такими как: процессоры, файловая система, почтовая служба, служба печати, база данных. Другие же компьютеры имеют возможность обращаться к этим службам, пользуясь услугами первых. Компьютер, управляющий тем или иным ресурсом, принято называть сервером этого ресурса, а компьютер, желающий им воспользоваться — клиентом. Конкретный сервер определяется видом ресурса, которым он владеет. Так, если ресурсом являются базы данных, то речь идет о сервере баз данных, назначение которого — обслуживать запросы клиентов, связанные с обработкой данных; если ресурс — это файловая система, то говорят о файловом сервере, или файл — сервере, и т. д.

В сети один и тот же компьютер может выполнять роль как клиента, так и сервера. Например, в информационной системе, включающей персональные компьютеры, большую ЭВМ и мини — компьютер под управлением UNIX, последний может выступать как в качестве сервера базы данных, обслуживая

запросы от клиентов — персональных компьютеров, так и в качестве клиента, направляя запросы большой ЭВМ.

Этот же принцип распространяется и на взаимодействие программ. Если одна из них выполняет некоторые функции, предоставляя другим соответствующий набор услуг, то такая программа выступает в качестве сервера. Программы, которые пользуются этими услугами, принято называть клиентами. Так, ядро реляционной SQL ориентированной СУБД часто называют сервером базы данных или SQL сервером, а программу, обращающуюся к нему за услугами по обработке данных, SQL — клиентом.

Первоначально СУБД имели централизованную архитектуру. В ней сама СУБД и прикладные программы, которые работали с базами данных, функционировали на центральном компьютере (большая ЭВМ или мини—компьютер). Там же располагались базы данных. К центральному компьютеру были подключены терминалы, выступавшие в качестве рабочих мест пользователей. Все процессы, связанные с обработкой данных, как то: поддержка ввода, осуществляемого пользователем, формирование, оптимизация и выполнение запросов, обмен с устройствами внешней памяти и т.д., выполнялись на центральном компьютере, что предъявляло жесткие требования к его производительности. Особенности СУБД первого поколения напрямую связаны с архитектурой систем больших ЭВМ и мини — компьютеров и адекватно отражают все их преимущества и недостатки. Однако нас больше интересует современное состояние многопользовательских СУБД, для которых архитектура клиент — сервер стала фактическим стандартом. Для более четкого представления о ее особенностях необходимо рассмотреть несколько моделей технологии клиент —сервер.

Если предполагается, что проектируемая информационная система (ИС) будет иметь технологию клиент —сервер, то это означает, что прикладные программы, реализованные в ее рамках, будут иметь распределенный характер. Иными словами, часть функций прикладной программы (или, проще, приложения) будет реализована в программе — клиенте, другая — в программе — сервере, причем для их взаимодействия будет определен некоторый протокол. Основной принцип технологии клиент — сервер заключается в разделении функций стандартного интерактивного приложения на четыре группы, имеющие различную природу.

Первая группа — это функции ввода и отображения данных. Вторая группа объединяет чисто прикладные функции, характерные для данной предметной области (например, для банковской системы — открытие счета, перевод денег с одного счета на другой и т.д.). К третьей группе относятся фундаментальные функции хранения и управления информационными ресурсами (базами данных, файловыми системами и т.д.). Наконец, функции четвертой группы — это служебные функции (играющие роль связок между функциями первых трех групп).

В соответствии с этим в любом приложении выделяются следующие логические компоненты :

* компонент представления, реализующий функции первой группы;

* прикладной компонент, поддерживающий функции второй группы;
* компонент доступа к информационным ресурсам, поддерживающий функции третьей группы, а также вводятся и уточняются соглашения о способах их взаимодействия (протокол взаимодействия).

Архитектура клиент — сервер определяется четырьмя факторами.

Во-первых, тем, в какие виды программного обеспечения интегрированы каждый из этих компонентов. Во-вторых, тем, какие механизмы программного обеспечения используются для реализации функций всех трех групп. В-третьих, как логические компоненты распределяются между компьютерами в сети. В-четвертых, какие механизмы используются для связи компонентов между собой.

Выделяются четыре подхода, реализованные в моделях:

* модель файлового сервера (File Server — FS);
* модель доступа к удаленным данным (Remoute Data Access — RDA);
* модель севера базы данных (DataBase Server — DBS);
* модель сервера приложений (Application Server — AS).

FS — модель является базовой для локальных сетей персональных компьютеров. Не так давно она была исключительно популярной среди отечественных разработчиков, использовавших такие системы, как FoxPro, Clipper, Clarion, Paradox и т.д. Суть модели проста и всем известна. Один из компьютеров в сети считается файловым сервером и предоставляет услуги по обработке файлов другим компьютерам. Файловый сервер работает под управлением сетевой операционной системы (например, Novell NetWare) и играет роль компонента доступа к информационным ресурсам (то есть к файлам). На других компьютерах в сети функционирует приложение, в кодах которого совмещены компонент представления и прикладной компонент. Протокол обмена представляет собой набор низкоуровневых вызовов, обеспечивающих приложению доступ к файловой системе на файл — сервере.

Рассмотрим работу системы, где используется СУБД в многопользовательском режиме. В таких системах на нескольких персональных компьютерах выполняется как прикладная программа, так и копия СУБД, а базы данных содержатся в разделяемых файлах, которые находятся на файловом сервере. Когда прикладная программа обращается к базе данных, СУБД направляет запрос на файловый сервер. В этом запросе указаны файлы, где находятся запрашиваемые данные. В ответ на запрос файловый сервер направляет по сети требуемый блок данных. СУБД, получив его, выполняет над данными действия, которые были декларированы в прикладной программе.

К технологическим недостаткам модели файл — сервер относят высокий сетевой трафик (передача множества файлов, необходимых приложению), узкий спектр операций манипуляции с данными (данные — это файлы), отсутствие адекватных средств безопасности доступа к данным (защита только на уровне файловой системы) и т.д. Собственно, перечисленное не есть недостатки, но — следствие внутренне присущих FS — модели ограничений, определяемых ее характером. Недоразумения возникают, когда FS — модель используют не по

назначению, например, пытаются интерпретировать как модель сервера базы данных. Место FS — модели в иерархии моделей клиент-сервер — это место модели файлового сервера, и ничего более. Именно поэтому обречены на провал попытки создания на основе FS — модели крупных корпоративных систем — попытки, которые предпринимались в недавнем прошлом и нередко предпринимаются сейчас.

Более технологичная RDA — модель существенно отличается от FS — модели характером компонента доступа к информационным ресурсам.

Это, как правило, SQL — сервер. В RDA-модели коды компонента представления и прикладного компонента совмещены и выполняются на компьютере-клиенте. Последний поддерживает как функции ввода и отображения данных, так и чисто прикладные функции. Доступ к информационным ресурсам обеспечивается либо операторами специального языка (языка SQL, например, если речь идет о базах данных), либо вызовами функций специальной библиотеки (если имеется соответствующий интерфейс прикладного программирования — API).

Клиент направляет запросы к информационным ресурсам (например, к базам данных) по сети удаленному компьютеру. На нем функционирует ядро СУБД, которое обрабатывает запросы, выполняя предписанные в них действия, и возвращает клиенту результат, оформленный как блок данных. При этом инициатором манипуляций с данными выступают программы, выполняющиеся на компьютерах — клиентах, в то время как ядру СУБД отводится пассивная роль — обслуживание запросов и обработка данных.

RDA — модель избавляет от недостатков, присущих как системам с централизованной архитектурой, так и системам с файловым сервером.

Прежде всего перенос компонента представления и прикладного компонента на компьютеры — клиенты существенно разгружает сервер БД, сводя к минимуму общее число процессов операционной системы. Сервер БД освобождается от несвойственных ему функций; процессор или процессоры сервера целиком загружаются операциями обработки данных, запросов и транзакций. Это становится возможным благодаря отказу от терминалов и оснащению рабочих мест компьютерами, которые обладают собственными локальными вычислительными ресурсами, полностью используемыми программами переднего плана. С другой стороны, резко уменьшается загрузка сети, так как по ней передаются от клиента к серверу не запросы на ввод/вывод (как в системах с файловым сервером), а запросы на языке SQL, их объем существенно меньше.

Основное достоинство RDA — модели — унификация интерфейса клиент — сервер в виде языка SQL. Действительно, взаимодействие прикладного компонента с ядром СУБД невозможно без стандартизованного средства общения. Запросы, направляемые программой ядру, должны быть понятны обоим. Для этого их следует сформулировать на специальном языке. Но в СУБД уже существует язык SQL, о котором уже шла речь . Поэтому целесообразно использовать его не только в качестве средства доступа к данным, но и стандарта общения клиента и сервера.

Такое общение можно сравнить с беседой нескольких человек, когда один отвечает на вопросы остальных (вопросы задаются одновременно). Причем делает это он так быстро, что время ожидания ответа приближается к нулю. Высокая скорость общения достигается прежде всего благодаря четкой формулировке вопроса, когда спрашивающему и отвечающему не нужно дополнительных консультаций по сути вопроса. Беседующие обмениваются несколькими короткими однозначными фразами, им ничего не нужно уточнять.

К сожалению RDA — модель не лишена ряда недостатков. Во-первых, взаимодействие клиента и сервера посредством SQL — запросов существенно загружает сеть. Во-вторых, удовлетворительное администрирование приложений в RDA — модели практически невозможно из-за совмещения в одной программе различных по своей природе функций (функции представления и прикладные).

Наряду с RDA — моделью все большую популярность приобретает перспективная DBS — модель. Последняя реализована в некоторых реляционных СУБД (Informix, Sybase, Microsoft SQL Server, Oracle). Ее основу составляет механизм хранимых процедур — средство программирования SQL — сервера. Процедуры хранятся в словаре базы данных, разделяются между несколькими клиентами и выполняются на том же компьютере, где функционирует SQL — сервер. Язык, на котором разрабатываются хранимые процедуры, представляет собой процедурное расширение языка запросов SQL и уникален для каждой конкретной СУБД.

В DBS — модели компонент представления выполняется на компьютере — клиенте, в то время как прикладной компонент оформлен как набор хранимых процедур и функционирует на компьютере — сервере БД. Там же выполняется компонент доступа к данным, то есть ядро СУБД. Достоинства DBS — модели очевидны : это и возможность централизованного администрирования прикладных функций, и снижение трафика (вместо SQL — запросов по сети направляются вызовы хранимых процедур), и возможность разделения процедуры между несколькими приложениями, и экономия ресурсов компьютера за счет использования единожды созданного плана выполнения процедуры. К недостаткам модели можно отнести ограниченность средств, используемых для написания хранимых процедур, которые представляют собой разнообразные процедурные расширения SQL, не выдерживающие сравнения по изобразительным средствам и функциональным возможностям с языками третьего поколения, такими как C или Pascal. Сфера их использования ограничена конкретной СУБД, в большинстве СУБД отсутствуют возможности отладки и тестирования разработанных хранимых процедур.

На практике часто используется смешанные модели, когда поддержка целостности базы данных и некоторые простейшие прикладные функции поддерживаются хранимыми процедурами (DBS — модель), а более сложные функции реализуются непосредственно в прикладной программе, которая выполняется на компьютере — клиенте (RDA — модель). Так или иначе

современные многопользовательские СУБД опираются на RDA — и DBS — модели и при создании ИС, предполагающих использование только СУБД, выбирают одну из этих двух моделей либо их разумное сочетание.

В AS — модели процесс, выполняющийся на компьютере—клиенте, отвечает, как обычно, за интерфейс с пользователем (то есть осуществляет функции первой группы). Обращаясь за выполнением услуг к прикладному компоненту, этот процесс играет роль клиента приложения (Application Client — AC). Прикладной компонент реализован как группа процессов, выполняющих прикладные функции, и называется сервером приложения (Application Server — AS). Все операции над информационными ресурсами выполняются соответствующим компонентом, по отношению к которому AS играет роль клиента. Из прикладных компонентов доступны ресурсы различных типов — базы данных, очереди, почтовые службы и др.

Биррел и Нельсон в 1984 году [54] предложили концепцию, которая позволяет программам вызывать процедуры, находящиеся на других машинах. Когда программа, находящаяся на машине А, вызывает процедуру, находящуюся на машине В, то вызывающая программа приостанавливается, и происходит выполнение вызванной процедуры на машине В. Результаты и параметры пересылаются от одной программы к другой и обратно. При этом весь механизм не заметен пользователю. Этот метод получил название "удаленного вызова процедур" (remote procedure call — RPC) [57].

Проблем здесь множество, несмотря на простоту идеи. Вызывающая программа и вызванная процедура выполняются на разных машинах в разном адресном пространстве, возможно с разными ресурсами, с разными операционными средами, поэтому многие вещи здесь будут усложнены. Поэтому при передаче параметров и результатов необходимо учитывать особенности каждой машины. Наконец, обе машины могут прекратить нормальное функционирование и т.п. Тем не менее этот метод широко используется многими распределенными системами (рис.3.11).

Существует два различных аспекта реализации метода удаленного вызова процедур. Во-первых, когда вызванная процедура начинает выполняться, вызывающий узел или система должны знать, в каком узле на текущее время выполняется эта процедура. Во-вторых, упомянутые два узла должны связываться между собой для обмена параметрами и результатами. Все это должно происходить скрытно от пользователя. На рис.3.12. показан примерный вид реализации этого метода, наиболее часто используемого в распределенных системах.

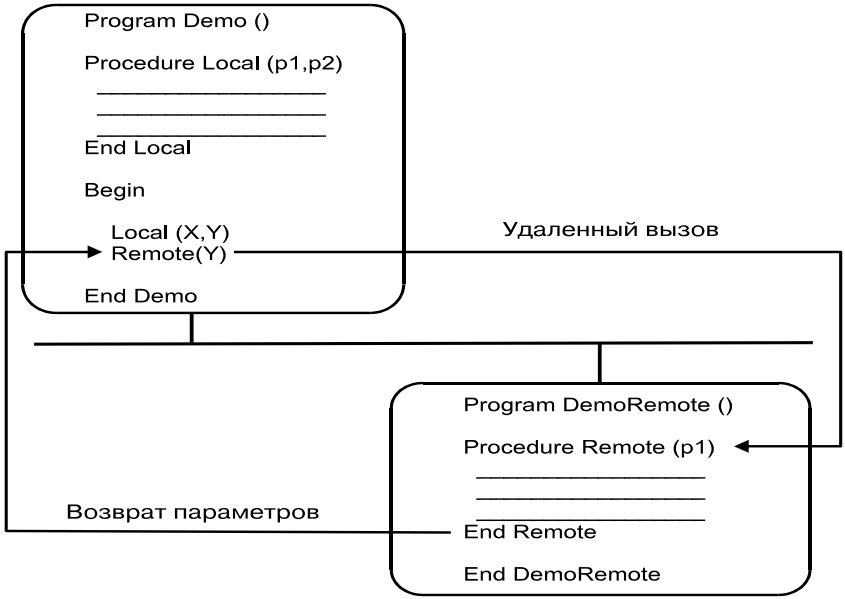

Рис.3.11. Удаленный вызов процедуры

Каждый отдельный вызов содержит некоторое количество компонент. Обязательно существует запрашивающая сторона (обычно это пользователь), система сигналов, с помощью которой производится вызов, и запрашиваемая сторона (обычно сервер), код которого является вызываемым. Причем все это описано на языке высокого уровня, не обеспечивающем дополнительных возможностей. Другие понятия этого метода связаны уже с особенностями взаимодействия между обеими сторонами и временем выполнения такой вызванной процедуры, которые актуальны для всех узлов системы.

Поэтому важное место здесь занимает функция упаковки и распаковки аргументов в сообщение и из сообщения. Вдобавок, здесь возможно предусмотреть обработку самих параметров в случае, если они содержат служебную информацию для всего механизма вызова.

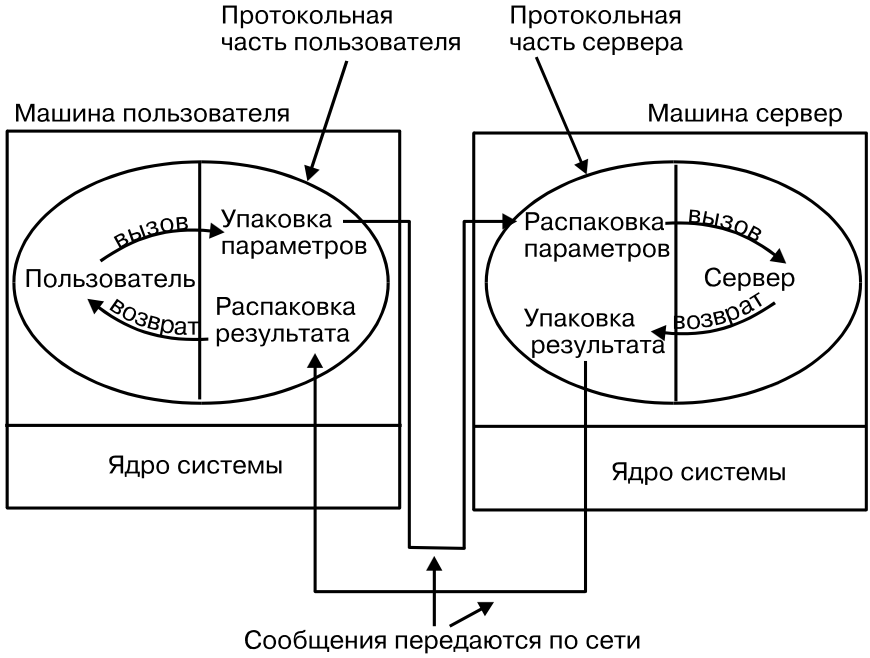

Рис.3.12. Вызовы и передача сообщений при использовании метода удаленного вызова процедур

Пример: Каждый эллипс представляет собой процесс (выделенную часть называют протоколом метода удаленного вызова процедур)

Необходимо заметить, что формализация функции, продуцирующей вышеупомянутые коды для обеих сторон, является открытым вопросом. На практике чаще всего это выполняется самим программистом, хотя упаковка и распаковка уже производятся самой системой.

Процесс удаленного вызова процедуры выглядит следующим образом:

- Вызывающая сторона производит вполне конкретный вызов процедуры, используя посылку.
- Эта посылка, содержащая параметры, упаковывается в пакет. Адрес узла, где будет выполняться необходимая процедура, и идентификатор процедуры включаются в сообщение.
- Сообщение посылается в соответствующий узел.
- Производятся действия по распаковке пакета с параметрами и идентификация процедуры.

- Производится обычный вызов соответствующей процедуры и ее выполнение.
- Результаты выполнения процедуры аналогичным образом посылаются обратно для пользователя, и в этот момент происходит обычный возврат из процедуры.

Невыясненным остается то обстоятельство, откуда пользователь будет знать адрес узла, в котором будет выполняться необходимая ему процедура. Возможно несколько решений данной проблемы:

* адрес узла может быть записан в вызывающую посылку при ее формировании. Это очень гибкий механизм.
* перед вызовом узел пользователя делает запрос в систему о местонахождении соответствующей процедуры и полученные данные включает в посылку. Это довольно громоздкий механизм, учитывая возможность поиска по всей сети, а иногда и между взаимосвязанными сетями.
* сохранение в сети таблицы, содержащей номера всех узлов и соответствующие процедуры, закрепленные за ними.
* наличие в каждом узле "образа ресурсов и активизированных процессов в сети", которые все время должны обновляться.

При последних двух способах узлы, желающие обратиться к другим узлам за необходимыми им процедурами или ресурсами, могут, обратясь к этой таблице, получить адрес этих узлов, если там находятся нужные им процедуры. Впервые это было предложено Кедаром [66], который ратовал за создание специальной распределенной базы данных, особенно если это касалось больших сетевых систем. Таким образом, вызванная процедура также легко сможет передать обратно результаты своего выполнения, если вызывающая сторона впишет в соответствующее поле базы данных координаты, куда необходимо возвратить данные. Процесс вычисления вызванной процедурой местоположения вызывающего процесса и посылка туда результатов своей работы называется связыванием. Тут можно провести некоторую аналогию между связыванием и процессом линкования объектного модуля, когда помимо всего согласовываются и связываются метки.

Нужно также заметить, что нет необходимости широкой модификации языка ни в одном из указанных трех способах для того, чтобы обеспечить механизм вызова удаленной процедуры. Весь этот механизм может быть реализован с помощью создания специальной библиотеки программ или утилит.

* *Проблемы реализации метода удаленного вызова процедур*

Существует целый ряд различий между удаленным и близким вызовом процедур. Если такая связь производится между машинами разных архитектур, возникает проблема представления данных, т.к. машины могут работать со словами различной длины. Одним из решений этой проблемы является наличие в методе

удаленного вызова процедур механизма преобразования всех данных в некоторый заранее оговоренный формат перед сеансом связи.

Другой проблемой является перевод ссылок (указателей). При отсутствии разделяемого адресного пространства данный метод не позволяет ссылкам быть посланными по всей сети. Поэтому в рамках этого метода не представляется возможным передавать параметры в виде ссылок .

Более серьезной проблемой, нежели предыдущие, является возможность возникновения аварийной ситуации сразу в обеих процедурах ── вызывающей и вызванной, с другой стороны, рандеву не произойдет, если хотя бы одна сторона не вышла на связь, а это значит, что оставшийся процесс может оказаться в дедлоке, либо в дедлоке окажутся обе стороны.

В случае нарушения связи вызывающая сторона оказывается в затруднительном положении, т.к. для нее остается неизвестным, когда произошло нарушение всего механизма и почему. Тут можно выделить три момента:

* узел, в котором находится вызванная процедура, аварийно завершил свой процесс до получения вызова;
* узел, содержащий вызванную процедуру, аварийно завершил процесс выполнения этой процедуры;
* после корректного завершения вызванной процедуры до возвращения результатов произошел сбой в этом узле.

Нужно отметить также возможность аварийного завершения вызывающего процесса, хотя эта проблема менее серьезная. Учитывая то, что вызывающая сторона, в случае нарушения сеанса связи, может не узнать, какой из указанных трех случаев имел место, в системе необходимо предусмотреть обеспечение минимальной информации о том, что происходит в системе. Информация периодически сохраняется системой и используется в случае необходимости восстановления работоспособности системы или исключения состояния "бесконечного откладывания" некоторых процессов или посылки информационных сообщений активизированных процессам. Проблема осложняется тем фактом, что из-за коммуникационных сбоев и попыток их исправления процедура может вызываться до тех пор, пока весь сеанс связи не закончится корректно. Различные реализации метода удаленного вызова процедур определяют происходящие в системе события и принимают необходимые действия, но все-таки еще не на удовлетворительном уровне. В настоящее время предложены два общих подхода к ликвидации последствий вышеуказанных сбоев:

Подход "последний из многих" (last-of-many). Вызванная процедура, как было сказано раньше, может выполняться много раз. Результаты возвращаются только после последнего корректного выполнения процедуры. Этот подход требует выполнения тождественной процедуры для того, чтобы осуществить необходимую проверку.

"*Жесткий" подход* (at-most-once). Он заключается в том, что при возникновении аварийной ситуации система принудительно запрещает сеанс связи до лучших времен.

В заключение следует отметить, что существуют не только многие варианты операций посылки/получения, но также различные типы специфических операций внутри самих процедур, связанные с применением последних в данном методе. Асинхронный механизм удаленного вызова процедур возможен в случае параллельного выполнения вызывающего процесса и вызванной стороны, при этом задача сбора результатов решается вызывающей стороной с помощью выполнения так называемых разделителей.

3.6.3. Языковые аспекты взаимодействия пользователя с РСОД

Цель написания РОС заключается в обеспечении поддержки выполнения или "развития" прикладных программ в распределенной среде. Эта поддержка может быть предложена на разных уровнях. Многие структурированные языки программирования были специально дополнены для написания программ, работающих в распределенной среде, например DP [55] и CLU/ARGUS [109]. Язык программирования может значительно облегчить написание "распределенных" программ различными способами. Это прежде всего определяется наличием в языке средств абстракции данных, разделяемого компилирования и т.д., которые и используются при написании программ и преобразовании их для работы в распределенной среде. Основные требования к языку:

* язык должен поддерживать метод удаленного вызова процедур;
* язык должен обеспечить гибкое написание программ в случае их применимости в разных системах;
* язык должен поддерживать возможность взаимодействия параллельно выполняющихся процессов, как это сделано, например, в языке ADA, OKKAM, CC++, HPF (High Performance Fortran) и Viena Fortran.

Следует заметить некоторую аналогию в написании ОС для распределенной вычислительной среды и языка для нее же.

3.6.4. Вопросы применения РОС

Сфера применения РСОД в значительной мере влияет на проектирование системы и на реализацию рассмотренных аспектов. Часто выигрывая в одном моменте применения, мы проигрываем в другом. Поэтому для конкретной области применения следует выбрать тот критерий (или критерии), где вероятность сбоев была бы минимальной, учитывая вероятность этого критерия. К проблемам сбоев и аварий в системе нужно подходить комплексно, т.к. в такой среде вопросы взаимосвязи и взаимодействия являются основополагающими.

В заключение, в качестве примера, рассмотрим некоторые распределенные ОС.

ОС AMOEBA [144] спроектирована для объединения независимых компьютеров в систему, кажущуюся пользователям отдельной системой с разделением времени. В общем пользователи осведомлены, где протекают их процессы, но они не знают, где хранится необходимая им информация и данные, или как происходит сохранение и копирование этой информации для ее применения

приложениями. Пользователи, имеющие опыт работы в параллельной среде, могут эксплуатировать эту систему с распределением работ между машинами. AMOEBA базируется на "микроядрах" (microkernel), каждое из которых может обрабатывать низкоуровневые процессы: примитивами управления памятью, коммуникациями сообщений и вводом/выводом. Файловая система и даже операционная система (не глобальная) могут рассматриваться здесь как пользовательские процессы. Такое разделение функций хорошо поддерживается "микроядерной" структурой системы.

AMOEBA имеет единый механизм именования и защиты всех объектов. Объекты, используемые системой для своего функционирования, являются относительно самостоятельными структурами и имеют все необходимое для обслуживания тех процессов, которые порождены ими. Эти объекты имеют криптографическую защиту против сбоев, поэтому суммарный результат такой защиты обеспечивает относительную безопасность всей системы.

В AMOEBA реализуется два коммуникационных метода ── метод удаленного вызова процедур для связи между двумя вычислительными узлами и надежный групповой метод коммуникации. При этом реализация метода удаленного вызова процедур гарантируется при помощи "жесткого" подхода (описан выше). Групповая коммуникация базируется на механизме оповещения в системе, обеспеченного последовательным алгоритмом. Оба метода работают в рамках FLIP — протокола и для пользователя скрытно объединены и не разделяемы.

Файловая система AMOEBA состоит из трех серверов: для хранения файлов, именования и служебных функций. Первый сервер поддерживает неизменяемые файлы, хранящиеся на диске или в СОЗУ. Другой сервер довольно устойчив к сбоям и производит необходимые действия с ASCII — строками, связанными с именованием объектов системы. Третий сервер часто используется для служебных функций и процедур, например, для коммуникационных нужд.

Другой РОС, находящей все большее применение, является РОС PVM. Системное программное обеспечение PVM (Parallel Virtual Machine) [89] позволяет объединить вычислительные узлы в вычислительную среду, в которой связанные или независимые программы, представленные в стандартизованном виде, могут выполняться, эффективно используя имеющуюся аппаратную поддержку. PVM объединяет набор отдельных гетерогенных вычислительных машин (узлов) в виде виртуальной параллельной вычислительной машины. PVM оперирует процедурами обработки сообщений, передачи данных, планирования выполнения задач в сети несовместимых компьютерных архитектур (DEC Alfa, 80386/486 PC, Cray-2, CM-5, IntelParagon, SPARC, C-90, IBM 370, HP-9000, NeXT, Sun 3, Cray S-MP и др.)

Вычислительная модель PVM — это общее понятие, которое объединяет широкий набор программных приложений. Программный интерфейс максимально стандартизирован, что позволяет интуитивно создавать простые программные структуры для параллельного исполнения. Пользователь оформляет свои приложения как совокупность взаимодействующих задач. Задача получает ресурсы PVM посредством использования библиотек стандартных интерфейсных подпрограмм. Эти подпрограммы позволяют инициализировать или уничтожать

задачи, а также выполнять передачу информации между задачами и их синхронизацию. Примитивы передачи сообщений в PVM ориентированы на осуществление связи между гетерогенными узлами с помощью строго типизированных конструкций данных для буферизации и передачи, которые обычно используются для передачи информации в РСОД.

Задачи в PVM могут осуществлять приоритетное управление и влиять на структуру виртуальной машины. Другими словами, в любой точке выполнения приложения, любая активная задача может запустить/ приостановить другие задачи или добавить/удалить компьютеры, используемые для этого приложения в виртуальной машине. Любой процесс может связываться и/или синхронизироваться с другим. Любой вычислительный узел или вычислительная структура (клайстер) может быть подключена к PVM системе с помощью использования конструкций высокоуровневого языка управления.

Для упрощения подготовки программ для PVM систем разработана графическая интерфейсная среда HeNCE (Heterogeneous Networc Computing Environment), которая облегчает создание, компилирование, выполнение и отладку параллельных программ в среде PVM. Причем пользователь может и не иметь навыков параллельного программирования.

Имеются и другие системы, подобные PVM, помогающие программистам более эффективно использовать РСОД. Среди наиболее известных следует отметить p4 [53], Express [81], Linda [63].

P4 — это библиотека макросов и подпрограмм разработанных в Apgonne National Laboratory для программирования программ, исполняемых в распределенных системах. Система **p4** поддерживает модели ВС с разделяемой и распределенной памятью. Для организации параллельных вычислений в модели ВС с разделяемой памятью **p4** обеспечивает доступ к ней множества виртуальных мониторов как множество отдельных примитивов. Для работы в ВС с распределенной памятью **p4** обеспечивает стандартизированный обмен сообщениями между инициализированными процессами в соответствии с содержимым специального текстового файла, описывающего структуру этих процессов. Управление процессом в **p4** основывается на файле конфигурации, который точно определяет управляющий блок и исполняемый файл (процесс), который будет запущен на каждой машине, а также число процессов, которые будут запущены на каждом узле, и другую дополнительную информацию. Следует выделить две наиболее важные особенности реализации управления в **p4**. Первое — разделение процессов на две категории: процессы "хозяин" и процессы "подчиненные", и применение мультиуровневых иерархий, создаваемых в процессе исполнения программ и определяемых в **p4** как клайстерная модель вычислений. Второе — основной способ создания процессов — статический через файл — конфигурации; создание динамического процесса возможно только процессом, созданным статически, который должен вызвать специальную функцию, порождающую новый процесс на локальной машине. Несмотря на эти ограничения,

различные приложения, при выполнении определенных соглашений, могут быть выполнены в системе **p4**.

Передача сообщений в системе **p4** достигается использованием традиционных примитивов передачи сообщений, формат сообщений сходен с форматами других систем передачи сообщений. Для семантики используются несколько вариантов, как то: гетерогенный обмен или блочная, или неблочная передача.

Философия организации вычислений в **Express** системе основана на использовании последовательной версии вводимого приложения (программы) с последующим пошаговым развитием этого приложения в параллельно исполняемую оптимизированную версию. Преобразование приложения начинается с использования VTOOL — графической программы, которая позволяют отображать последовательные алгоритмы в динамической форме. Преобразования данных и ссылки на отдельные структуры данных могут быть отображены в определенную демонстрационную графическую структуру алгоритма и могут быть обеспечены детальные описания исходной программы для распараллеливания. Эти действия выполняются программой FTOOL, которая обеспечивает глубинный анализ программы, включая анализ использования переменных, потоковые структуры с выделением потенциального параллелизма. FTOOL может работать с последовательными и параллельными версиями приложений. На третьем этапе используется система ASPAR — это автоматический параллелизатор, который преобразует C и Fortran программы для параллельного или распределенного выполнения с использованием моделей **Express** программирования.

Ядро **Express** системы — это множество библиотек для связи, ввода/вывода и графического интерфейса. Примитивы связи родственны основным примитивам других систем передачи сообщений и включают набор основных глобальных операций и работы с распределенными данными. В **Express** системе расширены подпрограммы ввода — вывода для обеспечения параллельного ввода — вывода, и подпрограммы, обеспечивающие графическое отображение множества конкурирующих процессов. Express также содержит систему NDB — параллельный отладчик, который в основном использует команды "dbx" интерфейса.

Главная концепция в системе **Linda** — это *пространственный кортеж* (*tuple-space*), абстракция, через которую взаимодействуют связанные процессы. Главная идея системы **Linda** была предложена как альтернативная парадигма двух традиционных методов параллельных вычислений, базирущихся на работе с разделяемой памятью и основанных на передаче сообщений в системах с распределенной памятью. Концепция пространственного кортежа — это существенная абстракция распределенной разделяемой памяти с одним важным отличием (пространственный кортеж ассоциативен), и некоторыми второстепенными отличиями (возможно разрушающее и неразрушающее чтение и различное сцепление семантик). Приложения, использующие систему **Linda,** погружаются в ВС, реализуя взаимодействие последовательных программ, конструкций, которые манипулируют пространственным кортежем.

С точки зрения приложения **Linda** — это множество конструкций расширения языка программирования, использующихся для облегчения параллельного программирования. Это обеспечивает для пользователя абстракцию разделяемой памяти для связи процессов без дополнительных требований к аппаратуре физического разделения памяти.

Linda система обычно использует специальное программное обеспечение, которое поддерживает пространство кортежей и операции с ними. В зависимости от оборудования (мультипроцессорная система с разделяемой памятью, параллельные компьютеры, сети рабочих станций, и т.д.) механизм пространственного кортежа применяется с использованием различных способов и с различными степенями эффективности. Недавно была предложена система близкая к проекту **Linda**. Эта система, названная **"Pirhana"**, предлагает систему распределения конкурирующих вычислений: вычислительные ресурсы (представлены как активные агенты) захватывают вычислительные задачи из очереди задач с учетом их доступности и наличия необходимых характеристик. Эта система может быть внедрена на инструментальные платформы параллельных вычислений и объявлена как **"Pirhana system"** или **"Pirhana Linda system"**.

3.7. ОПЕРАЦИОННЫЕ СИСТЕМЫ СИСТЕМ МАССОВОГО РАСПАРАЛЛЕЛИВАНИЯ (ОС СМР)

В этом разделе рассматриваются многопроцессорные операционные системы. Выделены основные принципы, в соответствии с которым и строятся многопроцессорные ОС СМР, а также указаны главные проблемы, возникающие при их проектировании и эксплуатации.

3.7.1. Классификация многопроцессорных ОС СМР

В настоящее время существует несколько подходов, используемых при разработке многопроцессорных ОС СМР. Основной проблемой, затрудняющей построение таких систем, является организация одновременной работы нескольких процессоров. Основные функции многопроцессорных ОС СМР состоят в следующем:

Распределение ресурсов системы между процессорами.

Обеспечение механизма защиты памяти.

Реконфигурация системы и перераспределение задач между ресурсами в случае выхода из строя одного из узлов.

Последняя функция особенно важна в управляющих системах реального времени и системах повышенной надежности. Кроме перечисленных функций, многопроцессорная ОС СМР должна обеспечивать эффективное планирование процессов, т.е. определение наиболее целесообразной последовательности погружения процессов в систему на выполнение.

Необходимость выполнения вышеперечисленных функций усложняет процесс разработки таких систем и делает их довольно сложными. Для решения этих задач используются как параллельные средства (аппаратная поддержка специальных команд, используемых при параллельном программировании), так и программное обеспечение, поддерживающее одновременное выполнение нескольких независимых задач.

В настоящее время при разработке многопроцессорных систем используется один из трех подходов:

ОС СМР, построенные по типу "ведущий — ведомый" (Master — Slave).

ОС СМР, построенные по типу "раздельный супервизор" (Separate supervisors)

ОС СМР, построенные по типу "плавающий супервизор" (Floating supervisors)

Рассмотрим первый подход. Данный тип ОС СМР используется для большинства многопроцессорных систем. При таком подходе все процессоры системы делятся на две категории:

Главный процессор — ведущий (Master).

Вспомогательный процессор — ведомый (Slave).

Главный процессор имеет статус выше всех остальных. Выполнение управляющих функций ОС СМР осуществляется на главном процессоре. После запуска системы вспомогательные процессоры обращаются к главному за получением работы, а также за предоставлением ОС СМР программного интерфейса. Заметим, что все сервисные программы ОС СМР должны быть выполняемы на любом процессоре, входящем в систему. На главном процессоре осуществляется планирование процессов во времени, а также распределение их по ресурсам (процессорам) или другим системным ресурсам, т.е. в пространстве.

Основным достоинством данного метода является относительная простота. Данную ОС СМР можно получить сравнительно несложным расширением возможностей многозадачных ОС СМР, используемых в однопроцессорных системах. Добавляются новые возможности, связанные с одновременным выполнением задач. Как достоинство можно отметить также и простоту управления ресурсами, поскольку все функции, связанные с управлением, решаются в одном узле.

Рассмотрим недостатки данного типа ОС СМР. Главной проблемой функционирования таких систем является их относительно низкая надежность. Поскольку управление системой осуществляется одним процессором, т.е. узлом системы, то выход его из строя приводит к приостановке работы всей системы. Эта проблема особенно важна для управляющих систем реального времени, требующих высокой надежности.

В добавление к этому можно говорить также о недостаточной эффективности управления ресурсами, поскольку один главный процессор не сможет обеспечить высокую загрузку всего множества подчиненных процессоров. Исходя из этого, можно заключить, что применение данного метода наиболее целесообразно для систем с небольшим числом процессоров, где высокая степень надежности не является основным критерием.

Наиболее эффективен данный метод в случае приложений, для которых работа, связанная с загрузкой системы, хорошо спланирована заранее. Целесообразно также использовать такую ОС СМР в асимметричных системах, где производительность главного процессора выше производительности вспомогательных процессоров.

Рассмотрим второй тип ОС СМР. Этот тип называется "Разделенный супервизор". Методы, используемые при данном подходе, схожи с методами, используемыми при построении программного обеспечения локальных систем. При запуске многопроцессорной системы каждый процессор начинает обрабатывать копию базового ядра ОС СМР.

Разделение ресурсов в данной ОС происходит на высоком уровне путем разделения файловой системы. Каждый ресурс имеет свою собственную файловую структуру. ОС СМР обеспечивает обмен данными между процессорами. Каждая копия базового ядра ОС СМР, выполняемая в процессорах, имеет свое множество таблиц локальных данных.

Наряду с этим в системе имеются таблицы глобальных данных. Обычно в них хранится краткая информация о занятости того или иного ресурса системы. Доступ к таблицам глобальных данных имеют все ресурсы системы в режиме разделенного времени. В каждый момент времени доступ к этим данным имеет только один ресурс. Способ организации обмена информацией между ресурсами зависит от способа соединения процессоров.

Основным достоинством данного метода является его высокая степень надежности. Поскольку локальное управление каждым ресурсом осуществляется автономно, то сбой в одном из узлов системы не приводит к приостановке работы всей системы. Однако, здесь необходимо отметить то, что ОС СМР такого типа должна прослеживать последовательность решаемых задач, т.е. план работы для каждого процессора, чтобы в случае необходимости иметь возможность продолжить работу системы, переслав задачи, решаемые в других узлах системы в менее загруженные рабочие узлы.

Как достоинство можно отметить и то, что каждый процессор имеет свою файловую систему, т.е. он может иметь свою собственную систему ввода/вывода, а, следовательно, можно строить системы с многочисленными каналами обмена информацией.

К недостаткам данного подхода можно отнести значительные затраты памяти, а также сложность организации взаимодействия между ресурсами. Для решения проблемы с памятью можно использовать метод буферизации. При этом часть программ ОС СМР загружается в процессоры, а остальная часть хранится в памяти, к которой возможен доступ всех ресурсов. Однако, здесь возникает трудность разделения ОС СМР на часто и нечасто используемые фрагменты, поскольку применение тех или иных подпрограмм ОС СМР зависит от конкретных приложений.

Данные ОС СМР наиболее рационально применять в системах, для которых основным показателем является надежность, а также там, где заранее можно определить класс наиболее часто выполняемых задач.

Рассмотрим третий тип ОС СМР — "плавающий супервизор". Данный метод построения можно рассматривать как попытку объединения двух предыдущих типов. Сущность данного подхода заключается в том, что базовое ядро ОС СМР в каждый момент времени выполняется в одном процессоре, однако место его выполнения не является постоянным, т.е. можно говорить о перемещении супервизора в определенном порядке по процессорам, входящим в систему. Количество процессоров, которые при этом могут запросить сервис супервизора, неограниченно.

Основным достоинством данного метода является его гибкость. Этот тип ОС позволяет добиться наибольшей загрузки системы. Конфликт при одновременном запросе сервиса супервизора решается с помощью приоритетов, которые устанавливаются либо статически, либо динамически.

Как недостаток следует отметить сложность построения таких систем, а также возможность конфликтов при обращении к табличным данным. Применять данный тип ОС СМР следует в системах, где одинаково важны как требования к надежности, так и требования к объему памяти, занимаемой ОС СМР. Следует также заметить, что данный тип ОС СМР позволяет добиться наиболее эффективного использования возможностей ресурсов.

На практике редко можно встретить ОС СМР, которая четко вписывалась бы в один из описанных типов. Как правило, разработанные ОС СМР имеют классификационные признаки всех типов.

Примером реализации некоторых функций ОС СМР является MPI [119].

MPI (Message Passing Interface) — стандарт, спецификация которого была закончена в апреле 1994 года. MPI это результат попытки определить синтаксис и семантику ядра библиотеки подпрограмм передачи сообщений, которая была бы полезна широкому кругу пользователей для эффективной организации параллельных вычислений в вычислительных системах массового распараллеливания. Главным преимуществом установленного стандарта передачи сообщений является переносимость. Одним из главных предпосылок развития MPI есть обеспечение продажи MPP систем с точно определенной системой подпрограмм, которые могут быть эффективно использованы пользователем, а в некоторых случаях и обеспечивать их аппаратную поддержку в масштабируемых системах массового параллелизма.

MPI более поздних версий — программное обеспечение для использования в распределенных системах. MPI включает такие понятия как управление процессом, конфигурации (виртуальной) машины и поддержки ввода/вывода. Ожидается, что MPI будет реализована как интерфейс связи, который будет строится на естественных возможностях аппаратных платформ, за исключением операций передачи данных, которые планируется реализовать на аппаратном уровне.

3.7.2. Требования и состав программного обеспечения СМР

В данной разделе мы рассмотрим основное программное обеспечение, необходимое для построения многопроцессорных систем. Вначале остановимся на тех отличиях, которые присущи ПО многопроцессорных систем по сравнению с ПО, используемых в многозадачной однопроцессорной среде. Данные отличия обусловлены двумя главными причинами: во-первых, архитектурными особенностями, присущими многопроцессорной системе, и, во-вторых, особыми проблемами, возникающими для параллельно выполняемых приложений. Следует подчеркнуть, что главная проблема — это параллельное выполнение приложений.

Многозадачная однопроцессорная система может эмулировать многопроцессорную среду, создавая многочисленные "виртуальные процессоры" (virtual processors) для пользователей. Примерами такой системы могут служить ОС Unix, OS/2, WINDOWS. Пользователь данной системы может организовать параллельное выполнение программ, при котором данные с выхода одной программы будут являться входными данными другой программы. Однако заметим, что параллельность выполнения задач в однопроцессорной системе чисто условная. В каждый момент времени, в течение кванта, выполняется какая-либо одна задача, при этом все активные ресурсы предоставляются только этой задаче.

В этом случае можно лишь абстрактно предположить одновременное выполнение программ на "виртуальных процессорах", хотя использование мультипрограммирования и позволяет реализовать в какой-то степени истинное совмещение (рис.3.13).

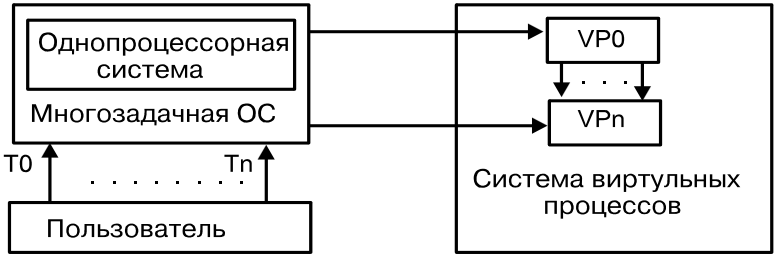

Рис.3.13. Структура многозадачной ОС в однопроцессорной системе

На уровне виртуальных процессоров имеются незначительные различия между многозадачной однопроцессорной средой и многопроцессорной системой. Однако наличие нескольких процессоров усложняет системное ПО, необходимое для такой системы.

Одной из архитектурных особенностей, которая может усложнить ПО многопроцессорных систем, является их неоднородность. Наличие функционально различных процессоров требует разнородного ПО. В данном случае "менеджер ресурсов" ОС СМР должен обеспечить динамическую диспетчеризацию поступающих задач, т.е. спланировать место их выполнения (рис.3.14).

Второй архитектурной особенностью, усложняющей ПО системы, является асимметрия памяти, т.е. различная степень доступа каждого процессора к общим участкам памяти системы, различный объем памяти у каждого процессора.

Как третью особенность архитектуры многопроцессорой системы, особенно влияющей на универсальность ПО, отметим различные способы конфигурации процессоров в системе.

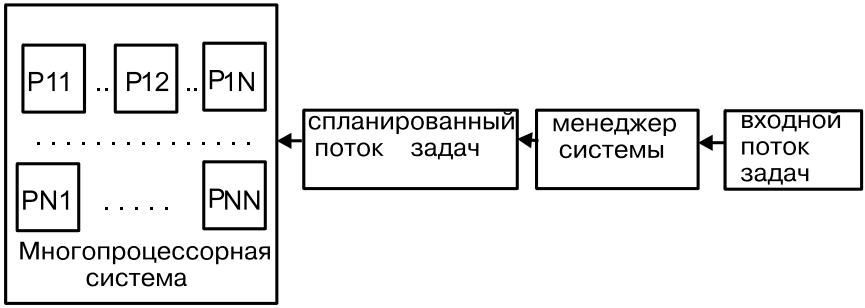

Рис.3.14. Структура многозадачной ОС в многопроцессорной системе

Рассмотрим вторую причину различия многозадачного ПО однопроцессорных и многопроцессорных систем. Это проблемы, связанные с параллельно выполняемыми приложениями. Базовым исполнительным элементом системы является процесс, т.е. независимая часть программы, которая выполняется процессором и использует программные и аппаратные ресурсы системы. В многопроцессорной среде процесс может выполнятся параллельно с другими процессами, приостанавливаясь для взаимодействия с другими процессами.

Исходя из этого, параллельно выполняемые приложения можно определить как программы, имеющие более двух процессов, между которыми возможно взаимодействие.

Параллельные процессы могут быть указаны в программе явно и неявно. Для указания явного параллелизма пользователь должен обладать средствами программирования, которые позволяли бы ему указывать параллельные места в программе. При неявном параллелизме такие места выявляются компилятором. В этом случае компилятор сканирует программу с целью выявления процессов, допускающих распараллеливание, т.е. осуществляется поиск фрагментов программы, которые можно идентифицировать как параллельные подпроцессы.

Для параллельно выполняемых приложений большую роль играют вопросы синхронизации. Эта проблема может послужить причиной приостановки работы системы, если механизмы синхронизации неэффективны и алгоритмы, их использующие, недостаточно хорошо спланированы.

Во многих системах аппаратная поддержка механизма синхронизации отсутствует. Поэтому его поддержку осуществляет программное обеспечение.

Введение аппаратно поддерживаемого механизма синхронизации значительно снижает программные затраты на его обеспечение.

Для упрощения проектирования сложных параллельно выполняемых приложений в настоящее время применяются специальные управляющие программные структуры (Program — Control Structures).

Первый тип такой структуры — это посылочно-основной тип (message-based). При такой организации вычисления представляются многочисленными однородными (гомогенными) процессами, которые выполняются независимо и взаимодействуют через посылки. Минимальный размер процесса зависит от системы. Второй тип управляющих структур — это процедурные (chore) структуры. В этой структуре вся программа разбивается на маленькие разделы. Процесс, который выполняется разделом программы, называется процедурой (chore). Важной характеристикой процедуры в процессе ее выполнения является ее завершенность и отсутствие взаимодействия между ней и другими процессами. При этом система избегает длительных ожиданий. Размер базовых процедур выбирается минимальным. Они имеют относительно малый объем вводимой исходной информации, и количество используемых объектов минимально. Поэтому при их взаимодействии не возникает задержек на ожидание информации от других процедур. Для своего решения одна процедура может потребовать выполнения малого числа добавочных процедур. Примером ОС, построенных с помощью подобных управляющих структур, является BCC - 500, Plurilus.

В качестве примера рассмотрим часть ОС CMP, которая управляет обменом между главной памятью и жестким диском. Основная процедура может включать:

* Дисковые команды, обеспечивающие передачу "страницы данных" из памяти на диск и обратно.
* Команды проверки готовности памяти и диска к обмену данными.

В состав главной процедуры входят также дополнительные процедуры, реализующие действия на низком уровне.

Третий тип управляющих структур, которые называются " производящие системы" (production system), сейчас часто используется в "системах искусственного интеллекта". Необходимость выполнение действий в них выражается в форме причинно-следственной связи. Всякий раз, когда определяется булевское выражение, истинное следствие может быть выполнено. В отличие от процедурных структур программа не разбивается на подпрограммы.

В "производящих системах" присутствует 4 ступени действий:

Управление отбором утверждений, которые подлежат оценке;

Приказ на проверку отобранных утверждений;

Отбор подмножества действительных следствий, которые выполнились;

Приказ на выполнение отобранных следствий.

Отметим, что все три управляющие структуры допускают параллельное выполнение приложений. Достоинством использования таких структур является то, что программист разрабатывает отдельные участки программы, а всеобщий контроль осуществляется системой при их выполнении.

Высокая степень конкуренции процессов на захват ресурсов в многопроцессорных системах может увеличить сложность решения задач управления, особенно при наличии многих процессоров.

В однопроцессорной системе в многозадачном режиме всегда можно избежать конфликтов путем тщательного планирования процессов, а в случае конфликта — приостановить активность одного из процессов. Программное обеспечение в этом случае должно только эффективно отслеживать возможные ошибки. При этом программное обеспечение, поддерживаемое аппаратно, более надежно.

Поведение исполняемых процессов в многопроцессорных системах более сложное, чем в однопроцессорных. Хотя параллельные программы могут быть и не очень сложными для выполнения, тем не менее возникают некоторые проблемы. С помощью определенных усилий программист может добиться правильной работы параллельных программ, однако обеспечение правильности совместной работы больших программных комплексов является трудной задачей.

3.7.3. Требования к операционной системе СМР

Рассмотрим общие задачи операционной системы. К ним относятся:

Предоставление программного интерфейса и набора услуг, освобождающих программиста от кодирования рутинных операций;

Предоставление пользовательского интерфейса (команд для управления функционированием компьютера) и набор услуг, освобождающих пользователя от выполнения рутинных операций;

Управление ресурсами:

- Упрощение доступа к ресурсам;
- Распределение ресурсов между конкурирующими процессами.

ОС СМР должна обеспечивать надежность выполнения приложений. Все эти и другие возможности ОС СМР определяют ее стоимость, которая иногда может быть неприемлема. Перечисленные выше задачи являются основными для ОС СМР, хотя возможны и отклонения от указанных целей. Степень доступности аппаратных средств программисту также может меняться. В многопроцессорных системах в ОС СМР возникают новые задачи. Такие системы требуют наличия в составе ОС СМР средств взаимодействия и синхронизации процессов.

Асинхронный супервизор процессов выполняет раздельное управление адресным пространством, управление процессом и синхронизацией. Эффективно спроектированные ОС СМР имеют модульную структуру и иерархическую организацию. Это делается для более быстрого выявления локальных ошибок и возможностей расширения ОС СМР.

Классические функции ОС СМР включают возможности создания объектов типа процессы и областей памяти, подчиненным им, которые определяются сегментами памяти. Управление и разделение областей памяти также является важной функцией ОС СМР. Другие функции управления взаимодействием процессов осуществляется через "почтовые ящики" (mailboxes) или "буфер

сообщений" (message buffers). В многопроцессорных системах процессы могут выполнятся параллельно до тех пор, пока они не требуют взаимодействия. Планирование и управление взаимодействием процессов рассматривается как синхронизация процессов, которая осуществляется через глобальные переменные. Необходимость взаимодействия процессов ограничивает их параллелизм. При этом возникают проблемы ограничения доступа к общему ресурсу. На практике эти задачи получили название "задачи производители — потребители". В качестве общего ресурса выступает склад. Доступ каждого из участков задачи к складу возможен только после прекращения доступа другого участка, т.е. когда процесс обращается к глобальным переменным, конкурирующий процесс не имеет доступа к этому участку.

В системах с многочисленными параллельными процессами и ресурсами, которые не могут быть использованы одновременно несколькими процессами, вводится требование к ОС СМР, связанное с обеспечением исключительного доступа одного процесса к этим ресурсам. Это требование сходно с вышеописанными по защите модуля данных во время обращения к нему одного из процессов. Процессы, требующие исключительного доступа к ресурсу, конкурируют за него. Конкуренция между процессами возможна также при обращении к так называемым, виртуальным ресурсам, таким как системные таблицы и буферы связи между взаимодействующими процессами. При этом необходимо обеспечить единичность доступа процесса к таким ресурсам. Этот исключительный доступ называется взаимным исключением между процессами. Требование взаимного исключения при использовании ресурсов подразумевает стремление либо к захвату, либо к освобождению ресурсов. Процессы взаимодействия и конкуренции реализуются с помощью механизма синхронизации. Все описанные выше требования сводятся к задачам процессорного расписания.

Если требуемый ресурс или объект недоступен, процесс приостанавливается, блокируется или его исполнение откладывается до тех пор, пока он не получает доступа к требуемому ресурсу. Существует два уровня исключительного доступа. Первый включает требование обеспечения доступа к структурам данных, которые могут иметь незначительные размеры. Второй уровень включает требование возможной существенной задержки до тех пор, пока физический ресурс, такой как процессор или печатающее устройство, не будет готово. Если задержки короткие, то не стоит "отключать внимание" процессора от процесса, который выполняется на другом процессоре. Если задержка превышает время, требуемое на переключение процесса, то необходимо "переключить внимание" процессора с одного процесса на другой для эффективного использования процессора. Возвращение к прерванному процессу осуществляется по сигналу от запрашиваемого устройства.

Разделение работы многочисленных процессов может быть выполнено размещением нескольких процессов совместно в разделяемой памяти и обеспечением механизма быстрого переключения внимания процессора от одного процесса к другому. Эта операция часто называется контекстное переключение. При разделении работы процессоров возникает три взаимосвязанные проблемы:

* Защита ресурсов и объектов одного процесса от специального или случайного повреждения другими процессами;
* Обеспечение взаимосвязи между процессами и между пользователями процессов, а также управление процессами;
* Размещение ресурсов среди процессов так, чтобы запрос ресурсов всегда удовлетворялся.

Цель защиты состоит в том, чтобы обращение к данным и процедурам происходило правильно. Когда два или больше процессов желают иметь доступ к множеству ресурсов внутри многопроцессорной системы, необходимо разместить ресурсы так, чтобы обращения процессов производились корректно, без тупиковых ситуаций, с максимальным удовлетворением запросов. Кроме того, если у процесса имеется возможность получить часть ресурсов, которые он потребовал, и потом, в процессе использования этих ресурсов, ему понадобились другие, необходимо обеспечить корректный переход на работу с другими ресурсами, т.е. обеспечить динамическое планирование использования ресурсов.