На правах рукописи

ЦЫМБЛЕР Михаил Леонидович

МЕТОДЫ ПОСТРОЕНИЯ ПРОГРАММНОГО КОМПЛЕКСА ДЛЯ УПРАВЛЕНИЯ ДАННЫМИ В ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМАХ С МАССОВЫМ ПАРАЛЛЕЛИЗМОМ

Специальность 05.13.18 – теоретические основы математического моделирования, численные методы и комплексы программ

Диссертация на соискание ученой степени
кандидата физико-математических наук

Научный руководитель:
СОКОЛИНСКИЙ Леонид Борисович,
канд. физ.-мат. наук, доцент

Челябинск-2000

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Одним из актуальных направлений системного программирования является создание системного программного обеспечения для современных многопроцессорных вычислительных комплексов с массивно-параллельной архитектурой.

Одной из основных сфер применения вычислительных систем с массовым параллелизмом в настоящее время являются фундаментальные научные и прикладные задачи, эффективное решение которых возможно только с использованием мощных вычислительных ресурсов. Примером могут служить задачи аэродинамики для самолетостроения и создания реактивных двигателей [14], моделирование управляемого термоядерного синтеза [10], распознавание изображений при навигации движущихся объектов [26], системы поддержки принятия решений, предсказание климата и глобальных изменений в земной коре [17] и др. Многие из упомянутых задач требуют обработки больших объемов данных, хранящихся на внешних носителях.

Другой важной областью применения вычислительных систем с массовым параллелизмом являются сверхбольшие базы данных. Как указывается в Асиломарском отчете о направлениях исследований в области баз данных [3], в ближайшем будущем крупные организации будут располагать базами данных объемом в несколько петабайт. Для эффективной обработки подобных объемов информации требуются параллельные машины с десятками тысяч процессоров. Основными приложениями параллельных машин баз данных являются системы поддержки принятия решений и сверхбольшие базы данных, содержащие объекты сложной структуры: графические, картографические, мультимедийные объекты и др. Одним из наиболее ярких примеров такого рода задач является задача по обработке базы данных EOS/DIS (Earth Observing System/Data Information System) [77, 81] Американского агентства по аэрокосмическим исследованиям

(НАСА). EOS/DIS накапливает данные о наблюдениях за состоянием Земли со спутников, и размер этой базы данных ежедневно увеличивается примерно на 1 терабайт.

В течение последнего десятилетия было создано достаточно большое число прототипов параллельных СУБД (например, Gamma [76], Bubba [69]) и коммерческих систем (например, NonStop SQL [110], NCR Teradata [97] и DB2 Parallel Edition [64]), однако в области параллельных систем баз данных остается много нерешенных проблем [3, 112].

Таким образом, разработка методов построения программных комплексов для управления данными в многопроцессорных вычислительных системах с массовым параллелизмом является одним из актуальных направлений научных исследований в области системного программирования.

В настоящее время одной из перспективных отечественных разработок в сфере многопроцессорных вычислительных систем является многопроцессорный вычислительный комплекс МВС-100/1000 [13, 18, 30]. МВС-100/1000 представляет собой семейство масштабируемых многопроцессорных вычислительных систем с массовым параллелизмом, являющихся совместной разработкой институтов РАН и промышленности.

Вычислительные комплексы МВС-100/1000 используются в ряде академических институтов и университетов России для решения широкого спектра фундаментальных научных и прикладных задач в областях управления динамическими системами и дифференциальных игр, механики сплошной среды, математического программирования, обработки изображений и распознавания образов и др. [42]. Решение многих из указанных задач связано с необходимостью организации хранения и обработки больших объемов персистентных данных. Однако для МВС-100/1000 в настоящее время отсутствует соответствующее специализированное системное программное обеспечение, позволяющее хранить и обрабатывать большие массивы данных. Вследствие этого актуальной темой является разработка

комплекса системных программ для управления данными в многопроцессорной вычислительной системе МВС-100/1000.

Системы МВС-100/1000 способны показывать производительность, сравнимую с лучшими зарубежными суперкомпьютерами [116], оставаясь при этом существенно более экономичными по сравнению с импортными аналогами. Поэтому разработка методов построения программных комплексов для управления данными в системах с массовым параллелизмом и реализованный на их основе программный комплекс для многопроцессорной вычислительной системы МВС-100/1000 имеет большую практическую ценность.

Целью данной диссертационной работы является исследование и разработка методов построения комплекса системных программ для управления данными в вычислительных системах с массовым параллелизмом. Данная цель предполагает решение следующих задач:

1. Исследование и анализ существующих методов организации хранения и передачи данных в многопроцессорных вычислительных системах с массовым параллелизмом.
2. Разработка новых методов управления данными в вычислительных системах с массовым параллелизмом, учитывающих особенности архитектуры современных многопроцессорных систем типа МВС-100/1000.
3. Разработка, реализация и отладка программного комплекса для организации хранения и передачи данных в многопроцессорной вычислительной системе МВС-100/1000.

Диссертационная работа выполнялась в рамках проекта Омега [31-32, 45-53, 58-60, 91, 103-106, 117-118], посвященного разработке высоко-параллельной системы управления базами данных для многопроцессорной вычислительной системы МВС-100/1000. Проект Омега выполняется при финансовой поддержке Российского фонда фундаментальных исследований (гранты 97-07-90148, 00-07-90077).

Данная диссертационная работа состоит из шести глав, заключения и приложения.

В первой главе анализируются архитектурные особенности вычислительных систем с массовым параллелизмом. Приводится классификация архитектур многопроцессорных вычислительных систем. Описывается архитектура многопроцессорной вычислительной системы МВС-100/1000, и анализируется структура задач, связанных с обработкой больших массивов данных на МВС-100/1000. На основании данного анализа предлагается подход к организации систем управления данными для массивно-параллельных платформ, основанный на введении трех уровней абстракции: аппаратного, физического и логического.

Во второй главе рассматриваются методы организации хранения и передачи данных в многопроцессорных вычислительных системах с массовым параллелизмом. Предлагается классифицировать данные, обрабатываемые в вычислительной системе с массивно-параллельной архитектурой, в зависимости от объема, времени их жизни и числа обращений к ним. На основании данной классификации предлагаются методы организации управления данными, учитывающие специфику архитектуры современных вычислительных систем с массовым параллелизмом.

В третьей главе описывается структура программного комплекса Омега для управления данными в многопроцессорной вычислительной системе МВС-100. Комплекс Омега включает в себя модуль топологии, менеджер нитей (легковесных процессов), систему передачи сообщений и систему хранения данных. Приводится описание принципов реализации модуля топологии, менеджера нитей и системы передачи сообщений.

Четвертая глава посвящена методам реализации системы хранения комплекса Омега. Описаны принципы реализации электронной дисковой подсистемы и системы управления файлами, являющихся составными частями системы хранения данных для МВС-100.

В пятой главе рассматриваются методы управления буферным пулом, используемые в системе управления файлами. Анализируется проблематика буферизации данных и приводится обзор наиболее известных методов управления буферным пулом. Описывается оригинальный метод организации вытеснения страниц из буферного пула, основанный на введении статического и динамического рейтингов страниц и использовании избыточного индекса буферного пула. Приводятся результаты численных экспериментов, подтверждающих эффективность предложенного метода.

Шестая глава посвящена технологическим аспектам разработки программного комплекса Омега. Описывается оригинальная технология коллективной разработки больших программных систем для МВС-100, структура программных средств поддержки данной технологии и компоненты расширения среды программирования (отладчик и профилировщик), разработанные в рамках проекта Омега.

В заключении перечислены основные результаты диссертационной работы.

В приложение вынесены протоколы взаимодействия клиентской и серверной частей системы хранения данных.

ГЛАВА 1. АНАЛИЗ АРХИТЕКТУРНЫХ ОСОБЕННОСТЕЙ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ С МАССОВЫМ ПАРАЛЛЕЛИЗМОМ

1.1 Обзор архитектур многопроцессорных вычислительных систем

К настоящему времени написано достаточно большое количество работ по систематизации архитектур вычислительных систем. Достаточно полный обзор методов описания и классификации вычислительных систем можно найти в работе [9]. В данном разделе рассматриваются классификация многопроцессорных архитектур в зависимости от наличия общей или распределенной оперативной памяти, а также классификации, предложенные Стоунбрейкером и Флинном.

1.1.1 Классификация аппаратных архитектур многопроцессорных вычислительных систем

Одним из основных признаков классификации многопроцессорных вычислительных систем является наличие общей или распределенной оперативной памяти. В соответствии с данным признаком различают следующие основные классы аппаратных архитектур многопроцессорных вычислительных систем: SMP архитектура, MPP архитектура и NUMA архитектура [11, 28, 98].

Система с *SMP (Symmetric MultiProcessing) архитектурой* состоит из нескольких однородных процессоров и массива общей памяти. Обычно процессоры подключаются к памяти с помощью общей шины. Для достижения хорошей производительности каждый процессор имеет собственную кэш-память, поскольку основная память работает слишком медленно по сравнению со скоростью процессоров. Когерентность кэшей процессоров поддерживается аппаратно. SMP-система работает под управлением

единой операционной системы (обычно типа ОС UNIX), которая в процессе работы автоматически распределяет процессы (нити управления) по процессорам. Обмен сообщениями в SMP-системах организован через общую память, что существенно упрощает взаимодействие процессоров между собой. Однако исследования показывают, что, начиная с некоторого числа процессоров, доступ к общей памяти в SMP-системе становится узким местом [68, 111]. Вследствие этого в реальных SMP-системах число процессоров ограничено 30 [98, 113]. Примерами систем с SMP архитектурой являются HP 9000 V class (Hewlett-Packard) и SMP-серверы на базе процессоров Intel (IBM, Compaq, Dell).

Система с *MPP (Massively Parallel Processing) архитектурой* строится из однотипных вычислительных узлов, включающих один или несколько центральных процессоров, локальную память, коммуникационный процессор или сетевой адаптер и жесткий диск. Узлы объединяются в систему с помощью некоторой коммуникационной среды (высокоскоростная сеть, коммутатор и др.). Обычно в MPP-систему добавляется узел, с которого осуществляется общее управление системой. На управляющем узле устанавливается операционная система, обеспечивающая функции запуска и завершения приложения в вычислительной системе. Остальные узлы работает под управлением облегченного варианта ОС, обеспечивающего только работу запускаемого на нем процесса. Общее число процессоров в MPP-системах может достигать нескольких тысяч. Примерами MPP-систем являются RS/6000 SP2 (IBM), CRAY T3E (SGI), SR8000 (Hitachi), транспьютерные системы Parsytec.

Кластерная архитектура [1] идеологически близка к MPP архитектуре. Кластерная система представляет собой набор рабочих станций или персональных компьютеров общего назначения, объединенных в систему с помощью одной из стандартных сетевых технологий (Fast или Gigabit Ethernet, Myrinet и др.) на базе шинной архитектуры или коммутатора. При

объединении в кластер компьютеров разной архитектуры говорят о гетерогенных (неоднородных) кластерах.

Система с *NUMA (Non Uniform Memory Architecture) архитектурой* состоит из однотипных базовых модулей, состоящих из небольшого числа процессоров и блока памяти. Модули объединены в сеть с помощью высокоскоростного коммутатора. Аппаратно поддерживается единое адресное пространство, то есть доступ к памяти удаленных модулей. При этом доступ к локальной памяти в несколько раз быстрее, чем к удаленной. В случае, если аппаратно поддерживается когерентность кэшей процессоров во всей системе, говорят об архитектуре *ccNUMA (cache-coherent NUMA)*. Обычно вся система работает под управлением единой ОС, как в случае SMP-систем. Масштабируемость NUMA-систем ограничивается объемом адресного пространства, возможностями аппаратуры поддержки когерентности кэшей и возможностями операционной системы по управлению большим числом процессоров. На настоящий момент максимальное число процессоров в NUMA-системах составляет 256. Примерами NUMA-систем являются Origin2000 (SGI), HPC 10000 (Sun), NUMA-Q 2000 (IBM).

1.1.2 Классификация Стоунбрейкера

Одним из важных приложений многопроцессорных вычислительных систем являются параллельные машины баз данных [15]. К параллельным машинам баз данных предъявляются следующие основные требования [36]: масштабируемость, балансируемость, отказоустойчивость и высокая готовность данных.

Масштабируемость – это возможность пропорционального увеличения общей производительности системы путем добавления соответствующих аппаратных ресурсов (процессоров, модулей памяти, дисковых накопителей и др.).

Балансируемость подразумевает возможность эффективной динамической балансировки загрузки процессоров.

Отказоустойчивость – это способность системы сохранять общую работоспособность при отказе части оборудования.

Под *высокой готовностью данных* понимается способность системы восстанавливать данные, утраченные в результате отказа оборудования, при одновременном сохранении способности выполнять запросы к базе данных.

В соответствии с классификацией Стоунбрейкера [109] аппаратные архитектуры параллельных систем баз данных могут быть разделены на три основных класса: архитектура с разделяемой памятью и дисками, архитектура с разделяемыми дисками и архитектура без совместного использования ресурсов. Далее указанные архитектуры анализируются в контексте основных требований, предъявляемых к параллельным машинам баз данных, и рассматривается возможность их реализации на базе аппаратных платформ, описанных в разделе 1.1.1.

В *системах с разделяемой памятью и дисками* (см. Рис. 1.1) все процессоры (**P**) с помощью общей шины соединяются с разделяемой памятью (**M**) и дисками (**D**). Обозначим такую архитектуру как *SE (Shared-Everything)*. В качестве аппаратной платформы для реализации SE-систем обычно используются машины с архитектурами SMP и NUMA.

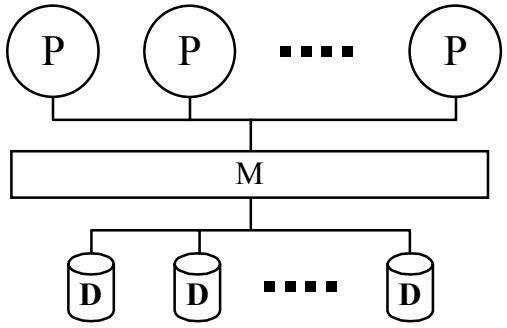


Рис. 1.1 Архитектура с разделяемой памятью и дисками

В SE-системах межпроцессорные коммуникации могут быть реализованы очень эффективно через разделяемую память. Поскольку в SE-системах каждому процессору доступны вся память и любой диск, проблема балансировки загрузки процессоров не вызывает принципиальных

трудностей (простаивающий процессор можно легко переключить с одного диска на другой). В силу этого SE-системы демонстрируют на небольших конфигурациях (не превышающих 20 процессоров) более высокую производительность по сравнению с остальными архитектурами [113].

Однако SE-архитектура имеет ряд недостатков, из которых наиболее существенными являются ограниченная масштабируемость и низкая аппаратная отказоустойчивость. При большом количестве процессоров в SE-системе начинают возникать конфликты доступа к разделяемой памяти, что может привести к серьезной деградации общей производительности системы [111]. В соответствии с этим масштабируемость реальных SE-систем ограничивается 20-30 процессорами [113]. Кроме этого, SE-системы не могут обеспечить высокую готовность данных при отказах аппаратуры. Выход из строя практически любой аппаратной компоненты приводит к выходу из строя всей системы. Действительно, отказ модуля памяти, шины доступа к памяти или шины ввода-вывода очевидно приводит к выходу из строя системы в целом. Выход из строя одного из процессоров также приводит к выходу из строя системы в целом в силу невозможности получить доступ к данным в кэш-памяти отказавшего процессора [98]. Что касается дисков, то обеспечение высокой готовности данных требует дублирования одних и тех же данных на разных дисках [83]. Однако поддержание когерентности зеркальных копий данных может существенным образом снизить общую производительность SE-системы в силу ограниченной пропускной способности шины ввода-вывода. Все это делает невозможным использование SE-архитектуры в чистом виде для систем с высокими требованиями к готовности данных.

В системах *с разделяемыми дисками* (см. Рис. 1.2) каждый процессор (**P**) имеет собственную приватную память (**M**). Процессоры соединяются друг с другом и с дисковыми подсистемами (**D**) с помощью некоторой соединительной сети (**N**). При этом любой процессор имеет доступ к любому диску. Обозначим такую архитектуру как SD (Shared-Disk).

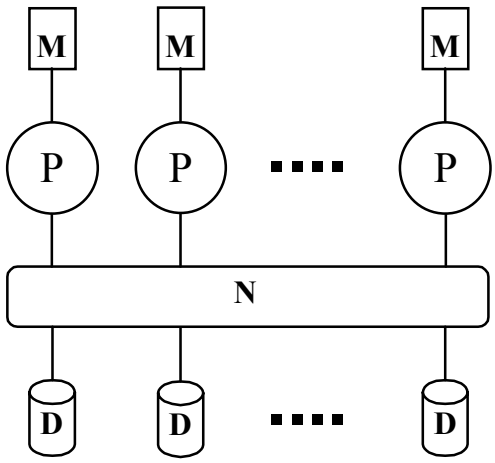


Рис. 1.2 Архитектура с разделяемыми дисками

SD-архитектура, по сравнению с SE-архитектурой, демонстрирует лучшую масштабируемость и более высокую степень доступности данных при отказах аппаратуры. Однако при реализации SD-систем возникает ряд серьезных технических проблем, среди которых наиболее существенными являются проблема организации глобальной таблицы блокировок и проблема обеспечения когерентности кэшей, связанная с поддержанием непротиворечивости образов одних и тех же дисковых страниц в буферных пулах различных процессоров [93]. Как указывается в [107], на сегодня нет весомых причин для поддержки SD-архитектуры в чистом виде.

В системах *без совместного использования ресурсов* (см. Рис. 1.3) каждый процессор (**P**) имеет собственную приватную память (**M**) и собственный диск (**D**). Процессоры соединяются друг с другом с помощью некоторой соединительной сети (**N**). Обозначим такую архитектуру как SN (Shared-Nothing). В качестве аппаратной платформы для реализации SN-систем обычно используются машины с архитектурой MPP и кластеры.

SN-архитектура имеет наилучшие показатели по масштабируемости и отказоустойчивости. Основным недостатком SN-архитектуры является потенциальная сложность с обеспечением сбалансированной загрузки процессоров. Действительно, в SN-системе невозможно непосредственно переключить простаивающий процессор на обработку данных, хранящихся на "чужом" диске. Чтобы разгрузить некоторый процессорный узел, необ-

ходимо часть "необработанных" данных переместить по соединительной сети на свободный процессорный узел. На практике это приводит к существенному падению общей эффективности системы из-за высокой стоимости пересылки больших объемов данных по соединительной сети. Поэтому перекосы в распределении данных по процессорным узлам могут привести к полной деградации общей производительности SN-системы [90].

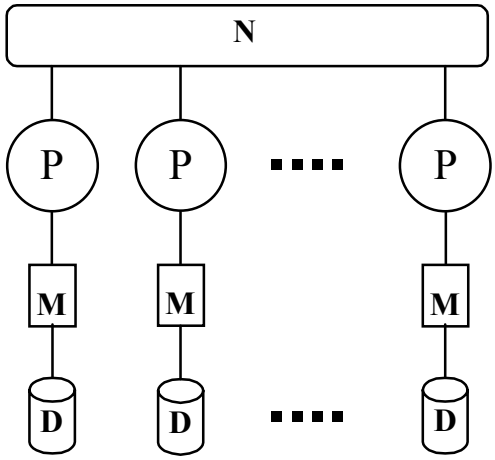


Рис. 1.3 Архитектура без совместного использования ресурсов

Для преодоления недостатков, присущих SE и SN архитектурам, в работе [68] было предложено рассматривать *иерархические (гибридные) архитектуры*, в которых SE-кластеры (нижний уровень иерархии) объединяются в SN-систему (верхний уровень иерархии) (см. Рис. 1.4). Обозначим такую архитектуру как CE (Clustered-Everything).

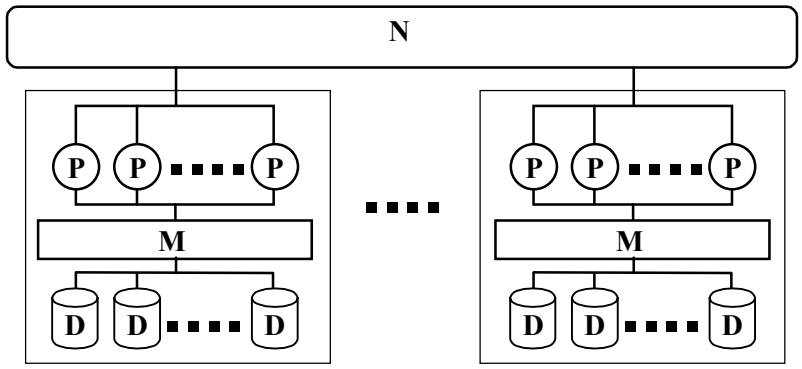


Рис. 1.4 Иерархическая CE-архитектура

CE-архитектура обладает хорошей масштабируемостью, подобно SN-архитектуре, и позволяет достигать приемлемого баланса загрузки, подобно SE-архитектуре [70, 86]. При проектировании CE-систем количество узлов в SE-кластере должно быть настолько велико, насколько это позволяют аппаратные ограничения [115]. Другими словами, CE-архитектура имеет тенденцию к эволюции в направлении к чистой SE-архитектуре, по мере того как будут сниматься ограничения на масштабируемость SE-кластеров.

Основным недостатком CE-архитектуры являются потенциальные трудности с обеспечением доступности данных при отказах аппаратуры на уровне SE-кластера. Для обеспечения высокой доступности данных необходимо дублирование одних и тех же данных на разных SE-кластерах, что требует пересылки по соединительной сети значительных объемов данных. Данное обстоятельство может существенным образом снизить общую производительность системы в режиме нормального функционирования и привести к тому, что SE-кластеры станут работать с производительностью, сопоставимой с производительностью однопроцессорных конфигураций.

1.1.3 Классификация Флинна

Классификация Флинна [79, 80] является одной из первых и наиболее известных классификаций архитектур вычислительных систем. Данная классификация базируется на понятии *потока*, под которым понимается последовательность элементов, команд или данных, обрабатываемая процессором. Каждый поток не зависит от других потоков, и каждый элемент потока может состоять из нескольких команд или данных. На основе числа потоков команд и потоков данных Флинн выделяет четыре класса архитектур: SISD, MISD, SIMD, MIMD.

Система с *SISD (single instruction, single data) архитектурой* предполагает поток управляющих инструкций и поток данных в единственном числе каждый (см. Рис. 1.5; в этом и последующих рисунках **P** означает

процессор, **I** – поток управляющих инструкций, $\mathbf{D_{inp}}$ – входной поток данных, $\mathbf{D_{out}}$ – выходной поток данных). В компьютерах данного класса имеется только один поток команд, все команды обрабатываются последовательно друг за другом, и каждая команда инициирует одну операцию с одним потоком данных. К классу SISD относят прежде всего классические однопроцессорные машины фон-неймановского типа, например, PDP-11 [43] или VAX 11/780 [12].

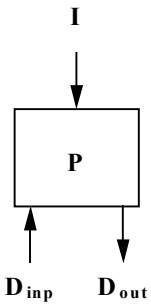


Рис. 1.5 SISD архитектура

Система с *SIMD (single instruction, multiple data) архитектурой* предполагает единственный поток управляющих инструкций и множественные потоки данных (см. Рис. 1.6).

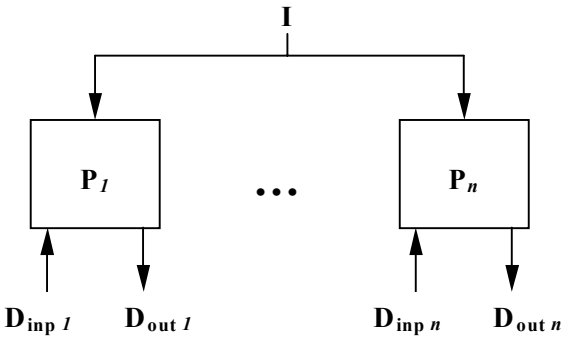


Рис. 1.6 SIMD архитектура

В системах подобного рода сохраняется один поток команд, включающий, в отличие от предыдущего класса, векторные команды. Данная особенность позволяет выполнять одну арифметическую операцию сразу над многими данными (элементами вектора). К данному классу относят, например, компьютеры ILLIAC IV [54] (обработка элементов вектора про-

изводится матрицей процессоров) и CRAY-1 [12] (обработка элементов вектора производится с помощью конвейера).

Система с *MISD (multiple instruction, single data) архитектурой* предполагает множественные потоки управляющих инструкций и единственный поток данных (см. Рис. 1.7).

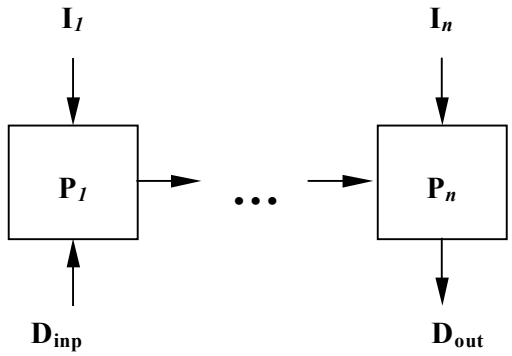


Рис. 1.7 MISD архитектура

В машине с данной архитектурой более одного процессора обрабатывают один и тот же поток данных. По мнению специалистов в области классификации многопроцессорных архитектур [74, 101], компьютеры с MISD-архитектурой в настоящее время отсутствуют.

Система с *MIMD (multiple instruction, multiple data) архитектурой* предполагает множественные потоки управляющих инструкций и множественные потоки данных (см. Рис. 1.8).

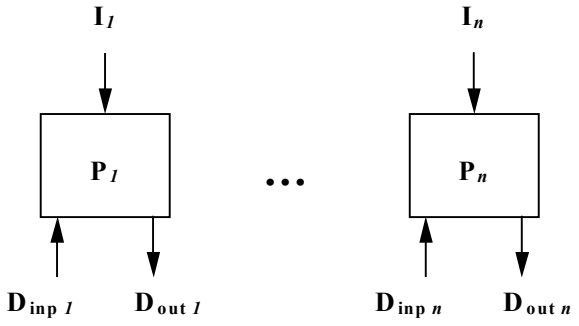


Рис. 1.8 MIMD архитектура

В вычислительной системе данного класса есть несколько устройств обработки команд, объединенных в единый комплекс и работающих каж-

дое со своим потоком команд и данных. Класс MIMD включает в себя широкий спектр многопроцессорных систем, примерами которых являются C.mmp [85], Cm* [6], Denelcor HEP [21], BBN Butterfly [5], CRAY T3D [54] и многие другие.

Классификация Флинна вплоть до настоящего времени является самой применяемой при начальной характеристике того или иного компьютера, поскольку точно определяет базовый принцип его работы. Недостатком данной классификации является чрезмерная заполненность класса MIMD. В частности, системы с SMP и MPP архитектурой попадают в один класс MIMD, но являются различными по способу организации памяти.

1.2 Архитектура МВС-100/1000

1.2.1 Аппаратная архитектура МВС

Вычислительные системы МВС-100/1000 [13, 18, 30, 116] являются совместной разработкой институтов РАН и промышленности.

МВС строится по модульному принципу. *Типовой процессорный модуль* (далее ТПМ) имеет архитектуру с разделяемой памятью и включает в себя вычислительный процессор **CP** и коммуникационный (сетевой) процессор **NP** (см. Рис. 1.9). Взаимодействие вычислительного и коммуникационного процессоров осуществляется через общую оперативную память (**SRAM**). Кроме этого, коммуникационный процессор имеет собственную приватную память (**DRAM**). Коммуникационный процессор оснащен высокоскоростными внешними каналами (линками) для соединения с другими ТПМ и модулями дисковой подсистемы (МДП). МДП представляет собой специальную интерфейсную плату со стандартным коммуникационным процессором, к которой по SCSI интерфейсу могут быть подключены накопители на жестких магнитных дисках. Коммуникационный процессор модуля дисковой подсистемы имеет четыре серийных линка для соединения с ТПМ.

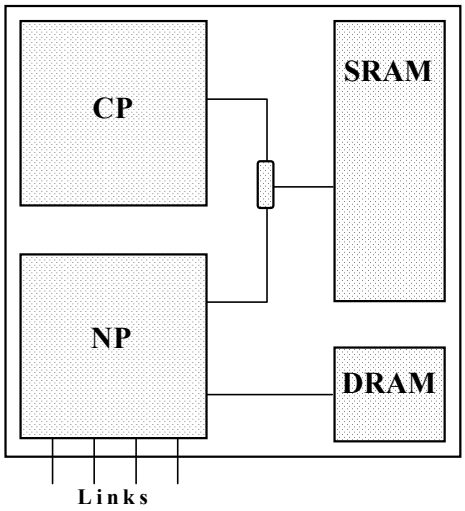


Рис. 1.9 Структура типового процессорного модуля МВС-100/1000

ТПМ устанавливаются в стойках промышленного стандарта (от 4 до 64 процессорных модулей в одной стойке). Стойки могут соединяться между собой для формирования единой вычислительной системы, объединяющей в себе до тысячи процессоров. Управление МВС осуществляется через host-машину, представляющую собой обычный персональный компьютер или рабочую станцию.

В качестве вычислительного процессора в ТПМ для МВС-100 используется суперскалярный процессор i860 фирмы Intel. В качестве коммуникационного процессора используется транспьютер Т805, имеющий четыре внешних канала с пропускной способностью 20 Мбит/с каждый. Модуль оснащается разделяемой оперативной памятью объемом 8-32 Мбайт.

В качестве вычислительного процессора в ТПМ для МВС-1000 используется процессор Alpha 21164 фирмы DEC-Compaq. В качестве коммуникационного процессора используется микропроцессор TMS320C44 фирмы Texas Instruments, имеющий 4 внешних канала с пропускной способностью 20 Мбайт/с каждый, либо микропроцессор SHARC ADSP 21060 фирмы Analog Devices, имеющий 6 внешних каналов с пропускной способностью 40 Мбайт/с каждый. Модуль оснащается разделяемой оперативной памятью объемом 0,1-2 Гбайт.

В соответствии с классификацией Флинна многопроцессорные системы семейства МВС-100/1000 относятся к классу MIMD. На уровне всей системы МВС можно рассматривать как систему класса MPP, поскольку имеет место распределенность оперативной памяти. На уровне процессорного модуля архитектура МВС подобна архитектуре SMP, поскольку вычислительный и коммуникационный процессоры разделяют общую память. Однако, в отличие от SMP-систем, процессоры, входящие в процессорный модуль МВС, не симметричны. Вычислительный процессор несет нагрузку, связанную с вычислениями, и не занимается обеспечением транзитных передач данных по соединительной сети. Основной задачей коммуникационного процессора, напротив, является организация межпроцессорных обменов данными.

С точки зрения классификации Стоунбрейкера вычислительная система МВС может рассматриваться как SN-система при условии, что к каждому процессорному модулю подключается отдельный диск. При подключении к различным ТПМ одного и того же диска МВС можно рассматривать как SD-систему.

1.2.2 Операционная система Router для МВС-100

В качестве операционной системы на МВС-100 используется ОС Router разработки ИПМ им. М.В. Келдыша РАН.

ОС Router [29] представляет собой распределенную операционную систему класса toolset. Она состоит из процессорного сервера, который запускается на каждом процессорном узле и host-сервера (iserver.exe), запускаемого на host-машине. И процессорный сервер, и host-сервер загружаются всякий раз перед загрузкой задачи пользователя.

ОС Router обеспечивает следующие основные функции:

- загрузку программ пользователя с host-машины на процессорные модули;

- обмен данными между программой и host-машиной в виде некоторого подмножества системных функций ввода/вывода UNIX;
- обмен сообщениями по линкам между программами, запущенными на различных вычислительных процессорах.

Задача пользователя может выполняться только на вычислительных процессорах. Задача пользователя представляет собой совокупность процессов. На каждом процессорном модуле может выполняться только один пользовательский процесс. Каждый процесс реализуется в виде программы, которая с точки зрения системы программирования является отдельным загрузочным модулем. Для обмена сообщениями с другими программами в тексте программы необходимо в явном виде использовать функции ОС Router, представленные на Рис. 1.10. Обмен сообщениями возможен между любыми двумя процессами и построен по принципу асинхронных виртуальных каналов.

```
/* Запуск чтения сообщения от процесса.
proc – номер процесса
buffer – указатель на буфер с сообщением
length – длина сообщения
Возвращает 0, если обмен запущен или отрицательный код ошибки. */
int r_read( int proc, void *buffer, int length);

/* Запуск записи сообщения указанному процессу.
proc – номер процесса
buffer – указатель на буфер с сообщением
length – длина сообщения
Возвращает 0, если обмен запущен или отрицательный код ошибки. */
int r_write( int proc, void *buffer, int length);

/* Проверка, свободен ли канал для приема сообщения.
proc – номер процесса
Возвращает 1, если чтение сообщения завершено или отрицательный код ошибки. */
int t_read( int proc);

/* Проверка, свободен ли канал для записи сообщения.
proc – номер процесса
Возвращает 1, если запись сообщения завершена или отрицательный код ошибки. */
int t_write( int proc);
```

Рис. 1.10 Функции обмена сообщениями ОС Router

ОС Router не предоставляет средств организации параллельных нитей управления (легковесных процессов) в рамках одного процесса (процессора).

ОС Router не обеспечивает средств написания программ на процессорной сети в целом. ОС Router не предоставляет также средств для работы с файлами дисковой подсистемы.

Для реализации функций обмена сообщениями ОС Router на нижнем уровне использует средства, предоставляемые пакетом INMOS Toolset. Инструментальный пакет *INMOS Toolset* [35] используется для разработки параллельных программ, которые должны работать в сети коммуникационных процессоров (транспьютеров). Средства пакета INMOS Toolset позволяют строить прикладные программы из произвольного числа независимо выполняющихся процессов, использующих одно и тоже виртуальное адресное пространство (сопроцессы) либо разные виртуальные адресные пространства (процессы). Процессы можно распределить по транспьютерам сети. Распределение процессов по транспьютерам, установление связи между ними называется конфигурированием прикладной программы. Информация о конфигурировании программы записывается на Си-подобном языке, называемом языком конфигурирования программ. Описание конфигурации программы используется сборщиком процессов для построения загружаемой в транспьютерную сеть программы.

В качестве операционной системы на МВС может использоваться также *ОС Helios* (распределенный аналог ОС UNIX), однако она практически не применяется для обработки больших массивов данных, поскольку не удовлетворяет требованию эффективности (в ОС Helios латентная часть организации передачи одного сообщения между процессорными узлами составляет около 3000 байт; для сравнения, в ОС Router латентная часть организации передачи одного сообщения составляет около 400 байт).

1.3 Иерархический подход к организации систем управления данными для массивно-параллельных вычислительных систем

Анализ задач, решаемых на МВС-100/1000, показывает, что имеется большой класс задач, связанных с обработкой больших объемов данных,

при распараллеливании которых обычно применяется следующий подход. Вычислительная задача разбивается на подзадачи. Для решения подзадач в процессорном массиве системы выделяются непересекающиеся группы процессорных модулей, называемых полями. Подзадача, в свою очередь, разбивается на процессы, и каждый процесс запускается на отдельном процессорном модуле поля. В соответствии со спецификой задач интенсивность межпроцессорных обменов данными внутри поля (в рамках одной подзадачи), как правило, существенно превосходит интенсивность обменов между полями (между подзадачами) вплоть до нескольких порядков. В качестве примеров задач указанного класса можно привести следующие задачи: численное моделирование высоковязких течений в верхней мантии земной коры [25, 57], численное моделирование динамики блоковой структуры литосферы [17, 34], расчет параметров акустических колебаний в вихревых и газовых потоках [23, 24], исследование методов высокоточной навигации [26] и др.

При реализации задач указанного класса прикладные программисты, выполняя разбиение множества процессоров, доступных задаче, на поля, и назначая роли процессоров внутри поля, как правило, не учитывают физическую топологию системы. При этом для реализации межпроцессорных коммуникаций прикладные программисты используют системные сервисы ОС Router. Однако реализация ОС Router рассчитана на произвольную топологию межпроцессорных соединений. Маршрут, по которому сообщение одного процессорного узла передается другому процессорному узлу, недетерминирован и определяется в зависимости от текущей загрузки всех узлов в сети. Вследствие этого функции межпроцессорного обмена, предоставляемые ОС Router, часто оказываются неадекватными и неэффективными для небольших полей с сильносвязной топологией. В связи с этим возникает необходимость в разработке комплекса системных программ, который мог бы использоваться прикладными программистами для

эффективного управления данными при реализации задач указанного класса.

Отмеченные особенности задач, связанных с хранением и передачей данных, и рассмотренная выше архитектура МВС-100/1000 делают целесообразным введение в структуре системы управления данными для массивно-параллельных платформ трех уровней абстракции: аппаратного, физического и логического.

Аппаратный уровень представлен несимметричным процессорным модулем, который состоит из коммуникационного и вычислительного процессоров с разделяемой памятью. Обмен данными между коммуникационным и вычислительным процессорами реализуется аппаратно и микропрограммно. Аппаратный уровень отражает особенности реализации процессорного узла.

На *физическом уровне* система представляет собой множество процессорных узлов, объединенных высокоскоростной соединительной сетью. Обмен данными на этом уровне реализуется на основе сервисов, предоставляемых операционной системой типа ОС Router. Физический уровень отражает способ объединения процессорных узлов в единую систему.

На *логическом уровне* множество процессорных узлов системы разбивается на непересекающиеся группы процессоров, называемых кластерами. *Процессорный кластер* состоит из фиксированного числа процессорных узлов, между которыми предполагается интенсивный обмен данными. Обмены данными между узлами разных процессорных кластеров полагаются имеющими относительно низкую интенсивность. Таким образом, на логическом уровне система представляет собой множество процессорных кластеров, объединенных высокоскоростной соединительной сетью. Логический уровень отражает способ разбиения процессорного массива на непересекающиеся кластеры.

В соответствии с данным подходом в работах [103, 106] была предложена трехуровневая иерархическая архитектура *CD_2 (Clustered-Disks*

with 2-processor modules) для использования платформы МВС-100/1000 в качестве параллельного сервера баз данных.

Первый уровень системной иерархии представлен типовыми процессорными модулями МВС-100/1000, которые были описаны выше.

Второй уровень системной иерархии представлен *Омега-кластерами*. Все Омега-кластеры имеют стандартную предопределенную архитектуру с небольшим количеством процессорных модулей и дисковой подсистемой, объединенными в единую сеть с высокой степенью связности (длина кратчайшего пути между любыми двумя узлами сети не должна превышать значения 2). Указанные ограничения делают возможным разработку протоколов обмена данными [104, 105], позволяющих реализовать межпроцессорные обмены между узлами Омега-кластера более эффективно, чем в ОС Router. Каждый кластер имеет свободные линки для связи с другими кластерами.

Примеры возможных конфигураций Омега-кластеров приведены на Рис. 1.11. Здесь ТПМ означает типовой процессорный модуль, МДП – модуль дисковой подсистемы.

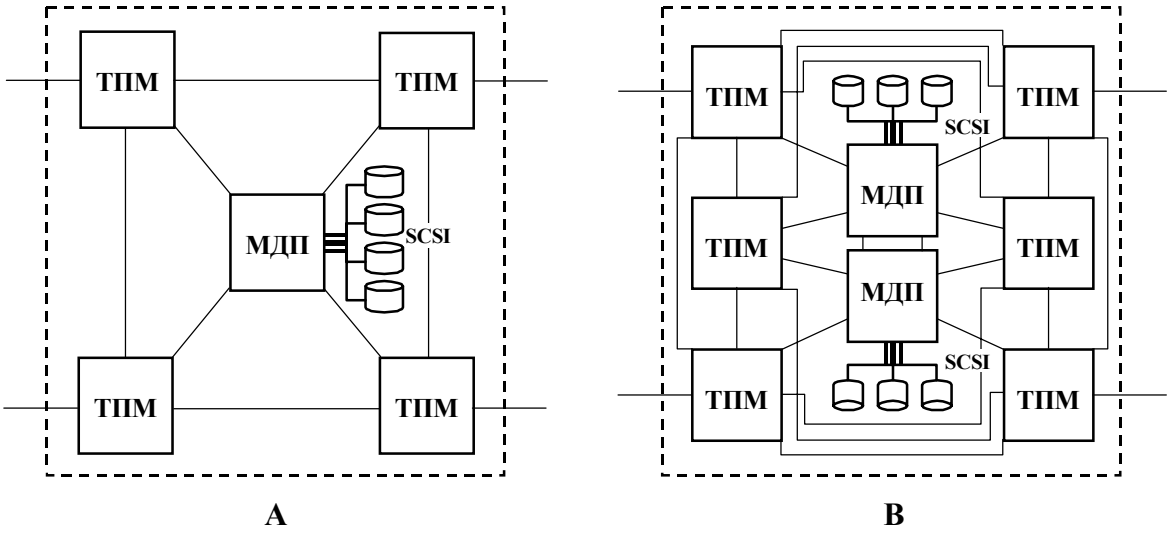


Рис. 1.11 Примеры конфигураций Омега-кластера

Конфигурация ***A*** предполагает использование ТПМ с четырьмя внешними каналами. Конфигурация ***B*** предполагает использование ТПМ с

шестью внешними каналами. Использование двух дисковых подсистем в конфигурации ***B*** позволяет повысить отказоустойчивость Омега-кластера.

На *третьем уровне системной иерархии* Омега-кластеры объединяются в единую систему по схеме SN. При этом топология межкластерных соединений не фиксируется и может варьироваться от простой линейки до гиперкуба.

Данная архитектура обеспечивает высокую готовность данных и возможность эффективной балансировки загрузки в условиях перекосов данных [48].

ГЛАВА 2. МЕТОДЫ ОРГАНИЗАЦИИ ХРАНЕНИЯ И ПЕРЕДАЧИ ДАННЫХ В МНОГОПРОЦЕССОРНЫХ ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМАХ С МАССОВЫМ ПАРАЛЛЕЛИЗМОМ

2.1 Распараллеливание задач обработки данных

Аппаратная архитектура МВС-100 предоставляет два возможных вида реального распараллеливания работ:

1. Распараллеливание путем разделения задачи на подзадачи и выполнение каждой подзадачи на отдельном процессорном модуле в виде *независимого процесса.* Подобный вид параллелизма важен для вычислительных задач, то есть задач, требующих значительных затрат процессорного времени при минимальных обменах данными между задачами.

2. Распараллеливание обменов данными между процессорными модулями, основанное на асинхронном режиме вызова системных функций обмена сообщениями (функций ОС Router). Подобный вид параллелизма важен для задач, связанных с интенсивными обменами данными между процессорными узлами.

Единицей распараллеливания для первого вида обычно является процесс. Единицей распараллеливания для второго вида является *легковесный процесс (нить управления)* [27].

Под процессом понимается задача, запущенная на отдельном процессорном модуле и выполняющаяся на вычислительном процессоре. На одном процессорном модуле не может выполняться более одного процесса. Данное понимание семантики слова процесс совпадает с его трактовкой в ОС Router. В соответствие с этим, обмен сообщениями между процессами базируется на соответствующих операциях ОС Router.

В силу особенностей программно-аппаратной платформы МВС-100/1000 невозможно динамическое перераспределение процессов по процессорам во время работы задачи. Поэтому процессы, нуждающиеся в динамическом планировании, необходимо организовывать в виде нитей (сопрограмм). Пусть, например, у нас имеется два процессора ***P1*** и ***P2*** и три процесса ***A***, ***B*** и ***C***. Пусть ***A*** всегда запускается на ***P1***, ***B*** всегда запускается на ***P2***, а ***C*** запускается на том из двух процессоров, который в данный момент менее загружен. Для организации подобного планирования создается две задачи ***X*** и ***Y***. ***X*** включает в себя в качестве сопрограмм (нитей) процессы ***A*** и ***C***, ***Y*** включает в себя в качестве сопрограмм процессы ***B*** и ***C***.

Таким образом, для эффективного распараллеливания обменов данными между процессорными модулями в структуре комплекса системных программ для управления данными должна присутствовать *подсистема поддержки легковесных процессов*. Данная подсистема должна располагаться на нижнем уровне системной иерархии и обеспечивать поддержку легковесных процессов (нитей управления), предоставляя средства для диспетчеризации и синхронизации нитей.

2.2 Классификация данных

Распараллеливание задач, решаемых на МВС-100/1000, обычно выполняется путем разделения задачи на подзадачи. Каждая подзадача выполняется на отдельном процессорном модуле в виде независимого процесса.

Данные, обрабатываемые процессом, могут иметь объем, позволяющий целиком поместить их в оперативную память процессорного модуля. Однако на практике часто встречается ситуация, когда данные не могут быть целиком помещены в оперативную память процессорного модуля и обрабатываются по частям.

Введем понятие блока данных как единицы структуризации данных, обрабатываемых процессом. *Блок данных* содержит взаимосвязанные дан-

ные, которые не могут обрабатываться процессом по частям (последовательно: сначала одна часть блока, затем другая). Неделимость блока заключается в том, что все его элементы должны быть доступны процессу в течение всего времени обработки блока. Примером разбиения данных на блоки может быть страничная организация файлов базы данных, хранящихся на диске. Примером блока данных в вычислительной задаче является массив коэффициентов системы алгебраических или дифференциальных уравнений.

Мы можем различать блоки данных в зависимости от *количества обращений* процесса к ним во время обработки. Блоки могут быть однократно или повторно (многократно) используемыми. Например, однобайтовое сообщение, передаваемое от одного процесса другому для их взаимной синхронизации является *однократно используемым блоком*, а страница файла базы данных или массив коэффициентов – *повторно используемым блоком*.

Блоки данных могут иметь различное *время жизни*: *до выполнения задачи*, *в период выполнения задачи* и *после выполнения задачи*. В рассматриваемом нами примере сообщение существует только в период выполнения задачи, а страницы файлов базы данных существуют не только в период выполнения задачи, но также до и после выполнения задачи.

Проведенные рассуждения позволяют нам выделить следующие три категории данных, обрабатываемых в вычислительных системах с массовым параллелизмом: персистентные данные, временные данные и сообщения [59].

Персистентные данные – это файлы данных, которые существуют до и после выполнения задачи. Мы будем предполагать, что в общем случае персистентные данные не помещаются в оперативную память процессорного модуля и поэтому обрабатываются блоками. Будем предполагать также, что блоки персистентных данных являются повторно используемыми. Обработка персистентных данных характерна для параллельных сис-

тем баз данных, основанных на реляционной [22] или объектно-ориентированной [63] моделях данных.

Временные данные – это данные, которые существуют только в период выполнения задачи. Мы будем предполагать, что временные данные не помещаются в оперативную память процессорного модуля и поэтому обрабатываются блоками. Будем считать также, что блоки временных данных являются повторно используемыми. Обработка временных данных характерна для вычислительных задач большой размерности.

Сообщения – это данные, которые могут быть целиком помещены в оперативную память процессорного модуля. Мы будем предполагать, что сообщения всегда являются однократно используемыми данными и существуют только в период выполнения задачи. Данные такой природы возникают, например, при синхронизации различных процессов.

На основании данной классификации в структуре комплекса системных программ для управления данными в многопроцессорной вычислительной системе с массовым параллелизмом должны выделяться следующие подсистемы: система передачи сообщений и система хранения персистентных и временных данных. *Система передачи сообщений* предоставляет средства обмена сообщениями между любыми двумя узлами вычислительной системы. *Система хранения персистентных и временных данных* предоставляет унифицированные средства для обработки персистентных и временных данных. В реализации функций указанных подсистем используются средства подсистемы поддержки легковесных процессов (нитей управления).

2.3 Методы построения комплекса системных программ для управления данными

Комплекс системных программ для управления данными в вычислительной системе с массовым параллелизмом целесообразно строить из следующих подсистем: менеджер легковесных процессов, система переда-

чи сообщений и система хранения персистентных и временных данных. Общая структура комплекса приведена на Рис. 2.1.

Менеджер легковесных процессов (нитей) обеспечивает организацию процессов в виде нитей управления и предоставляет средства для диспетчеризации и синхронизации нитей.

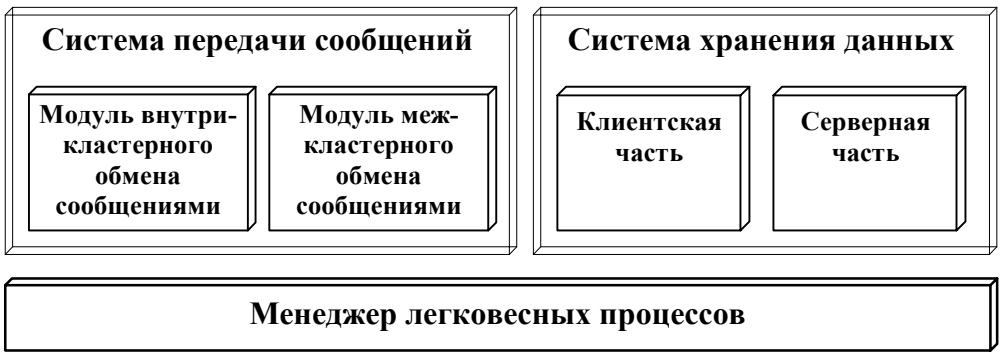


Рис. 2.1 Структура программного комплекса для управления данными

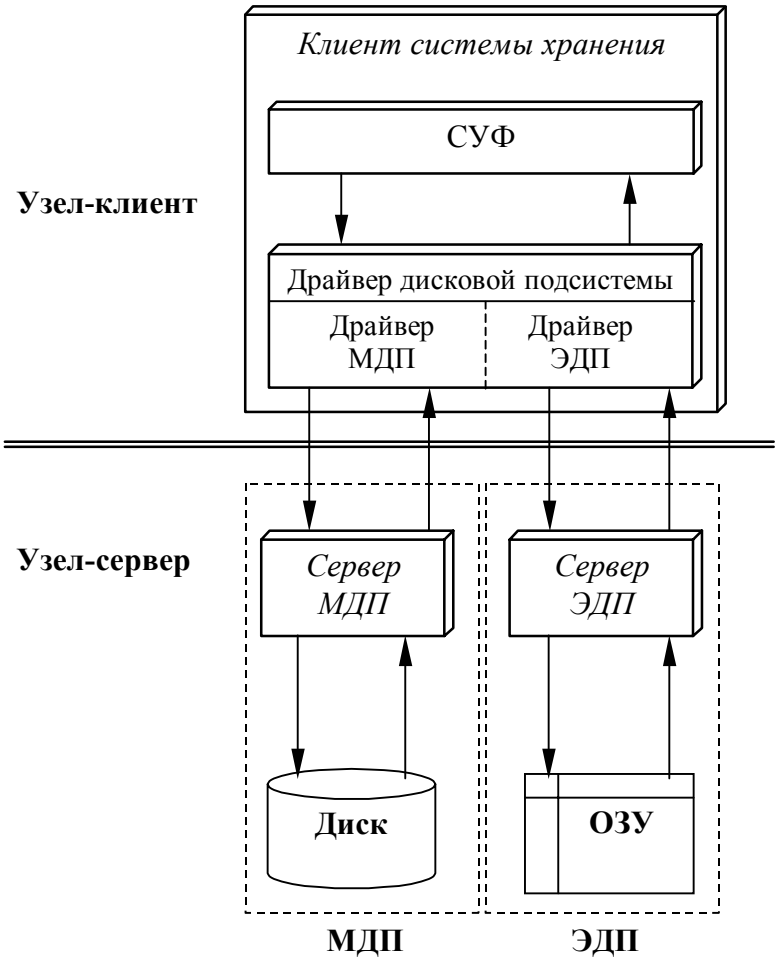
Система передачи сообщений обеспечивает средства обмена сообщениями между любыми двумя узлами вычислительной системы. В соответствии с иерархическим подходом к организации системы управления данными, предложенным в главе 1, система передачи сообщений строится из двух подсистем: модуля межкластерного обмена сообщениями и модуля внутрикластерного обмена сообщениями.

Модуль межкластерного обмена сообщениями обеспечивает передачу сообщений между узлами, принадлежащими различным процессорным кластерам. *Модуль внутрикластерного обмена сообщениями* обеспечивает средства для передачи сообщений в пределах одного процессорного кластера.

Система хранения персистентных и временных данных обеспечивает унифицированный интерфейс обработки персистентных и временных данных. При этом могут использоваться следующие устройства хранения данных или их комбинации: жесткий диск host-компьютера, модуль дисковой подсистемы и электронная дисковая подсистема. *Электронная диско-*

вая подсистема реализуется на базе типового процессорного модуля путем использования его оперативной памяти в качестве хранилища данных.

Архитектура системы хранения персистентных и временных данных приведена на Рис. 2.2. Система хранения данных строится на базе принципов архитектуры клиент-сервер [62, 96].



МДП – модуль дисковой подсистемы
ЭДП – электронная дисковая подсистема

Рис. 2.2 Архитектура системы хранения

Процессорные узлы, доступные задаче, разделяются на два непересекающихся подмножества: *узлы-клиенты* и *узлы-серверы*, а в структуре системы хранения выделяются *клиентская часть* и *серверная часть*. На узле-сервере запускается серверная часть системы хранения, которая обслуживает запросы клиентских частей, запускаемых на узлах-клиентах. В качестве узла-клиента выступает типовой процессорный модуль. В качест-

ве узла-сервера может фигурировать как типовой процессорный модуль, так и модуль дисковой подсистемы.

Клиентская часть системы хранения реализуется в виде иерархии двух подсистем: системы управления файлами и драйвера дисковой подсистемы.

Система управления файлами (СУФ) поддерживает представление данных в виде файлов. *Файл* – это последовательный набор записей одинаковой длины. *Запись* состоит из заголовка и одного информационного поля info. СУФ предоставляет следующие основные функции:

- создание и удаление файла;
- открытие и закрытие файла;
- выборка, добавление и удаление записей файла;
- организация последовательного просмотра записей файла.

Файл реализуется как набор страниц диска. *Набор страниц* представляет собой связный список страниц диска. Страница состоит из заголовка и одного информационного поля info.

СУФ обеспечивает буферизацию страниц диска. *Буферизация* заключается в выделении буферного пула фиксированного размера в оперативной памяти узла-клиента. При выполнении операции чтения страницы набора ее образ помещается в буферный пул. Операция записи страницы набора выполняет модификацию образа этой страницы в буферном пуле. Измененные образы страниц буферного пула сбрасываются на диск в некоторый определяемый системой момент. Поскольку в течение выполнения задачи возможны многократные обращения к одним и тем же страницам файла (набора), буферизация позволяет значительно сократить объем данных, передаваемых от узла-сервера к узлу-клиенту.

Драйвер дисковой подсистемы предоставляет средства для асинхронного чтения и записи страниц диска. Драйвер дисковой подсистемы инкапсулирует аппаратные особенности дисковой подсистемы. Подобный подход позволяет изолировать программный код, зависящий от устройства

конкретной дисковой подсистемы, на уровне драйвера дисковой подсистемы, и обеспечивает *аппаратную независимость СУФ*, как компоненты более высокого уровня системной иерархии. Для использования СУФ на другой аппаратной платформе необходимо разработать соответствующий драйвер дисковой подсистемы, а изменения в исходных текстах СУФ не требуются.

Драйвер дисковой подсистемы обеспечивает для СУФ унифицированный интерфейс доступа к данным, находящимся на устройстве хранения. Устройством хранения данных может служить модуль дисковой подсистемы (МДП) или электронная дисковая подсистема (ЭДП). В соответствии с этим возможны две различные реализации драйвера дисковой подсистемы: *драйвер МДП* и *драйвер ЭДП*.

Серверная часть системы хранения запускается на узле-сервере и обрабатывает запросы клиентских частей на чтение-запись данных, хранящихся на дисках.

Сервер системы хранения обслуживает очередь запросов клиентов на чтение и запись страниц диска. Для повышения скорости обработки запросы клиента на чтение и запись разных страниц диска должны выполняться сервером асинхронно, то есть независимо от порядка поступления запросов от клиента. Для поддержания непротиворечивости и целостности данных, хранящихся на диске, запросы клиента на чтение и запись одной и той же страницы диска должны выполняться сервером синхронно, то есть в порядке поступления запросов от клиента.

Система хранения предполагает *две различных реализации узла-сервера*: на базе модуля дисковой подсистемы и на базе электронной дисковой подсистемы.

В случае реализации узла-сервера *на базе модуля дисковой подсистемы (МДП)* в вычислительную систему встраивается МДП, к которому по SCSI интерфейсу подключаются дисковые накопители. В данном слу-

чае серверная часть системы хранения представлена сервером МДП, который запускается на коммуникационном процессоре МДП.

В случае реализации узла-сервера *на базе электронной дисковой подсистемы (ЭДП)* в качестве узла-сервера фигурирует типовой процессорный модуль, оперативная память которого используется в качестве хранилища данных. В данном случае серверная часть системы хранения представлена сервером ЭДП, который запускается на вычислительном процессоре узла-сервера.

Поскольку сервер МДП и сервер ЭДП предполагают запуск на различных аппаратных компонентах вычислительной системы, они должны иметь различную реализацию. В то же время в рамках одной задачи одновременно могут использоваться как серверы МДП, так и серверы ЭДП.

ГЛАВА 3. СТРУКТУРА ПРОГРАММНОГО КОМПЛЕКСА ОМЕГА ДЛЯ УПРАВЛЕНИЯ ДАННЫМИ В МНОГОПРОЦЕССОРНОЙ ВЫЧИСЛИТЕЛЬНОЙ СИСТЕМЕ МВС-100

3.1 Структура комплекса Омега

В соответствии с подходами, предложенными во второй главе, нами был разработан комплекс системных программ для управления данными в многопроцессорной вычислительной системе МВС-100, получивший название Омега. Структура комплекса изображена на Рис. 3.1.

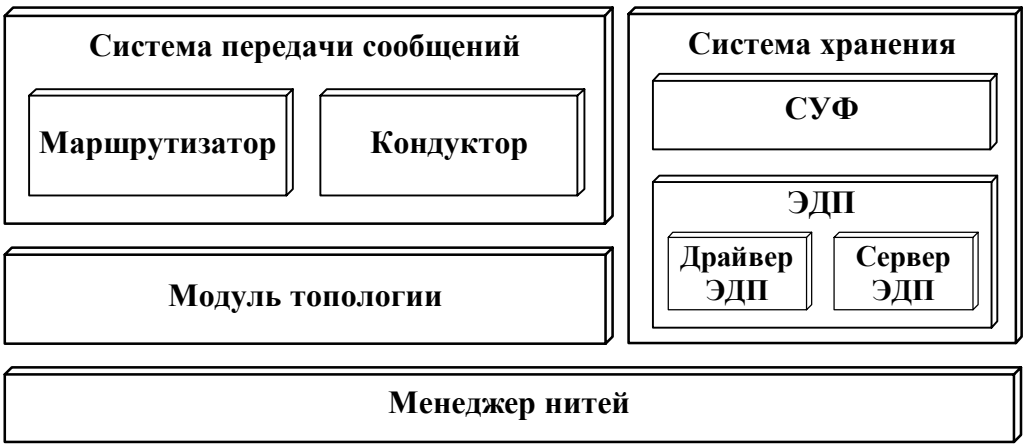


Рис. 3.1 Структура комплекса Омега

Программный комплекс Омега включает в себя менеджер нитей, модуль топологии, систему передачи сообщений и систему хранения данных.

Менеджер нитей обеспечивает поддержку легковесных процессов (нитей управления), позволяющих эффективно выполнять на одном процессорном узле несколько задач в режиме разделения времени. Нити используются в реализации других подсистем комплекса Омега. Принципы реализации менеджера нитей рассматриваются в разделе 3.2.

Модуль топологии инкапсулирует аппаратные особенности топологии МВС и определяет его как систему, состоящую из фиксированного

числа однотипных процессорных кластеров. Реализация модуля топологии рассматривается в разделе 3.3.

Система передачи сообщений обеспечивает средства асинхронного обмена данными между любыми двумя процессорными узлами вычислительной системы. Система передачи сообщений состоит из двух подсистем: кондуктора и маршрутизатора. *Кондуктор* обеспечивает передачу сообщений между узлами одного процессорного кластера. *Маршрутизатор* обеспечивает передачу сообщений на между узлами разных процессорных кластеров. Принципы реализации системы передачи сообщений изложены в разделе 3.4.

Система хранения обеспечивает унифицированный интерфейс обработки персистентных и временных данных на базе следующих устройств хранения: жесткий диск host-компьютера, модуль дисковой подсистемы и электронная дисковая подсистема. Система хранения состоит из двух подсистем: системы управления файлами и электронной дисковой подсистемы. *Система управления файлами (СУФ)* поддерживает понятие файла, как именованного набора записей фиксированной длины. СУФ предоставляет стандартные функции по работе с файлами: создание и удаление файла, открытие и закрытие файла, выборка, добавление и удаление записей файла и организация последовательного просмотра записей файла. *Электронная дисковая подсистема (ЭДП)* реализует виртуальный модуль дисковой подсистемы на базе типового процессорного модуля, оперативная память которого используется в качестве хранилища данных. ЭДП предоставляет драйвер ЭДП и сервер ЭДП. СУФ использует функции *драйвера ЭДП* для формирования запросов на чтение-запись данных, хранящихся на ЭДП; *сервер ЭДП* обеспечивает обработку этих запросов. Методы реализации системы хранения детально рассматриваются в главе 4.

3.2 Менеджер нитей

Менеджер нитей обеспечивает поддержку легковесных процессов (нитей управления), позволяющих эффективно выполнять на одном процессорном узле несколько задач в режиме разделения времени. Нити используются в реализации других подсистем комплекса Омега.

Основные константы, типы и функции, экспортируемые менеджером нитей, приведены на Рис. 3.2.

```
/* Указатель на фактор-функцию. */
typedef th_factor_t (*th_factorfn_t ) (void);

/* Указатель на функцию, представляющую тело нити. */
typedef int (*th_proc_t) (void*);

/* Инициализация менеджера нитей.
my_number – абсолютный номер собственного узла */
int th_init(int my_number);

/* Диспетчеризация нитей. */
void th_schedule(void);

/* Порождение новой нити.
proc – указатель на функцию, представляющую тело нити
param – указатель на список параметров, или NULL
factorfn – указатель на фактор-функцию, или NULL
type – тип нити
nice - приоритет нити (целое число от -20 до 20) */
void th_fork(th_proc_t proc, void * param, th_factorfn_t factorfn, th_type_t type, int nice);

/* Приостановка выполнения нити.
tid – идентификатор нити */
void th_suspend(int tid);

/* Возобновление выполнения нити.
tid – идентификатор нити */
void th_resume(int tid);
```

Рис. 3.2 Интерфейс менеджера нитей

Синхронизация и диспетчеризация нитей реализуется на базе модели "производитель-потребитель", предложенной в работе [51]. В данной модели каждый процесс рассматривается как корневая нить (на одном процессорном модуле МВС-100 может быть запущен только один процесс). С помощью вызова функции менеджера нитей **th_fork** корневая нить может породить произвольное число нитей, которые будут считаться *дочерними* по отношению к ней. Любая дочерняя нить аналогичным образом может породить произвольное число нитей, которые будут считаться дочерними

по отношению к данной нити. Таким образом, нити образуют иерархию, поддерживаемую менеджером нитей.

С каждой нитью связывается числовая функция, выдающая значения в интервале [0;1] и называемая *фактор-функцией*. Способ вычисления фактор-функции определяется пользователем при создании нити. Значение, выдаваемое фактор-функцией в некоторый момент времени, называется *Т-фактором* нити в данный момент времени. Т-факторы используются менеджером нитей для управления процессом диспетчеризации и синхронизации нитей.

Понятие Т-фактора интерпретируется на основе использования парадигмы производитель-потребитель. Предполагается, что каждая нить производит некоторые объекты, называемые *гранулами*. с каждой нитью связывается некоторый буфер конечной длины, организуемый как очередь (FIFO) и называемый *складом*. Когда нить производит новую гранулу, она помещает ее в свой склад. В данном контексте Т-фактор обозначает процент заполнения склада. При значении Т-фактора, равном единице, что соответствует полностью заполненному складу, нить переводится менеджером нитей в неактивное состояние. Как только значение ее Т-фактора становится меньше единицы, то есть в складе появляется свободное место, нить снова переводится в активное состояние.

Нить, являющаяся листом в дереве нитей, может производить гранулы вне зависимости от других нитей. Однако нить, имеющая дочерние нити, может произвести собственную гранулу, только *потребив* (забрав из склада) одну или более гранул, произведенных дочерними нитями. В этом отношении нити делятся на два класса: дизъюнктивные и конъюнктивные. *Конъюнктивной нити* для производства одной гранулы необходимо потребить по одной грануле от каждой дочерней нити. *Дизъюнктивной нити* для производства одной гранулы необходимо потребить только одну гранулу от любой из дочерних нитей. В соответствие с этой моделью менеджер нитей переводит конъюнктивную нить в неактивное состояние, если

хотя бы одна из дочерних нитей имеет нулевой Т-фактор. Дизъюнктивная нить переводится в неактивное состояние, если все ее дочерние нити имеют нулевой Т-фактор.

Для синхронизации и диспетчеризации нитей менеджер нитей в ходе каждого цикла диспетчеризации вычисляет Т-факторы для всех нитей. В соответствии с описанной выше дисциплиной, нити переводятся в пассивное или активное состояние. Управление получает активная нить, имеющая минимальное значение *динамического приоритета*. Обратная передача управления менеджеру нити выполняется явно путем вызова функции **th_schedule** после производства нитью очередной гранулы. Более детальное описание работы менеджера нитей и формулы вычисления динамического приоритета приводятся в работе [105].

3.3 Модуль топологии

Модуль топологии инкапсулирует аппаратные особенности топологии МВС-100. Основные константы и функции, экспортируемые модулем топологии, представлены на Рис. 3.3.

```
/* Общее число процессорных узлов */
#define TP_CPU_QTY                  8

/* Общее число процессорных кластеров */
#define TP_CLUSTER_QTY              2

/* Число узлов в процессорном кластере */
#define TP_NODES_IN_CLUSTER         4

/* Инициализация модуля топологии.
my_number – абсолютный номер собственного узла */
void tp_init(int my_number);

/* Вычисление абсолютного адреса процессорного узла
cluster – номер процессорного кластера.
node – номер узла в процессорном кластере
Возвращает неотрицательный абсолютный номер процессорного узла или отрицательный код ошибки. */
int tp_absaddr(int cluster, int node);

/* Вычисление номера собственного узла */
int tp_mynode(void);

/* Вычисление номера собственного процессорного кластера */
int tp_mycluster(void);
```

Рис. 3.3 Интерфейс модуля топологии

В модуле топологии определяется общее число процессорных узлов, доступных задаче, число процессорных кластеров и число узлов в кластере, и предоставляются функции для перевода физических номеров процессорных узлов в вычислительной системе в логические номера узлов в процессорных кластерах и наоборот.

3.4 Система передачи сообщений

Система передачи сообщений обеспечивает средства асинхронного обмена данными между любыми двумя процессорными узлами вычислительной системы. В соответствии с иерархической организацией системы Омега передача сообщений имеет два независимых уровня: внутрикластерный уровень и межкластерный уровень.

Процессорный кластер имеет стандартную предопределенную архитектуру с фиксированным количеством процессорных узлов, объединенных в единую сеть с высокой степенью связности (длина кратчайшего пути между любыми двумя узлами сети не должна превышать значения 2). На *внутрикластерном уровне* между процессорными узлами имеет место интенсивный обмен данными. На *межкластерном уровне* обмены данными между процессорными узлами полагаются имеющими относительно низкую интенсивность. Два уровня передачи сообщений изображены на Рис. 3.4.

В соответствии с этим система передачи сообщений состоит из двух подсистем: маршрутизатора и кондуктора.

Маршрутизатор обеспечивает передачу сообщений на межкластерном уровне. Реализация маршрутизатора рассматривается в разделе 3.4.1.

Кондуктор обеспечивает передачу сообщений на внутрикластерном уровне. В реализации кондуктора по существу используется знание предопределенной архитектуры процессорного кластера. Ограничения, накладываемые на топологию межпроцессорных соединений кластера, делают возможным разработку протоколов обмена данными [104, 105], которые

позволяют реализовать межпроцессорные обмены между узлами процессорного кластера более эффективно, чем в ОС Router. Реализация кондуктора рассматривается в разделе 3.4.2.

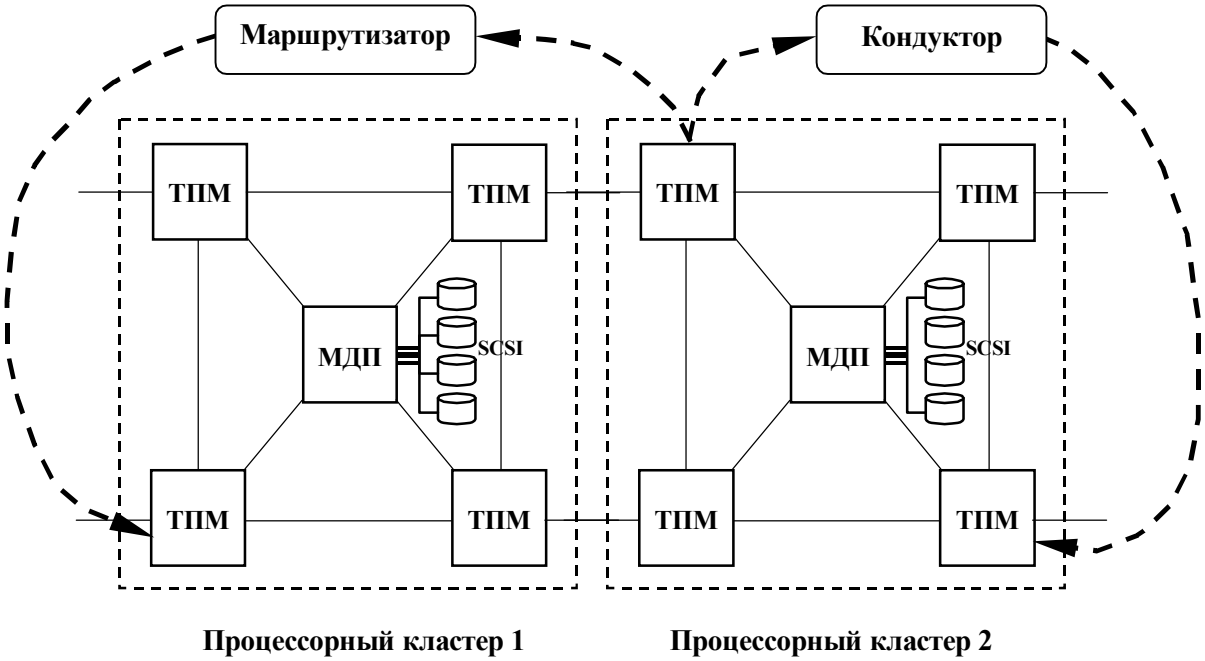


Рис. 3.4 Двухуровневая система передачи сообщений

3.4.1 Маршрутизатор

Маршрутизатор обеспечивает асинхронную передачу сообщений между узлами, принадлежащими различным процессорным кластерам. Интерфейс маршрутизатора представлен на Рис. 3.5.

Маршрутизатор поддерживает любое количество асинхронных виртуальных каналов (для каждого обмена создается свой канал) между любыми двумя процессорными узлами. Для идентификации каналов используется понятие порта: канал, соединяющий два узла, имеет одинаковые номера портов с обеих сторон.

Реализация маршрутизатора основана на протоколе обмена сообщениями, предложенном в работе [104]. Данный протокол обмена сообщениями характеризуется тем, что инициализация передачи сообщения от одного узла другому требует только двух элементарных обменов (элемен-

тарный обмен примерно эквивалентен передаче одного байта информации). В реализации передачи сообщений посредством каналов используется копирование данных типа "память-в-память" и динамическое выделение памяти. Однако это не сказывается на производительности маршрутизатора критическим образом, так как межкластерные обмены полагаются относительно редкими.

```
/* Тип канала маршрутизатора. */
typedef enum {
        RT_READ = 0, /* канал по чтению */
        RT_WRITE = 1 /* канал по записи */
} rt_type_t;

/* Уникальный идентификатор канала маршрутизатора. */
typedef int rt_rid_t;

/* Инициализация маршрутизатора.
my_number – абсолютный номер собственного узла */
void rt_init(int my_number);

/* Создание нового канала.
cluster – номер кластера-реципиента
node – номер узла-реципиента в процессорном кластере
port – номер порта
type – тип канала (чтение или запись)
buffer – указатель на буфер с сообщением
length – длина сообщения
Возвращает неотрицательный уникальный идентификатор канала или отрицательный код ошибки.
Создает канал и запускает обмен в асинхронном режиме. Обмен может произойти только тогда, когда на узле-реципиенте будет запущен обмен соответствующего типа по этому же каналу. После выполнения данной функции, для реального старта обмена, необходимо выполнить операцию диспетчеризации th_schedule(). */
rt_rid_t t rt_runchannel(int cluster, int node, int port, rt_type_t type, void *buffer, int length);

/* Состояние канала.
rid – уникальный идентификатор канала
Возвращает 1, если обмен завершен и 0 в противном случае. */
int rt_testchannel(rt_rid_t rid);
```

Рис. 3.5 Интерфейс маршрутизатора

3.4.2 Кондуктор

Кондуктор обеспечивает средства для асинхронной передачи сообщений в пределах одного процессорного кластера. Интерфейс кондуктора представлен на Рис. 3.6.

```
/* Тип канала кондуктора. */
typedef enum {
        CN_READ = 0, /* канал по чтению */
        CN_WRITE = 1 /* канал по записи */
} cn_type_t;

/* Уникальный идентификатор канала кондуктора. */
typedef int cn_cid_t;

/* Инициализация кондуктора.
my_number – абсолютный номер собственного узла */
void cn_init(int my_number);

/* Создание нового канала.
node – номер узла-реципиента в процессорном кластере
port – номер порта
type – тип канала (чтение или запись)
buffer – указатель на буфер с сообщением
length – длина сообщения
Возвращает неотрицательный уникальный идентификатор канала или отрицательный код
ошибки.
Создает канал и запускает обмен в асинхронном режиме. Обмен может произойти только тогда,
когда на узле-реципиенте будет запущен обмен соответствующего типа по этому же каналу.
После выполнения данной функции, для реального старта обмена, необходимо выполнить
операцию диспетчеризации th_schedule().
*/
cn_cid_t cn_runchannel(int node, int port, cn_type_t type, void *buffer, int length);

/* Состояние канала.
cid – уникальный идентификатор канала
Возвращает 1, если обмен завершен и 0 в противном случае. */
int cn_testchannel(cn_cid_t cid);
```

Рис. 3.6 Интерфейс кондуктора

Интерфейс кондуктора близок к интерфейсу маршрутизатора, однако кондуктор использует принципиально иной внутренний протокол. В соответствии с этим протоколом инициализация передачи сообщения от одного узла другому требует трех элементарных обменов, и при передаче сообщения не использует копирование данных типа "память-в-память". Данный протокол оказывается достаточно эффективным для внутрикластерных обменов сообщениями, поскольку длина кратчайшего пути между узлами мала (не больше двух), что обеспечивает относительно низкую стоимость одного элементарного обмена. Другой особенностью реализации кондуктора является то, что при создании нового виртуального канала не требуется динамического выделения памяти.

ГЛАВА 4. МЕТОДЫ РЕАЛИЗАЦИИ СИСТЕМЫ ХРАНЕНИЯ ДАННЫХ

4.1 Структура системы хранения данных

Система хранения данных предоставляет унифицированные средства для управления персистентными и временными данными. В качестве устройств хранения данных могут использоваться жесткий диск host-компьютера, модуль дисковой подсистемы и электронная дисковая подсистема.

Структура системы хранения представлена на Рис. 4.1. Система хранения строится на базе принципов архитектуры клиент-сервер. Процессорные узлы, доступные задаче, разделяются на две группы: *узлы-клиенты* и *узлы-серверы*. На узле-сервере запускается *серверная часть* системы хранения, которая обслуживает запросы *клиентских частей*, запускаемых на узлах-клиентах. Клиентскую часть системы хранения составляют система управления файлами, драйвер дисковой подсистемы и драйвер электронной дисковой подсистемы. Серверная часть системы хранения представлена сервером модуля дисковой подсистемы и сервером электронной дисковой подсистемы.

Электронная дисковая подсистема (ЭДП) реализует виртуальный модуль дисковой подсистемы на базе типового процессорного модуля, оперативная память которого используется в качестве хранилища данных. В состав электронной дисковой подсистемы входят драйвер ЭДП и сервер ЭДП. Принципы проектирования и реализации ЭДП описываются в разделе 4.2.

Система управления файлами (СУФ) поддерживает понятие файла, как именованного набора записей фиксированной длины. Файлы используются для унифицированного представления и обработки персистентных

и временных данных. СУФ представлена иерархией следующих подсистем: менеджер наборов и менеджер файлов.

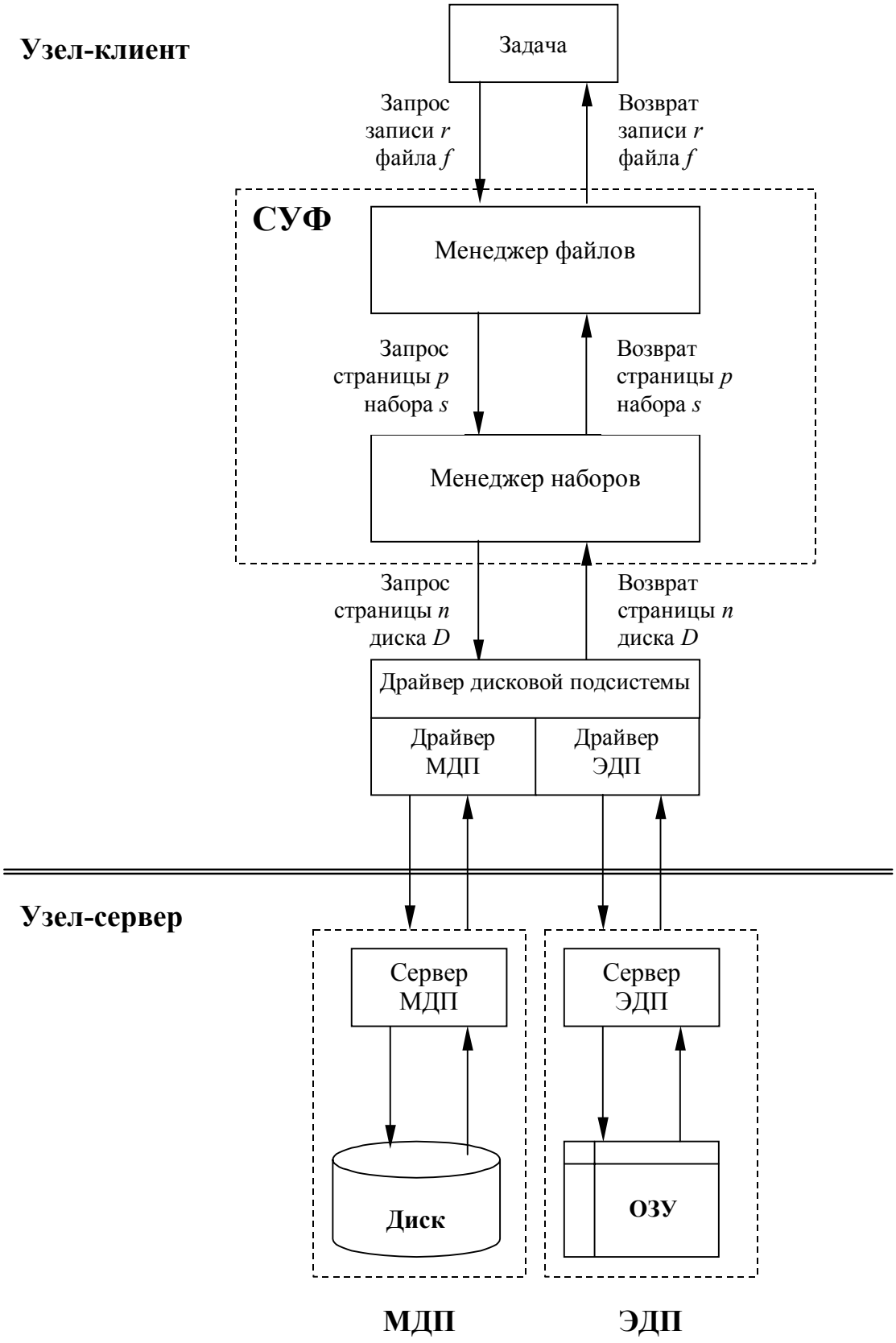


Рис. 4.1 Структура системы хранения данных

Менеджер наборов поддерживает представление данных, хранящихся на диске, в виде совокупности наборов (связных списков) страниц и обеспечивает буферизацию данных на основе единого буферного пула. Менеджер наборов использует оригинальный метод буферизации данных, описываемый в главе 5.

Менеджер файлов поддерживает представление данных в виде файлов и обеспечивает стандартные функции для управления файлами: создание, удаление, открытие и закрытие файла, выборка, добавление и удаление записей, организация последовательного просмотра записей файла и др.

Принципы проектирования и реализации СУФ детально описываются в разделе 4.3.

4.2 Электронная дисковая подсистема

Электронная дисковая подсистема (ЭДП) реализует виртуальный модуль дисковой подсистемы на базе типового процессорного модуля, оперативная память которого используется в качестве хранилища данных. В состав электронной дисковой подсистемы входят драйвер ЭДП и сервер ЭДП.

Драйвер ЭДП запускается на узле-клиенте и предоставляет функции асинхронного чтения и записи страниц диска электронной дисковой подсистемы. Принципы реализации драйвера ЭДП рассматриваются в разделе 4.2.2.

Сервер ЭДП запускается на узле-сервере. Сервер ЭДП обеспечивает страничную организацию электронного диска и обслуживает запросы, получаемые с узлов-клиентов, на чтение и запись страниц диска электронной дисковой подсистемы. Принципы реализации сервера ЭДП рассматриваются в разделе 4.2.3.

4.2.1 Принципы организации взаимодействия драйвера ЭДП и сервера ЭДП

Обмен данными между узлами-клиентами и узлом-сервером реализуется на основе специальных протоколов обмена сообщениями. Сообщения состоят из двух частей: заголовок и информационная часть *info*.

Заголовок сообщения – это запись с фиксированной структурой. Информационная часть не имеет структуры и представляет собой байтовый массив, длина которого совпадает с размером страницы. Передача сообщения выполняется в два этапа: сначала передается заголовок, а затем – *info*. Часть *info* может отсутствовать (не передаваться).

В реализации функций обмена сообщениями между узлом-клиентом и узлом-сервером используется техника командных тактов, предложенная в работе [105]. *Такт* представляет собой неделимую единицу выполнения операции. Между двумя тактами должна быть как минимум одна операция диспетчеризации **th_schedule** менеджера нитей. Такт с номером 0 означает, что операция еще не выполнялась. Значение такта -1 свидетельствует о завершении операции. Протокол взаимодействия узла-клиента и узла-сервера включает в себя такты клиента и соответствующие им такты сервера для синхронизации обмена данными при выполнении конкретной операции драйвера ЭДП. Протоколы взаимодействия узла-клиента и узла-сервера приводятся в Приложении.

4.2.2 Драйвер ЭДП

Драйвер ЭДП обеспечивает для СУФ унифицированный интерфейс доступа к страницам электронного диска. Интерфейс драйвера ЭДП представлен на Рис. 4.2. Драйвер ЭДП предоставляет следующие основные функции:

- *асинхронное* чтение указанной страницы диска в одностраничный буфер;

- *асинхронная* запись одностраничного буфера на указанную страницу диска;
- *синхронное* сохранение образа диска в файл на host-машине;
- *синхронная* загрузка образа диска из файла на host-машине.

```
/* Тип операции. */
typedef enum {
    DS_READ = 0, /* чтение данных с диска */
    DS_WRITE = 1 /* запись данных на диск */
} ds_optype_t;

/* Уникальный идентификатор операции (канала). */
typedef int ds_ioid_t;

/* Инициализация драйвера.
my_number – абсолютный номер собственного узла */
void ds_init(int my_number);

/* Запуск асинхронной операции чтения-записи.
page – номер страницы диска
type – тип операции (чтение или запись)
buffer – указатель на буфер с сообщением
Возвращает неотрицательный уникальный идентификатор операции или отрицательный код ошибки.
После выполнения данной функции, для реального старта операции, необходимо выполнить операцию диспетчеризации th_schedule(). */
ds_ioid_t ds_runoperation(int page, ds_optype_t type, void *buffer);

/* Состояние операции.
id – уникальный идентификатор операции
Возвращает 1, если обмен завершен и 0 в противном случае. */
int ds_testoperation(ds_ioid_t id);

/* Запуск синхронной операции сохранения образа диска в файл на host-машине.
filename – имя файла
Возвращает 0 в случае успеха или отрицательный код ошибки. */
int ds_dumpdisk(char *filename);

/* Запуск синхронной операции загрузки образа диска из файла на host-машине.
filename – имя файла
Возвращает 0 в случае успеха или отрицательный код ошибки. */
int ds_resetdisk(char *filename);
```

Рис. 4.2 Интерфейс драйвера ЭДП

Проектирование драйвера ЭДП основано на следующих принципах. Драйвер ЭДП реализует архитектуру виртуальных каналов. Это означает, что по линку, соединяющему узел-клиент и узел-сервер, параллельно может производиться любое количество операций чтения-записи.

Заголовок сообщения от узла-клиента имеет следующие поля:

- уникальный идентификатор операции в таблице операций узла-клиента;

- код операции (чтение/запись страницы или сохранение/загрузка образа диска);
- номер страницы диска (для операций чтения и записи страницы);
- символьная строка – имя файла (для операций сохранения и загрузки образа диска);
- номер узла-клиента.

При выполнении клиентом любых операций дескрипторы операций помещаются в таблицу операций. Таблица операций узла-клиента организована как очередь и имеет следующие поля:

- уникальный идентификатор операции;
- номер страницы диска (для операции чтения или записи страницы);
- символьная строка – имя файла (для операций сохранения и загрузки образа диска);
- указатель на буфер для части *info*;
- номер такта операции.

Таблица операций обрабатывается системной нитью узла-клиента. Фактор-функция системной нити такова, что системная нить активизируется только в двух следующих случаях: при завершении какой-либо асинхронной операции чтения или записи и при добавлении дескриптора в таблицу операций. Все остальное время системная нить узла-клиента находится в неактивном состоянии.

Для описания алгоритмов *реализации драйвера ЭДП* используются переменные состояния, изображенные на Рис. 4.3.

```
чтение_заголовка = FALSE;
заголовок_получен = FALSE;
операция_добавлена = FALSE;
чтение_запущено = FALSE;
запись_запущена = FALSE;
чтение_блокировано = FALSE;
запись_блокирована = FALSE;
заголовок /* буфер для приема заголовка сообщения от сервера */
tid_системной_нити /* идентификатор системной нити узла-клиента */
```

Рис. 4.3 Переменные состояния узла-клиента

Схема реализации фактор-функции системной нити драйвера ЭДП приведена на Рис. 4.4. Данная фактор-функция предполагает, что системная нить находится в неактивном состоянии, пока не происходит одно из двух событий: завершение какой-либо асинхронной операции чтения или записи по линку или добавление дескриптора новой операции в таблицу операций.

```
if (операция_добавлена) {
    операция_добавлена = FALSE;
    th_resume(tid_системной_нити);
}
if (чтение_запущено)
    if (t_read(сервер)) {
        чтение_запущено = FALSE;
        th_resume(tid_системной_нити);
    }
if (запись_запущена)
    if (t_write(сервер)) {
        запись_запущена = FALSE;
        th_resume(tid_системной_нити);
    }
return 0;
```

Рис. 4.4 Схема фактор-функции системной нити драйвера ЭДП

Схема реализации системной нити драйвера ЭДП изображена на Рис. 4.5. Системная нить перманентно выполняет следующую последовательность действий:

1) Проверка, получен ли заголовок сообщения от сервера. Если это событие произошло, изменяются значения соответствующих переменных состояния и выполняется блокирование линка по чтению.
2) Обработка таблицы операций. Для всех записей таблицы, у которых значение такта неотрицательно, значение такта увеличивается на 1, а затем выполняются действия, соответствующие такту данной операции (см. Приложение). Если это было последнее действие в такте, то его значение изменяется на –1, и запись удаляется из таблицы операций.
3) Запуск операции чтения заголовка сообщения от сервера, в случае если предыдущая операция чтения по линку завершена, и блокирование линка по чтению.

4) Перевод нити в неактивное состояние и выполнение операции диспетчеризации.

```
чтение_заголовка = TRUE;
чтение_блокировано = TRUE;
чтение_запущено = TRUE;
r_read(сервер, заголовок);
while (TRUE) {
    if (чтение_заголовка)
        if (t_read(сервер)) {
            чтение_блокировано = FALSE;
            чтение_заголовка = FALSE;
            заголовок_получен = TRUE;
    обработать каждый элемент таблицы операций узла-клиента {
        дескриптор.такт++;
        выполнить действия клиента в соответствии с номером такта протокола;
        if (дескриптор.такт == -1) {
            удалить_дескриптор_из_таблицы_операций_клиента();
        }
    }
    if (!чтение_блокировано)
        if (t_read(сервер)) {
            чтение_блокировано = TRUE;
            чтение_заголовка = TRUE;
            чтение_запущено = TRUE;
            r_read(сервер, заголовок);
        }
    th_suspend(tid_системной_нити);
    th_schedule();
}
```

Рис. 4.5 Схема реализации системной нити драйвера ЭДП

4.2.3 Сервер ЭДП

Сервер ЭДП запускается на узле-сервере и обслуживает запросы, получаемые с узлов-клиентов, на чтение и запись данных. Сервер ЭДП обеспечивает представление диска электронной дисковой подсистемы в виде набора пронумерованных страниц фиксированного размера. Размер страницы является параметром компиляции. По умолчанию размер страницы равен 4 Кбайт. Интерфейс сервера ЭДП приведен на Рис. 4.6.

Сервер ЭДП формирует и обрабатывает очередь запросов от узлов-клиентов на чтение и запись страниц электронного диска. Для повышения быстродействия работы сервера запросы на чтение и запись *разных страниц* диска выполняются сервером *асинхронно*, то есть независимо от порядка поступления запросов. Для поддержания непротиворечивости и целостности данных, хранящихся на диске, запросы на чтение и запись *одной*

и той же страницы диска выполняются сервером *синхронно*, то есть в порядке их поступления.

```
/* Запуск операции записи данных на страницу диска.
page – номер страницы диска
buffer – указатель на буфер с данными
Возвращает неотрицательный уникальный идентификатор операции или отрицательный код ошибки. */
int r_diskwrite(int page, void *buffer);

/* Проверка окончания операции записи данных.
id – уникальный идентификатор операции
Возвращает 1, если операция завершена и 0 в противном случае. */
int t_diskwrite(int id);

/* Запуск операции чтения данных со страницы диска.
page – номер страницы диска
buffer – указатель на буфер с данными
Возвращает неотрицательный уникальный идентификатор операции или отрицательный код ошибки. */
int r_diskread(int page, void *buffer);

/* Проверка окончания операции чтения данных.
id – уникальный идентификатор операции
Возвращает 1, если операция завершена и 0 в противном случае. */
int t_diskread(int id);

/* Сохранение образа диска в файл на host-машине.
filename – имя файла
Возвращает 0 в случае успеха или отрицательный код ошибки. */
int disk_dump (char *filename);

/* Загрузка образа диска из файла на host-машине.
filename – имя файла
Возвращает 0 в случае успеха или отрицательный код ошибки. */
int disk_reset (char *filename);
```

Рис. 4.6 Интерфейс сервера ЭДП

Проектирование сервера ЭДП основано на следующих принципах. Сервер ЭДП оформляется как самостоятельный процесс, обрабатывающий таблицу операций на стороне электронной дисковой подсистемы. Таблица операций сервера организована как очередь и имеет следующие поля:

- уникальный идентификатор операции клиента;
- номер узла-клиента;
- номер страницы диска (для операции чтения или записи страницы);
- символьная строка – имя файла (для операций сохранения и загрузки образа диска);
- указатель на буфер для части *info*;
- номер такта операции;

- номер дескриптора следующей операции клиента с этой же страницей диска или NIL, если таковых больше нет;
- флаг блокировки операции: TRUE – блокирована, FALSE – выполняется (данное поле необходимо для синхронизации операций чтения/записи одной и той же страницы).

Сообщение сервера состоит из двух частей: заголовка и *info*, передаваемых раздельно. Заголовок сообщения сервера имеет следующую структуру:

- уникальный идентификатор операции клиента;
- код операции, которую должен выполнить клиент.

При этом допустимы следующие два кода операции клиента:

1. Запустить чтение части *info*. В данном случае в качестве части *info* сервер передает содержимое соответствующей страницы диска.
2. Пометить операцию как завершенную. В данном случае часть *info* не передается.

Для описания алгоритмов *реализации сервера ЭДП* используются переменные состояния, изображенные на Рис. 4.7.

```
чтение_страницы = FALSE;
запись_страницы = FALSE;
сохранение_диска = FALSE;
загрузка_диска = FALSE;
чтение_заголовка[число_клиентов] = {FALSE};
чтение_блокировано[число_клиентов] = {FALSE};
запись_блокирована[число_клиентов] = {FALSE};
заголовок[число_клиентов] /* буфер для приема заголовков сообщений от клиентов */
```

Рис. 4.7 Переменные состояния сервера ЭДП

Схема реализации процесса сервера изображена на Рис. 4.8. Реализация процесса сервера близка к реализации системной нити драйвера ЭДП.

Процесс сервера ЭДП перманентно выполняет следующую последовательность действий:

```
for (I=0; I<число_клиентов; I++) {
    чтение_заголовка[i] = TRUE;
    чтение_блокировано[i] = TRUE;
    r_read(i, заголовок[i])
}
while (TRUE) {
    for (i=0; i<число_клиентов; i++) {
        if (чтение_заголовка[i])
            if (t_read(i)) {
                чтение_заголовка[i] = FALSE;
                чтение_блокировано[i] = FALSE;
                добавить_дескриптор_в_таблицу_операций_сервера(заголовок[i].код_операции);
                if (заголовок[i].код_операции == "чтение_страницы" || заголовок[i].код_операции ==
"запись_страницы")
                    добавить_дескриптор_в_очередь_операций_к_странице();
            }
    }
    обработать каждый элемент таблицы операций сервера {
        if (дескриптор.операция_блокирована)
            continue;
        дескриптор.такт++;
        выполнить действия сервера в соответствии с номером такта протокола;
        if (дескриптор.такт == -1) {
            удалить_дескриптор_из_таблицы_операций_сервера();
            if (дескриптор.код_операции == "чтение_стр" || дескриптор.код_операции ==
"запись_стр")
                удалить_дескриптор_из_очереди_операций_к_странице();
        }
    }
    for (i=0; i<число_клиентов; i++) {
        if (!чтение_блокировано[i])
            if (t_read(i)) {
                чтение_блокировано[i] = TRUE;
                чтение_заголовка[i] = TRUE;
                r_read(i, заголовок[i]);
            }
    }
}
```

Рис. 4.8 Схема реализации процесса сервера ЭДП

1) Для каждого клиента: проверка, получен ли заголовок сообщения от данного клиента. Если это событие произошло, изменяется значение соответствующей переменной состояния, выполняется блокирование соответствующего линка по чтению и добавляется новая запись в таблицу операций сервера. Обновляются очереди операций чтения-записи одной и той же страницы. Для всех элементов данных очередей, начиная со второго, флаг блокировки операции устанавливается в значение TRUE.
2) Обработка таблицы операций сервера. Для всех записей таблицы, у которых значение такта неотрицательно и флаг блокировки операции

имеет значение FALSE, значение такта увеличивается на 1, а затем выполняются действия, соответствующие такту данной операции (см. Приложение). Если это было последнее действие в такте, то его значение изменяется на –1, и запись удаляется из таблицы операций сервера и из очереди операций чтения-записи одной и той же страницы.

3) Запуск операции чтения заголовков сообщения от узлов-клиентов в случае, если предыдущие операции чтения по линкам завершены, и отсутствует блокирование линков по чтению.

4.3 Система управления файлами

Основным назначением системы управления файлами (СУФ) является поддержка понятия файла, как именованного набора записей фиксированной длины. Файлы используются для унифицированного представления и обработки персистентных и временных данных. СУФ представлена иерархией двух подсистем: менеджер наборов и менеджер файлов.

Менеджер наборов поддерживает представление данных, хранящихся на диске, в виде совокупности наборов (связных списков) страниц и обеспечивает буферизацию данных на основе единого буферного пула. Принципы проектирования и реализации менеджера наборов описаны в разделе 4.3.1.

Менеджер файлов поддерживает представление данных в виде файлов и обеспечивает стандартные функции для управления данными: создание, удаление, открытие и закрытие файла, выборка, добавление и удаление записей, организация последовательного просмотра записей файла. Принципы проектирования и реализации менеджера файлов описаны в разделе 4.3.2.

4.3.1 Менеджер наборов

Менеджер наборов обеспечивает представление данных, хранящихся на диске, в виде совокупности наборов страниц. Набор страниц представ-

ляет собой связный список страниц. Страница состоит из заголовка и одного информационного поля *info*. Менеджер наборов позволяет создавать и удалять наборы страниц, добавлять страницы в набор, удалять страницы из набора, осуществлять последовательный просмотр страниц набора, а также прямую выборку и выборку с упреждением страницы с указанным номером. Менеджер наборов обеспечивает буферизацию данных на основе единого буферного пула. Менеджер наборов использует оригинальный метод буферизации данных, описываемый в главе 5.

Интерфейс менеджера наборов представлен на Рис. 4.9. Доступ к содержимому страницы обеспечивается специальным оператором *выборки страницы* **pg_fetch**, который загружает страницу в буфер, если она там отсутствует, и выдает указатель на образ страницы в буфере.

Если в процессе работы с содержимым страницы прямо или косвенно вызывалась операция диспетчеризации **th_schedule** менеджера нитей, то перед продолжением работы с образом страницы необходимо вторично выполнить операцию **pg_fetch**. Это связано с тем, что во время передачи управления другой нити могло произойти откребление и вытеснение из буфера образа данной страницы в результате выполнения "параллельного" **pg_fetch**.

Оператор **pg_fetch** является синхронным в том смысле, что он возвращает управление только после того, как страница загружена в буфер (с возможным предшествующим вытеснением какой-либо другой страницы).

Если после загрузки страницы в буфер заранее известно, какую страницу придется читать следующей (а в случае баз данных это, как правило, известно), можно воспользоваться оператором *выборки с упреждением* **pg_prefetch.** Данный оператор осуществляет загрузку страницы в буфер в асинхронном режиме. Перед доступом к странице, после оператора **pg_prefetch**, должен быть выполнен оператор **pg_fetch** для того, чтобы гарантировать завершение загрузки страницы в буфер.

```
/* Идентификатор хранимого набора. */
typedef int pg_set_t;

/* Идентификатор открытого набора. */
typedef int pg_openset_t;

/* Инициализация менеджера наборов.
my_number – абсолютный номер собственного узла */
void pg_init(int my_number);

/* Создание пустого набора страниц.
set – идентификатор хранимого набора
Возвращает 0 в случае успеха или отрицательный код ошибки. */
int pg_screate(pg_set_t set);

/* Удаление хранимого набора страниц
set – идентификатор хранимого набора
Возвращает 0 в случае успеха или отрицательный код ошибки. */
int pg_sdrop(pg_set_t set);

/* Открытие хранимого набора страниц.
set – идентификатор хранимого набора
rating – статический рейтинг страниц набора
Возвращает идентификатор соответствующего открытого набора страниц или отрицательный
код ошибки. */
pg_openset_t pg_sopen(pg_set_t set, int rating);

/* Закрытие набора страниц.
openset – идентификатор открытого набора
Возвращает 0 в случае успеха или отрицательный код ошибки. */
int pg_sclose(pg_openset_t openset);

/* Добавление страницы в конец открытого набора.
openset – идентификатор открытого набора
Возвращает номер страницы на диске в случае успеха или отрицательный код ошибки. */
int pg_append(pg_openset_t openset);

/* Удаление страницы из хранимого набора.
set – идентификатор хранимого набора
page – номер страницы на диске
Возвращает 0 в случае успеха или отрицательный код ошибки. */
int pg_delete(pg_set_t set, int page);

/* Выборка страницы.
openset – идентификатор открытого набора
page – номер страницы на диске или NIL
Возвращает указатель на образ страницы в буфере в случае успеха или NULL в случае
нехватки памяти. */
void *pg_fetch(pg_openset_t openset, int page);

/* Выборка страницы с упреждением.
openset – идентификатор открытого набора
page – номер страницы на диске или NIL
Возвращает неотрицательное число в случае успеха или отрицательный код ошибки. */
int pg_prefetch(pg_openset_t openset, int page);
```

Рис. 4.9 Интерфейс менеджера наборов

Открытие набора страниц осуществляется с помощью операции **pg_sopen.** Для каждого открытого набора вводится атрибут *текущая страница.* При открытии набора данный атрибут устанавливается в значение **NIL**, что соответствует пустой ссылке.

Оператор выборки страницы **pg_fetch** устанавливает указатель текущей страницы на указанную страницу. Данная страница набора *закрепляется* в буфере, то есть она не может быть вытеснена из буфера ни при каких условиях. При выполнении следующего оператора **pg_fetch** над данным набором указатель текущей страницы переустанавливается на новую страницу. При этом предыдущая текущая страница набора *открепляется* от буфера, то есть она при необходимости может быть вытеснена из буфера.

С каждой страницей открытого набора, находящейся в буфере, связывается некоторое значение, называемое *рейтингом страницы*. Различаются статический и динамический рейтинги. *Статический рейтинг* задается пользователем при загрузке страницы в буфер. *Динамический рейтинг* является функцией от статического рейтинга и вычисляется менеджером наборов. Значение динамического рейтинга страницы может изменяться со временем. *Суммарный рейтинг* страницы определяется как сумма статического и динамического рейтингов страницы. Если возникает необходимость освободить в буфере место для новой страницы, то из буфера вытесняется страница с наименьшим суммарным рейтингом. Подробно метод вытеснения страниц из буферного пула, примененный в реализации менеджера страниц, описывается в главе 5.

Реализация менеджера наборов строится как иерархия модулей и объектов, изображенных на Рис. 4.10.

- BUF (Buffer) – буфер для буферизации страниц;
- DIR (Directory) – избыточный индекс буферизованных страниц (индекс для BUF);
- LIST – список свободного пространства диска;
- DD (Disk Directory) – заголовок диска;
- SAT (Set Allocation Table) –таблица размещения наборов;
- SET (Open Set Descriptor Table) – таблица дескрипторов открытых наборов.

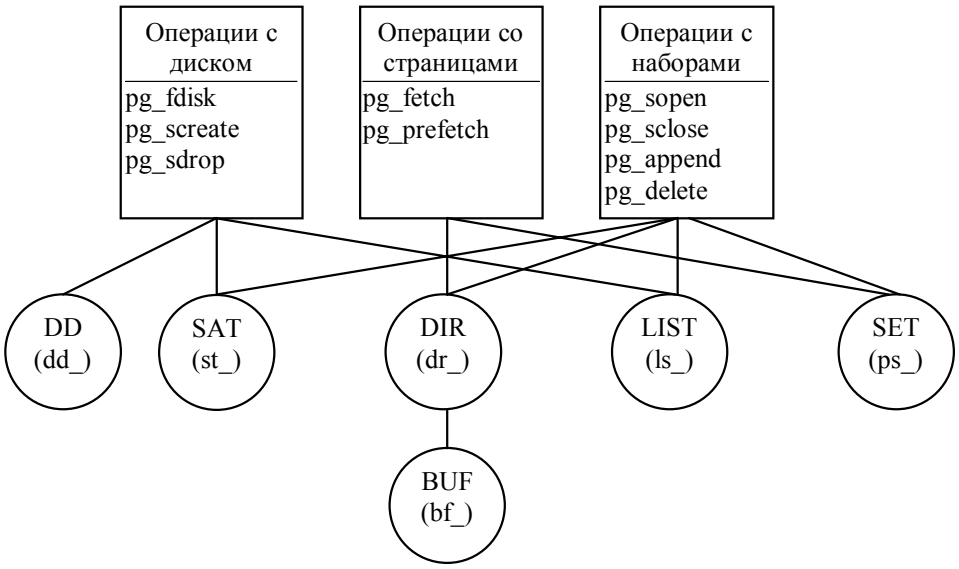


Рис. 4.10 Иерархия модулей и объектов менеджера наборов

Объект BUF (менеджер буферного пула) обеспечивает резервирование и освобождение непрерывных участков памяти в буферном пуле. Минимальной единицей выделения памяти является блок размером 4 Кб. Интерфейс объекта BUF представлен на Рис. 4.11. Основные операции менеджера буферного пула – выделить блок в буферном пуле и вернуть блок в буферный пул.

```
/* Выделение блока в буферном пуле.
Возвращает указатель на блок памяти в буферном пуле в случае успеха или NULL в случае
отсутствия свободных блоков. */
void *bf_alloc(void);

/* Освобождение блока памяти.
block – указатель на блок */
void bf_free(void *block);
```

Рис. 4.11 Интерфейс объекта BUF

Объект DIR реализует избыточный индекс буферного пула. Избыточность понимается в том смысле, что индекс DIR содержит количество позиций, превышающее количество страниц, которое может быть размещено в буферном пуле. Данная избыточность по существу используется в реализации метода **pg_prefetch**. Кроме того, избыточность индекса DIR может быть использована при определении стратегии вытеснения, путем анализа "истории" загрузки страниц в буфер. Индекс DIR используется также для организации асинхронных операций загрузки, сохранения и за-

мещения образа страницы в буферном пуле. Интерфейс объекта DIR приведен на Рис. 4.12.

```
/* Вычисление индекса элемента в таблице DIR.
openset – идентификатор открытого набора
page – номер страницы набора на диске
Возвращает индекс элемента в случае успеха или отрицательный код ошибки. */
int dr_find(pg_openset_t openset, int page);

/* Вставка нового элемента в таблицу DIR.
openset – идентификатор открытого набора
page – номер страницы набора на диске
Возвращает индекс нового элемента в случае успеха или отрицательный код ошибки. */
int dr_insert(pg_openset_t openset, int page);

/* Поиск "жертвы".
Возвращает индекс жертвы в случае успеха или отрицательный код ошибки. */
int dr_victim(void);
```

Рис. 4.12 Интерфейс объекта DIR

Метод **dr_insert** добавляет в DIR новый элемент с указанными значениями атрибутов. Если в DIR нет свободных позиций, то производится поиск неиспользуемого элемента с минимальным суммарным рейтингом. Новый элемент вставляется на место найденной "жертвы".

Метод **dr_victim** производит поиск в DIR элемента с наименьшим суммарным рейтингом среди элементов, указывающих на непустую страницу буфера, которая в настоящий момент не используется либо закончена выборка с упреждением этой страницы. Если такие элементы в DIR отсутствуют, то операция завершается с кодом ошибки "не хватает памяти".

Объект LIST реализует функции обработки списка свободного пространства. Объект LIST обеспечивает выделение непрерывных блоков страниц и утилизацию освобожденных страниц на диске. Интерфейс объекта LIST приведен на Рис. 4.13.

```
/* Выделение страницы из списка свободного пространства.
Возвращает номер страницы на диске в случае успеха или отрицательный код ошибки. */
int ls_alloc(void);

/* Возвращение страницы в список свободного пространства.
page – номер страницы набора на диске
Возвращает 0 в случае успеха или отрицательный код ошибки. */
int ls_free(int page);

/* Возвращение всех страниц диска в список свободного пространства. */
void ls_clear(void);
```

Рис. 4.13 Интерфейс объекта LIST

Объект DD (Disk Directory) предоставляет методы для работы с заголовком диска. Заголовок диска занимает нулевую страницу диска. В заголовке диска хранится список свободного пространства, таблица размещения наборов и другая служебная информация. Интерфейс объекта DD представлен на Рис. 4.14. Основные методы объекта DD – загрузить заголовок диска в оперативную память и сохранить изменения образа заголовка в оперативной памяти на диск.

```
/* Загрузка заголовка диска в оперативную память.
Возвращает указатель на образ заголовка в памяти или NULL в случае неудачи. */
void *dd_load(void);

/* Сохранение изменений образа заголовка диска.
Возвращает 0 в случае успеха или отрицательный код ошибки. */
int dd_save(void);
```

Рис. 4.14 Интерфейс объекта DD

Объект SAT (Set Allocation Table) реализует таблицу размещения наборов. Данная таблица хранится в заголовке диска и считывается в оперативную память каждый раз при инициализации менеджера наборов. Элементы таблицы SAT содержат следующую информацию: идентификатор набора, идентификатор первой страницы набора, идентификатор последней страницы набора и другую служебную информацию. Конкретным представлением объекта SAT является статический массив.

Объект SET (Open Set Descriptor Table) реализует таблицу дескрипторов открытых наборов. Таблица SET представляется в виде статического массива записей, организованного в виде хеш-таблицы. Для разрешения коллизий используется метод открытой адресации [20].

Алгоритмы, реализующие методы объектов менеджера наборов, детально описаны в работе [53].

4.3.2 Менеджер файлов

Менеджер файлов обеспечивает представление данных в виде совокупности файлов. Файл – это последовательный набор записей одинаковой длины. Запись состоит из заголовка и одного информационного поля *info*.

Интерфейс менеджера файлов изображен на Рис. 4.15.

```
/* Идентификатор хранимого файла. */
typedef pg_set_t fl_file_t;

/* Идентификатор открытого файла. */
typedef pg_openset_t fl_FILE_t;

/* УИД записи */
typedef struct {
    int page; /* номер страницы */
    int rec; /* номер записи на странице */
} fl_rid_t;

/* Инициализация менеджера файлов.
my_number – абсолютный номер собственного узла */
void fl_init(int my_number);

/* Создание пустого файла.
file – идентификатор хранимого файла
length – длина записи
Возвращает 0 в случае успеха или отрицательный код ошибки. */
int fl_fcreate(fl_file_t file, int length);

/* Удаление хранимого файла.
file – идентификатор хранимого файла
Возвращает 0 в случае успеха или отрицательный код ошибки. */
int fl_fdrop(fl_file_t file);

/* Открытие хранимого файла.
file – идентификатор хранимого файла
rating – статический рейтинг страниц файла
Возвращает идентификатор соответствующего открытого файла или отрицательный код
ошибки. */
fl_FILE_t fl_fopen(fl_file_t file, int rating);

/* Закрытие файла.
f – идентификатор открытого файла
Возвращает 0 в случае успеха или отрицательный код ошибки. */
int fl_fclose(fl_FILE_t f);

/* Добавление записи в конец открытого файла.
f – идентификатор открытого файла
Возвращает указатель на УИД добавленной записи или NULL в случае неудачи. */
fl_rid_t *fl_append(fl_FILE_t f);

/* Пометка текущей записи на удаление.
f – идентификатор открытого файла
Возвращает 0 в случае успеха или отрицательный код ошибки. */
int fl_delete(fl_FILE_t f);

/* Удаление всех записей с пометкой на удаление.
file – идентификатор хранимого файла
Возвращает 0 в случае успеха или отрицательный код ошибки. */
int fl_fpack(fl_file_t file);

/* Выборка записи.
f – идентификатор открытого файла
rid – указатель на УИД записи
Возвращает указатель на запись или NULL в случае ошибки. */
void *fl_fetch(fl_FILE_t f, fl_rid_t *rid);

/* Выборка записи с упреждением.
f – идентификатор открытого файла
rid – указатель на УИД записи
Возвращает неотрицательное число в случае успеха или отрицательный код ошибки. */
int fl_prefetch(fl_FILE_t f, fl_rid_t *rid);
```

Рис. 4.15 Интерфейс менеджера файлов

Менеджер файлов предоставляет следующие основные функции:

- создание и удаление файла;
- открытие и закрытие файла;
- выборка, добавление и удаление записей файла.

Реализация файлов осуществляется с использованием техники, описанной в [75]. Каждый файл размещается в отдельном наборе страниц. Соответствие между файлами и наборами страниц хранится в FAT (File Allocation Table). FAT размещается в непрерывной области диска, начиная с нулевой страницы.

Менеджер файлов поддерживает уникальный идентификатор записи (УИД). Структура УИД приведена на Рис. 4.16. В качестве УИД используется пара (<номер страницы>, <номер байта от конца страницы, содержащий адрес первого байта записи на странице>).

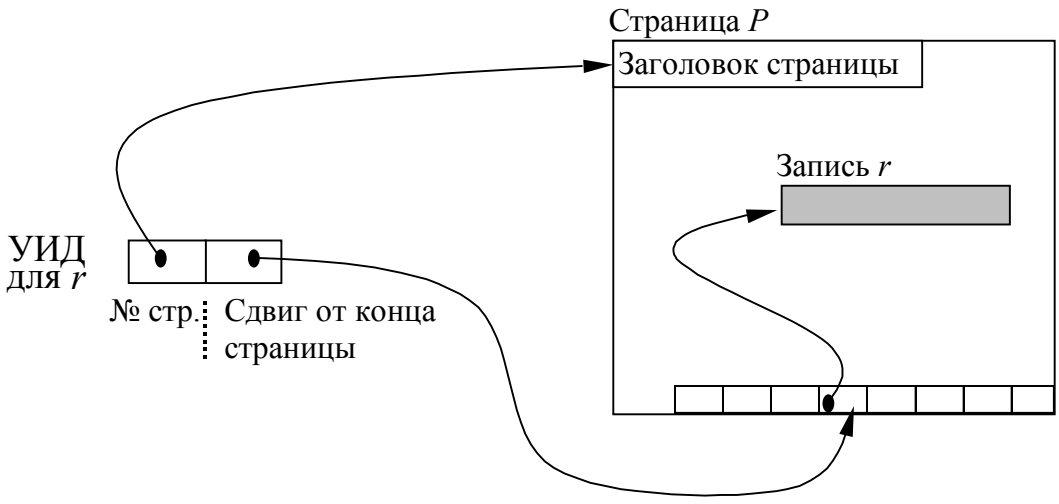


Рис. 4.16 Структура уникального идентификатора записи

УИД записи является уникальным в пределах диска. Значение УИД остается неизменным на протяжении всего времени жизни записи. УИД обеспечивает прямой доступ к записи, так как обращение к любой записи по ее УИД требует не более одной операции чтения/записи страницы.

ГЛАВА 5. МЕТОДЫ УПРАВЛЕНИЯ БУФЕРНЫМ ПУЛОМ В СИСТЕМЕ ХРАНЕНИЯ ДАННЫХ

5.1 Буферизация данных

В задачах, связанных с обработкой файлов, хранящихся на внешних носителях, узким местом могут стать интенсивные обмены данными с диском, поскольку скорость чтения данных с диска в среднем на порядок ниже скорости обработки данных процессором. Общепринятым подходом к решению данной проблемы является механизм буферизации данных.

Буферизация заключается в выделении буферного пула фиксированного размера в оперативной памяти. Доступ к содержимому страницы файла осуществляется только через ее образ в буферном пуле. То есть для обработки страницы файла необходимо сначала поместить ее образ в буферный пул. По окончании обработки измененный образ страницы сбрасывается на внешний носитель.

При использовании буферного пула алгоритм операции обращения к странице файла выглядит следующим образом. Сначала проверяется наличие образа запрашиваемой страницы в буферном пуле. Если образ страницы находится в буферном пуле, то алгоритм возвращает указатель на начальный адрес страницы и заканчивает работу. Если образ запрашиваемой страницы отсутствует в буферном пуле, то осуществляется поиск свободного места в буфере для последующей загрузки требуемой страницы. В случае, если в буфере отсутствует свободное место, выбирается некоторая неиспользуемая в данный момент страница, которая *вытесняется* на диск, а на ее место загружается запрошенная страница.

В данной главе мы будем использовать следующую терминологию.

Ситуацию, когда запрашиваемая страница файла находится в буферном пуле, назовем *попаданием*.

Ситуацию, когда при обращении к странице файла ее образ отсутствует в буферном пуле, назовем *промахом*.

Страницу, вытесняемую из буферного пула для того, чтобы освободить место для загрузки другой страницы, назовем *жертвой*.

При управлении вытеснением страниц оптимальной была бы политика, при которой в качестве жертвы всегда бы выбиралась та страница, к которой в дальнейшем дольше всего не будет обращений. Однако на практике такую политику вытеснения реализовать, как правило, не удается, поскольку для этого необходимо заранее знать всю последовательность обращений к страницам файла. В связи с этим возникает проблема подбора некоторой *стратегии* вытеснения страниц из буферного пула.

Стратегия вытеснения страниц задает правило, по которому страницы вытесняются из буферного пула.

Эффективность стратегии вытеснения определяется как соотношение числа попаданий к общему числу обращений к страницам файла в ходе выполнения той или иной задачи.

Существует большое количество работ, посвященных проблематике поиска эффективных стратегий вытеснения (см., например, обзор [78]). Однако одна и та же стратегия может показывать разную эффективность для различных классов задач. С другой стороны, ни одна из известных стратегий вытеснения не может обеспечивать одинаково высокую эффективность для всех случаев. Вследствие этого остаются актуальными вопросы разработки новых стратегий вытеснения страниц и методов управления буферным пулом.

Далее в разделе 5.2 приводится обзор известных стратегий вытеснения страниц из буферного пула и анализируются проблемы, связанные с их применением. Раздел 5.3 содержит описание оригинального метода вытеснения страниц, примененного при реализации системы управления файлами программного комплекса Омега. В разделе 5.4 приводятся результаты

численных экспериментов, исследующих эффективность предложенного метода.

5.2 Обзор стратегий вытеснения страниц из буферного пула

Стратегии вытеснения могут быть разделены на две группы [78]: стратегии, использующие загрузку страниц по требованию (demand paging strategies), и стратегии, использующие предварительную загрузку страниц (prepaging strategies). Стратегии вытеснения, использующие *загрузку страниц по требованию*, помещают в буферный пул только одну затребованную страницу. Стратегии вытеснения, использующие *предварительную загрузку страниц*, помещают в буферный пул не только затребованную страницу, но также m ($m \geq 1$) дополнительных страниц.

5.2.1 Стратегии вытеснения, использующие загрузку страниц по требованию

Стратегии вытеснения, использующие загрузку страниц по требованию, помещают в буферный пул только одну затребованную страницу. Стратегии данной группы предписывают вытеснять из буфера самую "старую" страницу. При этом "возраст" страниц определяется разными способами: по времени последнего обращения к странице, по количеству обращений к странице, по очередности помещения страницы в буфер и т.д. К данной группе стратегий относится довольно большое число стратегий.

Среди стратегий данной группы можно выделить три стратегии, не имеющие практического применения, но представляющие теоретический интерес. Это стратегии OPT, WORST и RANDOM.

Стратегия OPT [65, 92] предписывает вытеснять ту из неиспользуемых страниц, к которой дольше всего не будет обращений. В общем случае данную стратегию реализовать невозможно, так как для этого необходимо заранее знать всю цепочку обращений к страницам диска. Однако стратегия OPT полезна для сравнительной оценки эффективности других

стратегий, поскольку OPT показывает теоретический максимум эффективности стратегии для заданной последовательности обращений.

Алгоритм реализации стратегии OPT для известной последовательности обращений к страницам диска можно описать следующим образом. Пусть известна последовательность обращений к страницам диска $r_1, r_2, ..., r_N$. Запись $r_i=p$ означает, что в момент времени i происходило обращение к странице диска с номером p. Функция $future_i(p)$ возвращает момент времени X ($i<X\le N$), в который произойдет *следующее* обращение к странице с номером p, и определяется следующим образом:

$$future_i(p) = \begin{cases} X, \text{ если } i < X \le N \wedge r_X = p \wedge p \notin \{r_{i+1},...,r_{X-1}\} \\ \infty, \text{ если } p \notin \{r_{i+1},...,r_N\} \end{cases}$$

Если в момент времени t необходимо вытеснить некоторую страницу из буферного пула, то жертвой является страница буфера p с максимальным значением $future_t(p)$.

Стратегия WORST [78] является антиподом стратегии OPT и предполагает вытеснение той страницы, повторное обращение к которой будет раньше, чем к другим страницам буфера. Стратегия WORST применяется только для оценки относительной эффективности других стратегий, так как задает "абсолютный минимум" эффективности для заданной последовательности обращений. Для известной последовательности обращений к страницам диска алгоритм реализации стратегии WORST отличается от алгоритма реализации стратегии OPT только условием определения страницы-жертвы. Если в момент времени t необходимо вытеснить некоторую страницу из буферного пула, то жертвой является страница буфера p с минимальным значением $future_t(p)$.

Стратегия RANDOM выбирает страницу-жертву случайным образом. Эффективность, показываемая стратегией RANDOM, может служить критерием практического применения других стратегий вытеснения. Если стратегия показывает эффективность ниже, чем RANDOM, то ее применение на практике не оправдано.

Среди стратегий вытеснения, имеющих практическое применение, могут быть выделены *общие стратегии* и стратегии, которые получены путем их модификации. К общим стратегиям могут быть отнесены стратегии LRU, LFU, FIFO и LIFO.

В соответствии со *стратегией LRU (Least Recently Used)* из буферного пула вытесняется страница, к которой дольше всего не было обращений. Данная стратегия наиболее часто используется в операционных системах для буферизации доступа к файлам [16]. Стратегия LRU обычно реализуется путем введения атрибута образа страницы "время последнего обращения". При необходимости вытеснения страницы из буфера в качестве жертвы выбирается страница с минимальным значением данного атрибута.

Стратегия LFU (Least Frequently Used) вытесняет из буферного пула страницу, к которой было меньше всего обращений за время ее нахождения в буферном пуле. Данная стратегия обычно реализуется путем введения атрибута образа страницы "счетчик обращений к странице". При помещении страницы в буфер значение счетчика устанавливается в 1 и увеличивается всякий раз, когда происходит обращение к данной странице. При поиске жертвы осуществляется поиск страницы с минимальным значением счетчика. При применении данной стратегии возможна ситуация, когда за сравнительно небольшой промежуток времени наиболее интенсивно используемые страницы получат столь высокие значения данного атрибута, что никогда не будут вытеснены из буферного пула, несмотря на то, что в дальнейшем обращения к этим страницам могут отсутствовать. В силу данного обстоятельства стратегия LFU применяется преимущественно в модифицированном виде [78, 100].

Стратегия FIFO (First In First Out) моделирует очередь страниц буферного пула и предписывает вытеснение страницы, которая раньше всех остальных была туда помещена.

Согласно *стратегии LIFO (Last In First Out)* из буферного пула вытесняется та из неиспользуемых страниц, которая была загружена последней.

Среди стратегий вытеснения страниц, представляющих собой модификацию общих стратегий, можно отметить стратегии CLOCK, GCLOCK, LRU-K и 2Q.

Стратегия CLOCK [72] представляет собой некоторый вариант стратегии LRU. Схема работы стратегии CLOCK представлена на Рис. 5.1.

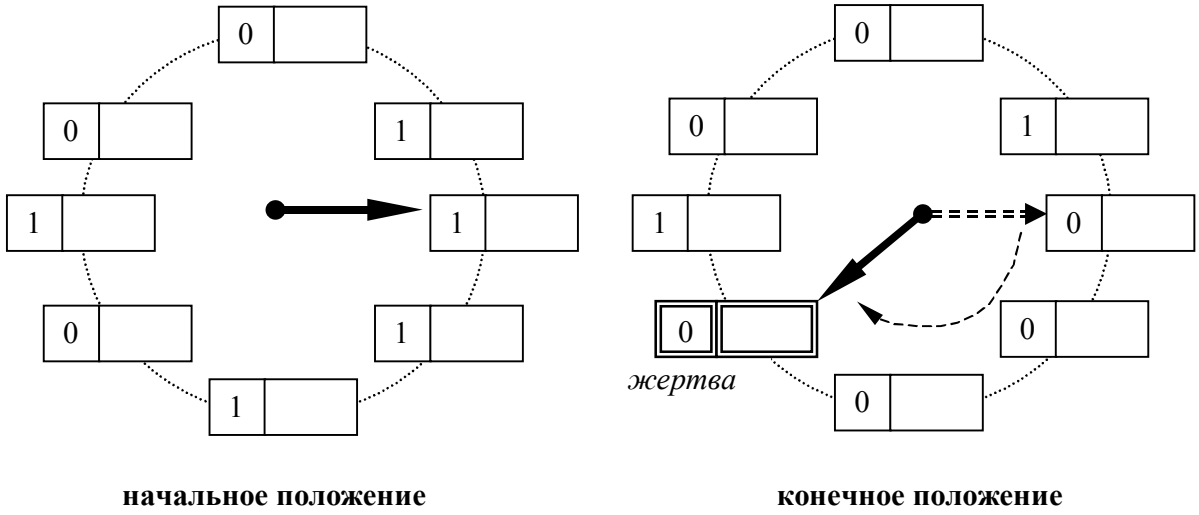


Рис. 5.1 Схема работы стратегии CLOCK

С каждой страницей в буфере связывается флаг, значением которого может быть 0 или 1. Если значение флага равно 0, то страница может быть вытеснена на диск; страницы со значением флага 1 из буфера не вытесняются. При помещении страницы в буфер ее флаг устанавливается в значение 1. При каждом обращении к образу страницы в буфере значение ее флага также изменяется на 1. Страницы буферного пула полагаются расположенными по кругу, как на циферблате часов. В случае необходимости вытеснить страницу из буфера осуществляется вращение "стрелки" часов, начиная с текущего положения. Жертвой является первая найденная страница с флагом, равным 0. При этом во время вращения стрелка, проходя через страницы с флагом 1, устанавливает их в 0.

Стратегия GCLOCK (Generalized CLOCK) [94] является модификацией стратегии CLOCK. Схема работы стратегии GCLOCK приведена на Рис. 5.2. Флаг, ассоциируемый со страницей буфера, заменяется на счетчик обращений к странице. Во время поиска жертвы значение счетчика обращений просмотренных страниц буфера уменьшается на 1. Вытеснению подлежит первая найденная страница, у которой значение счетчика равно нулю.

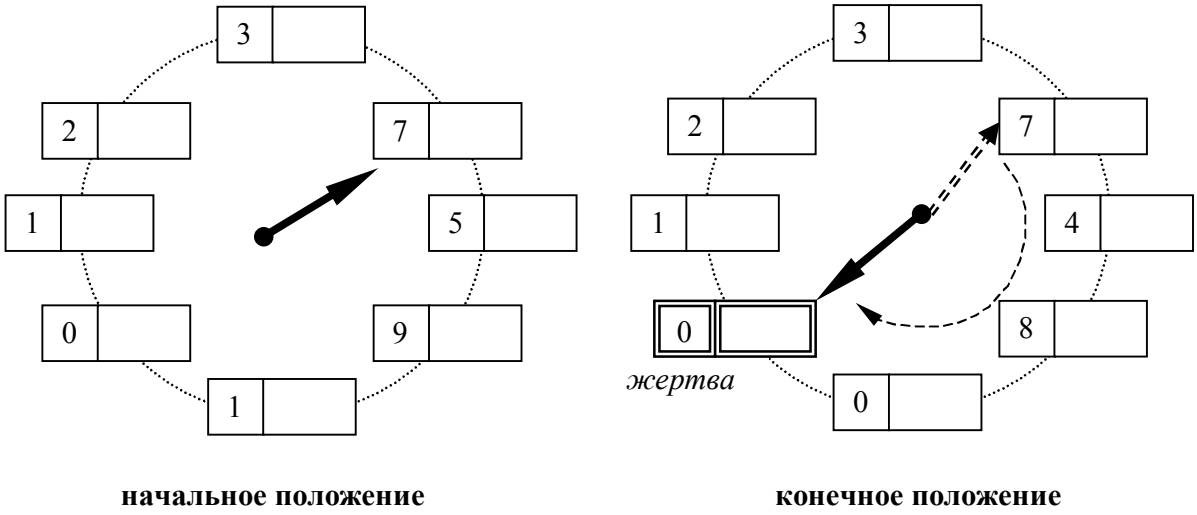


Рис. 5.2 Схема работы стратегии GCLOCK

Стратегия LRU-K [95] является обобщением стратегии LRU и при определении жертвы анализирует K последних обращений к странице. Алгоритм поиска жертвы в стратегии LRU-K основан на следующем определении расстояния между текущим и K-м предыдущим обращением к странице. Пусть известна последовательность обращений к страницам диска $r_1, r_2, ..., r_t$, где t – текущий момент времени. Запись $r_i=p$ означает, что в момент времени i происходило обращение к странице диска с номером p. Пусть $K \geq 1$, тогда расстоянием между текущим и K-м предыдущим обращением к странице с номером p называется число $dist_t(p,K)$, вычисляемое по следующему правилу:

$$dist_t(p,K) = \begin{cases} X, \text{ если } r_{t-X} = p \wedge \text{в } \{r_{t-X+1},...,r_t\} \;\; p \text{ встречается } K-1 \text{ раз} \\ \infty, \text{ если в } \{r_1,...,r_t\} \; p \text{ встречается менее } K \text{ раз} \end{cases}$$

При необходимости вытеснить страницу из буферного пула жертвой является страница p с максимальным значением $dist_t(p,K)$. Отметим, что в случае K=1 данная схема определяет классическую стратегию LRU.

Стратегия 2Q (two queues) [88] предполагает поддержку двух очередей страниц буферного пула: вспомогательной и основной. После первого обращения страница помещается во вспомогательную очередь. Если имело место повторное обращение к данной странице, то страница помещается в основную очередь. При необходимости вытеснить страницу из буферного пула поиск жертвы осуществляется во вспомогательной очереди по принципу FIFO.

5.2.2 Стратегии вытеснения, использующие предварительную загрузку страниц

Стратегии вытеснения, использующие предварительную загрузку страниц, помещают в буферный пул не только затребованную страницу файла, но также m (m≥1) следующих страниц данного файла. Это позволяет повысить эффективность операций чтения-записи данных в случае последовательного доступа к страницам файла.

Стратегия LRU-OBL (LRU-One Block Lookahead) [102] предполагает предварительную загрузку одной дополнительной страницы файла, которая является следующей за запрашиваемой (если только эта страница уже не находится в буфере). Определение страницы-жертвы производится по принципу LRU.

Стратегия W^2R (Weighing and Waiting Rooms) [87] является некоторой модификацией стратегии LRU-OBL. Буферный пул логически разделяется на две части: "комната ожидания" и "комната взвешивания". Предварительно загружаемая страница помещается в "комнату ожидания". В случае обращения к странице из "комнаты ожидания" она переводится в "комнату взвешивания". Поиск страницы-жертвы осуществляется только в "комнате взвешивания" по принципу LRU.

5.2.3 Проблемы, связанные с использованием стратегий вытеснения страниц

С использованием общих стратегий вытеснения связан ряд проблем. Первая из них заключается в том, что стратегия вытеснения может показывать в среднем удовлетворительную общую эффективность, но быть наихудшей в отдельных случаях доступа к базе данных.

Доступ к базе данных представляет собой комбинацию следующих разновидностей доступа к файлу [108]:

1) последовательный доступ к страницам файла без повторения;
2) последовательный доступ к страницам файла, повторяющийся в цикле;
3) случайная выборка страниц без повторения;
4) случайная выборка страниц с повторением.

Для случая 4 стратегия LRU, которая широко применяется в реализации операционных систем, дает эффективность, близкую к OPT. Но для остальных случаев стратегия LRU оказывается плохим выбором. Для случаев 1 и 3 следует использовать немедленное вытеснение страницы сразу после использования. В случае 2 последовательность обращений имеет вид $r_1, r_2, \ldots, r_n, r_1, r_2, \ldots, r_n, \ldots$. Очевидно, что LRU будет наихудшим выбором в данной ситуации. Если все n страниц, составляющих цикл, не могут быть одновременно помещены в буферный пул, то в данном случае следует использовать немедленное вытеснение страницы сразу после использования. В работе [89] указывается, что использование методов вытеснения, учитывающих специфику алгоритмов баз данных, может увеличить эффективность стратегии на 10-15% по сравнению со стратегией LRU.

Вторая проблема связана с тем, что общие стратегии вытеснения могут быть неприменимыми в ряде случаев, поскольку они не поддерживают избирательное вытеснение страниц. Под избирательным вытеснением понимается возможность явного задания порядка вытеснения некоторых страниц из буферного пула. Примером необходимости избирательного вы-

теснения является порядок вытеснения страниц для обеспечения корректного восстановления базы данных после сбоя. Если для восстановления данных используется механизм журнализации [82], то страница журнала, содержащая флаг "зафиксировано", должна вытесняться из буферного пула на диск только после вытеснения всех страниц, содержащих изменения, выполненные в ходе транзакции.

Избирательное вытеснение может быть также полезным при сканировании отношения реляционной базы данных с использованием индексного файла в виде B-дерева. В этом случае к странице буферного пула, содержащей корень этого дерева, будут выполняться циклические обращения. При использовании стратегии вытеснения LRU данная страница будет самой "старой" страницей индекса. Если индекс отношения не помещается целиком в буферный пул, то страница, содержащая корень B-дерева, будет циклически загружаться в буфер и вытесняться на диск, что приведет к большим накладным расходам [82].

Рассмотренные выше общие стратегии вытеснения и их модификации не предоставляют механизмов для решения указанных проблем, вследствие чего разработка новых стратегий и методов вытеснения страниц из буферного пула остается актуальной задачей.

5.3 DIR-МЕТОД УПРАВЛЕНИЯ БУФЕРНЫМ ПУЛОМ

При реализации системы управления файлами программного комплекса Омега нами был разработан оригинальный метод вытеснения страниц, получивший название DIR-метода [58, 118]. Основными принципами, использованными при разработке DIR-метода, являлись следующие:

1. Поддержка избирательного вытеснения страниц.
2. Возможность эмуляции практически всех общих стратегий вытеснения страниц.
3. Возможность динамического выбора адекватной стратегии вытеснения страниц.

Реализация данных принципов базируется на следующих двух концепциях:

- введение для страниц статического и динамического рейтингов;
- использование избыточного индекса буферного пула.

Механизм статического рейтинга позволяет реализовать избирательное вытеснение страниц. Механизм динамического рейтинга позволяет моделировать различные стратегии вытеснения страниц. Концепция рейтингов страниц изложена в разделе 5.3.1.

Использование избыточного индекса буферного пула позволяет накапливать и сохранять статистику использования страниц, анализ которой дает возможность динамического выбора наиболее адекватной стратегии вытеснения страниц. Концепция избыточного индекса буферного пула рассматривается в разделе 5.3.2.

5.3.1 Рейтинги страниц

Статический рейтинг – это целое число от 0 до 20. Статический рейтинг является атрибутом *открытого* файла и определяется пользователем при выполнении операции открытия файла. Все страницы файла имеют одинаковы статический рейтинг, который остается неизменным, пока файл открыт. При закрытии файла значение статического рейтинга его страниц теряется.

Механизм статического рейтинга позволяет реализовать избирательное вытеснение страниц. Страницы, имеющие больший статический рейтинг, не будут вытесняться из буферного пула до тех пор, пока в нем имеются страницы с меньшим статическим рейтингом.

Динамический рейтинг – это некоторая функция, принимающая значения в интервале [0;1[. Аргументами данной функции являются атрибуты статистики использования страниц, хранящиеся в избыточном индексе буферного пула. Примерами атрибутов статистики использования страниц являются следующие атрибуты: число обращений к странице, время по-

следнего обращения к странице, время помещения страницы в буфер и др. Более детально статистические атрибуты рассматриваются в разделе 5.3.2. Различные страницы одного и того же файла могут иметь различные динамические рейтинги. Значение динамического рейтинга страницы может меняться во время нахождения страницы в буфере. Способ вычисления динамического рейтинга задается системой.

Динамический рейтинг реализует и обобщает понятие "возраста" страницы. Механизм динамического рейтинга позволяет моделировать различные стратегии вытеснения страниц.

Суммарный рейтинг страницы получается как сумма статического и динамического рейтингов. Если необходимо освободить место в буферном пуле, то вытесняется страница, имеющая минимальный суммарный рейтинг. Если таких страниц несколько, то вытесняется страница, дольше всего находящаяся в буферном пуле.

5.3.2 Избыточный индекс буферного пула (DIR)

Избыточный индекс буферного пула DIR представляет собой таблицу в оперативной памяти, в которой, помимо указателей на образы страниц, находящихся в данный момент в буфере, хранится статистическая информация об использовании этих страниц. Структура DIR изображена на Рис. 5.3.

Избыточность DIR выражается в том, что после вытеснения страницы из буфера соответствующий элемент в DIR сохраняется. Это позволяет накапливать дополнительную статистическую информацию, которая используется приопределении очередной жертвы.

Длина DIR определяется как kM, где M – длина буферного пула в страницах, а k – некоторое целое, большее 1. Число k называется *кратностью* DIR.

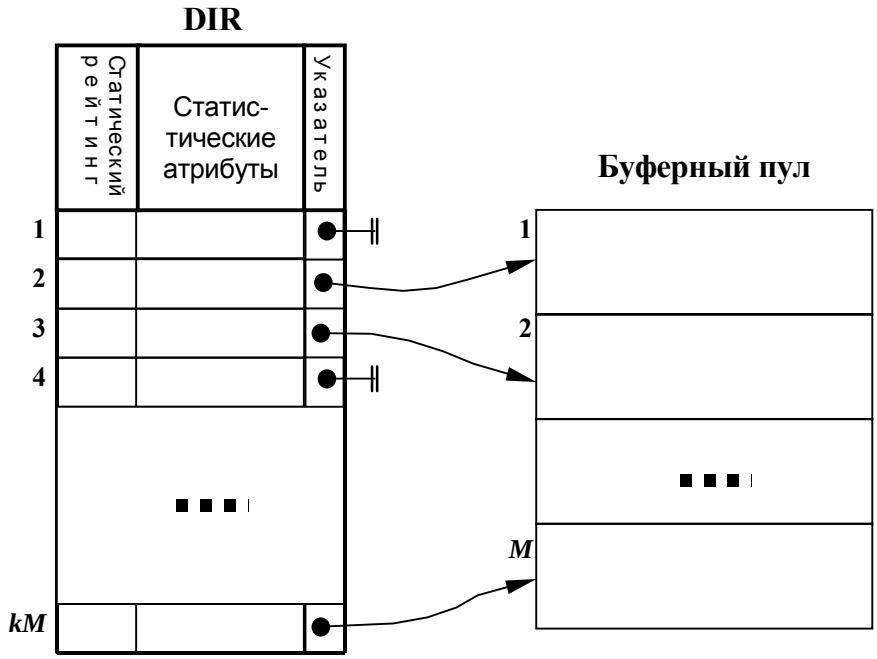


Рис. 5.3 Структура избыточного индекса буферного пула (DIR)

В DIR хранится статистика использования страниц, которые находились или находятся в буфере. Основные статистические атрибуты, хранящиеся в DIR, приведены в Табл. 5.1.

Табл. 5.1 Основные статистические атрибуты, хранящиеся в DIR

Атрибут	Семантика
HC (hit counter)	счетчик попаданий
FC (fault counter)	счетчик промахов
HT (hit time)	время последнего попадания
FT (fault time)	время последнего промаха
LT (load time)	время помещения страницы в буфер

Статистические атрибуты позволяют моделировать в рамках DIR-метода практически любую общую стратегию вытеснения. Соответствующие формулы подсчета динамического рейтинга страниц для общих стратегий приведены в Табл. 5.2.

Табл. 5.2 Моделирование общих стратегий при помощи динамического рейтинга

Стратегия	Динамический рейтинг
LRU	NORM(*HT*)
LFU	NORM(*HC*)
FIFO	NORM(*LT*)
LIFO	1/NORM(*LT)*

Здесь *NORM* – функция нормирования, приводящая значение к диапазону [0;1[.

Алгоритм операции выборки страницы, в DIR-методе имеет две фазы: добавление элемента в DIR и добавление страницы в буфер.

На первой фазе проверяется наличие в DIR дескриптора выбираемой страницы. Если искомый элемент отсутствует в DIR, то выполняется его добавление в DIR. Если при этом в DIR отсутствуют свободные позиции, то осуществляется поиск элемента-"жертвы" в DIR, на место которой записывается новый элемент. Поиск жертвы ведется среди элементов DIR, имеющий пустой указатель на образ страницы в буфере. Жертвой является элемент с наименьшим суммарным рейтингом.

На второй фазе осуществляется поиск страницы в буфере. Если образ страницы находится в буфере, то алгоритм заканчивает работу. В противном случае проверяется наличие свободного блока в буферном пуле. Если свободный блок есть, то в него загружается соответствующая страница диска, и алгоритм заканчивает работу. Если свободных блоков нет, то осуществляется поиск страницы-"жертвы", которая вытесняется на диск, а на ее место загружается требуемая страница. Жертвой является страница с наименьшим суммарным рейтингом.

5.3.3 Построение стратегий вытеснения на базе DIR-метода

В описанном выше алгоритме выборки страницы вытеснение на основании суммарного рейтинга страницы применяется не только к страни-

цам, образы которых в данный момент находятся в буфере (*вытеснение из буфера*), но и к страницам, образы которых отсутствуют в буфере (*вытеснение из DIR*). Причем, для этого могут использоваться различные формулы вычисления динамического рейтинга страниц. Таким образом, DIR-метод позволяет реализовать широкий класс стратегий вытеснения, каждую из которых можно однозначно определить, указав формулу подсчета динамического рейтинга для страниц буфера и формулу подсчета динамического рейтинга для элементов DIR.

На базе DIR-метода был разработан ряд оригинальных стратегий вытеснения, некоторые из которых приведены в Табл. 5.3.

Табл. 5.3 Примеры стратегии вытеснения на базе DIR-метода

Стратегия	Динамический рейтинг
DIR_{FT}	NORM(*HT*+*FT*/*k*)
DIR_{FC}	NORM(*HC*+*FC*/*k*)
DIR_{LT}	NORM(*LT*+*FT*/*k*)
DIR_{HCFT}	NORM(*HC*+*FT*)

Здесь *k* обозначает кратность DIR, *NORM* – функция нормирования, приводящая значение к диапазону [0;1[. В приведенных примерах формулы подсчета динамического рейтинга для вытеснения страниц из буфера совпадают с формулами подсчета для вытеснения страниц из DIR.

5.4 Численные эксперименты

Для исследования эффективности DIR-метода буферизации страниц были проведены численные эксперименты. При проведении экспериментов ставились следующие цели:

- сравнить эффективность общих стратегий вытеснения страниц и стратегий, использующих DIR-метод;
- сравнить между собой эффективность DIR-стратегий с различными формулами вычисления динамического рейтинга;

- исследовать влияние кратности DIR на эффективность стратегий, использующих DIR-метод.

Эксперименты заключаются в применении заданной стратегии вытеснения к последовательности из L случайных обращений к страницам диска 1, …, N с неравномерным распределением частот обращения. Эффективность стратегии вычисляется как $\frac{H}{L}$, где H – число попаданий. Размер буферного пула в страницах варьируется от 1 до N. Параметры экспериментов, используемые по умолчанию, представлены в Табл. 5.4.

Табл. 5.4 Параметры численных экспериментов

Параметр	Семантика	Значение
L	Количество обращений к страницам диска	3000
N	Количество страниц диска, к которым осуществляются обращения	1000
$Z(i)$	Функция распределения частот обращений	$\left(\frac{i}{N}\right)^{\log 0,8 / \log 0,2}$
k	Кратность DIR	10

В качестве распределения частот обращений к страницам диска 1, …, N берется распределение Зипфа [71]. В соответствии с данным распределением вероятность обращения к странице с номером, меньшим либо равным i, определяется по формуле

$$\left(\frac{i}{N}\right)^{\log 0,8 / \log 0,2}$$

Данное распределение моделирует так называемый закон Зипфа "80-20", согласно которому 80% запросов к базе данных адресуется к 20% значений базы данных. Закон Зипфа отражает типичную ситуацию распределения частот обращений к страницам диска при обработке данных [68, 82].

5.4.1 Сравнение эффективности общих стратегий с DIR-стратегией

Для сравнительного анализа эффективности стратегий вытеснения страниц, использующих DIR, и общих стратегий, была взята стратегия DIR_{FC} и общая стратегия LRU. Стратегия LRU среди общих стратегий считается лучшей в среднем (то есть когда частоты обращений к страницам распределены равномерно) [108].

Результаты эксперимента приведены на Рис. 5.4.

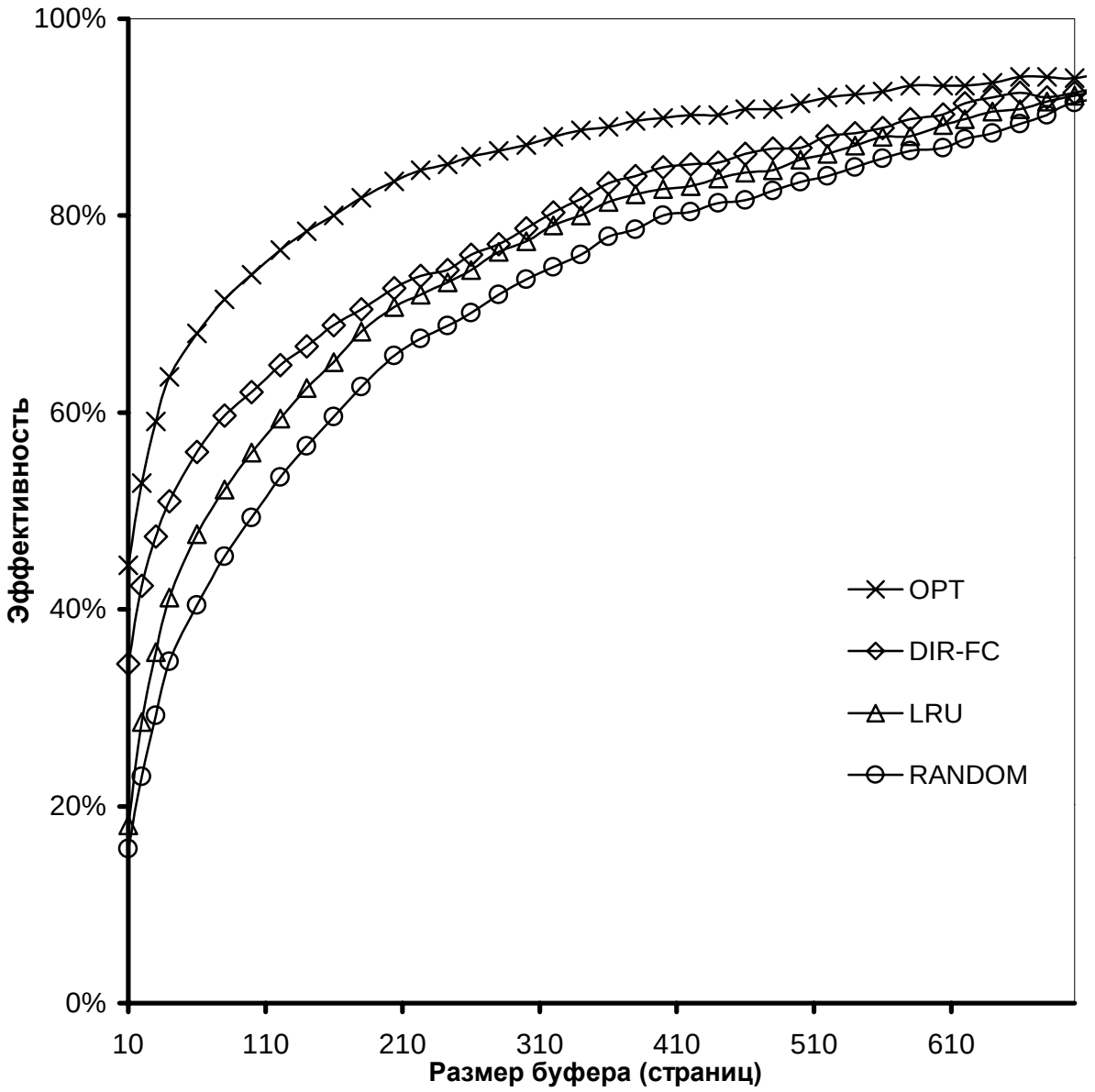


Рис. 5.4 Сравнение эффективности общих стратегий и стратеги DIR_{FC}

Для буферного пула небольшого размера (10-30 страниц) эффективность DIR_{FC} превосходит эффективность LRU на 15%. Затем это превос-

ходство снижается до 10% при размере буферного пула 30-100 страниц и 5% при буфере в 100-200 страниц. При больших размерах буфера обе стратегии показывают практически одинаковую эффективность. Данная тенденция характерна для любых стратегий, так как при размере буфера в страницах, близком к числу используемых страниц, вытеснение страниц практически отсутствует и различия в стратегиях вытеснения нивелируются.

Таким образом, в данном эксперименте показана более высокая эффективность DIR-стратегии вытеснения по сравнению с классической стратегией LRU.

5.4.2 Сравнение эффективности различных DIR-стратегий

Целью данного эксперимента было сравнение эффективности различных DIR-стратегий. Для сравнительного анализа были взяты стратегии DIR_{FC} и $DIR_{FT.}$

Результаты эксперимента приведены на Рис. 5.5. Эффективность стратегии DIR_{FC} для буферного пула размером 10-60 страниц выше, чем у DIR_{FT}, в среднем на 10%. При размере буфера более 150 страниц стратегии показывают практически одинаковую эффективность.

Результаты данного эксперимента показывают более высокую эффективность стратегии DIR_{FC}, которая при подсчете динамического рейтинга страниц использует атрибут "счетчик промахов".

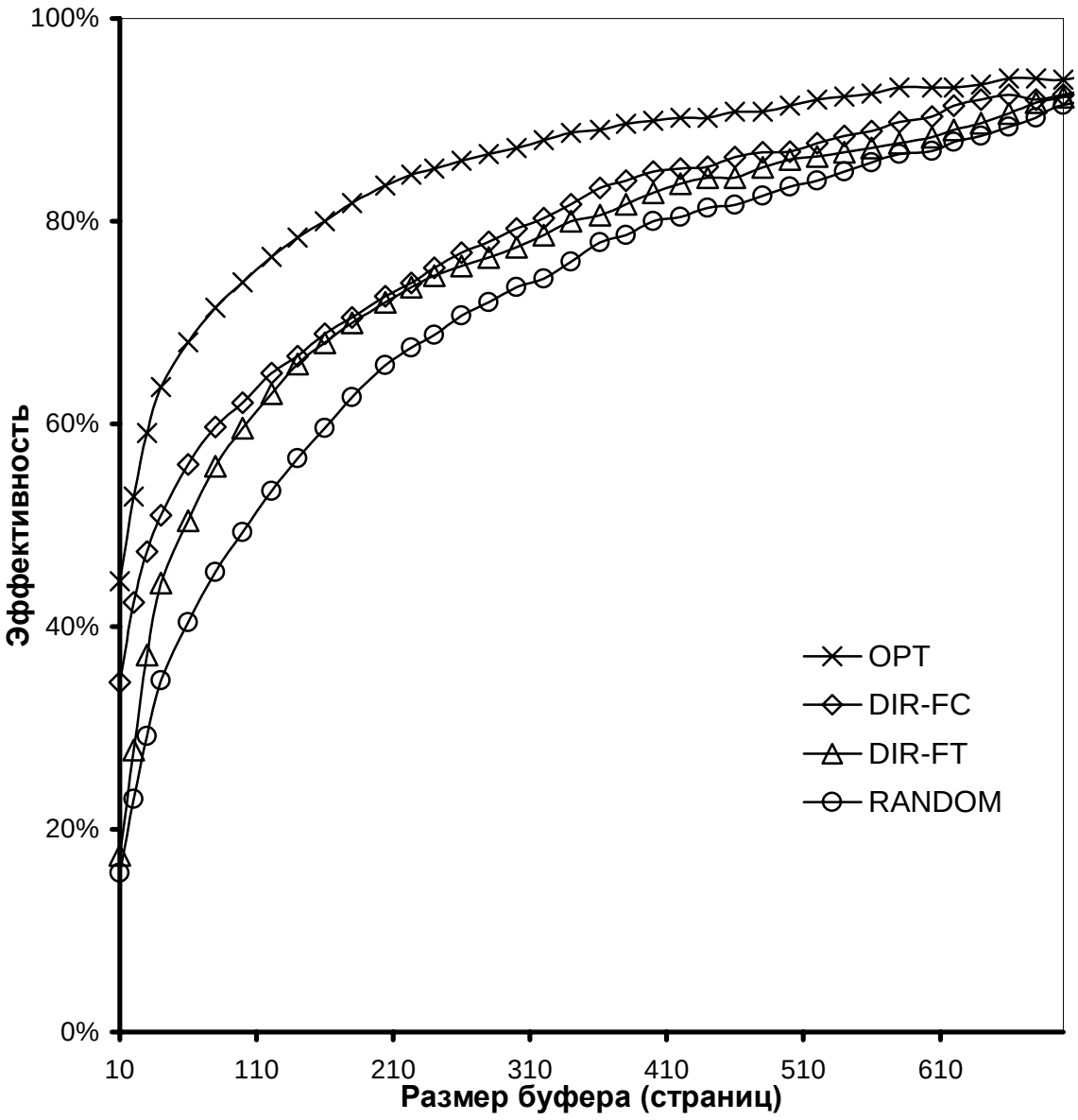


Рис. 5.5 Сравнение эффективности различных DIR-стратегий

5.4.3 Исследование влияния кратности DIR на эффективность DIR-стратегии

Целью данного эксперимента было определение значения k – кратности длины DIR, при котором можно получить наибольшую эффективность DIR-стратегии. Для сравнительного анализа была выбрана стратегия DIR_{FC}, которая в двух предыдущих экспериментах показала наилучшую эффективность. В данном эксперименте сравнивалась эффективность стра-

тегии DIR_{FC} при различных значениях *k*: 5, 10, 15, 20, 25, и 30 (соответственно 0,5%, 1%, 1,5%, 2%, 2,5% и 3% от общего числа страниц на диске).

Результаты эксперимента приведены в Табл. 5.5. При увеличении кратности с 5 до 20 (с 0,5% до 2% от общего числа страниц на диске) эффективность стратегии DIR_{FC} возрастает. При значении $k>20$ эффективность стратегии уменьшается. При размере буфера более чем 200 страниц стратегия показывает практически одинаковую эффективность, независимо от значения *k*.

Таким образом, результаты данного эксперимента показывают, что при значении кратности DIR $k=0{,}02D$, где D – общее число страниц на диске, стратегия DIR_{FC} показывает наибольшую эффективность.

Табл. 5.5 Влияние кратности DIR на эффективность стратегии DIR_{FC}

Размер буфера (страниц)	Эффективность стратегии DIR_{FC}, %				
	k=5	k=10	k=20	k=25	k=30
10	33,70	34,50	**34,11**	34,40	34,45
20	42,20	42,40	**41,95**	40,40	41,80
30	46,90	47,40	**46,55**	44,90	46,55
40	49,60	51,00	**51,30**	49,40	49,90
60	54,30	56,00	**59,40**	55,60	54,41
80	57,40	59,70	**63,50**	59,80	59,00
100	60,40	62,10	**66,90**	63,80	62,60
120	62,80	64,80	**69,80**	66,90	65,59
140	64,70	66,70	**72,20**	69,80	68,60
160	66,50	68,90	**73,60**	71,40	70,50
180	68,70	70,50	**74,80**	73,40	72,28
200	70,80	72,60	**76,20**	74,60	74,00
220	71,60	73,90	**77,30**	76,00	75,40
240	72,90	74,70	**79,00**	77,20	77,20
260	73,80	76,00	**80,00**	78,20	77,80
280	75,70	77,10	**80,90**	79,00	79,00
300	76,55	78,70	**81,80**	81,10	81,10
320	78,30	80,30	**82,00**	81,90	82,00
340	79,80	81,70	**82,70**	82,20	82,10
360	80,80	82,30	**83,70**	82,50	82,40
380	81,70	83,10	**84,30**	82,70	82,70
400	82,80	84,00	**84,60**	82,90	82,90
420	83,60	84,40	**84,80**	83,20	83,10
440	84,80	85,30	**85,00**	83,90	83,50
460	85,00	86,40	**85,40**	84,10	84,00
480	85,50	86,80	**85,80**	84,60	84,20
500	86,20	86,90	**86,95**	85,20	85,00

ГЛАВА 6. ТЕХНОЛОГИЧЕСКИЕ АСПЕКТЫ РАЗРАБОТКИ ПРОГРАММНОГО КОМПЛЕКСА ДЛЯ УПРАВЛЕНИЯ ДАННЫМИ

6.1 Технология коллективной разработки программного комплекса Омега

Эффективная разработка больших программных комплексов сегодня не возможна без использования современных технологий программирования. Технологии программирования направлены на решение проблем, связанных с созданием программного обеспечения, обладающего такими качествами, как надежность, модульность, сопровождаемость, самодокументируемость, модифицируемость, повторная используемость кода. Указанные проблемы возникают при создании программного обеспечения для различных предметных областей. Вследствие этого в сфере разработки новых технологий программирования, в том числе для параллельного программирования, ведутся интенсивные научные исследования (см., например, [8, 40, 41, 55]).

По определению, данному в работе [41], под технологией программирования понимается совокупность концептуальных, организационных и программных средств, используемых для создания программного обеспечения.

Концептуальные средства технологии программирования определяют стиль и методы проектирования и разработки программного обеспечения.

Организационные средства технологии программирования определяют форму труда в коллективе программистов и включают в себя средства контроля за выполнением работ, методики оценки принимаемых на стадии разработки решений и т.п. Примерами организационных средств могут

служить метод бригады главного программиста [4] и экстремальное программирование [2].

Программные средства осуществляют программную поддержку жизненного цикла программного обеспечения на этапах анализа, спецификации, проектирования, кодирования, тестирования и сопровождения [66]. К программным средствам технологии программирования можно отнести технологические программные комплексы (см., например, [33, 38, 44]), системы версионирования исходных текстов программ и т.п.

Для организации коллективной разработки подсистем, входящих в программный комплекс Омега, нами была предложена технология коллективной разработки программ для МВС-100 [47, 117], включающая в себя концептуальные, организационные и программные средства.

Технология коллективной разработки системы Омега базируется на следующих трех концептуальных компонентах: репозитории проекта, библиотеке проекта и справочнике по функциям библиотеки проекта.

Репозиторий проекта хранит все исходные тексты подсистем и все изменения, которые были в них произведены после включения в репозиторий. Используя репозиторий проекта, разработчики могут получать актуальные версии файлов исходных текстов по номеру версии проекта, дате создания и модификации и т.п.

Библиотека проекта (Ω-Lib) хранит объектный код функций подсистем проекта. Разработчик использует библиотеку проекта для компиляции исходных текстов своих подсистем, использующих функции подсистем проекта, созданных другими разработчиками.

Справочник по функциям Ω-Lib представляет собой гипертекстовый документ в формате HTML, который может просматриваться любым WWW-обозревателем.

6.2 Организационные средства технологии коллективной разработки

В коллективе, выполняющем работы по проекту, выделяются руководитель проекта и технолог проекта.

Руководитель проекта осуществляет общее руководство проектом. В функции руководителя входит, например, распределение работ для рядовых разработчиков и контроль сроков разработки.

Технолог проекта – это специально выделенный разработчик, который поддерживает репозиторий проекта, библиотеку проекта и Справочник по функциям Ω-Lib.

Разработка подсистем проекта организована таким образом, что каждая подсистема разрабатывается одним разработчиком. При этом один и тот же разработчик может заниматься разработкой более чем одной подсистемы одновременно.

Организация технологического цикла коллективной разработки представлена на Рис. 6.1.

Для обращения к функциям подсистем библиотеки программист получает из репозитория тексты соответствующих заголовочных файлов. Затем программист строит загрузочные модули тестов разрабатываемой им подсистемы, при этом в качестве параметра компиляции указывается библиотека Ω-Lib. Далее загрузочные модули тестов выполняются на МВС. По полученным отладочным дампам анализируются и локализуются ошибки. В исходные тексты вносятся исправления, и цикл повторяется. После того, как все тесты новой подсистемы прошли без ошибок, ее исходные тексты передаются технологу проекта. При этом данные тексты должны содержать спецификации функций подсистемы, пригодные для включения в Справочник по функциям Ω-Lib.

Технолог выполняет следующую стандартную последовательность действий. Он помещает полученные исходные тексты подсистемы в свою рабочую копию проекта. После этого технолог собирает и запускает ком-

плексный тест системы. Если при выполнении теста возникли ошибки, подсистема возвращается программисту на доработку. Если комплексный тест прошел нормально, то технолог помещает исходные тексты подсистемы в репозиторий. Затем создается новый вариант библиотеки Ω-Lib, включающий в себя функции подсистемы, полученной от разработчика. В завершении цикла генерируются HTML страницы, содержащие справочную информацию по функциям новой подсистемы, которые добавляются в Справочник по функциям Ω-Lib.

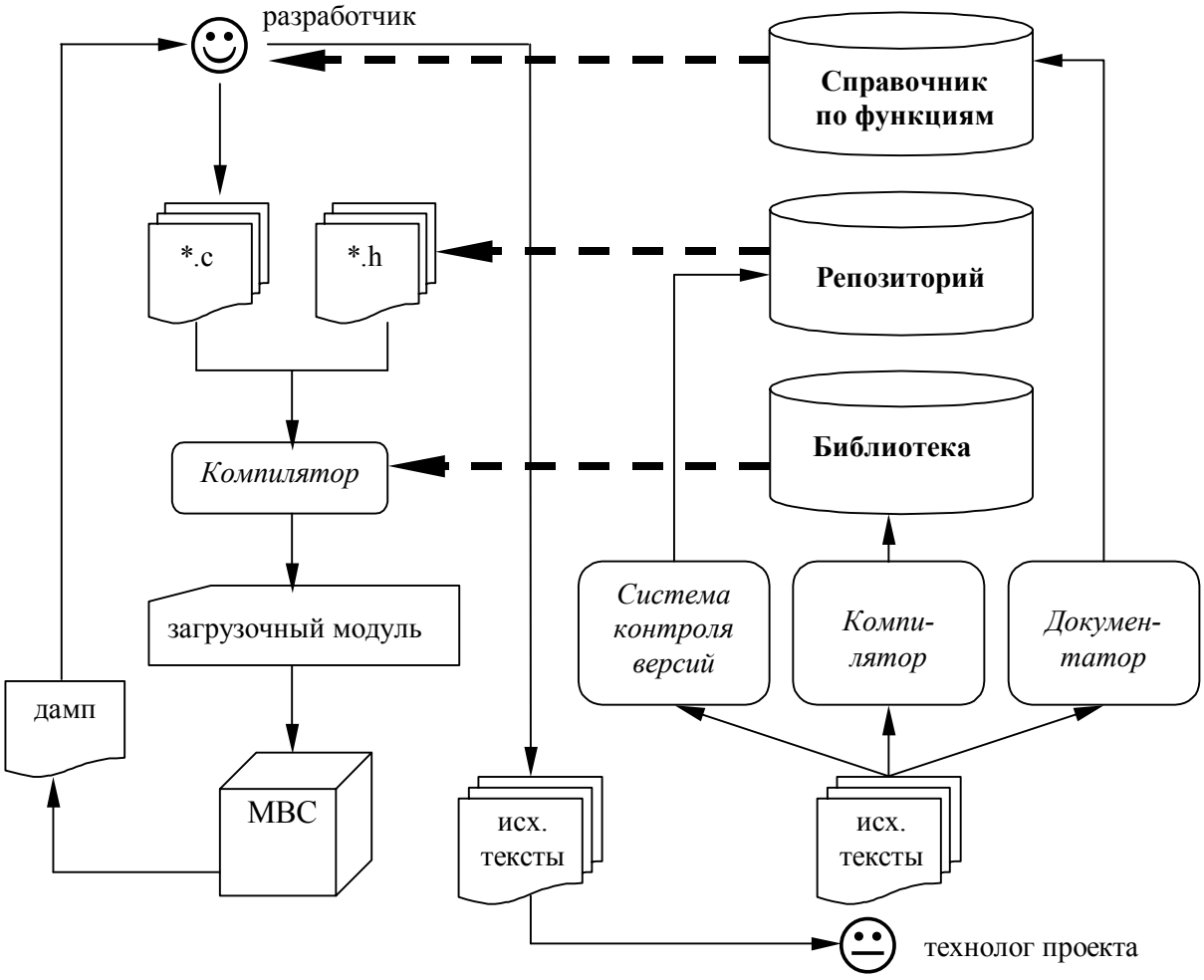


Рис. 6.1 Технологический цикл коллективной разработки системы Омега

Если при дальнейшей разработке проекта в подсистеме обнаруживается ошибка, то программист копирует в свой рабочий каталог библиотеку Ω-Lib и удаляет из этой копии библиотеки все модули своей подсистемы, получая библиотеку Ω'-Lib. После этого программист возобновляет работу

над своей подсистемой, только теперь в технологическом цикле вместо Ω-Lib он использует Ω'-Lib.

6.3 Программные средства технологии коллективной разработки

Программные средства технологии разработки комплекса Омега делятся на три группы: средства поддержки коллективной разработки, средства расширения среды программирования для МВС-100 и средства поддержки интегрированной среды разработки.

6.3.1 Средства поддержки коллективной разработки

Средства поддержки коллективной разработки обеспечивают ведение репозитория, библиотеки проекта и справочника по функциям проекта. Репозиторий поддерживается с помощью специальной системы контроля версий. Библиотека проекта поддерживается с помощью компилятора и программы-библиотекаря. Для генерации справочника по функциям используется документатор исходных текстов программ. Документатор анализирует комментарии специального вида и переводит их в формат HTML.

В качестве программных средств для указанных целей были выбраны система контроля версий CVS и документатор DOC++. Данные программные продукты являются разработками независимых фирм.

Система контроля версий CVS [67] является основным инструментом версионирования и поддержки коллективной разработки проекта.

Система функционирует в среде ОС UNIX и представляет собой CASE-пакет разработчика (toolkit), который можно использовать на этапе кодирования и сопровождения. В CVS поддерживаются понятия репозитория и рабочей копии проекта. В репозитории хранятся полные копии всех файлов и каталогов проекта. Каждый участник проекта работает со своей индивидуальной рабочей копией проекта.

Создав свою рабочую копию проекта, участник проекта должен затем периодически выполнять ее обновление. Изменения в файлах рабочей копии не будут доступны другим участникам проекта, пока не будет вы-

полнена их синхронизация. Выполняя синхронизацию, CVS заставляет разработчика прокомментировать внесенные изменения. При обновлении и синхронизации рабочей копии проекта CVS осуществляет автоматическое слияние изменений, внесенных другими разработчиками, если эти изменения не затрагивают одни и те же строки файлов. В случае конфликта изменений в рабочей копии файла с содержанием этого файла в репозитории конфликтующие строки помечаются в файле специальными маркерами, и разработчик должен разрешить конфликт "вручную", после чего выполнить синхронизацию изменений.

Разработчик может просмотреть историю изменения заданного файла, сравнить разные версии одного файла и выяснить статус файла (нужно ли выполнить синхронизацию файла, нуждается ли файл в обновлении, имеются ли конфликты).

Каждая версия файла имеет номер ревизии, который автоматически изменяется при выполнении синхронизации. Номера ревизий могут выглядеть так: 1.1, 1.2, 1.3, или, в случае создания ветки проекта 1.2.2.1. и даже 1.3.2.2.4.5. Файлы проекта (или релиз) с различными номерами ревизий можно объединять одним именем. Например, release-1, last_week_release, release-1_patches. Разработчик может получить копию файла или всего проекта по заданному номеру ревизии или имени релиза.

При отладке либо сопровождении предыдущих релизов проекта часто возникает потребность внести в них какие-либо изменения, а затем выполнить слияние этих изменений в текущую разработку. В CVS поддерживаются понятия основного пути (main trunk) и ветки (branch) разработки проекта. Для решения указанной проблемы разработчику необходимо получить копию нужного релиза проекта, создать его ветку, а после внесения необходимых изменений выполнить их слияние и синхронизацию.

Система CVS представляется весьма полезным средством для версионирования проекта при коллективной разработке сложных программных систем.

Система документирования исходных текстов DOC++ [114] позволяет готовить высококачественную документацию по программной системе непосредственно в процессе работы с исходными текстами программ на языке С. Данный подход позволяет выполнить требование самодокументируемости исходных текстов и при этом избежать дублирования информации в исходных текстах и в документации.

Основная идея документатора состоит во введении специального вида комментариев следующего формата: **/** . . . */.** Такие комментарии, в отличие от комментариев с одной звездочкой вначале, анализируются документатором и переводятся в формат HTML. При этом документатор использует ряд зарезервированных слов, помещаемых в комментарий, и начинающихся с символа **@**. На Рис. 6.2 приведен пример фрагмента программы на языке Си, содержащий спецификацию функции, подготовленную для документатора DOC++. Исходные тексты, подготовленные подобным образом, автоматически компилируются документатором в качественную документацию в формате HTML.

```
/**
После выполнения данной функции для реального старта операции, необходимо выполнить
операцию диспетчеризации th_schedule().
@memo Запуск асинхронной операции менеджера дисков.
@param page – номер страницы диска
@param buffer – указатель на буфер с данными
@param type – тип операции
\begin{verbatim}
        DS_READ     чтение данных с диска
        DS_WRITE    запись данных на диск
\end{verbatim}
@return неотрицательный уникальный идентификатор операции или отрицательный код
ошибки
begin{verbatim}
        ENOINIT     не было инициализации подсистемы
        EINVAL      неверный номер страницы
\end{verbatim}
@see th_schedule ds_ioid_t
*/
ds_ioid_t ds_runoperation(int page, ds_optype_t type, void *buffer);
```

Рис. 6.2 Пример спецификации функции

Основными преимуществами данного подхода к подготовке документации по проекту являются постоянная актуальность и адекватность документации, а также возможность работы с данной документацией на

всех аппаратно-программных платформах, имеющих HTML обозреватель. Кроме этого, данный подход обеспечивает возможность работы с документацией по проекту через Internet.

6.3.2 Средства расширения среды программирования

Наличие средств отладки и профилирования программ является одним из важных факторов обеспечения эффективной разработки программного обеспечения [7]. При разработке программ для многопроцессорных систем средства отладки и профилирования позволяют снизить сложность реализации взаимодействия и синхронизации параллельных процессов [37], и могут дополняться средствами визуализации параллельных процессов [39]. Разработанные средства расширения среды программирования (отладчик и профилировщик) предоставляют инструментарий, отсутствующий в среде МВС-100.

Отладчик выполняет трассировку программы и отладочную печать. Интерфейс отладчика приведен на Рис. 6.3.

```
/* Печать отладочного сообщения.
format – строка форматирования
[, argument, ...] – список переменных */
void db_printf(char *format [, argument, ...]);

/* Установка режима отладки.
mode – режим (1 – дамп, 0 – трассировка). */
void db_setmode(int mode);

/* Установка номеров процессоров, для которых следует выводить отладочные сообщения.
scale – шкала отладочной печати: если i-й бит в scale установлен, то сообщение будет
выводиться на i-м процессоре */
void db_setscale(int scale);
```

Рис. 6.3 Интерфейс отладчика

Отладчик позволяет задавать номера процессоров, для которых следует выводить отладочные сообщения. В отладочные сообщения автоматически могут включаться номера процессоров. Отладчик поддерживает два режима работы: режим выдачи отладочной информации в виде дампа и режим трассировки, обеспечивающий пошаговое выполнение программы.

Разумеется, подобный отладчик не может заменить полноценный символьный отладчик, однако его использование, как показывает опыт, позволяет существенно повысить эффективность процесса отладки.

Профилировщик предоставляет средства для учета общего объема данных, вовлекаемых в обмен с диском и в передачу по линкам. Функции профилировщика используются для оптимизации быстродействия задач, обрабатывающих большие объемы данных. Интерфейс профилировщика представлен на Рис. 6.4.

```
/* Создание профилировочного тега.
tag – профилировочный тег */
int pf_create(int tag);

/* Начало профилирования по тегу.
tag – профилировочный тег */
int pf_start(int tag);

/* Окончание профилирования по тегу.
tag – профилировочный тег */
int pf_stop(int tag);

/* Число операций чтения по линку.
tag – профилировочный тег */
int pf_linkread(int tag);

/* Число операций записи по линку.
tag – профилировочный тег */
int pf_linkwrite(int tag);

/* Число операций чтения с диска.
tag – профилировочный тег */
int pf_diskread(int tag);

/* Число операций записи на диск.
tag – профилировочный тег */
int pf_diskwrite(int tag);
```

Рис. 6.4 Интерфейс профилировщика

Профилировщик использует концепцию профилировочных тегов. *Профилировочный тег* является некоторым идентификатором, позволяющим пользователю суммировать результаты профилирования, полученные при выполнении одной задачи на нескольких процессорных узлах.

Функция **pf_create** создает в таблице профилировщика учетную запись с указанным тегом. Ключевыми атрибутами записи являются тег и идентификатор текущей нити управления. Учетная запись, кроме того, содержит поля, в которых будут суммироваться передачи по линкам (сообщения) и обмены с дисками, инициируемые данной нитью управления.

Функция **pf_start** обнуляет счетчики сообщений и обменов, и включает режим профилирования для указанного тега. Функция **pf_stop** выключает режим профилирования для указанного тега.

Каждый обмен с диском и каждая передача сообщения, выполняемая некоторой нитью, увеличивают соответствующие счетчики всех своих профилировочных записей, для которых включен режим профилирования.

Профилировочный тег, по существу, является некоторым идентификатором, позволяющим пользователю суммировать результаты профилирования, полученные при выполнении одной задачи на нескольких процессорах.

6.3.3 Средства поддержки интегрированной среды разработки

Интегрированная среда разработки предоставляет участникам проекта удобные средства для создания, компиляции и тестирования программ на МВС-100. Необходимость создания подобной среды обусловлена тем, что базовое программное обеспечение, поставляемое с МВС-100, не предоставляет готовой интегрированной среды разработки.

В качестве компилятора исходных текстов на МВС-100 используется *компилятор Portland Group C (PGCC)* [99] для суперскалярного процессора i860 фирмы Intel. Данный компилятор является совместимым со стандартом ANSI C и разработан специально для достижения максимальной производительности на суперкомпьютерах, разработанных на базе i860. Компилятор PGCC поддерживает глубокую векторизацию, открытую подстановку функций и многоуровневую оптимизацию программного кода.

PGCC включает в себя компоновщик, позволяющий получать загрузочные модули, которые могут быть непосредственно выполнены на i860. Компилятор также включает в себя средства создания библиотек объектных модулей (в виде программы-библиотекаря ar860), которые могут использоваться при компоновке программ. Версия компилятора PGCC, поставляемая с МВС-100, разработана для среды DOS и представляет собой

кросскомпилятор. К недостаткам данной версии следует отнести отсутствие символьного отладчика и профилировщика.

Запуск загрузочных модулей на МВС-100 осуществляется с помощью *загрузчика run860*, работающего в среде MS-DOS. Загрузчик запускает на host-машине программу iserver.exe (host-сервер ОС Router), которая обеспечивает запуск задачи пользователя с host-машины на процессорные модули МВС-100.

Поскольку базовое программное обеспечение, поставляемое с МВС-100, предназначено для работы в среде MS-DOS, первоначально в качестве host-машины использовался сервер, работающий под управлением ОС MS-DOS и находящийся в одной локальной сети с компьютерами разработчиков. Данная среда разработки обладает следующими недостатками:

- не поддерживается доступ к репозиторию проекта;
- существенно затруднена коллективная разработка программ, поскольку на стадиях отладки и тестирования требуется присутствие разработчика за консолью host-машины;
- отсутствует возможность работы в данной среде из глобальной сети Internet, поскольку MS-DOS не предоставляет средств поддержки удаленного доступа.

В соответствии с этим были сформулированы следующие требования к интегрированной среде разработки программ для МВС-100:

- мобильность, то есть прозрачный доступ к компонентам среды с различных аппаратно-программных платформ;
- поддержка многопользовательского режима разработки;
- возможность организации удаленных рабочих мест для работы из Internet.

В качестве основы формирования интегрированной среды разработки, удовлетворяющей данным требованиям, была выбрана ОС UNIX/Linux.

Структура интегрированной среды разработки изображена на Рис. 6.5. Центральным звеном интегрированной среды является Linux-сервер, играющий роль host-машины для MBC-100. На данном сервере устанавливаются система контроля версий CVS, документатор DOC++ и компилятор PGCC, описанные выше. Для запуска PGCC и run860, в том числе удаленного, используется DOS-эмулятор. Это позволяет вписать весь технологический цикл разработки в рамки одной операционной системы UNIX/Linux.

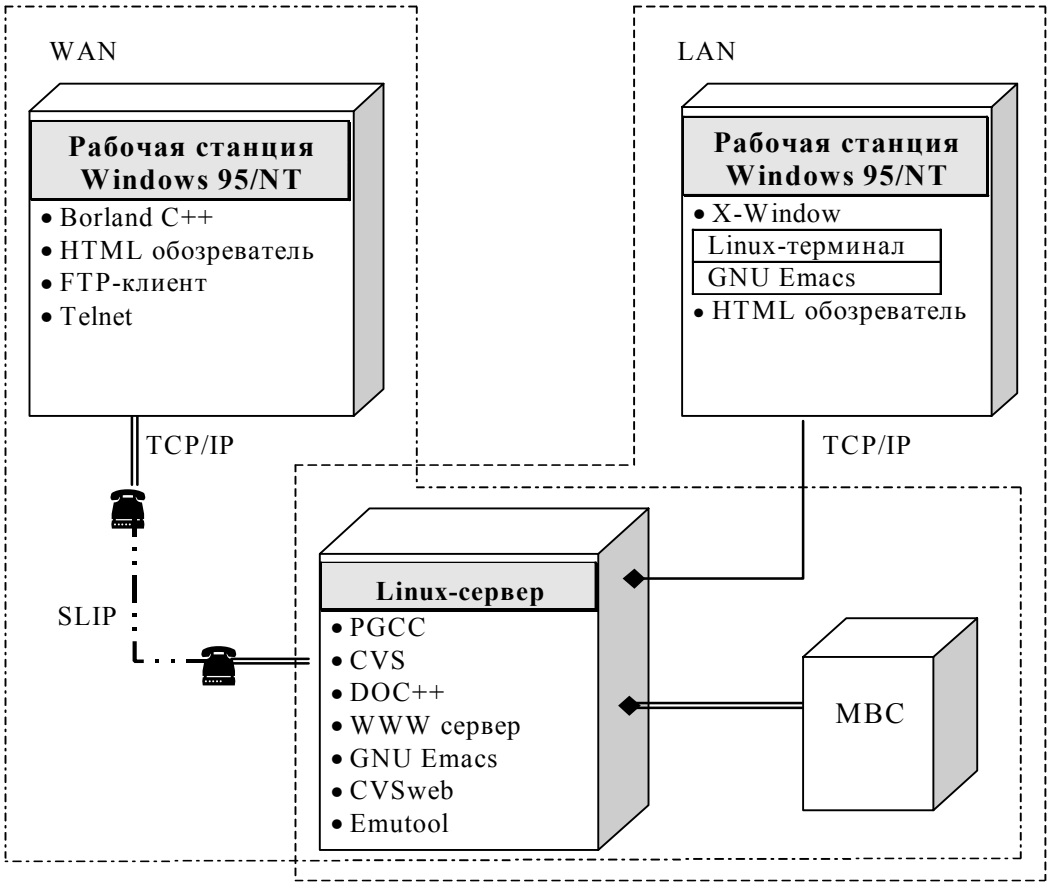


Рис. 6.5 Интегрированная среда разработки комплекса Омега

Интегрированная среда предусматривает два варианта рабочих места программиста. *Локальное рабочее место* предназначено для разработчиков, компьютеры которых находятся в локальной сети, включающей в себя host-машину для MBC-100. На рабочих станциях разработчиков устанавливается ОС Windows 98/NT. Для работы с Linux-сервером на рабочих станциях используются X-терминалы. В качестве редактора используется

редактор GNU Emacs [61], имеющий встроенную поддержку языка Си и встроенный интерфейс с системой контроля версий CVS. Фактически Emacs играет роль интегрирующей оболочки, позволяющей редактировать, компилировать и запускать программы на языке Си. *Удаленное рабочее место* предназначено для работы из Internet. На удаленной рабочей станции устанавливается ОС Windows 98/NT. В качестве UNIX-терминала используется Telnet. Копии исходных текстов получаются с Linux-сервера по FTP и находятся на удаленной рабочей станции. В качестве редактора в данном варианте рабочего места может использоваться, например, оболочка Borland C++, предоставляющая мощные средства для компиляции, запуска и отладки аппаратно независимых частей программной системы.

Система контроля версий CVS, как и многие утилиты ОС UNIX, поддерживает лишь интерфейс командной строки. При работе с репозиторием проекта данное обстоятельство в определенной степени затрудняет отслеживание изменений в исходных текстах и получение последних версий файлов. Для решения данной проблемы на host-машине был установлен web-сервер и пакет CVSweb [73]. Пакет CVSweb представляет собой web-оболочку для CVS и дает возможность просматривать содержимое репозитория, созданного CVS, с помощью HTML обозревателя. При просмотре информация отображается в виде соответствующей структуры каталогов и файлов проекта. С каждым именем файла связывается коллекция гиперссылок, при выборе которых можно загрузить соответствующую ревизию этого файла. CVSweb также предоставляет web-форму, с помощью которой можно отобразить различия между произвольными ревизиями одного файла, в том числе в различных ветках проекта. Установка CVSweb предоставляет разработчикам прозрачный способ отслеживания изменений в исходных текстах проекта и получения их последних версий.

Для компиляции исходных текстов и запуска полученных загрузочных модулей на МВС-100 разработчиками использовался эмулятор MS-DOS в среде UNIX. Однако использование эмулятора накладывает оп-

ределенные ограничения на ресурсы, выделяемые для DOS-приложений. Например, использование оболочки Borland C++ в эмуляторе является затруднительным. Помимо этого при работе через Telnet не полностью эмулируются функциональная клавиатура и мышь. То есть эмулятор фактически предоставляет удаленному разработчику возможность работы только в режиме командной строки MS-DOS. Таким образом, использование DOS-эмулятора представляет собой дополнительный этап, ухудшающий эргономические показатели среды разработки, и затрудняющий отладку и тестирование программ на МВС-100.

Для решения указанной проблемы был разработан пакет EMUtool [60], с помощью которого от разработчика скрывается использование MS-DOS и обеспечивается запуск DOS-приложений непосредственно из командного интерпретатора UNIX/Linux. EMUtool представляет собой набор программ (скриптов) на языке shell [19], использующих специализированную конфигурацию эмулятора MS–DOS для "беспилотного" запуска DOS-программ в среде ОС UNIX/Linux. Пакет включает в себя фильтр, который приводит диагностические сообщения DOS-программ к стандартному виду, пригодному для интеграции с другими средствами UNIX. В результате запуск DOS-программ в среде ОС UNIX/Linux производится командной строкой DOS-формата. Такое решение обеспечивает разработчикам удобный способ компиляции программ с помощью PGCC и запуск полученных загрузочных модулей на МВС-100 как с локального, так и с удаленного рабочего места.

Таким образом, средства поддержки интегрированной среды разработки обеспечивают возможность выполнения всех технологических операций как в локальной сети (intranet), так и в глобальной сети Internet. При этом интерфейсы локального и удаленного рабочих мест практически идентичны, что позволяет одному и тому же программисту работать как на локальной рабочей станции, так и на компьютере, подключенном к сети через Internet.

ЗАКЛЮЧЕНИЕ

В диссертационной работе были рассмотрены методы построения комплекса системных программ для управления данными в многопроцессорных вычислительных системах с массовым параллелизмом. В качестве платформы для практической реализации указанных методов использовался отечественный многопроцессорный вычислительный комплекс МВС-100. Получены следующие основные результаты:

1. Предложена методика построения программного комплекса для хранения и передачи данных в вычислительных системах с массовым параллелизмом, учитывающая особенности архитектуры современных многопроцессорных систем.
2. Разработана и реализована система управления файлами для многопроцессорного вычислительного комплекса МВС-100, включающая в себя следующие подсистемы:
   - менеджер наборов, поддерживающий представление данных в виде совокупности наборов (связных списков) страниц и обеспечивающий буферизацию данных на основе единого буферного пула;
   - менеджер файлов, поддерживающий представление данных в виде файлов (именованных множеств неструктурированных записей одинаковой длины) и обеспечивающий стандартные средства для управления данными;
   - электронную дисковую подсистему (ЭДП), реализующую виртуальный модуль дисковой подсистемы и состоящую из сервера ЭДП и драйвера ЭДП, поддерживающих страничное представление диска и обеспечивающих операции чтения-записи данных на основе эффективных протоколов взаимодействия.
3. Разработан метод вытеснения страниц из буферного пула, получивший название DIR-метода. DIR-метод базируется на введении стати-

ческого и динамического рейтингов страниц и использовании избыточного индекса буферного пула (DIR). DIR-метод позволяет:

- моделировать практически любую общую стратегию вытеснения страниц;
- назначать и динамически изменять стратегию вытеснения для отдельных наборов страниц.

4. Предложена методика построения эффективных стратегий вытеснения страниц, базирующихся на DIR-методе. Проведены численные эксперименты, подтверждающие более высокую эффективность DIR-стратегий по сравнению с классическими.
5. Предложена технология разработки больших программных комплексов для многопроцессорной вычислительной системы МВС-100. Данная технология апробирована при разработке программного комплекса Омега и обеспечивает:
   - поддержку коллективной разработки программных комплексов для МВС-100 на этапах кодирования, отладки, тестирования и сопровождения;
   - среду программирования с инструментарием отладки и профилирования параллельных программ на МВС-100;
   - возможность участия в разработке удаленных разработчиков, предоставляя удаленный доступ к программным ресурсам host-компьютера и МВС-100.

ЛИТЕРАТУРА

1. *Андреев А., Воеводин В., Жуматий С.* Кластеры и суперкомпьютеры: близнецы или братья? // Открытые системы. - 2000. - № 5-6. - С. 9-14.
2. *Бек К.* Экстремальное программирование // Открытые системы. - 2000. - № 1-2. - С. 59-65.
3. *Бернштейн Ф. и др.* Программа исследований в области баз данных на следующее десятилетие // Открытые системы. - 1999. - № 1. - С. 61-68.
4. *Брукс Ф.П.* Как проектируются и создаются программные комплексы. - М.: Наука, 1979. - 252 с.
5. *Бэбб Р., Мак-Гроу Дж. и др.* Программирование на параллельных вычислительных системах. - М.: Мир, 1991. - 376 с.
6. *Валях Е.* Последовательно-параллельные вычисления. - М.: Мир, 1985. - 456 с.
7. *Ван Тассел Д.* Стиль, разработка, эффективность, отладка и испытание программ. - М.: Мир, 1985. - 281 с.
8. *Вельбицкий И.В.* Технология программирования. - Киев: Техніка, 1984. - 250 с.
9. *Воеводин Вл.В., Капитонова А.П.* Методы описания и классификации вычислительных систем. Учебное пособие. - М.: Изд.-во МГУ, 1994. - 103 с.
10. *Воронков Б.В., Масленников М.В.* Математическая модель физических процессов в ядерном реакторе // Сб. науч. тр. Современные проблемы математической физики и вычислительной математики. - М.: Наука, 1982. - С. 76-101.
11. *Вьюкова Н.* Сервер для кластерных и массово-параллельных архитектур // Открытые системы. - 1995. - № 4. - С. 17-21.
12. *Головкин Б.А.* Параллельные вычислительные системы. - М.: Наука, 1980. - 520 с.

13. *Гольдштейн М.Л.* Мультипроцессорная вычислительная система на базе транспьютерной идеологии // Алгоритмы и программные средства параллельных вычислений. Сб. науч. тр. Екатеринбург: УрО РАН. - 1995. - С. 61-68.
14. *Гусев А.В., Луцкий А.Е., Петрушенков И.Л.* Приложение многопроцессорных систем в аэродинамическом проектировании самолетов // Вопросы атомной науки и техники. Серия "Математическое моделирование физических процессов". - 1992. - Вып. 3. - С. 11-14.
15. *Девитт Д., Грэй Д.* Параллельные системы баз данных: будущее высокоэффективных систем баз данных // СУБД. - 1995. - № 2. - С. 8-21.
16. *Дейтел Г.* Введение в операционные системы. - М.: Мир, 1987. - 231 с.
17. *Желиговский В.А., Пинский В.И., Розенберг В.Л.* Параллельная реализация блоковых моделей динамики литосферы // Распределенные системы: оптимизация и приложения в экономике и науках об окружающей среде. Екатеринбург. УрО РАН. - 2000. - С. 315-318.
18. *Забродин А.В., Левин В.К.* Опыт разработки параллельных вычислительных технологий. Создание и развитие семейства МВС // Высокопроизводительные вычисления и их приложения: Труды Всероссийск. науч. конф. (30 октября – 2 ноября 2000 г., г. Черноголовка). - М.: Изд.-во МГУ, 2000. - С. 3-8.
19. *Керниган Б.В., Пайк Р.* UNIX – универсальная среда программирования. - М.: Финансы и статистика, 1992. - 304 с.
20. *Кнут Д.* Искусство программирования для ЭВМ, т. 3, Сортировка и поиск. - М.: Мир, 1978. - 844 с.
21. *Ковалик Я.* Высокоскоростные вычисления. Архитектура, производительность, прикладные алгоритмы и программы суперЭВМ. - М.: Радио и связь, 1988. - 432 с.
22. *Кодд Е.Ф.* Реляционная модель для больших совместно используемых банков данных // СУБД. - 1995. - № 1. - С. 145-169.

23. *Коковихина О.В.* Параллельный вариант программы расчета параметров акустических колебаний в вихревых потоках // Алгоритмы и программные средства параллельных вычислений. Сб. науч. тр. Екатеринбург. УрО РАН. - 1998. - С. 150-162.
24. *Коковихина О.В.* Распараллеливание алгоритма решения задачи о распространении акустических колебаний в газовых потоках // Алгоритмы и программные средства параллельных вычислений. Сб. науч. тр. Екатеринбург. УрО РАН. - 1995. - С. 79-85.
25. *Короткий А.И., Решетов В.М., Цепелев А.И.* Применение многопроцессорных ЭВМ для моделирования движения вязкой среды // Высокопроизводительные вычисления и их приложения: Труды Всероссийск. науч. конф. (30 октября – 2 ноября 2000 г., г. Черноголовка). - М.: Изд.-во МГУ, 2000. - С. 265-268.
26. *Костоусов В.Б.* Реализация алгоритмов высокоточной навигации по геофизическим полям на параллельных вычислительных системах // Алгоритмы и программные средства параллельных вычислений. Сб. науч. тр. Екатеринбург. УрО РАН. - 1995. - С. 86-100.
27. *Кузнецов С.Д.* Операционные системы для управления базами данных // СУБД. - 1996. - № 3. - С. 95-102.
28. *Кузьминский М., Волков Д.* Современные суперкомпьютеры: состояние и перспективы // Открытые системы. - 1995. - № 6. - С. 33-40.
29. *Лацис А.О.* Разработка ОС коллективного использования для многопроцессорной супер-ЭВМ МВС-100 // Транспьютерные системы и их применение: Тез. докл. Всероссийск. науч. конф. - М.: ИПМ им. Келдыша, 1995. - С. 17-24.
30. *Левин В.К.* Отечественные суперкомпьютеры семейства МВС. - http://parallel.ru/mvs/levin.html.

31. *Лымарь Т.Ю.* Уточнение стоимостной модели оптимизации запросов с учетом распределения процессоров // Алгоритмический анализ некорректных задач: Тез. докл. Всероссийск. науч. конф. (Екатеринбург, 2-6 февраля 1998 г.). - Екатеринбург: УрГУ, 1998. - С. 149-150.
32. *Лымарь Т.Ю., Соколинский Л.Б.* Инкапсуляция параллелизма в исполнителе запросов СУБД Омега // Высокопроизводительные вычисления и их приложения: Труды Всероссийск. науч. конф. (30 октября - 2 ноября 2000 г., г. Черноголовка). - М.: Изд.-во МГУ, 2000. - С. 136-140.
33. *Мельников И.А., Раабе А.С., Тамм Б.Г.* Инструментарий машинной поддержки цикла жизни программного обеспечения. Обзор западных средств // Прикладная информатика. - 1988. - Вып. 14. - С. 16-40.
34. *Мельникова Л.А., Розенберг В.Л.* Численное моделирование динамики блоковой структуры на МВС // Алгоритмы и программные средства параллельных вычислений. Сб. науч. тр. - Екатеринбург: УрО РАН, 1998. - С. 221-235.
35. *Митчел Д.А.П., Томпсон Дж.А., Мансон Г.А., Брукс Г.Р.* Внутри транспьютера. - М.: Мейкер, 1993. - 206 с.
36. *Оззу М., Валдуриз П.* Распределенные и параллельные системы баз данных // СУБД. - 1996. - № 4. - С. 4-26.
37. *Петренко А.К.* Методы отладки и мониторинга параллельных программ // Программирование. - 1994. - № 3. - С 39-63.
38. *Позин Б.А.* Современные средства программной инженерии для создания открытых прикладных информационных систем // СУБД. - 1995. - № 1. - С. 139-144.
39. *Самофалов В.В., Шарф С.В.* Визуальный процесс: проектирование, использование и роль в параллельном программировании // Алгоритмы и программные средства параллельных вычислений. Сб. науч. тр. - Екатеринбург: УрО РАН, 1995. - С. 170-181.

40. *Самофалов В.В., Василиади А.А.* Сборочное параллельное программирование // Вестник Челябинского университета. Серия математика, механика. - 1999. - № 2(5). - С. 161-175.

41. *Сафонов В.О.* Языки и методы программирования в системе "Эльбрус". - М.: Наука, 1989. - 392 с.

42. *Сидоров А.Ф., Гасилов В.Л., Кукушкин А.П.* Разработка высокопроизводительных алгоритмических и программных средств на базе параллельных технологий // Алгоритмы и программные средства параллельных вычислений. Сб. науч. тр. – Екатеринбург: УрО РАН, 1995. - С. 3-20.

43. *Сингер М.* Мини-ЭВМ PDP-11: Программирование на языке ассемблера и организация машины. - М.: Мир, 1984. - 272 с.

44. *Соколинский Л.Б.* Программная поддержка ТИП-технологии программирования на МВК "Эльбрус" // Технология программирования, инструментальное и системное программное обеспечение ЭВМ: Тез. докл. региональн. конф. - Пермь: ПГУ, 1989. - С. 32-33.

45. *Соколинский Л.Б.* Разработка параллельной системы управления базами данных для мультипроцессорной вычислительной системы МВС-100 // Информационный бюллетень Ассоциации математического программирования. - Екатеринбург: УрО РАН, 1997. - № 7. - С. 210-211.

46. *Соколинский Л.Б.* Эффективная организация легковесных процессов в параллельной СУБД Омега для МВС-100 // Фундаментальные и прикладные аспекты разработки больших распределенных программных комплексов: Тез. докл. Всероссийск. науч. конф. (21-26 сентября 1998 г., г. Новороссийск). - М.: Изд.-во МГУ, 1998. - С. 132-138.

47. *Соколинский Л.Б.* Структура средств компьютерной поддержки процесса прототипирования параллельной СУБД Омега для мультипроцессорной вычислительной системы МВС-100/1000 // Программные продукты и системы. - 1999. - № 2. - С. 15-19.

48. *Соколинский Л.Б.* Проектирование и анализ архитектур параллельных машин баз данных с высокой отказоустойчивостью // Высокопроизводительные вычисления и их приложения: Труды Всероссийск. науч. конф. (30 октября – 2 ноября 2000 г., г. Черноголовка). - М.: Изд.-во МГУ, 2000. - С. 56-61.

49. *Соколинский Л.Б., Лымарь Т.Ю.* О выборе оптимального плана выполнения запроса в параллельной системе баз данных // Проблемы оптимизации и экономические приложения: Тез. докл. междунар. конф. - Омск: ОмГУ, 1997. - С. 146.

50. *Соколинский Л.Б., Сбитнев К.В.* Internet версия электронного толкового словаря по программированию и базам данных // Научный сервис в сети Интернет: Тез. докл. Всероссийск. науч. конф. (20-25 сентября 1999 г., г. Новороссийск). - М.: Изд.-во МГУ, 1999. - С. 234-239.

51. *Соколинский Л.Б., Цымблер М.Л.* Проект создания параллельной СУБД Омега на базе суперкомпьютера МВС-100/1000 // Телематика'98: Тез. докл. Всероссийск. науч.-метод. конф. (7-10 июня 1998 г., Санкт-Петербург). - СПб: Вузтелекомцентр, 1998. - С. 154-155.

52. *Соколинский Л.Б., Цымблер М.Л.* Использование МВС-100 в качестве машины баз данных // Информационный бюллетень Ассоциации математического программирования. - Екатеринбург: УрО РАН, 1999. - № 8. - С. 251-252.

53. *Соколинский Л.Б., Цымблер М.Л.* Принципы реализации системы управления файлами в параллельной СУБД Омега для МВС-100 // Вестник Челябинского университета. Серия математика, механика. - 1999. - № 2(5). - С. 176-199.

54. *Фернбах С.* Супер ЭВМ. Аппаратная и программная организация. - М.: Радио и связь, 1991. - 320 с.
55. *Фуксман А.А.* Технологические аспекты создания программных систем. - М.: Статистика, 1979. - С. 184.
56. *Хокни Р., Джессхоуп К.* Параллельные ЭВМ. Архитектура, программирование и алгоритмы. - М.: Радио и связь, 1986. - 392 с.
57. *Цепелев И.А., Короткий А.И. и др.* Параллельные алгоритмы решения задачи моделирования высоковязких течений в верхней мантии // Алгоритмы и программные средства параллельных вычислений. Сб. науч. тр. - Екатеринбург: УрО РАН, 1998. - С. 301-317.
58. *Цымблер М.Л., Соколинский Л.Б.* Выбор оптимальной стратегии вытеснения страниц в параллельной СУБД Омега для мультипроцессорной системы МВС-100 // Распределенные системы: оптимизация и приложения в экономике и науках об окружающей среде (DSO'2000). Сб. докл. к Междунар. конф. (Екатеринбург, 30 мая – 2 июня 2000 г.). - Екатеринбург: УрО РАН, 2000. - С. 337-340.
59. *Цымблер М.Л., Соколинский Л.Б.* Организация обработки больших объемов данных в многопроцессорных системах с массовым параллелизмом // Высокопроизводительные вычисления и их приложения: Труды Всероссийск. науч. конф. (30 октября – 2 ноября 2000 г., г. Черноголовка). - М.: Изд.-во МГУ, 2000. - С. 186-190.
60. *Цымблер М.Л., Соколинский Л.Б., Федрушков В.В.* Использование Internet-технологий в коллективной разработке больших программных систем // Научный сервис в сети Интернет: Тез. докл. Всероссийск. науч. конф. (20-25 сентября 1999 г., г. Новороссийск). - М.: Изд.-во МГУ, 1999. - С. 207-210.
61. *Шалунов С.В.* Операционная среда Emacs // Открытые системы. - № 4. - 1997. - С. 11-15.
62. *Шэнк Дж.* Технология клиент/сервер и ее приложения. - М.: ЛОРИ, 1995. - 418 с.

63. *Bancilhon F.* Object Databases // ACM Computing Surveys. - March 1996. - Vol. 28. - No. 1. - P. 137-140.
64. *Baru C.K., et al.* DB2 Parallel Edition // IBM System Journal. - 1995. - Vol. 34. - No. 2. - P. 292-322.
65. *Belady L.A.* A Study of Replacement Algorithms for Virtual Storage Computers // IBM Systems Journal. - 1966. - Vol. 5. - No. 2. - P. 78-101.
66. *Bell D., Morrey I., Pogh J.* Software Engineering. A programming Approach. - Prentice Hall, 1992. - 338 P.
67. *Berliner B.* CVS: Parallelizing Software Development. - http://www.hu.freebsd.org/hu/doc/psd/28.cvs/paper.html
68. *Bhide A.* An Analysis of Three Transaction Processing Architectures // Proc. of the Int. Conf. on Very Large Data Bases (VLDB'88), August 29 – September 1, 1988, Los Angeles, California, USA,. - Morgan Kaufmann, 1988. - P. 339-350.
69. *Boral H., et al.* Prototyping Bubba: a Highly Parallel Database System // IEEE Transactions on Knowledge and Data Engineering. - March 1990. - Vol. 2. - No. 1. - P. 4-24.
70. *Bouganim L., Florescu D., Valduriez P.* Dynamic Load Balancing in Hierarchical Parallel Database Systems // Proc. of the Int. Conf. on Very Large Data Bases (VLDB'96), Mumbai (Bombay), India. September 1996. - P. 436-447.
71. *Copeland G., Keller T., Smith M.* Database Buffer and Disk Configuring and the Battle of the Bottlenecks // Proc. of the 4th Int. Workshop on High Performance Transaction Systems, September 1991. - P. 94-102.
72. *Corbato F.J.* A Paging Experiment with the Multics System. MIT Project MAC Report MAC-M-384. - 1968. - 127 P.
73. Cyclic CVSweb page. - http://www.cyclic.com/cyclic-pages/web-cvsweb.html
74. *Dasgupta S.* A Hierarchical Taxonomic System for Computer // Computer. - 1990. - Vol. 23. - No. 3. - P. 64-74.

75. *Date C.J.* An Introduction to Database Systems (6th edition). Reading, Mass.: Addison-Wesley, 1995. - P. 686.
76. *DeWitt D.J., et al.* The Gamma Database Machine Project // IEEE Transactions on Knowledge and Data Engineering. - March 1990. - Vol. 2. - No. 1. - P. 44-62.
77. *Dozier J.* Access to Data in NASA's Earth Observing System // Proc. of the 1992 ACM SIGMOD Int. Conf. on Management of Data, San Diego, California, June 2-5, 1992. - ACM Press, 1992. - P. 1-3.
78. *Effelsberg W., Harder T.* Principles of Database Buffer Management // ACM Transactions on Database Systems. - December 1984. - Vol. 9. - No. 4. - P. 560-595.
79. *Flynn M.J., Rudd K.W.* Parallel architectures // ACM Computing Surveys. - March 1996. - Vol. 28. - No. 1. - P. 67-70.
80. *Flynn M.* Some Computer Organizations and Their Effectiveness // IEEE Transactions on Computers, 1972. - Vol. 21. - No. 9. - P. 948-960.
81. *Frew J., Dozier J.* Data Management for Earth System Science // ACM SIGMOD Record. - March 1997. - Vol. 26. - No. 1. - P. 27-31.
82. *Garcia-Molina H., Ullman J.D., Widom J.* Database System Implementation. - Prentice Hall, 2000. - 653 P.
83. *Golubchik L., Muntz R.R.* Fault Tolerance Issues in Data Declustering for Parallel Database Systems // Data Engineering Bulletin. - 1994. - Vol. 17. - No. 3. - P. 14-28.
84. *Handler W.* The Impact Classification Schemes on Computer Architecture // Proc. of the Int. Conf. on Parallel Processing. - 1977. - P. 7-15.
85. *Hockney R.* Parallel Computers: Architecture and Performance // Proc. of the Int. Conf. on Parallel Computing. - 1986. -P. 33-69.

86. *Hua K.A., Lee C., Peir J.-K.* Interconnecting Shared-Everything Systems for Efficient Parallel Query Processing // Proc. of the 1st Int. Conf. on Parallel and Distributed Information Systems (PDIS'91), Fontainebleu Hilton Resort, Miami Beach, Florida, December 4-6, 1991. - IEEE-CS, 1991. - P. 262-270.
87. *Jeon H.S., Noh S.H.* A Database Disk Buffer Management Algorithm Based on Prefetching // Proc. of the 1998 ACM CIKM Int. Conf. on Information and Knowledge Management, Bethesda, Maryland, USA, November 3-7, 1998. - ACM Press, 1998. - P. 167-174.
88. *Johnson T., Shasha D.* 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm // Proc. of the Int. Conf. on Very Large Data Bases (VLDB'94). - 1994. - P. 439-450.
89. *Kaplan J.* Buffer Management Policies in Database System. Master of Science Thesis. - University of California, Berkeley, 1980.
90. *Lakshmi M.S., Yu P.S.* Effectiveness of Parallel Joins // IEEE Transactions on Knowledge and Data Engineering. - December 1990. - Vol. 2. - No 4. - P. 410-424.
91. *Lymar T.Y., Sokolinsky L.B.* Data Streams Organization in Query Executor for Parallel DBMS // Proc. of the 4th IEEE Int. Workshop on Databases and Information Systems (DB&IS), Lithuania, Vilnius, May 1-5, 2000. - Vilnius: Technica, 2000. - Vol. 1. - P. 85-88.
92. *Mattson R.L. et al.* Evaluation techniques for storage hierarchies // IBM Systems Journal. - June 1970. - No. 9. - P. 78-117.
93. *Mohan C., Pirahesh H., Tang W. G., Wang Y.* Parallelism in Relational Database Management Systems // IBM Systems Journal. - 1994. - Vol. 33. - No. 2. P. 349-371.
94. *Nicola V.F., Dan A., Dias D.M.* Analysis of the Generalized Clock Buffer Replacement Scheme for Database Transaction Processing // ACM SIGMETRICS Performance Evaluation Review. - June 1992. - Vol. 20. - No. 1. - P. 34-46.

95. *O'Neil E.J., O'Neil P.E., Weikum G.* The LRU-K Page Replacement Algorithm For Database Disk Buffering // Proc. of the 1993 ACM SIGMOD Int. Conf. on Management of Data, Washington, D.C., May 26-28, 1993. - ACM Press, 1993. - P. 297-306.
96. *Orfali R, Harkey D., Edwards J.* Essential Client/Server Survival Guide. - NY: John Wiley, 1994. - P. 109.
97. *Page J.* A Study of a Parallel Database Machine and its Performance: the NCR/Teradata DBC/1012 // Proc. of the 10th British National Conf. on Databases (BNCOD'10) Aberdeen, Scotland, July 6-8, 1992. Lecture Notes in Computer Science. - Springer, 1992. - Vol. 618. - P. 115-137.
98. *Pfister G.* Sizing Up Parallel Architectures // DataBase Programming & Design OnLine (http://www.dbpd.com). - May 1998. - Vol. 11. - No. 5.
99. PGI Supercompilers and Advanced Development Tools for the Intel i860. - http://www.pgroup.com/i860_home.html
100. *Robinson J.T., Devarakonda N.V.* Data Cache Management Using Frequency-Based Replacement // Proc. of the 1990 ACM SIGMETRICS Int. Conf. - 1990. - P. 134-142.
101. *Skillicorn D.* A Taxonomy for Computer Architectures // Computer. – 1988. - Vol. 21. - No. 11. - P. 46-57.
102. *Smith A.J.* Disk Cache-Miss Ratio Analysis and Design Considerations // ACM Transactions on Computer Systems. - August 1985. - Vol. 3. - No 3. - P. 161-203.
103. *Sokolinsky L., Axenov O., Gutova S.* Omega: The Highly Parallel Database System Project // Proc. of the 1st East-European Symposium on Advances in Database and Information Systems (ADBIS'97), St.-Petersburg, September 2-5, 1997. - Vol. 2. - P. 88-90.
104. *Sokolinsky L.B.* Interprocessor Communication Support in the Omega Parallel Database System // Proc. of the 1st Int. Workshop on Computer Science and Information Technologies (CSIT'99), Moscow, Russia, January 18-22, 1999. - MEPhI Publishing, 1999. - Vol. 2. - P. 114-123.

105. *Sokolinsky L.B.* Operating System Support for a Parallel DBMS with an Hierarchical Shared-Nothing Architecture // Advances in Databases and Information Systems, 3rd East European Conf. (ADBIS'99), Maribor, Slovenia, September 13-16, 1999. Proc. of Short Papers. - Maribor University Publishing, 1999. - P. 38-45.
106. *Sokolinsky L.B.* Choosing Multiprocessor System Architecture for Parallel Database Systems // Proc. of the 2nd Int. Workshop on Computer Science and Information Technologies (CSIT'2000), Ufa, September 18-23, 2000. - Ufa State Aviation Technical University, 2000. - Vol. 1. - P. 179-186.
107. *Stonebraker M., Hellerstein J.M.* Introduction to Chapter 5: Parallel Database Systems / Readings in database systems (3rd ed.). - Morgan Kauffman Publishers, 1998. - P. 399-402.
108. *Stonebraker M.* Operating System Support for Database Management // Communications of the ACM. - July 1981. - Vol. 24. - No. 7. - P. 412-418.
109. *Stonebraker M.* The case for shared nothing // Database Engineering Bulletin. - March 1986. - Vol. 9. - No. 1. - P. 4-9.
110. *Tandem Database Group.* NonStop SQL: A Distributed, High-Performance, High-Availability Implementation of SQL // Proc. of the 2nd Int. Workshop on High Performance Transaction Systems, Pacific Grove, California, USA, September 28-30, 1987. Lecture Notes in Computer Science. – Springer, 1989. - Vol. 359. - P. 60-104.
111. *Thakkar S. S., Sweiger M.* Performance of an OLTP Application on Symmetry Multiprocessor System // Proc. of the 17th Annual Int. Symposium on Computer Architecture. Seattle, WA, June 1990. - IEEE Computer Society Press, 1990. - P. 228-238.
112. *Valduriez P.* Parallel Database Systems: Open Problems and New Issues // Distributed and Parallel Databases. - April 1993. - Vol. 1. - No. 2. - P. 137-165.

113. *Valduriez P.* Parallel Database Systems: the Case for Shared-something // Proc. of the 9th Int. Conf. on Data Engineering, April 19-23, 1993, Vienna, Austria. - IEEE Computer Society, 1993. - P. 460-465.
114. *Wunderling R., Zöckler M.* DOC++. A Documentation System for C/C++ and Java. - http://www.zib.de/Visual/software/doc++
115. *Xu Y., Dandamudi S.P.* Performance Evaluation of a Two-Level Hierarchical Parallel Database System // Proc. of the Int. Conf. on Computers and Their Applications, Tempe, Arizona, March, 1997. - P. 242-247.
116. *Zabrodin A.V., Levin V.K., Korneev V.V.* The Massively Parallel Computer System MBC-100 // Proc. of the Int. Conf. on Parallel Computing Technologies (PaCT'95). Lecture Notes in Computer Science. - 1995. - Vol. 964. - P. 342-356.
117. *Zymbler M.L.* Computer Aided Design Facilities for Prototyping the Omega DBMS // Proc. of the 1st Int. Workshop on Computer Science and Information Technologies (CSIT'99), January 18-22, 1999, Moscow, Russia. - MEPhI Publishing, 1999. - Vol. 2. - P. 124-131.
118. *Zymbler M.L., Sokolinsky L.B.* Implementation Principles of File Management System for Omega Parallel DBMS // Proc. of the 2nd Int. Workshop on Computer Science and Information Technologies (CSIT'2000), Ufa, Russia, September 18-23, 2000. - Ufa State Aviation Technical University, 2000. - Vol. 1. - P. 173-178.

ПРИЛОЖЕНИЕ

Протоколы взаимодействия клиентской и серверной частей системы хранения данных

Чтение страницы с диска

T	Клиент	T	Сервер

1.	if (запись_блокирована) {
	такт--;
	break;
	}
	запись_блокирована= TRUE;
	запись_запущена = TRUE;
	r_write(сервер, {ИД; клиент;
	"чтение_стр"; номер_страницы});
		1.	if (загрузка_диска) {
			такт --;
			break;
			}
			if (запись_блокирована[клиент]) {
			такт --;
			break;
			}
			чтение_страницы = TRUE;
			r_diskread(номер_страницы, info);
			запись_блокирована[клиент] = TRUE;
			r_write(клиент, {ИД; клиент;
			"читай_info"});

2.	if (!заголовок_получен) {
	такт--;
	break;
	}
	if (дескриптор.ИД_операции !=
	заголовок.ИД_операции) {
	такт--;
	break;
	}
	запись_блокирована = FALSE;
	заголовок_получен = FALSE;
	чтение_блокировано = TRUE;
	чтение_запущено = TRUE;
	r_read(сервер, info);
		2.	if (!t_diskread()) {
			такт --;
			break;
			}
			if (!t_write(клиент)) {
			такт --;
			break;
			}
			r_write(клиент, info);

3.	if (!t_read(сервер)) {
	такт--;
	break;
	}
	чтение_блокировано = FALSE;
	такт = -1;
		3.	if (!t_write(клиент)) {
			такт--;
			break;
			}
			запись_блокирована[клиент] = FALSE;
			чтение_страницы = FALSE;
			такт = -1;

Запись страницы на диск

T	Клиент

1.	if (запись_блокирована) {
	такт--;
	break;
	}
	запись_блокирована = TRUE;
	запись_запущена = TRUE;
	r_write(сервер, {ИД; клиент;
	"запись_стр"; номер_страницы});

2.	if (!t_write(сервер)) {
	такт --;
	break;
	}
	запись_запущена = TRUE;
	r_write(сервер, info);

3.	if (!t_write(сервер)) {
	такт --;
	break;
	}
	запись_блокирована = FALSE;
	такт=-1;

T	Сервер

1.	if (сохранение_диска) {
	такт --;
	break;
	}
	if (чтение_блокировано[клиент]) {
	такт --;
	break;
	}
	запись_страницы = TRUE;
	чтение_блокировано[клиент] = TRUE;
	r_read(клиент, info);

2.	if (!t_read(клиент)) {
	такт --;
	break;
	}
	r_diskwrite(номер_страницы, info);
	чтение_блокировано[клиент] = FALSE;

3.	if (!t_ diskwrite()) {
	такт --;
	break;
	}
	запись_страницы = FALSE;
	такт=-1;

Загрузка образа диска из файла на host-машине

T	Клиент

1.	if (запись_блокирована) {
	такт--;
	break;
	}
	запись_блокирована = TRUE;
	запись_запущена = TRUE;
	r_write(сервер, {ИД; клиент;
	"загрузка_диска"; имя_файла});

2.	if (!пришел_заголовок) {
	такт--;
	break;
	}
	if (дескриптор.ИД_операции !=
	заголовок.ИД_операции) {
	такт--;
	break;
	}
	заголовок_получен = FALSE;
	запись_блокирована = FALSE;
	такт = -1;

T	Сервер

1.	if (чтение_страницы || сброс_диска) {
	такт --;
	break;
	}
	if (запись_блокирована[клиент]) {
	такт --;
	break;
	}
	загрузка_диска = TRUE;
	запись_блокирована[клиент] = TRUE;
	disk_reset(имя_файла);
	r_write(клиент, {ИД; "завершена"});

2.	if (!t_write(клиент)) {
	такт--;
	break;
	}
	загрузка_диска = FALSE;
	запись_блокирована[клиент] = FALSE;
	такт = -1;

T	Клиент

1.	if (запись_блокирована) {
	  такт--;
	  break;
	}
	запись_блокирована = TRUE;
	запись_запущена = TRUE;
	r_write(сервер, {ИД; клиент;
	"сброс_диска"; имя_файла});

2.	if (!пришел_заголовок) {
	  такт--;
	  break;
	}
	if (дескриптор.ИД_операции !=
	заголовок.ИД_операции) {
	  такт--;
	  break;
	}
	заголовок_получен = FALSE;
	запись_блокирована = FALSE;
	такт = -1;

T	Сервер

1.	if (запись_страницы || загрузка_диска) {
	  такт --;
	  break;
	}
	if (запись_блокирована[клиент]) {
	  такт --;
	  break;
	}
	сброс_диска = TRUE;
	запись_блокирована[клиент] = TRUE;
	disk_dump(имя_файла);
	r_write(клиент, {ИД; "завершена"});

2.	if (!t_write(клиент)) {
	  такт--;
	  break;
	}
	сброс_диска = FALSE;
	запись_блокирована[клиент] = FALSE;
	такт = -1;