Оглавление

Тема 1. Введение в дисциплину

Создание программного обеспечения с некоторой точки зрения может рассматриваться как разновидность проектирования. Изначально есть осознание, что существует некоторая проблема, которую хотелось бы решить. Или некоторая проблема уже имеет приемлемое решение, но возникает идея расширения возможностей или качества этого решения. Далее проблема обдумывается, уточняется, подвергается структурированию и декомпозиции на более мелкие фрагменты. В результате этого: 1) демонстрируется возможность решения проблемы при имеющихся у разработчика технологиях и ресурсах, 2) формируется спецификация, которая детально описывает, какие задачи нужно решить для решения проблемы в целом. Спецификация позволяет исключить большие различия в понимании сути проблемы (заказчиками, разработчиками и пользователями), а иногда и принятых способов ее решения на высоком, абстрактном уровне.

Имея на входе разработанную спецификацию, разработчик проектирует программную систему. При этом, в частности, он синтезирует структуру системы, которую нужно построить. Для каждого элемента этой структуры определяется его назначение, взаимосвязь с другими элементами. Описание этих элементов (подсистем, компонентов, модулей, сервисов, классов) детализируется, уточняется список реализуемых ими функций. Одновременно с синтезом структуры изучаются существующие доступные решения. Те из них, которые приемлемо решают какие-либо задачи спецификации, заимствуются.

После проектирования происходит написание кода всех модулей системы, тестирование и отладка. От полученного кода в первую очередь требуется, чтобы он решал проблему так, как это описано в спецификации (функциональные требования). Когда это условие выполнено, следующим по важности является то, какое качество имеет построенное решение. Оно определяется так называемыми нефункциональными свойствами программного обеспечения. Среди таких свойств – надежность, эффективность реализации, переносимость, адаптируемость под

изменяющиеся требования, удобство пользовательского интерфейса, способность взаимодействовать с другими системами. Для разных задач ценность различных требований меняется. Однако, как правило, самым важным является надежность, а следующим за ним – эффективность реализации. Именно обеспечению этого качества посвящен данный курс. При достижении достаточного уровня надежности и соответствия функциональным требованиям во многих случаях свойством, в наибольшей степени определяющим качество построенной программной реализации, является эффективность реализации. Для повышения эффективности программного обеспечения производят его оптимизацию. Чтобы была возможность оценить, насколько успешно проведена оптимизация, нужно иметь средства ее изменения – числовые критерии, которые можно измерить или определить. Такие критерии оптимизации весьма разнообразны, например:

1) время выполнения программы, процедуры, обработки одного запроса, вычисления новых значений заданного числа элементов (клеток или точек на сетке);
2) размер бинарного кода программы;
3) время реакции на поступающие программе запросы;
4) объем данных, передаваемых при взаимодействии распределенного приложения через сеть;
5) энергия, затрачиваемая на выполнение некоторой операции во встраиваемой вычислительной системе;
6) объем внешней памяти, используемой программой при решении некоторой задачи.

Далее мы сосредоточимся на двух основных критериях:

1) Время выполнения (программы, подпрограммы),
2) Объем бинарного кода программы.

Наибольшее внимание мы уделим критерию времени выполнения программы, рассмотрев различные подходы к его уменьшению. Некоторые из других перечисленных критериев кратко рассмотрены в теме 7 “Специфика оптимизации ПО в основных проблемных областях”.

Чтобы эффективно оптимизировать программное обеспечение необходим определенный уровень понимания архитектуры и организации вычислительных систем, для этого в курс введен раздел 2 “Архитектура вычислительных систем”. Далее с аналогичными целями рассматривается архитектура ПО. Для ознакомления с инструментарием разработчика,

который используется при оптимизации, введен раздел 4. Ключевыми разделами курса являются раздел 5 “Оптимизация кода в компиляторах” и раздел 6 “оптимизация программного обеспечения разработчиком”. С целью дать более разностороннее представление об оптимизации в различных проблемных областях введен раздел 7.

Тема 2. Архитектура вычислительных систем

2.1. Понятие архитектуры (компьютеров, вычислительных систем) было введено Фредериком Бруксом при работе над серией машин IBM System 360. В широком смысле оно включает в себя как видимую программисту или программе “интерфейсную” сторону компьютера, так и внутреннее устройство, поддерживающее работу этого интерфейса. В узком смысле к архитектуре относят только интерфейсную часть (систему команд с форматами команд и обрабатываемых данных, видимые регистры и их назначение, организацию адресного пространства памяти и механизмов ввода/вывода и прерываний. Далее мы используем термин архитектура в этом узком смысле. Устройство компьютера, реализующее архитектуру будем называть микроархитектурой или организацией компьютера. Одна и та же архитектура может быть реализована практически бесчисленным количеством различных микроархитектур.

Знание архитектуры является необходимым при написании системных программ на ассемблере и языках низкого уровня. Для того, чтобы эти программы были эффективными кроме знания архитектуры, требуется и определенный уровень понимания принципов организации.

2.2. Архитектура Фон Неймана. С 1936 по 1944 несколько групп разработчиков независимо друг от друга выработали принципы построения цифровых вычислительных машин и реализовали первые работающие прототипы. В их числе немецкий исследователь Конрад Цузе (Conrad Zuse), американские исследователи Джон Атанасов (John Atanasov), Джон Мочли (John Mauchley), Джон Экерт (John Eckert) и Джон фон Нейман (John von Neumann). Принципы построения цифровой вычислительной машины (компьютера), выработанные коллективом Мочли, Экерта, фон Неймана и Голдштейна были сформулированы в черновике фон Неймана, и вошли в историю вычислительной техники как архитектура фон Неймана. Кратко рассмотрим их.

- **Структура системы.** Компьютер состоит из памяти, арифметическо-логического устройства, управляющего устройства и устройств ввода/вывода. Все эти составляющие соединяются между собой шиной.
- **Принцип программного управления.** Структура компьютера не зависит от решаемой на нем задачи. Компьютер управляется программой, состоящей из команд, хранящихся в памяти.
- **Принцип однородности памяти.** Команды программы и данные хранятся в одной и той же памяти.
- **Принцип адресности.** Вся память разбита на маленькие ячейки одинакового размера. Каждая из них имеет уникальное имя, называемое адресом. Процессор может в любой момент времени прочитать или записать любую ячейку по ее адресу.
- **Последовательное исполнение и переходы.** Команды выполняются последовательно, в том порядке, в котором они хранятся в памяти. Для изменения этого порядка исполнения вводятся команды условного и безусловного переходов.
- **Принцип двоичного кодирования.** Все данные и команды представлены числами в двоичном формате.

2.3. Современные архитектуры, их особенности.

Классическая архитектура фон Неймана в неизменном виде не позволяет повышать производительность компьютеров и вычислительных систем из-за заложенных в нее ограничений: последовательного исполнения программ, ограниченной пропускной способности общей шины. Также в процессе эволюции главных компонентов компьютера возник сильный дисбаланс в скорости работы центрального процессора и памяти. В дополнение ко всему этому произошел переход от систем, выполняющих только одну программу в данный момент времени, к многозадачным системам. Все это привело к ряду усовершенствований этой базовой архитектуры. Главные из них – это: 1) формирование нескольких классов архитектур по количеству команд и их сложности, 2) развитие иерархической памяти, 3) встраивание механизмов защиты, 4) введение разных видов параллельного выполнения программ. Рассмотрим их подробнее.

2.4. Классы архитектур по сложности системы команд.

Рассмотрим основные появившиеся классы архитектур с различными наборами команд.

Компьютер без системы команд (**ZISC** – Zero Instruction Set Code, или **NISC** – No Instruction Set Code) представляют собой специализированные вычислительные устройства, например, аппаратные реализации алгоритмов работы нейронных сетей. Эта архитектура не является разновидностью архитектуры фон Неймана. В качестве основной архитектуры для построения некоторой вычислительной системы ZISC практически никогда не применяется, но она может быть основой для различных ускорителей, специализированных счетных сопроцессоров, работающих как ведомые устройства в рамках вычислительной системы с обычной архитектурой фон Неймана.

Компьютер с одной командой (**OISC** – One Instruction Set Computer или **URISC** – Ultimate RISC). Наиболее известной архитектурой, которая относится к OISC, является TTA (Transport Triggered Architecture). Выполнение различных операций (арифметических, битовых) производится записью исходных данных в ячейки с определенными адресами и считыванием результатов из других ячеек с известными адресами. В TTA единственная команда – это пересылка данного из ячейки с одним адресом в другую ячейку. В чистом виде такой архитектурой достаточно сложно пользоваться, но сама идея достаточно широко применяется в большинстве архитектур, например, для работы с устройствами ввода/вывода через их порты (также ячейки памяти в общем адресном пространстве оперативной памяти или в отдельном адресном пространстве портов ввода/вывода).

Архитектура с минимальным набором команд (**MISC** – Minimal Instruction Set Computer) характеризуется очень небольшим числом простых команд. Примерная верхняя граница числа команд – 32 (не считая такие базовые, как NOP. CPUID). В качестве примера MISC архитектур можно привести форт-процессоры, например, GreenArrays F18 или Atmel Marc4.

Архитектура с уменьшенным набором команд (**RISC** – Reduced Instruction Set Computer) на настоящий момент является, пожалуй, наиболее распространённой. Характерные ее черты - относительно небольшое число команд, малое число их форматов, простота этих форматов, малое число режимов адресации, большой объем регистрового файла, аппаратные или программные способы эффективного использования этого регистрового файла, LOAD-STORE архитектура, отсутствие микропрограммного управления, трехадресная архитектура.

Исторически принципы RISC были сформулированы при анализе ограничений CISC архитектуры в начале 1980-х годов.

Достоинствами RISC являются:

1) Высокая скорость выполнения команд (следствие простоты форматов, а значит и декодера команд).
2) Одинаковое время выполнения для большинства команд (а, значит и высокая эффективность конвейерного исполнения команд)
3) Относительная простота создания компилятора, генерирующего эффективный код (следствие малого числа простых команд).
4) Минимизация негативного эффекта существенной разницы в скорости работы процессора и памяти (большой регистровый файл позволяет относительно редко обращаться к оперативной памяти).

К числу основных недостатков можно отнести низкую плотность кода по сравнению с кодом в CISC архитектурах.

К настоящему моменту RISC системы имеют широкое применение как в высокопроизводительных вычислительных системах, так и маломощных встраиваемых системах. В персональных компьютерах они применяются ограниченно.

Архитектура со сложным набором команд (**CISC** – Complex Instruction Set Architecture) характеризуется большим числом команд, разнообразием их форматов и используемых режимов адресации, микропрограммным управлением, малым объемом регистровой памяти процессора, Реализации CISC основываются на принципе микропрограммирования. Он заключается в способе выполнения команд процессора как последовательностей (или даже процедур), состоящих из более примитивных микрокоманд. Формат микрокоманд максимально упрощен по сравнению с командами.

К достоинствам CISC архитектур относятся легкость написания программ на ассемблере и высокая плотность кода. Недостатки – сложность написания эффективного оптимизирующего компилятора, низкая эффективность использования памяти. По мере эволюции CISC архитектур они сохраняли существующие команды для обратной совместимости и пополнялись новыми. В итоге некоторые из CISC имеют сотни команд. Анализ частоты их использования показал, что активно используется лишь некоторая их небольшая часть. Сложность декодера и наличие большого числа малоиспользуемых команд одновременно с малым объемом площади кристалла, выделенным для регистрового файла является некоторым перекосом, сильно снижающим эффективность/производительность компьютера.

CISC системы по-прежнему широко распространены, особенно архитектуры x86/x86_64, широко применяющиеся в персональных компьютерах и высокопроизводительных вычислительных системах.

Архитектура с очень длинными словами (**VLIW** - Very Long Instruction Word) (и ее дальнейшее развитие – **EPIC**) с точки зрения сложности системы команд отличается наиболее сложными форматами, в которых представимы связки параллельно исполняемых команд. На нижнем уровне команды имеют относительно простые форматы, но их композиции в связки команд увеличивают уровень сложности Детальнее класс рассмотрен чуть ниже – в подразделе о видах параллелизма, так как эта архитектура представляет собой пример интенсивного использования параллелизма на уровне команд.

2.5. Иерархия памяти в современных архитектурах.

Виды памяти. В современных компьютерах на самом верхнем уровне можно выделить два основных вида памяти: оперативную и внешнюю. Оперативная память служит для хранения обрабатываемых программной данных. Она непосредственно доступна из программы, обладает высокой скоростью доступа и имеет относительно небольшой размер. Внешняя память служит для долговременного хранения данных и программ. Ее объем велик по сравнению с оперативной памятью, однако она не доступна непосредственно из программы, и скорость доступа к ней на порядки медленнее.

Для реализации этих видов памяти применяется широкий спектр технологий. Некоторая их общая классификация с основными современными и некоторыми историческими классами приведена на рис. 2.5.1.

Рис. 2.5.1. технологии построения памяти

Технологии построения оперативной памяти. Исторически для построения оперативной памяти использовались весьма разнообразные технологии: трубка Вильямза, селектрон, память на магнитных сердечниках (см. рис. 2.5.2). С появлением транзистора в 1948 году в AT&T Bell Labs (и далее микросхем в 1958 в Texas Instruments) практически всегда для построения оперативной памяти стали использовать полупроводниковые технологии. На данный момент сформировались два основных класса полупроводниковой памяти, которые используются для построения оперативной памяти. Это – статическая и динамическая память (в контексте разработки программного обеспечения используются эти же термины, но с другим смыслом). Более широкий спектр технологий полупроводниковой памяти, применяемых как для построения оперативной, так и постоянной памяти приведен на рис. 2.5.3.

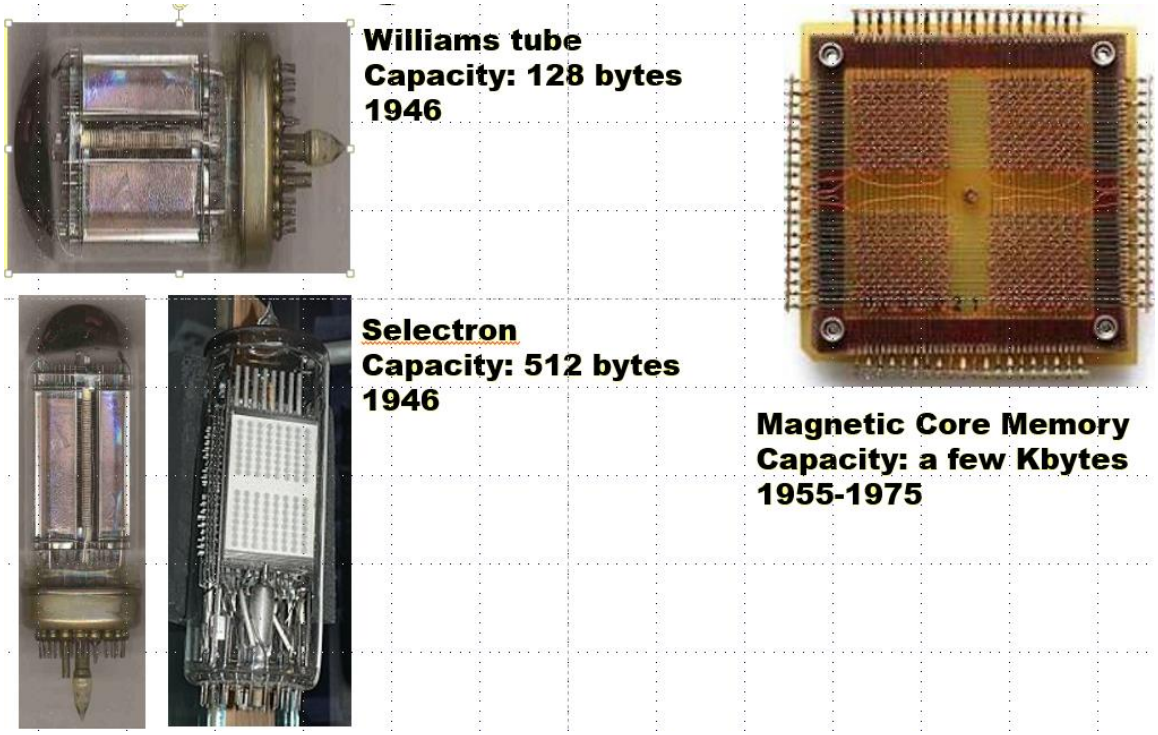

Рис. 2.5.2. технологии построения оперативной памяти до полупроводников

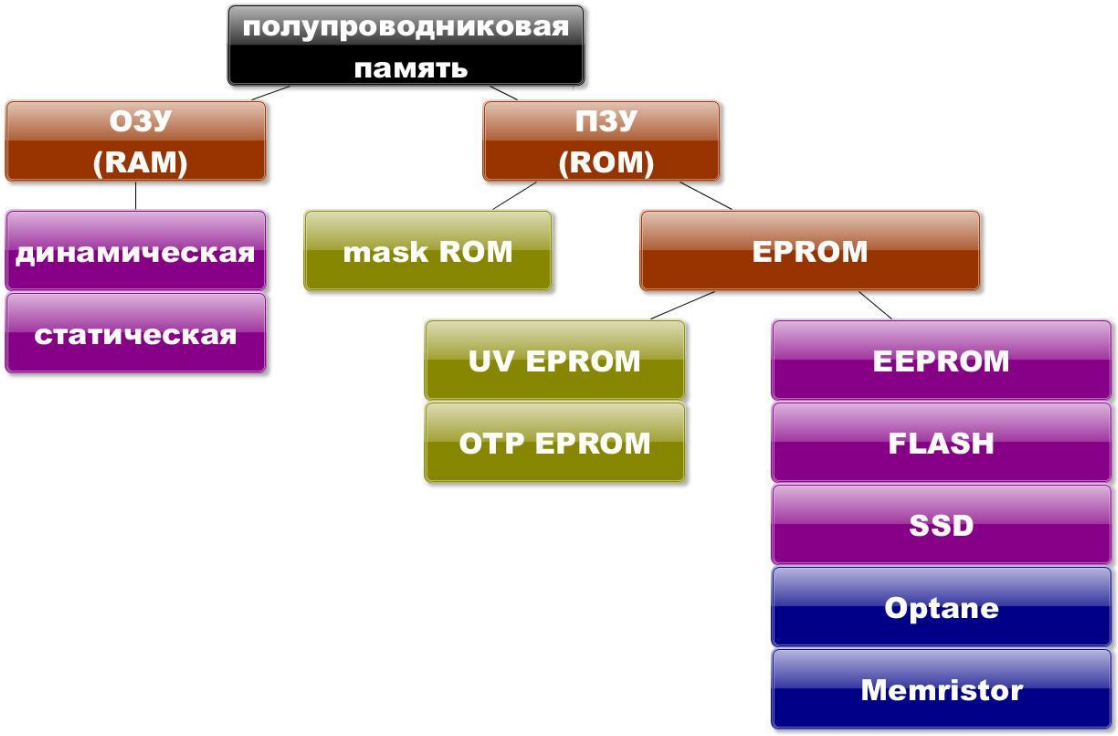

Рис. 2.5.3. Виды полупроводниковой памяти (оперативной и постоянной)

В *Статической памяти* хранение одного бита реализуется на основе триггера (триггер – логическое устройство с двумя устойчивыми состояниями)

на нескольких транзисторах. Она очень быстрая, экономная по питанию, но имеет относительно невысокую плотность (объем памяти на число транзисторов или на кристалл).

Динамическая память использует для хранения бита информации емкость. Она имеет более высокую плотность (и, как следствие, более низкую стоимость при одинаковом объеме) по сравнению со статической памятью. Но у нее есть свои недостатки. Она более медленная, менее экономная по потребляемой энергии. Также для нее требуется регулярно (много раз в секунду) делать перезапись для обновления. Причина этого – в том, что емкости, хранящие заряд, постепенно разряжаются.

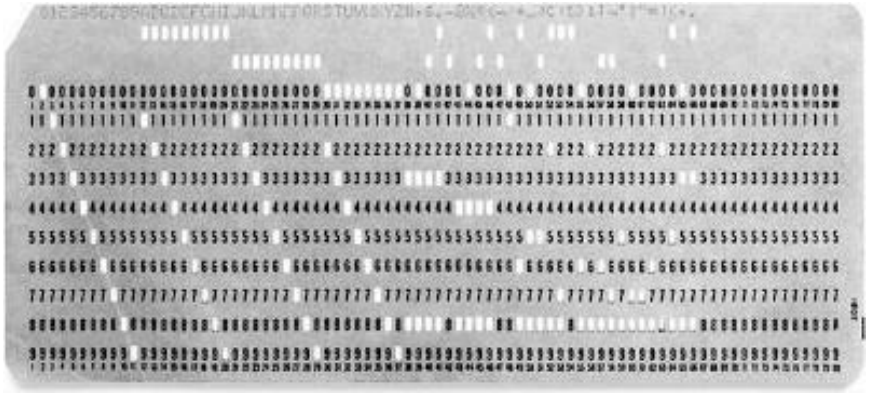

Рис. 2.5.4. Стандартная перфокарта IBM

Рис. 2.5.5. Считыватель с перфокарт IBM 2540

Технологии построения внешней памяти. Первые цифровые компьютеры использовали для хранения данных технологию, возникшую задолго до их появления – *перфокарты*, бумажные карточки, в которых

единичные биты кодировались перфорациями. Идея перфокарты была предложена Басилем Бушоном в 1725 году, и получила свое первое практическое воплощение в 1804 году в программируемом ткацком станке Жакара. Широкое распространение получила с появлением примерно в 1890 году устройства для обработки статистических данных – табуляторе Холлерита. В 1937 году фирмой IBM печаталось 10 млн. перфокарт ежедневно. На типичной перфокарте (см. рис 2.5.4), использовавшейся для цифровых компьютеров, можно сохранить 80 байт. Наиболее совершенное устройство для считывания перфокарт, IBM 2540 (см. рис 2.5.5), производившееся в 1960-е г.г. обеспечивало скорость считывания примерно 1.3 КБ/сек. Объем бункера перфокарт составлял 3100 шт. (около 240 Кбайт). Как объем хранимых на перфокартах данных, там и скорость считывания были сильно ограничены.

Важным шагом по увеличению объема внешней памяти и скорости доступа к ней стало внедрение *стримеров*, накопителей для магнитной ленты в 1951 г. фирмой Univac и в 1952 г. фирмой IBM. Характеристики даже самых первых моделей стримеров давало скорости доступа на порядок выше, чем у перфоленты и перфокарт. Емкость катушки изначально составляла несколько мегабайт. Новым важным качеством была возможность неограниченной перезаписи информации на носитель. Современный серийный стример IBM 3592 позволяет записать на один картридж 15 Тбайт данных при цене картриджа примерно 155 долларов. Экспериментальный накопитель, созданный совместно IBM и Sony, позволяет сохранить на одну ленту 185 Тбайт данных. Накопители на магнитной ленте по-прежнему широко используются для архивного хранения данных.

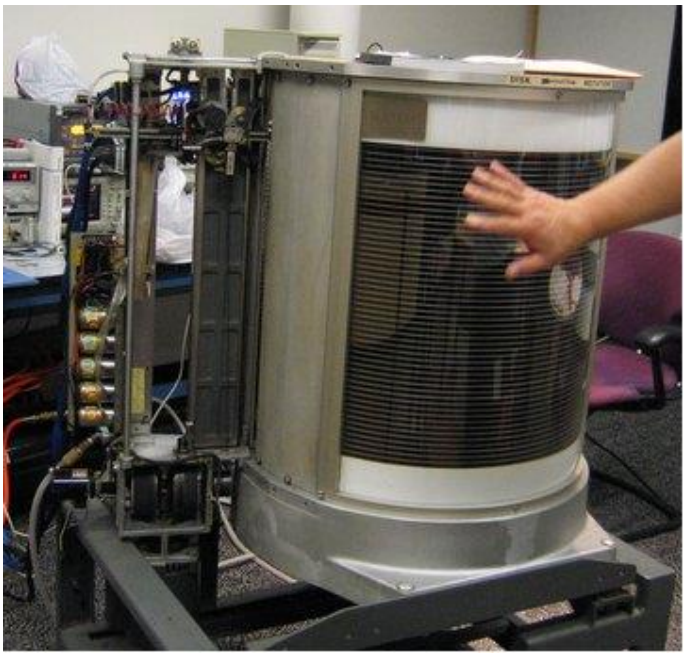

Рис. 2.5.6. Жесткий диск IBM RAMAC

Главный недостаток накопителей на магнитной ленты, низкая скорость произвольного доступа к данным, преодолен в накопителях на *жестких дисках*, внедренных IBM в 1956 г (рис. 2.5.6).

До недавнего времени жесткие диски являлись основным видом внешней памяти компьютеров. Их емкость, плотность записи возросли на несколько порядков, а стоимость одного мегабайта на несколько порядков снизилась. Но один параметр, время доступа, улучшился с 1956 года лишь в разы, и по-прежнему составляет более 1 мсек. Это сильно контрастирует со скоростью работы современного процессора и оперативной памяти. Действительно, за время одной операции чтения с диска процессор может успеть выполнить несколько миллионов простых команд (типа копирования регистров или сложения чисел в регистрах).

Большое время доступа у жестких дисков, а также их ненадежность из-за наличия механически подвижных частей преодолевается в устройствах твердотельной памяти. Основной недостаток современной твердотельной памяти – ограниченное число перезаписей памяти, после которого она утрачивает способность модифицироваться. Новые поколения твердотельной память (например, Intel Optane) постепенно преодолевают эти ограничения.

Возможные дальнейшие направления развития технологий памяти – это мемристоры и микро/наноэлектромеханические структуры (MEMS/NEMS). Память на MEMS имеет достоинства твердотельной памяти, но при этом в ней нет ограничения на число перезаписей. Кроме этого она обладает устойчивостью к излучениям. Это делает ее удобной для космических применений.

Ряд других технологий используют для устойчивого хранения информации самые разнообразные явления из области физики, химии и биологии. В качестве носителя информации пытаются использовать широкий спектр материалов – от бумаги до полимеров и до стекла.

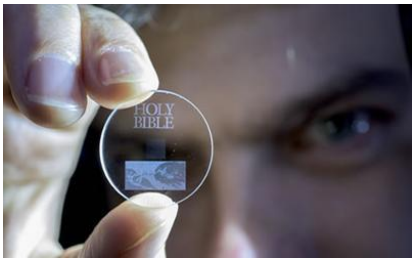

Рис. 2.5.7. Носитель на кварцевом стекле Univ. Southhampton

Технология однократной записи данных на стеклянный носитель с помощью лазера была разработана в университетах Киото и Саусхэмптон. Технология обеспечивает запись 360 Тбайт данных на один носитель (рис. 2.5.7). Время хранения практически не ограничено.

Рассмотренные технологии обладают существенно различающимися характеристиками (на несколько порядков) от крошечных по объему, но очень быстрых до огромных и очень медленных. Задачей разработчиков вычислительных систем было объединение этих технологий в единую иерархию памяти, которая в большинстве ситуаций выглядит как одновременно очень большая и очень быстрая память. Два основных механизма построения этой иерархии – это кэш память и виртуальная память. Кроме них подсистема памяти реализует и другие механизмы, позволяющие повысить скорость обращения к памяти: блочная структура, интерливинг, аппаратная предвыборка, программная предвыборка.

Локальность доступа к данным. Итальянский инженер и экономист Вильфредо Парето обратил внимание на особенность, присущую многим явлениям из самых различных областей. В абстрактном виде ее можно сформулировать так: 80% следствий следует из 20% причин. Анализируя доступ к данным при выполнении программы, можно заметить, что в заданный промежуток времени большая часть обращений происходит к небольшой части данных. По мере исполнения программы набор активно использующихся данных может меняться. Это свойство получило название локальности.

Различают два вида локальности: временную и пространственную. *Временная локальность* проявляется в том, что если произошло обращение к некоторой ячейке памяти, то высока вероятность повторного обращения к ней в ближайшее время. *Пространственная локальность* выражается в том, что, если произошло обращение к некоторой ячейке памяти, высока вероятность обращения к соседним с ней ячейкам.

Обзор современной иерархии памяти. Практические во всех современных архитектурах (за исключением некоторых узко специализированных) используется композиция из нескольких видов памяти с разными характеристиками. Цель, с которой сделано это усложнение, в том, чтобы создать иллюзию быстрой памяти большого объема, используя при

этом быструю память малого объема и большую медленную память. Эта цель достигается не при произвольных условиях, а только когда в исполняемой программе наблюдается локальность (временная и пространственная). Соответственно, степень достижения этой локальности в существенной степени определяет, насколько эффективно программа пользуется памятью. Некоторые подходы к обеспечению этой эффективности рассмотрены в разделе 6 “Оптимизация программного обеспечения разработчиком”.

К настоящему моменту сложилась примерно следующая иерархия уровней памяти (см. рис. 2.5.8):

1) регистровая память, расположена в микропроцессоре, крошечный размер (от нескольких до нескольких сотен ячеек размером с машинное слово), минимальное время доступа (в современных микропроцессорах широкого применения около 1 нс.). Для построения используется статическая полупроводниковая память.
2) Кэш память первого уровня, расположена в микропроцессоре, каждое ядро процессора имеет собственную кэш память для данных и для команд, примерный объем 4-64 КБ, время доступа – в два-три раза медленнее, чем к регистрам. Для построения используется статическая полупроводниковая память.
3) Кэш память второго и третьего уровня, также расположена в микропроцессоре, может быть как собственная у каждого ядра, так и общая для всех ядер, разделение на отдельный кэш для команд и данных не производится, время доступа – медленнее в 2-3 раза, чем у кэш-памяти первого уровня, объем может достигать десятком мегабайт. Для построения используется статическая полупроводниковая память.
4) Оперативная память расположена отдельно от микропроцессора (хотя в последние годы появилось несколько высокопроизводительных архитектур, где небольшая часть оперативной памяти – до 16 Гбайт – может располагаться на самом кристалле микропроцессора). Время доступа к оперативной памяти в современных система составляет примерно 60-120 нс. Это в несколько раз больше времени доступа к кэш-памяти третьего уровня и практически на два порядка больше времени доступа к регистру процессора. Объем оперативной памяти варьируется от очень небольшого во встраиваемых системах до нескольких мегабайт в персональных компьютерах и практически до 1 терабайта всамых производительных вычислительных системах.

5) Внешняя память служит для долговременного хранения данных и программ с возможностью произвольного доступа к любым хранящимся в ней данным. На настоящий момент происходит плавный переход от использования жестких дисков к твердотельной памяти в качестве внешней. Объем внешней памяти варьируется от десятков гигабайт до нескольких терабайт в персональных компьютерах и до многих терабайт в высокопроизводительных системах. Время доступа зависит от используемой технологии. Жесткие диски, в которых время доступа в основном состоит из времени позиционирования считывающей головки на требуемую дорожку и времени вращения диска, скорость доступа на несколько порядков медленнее, чем у оперативной памяти. Время доступа – не менее нескольких миллисекунд. Твердотельные диски существенно быстрее (иногда более чем на порядок), но при текущих возможностях технологии у них есть собственный недостаток – ограничение на число перезаписей.
6) Вторичная внешняя память применяется только в системах, где требуется хранить гигантские объемы данных. Она использует стримеры (накопители на магнитной ленте для хранения данных в цифровой форме). Соответственно, она не обеспечивает произвольного доступа к данным. Чтобы считать нужный блок, приходится отмотать катушку магнитной ленты до нужной позиции – это может занимать десятки секунд.

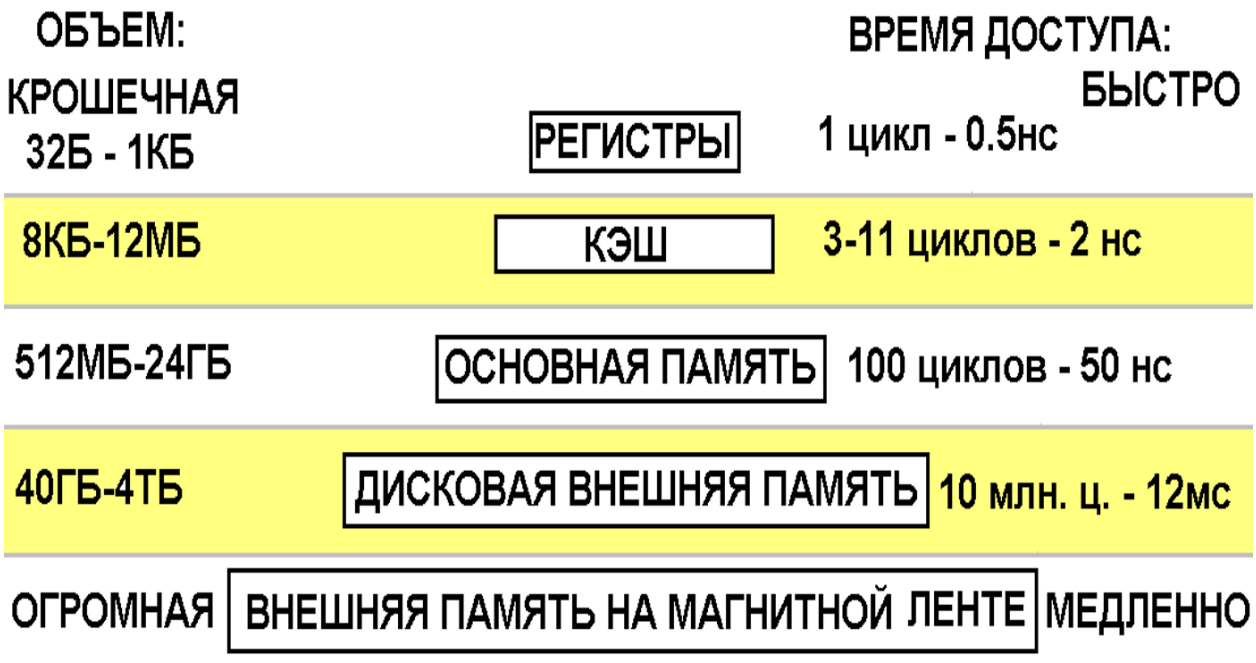

Рис. 2.5.8. Иерархия памяти в современном компьютере

Из приведенной диаграммы видно, что время доступа к регистру внутри процессора и к оперативной памяти отличается по времени на два порядка. Это является проявлением тенденции, которая наблюдалась примерно с 1980 г. Как производительность микропроцессоров, так и скорость работы оперативной памяти росли в геометрической прогрессии. Для производительности процессоров это было названо законом Мура, в честь Гордона Мура из Интел, который первым заметил это явление. Однако, коэффициент прогрессии, и, соответственно, скорости роста отличались примерно на порядок. Примерные графики за некоторую часть периода времени взяты из [Carvalho02] и приведены на рис. 2.5.9.

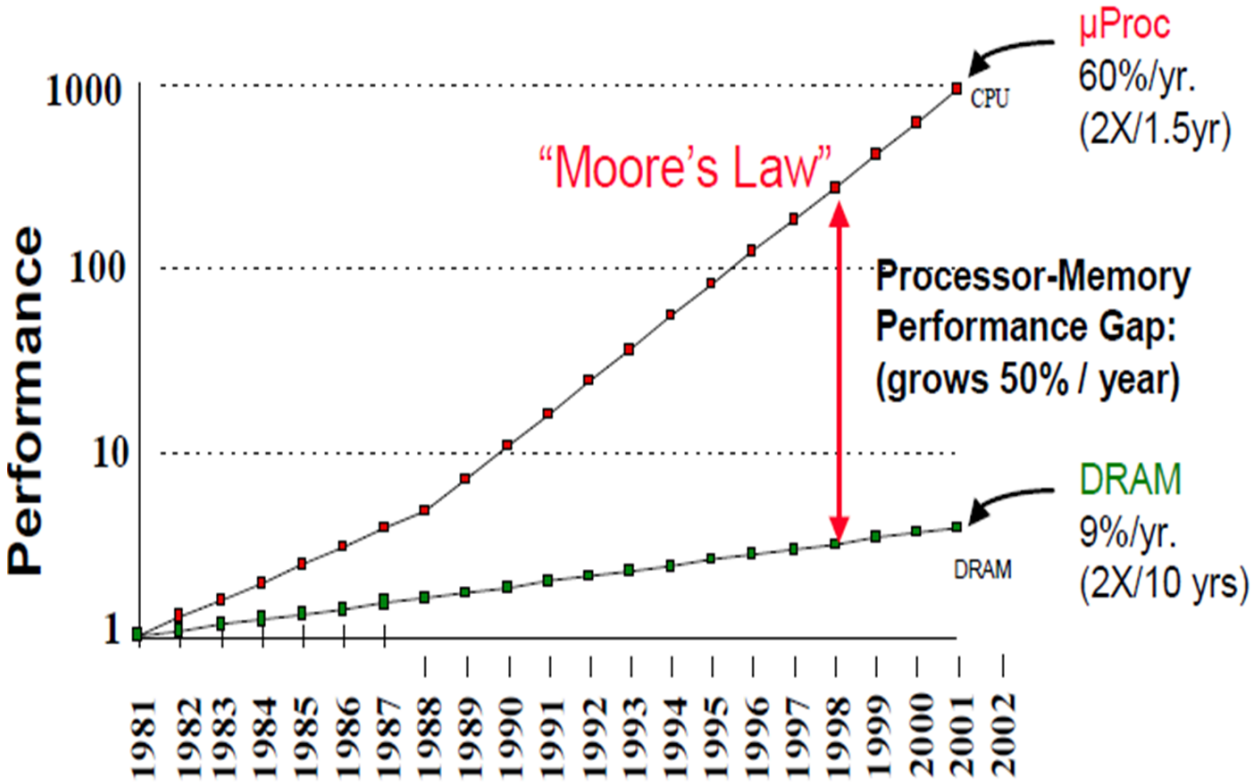

Рис. 2.5.9. Разрыв роста производительности центрального процессора и оперативной памяти

2.6. Кэш память

При разрыве в скорости работы процессора и оперативной памяти примерно в 100 раз улучшения в скорости работы процессора должны были бы нивелироваться. Чтобы разрешить эту проблему, между быстрым процессором (с его быстрыми и немногочисленными регистрами) и медленной оперативной памятью был введен дополнительный уровень памяти – кэш-память. Она существенно меньше по размеру, чем оперативная память, но по

скорости она лишь немного уступает регистровой памяти. Как следует из названия (cache означает тайник), этот уровень прозрачен. Программы его не видят. Он перехватывает все запросы процессора к памяти и пытается обработать их самостоятельно, без обращений к памяти. Это удается, если требуемые процессору данные уже загружены в кэш из оперативной памяти ранее (эта ситуация называется попаданием). Если их там еще нет, то приходится обращаться к оперативной и загружать их в кэш.

Блочная структура оперативной памяти. При промахе при чтении данных из кэша, кэш контроллер перенаправляет запрос дальше, через шину памяти к оперативной памяти. Вне зависимости от того, потребовала ли программа загрузить один байт или одно машинное слово, из оперативной памяти в строку кэш памяти загружается сразу блок данных. Размер блока и строки кэш памяти совпадают. Они фиксированы (их нельзя поменять), но могут различаться для различных архитектур. Размер блока памяти в архитектуре x86 составляет 64 байта. Блочная структура памяти – это проявление того факта, что шина памяти имеет большую разрядность (много машинных слов). Эта большая разрядность вместе с высокой частотой, на которой работает эта шина, обеспечивает высокую пропускную способность. При загрузке по одному слову или байту такая пропускная способность была бы недостижима.

Структура кэш памяти. Кэш состоит из строк. Каждая строка позволяет хранить фрагмент данных из памяти, информацию об адресе этого фрагмента и биты, показывающие, 1) если ли в этой строке какие-либо загруженные данные, и 2) изменялись ли данные после из загрузки в кэш.

Главные характеристики кэша – это: 1) его размер, 2) размер строки, 3) используемый алгоритм отображения и степень ассоциативности, 4) алгоритм вытеснения, 5) алгоритм записи. Алгоритм отображения определяет, в какую строку будет загружен блок памяти с определенным адресом. Алгоритм вытеснения срабатывает в тот момент, когда кэш (или его часть) полностью заполнены, и нет свободной строки для загрузки новых данных из оперативной памяти. В этот момент алгоритм вытеснения определяет, какие данные необходимо выгрузить из кэша, чтобы освободить место для новых. Алгоритм записи определяет, в какой момент данные, которые были изменены, нужно сохранить из кэша обратно в оперативную память.

Алгоритмы отображения. Их сравнительный анализ. Существует три основных вида алгоритмов отображения в кэш памяти. Это – прямой, множественно-ассоциативный и полностью ассоциативный.

При прямом отображении каждый блок памяти может загружаться только в одну строку кэш памяти. При множественно-ассоциативном отображении таких строк несколько (обычно от двух до шестнадцати). При полностью ассоциативном отображении любой блок памяти может загружаться в любую стоку кэш памяти.

Кэш с прямым отображением – наиболее простой в реализации. Любой блок памяти имеет только одну строку кэша, в которую он может быть загружен (см. рис. 2.6.1). Если взять адрес блока (это часть адреса первой ячейки блока без последних битов, которые определяют *смещение* ячеек внутри блока, для x86 этих битов пять), то младшая его часть (*индекс*) равна номеру строки кэш памяти, куда этот блок может быть загружен (см. рис 2.6.2). Старшая часть называется *тэгом*.

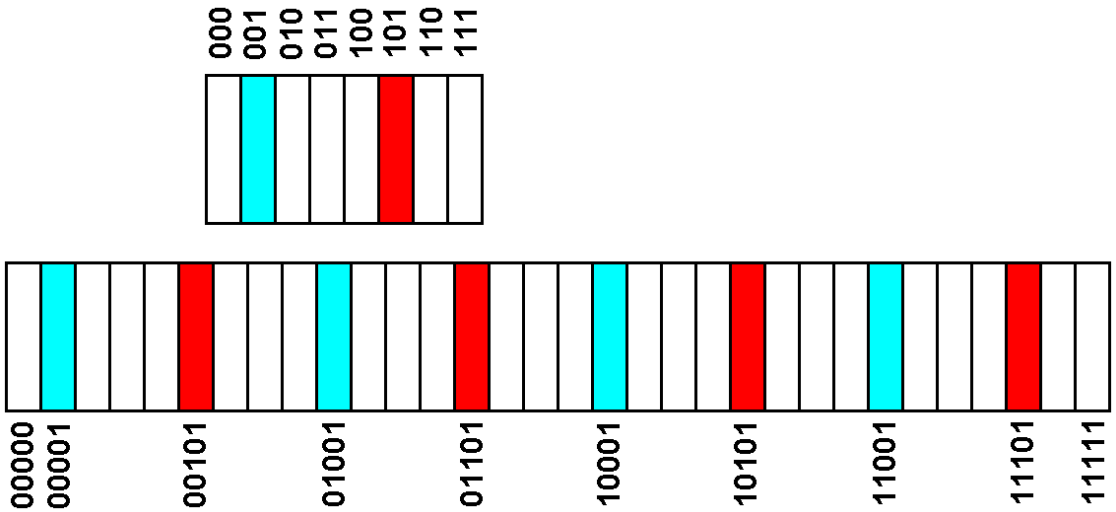

Рис. 2.6.1. Соответствие адресов и строк при прямом отображении

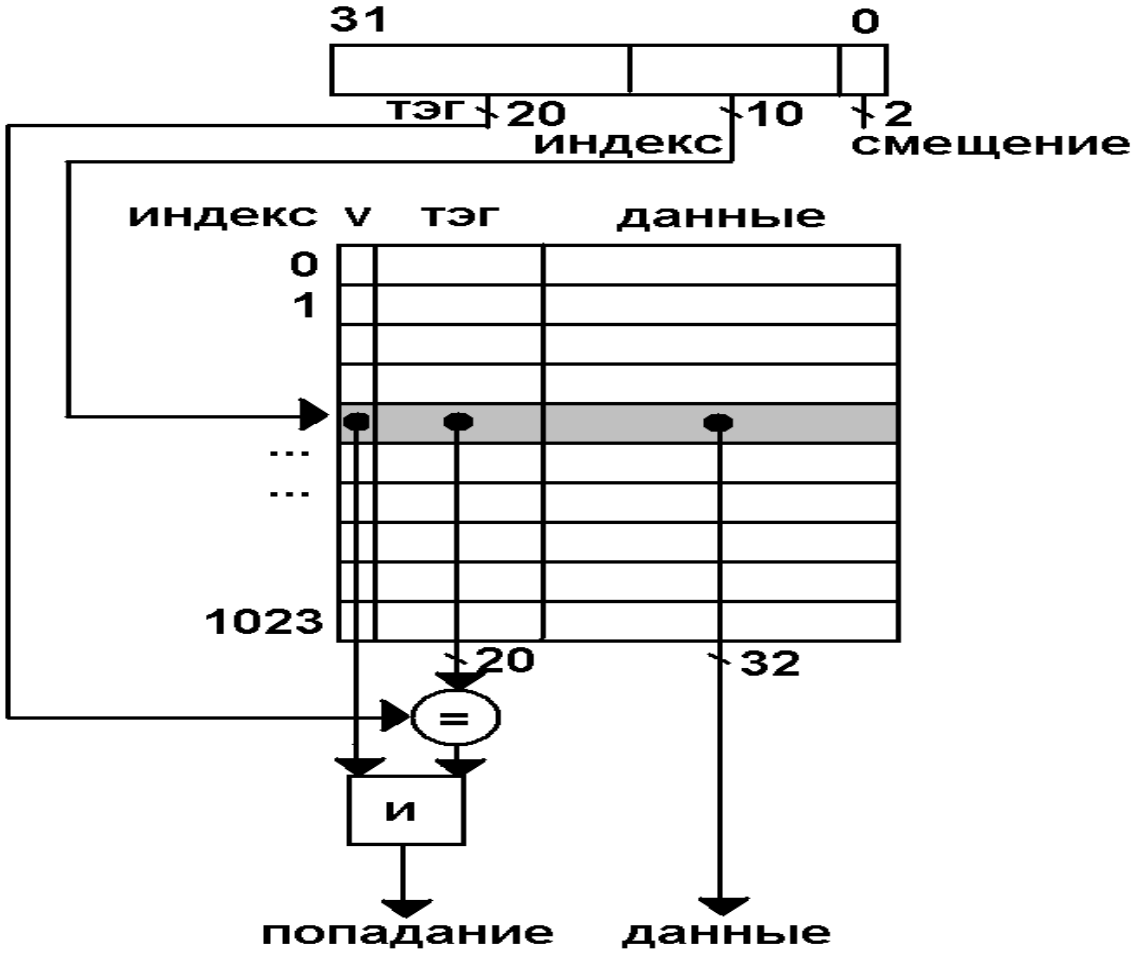

Рис. 2.6.2. Преобразование адресов и доступ при прямом отображении

В приведенном на рис. 2.6.2 примере размер адреса – 32 бита, размер смешения в блоке – 2 бита (размер блока – 4 байта), индекс – 10 бит (число строк кэша – 1024), размер тэга – 20 бит.

Пример 1. Рассмотрим последовательность действий при трех обращениях к памяти. Первые два происходят по адресам:

10101010101010101010 0000000101 01_2 (двоичное представление),

10101010101010101010 0000000101 10_2

а третье – по адресу

10101010101010111111 0000000101 11_2

У них разные смещения внутри блока (**01_2, 10_2** и **11_2**), но на взаимодействие кэша и оперативной памяти это не влияет. У них один и тот же индекс – **0000000101_2**, то есть блоки, содержащие ячейки с этими адресами,

проецируются в одну и ту же строку кэша – $\mathbf{5_{10}=101_2}$. Тэги у них, разумеется отличаются, ведь это адреса ячеек, принадлежащих разным блокам.

При первом обращении из адреса выделяется индекс $\mathbf{0000000101_2=5_{10}}$. Для пятой строчки кэша проверяется, совпадает ли тэг с тэгом в переданном адресе. В этой строке еще нет никаких загруженных данных, значение тэга – 0, а переданный тэг равен $\mathbf{10101010101010101010_2}$. Тэги не совпадают, значит требуемых данных в кэше нет. Соответствующий блок памяти загружается в пятую строку кэша, тэг этой строки устанавливается в $\mathbf{10101010101010101010_2}$. После этого байт в этой строке кэша по смещению $\mathbf{01_2}$ передается в процессор.

При втором обращении индекс не изменился, значит снова требуется пятая строка кэша. Теперь тэг из переданного адреса и тэг в строке кэша совпадают. Это означает, что затребованные данные уже лежат в этой строке кэша. Байт по нужному смещению ($\mathbf{10_2}$) посылается в процессор.

При третьем обращении индекс снова равен 5. Для пятой строки сравниваются тэги. В переданном адресе он равен $\mathbf{10101010101010111111_2}$, а сохраненный в пятой строке - $\mathbf{10101010101010101010_2}$. Так как тэги не совпадают, строка кэша хранит не те данные, которые нужно сейчас загрузить. Эти данные стираются (в память обратно они не сохраняются, так как их не модифицировали). Вместо них загружаются данные нового блока. Тэг пятой строки меняется на $\mathbf{10101010101010111111_2}$. После этого ячейка по смещению $\mathbf{11_2}$ пересылается в процессор.

В реальных реализациях кэша размер блока и строки больше размера машинного слова, и в процессор загружается не вся строка кэша, а только один байт или слово по заданному адресу. Структура такого кэша слегка усложнена по сравнению с рассмотренной выше (см. рис. 2.6.3). Отличие в работе от последовательности, рассмотренной в примере 1, заключается в том, что когда требуемые данные обнаружены, они подаются на мультиплексор, который выбирает из все данных строки только то слово, которое было затребовано процессором. Для этого на мультиплексор подаются биты адреса, относящиеся к смещению внутри блока. Младшие биты смещения не подаются, так как они нужны для выбора отдельного байта внутри слова.

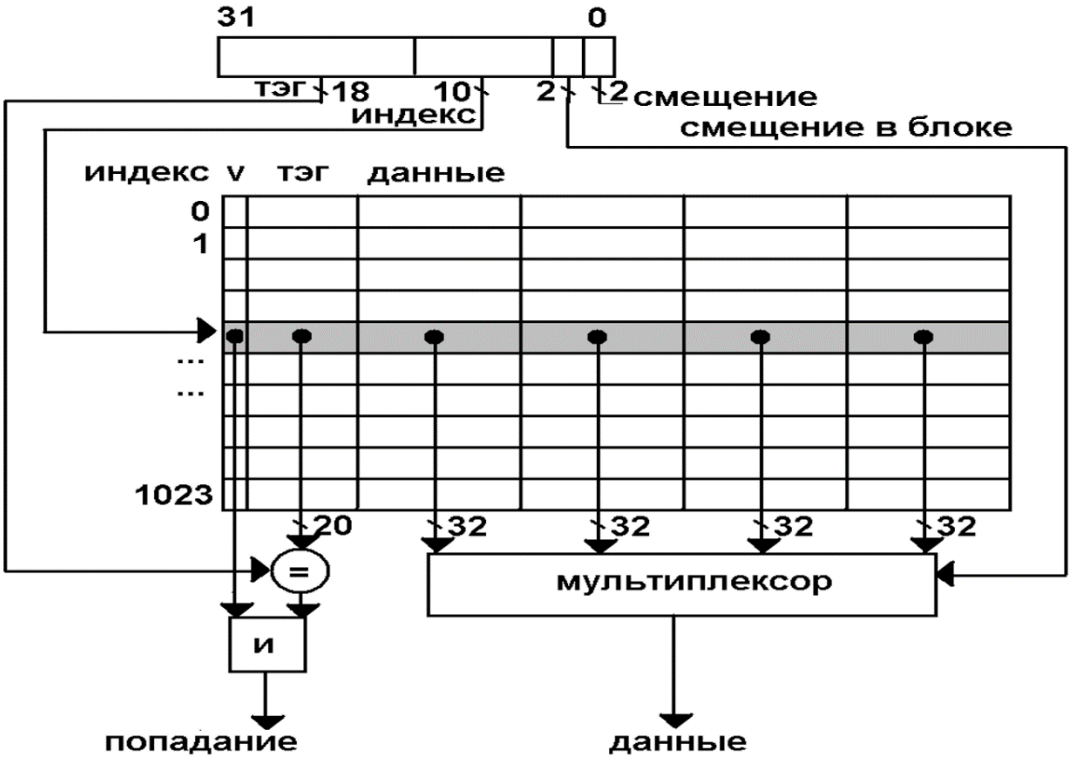

Рис. 2.6.3. Кэш с прямым отображением при размере блока и строки в 16 байт

Из примера 1 видно, что для кэша с прямым отображением могут быть такие последовательности доступа, которые используют только одну строку кэша. Такая ситуация называется буксованием кэша. При этом скорость работы программы сильно (до одного порядка!) замедляется. Обычно она возникает при обходе массивов с расстоянием (в адресном пространстве) между последовательно считываемыми элементами равным размеру кэш памяти. Ее более подробное описание и способы устранения рассматриваются в гл. 6 этого конспекта лекций.

Множественно-ассоциативный кэш в значительной степени преодолевает недостатки кэша с прямым отображением. В нем гораздо реже возникает буксование. В множественно-ассоциативном кэше любой блок памяти может проецироваться не в одну сроку кэша, а в некоторое множество блоков одной строки (см. рис. 2.6.4). Мощность этого множества называется степенью ассоциативности. Для разных реализаций кэша степень ассоциативности варьируется от двух до шестнадцати. Можно рассматривать множественно-ассоциативный кэш как набор из нескольких кэшей прямого отображения, их называют банками. Разумеется, их количество совпадает со степенью ассоциативности.

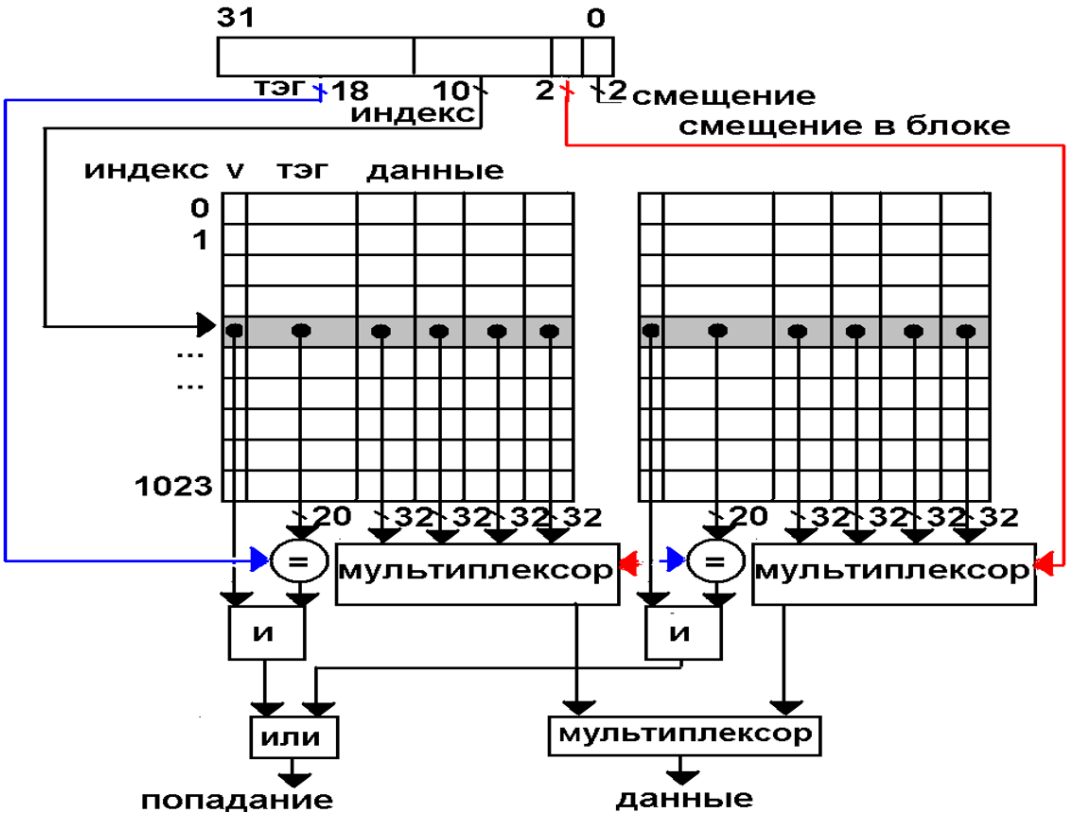

Рис. 2.6.4. Кэш со степенью ассоциативности 2

Для кэша с прямым отображением алгоритм определения, лежит ли в данной строке требуемый комплект данных был тривиален. Нужно было сравнить тэг запрашиваемого адреса и тэг, сохраненный в строке с выбранным индексом (см. пример 1 выше). Для множественно-ассоциативного кэша, где есть несколько блоков в одной строке, тэг запрашиваемого адреса нужно сравнивать с тэгом каждого блока. Попадание определяется пирамидой схем “или”. Для приведенного на рис. 2.6.4 примера со степенью ассоциативности 2 этот каскад состоит из одной схемы “или”.

Алгоритм размещения во множественно-ассоциативном кэше усложняется по сравнению с кэшем прямого отображения, где в строке был только один блок и рассматривалось только две ситуации: 1) строка свободна, можно загружать данные, 2) строка занята другими данными, надо их аннулировать или выгрузить в память, а потом загрузить новые. В множественно-ассоциативном кэше в случае наличия свободных блоков в строке можно выбрать любой из них для использования. Если же все блоки заняты, то должен выполниться алгоритм вытеснения, который выберет блок строки, из которого нужно вытеснить старые данные. Алгоритмы вытеснения рассматриваются ниже.

Полностью ассоциативный кэш позволяет загрузить любой блок памяти в любой блок кэша. В данном случае можно считать, что в кэше всего

одна строка из множества блоков. В адресе больше нет группы битов индекса строки. Старшие биты по-прежнему образуют тэг, а младшие – смещение внутри блока.

Для определения, загружен ли в ассоциативный кэш требуемый блок, нужно сравнивать тэг из адреса с тэгами всех блоков кэша. При большом размере кэша это существенно замедляет скорость проверки. Аналогично, усложняется работа алгоритма вытеснения, который выбирает блок для вытеснения не из нескольких (до 16), а из большого числа.

Построить большой и одновременно быстрый полностью ассоциативный кэш невозможно. На практике удобнее использовать множественно-ассоциативный кэш. Полностью ассоциативный кэш практически никогда не применяется для построения кэш памяти. Однако, он, например, используется для реализации TLB для ускорения трансляции виртуальных адресов в физические (раздел по виртуальной памяти см. ниже).

Алгоритмы записи определяют в какой момент данные из кэша, которые были модифицированы, будут сохранены обратно в оперативную память (или кэш следующего уровня). Основные используемые алгоритмы:

1) Сквозная запись (write-through) – сохранение происходит сразу после изменения данных в кэше. Наиболее просто реализуется, но может генерировать большой поток данных при постоянной перезаписи строк без их вытеснения.

2) Write-around – запись результатов сразу в память, минуя кэш. Для случаев, когда записываемые данные не будут с ближайшее время снова считываться, этот алгоритм позволяет сократить число запросов к кэшу. Однако, при чтении данных после их записи происходит промах, и они снова загружаются из оперативной памяти в кэш.

3) Обратная запись (write-back) – сохранение происходит перед тем, как нужно вытеснить данные из кэша (конечно, при условии, что они были изменены). Сложнее реализуется, но более экономно использует шину памяти. Некоторым недостатком является увеличение времени загрузки данных в кэш при вытеснении старых данных.

4) Комбинированная запись (write-combine) – использует буферизацию запросов на запись. Сначала готовится пакет для записи, а потом он целиком сохраняется в оперативную память (пакетный режим – burst mode).

В современных системах наиболее распространен алгоритм обратной записи.

Алгоритмы вытеснения выбирают блок (из набора) должен быть выгружен, когда в кэш нужно загрузить новые данные, а свободных строк или блоков не осталось. В качестве наиболее распространенных алгоритмов вытеснения можно привести следующие:

1) RR. Случайное замещение (random replacement) – блок выбирается случайно. Простая реализация, но низкая эффективность.

2) LRU. Наиболее давно использовавшийся (least recently used) – вытесняется блок, который больше всего не использовался. Сложная и более медленная реализация, чем у других алгоритмов при степени ассоциативности выше 4.

3) Pseudo-LRU. Псевдо-LRU – более простой и быстрый, чем LRU. В большинстве случаев правильно определяет блок, который дольше всего не использовался.

Кэш память в современных архитектурах как правило представлена двумя или тремя уровнями кэш памяти – от самой быстрой и маленькой L3 до большой и медленной L3. Кэш L1 имеет размер в несколько десятков килобайт. Обычно он делится на кэш команд и кэш данных, так как разделение потоков обращений к памяти для загрузки кода и данных повышает локальность каждого из них. Кэш L2 имеет размер от нескольких сотен килобайт до нескольких мегабайт. Кэш L3 хранит от нескольких мегабайт до нескольких десятков мегабайт. Поскольку большинство современных микропроцессоров – многоядерные, для каждого из уровней кэша разработчики выбирают один из двух вариантов реализации: 1) каждое ядро имеет свой локальный кэш, 2) все ядра пользуются общим кэшем. Второй вариант позволяет сформировать кэш большего объема, но при этом растет время обращения к нему. Почти всегда кэши данных и команд L1 выделяются для каждого ядра отдельно, а кэши L3 чаше бывают общими для всех ядер. Пример с организацией кэш памяти в процессоре IBM Power7 приведен на рис. 2.6.5. Основные параметры кэш памяти для некоторых современных процессоров приведены на рис. 2.6.6.

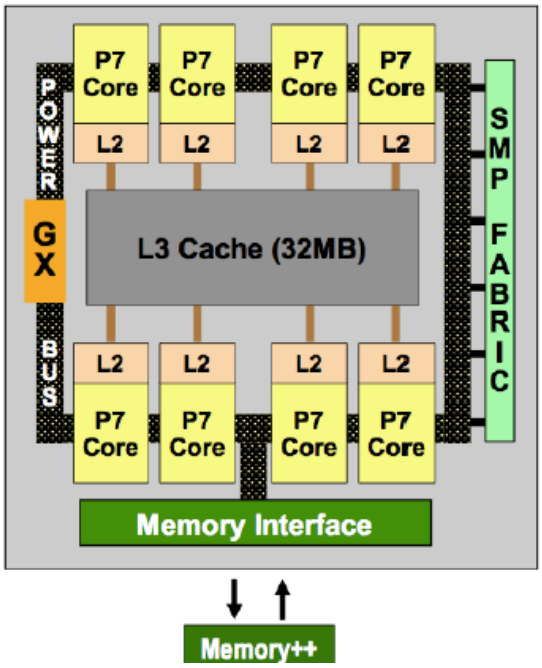

Рис. 2.6.5. Кэш в IBM Power7

Процессор	Кэш-память 1 уровня		Кэш-память 2 уровня		Кэш-память 3 уровня
	Данных	Команд	Данных	Команд	
Intel Atom N570, 2 ядра	24 КБ, ст.а.6, 64 Б	32 КБ, ст.а.8, 64 Б	512 КБ, ст.а.8, 64 Б		
Intel Xeon E7-8870, 10 ядер	16 КБ, ст.а.8, 64 Б	32 КБ, ст.а.4, 64 Б	256 КБ, ст.а.8, 64 Б		30 МБ, ст.а.24, 64 Б, общий для всех ядер
Intel Xeon E5-2687W, 8 ядер	32 КБ, ст.а.8, 64 Б	32 КБ, ст.а.8, 64 Б	256 КБ, ст.а.8, 64 Б		20 МБ, ст.а.20, 64 Б, общий для всех ядер
AMD Opteron 6386 SE, 16 ядер	16 КБ, ст.а.4, 64 Б	64 КБ, ст.а.2, 64 Б, общий для 2-х ядер	2 МБ, ст.а.16, 64 Б, общий для 2-х ядер		8 МБ, ст.а.64, 64 Б, общий для 8-ми ядер
Intel Itanium 9560, 8 ядер	16 КБ, ст.а.4, 64 Б	16 КБ, ст.а.4, 64 Б	256 КБ, ст.а.8, 128 Б	512 КБ, ст.а.8, 128 Б	32 МБ, ст.а.32, 128В, общий для всех ядер
IBM Power7+, 8 ядер	32 КБ, ст.а.8, 128 Б	32 КБ, ст.а.4, 128 Б	256 КБ, ст.а.8, 128 Б		80 МБ, ст.а.8, 128 Б, общий для всех ядер
Fujitsu SPARC64 VIIIfx, 8 ядер	32 КБ, ст.а.2, 128 Б	32 КБ, ст.а.2, 128 Б	6 МБ, ст.а.12, 128 Б, общий для всех ядер		
Intel Xeon Phi 5110P, 60 ядер	32 КБ, ст.а.8, 64 Б	32 КБ, ст.а.8, 64 Б	256 КБ, ст.а.8, 64 Б		

Рис. 2.6.6. Параметры кэша у современных микропроцессоров

2.7. Предвыборка данных.

Значительная доля программ (вычислительных, мультимедиа, обработка сигналов и т.д.) производит последовательную обработку элементов массивов. Разработчики многих современных микропроцессоров аппаратно реализуют возможность определение такой ситуации, анализируя последовательности адресов, по которым процессор производит доступ к оперативной памяти. Когда последовательное обращение обнаружено, запускается загрузка следующего блока памяти. Так как обработка ячеек массива, хранящихся в уже загруженном в кэш блоке, занимает некоторое время, счет и загрузка происходят параллельно. Когда ячейки блока обработаны, и программа запрашивает ячейку следующего блока, этот блок или уже загружен (тогда чтение ячейки будет выполнено со скоростью кэша) или в какой-то стадии загрузки (чтение ячейки произойдет и в этом случае быстрее, чем чтение из памяти без предвыборки). Сравнение загрузки ячеек при их последовательной обработке без аппаратной предвыборки и с ней приведено на рис. 2.7.1.

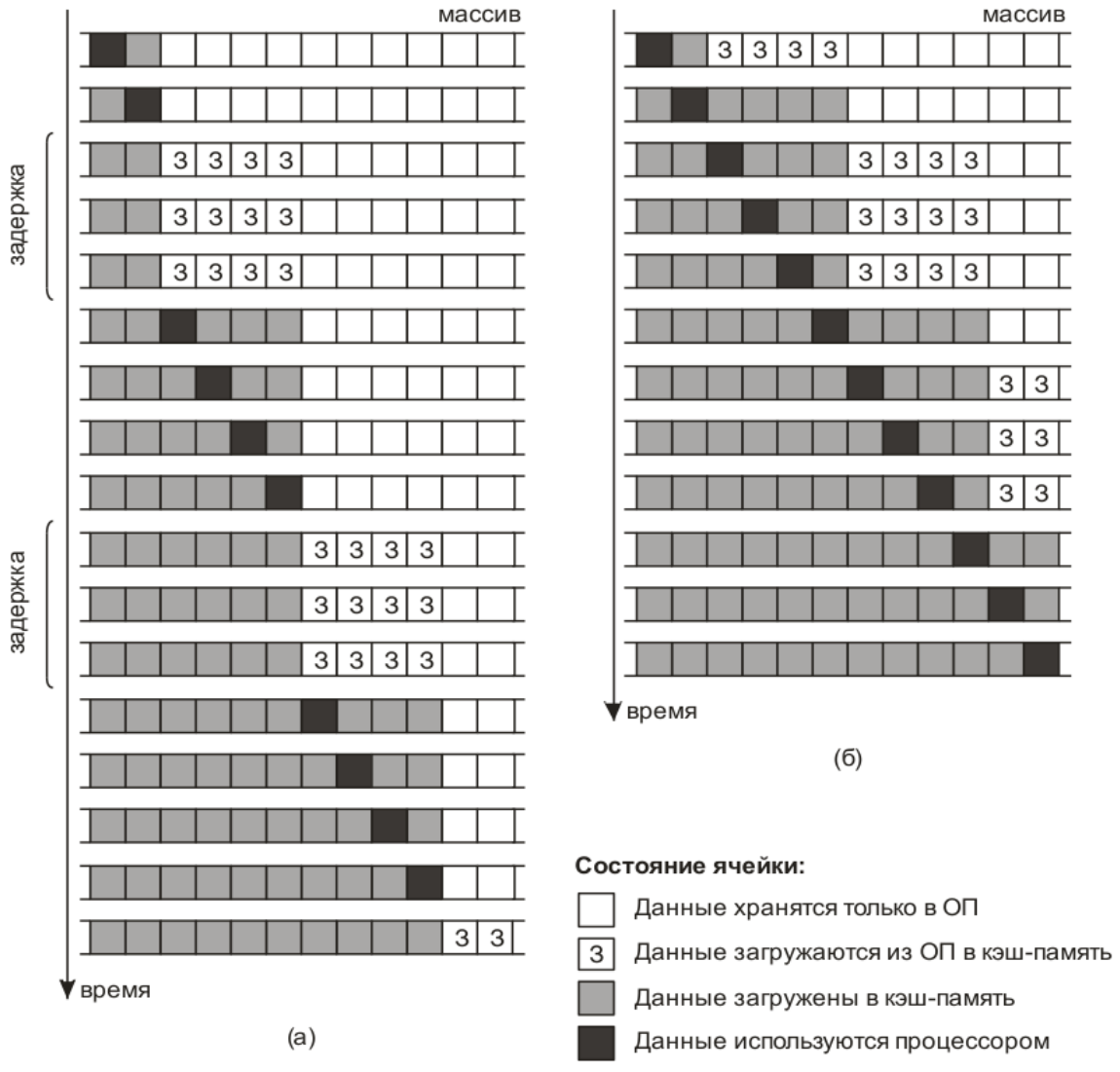

Рис. 2.7.1. Загрузка при последовательной обработке ячеек (а) без аппаратной предвыборки, (б) с аппаратной предвыборкой

Аппаратная предвыборка работает спекулятивно. Загружаются данные по результатам анализа последовательности адресов для чтения данных, без знания логики работы программы. Иногда это приводит к загрузке данных, которые не будут использоваться. Например, может загрузиться блок памяти, следующий непосредственно за блоком, в котором находятся последние ячейки последовательного обрабатываемого массива. При неблагоприятном стечении обстоятельств это может привести к тому, что из кэша будут вытеснены нужные данные. Но доля таких ошибок мала, и общий эффект от аппаратной предвыборки – положительный.

В программах часто происходит последовательный обход нескольких массивов одновременно. Блок аппаратной предвыборки современных микропроцессоров может распознавать и обрабатывать несколько обход одновременно. В некоторых архитектурах распознаются и о последовательные обходы в сторону уменьшения адресов (т.е., например, можно обрабатывать массив, идя от последнего элемента к первому, и пользоваться предвыборкой). Количество одновременно обрабатываемых блоком аппаратной предвыборки обходов называется количеством потоков предвыборки и является одной из его основных характеристик.

2.8. Виртуальная память

Практически начиная с самого начала применения ЭВМ, у программистов возникает желание разместить в памяти большие объемы данных и программ. В первых ЭВМ для того чтобы можно было выполнять программы, код которых был больше оперативной памяти (или той ее части, которая оставалась после того, как выделена память для данных), стали использовать так называемые оверлеи. Этот был программный механизм, при котором программа делилась на резидентную часть (всегда в оперативной памяти, пока программа исполняется) и множество подгружаемых модулей – оверлеев. Если требовалось загрузить другой оверлей, приходилось выгружать один из загруженных ранее, и его функции становились временно недоступными (до его следующей загрузки). По мере увеличения сложности программ, такое ручное управление загрузкой частей программ становилось все более трудоемким по реализации и менее эффективным. В определенный момент исследователи пришли к выводу, что автоматический механизм будет

более эффективным. После этого произошел переход к такому механизму, виртуальной памяти. Сначала это было реализовано для крупных машин в 1960-е годы (например, IBM System 360/System 370), а через два десятилетия – для персональных компьютеров. Первый механизм виртуальной памяти в архитектуре x86 появился в 80286 процессоре в 1982 году. В нем была реализована сегментная виртуальная память. Современная сегментно-страничная виртуальная память (используется обычно как страничная) появилась в 80386 в 1985 году.

Виртуальная память – это механизм управления иерархической памятью компьютера, который позволяет: 1) размещать в памяти и одновременно выполнять несколько процессов (запущенных программ), изолируя их друг от друга, 2) создавать иллюзию у каждого из процессов, что объем доступной ему оперативной памяти практически неограничен.

Изоляция процессов достигается за счет создания виртуального адресного пространства для каждого из процессов. Адресное пространство каждого из процессов доступно только ему и операционной системе. Другие процессы не имеют доступа к нему, исключая случай, когда они используют средства коммуникации между процессами (IPC: общие окна в памяти, очереди сообщений и разнообразные средства синхронизации).

Иллюзия неограниченности оперативной памяти процесса достигается за счет: 1) большого размера адресного пространства процесса (адрес в этом пространстве, виртуальный адрес, имеет больший размер, чем физический адрес) и 2) расширения оперативной памяти за счет использования внешней памяти. Для программы это расширение незаметно, она видит одноуровневую адресуемую память, объем которой равен всему адресному пространству независимо от объема физической памяти компьютера и объема памяти, необходимой для других программ, участвующих в мультипрограммной обработке. Те данные, которые программа использует, находятся в оперативной памяти. Те же, которые давно не использовались, при нехватке оперативной памяти могут перемещаться во внешнюю.

Физически виртуальная память представляет собой совокупность всех ячеек памяти – оперативной и внешней (наличие внешней памяти обязательно). Она имеет сквозную нумерацию от нуля до предельного значения адреса. Адреса виртуального пространства называются виртуальными, адреса физического пространства (оперативной памяти) называются физическими.

Основные способы организации виртуальной памяти – сегментный, страничный и сегментно-страничный. При сегментной организации виртуальное адресное пространство состоит из набора блоков переменного размера, называемых сегментами.

В настоящее время наиболее распространенным способом организации является страничный способ. Страничная виртуальная память делится на блоки фиксированного размера – виртуальные страницы (*page*). Физическая память также делится на блоки фиксированного размера – физические страницы (*page frame*). Размеры виртуальных и физических страниц совпадают. Физические страницы используются для хранения виртуальных страниц. В некоторых современных реализациях виртуальной памяти допускается использование страниц нескольких различных (но не произвольных, как в сегментной виртуальной памяти) размеров (несколько килобайт, несколько мегабайт и примерно один гигабайт).

Сегментно-страничная виртуальная память представляет собой гибрид сегментной и страничной виртуальных памятей. В виртуальном адресном пространстве на верхнем уровне есть множество сегментов. Каждый из сегментов состоит из страниц.

Адрес виртуальной (физической) страницы состоит из номера страницы и смещения (адреса относительно начала страницы). Страницы не имеют прямой связи с логической структурой данных и программ. Страничная организация памяти представляет память как набор страниц равного размера. В любой момент только часть страниц виртуальной памяти присутствует в оперативной памяти, т.е. та часть, которая необходима активным задачам (рис. 2.8.1). Страничная организация сокращает объем пересылок данных между оперативной памятью и внешней памятью, поскольку:

- загрузка и выгрузка страниц в оперативную память выполняются по мере необходимости (загрузка по требованию),
- отсутствует фрагментация, так как страница имеет фиксированный размер.

Кроме того, страничная организация памяти позволяет увеличить число одновременно выполняемых программ.

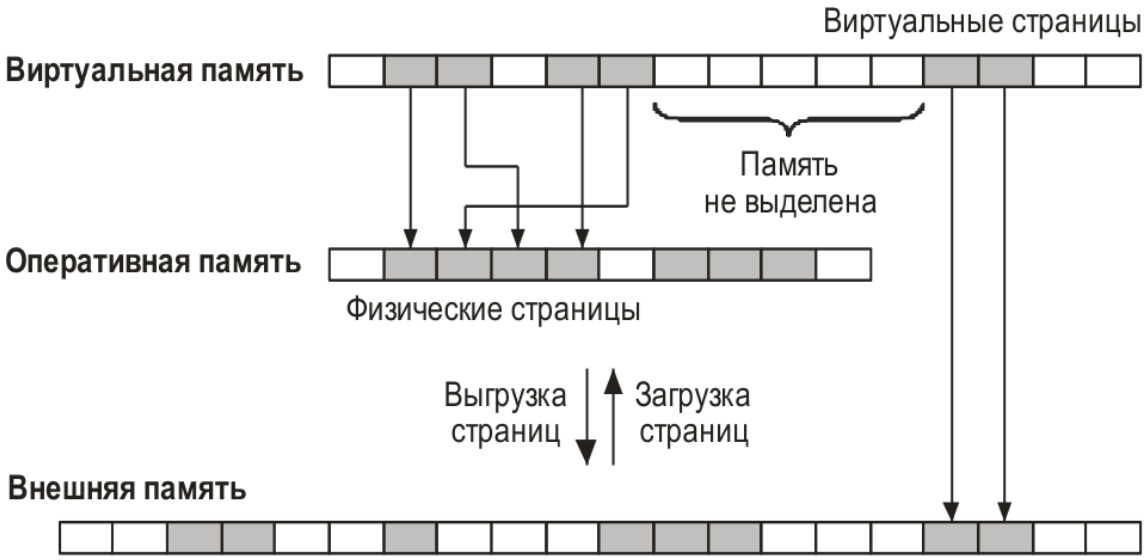

Рис. 2.8.1. Организация страничной виртуальной памяти

При использовании виртуальной памяти для каждой запущенной программы ОС создает собственное виртуальное адресное пространство (Рис. 2.8.2). Виртуальное адресное пространство описывается двумя таблицами: таблицей страниц и картой диска. Таблица страниц устанавливает соответствие виртуальных и физических адресов страниц. Карта диска содержит информацию о расположении страниц во внешней памяти. Процесс доступа к данным по их виртуальным адресам выполняется следующим образом.

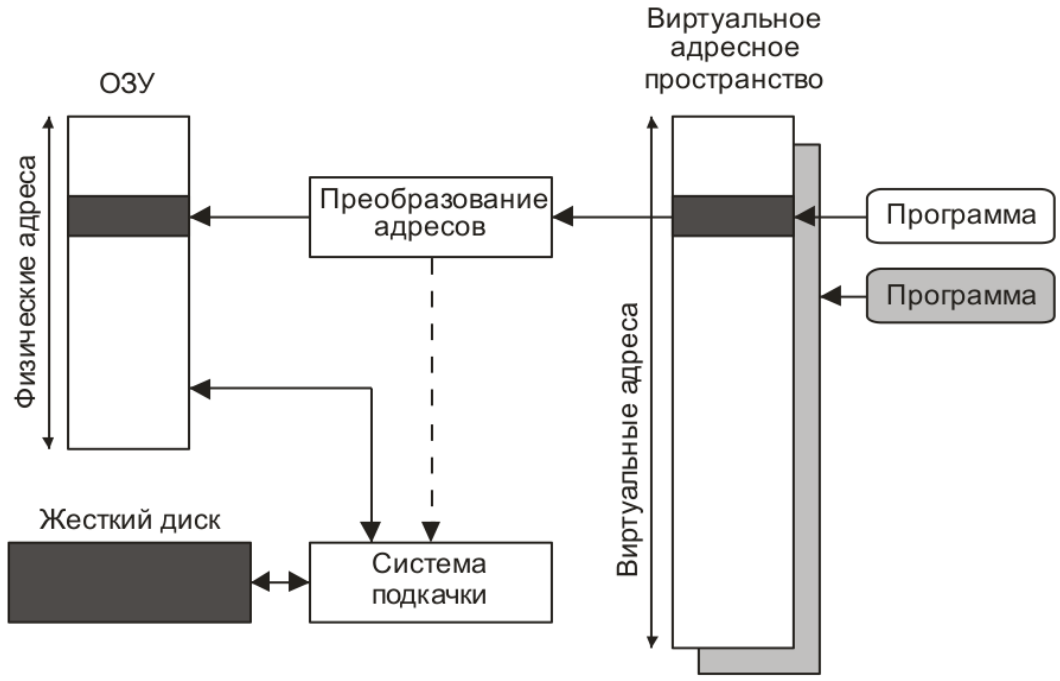

Рис. 2.8.2. Отображение виртуального адресного пространства на физическое

Процессор выставляет на шину адреса виртуальный адрес. Виртуальный адрес представлен парой чисел (p,s), где p – номер виртуальной страницы, а s – смещение внутри страницы (Рис. 2.8.3). Физический адрес аналогично представлен парой чисел (n,s), где n – номер физической страницы, а s – смещение внутри страницы. Виртуальный адрес преобразуется в физический адрес устройством управления памяти. Это устройство может находиться на микросхеме процессора или на отдельной микросхеме рядом с процессором. Преобразование осуществляется с помощью таблицы страниц по следующей схеме.

1) Если физическая страница находится в оперативной памяти, то в таблице страниц считывается строка p. Она содержит номер физической страницы n, по которому однозначно определяется физический адрес этой страницы. Искомый физический адрес вычисляется суммированием физического адреса физической страницы n и смещения s. Кроме физического адреса страницы, в строке таблицы может храниться информация о том, выделена ли оперативная память для данной страницы, происходила ли запись в страницу после ее подгрузки, разрешено ли чтение или запись в эту страницу.
2) Если страница расположена во внешней памяти, то ее нужно подгрузить в свободную страницу оперативной памяти. Если свободной страницы нет, то по любому алгоритму вытеснения выбирается и освобождается одна из занятых страниц. Данные из этой страницы предварительно выгружаются во внешнюю память.

В современных процессорах для ускорения доступа к таблице преобразования адресов используются:

- многоуровневые таблицы страниц,
- буферы быстрого преобразования адресов (TLB).

TLB представляет собой полностью ассоциативную кэш-память или множественно-ассоциативную кэш-память с высокой степенью ассоциативности и временем доступа, сравнимым с кэш-памятью 1-го уровня. В ее памяти тэгов хранятся номера виртуальных страниц, а в памяти данных – номера физических страниц для нескольких последних операций трансляции адресов. Каждая строка таблицы содержит несколько признаков (достоверности, модификации, права доступа и т.д.) (см. рис. 2.8.4).

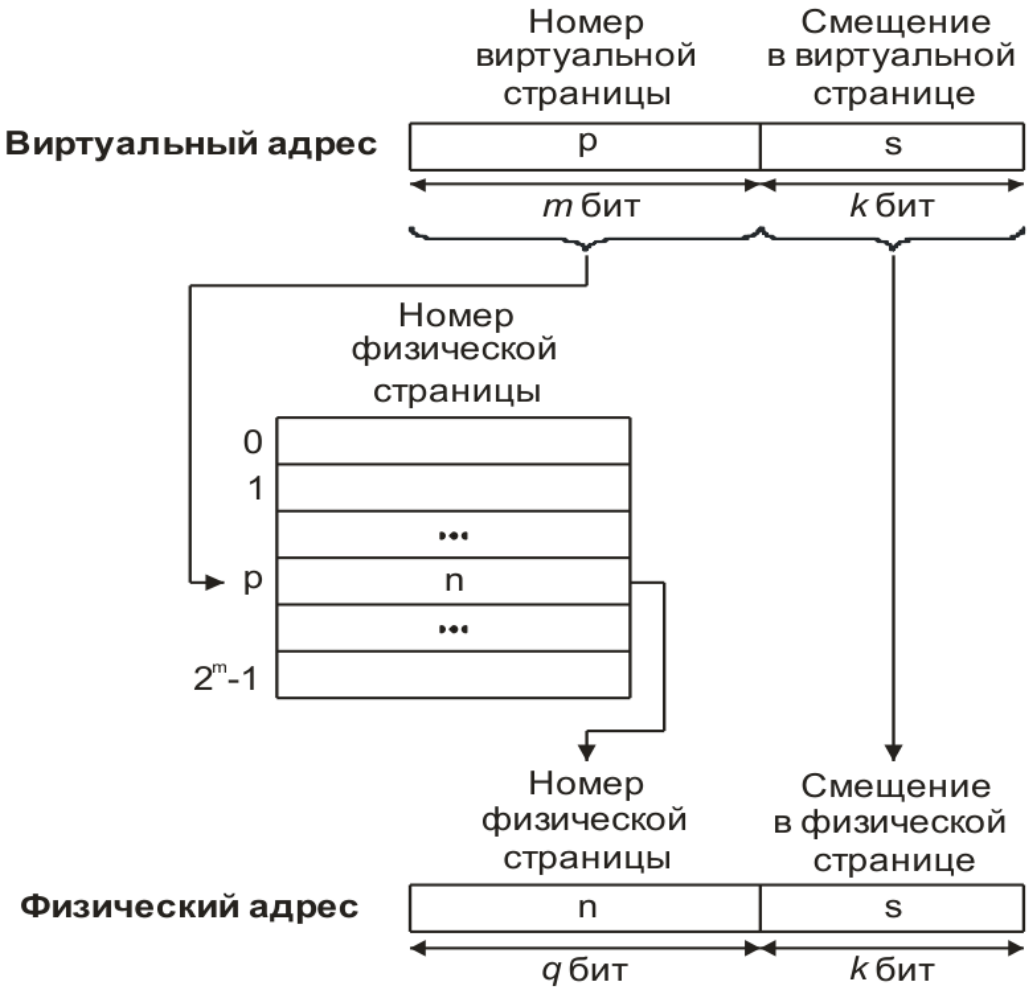

Рис. 2.8.3. Преобразование адресов в страничной виртуальной памяти

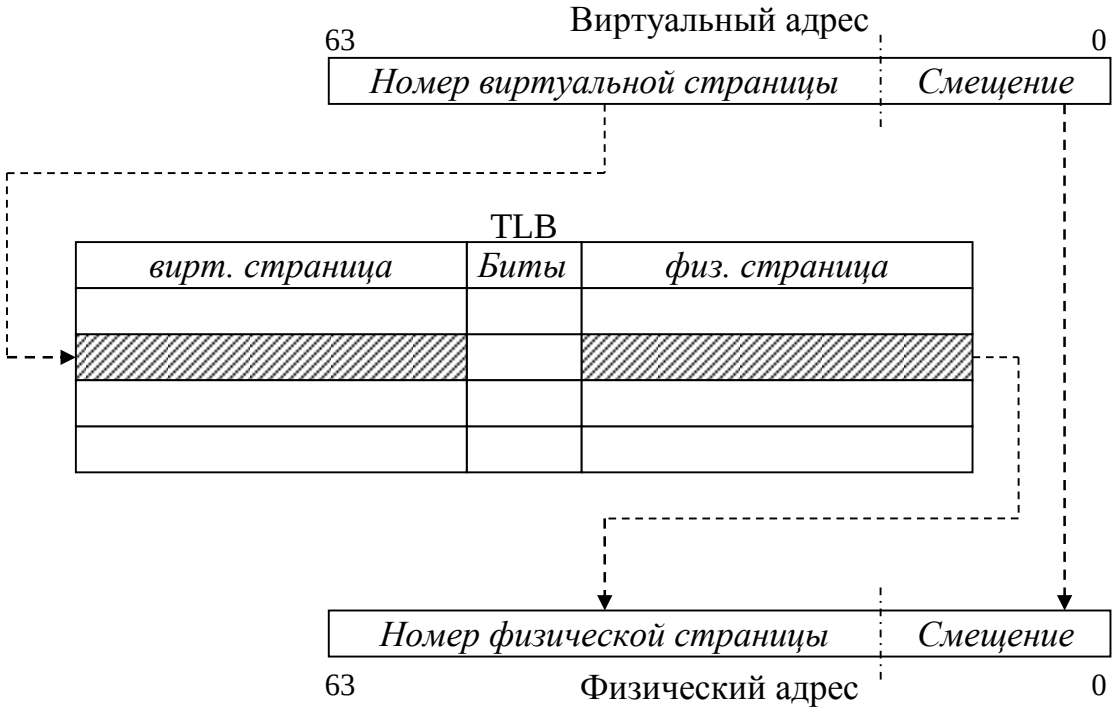

Рис. 2.8.4. Преобразование адресов через TLB

Виртуальная память и кэш-память имеют много сходств. Во-первых, главная функция, которую они реализуют, заключается в построении гибридной памяти, состоящей из нескольких уровней в иерархии памяти, которая кажется такой же быстрой, как память верхнего уровня и такой же большой, как память нижнего уровня. Как и в случае с кэш-памятью, эффективность использования виртуальной памяти в наибольшей степени определяется соблюдением свойства временной и пространственной локальностей доступа к данным. В случае ее эффективного использования она имеет быстродействие почти как у оперативной памяти. В противном случае ее быстродействие замедляется до уровня внешней памяти.

2.9. Перспективы развития подсистемы памяти.

Современная иерархическая организация памяти обладает одновременно высокой скоростью и большим объемом. Но фундаментальные ограничения, заложенные в архитектуру фон Неймана, в полной мере не преодолены (даже с введением параллелизма на различных уровнях). По-прежнему, узким местом остаются шины, по которым ядра процессоров обмениваются данными с памятью. Возможные дальнейшие пути по частичному или полному устранению этой проблемы: 1) переход к архитектурам с мелкозернистым параллелизмом, 2) более широкое применение памяти с встроенными устройствами обработки данным (PIM – processing in memory) и ассоциативной памяти.

Среди прочих перспективных направлений можно перечислить: 1) построение систем с распределенной памятью (существующий прототип для этого – современные параллельные компьютеры с архитектурой NUMA), 2) построение систем с переменной длинной адреса.

2.10. Параллелизм в современных архитектурах.

Введение параллелизма на настоящий момент является основным способом повышения производительности компьютеров. В современных компьютерах параллелизм присутствует в самых разнообразных видах. Различают два основных вида параллелизма: крупноблочный и мелкозернистый (см. рис. 2.10.1). Крупноблочный характеризуется малым числом параллельно работающих вычислителей, каждый из которых имеет

сложную организацию. Объемы обработки данных внутри вычислителей существенно превышают объемы обмена данными между ними. Мелкозернистый параллелизм характеризуется большим числом простых параллельно работающих вычислителей. Объемы передачи данных между ними сопоставимы с объемами их обработки внутри вычислителей.

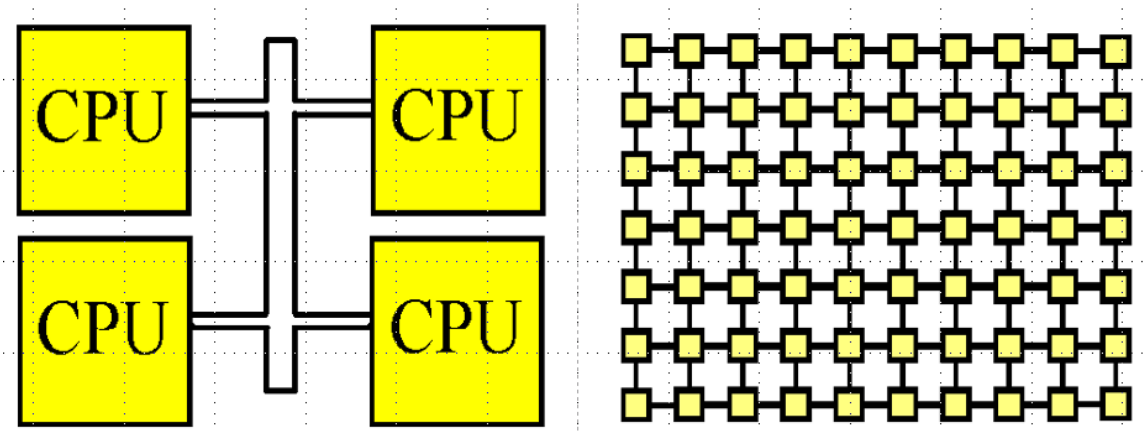

Рис. 2.10.1. Крупноблочный параллелизм (слева) и мелкозернистый параллелизм (справа)

По виду различают функциональный параллелизм и параллелизм по данным (см. рис. 2.10.2). При функциональном параллелизме параллельно работающие вычислители обрабатывают различные операции или реализуют разные этапы обработки общей задачи. При параллелизме по данным вычислители одинаково обрабатывают свои комплекты данных.

Рис. 2.10.2. Функциональный параллелизм (совокупность блоков разного цвета) и параллелизм по данным (блоки одного цвета)

Различают следующие уровни параллелизма:

- арифметический уровень (ускорение выполнения арифметических операций за счет введения структурной избыточности),

- уровень команд (одновременное выполнение нескольких инструкций процессора),
- потоковый уровень (аппаратная поддержка выполнения нескольких потоков команд),
- уровень задач.

Существуют разнообразные классификации параллельных компьютеров. Наиболее известной из них является классификация Флинна. На рис 2.10.3 приведены ее классы и примеры видов архитектур, принадлежащих этим классам.

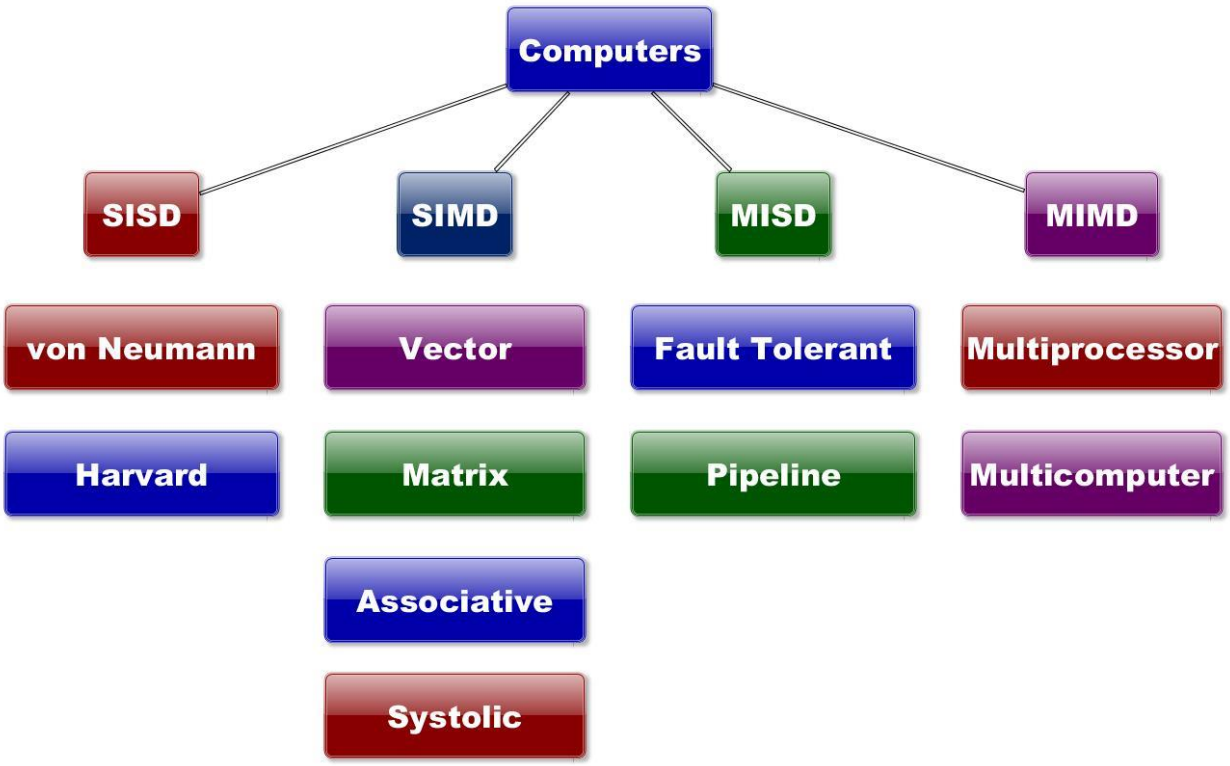

Рис. 2.10.3. Классификация Флинна

2.11. SIMD параллелизм

Обычные (скалярные) команды за раз обрабатывают только один комплект данных (см. рис. 2.11.1). Во многих современных архитектурах присутствует SIMD параллелизм под названием векторных расширений. Они представляют собой набор регистров для хранения векторов чисел и команды для выполнения операций над всеми элементами вектора одновременно (см. рис. 2.11.2).

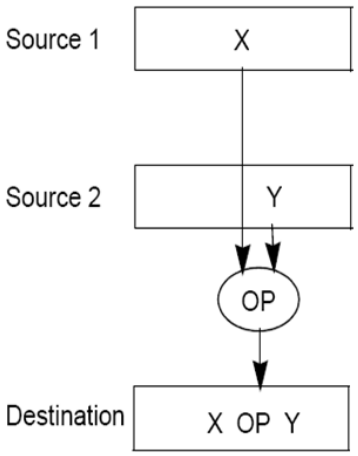

Рис. 2.11.1. Скалярная команда

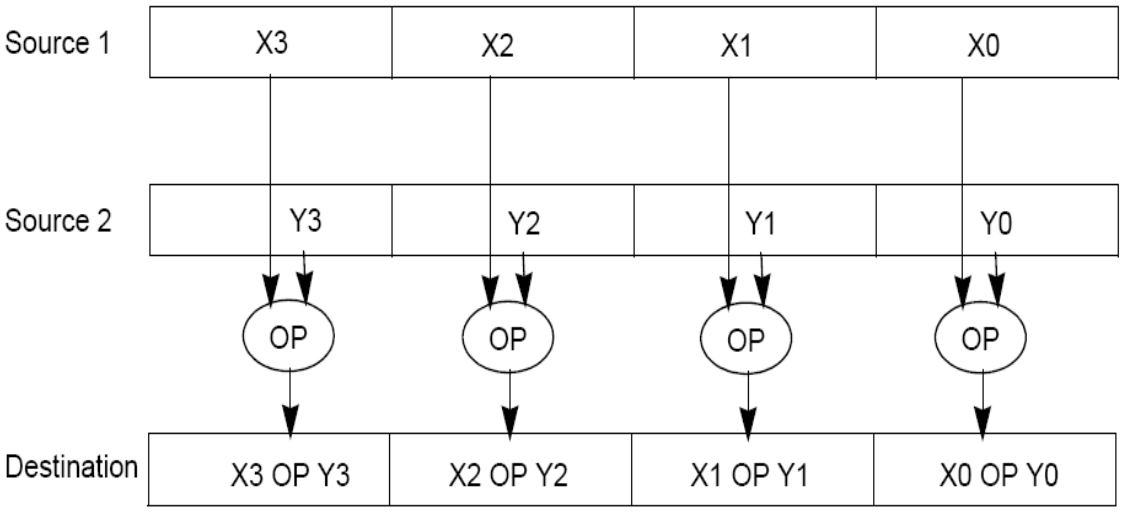

Рис. 2.11.2. Векторная команда

Выигрыш во времени у векторных команд (по сравнению со скалярными) достигается по ряду причин. Во-первых, декодирование команды выполняется один раз для нескольких одинаковых операций с различными данными. Во-вторых, для компонентов векторной операции не требуется проверять информационные зависимости. И, в-третьих, вычислительные устройства загружаются равномерно, поскольку одна и та же операция над различными данными выполняется за равное время.

Векторные команды могут быть многочисленными. Перечислим их на примере векторных расширений x86/x86_64 SSE:

- арифметические (ADDPS, ADDSS, SUBPS, DIVPS, SQRTPS, RSQRTPS, MAXPS, MINPS);

- логические (ANDPS, XORPS);
- сравнение (CMPPS, CMPSS);
- сдвиг;
- перемещение данных (MOVUPS, MOVAPS, MOVSS);
- перестановка и распаковка элементов (SHUFPS);
- преобразование формата.

Использование векторных расширений SSE для повышения эффективности программ рассматривается в гл. 7.

2.12. Параллелизм на уровне команд

Параллелизм на уровне команд предполагает одновременное выполнение процессором нескольких независимых команд программы. Для этого процессоры с параллелизмом на уровне команд (ILP-процессоры) имеют несколько функциональных устройств. Кроме того, каждая стадия конвейера ILP-процессора способна обрабатывать более одной команды за такт. Максимальное число команд, обработку которых конвейер может завершать за такт, называется шириной конвейера. Ширина конвейера определяется шириной самой «узкой» стадии конвейера. Реальная производительность ILP-процессора на конкретной программе зависит от количества независимых команд, которые могут быть выполнены параллельно, и ограничивается шириной конвейера. Так как большинство используемых языков программирования являются последовательными (Си, Фортран, Java), то перед ILP-процессорами встает сложная задача выявления параллелизма в последовательных программах.

По способу выявления независимых команд компьютеры данного класса различаются суперскалярные процессоры и VLIW-процессоры (Very Long Instruction Word), т.е. процессоры с длинным командным словом.

Суперскалярный процессор получает от компилятора программу в виде последовательности команд. Специальное устройство в процессоре, называемое диспетчером, динамически выявляет в этой последовательности независимые команды, которые затем распределяются по нескольким функциональным устройствам для параллельного выполнения.

Механизм работы диспетчера суперскалярного процессора следующий. Множество очередных последовательно идущих и ожидающих выполнения

команд программы образуют так называемое «окно просмотра». В рамках этого окна диспетчер ищет готовые к выполнению команды, по несколько за такт, и отправляет их на выполнение к соответствующим функциональным устройствам. Команда считается готовой к выполнению, если все ее зависимости разрешены (все команды, от которых она зависит, уже выполнены). Типичным размером окна просмотра современного микропроцессора является 100-200 команд.

В зависимости от способа выбора команд на выполнение различают процессоры с упорядоченным выполнением команд и с выполнением команд вне порядка. В процессоре с упорядоченным выполнением команд диспетчер рассматривает команды только в том порядке, в котором они идут во входной последовательности. Примерами суперскалярных процессоров с упорядоченным выполнением команд могут служить процессоры UltraSPARC T1, Intel Pentium, Intel Atom. В процессоре с выполнением команд вне порядка (OoO) диспетчер рассматривает команды в пределах всего окна просмотра. Таким образом, OoO-процессор в действительности выполняет команды не в том порядке, в котором они следуют в программе, а переупорядочивает их без нарушения логики программы. Как результат, производительность OoO-процессора обычно выше. На последней ступени конвейера правильный порядок выдачи результатов восстанавливается, чтобы обеспечить корректное выполнение программы. Примерами суперскалярных микропроцессоров с выполнением команд вне порядка являются процессоры Intel Pentium III, Intel Pentium 4, Intel Xeon, AMD Opteron.

Существенным для производительности суперскалярного процессора является наличие в программе независимых команд. Ограниченный набор архитектурных регистров приводит к появлению в коде так называемых ложных зависимостей, т. е. таких ситуаций, когда две команды логически независимы (например, две команды записи значения в регистр), но используют один и тот же архитектурный регистр. Для повышения производительности суперскалярный процессор вынужден разрешать ложные зависимости с помощью механизма переименования регистров. Переименование регистров – это динамическое отображение архитектурных регистров, использующихся в коде программы, на аппаратные регистры. Количество архитектурных регистров обычно невелико, типичными являются числа 8, 16, 32. Количество аппаратных регистров значительно больше, например, 40, 72, 80, 96. В случае ложной зависимости командам назначаются разные архитектурные регистры, в результате чего они становятся независимыми и могут выполняться параллельно.

Очевидно, что динамическое переименование регистров и выявление независимых команд увеличивают сложность аппаратного обеспечения. Значительная часть кристалла суперскалярного процессора занимает управляющая логика, в результате чего под ресурсы (регистры, функциональные устройства, кэш-память) остается меньше места. Еще одним недостатком является то, что суперскалярный процессор в динамике не может выявить все независимые команды в программе. Это объясняется, во-первых, требованием использовать быстрые алгоритмы для поиска независимых команд, и, во-вторых, ограниченной длиной окна просмотра. Преимуществом суперскалярного процессора является использование для распараллеливания информации, которая доступна только в момент выполнения программы. Кроме того, производительность суперскалярных процессоров, в отличие от VLIW-процессоров, в меньшей степени зависит от качества кода, что обеспечивает хорошую переносимость программ и хорошую производительность «в среднем».

Компилятор для суперскалярного процессора в распараллеливании команд участвует косвенно, поскольку в коде, который он генерирует, нет указаний на независимые команды. Максимум, что может сделать такой компилятор – это расположить независимые команды в последовательности рядом, чтобы диспетчер процессора мог легко их найти.

Примерами суперскалярных архитектур являются x86/x86-64, ARM, POWER, PowerPC, Alpha, SPARC.

VLIW-процессоры, т.е. процессоры с длинным командным словом, получают от компилятора программу в виде последовательности связок команд. Команды в каждой связке являются независимыми друг от друга и могут выполняться параллельно. Связка простых команд образует одну составную команду, или длинное командное слово. На Рис. 2.12.1 представлена связка команд VLIW-процессора Transmeta Crusoe, состоящая из восьми независимых простых команд.

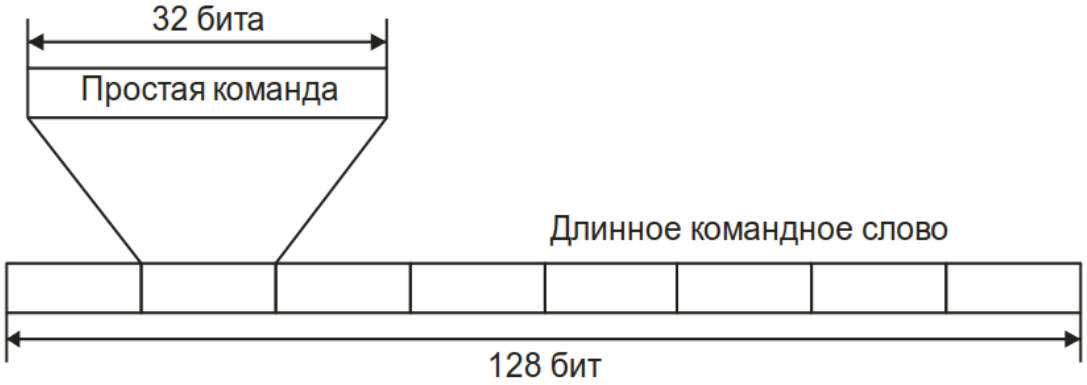

Рис. 2.12.1. Структура связки команд VLIW-процессора

Компилятор для VLIW-процессора выполняет статический анализ программы, выявляет независимые операции и формирует последовательность связок для выполнения VLIW-процессором. Число команд в связке определяется архитектурой процессора и равно ширине конвейера. Для каждой простой команды в коде указаны все необходимые аппаратные ресурсы для ее выполнения: аппаратные регистры, функциональные устройства. Если компилятор не нашел достаточное количество независимых команд для формирования связки, он дополняет ее командами NOP («нет операции»). По сути, компилятор выполняет почти полное детальное планирование выполнения потока команд на VLIW-процессоре.

Такой способ организации параллельного выполнения команд значительно упрощает логику работы процессора: нет проверки зависимостей между командами, которые компилятор объявил независимыми, нет внеочередного выполнения команд, поскольку компилятор уже определил порядок выдачи команд, нет необходимости проверять конфликты по ресурсам, поскольку все необходимые ресурсы уже назначены. Диспетчер VLIW-процессора только выполняет закодированные в программе указания. Управляющая логика занимает мало места на кристалле VLIW-процессора, оставляя больше места для ресурсов. В результате ширина конвейера типичных VLIW-процессоров (6-8 команд) больше ширины конвейера типичных суперскалярных процессоров (3-5 команд). Более того, реально достигаемая производительность VLIW-процессоров на большинстве задач выше, чем у суперскалярных процессоров. Это объясняется тем, что компилятору доступен для анализа весь код программы, а не короткое окно просмотра, и компилятор не связан при планировании жесткими временными рамками.

С другой стороны, формирование подробного плана выполнения программы на стадии компиляции приводит к тому, что часть параллелизма не может быть выявлена вообще, поскольку у компилятора нет информации о зависимостях, которые формируются в процессе вычисления. В ряде случаев команды могут оказаться зависимыми или независимыми при разных запусках программы (например, при различных входных данных программы). Компилятор для VLIW-процессора генерирует параллельный код на самый общий случай, т.е. код, который должен правильно работать при любых

входных данных программы. И если для двух команд есть возможность оказаться зависимыми, то компилятор вынужден назначить им последовательное выполнение, даже если в подавляющем большинстве случаев эти команды бы были независимыми. Кроме того, в отличие от суперскалярных процессоров, производительность VLIW-процессоров сильно зависит от качества кода, а VLIW-код жестко «привязан» к конкретной микроархитектуре процессора.

Примерами VLIW-архитектур являются Intel i860, Intel Itanium, Elbrus 2000, процессоры фирмы Transmeta. Одна из первых VLIW архитектур The Multiflow TRACE 7/300 (см. рис. 2.12.2) позволяла работать со связками из семи команд: одна команда ветвления, четыре команды целочисленной арифметики (две из которых могли быть командами доступа к памяти) и две команды с плавающей запятой.

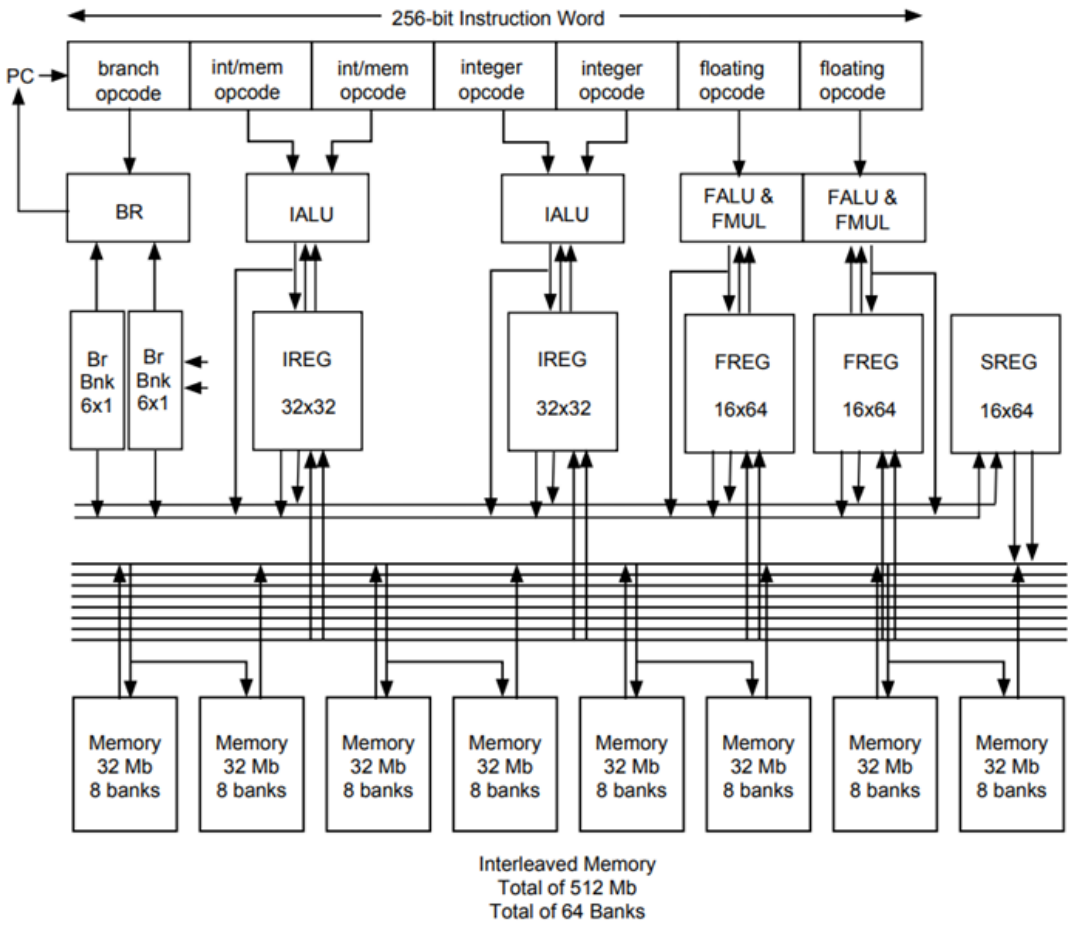

Рис. 2.12.2. Структура The Multiflow TRACE 7/300

2.13. Параллелизм на уровне потоков

Термин «поток» в зависимости от контекста может использоваться в различных смыслах. Например, программный поток – это термин операционной системы, определяемый как последовательность команд, выполняемых в составе процесса. В данном разделе используется другое определение потока – как потока команд, поступающего на вход процессору. Процессор может выполнять поток команд независимо от других потоков команд или одновременно с ними, если у процессора есть аппаратные ресурсы для этого. Параллелизм на уровне потоков (или многопоточность) означает выполнение нескольких потоков команд, которые относятся к разным одновременно выполняющимся программам (или к разным веткам одной параллельной программы).

Производительность процессоров до 2000-х годов повышалась в существенной степени за счет повышения тактовой частоты. После этого и по настоящее время повышение производительности связано с введением параллелизма, в первую очередь на уровне потоков – с помощью введения аппаратной поддержки многопоточности и на уровне задач – с помощью многоядерных архитектур и мультикомпьютеров.

Различают программную и аппаратную многопоточность. Программная многопоточность – это способ выполнения нескольких потоков команд на одном процессоре путем периодического переключения его с одного потока на другой. Недостатком программной многопоточности является то, что переключение процессора между потоками средствами операционной системы выполняется сравнительно долго (тысячи тактов процессора). Все современные операционные системы реализуют программную многопоточность, чтобы создать видимость параллельного выполнения программ.

Аппаратная многопоточность – это аппаратно поддержанный способ выполнения нескольких потоков команд на одном процессоре. Аппаратная поддержка многопоточности заключается в организации совместного использования ступеней конвейера и других ресурсов процессора (регистров, кэш-памяти) несколькими потоками команд. При одновременном выполнении команд из нескольких потоков ресурсы процессора загружаются более эффективно, чем при одном потоке, так, что они меньше простаивают. Для операционной системы один многопоточный процессор логически выглядит как несколько процессоров (по числу поддерживаемых им потоков команд).

Существует несколько способов организации аппаратной многопоточности в микроархитектуре:

- При крупнозернистой многопоточности конвейер в каждый момент времени выполняет команды только одного потока. Если в какой-то момент произошла длительная задержка (в результате промаха при обращении в кэш-память, зависимости по управлению или по другой причине), то процессор автоматически переключается на другой поток команд и начинает выполнять его команды. Аппаратная поддержка переключения потоков позволяет делать это быстро, без каких-либо задержек. Действительно параллельного выполнения потоков команд здесь, как и при программной многопоточности, не происходит. Ускорение достигается за счет того, что длительные задержки одного потока оказываются скрытыми за выполнением команд другого потока.
- При мелкозернистой многопоточности каждая ступень конвейера на каждом такте переключается между потоками. Таким образом, даже небольшие задержки в несколько тактов оказываются скрытыми. Параллельного выполнения потоков здесь также не происходит. Примером реализации мелкозернистой многопоточности является процессор UltraSPARC T1.
- При одновременной многопоточности ступени конвейера могут обрабатывать одновременно несколько команд с разных потоков. Таким образом, происходит действительно параллельное выполнение команд разных потоков. Ускорение здесь достигается за счет того, что недозагруженные и простаивающие при одном потоке ступени конвейера загружаются командами из других потоков. Примером реализации одновременной многопоточности является технология Intel Hyper-threading в процессорах архитектуры x86/64.

Большинство стадий процессора не различают команды нескольких потоков и могут обрабатывать их единообразно. Ресурсы процессора, отвечающие за хранение данных (кэш-память, регистровые файлы, различные таблицы и буферы), в процессорах с аппаратной поддержкой многопоточности используются одним из следующих способов: они или дублируются для всех аппаратных потоков, или делятся поровну между аппаратными потоками, или же совместно ими используются. Например, кэш-память второго уровня в ранних многопоточных процессорах Intel делилась строго на две части (по числу ядер), а в более поздних стала использоваться совместно всеми ядрами.

Для того чтобы использовать возможности аппаратной поддержки многопоточности, необходимо либо выполнять больше одной программы,

либо, если выполняется одна программа, она должна иметь несколько потоков команд, т. е. быть параллельной. При выполнении одной последовательной программы многопоточный процессор работает не лучше (иногда даже хуже за счет меньшего числа ресурсов), чем однопоточный.

2.14. Многоядерная архитектура

Ядра процессора, как правило, размещаются на одном кристалле (см. рис. 2.14.1). На том же кристалле располагаются общие для ядер контроллер памяти и шинный интерфейс. Контроллер памяти выступает «посредником» между процессором и памятью. Шинный интерфейс служит для подключения контроллеров периферийных устройств к процессору. Оперативная память расположена отдельно от процессора, на системной плате. Кроме того, на системной плате расположены контроллеры периферийных устройств (например, контроллер дисков или контроллер шины USB).

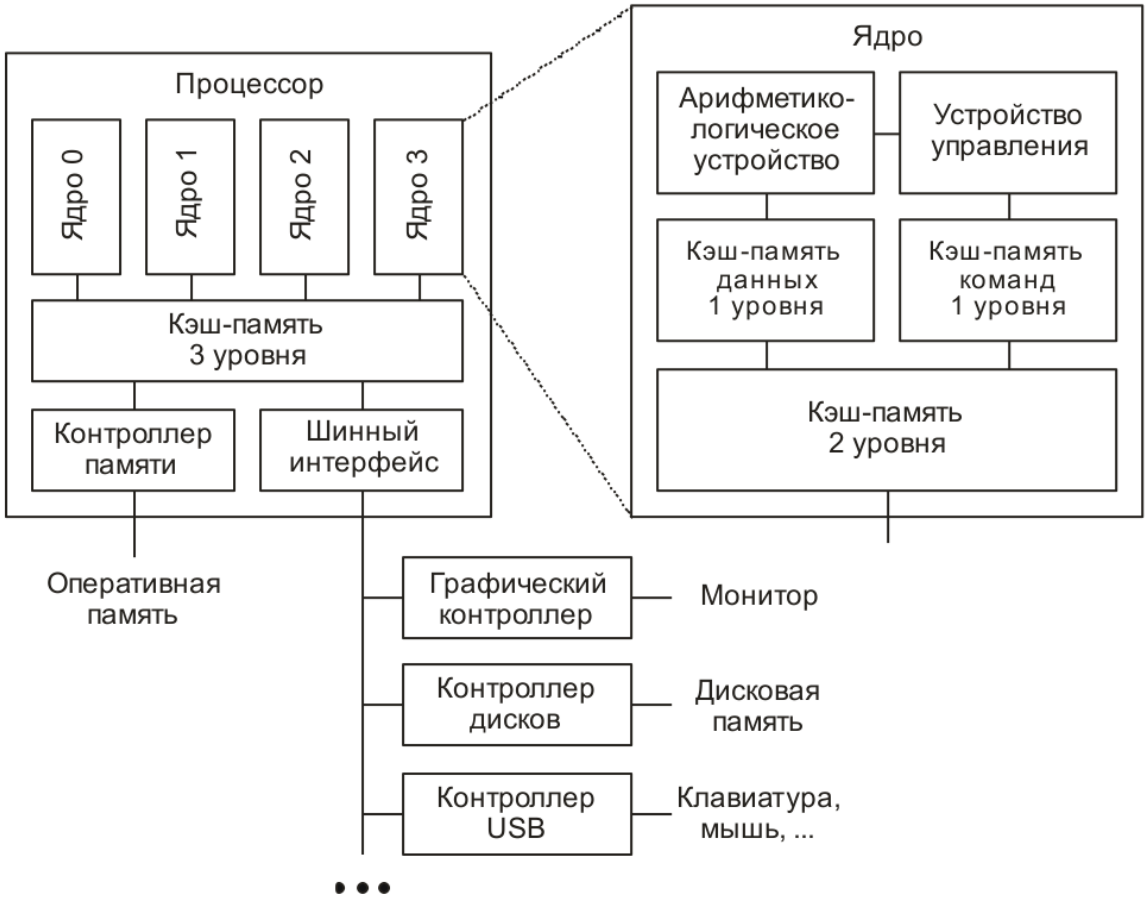

Рис. 2.14.1. Структура компьютера с многоядерным процессором

Большинство современных высокопроизводительных процессоров – многоядерные. Для того, чтобы они работали с максимальной производительностью необходимо одновременное исполнение нескольких

процессов или потоков. Введение в построение многопоточных приложений дается в гл. 6 данного конспекта лекций. В таблице 2.14.2 приводятся данные о количестве ядер в некоторых современных процессорах.

Табл. 2.14.2. Количество ядер в современных процессорах

Архитектура	Число ядер	Прочее
Intel Xeon Phi 2 KNL	64-72	64 bit
AMD APU Zen	4-32	64 bit
Qualcomm Snapdragon 810	8	64 bit, 2GHz, 4*A57, 4*A53
Green Arrays GF144	144	14мкВт-650мВт
Tilera TILE-Gx8072	72	

.

Общая структура 72-ядерного процессора Tilera TILE-G8072 приведена на рис. 2.14.3. Структура Intel Xeon Phi 2 Knights Landing с 32 блоками по 2 ядра каждое, основные параметры и состав блока приведены на рис. 2.14.4. В каждом блоке кроме ядер есть кэш L2 на 1 МБ и 4 VPU (векторных устройства). На самом кристалле расположено 16Гб оперативной памяти. В дополнение к этому можно подключить до 384Гб дополнительной оперативной памяти. Производительность на векторных операциях над числами с плавающей запятой с одинарной точностью превышает 6 TFLOPS (6 трлн. операций), а на двойной превышает 3 TFLOPS.

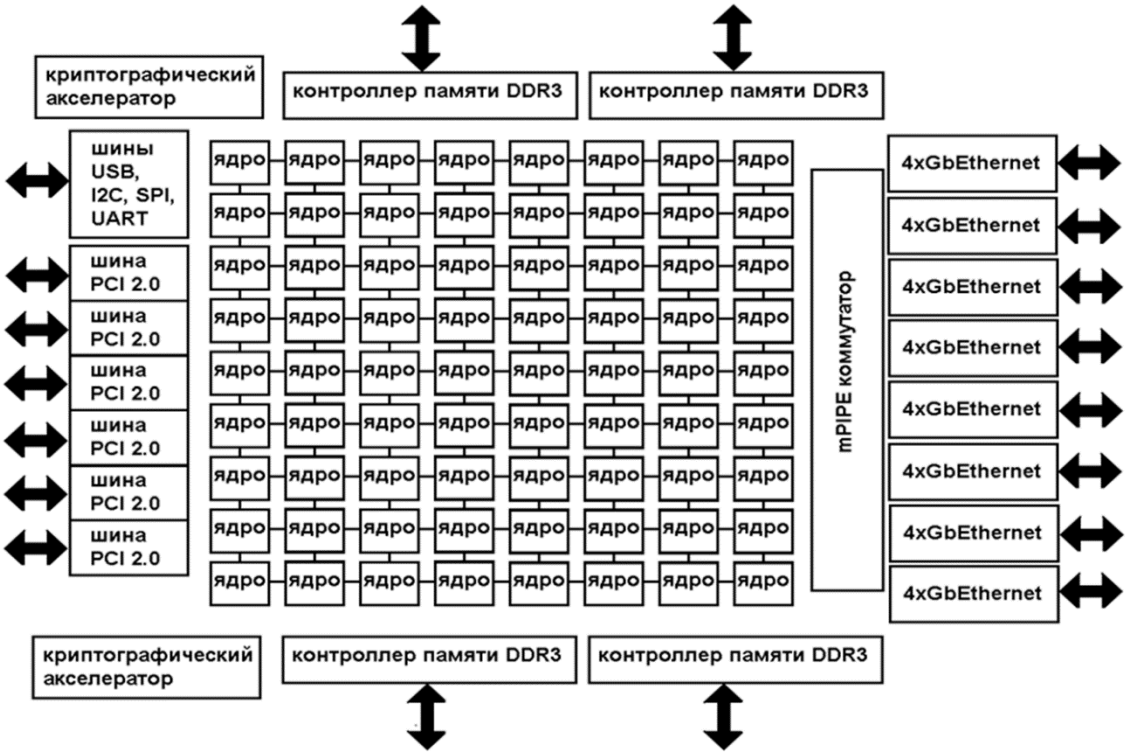

Рис. 2.14.3 Общая структура Tilera TILE-G8072

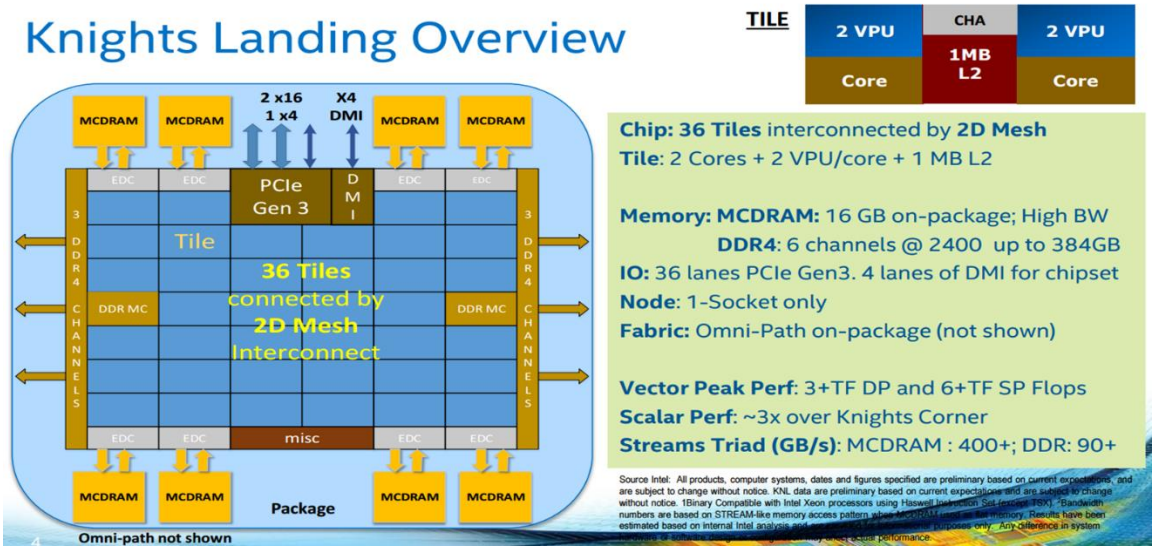

Рис. 2.14.4 Общая структура и параметры Intel Xeon Phi 2 KNL с 72 ядрами

2.15. Производительность.

Производительность компьютера можно оценивать с помощью различных показателей. Простейший из них, тактовая частота, отражает только производительность центрального процессора. Тактовая частота

(количество тактов в секунду) показывает количество простых команд (с длительностью в 1 такт) может выполнить процессор за одну секунду. В современных высокопроизводительных процессорах она лежит в диапазоне от 2 до 5.5 ГГц. То есть время выполнения простой команды составляет доли наносекунды. Для различных специализированных микропроцессоров и микроконтроллеров ее диапазон шире – от 1 МГц до более чем 1ГГц. Тактовая частота слабо соответствует общей производительности компьютера. Во-первых, многие команды выполняются за несколько тактов. Во-вторых, интегральная производительность узла является функцией от многих параметров, наиболее существенные из которых – объем кэша и оперативной памяти, время доступа к кэшу, оперативной памяти и внешней памяти, наличие и характеристики дополнительных вычислительных устройств в узле (акселераторы, GPU, FPGA). В-третьих, общая производительность зависит от того, какая задача выполняется в узле. Одни задачи в большей требуют интенсивный счет, и здесь становятся важными набор архитектура и организация ЦП и GPU. Другие задачи интенсивно используют оперативную память, и тогда на первый план выступает организация подсистемы памяти (кэш, оперативная память). Третьи задачи интенсивно работают с внешней памятью. Соответственно, для них существенными являются задержки при обращении к внешним устройствам и пропускная способность шины, по которой они подключены.

Другой показатель производительности – число операций, производимых компьютером за единицу времени, например, миллиардов операций над числами с плавающей запятой за одну секунду (GFLOPS) или количество обсчитанных клеток массива за секунду в миллионах (MCell/sec). Различают пиковую производительность и реальную. Пиковая (применительно к GFLOPS) может быть рассчитана из количества одновременно обрабатываемых чисел и времени выполнения операции. Реальная производительность, достигаемая на тех или иных вычислениях, всегда ниже пиковой (иногда в несколько раз). Более того, она отличается для разных задач. Она обычно выше для задач с интенсивным счетом, чем для задач с интенсивным использованием оперативной памяти. Рейтинг самых производительных компьютеров мира (top500.org) рассчитывается на тестах High Performance LinPACK и на настоящий момент измеряется даже не в GigaFLOPS, а в TeraFLOPS и PetaFLOPS (см. табл. 2.15.1).

Табл. 2.15.1. Рейтинг TOP500 за июнь 2018 года

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM DOE/SC/Oak Ridge National Laboratory United States	2,282,544	122,300.0	187,659.3	8,806
2	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway , NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
3	**Sierra** - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband , IBM DOE/NNSA/LLNL United States	1,572,480	71,610.0	119,193.6	
4	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000 , NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482
5	**AI Bridging Cloud Infrastructure (ABCI)** - PRIMERGY CX2550 M4, Xeon Gold 6148 20C 2.4GHz, NVIDIA Tesla V100 SXM2, Infiniband EDR , Fujitsu National Institute of Advanced Industrial Science and Technology (AIST) Japan	391,680	19,880.0	32,576.6	1,649
6	**Piz Daint** - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 , Cray Inc. Swiss National Supercomputing Centre (CSCS) Switzerland	361,760	19,590.0	25,326.3	2,272
7	**Titan** - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x , Cray Inc. DOE/SC/Oak Ridge National Laboratory United States	560,640	17,590.0	27,112.5	8,209
8	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom , IBM DOE/NNSA/LLNL United States	1,572,864	17,173.2	20,132.7	7,890
9	**Trinity** - Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect , Cray Inc. DOE/NNSA/LANL/SNL United States	979,968	14,137.3	43,902.6	3,844
10	**Cori** - Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect , Cray Inc. DOE/SC/LBNL/NERSC United States	622,336	14,014.7	27,880.7	3,939

Тема 3. Архитектура программного обеспечения

Один из принципов архитектуры фон Неймана, принцип программного управления, говорит о том, что структура компьютера не зависит от решаемой на нем задачи. Компьютер управляется программой, состоящей из команд, хранящихся в памяти. В первых компьютерах программы имели простой набор функций (обычно численные расчеты). Программа загружалась в компьютер и монопольно пользовалась всеми его ресурсами. По мере усложнения сложности решаемых с помощью программ задач и реализуемых ими функций, стал выявляться набор функций, который был общим для всех

или многих программ. Программное обеспечение разделилось на системное и прикладное. Системное программное обеспечение реализует общие функции, а прикладное – специфические функции конкретной решаемой задачи.

3.1. Системное ПО.

Главные функции системного ПО – управление ресурсами компьютера (в первую очередь памятью и устройствами ввода/вывода), реализация файловой системы для удобного представления данных во внешней памяти и управление выполняющимися прикладными программами. Центральные функции (управление памятью, процессами) обычно реализуется ядром операционной системы (ОС). Поддержка различных файловых систем, протоколов нижних уровней и разнообразных внешних устройств чаще представлена в загружаемых модулях ядра. Разнообразные сервисы (журналирование событий, организация очереди принтера) обычно реализуются отдельными системными программами.

Понятие задачи (процесса) и потока. Под задачей (процессом) будем понимать экземпляр запущенной программы. Два запуска программы редактора текстов образуют в ОС два отдельных и независимых процесса. Самые главные составляющие процесса – это:

1) набор значений регистров процесса,

2) дескриптор процесса,

3) виртуальное адресное пространство со всеми значениями ячеек памяти.

При переключении между процессами ОС сохраняет значения регистров одного процесса и восстанавливает значения регистров другого процесса, которые или сформированы загрузчиком при старте нового процесса, или сохранены ранее при переключении с данного процесса. Основные виды данных, которые есть в дескрипторе процесса – это:

1) идентификаторы процесса и его группы,

2) идентификатор пользователя и его группы,

3) приоритет процесса.

В адресном пространстве хранятся:

1) переменные окружения процесса и текущая рабочая директория,

2) исполняемый код программы из главного исполняемого файла программы и из динамически подгружаемых библиотек (TEXT SEGMENT),

3) один или несколько стеков вызовов подпрограмм (STACK),

4) область памяти для статических объектов данных (DATA SEGMENT, BSS),

5) область динамически выделяемой памяти (HEAP),

6) область памяти для данных с выделением непосредственно через функции по работе с виртуальной памятью,

7) дескрипторы открытых файлов,

8) указатели на функции обработчиков сигналов,

9) объекты для взаимодействия между процессами (общие окна в памяти, семафоры, мутексы, очереди сообщений и т.д.).

Пример адресного пространства в GNU Linux см. на рис. 3.1.1., а для Win32/64 – на рис. 3.1.2. Кроме данных самого процесса, в адресное пространство проецируются код и данные ядра ОС (KERNEL SPACE). Часть адресного пространства, выделенная для ОС, не видна прикладной программе. Варианты разбиения адресного пространства на системную и прикладную части приведены на рис. 3.1.3.

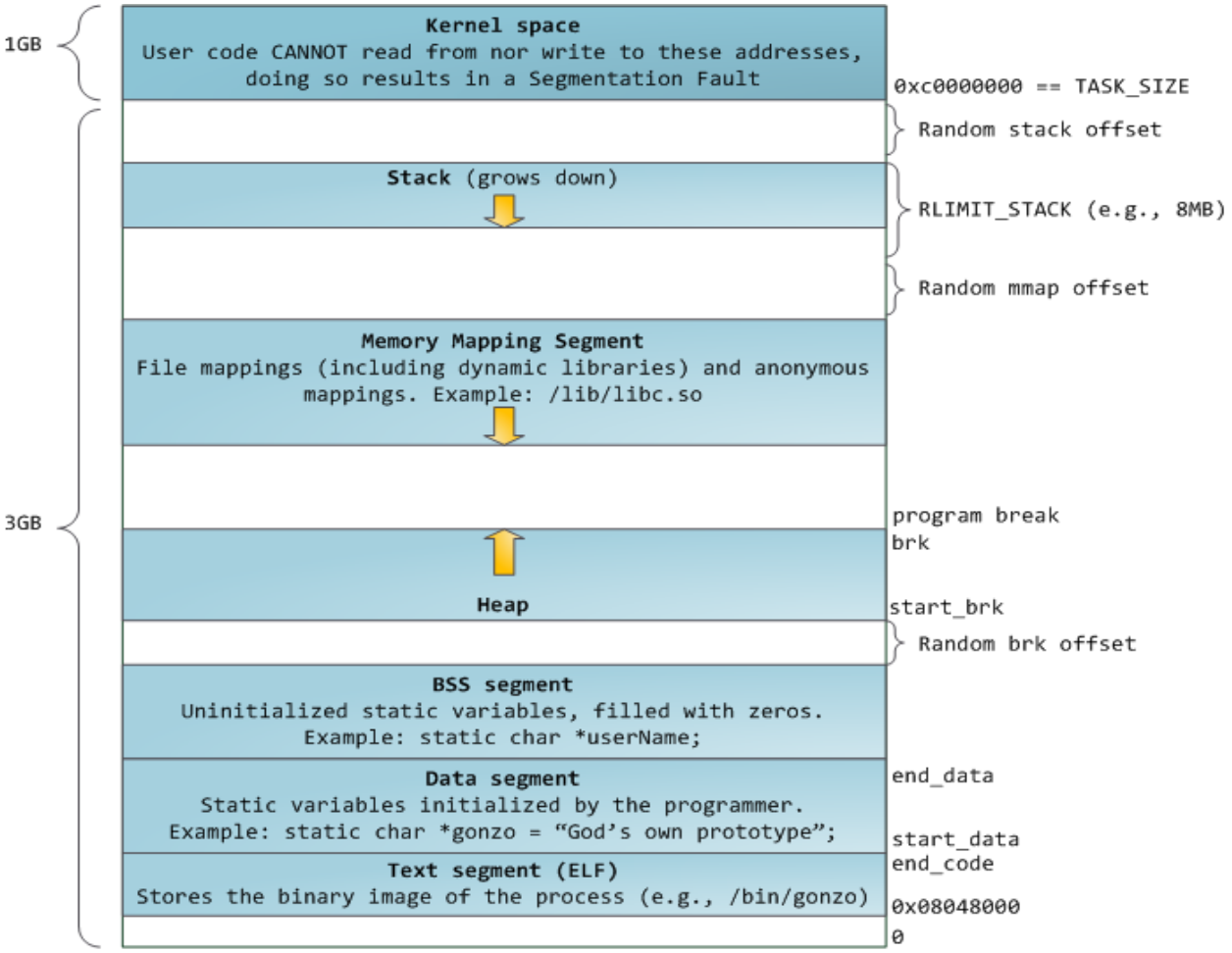

Рис. 3.1.1. Адресное пространство процесса в GNU Linux

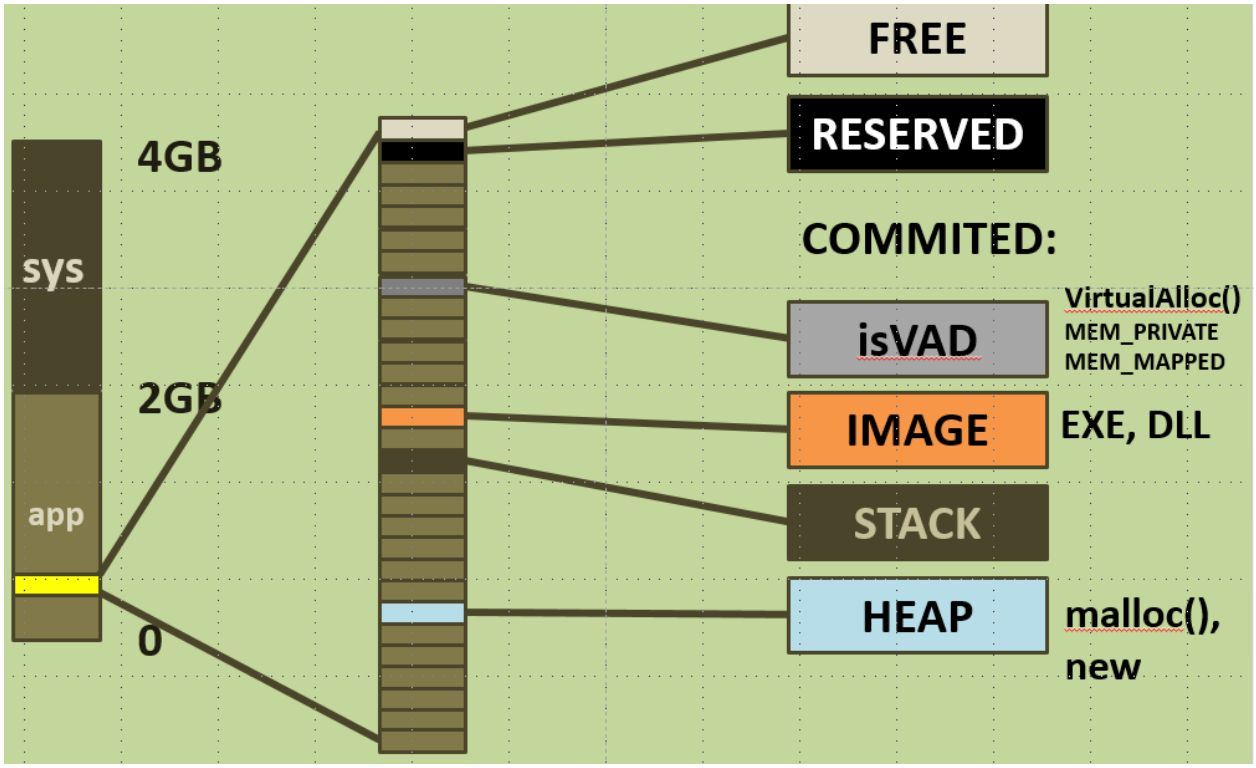

Рис. 3.1.2. Адресное пространство процесса в GNU Linux

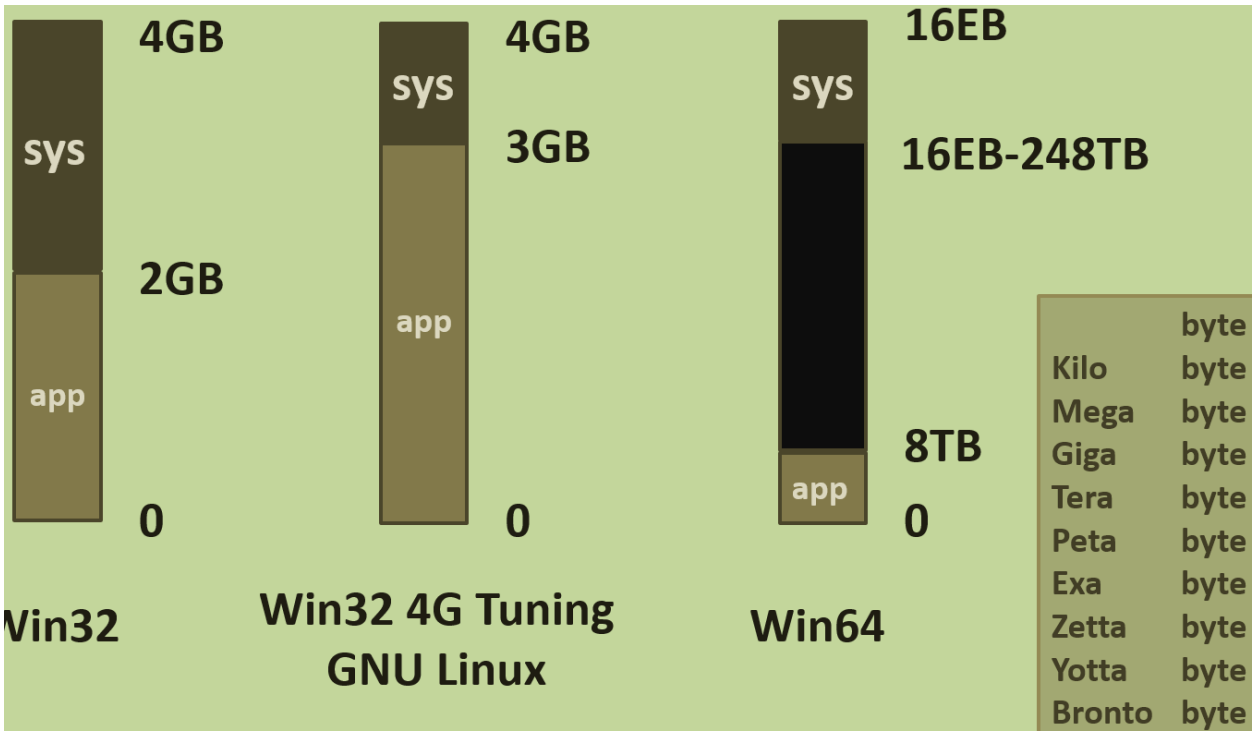

Рис. 3.1.3. Разбиение адресного пространства процесса в GNU Linux и Windows на системную и прикладную части

Внутри процесса может выполняться один или несколько потоков команд. Потоки пользуются общими ресурсами процесса, в котором они существуют. В современных ОС планировщик обычно работает с потоками, а не с процессами. Каждый поток имеет собственный набор регистров (включая адрес текущей команды и адрес вершины стека), собственный стек и небольшое собственное хранилище данных (TLS - Thread Local Storage) внутри общего адресного пространства процесса. Введение в программирование многопоточных приложений с использованием POSIX Thread и OpenMP приведено далее в гл. 6 этого конспекта лекций.

Планировщик задач и потоков. Современные системы, как правило, рассчитаны на выполнение многих задач в режиме разделения времени, т.е. с таким быстрым переключением между задачами, что у пользователя возникает иллюзия их одновременного выполнения. Как было рассмотрено выше, многие современные компьютеры имеют в своем составе многоядерные процессоры. Ядра ОС используют это для истинно параллельного исполнения задач. Задача планировщика – переключение между задачами или потоками и выбор очередного потока (или задачи), которые будет выполняться.

Существуют разные дисциплины выбора очередной задачачи. Кратко рассмотрим некоторые из них.

Исторически одной из первых применяемых дисциплин была “Короткие Задачи Вперед” (Short Jobs First) (см. рис. SC_1). Как следует из названия, планировщик запускает задачи с наименьшей оценкой времени исполнения. Дисциплина простая в реализации, но имеет два существенных недостатка. Во-первых, длинные задачи могут никогда не начать исполняться, если постоянно идет поток коротки. Второй недостаток существеннее. Для задач нужно заранее знать оценку времени ее исполнения. Это было выполнимо для систем пакетной обработки с набором счетных задач. Но в современной среде исполнения программ таких оценок нет. В современных планировщиках эта дисциплина не используется. Одна в некоторой степени (как составная часть более сложных дисциплин) продолжается применяться в менеджерах очередей для суперкомпьютеров и кластеров (см. например, менеджер очередей PBS).

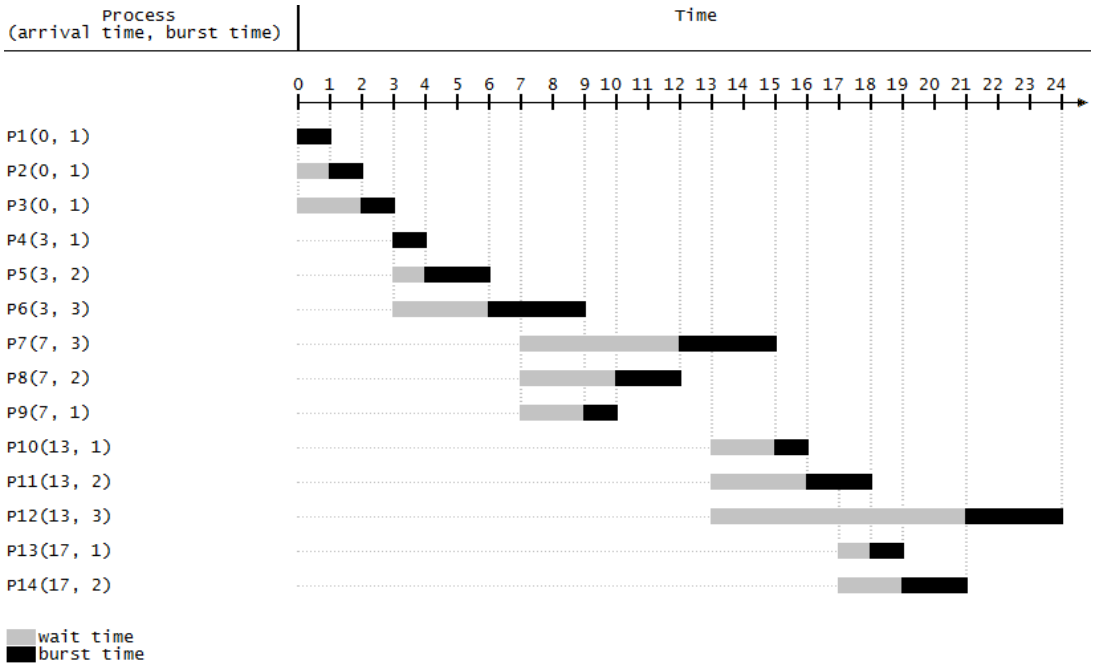

Рис. SC_1. Дисциплина Short Jobs First

Дисциплина обработки по кольцу (Robin Round) (см. рис SC_2) поочередно выполняет все активные задачи. Реализация этой дисциплины проста, но она в неизменном виде практически не используется, так как в ней не учитывается важность разных задач, все получают примерно одинаковую долю процессорного времени.

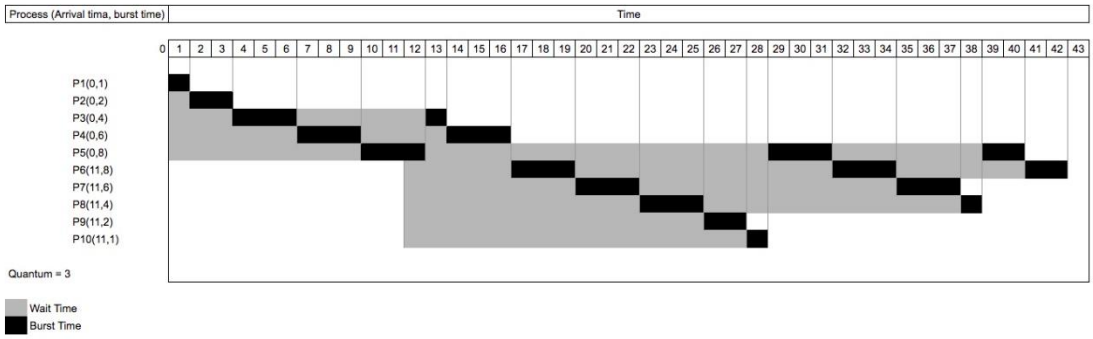

Рис. SC_2. Дисциплина Robin Round

Во многих современных планировщиках используется дисциплина обработки по кольцу, дополненная приоритетами. Для каждого приоритета формируется свое кольцо задач или потоков. Переход к исполнению на менее приоритетном кольце происходит только тогда, когда на текущем кольце все задачи по тем или иным причинам не готовы исполняться (сон, ожидание завершения операции ввода/вывода и т.д.).

При построении многопоточного приложения следует аккуратно пользоваться системной приоритетов. Во многих случаях достаточно оставить приоритеты по умолчанию. Если есть некоторые обработчики событий, которые должны срабатывать как можно быстрее, но время работы которых мало, то потоку для их исполнения можно повысить приоритет. При этом после выполнения этих обработчиков поток следует приостановить до следующего срочного запроса. Процедуры с длительными вычислениями никогда не следует запускать в потоках с высоким приоритетом. Если эти вычисления не сопряжены с непосредственным взаимодействием с пользовательским интерфейсом программы, то для потока, наоборот, можно снизить приоритет до фонового уровня.

К другим основным задачам ядра ОС относится ввод/вывод, поддержка драйверов устройств ввода/вывода. Драйвера могут быть частью ядра или внешними подгружаемыми модулями. В контексте данного курса эта чать функционала представляет меньший интерес. При необходимости ознакомления с деталями реализации ядер рекомендуется для начала выбрать доступное в исходных текстах ядро какой-либо небольшой ОС для встраиваемых систем.

Сервисы ОС. Системные функции, не вошедшие в ядро, могут быть в подгружаемых модулях ядра (если конкретная ОС реализует такую

возможность), в виде отдельных сервисов и утилит. Сервисы могут реализовываться как отдельными программами, выполняемыми в отдельных процессах, так и в виде динамически компонуемых библиотек. В последнем случае один системный процесс может использоваться многими сервисами. Бывает и обратная ситуация. Интенсивно используемый сервис (например, Web сервер при высокой загрузке) может поддерживаться несколькими одновременно работающими процессами. Типичная ОС для персонального компьютера или небольшого сервера может иметь приблизительно следующий набор основных сервисов:

- Аутентификация пользователей
- Файловые системы
- Терминальный ввод-вывод
- Журналирование системных событий
- Квотирование ресурсов пользователям
- Очередь принтера
- Поддержка устройств, выполняющих функцию вычислительных сопроцессоров (акселераторы, графические карты, ПЛИС)

Middleware. Многие системные функции изначально не включались в состав ОС. Например, это функции поддержки выполнения распределенных и параллельных приложений на ОС, которые сами по себе не являются распределенными. Такой класс системного ПО называется middleware. Он обеспечивает взаимодействие между процессами, выполняющимися на разных компьютерах, с помощью очередей сообщений, поддерживают каталоги функциональных интерфейсов, их реализаций и распределенное хранилище файлов. Примеры middleware (разнотипные и очень отличающиеся по сложности): CORBA, SOA, JMQ, MSMQ. В плане реализации эффективного ПО middleware дает возможность понизить трудоемкость распределенных решений, когда производительности централизованных и локальных не достаточно для поставленных требований.

Микросервисная архитектура. Микросервисная архитектура – современная технология, основывающаяся на принципах модульного программирования и развивающая их. Рассмотрим ключевые свойства архитектуры.

1. Структура приложения. Приложение состоит из слабосвязанных модулей небольшого размера, которые реализуют только одну функцию. Эти модули называются микросервисами.

2. Микросервисы выполняются в отдельных процессах и взаимодействуют между собой асинхронной посылкой сообщений. Они обладают определенной устойчивостью к сбоям в других микросервисах, которые они используют
3. Инфраструктура позволяет легко добавлять новые микросервисы и заменять старые.
4. Приложение может быть разнородно в плане используемой платформы. Разные микросервисы могут быть написаны на разных языках, использовать разные базы данных, разный форматы представления данных и разные протоколы взаимодействия.

Более мелкомодульная структуризация в первую очередь облегчает сопровождение реализованного программного обеспечения. Это, однако, в большей степени относится к эффективности реализации ПО, а не эффективности ПО. В плане эффективности ПО эта структуризация упрощает выявление тех частей приложения, которые в наибольшей степени влияют на общую производительность. Например, это процедуры с интенсивным счетом, которые регулярно запускаются при многих сценариях работы. Оптимизировать отдельные фрагменты кода малого размера существенно проще, чем традиционные модули больших размеров.

При составлении материала в этом разделе использовались источники:

https://martinfowler.com/articles/microservices.html

https://en.wikipedia.org/wiki/Microservices

Самый большой пласт системного ПО – это разнообразные системные библиотеки, расширяющие функционал ОС по многочисленным более частным направлениям, но которые по-прежнему не имеют четкой прикладной ориентации. Это, например, библиотеки поддержки сетевых протоколов, форматов данных, описания синтаксиса языков, разбора исходных текстов на разных языках программирования и описания данных, библиотеки математических функций, библиотеки поддержки периферийных устройств без драйверов, работающий в ядре в привилегированном режиме.

Библиотеки поддержки сетевых протоколов. Программная часть современной инфраструктуры сети Интернет использует сотни разных протоколов. Некоторые из них тривиальны, например, NTP. Другие отличаются особой сложностью, например, ssh. Их самостоятельная реализация отвлекла бы существенную часть ресурсов разработчиков от основных задач из проекта. Поэтому, целесообразно использовать уже

готовые реализации требуемых протоколов. Например, для поддержки протокола ssh на стороне клиентской программы разработаны две библиотеки: libssh, libssh2.

Общая рекомендация заключается в том, что на этапе проектирования, когда требуемый функционал разрабатываемого приложения уже достаточно детализирован, желательно провести поиск существующих решений, например, доступных в виден библиотек, которые реализуют части требуемого функционала. Это сократит дальнейший объем работ по кодированию. Во-первых, это даст больше времени на оптимизацию того специфического функционала приложения, который нельзя нигде заимствовать. С другой стороны, обычно отлаженная заимствованная библиотека дает существенно большую эффективность реализации, чем реализация того же функционала специально только для одного приложения. Например, функции для выполнения операций линейной алгебры, которые реализованы самостоятельно разработчиком для отдельного приложения, будут иметь худшее время выполнения по сравнению со стандартными реализациями от Intel или NVIDIA.

Тема 4. Инструментарий разработчика программного обеспечения

4.2. Компилятор и его составляющие. Назначение компилятора – преобразовать исходную программу, написанную программистом на языке высокого уровня в бинарный вид, который может быть исполнен процессором. Основные этапы преобразования программы – это препроцессирование, лексический анализ, синтаксический анализ, генерация кода, оптимизация кода.

Лексический анализ служит для преобразования последовательности символов в последовательность смысловых элементов программы – лексем. Лексический анализ производится частью транслятора, называемой сканером. Обычно выделяется несколько групп лексем. Это – ключевые слова языка, знаки операций, идентификаторы, числовые константы, строковые константы.

В процессе синтаксического анализа формируется описание абстрактной структуры модуля (или файла) программы (см. далее схемы программ). На базе этой структуры генератор кода формирует объектный файл, содержащий исполняемый код для процедур и функций отдельного файла с исходным текстом. Объектный файл не является готовой исполняемой

программы, это некоторый ее фрагмент. Сборка этих фрагментов в исполняемый файл делается вне компилятора, компоновщиком (см. далее по тексту). Оптимизация кода может происходить в различные моменты компиляции. Некоторые тривиальные преобразования, как свертка константных выражений может происходить при препроцессировании. Платформенно независимые оптимизации могут происходить при синтаксическом анализе или после него. Платформенно зависимая оптимизация кода может производиться вместе с его генерацией, или после нее. Примеры оптимизации в GCC рассматриваются в следующей главе этого конспекта лекций.

4.3. Статические библиотеки и библиотекари. В крупных проектах количество модулей велико. Как следствие велико и количество сгенерированных компилятором объектных файлов. Часто разные проекты могут иметь больное пересечение по общим используемым модулям и их объектным файлам. Для упрощения описания трансляции программ проектов из исходного в исполняемый вид, удобнее заменить работу с огромным числом объектных файлов с небольшим числом их коллекций, называемых библиотеками. Так как существует еще один вид библиотек, рассматриваемый ниже, для устранения неоднозначности эти библиотеки иногда называют статическими библиотеками. Основная утилита для работы с библиотеками – библиотекарь. Главные функции, которые она предоставляет разработчику

Компоновщик (linker) – утилита, собирающая исполняемые файлы (а также динамически компонуемые библиотеки, они описаны чуть ниже) из набора отдельных объектных файлов и статических библиотек. Это преобразование называется компоновкой, а иногда, сборкой или линковкой.

Загрузчик – модуль (обычно он является частью ядра ОС), который загружает и запускает на исполнение программу в бинарном виде (а также загружает необходимые ей динамические библиотеки). До широкого распространения виртуальной памяти, загрузчик в полном смысле этого слова перерез запуском программы загружал весь ее бинарный код (или основную его часть, ести использовались оверлеи). В современных ОС загрузка – это фактически некоторый этап порождения нового процесса (экземпляра запущенной программы). При этом код программы не загружается сразу в память, а только проецируется на созданное для процесса виртуальное адресное пространство. Как только начинается исполнение, возникает ситуация отсутствия нужной страницы с кодом, и механизм виртуальной памяти тогда уже загружает одну или несколько недостающих страниц. То есть в итоге по-настоящему в память подгружаются только те части кода,

которые нужно исполнить. Это существенно (вполне заметно для пользователя) сократило время запуска программы.

Динамические компонуемые библиотеки – библиотеки процедур в бинарном виде, которые, в отличие от обычных (статических) библиотек, не добавляются к исполняемому файлу программы на этапе компоновки, а загружаются непосредственно при исполнении программы в адресное пространство процесса.

4.4. Средства отладки.

Основное инструментальное средство для отладки, отладчик (англ. debugger), давно и широко распространено. Отладчик служит для поиска ошибок самого разного вида путем прогона программы в особом, отладочном режиме. Отладчик может быть как независимой утилитой (GNU GDB), так и быть частью интегрированной среды разработки программ (Microsoft Visual C++ 7.0). С точки зрения пользовательского интерфейса отладчики варьируются от текстовых (GNU GDB) до графических (отладчик в KDE KDeveloper, Microsoft Visual C++ 7.0). Базовый набор функций у отладчиков приблизительно одинаков. Они позволяют:

- инициировать выполнение программы в отладочном режиме;
- останавливать выполнение программы;
- приостанавливать и продолжать выполнение программы
- выполнять пошаговое выполнение;
- просматривать и изменять значения переменных;
- просматривать и изменять значение байтов областей памяти,
- просматривать иерархию вызовов подпрограмм;
- устанавливать и убирать точки останова по строкам исходного текста программы;
- устанавливать и убирать точки останова по командам микропроцессора в исполняемой программе.

Исполнение программы в отладочном режиме может приостанавливаться в следующих случаях:

- по команде пользователя;
- при достижении команды завершения исполнения в отлаживаемой программе;
- при возникновении исключительной ошибочной ситуации;
- при достижении точки останова (англ. breakpoint).

Набор исключительных ошибочных ситуаций зависит от платформы, так как в значительной степени определяется тем, какие прерывания и исключения реализованы в используемом микропроцессоре. Обычно набор таких ситуаций включает:

- переполнение стека;
- ошибка доступа к памяти;
- целочисленное деление на нуль;
- неверный код операции.

Пошаговое выполнение может производиться с разной степенью детализации:

- по строкам исходного текста программы без захода в вызываемые подпрограммы;
- по строкам исходного текста программы с заходом в вызываемые подпрограммы;
- по командам микропроцессора в исполняемой программе.

Хотя отладчики позволяют отлаживать широкий спектр ошибок, они имеют ограничения применимости. Во-первых, выявляемые ошибки - конкретные реализационные ошибки в текстах программ. Более глубокие причины возникновения той или иной ошибки отладчик не покажет. Для этого после сеанса отладки разработчику необходимо анализировать спецификации, проектную документацию и текст программы. Также бывают ошибки, которые происходят при исполнении обычной версии бинарного исполняемого файла, а при запуске отладочной версии они не проявляются. В таком случае можно использовать трассировку. Бывают ситуации, когда нет возможности использовать отладчик в интерактивном режиме. Например, некоторая программа должна запускаться асинхронно, в непредсказуемые моменты времени. Это типичный случай для процедур обработки прерываний, программ, реализующих серверные программы, обрабатывающие клиентские запросы с удаленных машин. Другой пример – программа должна исполниться в реальном времени, без задержек. В подобных случаях также можно пользоваться трассировкой. После каждого запуска остается свой журнальный файл. Если некоторый запуск сопровождался ошибками, можно анализировать соответствующий ему журнальный файл.

Существуют специализированные отладчики, например, отладчики для параллельных программ для систем реального времени. Кроме отладчиков есть и другие инструментальные средства отладки. Они ориентированы на поиск ошибок определенного вида (см. Electric Fence ниже).

Кроме собственно отладчиков при отладке полезны утилиты, показывающие состояние системы, которые отвечают на такие вопросы, как:

- какие процессы исполняются сейчас;
- сколько памяти (физической, виртуальной) занимает процесс;
- сколько процессорного времени потребляет процесс;
- сколько потоков у процесса;
- сколько данных принято и получено процессом по сети;
- сколько данных было записано и прочитано с внешней памяти.

Подобные утилиты имеются в среде Microsoft Visual C++. Иногда достаточно использовать Менеджер задач Windows. В ОС Linux ответы на подобные вопросы можно получить с помощью многочисленных утилит, например:

- top – информация о выполняемых системой процессах
- ifconfig – информация о сетевых интерфейсах и их загрузке
- lsmod – список подгружаемых модулей ядра, и кем они сейчас используются

При отладке программ с графическим пользовательским интерфейсом важно знать ответы на такие вопросы, как:

- какие окна созданы в системе;
- какие дочерние окна есть к данного окна;
- какие сообщения были посланы данному окну.

В среде Microsoft Visual C++ для таких целей используется утилита Microsoft Spy++.

4.5. Таймеры и средства измерения времени.

Отсчет времени с высокой точностью имеет огромное значение для организации работы как самих вычислительных систем, так и программных систем, исполняющихся на них. На самом фундаментальном уровне отчет точных промежутков времени необходим для нормального функционирования цифровых схем, реализующих всю логику работы компьютера при использовании синхронного режима работы. Существует два основных режима работы цифровых схем: синхронный и асинхронный. Синхронный режим основан на привязке всех изменений состояния в цифровой схеме к изменению периодического сигнала, синхросигнала. Например, все схемы в машине могут срабатывать в момент, когда

синхросигнал меняется с нулевого состояния на единичное (или, как говорят, по переднему фронту синхросигнала). В настоящее время подавляющее число реализаций используют именно синхронный режим. Следующее важное применение точного отсчета времени – обеспечение своевременной реакции вычислительной системы на события, связанные с организацией нормального функционирования динамической памяти (ее нужно регулярно обновлять) и управлением устройствами ввода/вывода и внешней памяти. На более высоком уровне, уровне ядра ОС, отсчет времени нужен для организации выполнения нескольких задач на одном процессоре. Еще на более высоком уровне точный отсчет времени нужен для приложений, которые ведут обработку данных или управление каким-либо оборудованием в реальном времени. Это может быть мягкое реальное время, например, при выводе видеоданных на экран, когда желательно выводить очередной кадр видеоряда четко через определенный промежуток времени (порядка 30 мсек). Некоторая задержка нежелательна, но не катастрофична. В системах же жесткого реального времени никакие задержки или опережения в обработке событий не являются допустимыми. В зависимости от характера управляемой системы, любые такие отклонения могут приводить к материальным потерям или даже представлять угрозу жизни (например, для встраиваемых систем, контролирующих дозу облучения пациента в некоторых медицинских диагностических системах). В контексте же данного курса, основное назначение измерения времени – это оценка, насколько эффективно реализована программа или отдельная процедур по эффективности использования процессорного времени.

Основные технологии, используемые в вычислительной технике для генерации периодического сигнала – это кварц, RC-цепочка и микро электромеханическая структура (microelectromechanical structure, MEMS). Традиционно для получения качественного периодического сигнала использовались кварцы. Однако, их сложно миниатюризировать, и, тем более, разместить непосредственно на микросхеме. Поэтому, когда не требуется очень точный отсчет времени, их заменяют резонаторами на RC-цепочке, непосредственно интегрированными в микросхему. Последние годы в качестве резонаторов стали использовать MEMS. Они, с одной стороны, генерируют более качественный синхросигнал, чем кварц (выше добротность). С другой стороны, MEMS можно размещать непосредственно на кристалле, изготовленном по обычной технологии.

Аппаратные средства измерения времени. Существенно различаются для разных архитектур в деталях реализации, но имеют некоторые общие свойства. Как правило, в компьютере присутствуют следующие механизмы:

1) Часы реального времени – служат для поддержания правильного астрономического времени в компьютере, даже когда он выключен.
2) Программируемый интервальный таймер – устройство, посылающее сигнал процессору, когда истек некоторый заданный интервал времени.
3) Счетчик тактов микропроцессора ведет учет количества тактов, произошедших с момента включения компьютера или его перезагрузки.

Как правило, средства защиты доступа сконфигурированы в современных системах таким образом, что обычная прикладная программ из всех этих аппаратных механизмов может пользоваться только счетчиком тактов.

В машинах с архитектурой x86/x86_64 имеются следующие аппаратные средства измерения времени: часы реального времени (Real Time Clock – RTC), программируемый интервальный таймер (Programmable Interval Timer – PIT), высокоточный таймер событий (HPET, High Precision Event Timer), интерфейс расширенной конфигурации и питания (Advanced Configuration and Power Interface - ACPI). В качестве источников для получения точного астрономического времени могут использоваться сетевой интерфейс (с использованием протокола прикладного уровня NTP), приемник GPS, беспроводный сетевой интерфейс LTE, а также приемники других интерфейсов, передающих сигнал точного времени. Рассмотрим подробнее RTC, PIT, HPET.

Часы реального времени в x86, RTC, ведут отсчет астрономического времени (даже при отключенном питании, при наличии функционирующего элемента питания). Также RTC можно настроить для периодической генерации прерываний по истечении промежутка времени. Величина промежутка варьируется в диапазоне от примерно 30 мксек до 500 мсек (при стандартной конфигурации RTC). Как правило, современные ОС используют RTC только для хранения астрономического времени. Для всех других целей используются рассматриваемые ниже таймеры. Современные таймеры позволяют измерять промежутки времени с большей точностью. Чем RTC.

Программируемый интервальный таймер в x86, PIT имеет в своем составе три отдельных таймера, каждый из которых может генерировать сигнал с программно настраиваемой частотой. Первый таймер обычно используется для регенерации динамической памяти, а второй – для

управления динамиком. Для использования в качестве генератора прерываний остается только нулевой. Изначально он использовался в ОС для реализации переключения задач и измерения интервалов времени. Но в современных системах ему на смену пришли более мощные интервальные таймеры – таймеры в APIC и таймер HPET.

Современный таймер в машинах с архитектурой x86_64 – это высокоточный таймер событий. В отличие от предыдущих таймеров, PIT и APIC, у него выше точность (минимальная частота у счетчика – 10МГц), и в нем имеется большое число счетчиков и таймеров. Он допускает легкую конфигурацию как для настройки периодической генерации прерывания по таймеру, так и для задания однократного срабатывания. Применение HPET позволило усовершенствовать планировщики задач в ОС, отказаться от постоянной периодической генерации прерывания по таймеру для вызова планировщика и перейти к исполнению программ пользователя с квантами времени переменной длины. Это повысило эффективность исполнения программ.

Механизм, обладающий наибольшей точностью при измерении малых промежутков времени – это счетчик тактов микропроцессора. В отличие от рассмотренных ранее механизмов, счетчик тактов не скрывается от прикладной программы механизмами защиты, им можно пользоваться непосредственно из программы, вызвав соответствующую инструкцию. На листинге 4.5.1 приведен фрагмент кода для измерения времени рассматриваемым способом, который взят из [tsc].

```
static inline unsigned long long getticks(void)
{
        unsigned int lo, hi;
        // RDTSC copies contents of 64-bit TSC into EDX:EAX
        asm volatile("rdtsc" : "=a" (lo), "=d" (hi));
        return (unsigned long long)hi << 32 | lo;
}
```

Листинг 4.5.1. Чтение содержимого счетчика тактов x86.

Таймеры ОС основываются на перечисленных ранее аппаратных механизмах. ОС поддерживает несколько таймеров, так как нужно решить несколько разных задач. Рассмотрим эти задачи и таймеры, которые обеспечивают их решение.

1) Учет астрономического времени производится системным таймером (system time, wall time). Начало отсчета привязано к некоторому известному моменту времени. Для разных ОС оно отличается. Например, в UNIX и GNU Linux отсчет начинается с 00:00 1 января 1970 года по Гринвичу. Единица измерения –количество прошедших секунд с указанного момента. В Windows NT отсчет производится с 00:00 1 января 1601 года по Гринвичу. Единица измерения – число 100 нсек. Интервалов, прошедших с этой точки отсчета. Аппаратные средства измерения времени в обычном компьютере не очень точные, и время в некотором компьютере может постепенно существенно отклоняться от эталонного. Для этого существует возможность его программно скорректировать (через системный вызов). Это может делать как пользователь, так и сервисная программа, получающая данные о точном времени с сервера времени в сети Интернет. Реализация таймера системного времени обычно считывает показание часов реального времени при старте системы, а дальше производит приращения, используя программируемый интервальный таймер.
2) Измерение интервалов времени, например, времени работы целой программы или отдельной подпрограммы, не может быть надежным, если использовать таймер, который в непредсказуемые моменты времени может измениться как в большую, так и в меньшую сторону. Поэтому, для измерения интервалов времени желательно избегать использования системного таймера. Для такой цели в современных ОС вводят монотонный таймер. Он начинает отсчет с момента включения или перезагрузки компьютера (или с момента, когда ядро ОС загружено и проинициализировало работу с монотонным таймером). Монотонный таймер может иметь более маленькие интервалы приращения, чем системный. Это обеспечивает как минимум большую разрешающую способность, а, как правило, и большую точность измерений. Монотонный таймер нельзя программно изменить.
3) Измерение времени работы отдельного процесса можно производить, используя таймер времени выполнения процесса. Каждый процесс имеет свой собственный таймер. Так как в современных системах для организации работы используется режим разделения времени, за некоторый интервал времени на процессоре могут запускаться несколько разных процессов, и время работы одного процесса может при высокой загрузке системы быть заметно меньше прошедшего

интервала времени. Это означает то, что во многих ситуациях монотонный таймер может давать сильно искаженные результаты (если нам нужно замерить не интервал прошедшего времени, а только ту его часть, в течение которой исполнялся интересующий нас процесс). Эта ситуация может возникать не только на одноядерных процессорах, но и на многоядерных, когда число активно работающих процессов превышает число доступных ядер. Грубый способ обнаружения этой ситуации в Windows – просмотр доли времени процессора, которая тратится на рассматриваемый процесс. Если она меньше 100%, ядро в режиме разделения времени исполняет еще какие-то процессы.

4) В некоторых ОС для многопоточных приложений можно использовать таймер времени выполнения потока.

Функции стандартной библиотеки. Программы пользуются описанными таймерами ОС не напрямую, а используя функции стандартной библиотеки. Некоторые функции относятся к стандарту Си/Си++ и код на их основе преносим под разные ОС. Примеры таких функций: gettimofday. Другие функции – специфичны для ОС. Это, например, - GetTickCounter для Windows или clock_gettime для UNIX/GNU Linux (см. табл. 4.5.2, 4.5.3).

<table>
<tr><th>ОС</th><th>Астрономическое время</th><th>Монотонный таймер</th><th>Время выполнения процесса</th><th>Время выполнения потока</th></tr>
<tr><td rowspan="2">Win32/64</td><td>GetLocalTime
GetSystemTime</td><td>GetTickCount
GetTickCount64</td><td>GetProcessTimes</td><td>GetThreadTimes</td></tr>
<tr><td rowspan="3">UNIX/GNU Linux/POSIX</td><td rowspan="2"></td><td rowspan="2">std::chrono::
steady_clock
::now()</td><td colspan="2" rowspan="2"></td></tr>
<tr><td colspan="3">clock_gettime</td></tr>
<tr><td>gettimeofday</td><td></td><td>times</td><td></td></tr>
</table>

Табл. 4.5.2. Функции измерения времени в Windows и UNIX/GNU Linux.

функция	платформа	Тип таймера	Разрешающая способность

GetTickCount	Win32/64	монотонный	10-16 мсек
GetTickCount64	Win64/64	монотонный	10-16 мсек
QueryPerformance Counter	Win32/64	монотонный	< 1 мксек

Табл. 4.5.2. Характеристики функций измерения времени.

Подробнее о функциях измерения времени см. в [timers].

4.6. Средства профилирования.

Современные программы имеют высокую структурную сложность. Разные фрагменты программы могут иметь существенно разную частоту использования. Также сильно различается время их выполнения. Поэтому, когда необходимо уменьшить время исполнения программы, т.е. оптимизировать ее по критерию времени исполнения, первая задача, которую необходимо решить – поиск тех фрагментов программы, которые потребляют наибольшую долю от общего времени ее исполнения. Так как структура обычно велика, ручной поиск таких фрагментов был бы весьма затратным по времени и усилиям разработчика. Для упрощения решения этой задачи появился класс утилит, автоматизирующих сбор статистики частоты вызовов подпрограмм и общего времени исполнения каждой из них. Это – профилировщики (англ. – profiler).

GNU profiler – отдельная утилита, не интегрированная в среду разработчика. Командный интерфейс позволяет автоматизовать профилирование с помощью GNU Profiler. Для активации профилирования компиляцию в GCC нужно производить с дополнительным ключом -pg:

```
g++ -g -pg -o 1.elf 1.c
```

В результате исполнения с профилированием можно получить отчет о количестве вызовов функций и времени их исполнения (табл. 4.6.1.).

Табл. 4.6.1 Отчет, сгенерированный GNU Profiler по результатам выполнения программы

% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
0.00	0.00	0.00	1025	0.00	0.00	a
0.00	0.00	0.00	1	0.00	0.00	b

MultiProcessing Environment (MPE) (см. рис. 4.6.1.) - инструмент для анализа производительности MPI программ. Он основан на генерации трассировочного файла в процессе исполнения программы и его последующем анализе (postmortem profiling).

MPE содержит следующие компоненты:

1) набор профилирующих библиотек для сбора информации о поведении MPI программ;

2) скрипт для компиляции MPI программ с поддержкой MPE (mpecc для программ на C);

3) визуализатор трассировочных файлов Jumpshot;

4) утилиты по работе с трассировочными файлами;

5) библиотека проверки коллективных операций и типов данных;

6) генератор исходных текстов функций интерфейса MPI с поддержкой профилирования:

7) подпрограммы настройки отладчика.

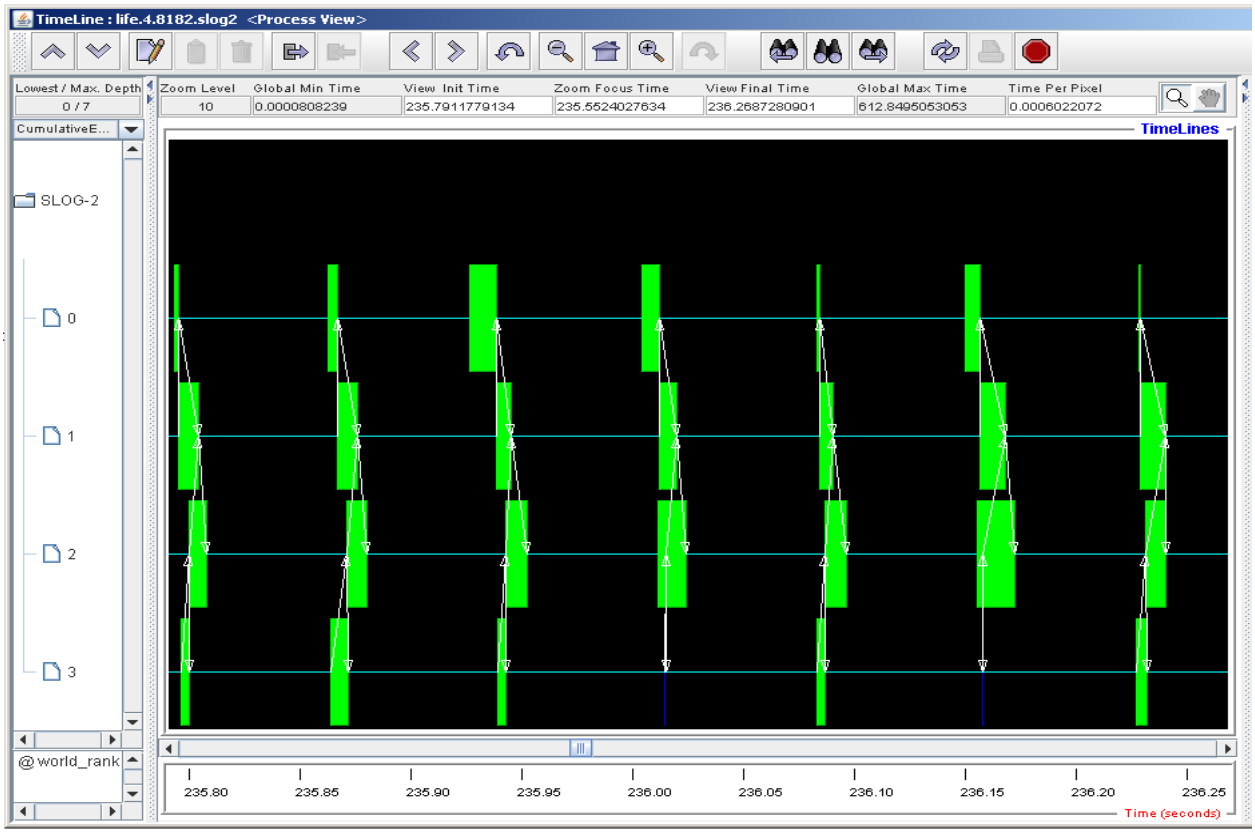

Рис. 4.6.1. Скринщот профилировщика MPE в режиме детализированного просмотра времени выполнения отдельных операций для MPI программы, запущенной на четырех узла.

4.9. Средства верификации

Средства верификации помогают разработчику в поиске некоторых типичных ошибок. В первую очередь это ошибки в работе с памятью, ошибки многопоточных приложений.

Пример средства поиска ошибок работы с памятью – Electric Fence. Он позволяет локализовать такие ошибки по работе с памятью, как выход за пределы выделенного блока динамической памяти. Рассмотрим пример на листинге 4.9.1. Запрошенный размер при выделении с использованием динамической памяти (кучи) – 5 байт. В функции strcpy в выделенный блок копируется строка большей длины. При обычной сборке (и в релизе, и в отладочном варианте) такая ошибка в момент ее совершения себя обычно не проявляет. Последствия будут заметны существенно позже, и локализовать ошибочный код в большом проекте весьма сложно.

Листинг 4.9.1. Пример программы с ошибкой по работе с памятью – выходом за пределы выделенного блока.

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

char *block;

main(){
 block = malloc(5);
 strcpy(block, "01234567890123456789");
 printf("%s\n", block);
}
```

Если же подключить Electric Fence (*gcc -g memerr.c -o memerr -lefence*), то при отладке в gdb ошибка будет выявлена сразу, при вызове strcpy (см. листинг 4.9.2).

Листинг 4.9.2. Аварийная остановка при выходе за границы выделенного блока.

```
gcc -g memerr.c -o memerr -lefence
./memerr
gdb memerr
(gdb) run
Starting program: /home/most65535/tmp.20180504/memerr
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
  Electric Fence 2.2 Copyright (C) 1987-1999 Bruce Perens <bruce@perens.com>
Program received signal SIGSEGV, Segmentation fault.
0x0000000000400719 in main () at memerr.c:11
11      strcpy(block, "01234567890123456789");
(gdb)
```

Некоторые инструментальные средства выходят за рамки функционала отдельных отладчиков, профилировщиков и средств верификации. Пример – система valgrind. Она обладает функционалом средств верификации и средств профилирования.

Тема 5. Оптимизация кода в компиляторах

Промежуточное представление программ. Для проведения оптимизации программа преобразуется из исходного вида в некоторое внутреннее представление, называемое схемой программы [Касьянов]. Оно вводится потому, что с ним удобнее работать, подвергать его разным оптимизационным преобразованиям, так как оно структурировано, упрощено, в нем не остается синтаксических особенностей, присущих исходному представлению. Кроме этого, вследствие структурированности, время доступа к фрагментам структуры в промежуточном представлении существенно ниже.

5.1. Анализ потока управления – это статический анализ кода программы, цель которого – определить порядок выполнения операторов программы. Этот порядок представляется в виде графа потока управления. Для императивных

языков программирования (Си, Си++, Фортран) в явном виде представлен в исходном тексте программ.

Понятие графа потока управления. Граф потока управления (англ. Control flow graph – CFG) – ориентированный граф, вершины которого соответствуют операторам или блокам программы, состоящим из последовательно выполняемых операторов (далее в этом разделе будем считать термины вершина и блок синонимами). Дуги графа соответствуют переходам между этими блоками программы. Имеется четыре вида вершин графа: начальный блок, преобразователь, распознаватель, блок останова. В любом управляющем графе есть только один начальный блок. Число преобразователей и распознавателей – произвольное. Из начального блока и преобразователя выходит ровно одна дуга. Из распознавателя выходит более одной дуги. Из блока останова нет исходящих дуг.

Примеры задач, решаемые анализом потока управления: 1) определение достижимости блока программы, 2) выявление циклов в подпрограммах и иерархии их вложенности.

Достижимость вершины графа, соответствующей некоторому блоку кода, - это существование пути из начального оператора в эту вершину. Если блок кода недостижим, то во многих случаях оптимизатор может его исключить, что сократит размер бинарного кода программы.

Определение иерархии вложенности циклов также необходимо для оптимизации кода программ. Чем глубже вложен цикл, тем большее число раз он выполняется. Оптимизация его кода даст существенно более ощутимый эффект по сравнению с прочим кодом. В частности, переменные, используемые в самом вложенном цикле, - самые первые кандидаты на размещение в регистрах процессора.

Обратной дугой (backedge) называют такую дугу, которая выходит из некоторого блока N и заходит в предшествующий ему блок (лежащий на пути из начального блока до блока N).

Говорят, что блок N доминирует блок M, если любой путь из начального блока до блока M проходит через бок N. Если между блоком M и его доминатором N нет другого промежуточного доминатора блока M, то M называют непосредственным доминатором. У любого блока (конечно, кроме начального) есть только один непосредственный доминатор.

Блок P постдоминирует блок N, если любой путь от N до блока останова проходит через P.

Для отыскания циклов сначала проводят поиск в глубину, выявляя все обратные дуги, а потом для каждого блока вычисляются его доминаторы.

Алгоритм поиска обратных дуг таков:

Главный алгоритм:

1. Все блоки помечаются как “не пройден”.
2. Для входного блока выполняется вспомогательный алгоритм.

Вспомогательный алгоритм:

1. Выбранный блок N помечается как “в обработке”.
2. Для каждой исходящей из N дуги проверяется статус последующего блока. Если он “в обработке”, то дуга помечается как “обратная дуга”. Если он “не пройден”, то для него выполняется вспомогательный алгоритм.
3. Выбранный блок N помечается как “обработан”.

С каждой обратной дугой из блока Y в доминирующий его блок X связан цикл. X называется заголовком цикла (loop header). Тело цикла (loop body) состоит из всех блоков, которые доминируются блоком X и из которых есть путь в блок Y.

Если известен заголовок цикла X и обратная дуга из Y в X, тело цикла вычисляется следующим образом:

1. Снять отметку со всех блоков.
2. Пометить X как “посещен”.
3. Применить описанный выше алгоритм поиска в глубину в обратном направлении от Y со следующим изменением. Всем посещенным блокам ставится отметка “посещен”. Алгоритм применяется до тех пор, пока не будет достигнут блок, в котором уже стоит отметка “посещен”. Этим блоком будет X.
4. Все отмеченные блоки образуют тело цикла.

Пример исходного графа потока управления приведен на рис. C_1, (а), а результат работы алгоритма – на рис. C_1, (б).

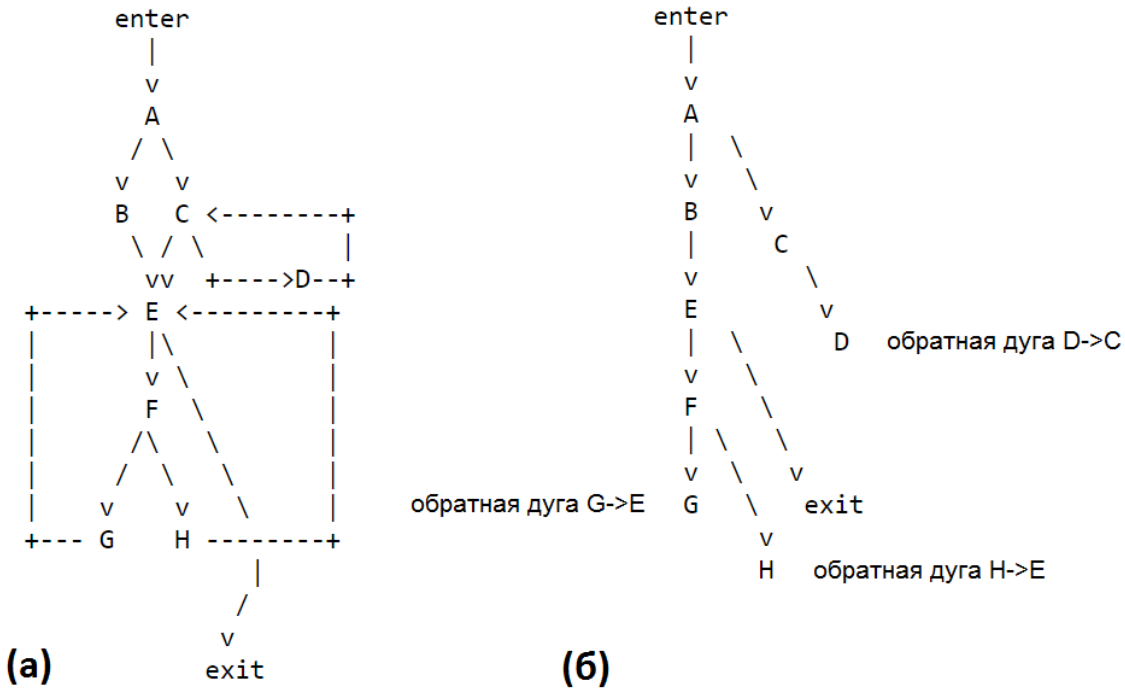

Рис. C_1. Пример исходного графа потока управления (а) и результат выполнения алгоритма поиска вглубь для нахождения обратных дуг (б)

Циклы с различными заголовками либо не связаны, либо один из них полностью вложен в другой. В последнем случае все блоки вложенного цикла принадлежат телу внешнего цикла (см. примеры на рис. C_2, C_3).

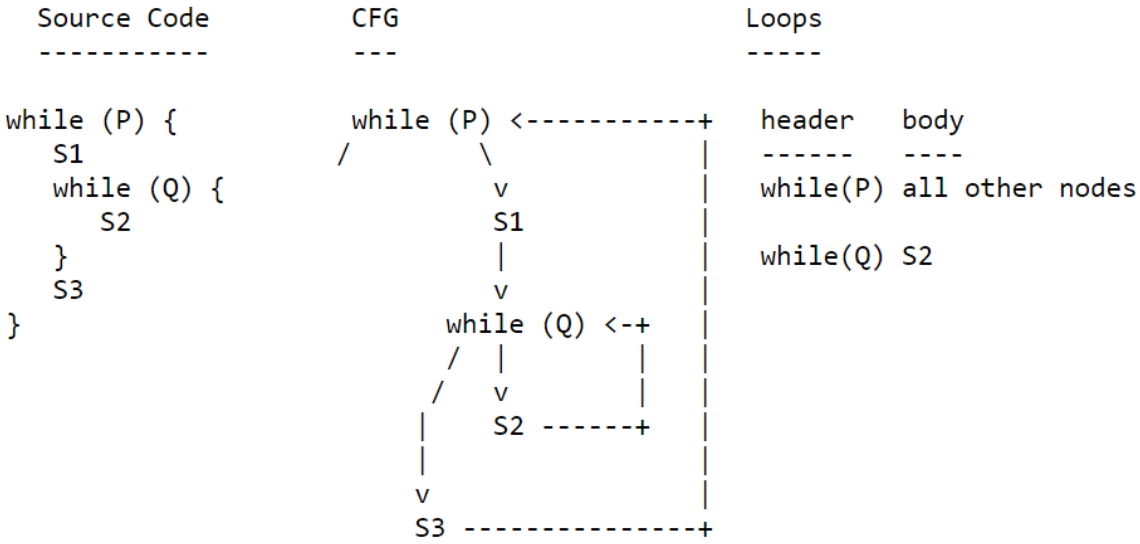

Рис. C_2. Вложенные циклы, пример 1

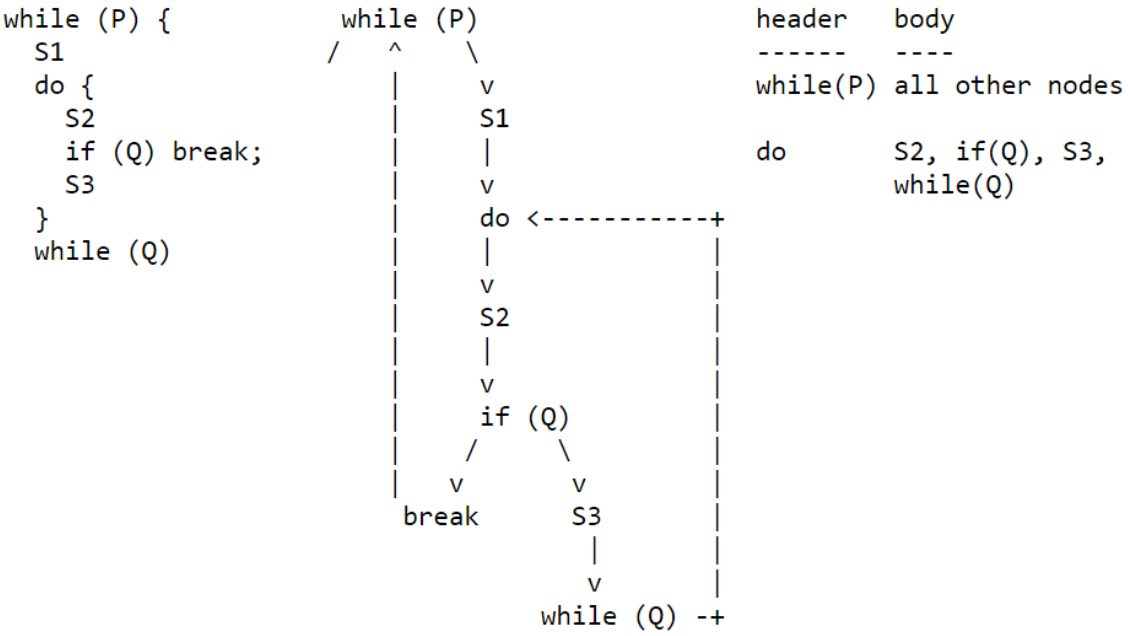

Рис. C_3. Вложенные циклы, пример 2

Пример двух циклов с общим заголовком, которые не вложены один в другой приведены на рис. C_4. В подобном случае их лучше рассматривать как один цикл.

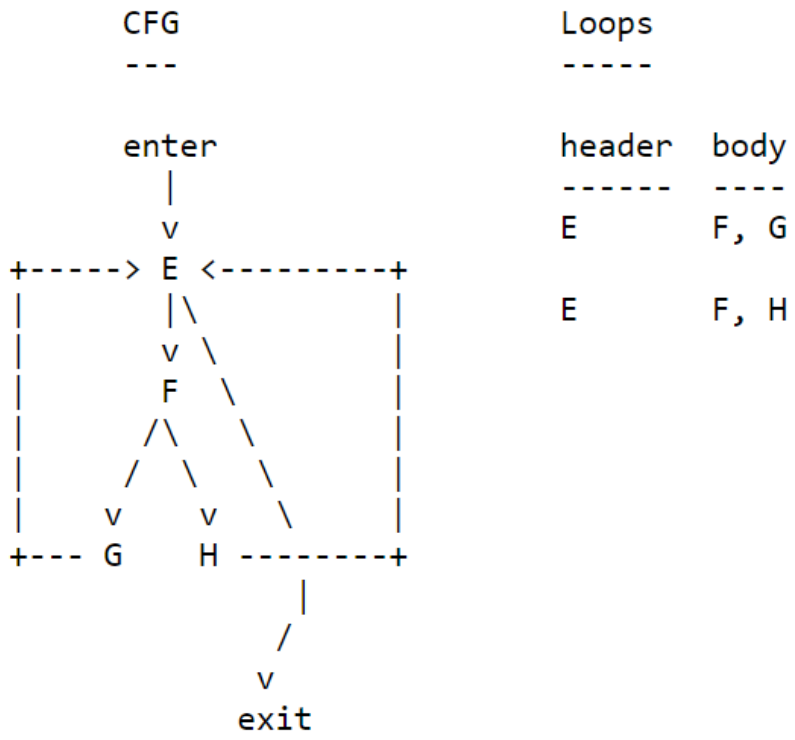

Рис. C_4. Два цикла с общим заголовком

Для нахождения циклов нужно вычислить доминаторы для блоков графа. Рассмотрим простой алгоритм для решения этой задачи [Cooper]. Более эффективный алгоритм см. в [Tarjan].

5.2. Анализ потока данных.

Задача анализа потока данных – сбор информации об использовании переменных и доступности результатов вычислений в различных точках программы. Результаты такого анализа позволяют производить различные оптимизации кода, например, исключение вычислений, результаты которых далее не используются, вынесение вычислений из веток условных операторов и т.д.

5.3. Промежуточные представления программ

Между высокоуровневыми языками программирования и машинным кодом есть большой семантический разрыв. Прямая трансляция бывает затруднительной. Многие компиляторы используют многопроходный анализ исходных текстов и несколько слоев промежуточных представлений. Эти промежуточные представления в минимальном вариант упрощают реализацию трансляции, дробя ее на более простые фазы (применение принципа “разделяй и властвуй” в борьбе со сложностью при построении компиляторов). В более широком контексте эти промежуточные представления позволяют создать экосистему для построения трансляторов с различных языков для большого числа платформ и архитектур.

Исторически первые промежуточные представления возникли вместе с первыми языками более высокого уровня, чем ассемблер. Из сохранившихся и по прежнему известных можно перечислить SECD для Lisp систем, представление в Forth, BCPL, код UCSD P-code. Широко распространенное современное представление реализовано в LLVM. На базе LLVM реализовано большое число трансляторов и других инструментов разработчика.

LLVM – технология и инфраструктура, в рамках которой сформирована большая библиотека компонентов компиляторов, интерпретаторов, средств обработки и анализа кода программ (компиляторы, анализаторы, верификаторы). Концептуально она является дальнейшим развитием технологий, рассмотренных выше.

JIT-компиляция – компиляция, которая происходит непосредственно при выполнении программы. Она применяется в средах программирования с интерпретацией для ускорения выполнения интерпретируемой программы. JIT-компиляция позволяет для таких сред обеспечить скорость выполнения программ, которая меньше скорости выполнения оптимизированного кода Си-программы примерно всего лишь в два раза. JIT-компиляция позволяет

сгенерировать бинарный код с учетом конфигурации компьютера, на котором исполняется программа.

5.4. Примеры оптимизации в компиляторе GCC.

Полный список оптимизационных преобразований см. в [GCCopti]. Так как их число велико, для упрощения работы с компилятором реализовано несколько режимов их применения, называемых уровнями оптимизации. Каждый уровень имеет свое назначение. Рассмотрим их, используя [GCCopti, quora_gcc_opti_levels].

-O0 – оптимизация отключена. Бинарный код напрямую соответствует исходному. Уровень выбирается по умолчанию, но применять его имеет смысл только при отладке (особенно при отладке по машинным командам) и для анализа сгенерированного кода.

-O1 – включены базовые оптимизации. Самая быстрая компиляция из всех уровней. Можно применять при сборке рабочих (внутренних, для разработчиков) версий многомодульных проектов, если время сборки заметно сказывается на накладных расходов во времени одной итерации разработки. Как отмечает Эдмонд Лау [Lau], в некоторых организациях сборка очередной рабочей версии разрабатываемого продукта производится до 50 раз за сутки.

-O2 – включены все основные оптимизации по скорости, кроме агрессивных. Наиболее используемый разработчиками уровни. Например, с его использованием собирается ядро ОС GNU Linux.

-O3 – в дополнение к оптимизациям с –O2 активны и агрессивные оптимизации скорости, которые увеличивают объем бинарного кода, например, раскрутка цикла и подстановка тел функций вместо вызовов (function inlining). Иногда переход к –O3 от –O2 повышает скорость выполнения программы, но происходит это не всегда.

-Os – оптимизация размера кода программы. Программы, оптимизированные на этом уровне, по сравнению с –O2 имеют меньший объем кода, но, как правило, выполняются медленнее. В отдельных случаях, благодаря более эффективному использованию L2 кэша команд, могут, наоборот, выполняться быстрее.

-Ofast в дополнение к –O3 включает еще более мощные оптимизации, которые, однако, могут приводить к генерации некорректного кода. Например, может снижаться точность вычислений для чисел с плавающей запятой.

-Og – включены только те оптимизации (как по скорости, так и по размеру), которые не мешают процессу отладки программы. Уровень удобен для проведения отладки.

Разные уровни удобны в различных ситуациях. В большом проекте не требуется использовать один и тот же уровень оптимизации для всех модулей. Например, по умолчанию можно использовать –O2. Для модулей, содержащих функции с большим объемом вычислений, можно проверить целесообразность использования –O3, а для кода плагинов, которые используются очень редко, можно выбрать –Os.

Рассмотрим конфигурацию небольшого проекта из двух модулей, каждый из которых оптимизируется на разных уровнях.

Рассмотрим примеры некоторых из самых важных оптимизационных преобразований, реализованных в GCC.

Оптимизация удаления недостижимого кода (-fdce, dead code elimination) включена на всех уровнях. На табл. 1.fdce для фрагмента исходной программы с тождественно ложным условием (оранжевая строка) бинарный код отсутствует. Отметим, что для условного оператора выше, где есть проверка на равенство нулю, эта оптимизация не применяется.

Исходный текст	Неоптимизированный вариант	Оптимизированный вариант
#include <stdio.h>	(недоступен)	

void main(void){
unsigned int x, y;
		main:
		.LFB0:
		.cfi_startproc
		pushq %rbp
		.cfi_def_cfa_offset 16
		.cfi_offset 6, -16
		movq %rsp, %rbp
		.cfi_def_cfa_register 6
		subq $16, %rsp

y = 55; **x = 55;**		**movl $55, -8(%rbp) #, y** **movl $55, -4(%rbp) #, x**
if(x == 0)		**cmpl $0, -4(%rbp) #, x** **jne .L3**
y = 77;		**movl $77, -8(%rbp) #, y**
if(0) **y = 88;**		
printf("y is %d\n", y);		.L3: movl -8(%rbp), %eax

		movl %eax, %esi
		movl $.LC0, %edi
		movl $0, %eax
		call printf #

}		leave .cfi_def_cfa 7, 8 ret .cfi_endproc

Табл. 1.fdce. Пример применения оптимизации удаления недостижимого кода.

В таблице 1.O1 приведен исходный текст и два варианта ассемблерного листинга – неоптимизированный и с оптимизацией на уровне O1 (в первую очередь размещение переменных в регистрах и оптимизация цикла).

Исходный текст	Неоптимизированный вариант	Оптимизированный вариант
#include <stdio.h>		

void main(void){
int a, b, c;
	main:
	.LFB0:
	.cfi_startproc
	pushq %rbp
	movq %rsp, %rbp
	subq $16, %rsp
		main:
		.LFB24:
		.cfi_startproc

		movl $0, %eax #c
a = 1;	movl $1, -12(%rbp)	movl $3, %ecx #,b
b = 3;	movl $3, -8(%rbp)	movl $1, %edx #a
for(c = 0; c < 8; c++){	movl $0, -4(%rbp) jmp .L2 .L3:	. L2:

b += a * c;	movl -12(%rbp), %eax #a,
	imull -4(%rbp), %eax #c
	addl %eax, -8(%rbp) #b
	movl -8(%rbp), %eax #b
		movl %edx, %esi #a
		imull %eax, %esi #c
		addl %esi, %ecx #b

a += b - 1;	subl $1, %eax addl %eax, -12(%rbp) #a	leal -1(%rcx,%rdx), %edx

}	addl $1, -4(%rbp) #c
	.L2:
	cmpl $7, -4(%rbp) #c
	jle .L3
		addl $1, %eax #c
		cmpl $8, %eax #c
		jne .L2

printf("a is %d\n", a);	movl -12(%rbp), %eax #a movl %eax, %esi	subq $8, %rsp #, .cfi_def_cfa_offset 16

	movl $.LC0, %edi
	movl $0, %eax
	call printf
		movl $.LC0, %esi #,
		movl $1, %edi #,
		movl $0, %eax #,
		call __printf_chk #
		addq $8, %rsp #,

}	leave ret .cfi_endproc	.cfi_def_cfa_offset 8 ret .cfi_endproc

Табл. 1.O1. Пример применения оптимизаций на уровне O1 – размещение локальных переменных в регистрах и оптимизация цикла.

Для следующих оптимизаций мы не будем приводить полный ассемблерный листинг функции main, а рассмотрим только его фрагменты, непосредственно затрагиваемые рассматриваемой оптимизацией.

Рассмотрим оптимизацию по раскрутке цикла (см. табл. 1.unroll). Можно заметить, что тело цикла повторяется 8 раз для смежных итераций. В итоге, общее число итераций в оптимизированном варианте в 8 раз меньше, т.е., хотя размер кода вырос, число проверок условия цикла и переходов на начало тела цикла уменьшилось в 8 раз.

Исходный текст	Неоптимизированный вариант	Оптимизированный вариант
#include <stdio.h> #define SIZE (1024 * 1024) unsigned a[SIZE], b[SIZE], c[SIZE]; void main(void){ unsigned int pos;		
for(pos = 0; pos < SIZE; pos++){	movl $0, %eax .L2:	movl $0, %eax .L2:
a[pos] = 1;	movl $1, a(%rax)	movl $1, a(%rax)
b[pos] = 10;	movl $10, b(%rax)	movl $10, b(%rax)
		leaq 4(%rax), %rdx movl $1, a(%rdx)
		movl $10, b(%rdx)
		leaq 8(%rax), %rcx movl $1, a(%rcx)
		movl $10, b(%rcx)
		leaq 12(%rax), %rsi movl $1, a(%rsi)
		movl $10, b(%rsi)

		leaq 16(%rax), %rdi movl $1, a(%rdi)
		movl $10, b(%rdi)
		leaq 20(%rax), %r8 movl $1, a(%r8)
		movl $10, b(%r8)
		leaq 24(%rax), %r9 movl $1, a(%r9)
		movl $10, b(%r9)
		leaq 28(%rax), %r10 movl $1, a(%r10)
		movl $10, b(%r10)

}	addq $4, %rax
	cmpq $4194304, %rax
	jne .L2
		addq $32, %rax
		cmpq $4194304, %rax
		jne .L2

for(pos = 0; pos < SIZE; pos++){	movl $0, %eax .L3:	movl $0, %r11d .L3:

c[pos] = a[pos] + b[pos];	movl a(%rax), %edx
	addl b(%rax), %edx
	movl %edx, c(%rax)
		movl a(%r11), %eax
		addl b(%r11), %eax
		movl %eax, c(%r11)

		leaq 4(%r11), %rdx
		movl a(%rdx), %ecx
		addl b(%rdx), %ecx
		movl %ecx, c(%rdx)

		leaq 8(%r11), %rsi
		movl a(%rsi), %edi
		addl b(%rsi), %edi
		movl %edi, c(%rsi)

		leaq 12(%r11), %r8
		movl a(%r8), %r9d
		addl b(%r8), %r9d
		movl %r9d, c(%r8)

		leaq 16(%r11), %r10
		movl a(%r10), %eax
		addl b(%r10), %eax
		movl %eax, c(%r10)

		leaq 20(%r11), %rdx
		movl a(%rdx), %ecx
		addl b(%rdx), %ecx
		movl %ecx, c(%rdx)

		leaq 24(%r11), %rsi
		movl a(%rsi), %edi
		addl b(%rsi), %edi
		movl %edi, c(%rsi)

		leaq 28(%r11), %r8
		movl a(%r8), %r9d
		addl b(%r8), %r9d
		movl %r9d, c(%r8)

}	addq $4, %rax
	cmpq $4194304, %rax
	jne .L3
		addq $32, %r11
		cmpq $4194304, %r11
		jne .L3

pos = 1024;
printf("c[%d] is %d\n",
pos, c[pos]);
}		

Табл. 1.unroll. Пример применения оптимизации по раскрутке цикла.

В качестве завершающего примера рассмотрим набор оптимизаций, векторизующих вычисления с плавающей запятой с помощью расширений SSE в архитектурах x86/x86_64 (табл. 1.simd). Это пример платформенно-зависимой оптимизации. Многие из рассмотренных выше оптимизаций можно реализовывать (хотя это не обязательно именно так в GCC) до фазы генерации бинарного кода для конкретной платформы, используя внутренне представление программы (схему программы). Неоптимизированный вариант также использует SSE для арифметики, но при этом задействованы скалярные команды, которые обрабатывают только одно число за раз (adds). В векторизованной версии использована векторная команда addps, которая производит сложение сразу над четырьмя парами чисел типа float. Соответственно, в конце каждой итерации происходит смещение сразу на четыре элемента (addq **$16**, %rax), а не на один, как в неоптимизированной версии (addq **$4**, %rax).

Исходный текст	Неоптимизированный вариант	Оптимизированный вариант
float A[1000000]; float B[1000000]; float C[1000000]; void main(void){ int x; …		
for(x = 0; x < 1000000; x++) C[x] = A[x] + B[x];	movl $0, %eax .L3: **movss** A(%rax), %xmm0	movl $0, %eax .L3: **movaps** A(%rax), %xmm0

	addss B(%rax), %xmm0
	movss %xmm0, C(%rax)
	addq **$4**, %rax
	cmpq $4000000, %rax
	jne .L3
		addps B(%rax), %xmm0
		movaps %xmm0, C(%rax)
		addq **$16**, %rax
		cmpq $4000000, %rax
		jne .L3

…		
}		

Табл. 1.simd. Пример применения оптимизаций, векторизующих арифметику с плавающей запятой

Тема 6. Оптимизация программного обеспечения разработчиком

6.1. Критерии оптимизации.

Как отмечалось во введении, критерии оптимизации весьма разнообразны:

1) время выполнения программы, процедуры, обработки одного запроса;
2) размер бинарного кода программы;
3) время реакции на поступающие программе запросы;
4) объем данных, передаваемых при взаимодействии распределенного приложения через сеть;
5) энергия, затрачиваемая на выполнение некоторой операции во встраиваемой вычислительной системе;
6) объем внешней памяти, используемой программой при решении некоторой задачи.

В этом разделе рассматривается только критерий - время выполнения (программы, подпрограммы). Под временем выполнения может пониматься как затрачиваемое процессорное время, так и время ожидания пользователя. Для программ, взаимодействующих с пользователями, более существенным является время ожидания пользователя. Для его уменьшения иногда приходится увеличивать потребление процессорного времени. Увеличивая число потоков (и, соответственно, загруженных ядер процессора), можно уменьшить время ожидания пользователя. Но, так как эффективность распараллеливания может быть меньше единицы, потребляемое процессорное

время может возрасти. В частности, оно может расходоваться на такие возникающие при многопоточности накладные расходы, как синхронизация между потоками.

6.2. Этапы разработки и связанная с ними оптимизация.

В некотором смысле оптимизация начинается уже на этапе формирования требований. У разработчиков есть цель построить программную систему, решающую некоторую проблему или дающую новые возможности пользователям. То, в какие требования трансформируется эта цель в огромной степени определяет успех последующей реализации, ее трудоемкость и качество получаемого решения. Рассмотрим несколько примеров.

1) Программа моделирования движения физических тел, между которыми действует сила гравитации. Как известно, сила гравитации действует между всеми телами, даже с минимальной массой и наибольшим удалением друг от друга. Т.е. при подсчете суммарной силы, действующей на тело, нужно учесть вклад от всех остальных тел. Но, так как точность вычислений конечная, вкладом от удаленных тел можно пренебречь. Если разбить все моделируемое пространство на фрагменты, то можно определить радиус, дальше которого вклад в результирующую силу можно не учитывать. Т.е. подсчет можно провести только для тел, находящихся в соседних фрагментах внутри сферы некоторого радиуса вокруг фрагмента, содержащего тело, для которого производится обсчет. Фактически меняется формулировка решаемой задачи. Сложность алгоритма для исходного алгоритма была квадратичной (т.е. пропорциональной квадрату количества учитываемых в модели тел) на линейную или степенную со степенью ниже двух (в зависимости от того, как происходит разбивка на фрагменты, числа тел, равномерности их распределения по пространствую и других факторов).
2) Программа построения качественного расписания движения транспорта, доставляющего продукцию со складов в магазины. Поиск оптимального решения существенно сложнее, чем сложность для решения исходной задачи предыдущего примера, так как там была полиномиальная сложность, а здесь – экспоненциальная. В реальности это означает, что размеры решаемой за разумное время задачи весьма ограничена. Многократный рост производительности компьютеров только незначительно увеличить размеры задачи (на какое-то

количество элементов, а не в разы). Если заменить задачу поиска самого оптимального решения на поиск субоптимального решения (например, с оптимальностью всего на несколько процентов меньше, чем у идеального решения), задача перестает быть экспоненциальной, и может достаточно быстро решаться для больших размеров задачи.

Оптимизация на этапе разработки в первую очередь сводится к поиску, выбору и разработке конфигурации используемой вычислительной системы, эффективных алгоритмов и представлений данных. Неудачный выбор алгоритма не может быть скомпенсирован его качественной реализацией на этапе разработки. Рассмотрим несколько пример - систему индексации и поиска в большой библиотеке текстов. При выборе линейного списка в качестве структуры для реализации ассоциативного поиска по ключу производительность полученного решения получится низкой даже при относительно небольших размерах словарей. Если же необходимо проиндексировать сотни тысяч или миллионы слов, производительность станет неприемлемой. Одни из ключевых вопросов на этапе проектирования для такой задачи – это выбор адекватной структуры данных для индексирования. Подходящими кандидатами, например, являются префиксные деревья и хэш-таблицы.

6.3. Оптимизация использования подсистемы памяти

Эффективность использования памяти складывается из эффективности использования кэша, применения предвыборки и эффективности использования виртуальной памяти.

6.3.1. Эффективность использования кэш памяти определяется долей промахов при обращении к памяти. Чем она ниже, тем меньше время доступа к данным. Можно выделить следующие виды промахов:

1) *Холодный промах* (cold miss, compulsory miss) случается, когда происходит первое (на некотором этапе работы программы) обращение к требуемым данным и они еще не загружались в кэш. Холодный промах нельзя устранить. При первом обращении к данным они всегда считываются из оперативной памяти.
2) *Промах по объему* (capacity miss) случается, когда множество активно используемых данных превышает размер кэша.
3) *Промах по конфликту* (conflict miss) происходит, когда обращаются к блокам памяти, занимающим одну и ту же строку кэша с прямым

отображением или блоки одной и той же строки у множественно-ассоциативного кэша

4) Промах при одновременном использовании блока несколькими ядрами процесса может возникать при некоторых реализациях поддержания когерентности кэша.

6.3.2. Промахи по объему устраняются уменьшением объема активно используемых данных. В некоторых задачах этот объем объективно обусловлен требованиями. Там, где нужен многократный обход больших массивов, можно разбить их на части (tiles) и производить многократный обход внутри этих частей, последовательно переходя между ними. В случае. Если ни требуется высокая точность, можно уменьшить размер обрабатываемых данных, перейдя к типу данных с более низкой точностью (например, от double к float).

Во многих случаях объем активно используемых данных увеличен искусственно, не от требований задачи, а от особенностей реализации. Рассмотрим типичные подобные случаи и способы их преодоления.

Данные на границе блоков памяти. Если некоторая переменная разместилась в памяти на границе двух блоков, то для её загрузки в кэш-память понадобится загрузить оба этих блока, хотя размер переменной может быть меньше одного блока. Это означает, что в кэш-памяти будет занято больше места, чем требуется, а сама загрузка потребует больше времени.

С точки зрения отображения данных из оперативной памяти на кэш-память, первая имеет блочную структуру, где размер блока равен размеру кэш-строки. По адресу переменной можно определить, в каком блоке памяти она находится. Переменные и структуры данных, имеющие размер более одного байта, могут располагаться в памяти на стыке двух блоков, т.е. первые несколько байт переменной могут располагаться в одном блоке, а несколько – в следующем (Рис. Z_1). Таким образом, во-первых, время доступа к такой переменной практически удваивается, так как вместо одного блока требуется загрузить два. Во-вторых, для хранения переменной в кэш-памяти потребуется две строки вместо одной. Кроме того, в некоторых архитектурах доступ в память, например, загрузка переменной в регистр, может осуществляться только по адресу, выровненному относительно размера переменной. Если же требуется загрузить переменную, расположенную по невыровненному адресу, то для этого потребуется загружать её в несколько шагов, сначала загрузив одну часть, потом другую, и затем объединив их в одном регистре. Всё это негативно сказывается на скорости доступа в память.

Рис. Z_1. Примеры (а) невыровненных и (б) выровненных переменных в памяти

Для решения этой проблемы применяют выравнивание данных. Объект данных выровнен по размеру блока n, если адрес этого объекта делится на размер блока нацело. Если переменная выровнена относительно размера блока памяти, то она занимает минимальное число блоков памяти и строк кэша.

Выравнивание данных – это изменение положения переменных в памяти таким образом, чтобы они были выровнены относительно некоторой величины. Для массивов и больших структур данных применяют выравнивание относительно размера блока. Для обычных переменных применяют выравнивание по размеру переменной. На Рис. Z_1 приведены примеры невыровненных (а) и выровненных (б) переменных, причём переменная a выровнена относительно своего размера, а переменная b выровнена, кроме этого, еще и относительно размера блока.

Выравниванием данных в оперативной памяти для обычных переменных занимается компилятор, и, как правило, человеку не нужно об этом заботиться. Внимание следует уделить пользовательским структурам данных и динамически выделяемой оперативной памяти. Чтобы проверить выровненность данных, можно преобразовать адрес переменной в число и проверить его делимость на интересующее смещение, например, на размер строки кэш-памяти.

Возможен особый случай нарушения выровненности данных. В языках C/C++ допускается явная работа с указателями. Сдвиг выровненного указателя на число байт, не кратное размеру элемента, и последующее его использование приведет к обращению в оперативную память по невыровненному адресу (листинг Z_1). Таких случаев следует избегать.

Листинг Z_1. Пример доступа к невыровненным данным

```
1   double x[N+1], *p=(double*)&(((int*)x)[1]);
2   for (i=0;i<N;i++) p[i]=1.;
```

Управлять выравниванием данных можно с помощью ключей и директив компилятора, которые позволяют запретить и разрешить выравнивание, установить размер блока, относительно которого выравнивать. На листинге Z_2 приведен пример описания структуры, в которой значение выравнивания полей установлено в 1 байт.

Листинг Z_2. Задание размера блока выравнивания в 1 байт

```
01 #pragma pack(push)/* сохранить текущее значение выравнивания */
02 #pragma pack(1)/* установить выравнивание по границе в 1 байт */
03
04 struct TighlyPackedStruct{
05   char field_char1;
06   long field_long1;
07   char field_char2;
08 };
09
10 #pragma pack(pop)/*восстановить исходное значение выравнивания*/
```

Для динамического выделения выровненной памяти можно воспользоваться такими функциями, как _mm_malloc или posix_memalign (см. листинг Z_3). Можно также выделить область памяти большего размера с помощью стандартной функции malloc и сделать в начале области необходимый для выравнивания отступ.

Листинг Z_3. Динамическое выделение памяти с выравниванием

```
01 #include <stdlib.h>
02 double *foo(void) {
03   double *var;
04   int     ok;
05   /* создать массив из 10 чисел double */
06   ok = posix_memalign((void**)&var, 64, 10*sizeof(double));
07
08   if(ok != 0)
```

```
09      return NULL;
10
11    return var;
12}
```

Разреженное размещение данных в памяти. Ещё одним случаем, когда данные располагаются в памяти неэффективно с точки зрения подсистемы памяти, является ситуация разреженного размещения данных. Речь здесь идёт не об одиночных элементах в памяти, а о группах переменных, связанных локальностью обращений во времени. Такая группа переменных может размещаться в памяти плотно или разреженно (Рис. Z_2). Плотное размещение означает, что некоторая область памяти заполнена в основном этими переменными, и в этой области отсутствуют вовсе или встречаются редко неиспользуемые ячейки, или используемые под хранение других переменных. Разреженное размещение, напротив, означает, что данные «разбросаны» по памяти на значительные расстояния друг от друга (больше размера кэш-строки).

Рис. Z_2. (а) плотное и разреженное размещение данных в памяти

С точки зрения подсистемы памяти важно, чтобы данные располагались плотно, так как обращения к плотно расположенным данным, как правило, выполняются быстрее, чем к разреженным. Это обусловлено двумя основными причинами – блочной загрузкой и аппаратной предвыборкой данных в кэш-память.

Плотным размещением простых переменных (глобальных и локальных) в оперативной памяти занимается, как правило, компилятор. Однако он не может изменить размещение и структуру данных в оперативной памяти, которую задаёт программист. Например, пусть в некоторой задаче имеется множество точек на плоскости, представленных своими координатами – парами вещественных чисел. Программист может создать массив пар чисел (вариант 1 в листинге Z_4), а может задать два массива с координатами точек по осям x и y отдельно (вариант 2 в листинге Z_4). Расход памяти при этом не изменится, но изменится размещение переменных в оперативной памяти. Если при обработке точек требуются обе их координаты, то первый способ размещения данных в оперативной памяти будет более эффективным за счёт того, что часто обе координаты точки будут находиться в одном и том же блоке оперативной памяти (Рис. Z_3, а). Особенно заметной разница будет при обработке точек в случайном порядке. Если же обработка точек осуществляется по каждой координате независимо от другой, то предпочтительным будет второй вариант (Рис. Z_3, б), особенно при обработке точек по порядку (листинг Z_4).

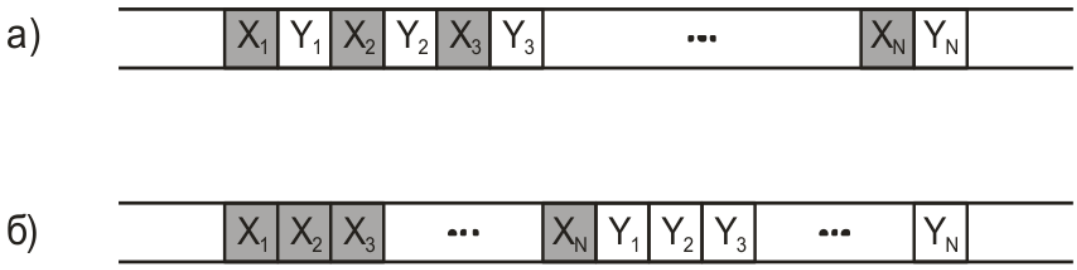

Рис. Z_3. Два способа размещения данных: (а) массив структур, (б) несколько структур

Листинг Z_4. Представление в программе множества точек на плоскости

Вариант 1	Вариант 2
`1 struct point { float x, y; };` `2 struct point points[N];`	`1 float x[N];` `2 float y[N];`

Другой пример, когда плотность размещения данных в оперативной памяти может быть повышена, связан с выравниванием данных внутри структур данных, которое выполняет компилятор. Рассмотрим листинг Z_5.

Листинг Z_5. Выравнивание данных внутри структур

```
1   struct Item1
2   { int a;      // 4 байта
3     double x;   // 8 байт
4     int b;      // 4 байта
5   };
```

```
1   struct Item2
2   { double x;   // 8 байт
3     int a;      // 4 байта
4     int b;      // 4 байта
5   };
```

Структура *Item1* в памяти будет автоматически размещена так, как показано на Рис. Z_4, а. Видно, что два поля по 4 байта в ней не используются, а только занимают место: первое поле обеспечивает выравнивание элемента x, второе поле обеспечивает выравнивание размера всей структуры.

Компилятор дополняет структуру до размера, кратного размеру наибольшего ее элемента. Если использовать в программе большой массив таких структур, то потери оперативной памяти и места в кэш-памяти могут оказаться существенными.

Рис. Z_4. Размещение данных внутри структур: (а) неоптимальное, (б) оптимальное

Более плотным является размещение в оперативной памяти структуры Item2, которое показано на Рис. 33б. Элементы этой структуры упорядочены по убыванию их размеров, что позволяет эффективно разместить их в оперативной памяти.

Можно выделить ряд рекомендаций, которые позволяют избежать проблем, связанных с плотностью размещения данных в оперативной памяти.

1. Разрешить компилятору производить выравнивание данных (обычно он это делает по умолчанию).

2. Избегать особых ситуаций явного сдвига адреса элемента на значение, не кратное размеру элемента (как в листинге Z_1).

3. Группировать данные в соответствии с их использованием.

4. Размещать данные в структурах как можно более плотно.

Нарушение локальности во времени обращений в память. В программах довольно часто один и тот же блок памяти многократно загружается в кэш-память. Такая ситуация возникает, если к тому времени, как блок оперативной памяти понадобился снова, он уже был вытеснен из кэш-памяти другими блоками. Часто это свидетельствует о неэффективном использовании кэш-памяти, ведь значительно лучше было бы загрузить переменную в кэш лишь один раз и выполнить над ней все необходимые вычисления. Количество кэш-промахов при этом сократится, а значит и программа ускорится.

Для решения этой проблемы программу перестраивают таким образом, чтобы если некоторая переменная была загружена из оперативной памяти в кэш, то над ней выполняются все возможные вычисления, пока она ещё находится в кэш-памяти. Этот приём реализуется в блочном алгоритме умножения матриц (листинг Z_B_2).

Листинг Z_B_1. Обычный алгоритм умножения матриц

```
1   for (i=0;i<n;i++)
2    for (j=0;j<n;j++)
3     for (k=0;k<n;k++)
4      c[i][k] += a[i][j]*b[j][k];
```

Листинг Z_B_2. Блочный алгоритм умножения матриц

```
1   for (ii=0;ii<n;ii=ii+nb)
2    for (jj=0;jj<n;jj=jj+nb)
3     for (kk=0;kk<n;kk=kk+nb)
4      for (i=ii;i<ii+nb;i++)
5       for (j=jj;j<jj+nb;j++)
6        for (k=kk;k<kk+nb;k++)
7          c[i][k] += a[i][j]*b[j][k];
```

В блочном алгоритме тот же набор операций, что и в обычном алгоритме (листинг Z_B_1), но они сгруппированы таким образом, как если бы матрицы a и b были разделены на блоки (подматрицы), и умножение производится над блоками последовательно (листинг Z_B_2). Размеры блоков выбираются таким образом, чтобы в кэш-памяти одновременно могли поместиться данные обоих перемножаемых блоков и блок частичного результата. Как следствие,

до окончания перемножения блоков данные будут находиться в кэш-памяти, что положительно скажется на времени выполнения программы.

Неупорядоченный обход данных в памяти. Неупорядоченный доступ в оперативную память по произвольным адресам является неэффективным, так как с высокой вероятностью запрашиваемых данных не окажется в кэш-памяти и произойдёт кэш-промах. Доступ же в память последовательно, в порядке возрастания (или убывания) адресов с фиксированным шагом является наиболее предпочтительным. Выгода такого обхода памяти обуславливается двумя основными причинами – блочной загрузкой и аппаратной предвыборкой данных в кэш-память.

Рассмотрим рис. Z_C_1. При случайном обходе некоторого массива после обращения к некоторому элементу содержащий его блок памяти загружается в кэш-память и находится там некоторое время. Некоторое время спустя этот блок вытесняется из кэш-памяти другими, вследствие чего повторное обращение к элементам этого блока снова приводит к кэш-промахам. При последовательном же обходе памяти после первого кэш-промаха идет обращение к другим элементам того же блока. В это время блок ещё находится в кэш-памяти, поэтому кэш-промахов не происходит. Таким образом, уменьшается количество кэш-промахов, и повышается эффективность использования канала кэш-память – оперативная память.

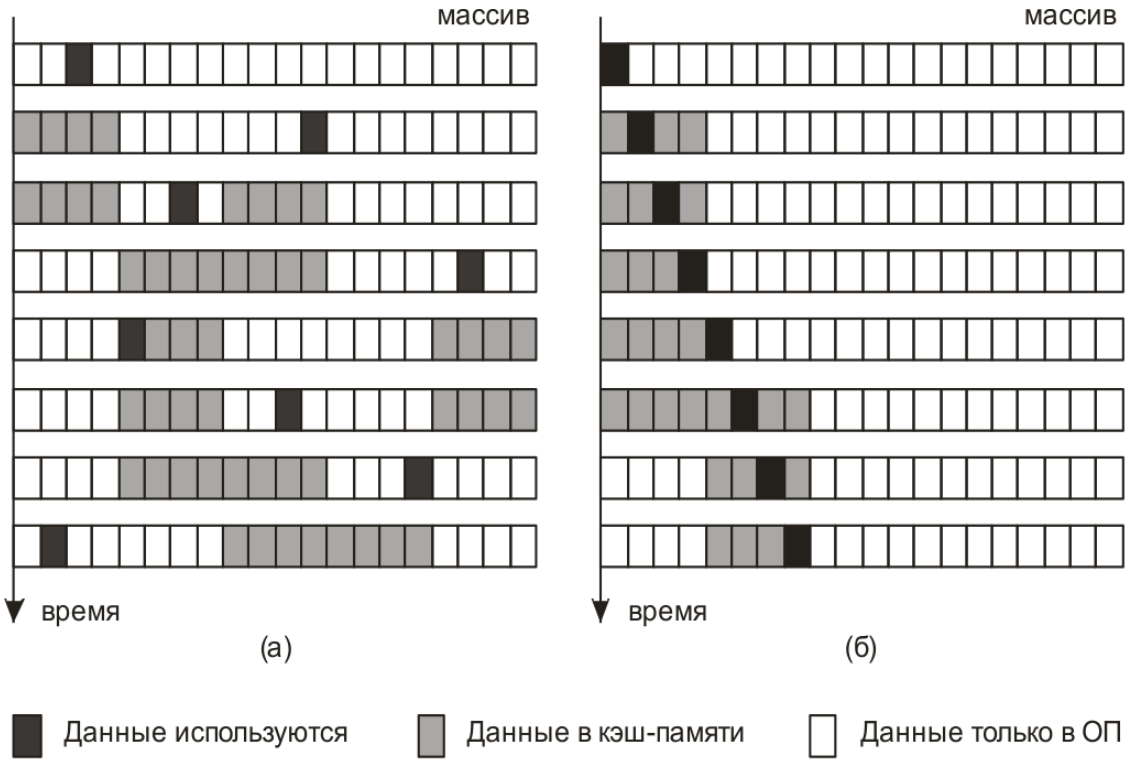

Рис. Z_C_1. Работа кэша при случайном (а) и последовательном (б) обходах памяти

На практике для повышения эффективности прикладных программ идея последовательного обхода оперативной памяти может быть использована следующим образом. Если требуется обработать элементы некоторого массива, то, по возможности, его элементы следует перебирать от первого элемента до последнего элемента с шагом в один элемент. Неоднозначность возникает при обходе многомерных массивов. Очевидно, что порядок обхода определяется порядком вложенности циклов, перебирающих индексы массива.

По стандарту языка C/C++ массивы располагаются в памяти по строкам (Рис. X_C_2). Соответственно, последовательный обход в языке C/C++ будет осуществляться, если порядок вложенности циклов установить от первого (левого) индекса к последнему (правому). Самый вложенный цикл – по последнему индексу (листинг X_B_3).

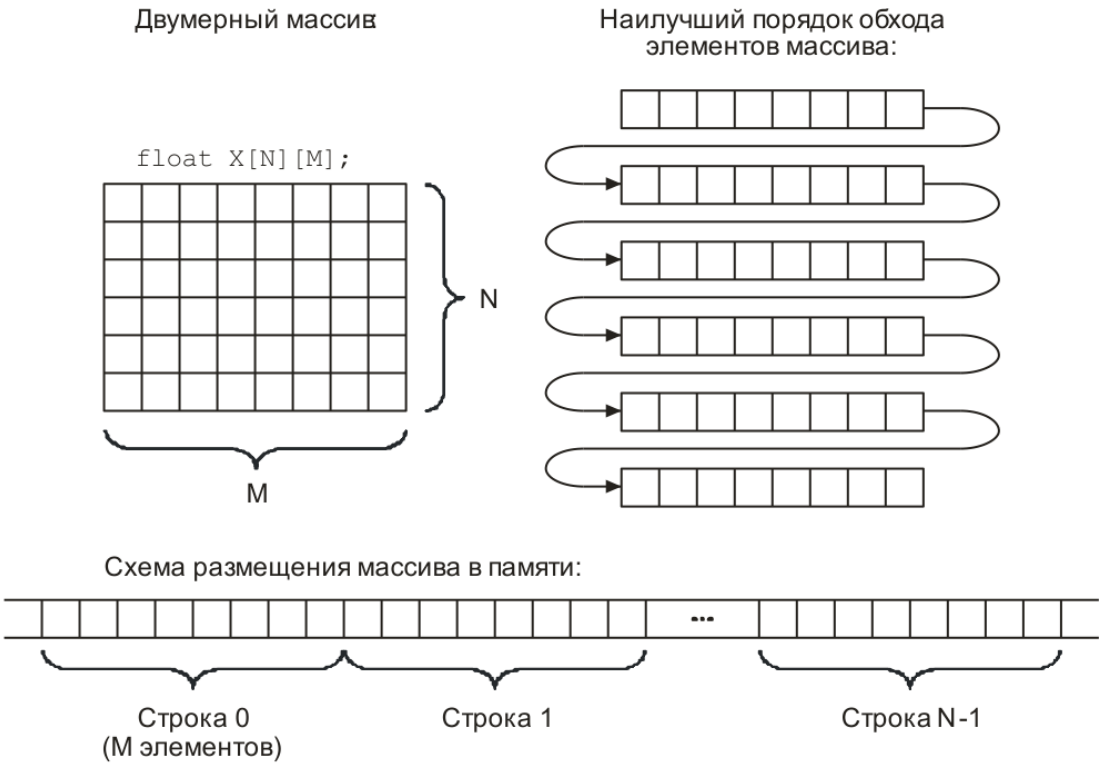

Рис. X_C_2. Расположение в памяти двумерного массива

Листинг. X_B_3. Последовательный обход двумерного массива в языке C/C++

```
1    int X[N][M];
2    for (i=0;i<N;i++)
3      for (j=0;j<M;j++)
```

```
4         X[i][j]=1;
```

Приведем несколько вариантов действий, позволяющие достигнуть последовательного обхода памяти в программе, где таковой нарушается.

1. **Перестановка циклов.** Если алгоритм допускает изменение порядка вложенности циклов, то можно расположить их в том порядке, в котором будет происходить последовательный обход массива (Рис. X_C_2). Это самый простой вариант, обычно требующий меньше всего изменений в исходном коде.
2. **Изменение порядка следования размерностей массива.** Если перестановка циклов невозможна вследствие особенностей алгоритма, то может помочь предварительное переупорядочение элементов массива таким образом, чтобы в циклах вычислений он обходился последовательно. Например, можно поменять порядок следования размерностей многомерного массива.

На Рис. X_C_3 приведено время выполнения программы умножения плотных матриц в зависимости от порядка вложенности циклов. Как видно, при различных порядках обхода данных время может отличаться на порядок.

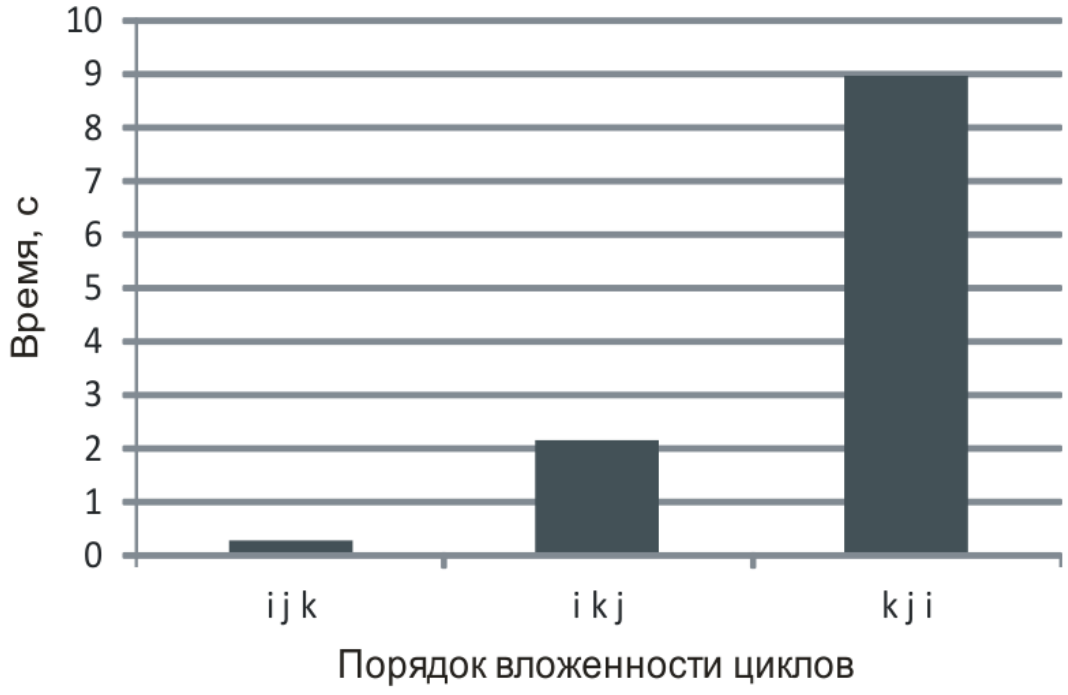

Рис. X_C_3. Влияние порядка вложенности циклов на время выполнения программы умножения матриц

6.3.3. Промахи по конфликту приводят к буксованию кэша и снижению скорости выполнения соответствующего фрагмента программы

практически на порядок. Они могут возникать при обходе массивов данных с шагом кратным размеру банка кэш памяти и количестве элементов, которые нужно обойти, превышающим степень ассоциативности кэша. Например, в процессорах DEC Alpha 21264 и AMD Opteron 840 кэш-память первого уровня – множественно-ассоциативная со степенью ассоциативности равной 2 и размером 64 КБ. Размер банка кэш-памяти равен 64 КБ / 2 = 32 КБ. Значит, обращения к трем или более элементам, отстоящим друг от друга на расстояние, кратное 32 КБ, вызовут эффект буксования кэш-памяти.

На рис. Z_5 показано время работы программы численного моделирования, в которой выполнялся обход трехмерного массива 8-байтовых элементов размером N×N×N. При обращении к каждому элементу во время обхода массива также выполнялись обращения ко всем его соседям по трем измерениям, отстоящим в памяти в обе стороны на расстояния в 1, N и N×N элементов. Видно, что при N=128, происходят три обращения в память на расстоянии N×N×8Б = 128 КБ, что приводит к кэш-буксованию и существенному замедлению работы программы.

Рис. Z_5. Влияние эффекта буксования на время выполнения программы

Далее будут рассмотрены несколько типичных примеров ситуаций, в которых может появляться эффект «буксования» кэш-памяти. Рассмотрим пример из листинга Z_5. Во внутреннем цикле происходит перебор элементов массива a с шагом 4096 элементов, т.е. 4096×8 Б = 32768 Б = 32 КБ, что приводит к буксованию кэш-памяти.

Листинг Z_5. Буксование кэш-памяти при обработке одного массива

```
1   double a[4096000], sum[4096];
2   int i, j;
3   for(i=0; i<4096; i++) {
4     sum[i]=0;
5     for(j=0; j<1000; j++)
6       sum[i] += a[i+j*4096];
7   }
```

Листинг Z_6. Буксование кэш-памяти при обработке нескольких массивов

```
1   double a[4096], b[4096], c[4096];
2   int i;
3   for(i=0; i<4096; i++)
4      c[i]=a[i]+b[i];
```

В примере из листинга Z_6массивы размером 4096 элементов, т.е. 32 КБ, расположены в памяти последовательно друг за другом. Элементы этих массивов с одинаковыми индексами отстоят друг от друга в памяти как раз на размер массива, т.е. на 32 КБ. Значит, в приведенном цикле будут обращения к 3-м элементам, отстоящим на 32 КВ, что приведет к «буксованию» 2-ассоциативной кэш-памяти (Рис. Z_6).

Для того, чтобы избежать эффекта «буксования» кэш-памяти, обычно используются два приёма:

1. изменение порядка обхода элементов;

2. изменение расстояния между элементами.

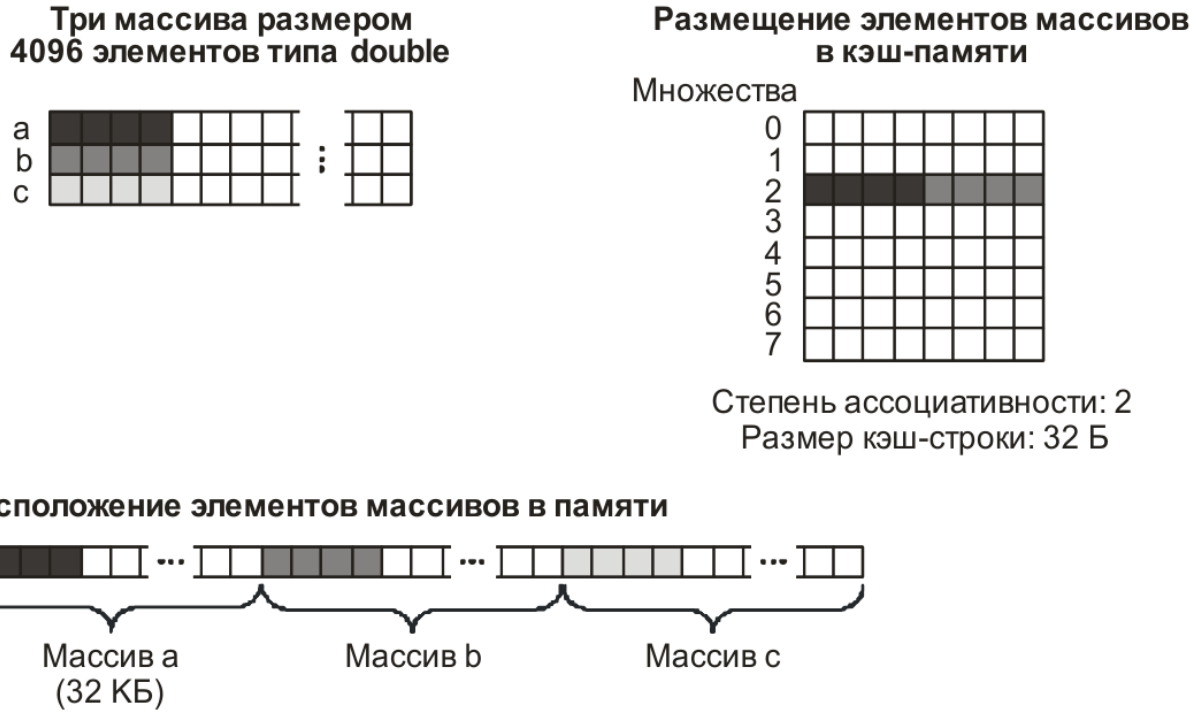

Рис. Z_6. Эффект кэш-буксования при работе с одномерными массивами

В ряде случаев избавиться от «буксования» кэш-памяти можно, изменив порядок обработки элементов массива. Способы изменения обхода массива будут рассмотрены далее.

В некоторых случаях порядок обработки данных изменить нельзя, т.к. он зафиксирован алгоритмом. Тогда единственным вариантом улучшения является изменение размещения данных в оперативной памяти, а именно, расстояния между конфликтующими элементами. Если такие элементы находятся в разных массивах, то можно изменить расстояние между ними, не изменяя сами массивы. Например, изменения расстояния между массивами a, b и c из листинга Z_6 можно добиться так, как показано в листинге Z_7.

Листинг Z_7. Изменение расположения массивов в памяти

```
1   double a[4096], tmp1[8], b[4096], tmp2[8], c[4096];
```

С тем же эффектом можно увеличить размер массива фиктивными (не использующимися) элементами, например, до 4096+8 (Рис. Z_7). Заметим, что для наилучшего результата величина прироста размера массива должна быть не меньше размера кэш-строки. Если добавленные элементы занимают значительный объем памяти, то их можно использовать под хранение других данных задачи.

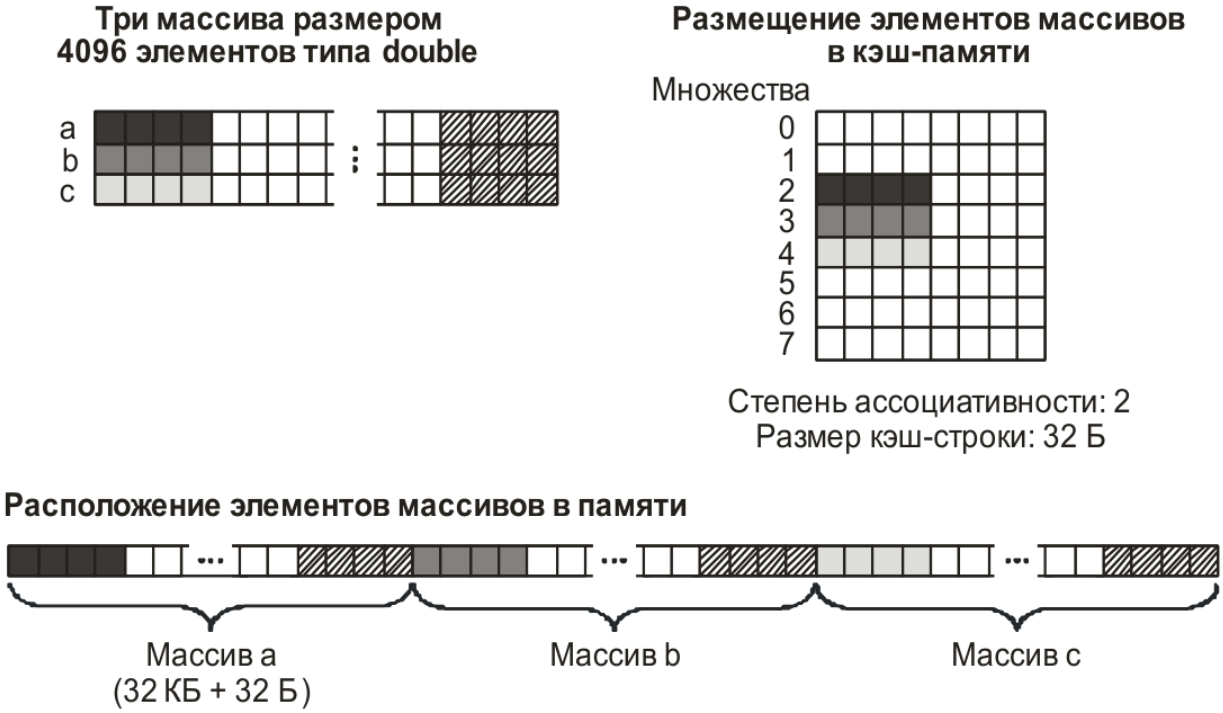

Рис. Z_7. Изменение расположения массивов в памяти для устранения эффекта кэш-буксования

6.4. Использование SIMD расширений.

SIMD параллелизм можно эффективно применять для широкого спектра задач:

- обработка аудиоданных;
- обработка графики;
- локальная фильтрация сигналов и изображений;
- моделирование клеточного автомата;
- трафаретные вычисления и явные схемы;
- операции с векторами и матрицами.

Общее свойство этих задач – однотипная обработка большого числа элементов (отсчетов, пикселей, вокселей, клеток, узлов). Для задач, не обладающих таким свойством, применение SIMD параллелизма вряд ли даст ощутимые преимущества. Примеры этих задач – синтаксический анализ, вычисления с рекурсией, реализация конечных автоматов в управлении и моделировании и т.д.

Современные высокопроизводительные микропроцессоры, включая x86_64 и ARM, имеют векторные расширения, позволяющие обработать комплект из нескольких чисел. Процессоры с архитектурой x86_64, в зависимости от модели, поддерживают наборы векторных команд MMX, SSE, SSE2, SSSE3, AVX, AVX2, AVX512.

MMX появилось первым, и имеет некоторые неудобные ограничения. В частности, MMX позволяет работать только с целыми числами. Команды MMX нельзя использовать вместе с командами математического сопроцессора. Появившееся позже расширение SSE преодолело эти ограничения. Им удобно пользоваться при написании программ для многих вычислительных задач.

Использовать SSE можно несколькими способами:

1) *Подключение стандартных библиотек с векторизованным кодом.* Не требуется самостоятельно программировать с использованием команд SSE, библиотеки обеспечивают высокую эффективность.

Главное ограничение – можно выполнить только ограниченный набор стандартных функций, которые реализованы в той или библиотеке, например, операции над матрицами.

2) Автоматическая векторизация кода компилятором.
3) *Написание векторизованной версии подпрограмм на ассемблере* (в отдельном модуле или в виде ассемблерной вставки с программу на Си/Си++).
4) Использование стандартных встраиваемых функций, называемых *интринсиками*. Интринсики, в отличие от обычных функций, не вызываются, а встраиваются непосредственно в код, где они нужны. Если их размер мал, то это не приводит к существенному увеличению размера кода.

Векторные инструкции SSE можно разделить на следующие основные классы:

- Арифметические (ADDPS, ADDSS, SUBPS, DIVPS, SQRTPS, RSQRTPS, MAXPS, MINPS);
- Логические (ANDPS, XORPS);
- Сравнение (CMPPS, CMPSS);
- Сдвиг;
- Перемещение данных (MOVUPS, MOVAPS, MOVSS);
- Перестановка и распаковка элементов (SHUFPS);
- Преобразование формата

Рассмотрим пример реализации скалярного произведения двух векторов чисел с плавающей запятой одинарной точности сначала в обычном последовательном виде (листинг 6.4.1.), а затем – в векторизованном (листинг 6.4.2.).

```
float inner1(float *x, float* y, int n)
{
  float s = 0;
  for(int i=0; i<n; ++i)
    s += x[i]*y[i];
  return s;
}
```

Листинг 6.4.1. Последовательная реализация скалярного умножения векторов.

```
#include <xmmintrin.h>
float inner2(float* x,
             float* y,
             int n)
{
  __m128 *xx, *yy;
  __m128 p,s;
  xx = (__m128*)x;
  yy = (__m128*)y;
  s = _mm_set_ps1(0);

  for(int i=0; i<n/4; ++i)
  {
    p = _mm_mul_ps(xx[i],yy[i]);
    s = _mm_add_ps(s,p);
  }
  p = _mm_movehl_ps(p,s);
  s = _mm_add_ps(s,p);
  p = _mm_shuffle_ps(s,s,1);
  s = _mm_add_ss(s,p);
  float sum;
  _mm_store_ss(&sum,s);
  return sum;
}
```

Листинг 6.4.2. SIMD реализация скалярного умножения векторов.

Использование ассемблерных вставок более трудоемко. Приведем несколько простейших примеров программ на Си с ассемблерными вставками с SIMD/SSE командами. Сложение серий их восьми целых чисел с использованием MMX показано на листинге 4.6.3. Использована ассемблерная нотация Intel, которая применяется в Windows. Практически аналогичный пример, в котором вместо сложения проводится векторное сравнение – на листинге. 4.6.4. Деление серий из четырех чисел float с использованием SSE – на листинге 4.6.5. Аналогичный пример, но в нотации American Telephones&Telegraph (AT&T) показан на листинге 4.6.6. Все листинго, кроме последного можно собрать в Windows, напрмер, в Microsoft Visual Studio.

Последний листинг можно скомпилировать в Linux с использованием, например, GCC. Рекомендуется скопировать эти листинги, собрать и запустить программы. При делении с float числами следует обратить внимание, что при делении числа на нуль в качестве результата будет напечатано inf или –inf (т.е. плюс бесконечность или минус бесконечность). При неопределенности вида нуль на нуль выводится NaN (Not A Number). Консольный вывод для листинга 4.6.3. показан на рис. 4.6.1.

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

struct VectorB8{
	unsigned char b0;
	unsigned char b1;
	unsigned char b2;
	unsigned char b3;
	unsigned char b4;
	unsigned char b5;
	unsigned char b6;
	unsigned char b7;
};

void MMX_AddB8(struct VectorB8 *Op_A, struct VectorB8 *Op_B){
	__asm
	  {
		MOV EAX, Op_A
		MOV EBX, Op_B

		MOVQ MM0, qword ptr [EAX]
		MOVQ MM1, qword ptr [EBX]

		PADDB MM0, MM1 – векторное сложение

		MOVQ qword ptr [eax], MM0

		   EMMS – переключение обратно в режим сопроцессора 8087
	}
}
```

```
void main(){
	struct VectorB8 a, b;

	  a.b0 = 0;
	  a.b1 = 1;
	  a.b2 = 2;
	  a.b3 = 3;
	  a.b4 = 4;
	  a.b5 = 5;
	  a.b6 = 6;
	  a.b7 = 7;

	  b.b0 = 0;
	  b.b1 = 10;
	  b.b2 = 20;
	  b.b3 = 30;
	  b.b4 = 40;
	  b.b5 = 50;
	  b.b6 = 60;
	  b.b7 = 70;

	  MMX_AddB8(&a, &b);

	  printf("c.b0 = %d, c.b1 = %d, c.b2 = %d, c.b3 = %d "
		"c.b4 = %d, c.b5 = %d, c.b6 = %d, c.b7 = %d\n",
		a.b0, a.b1, a.b2, a.b3,
		a.b4, a.b5, a.b6, a.b7);
}
```

Листинг 4.6.3. Сложение серий из восьми целых чисел с использованием MMX

```
Far Manager, version 3.0 (build 4400) x64
Copyright © 1996-2000 Eugene Roshal, Copyright © 2000-2015 Far Group

C:\{spaces}\{finished}\[lecture_slide.1]\3.simd\simd.1]>test1simd1.001.mmx.paddb.1.exe
c.b0 = 0, c.b1 = 11, c.b2 = 22, c.b3 = 33 c.b4 = 44, c.b5 = 55, c.b6 = 66, c.b7 = 77

C:\{spaces}\{finished}\[lecture_slide.1]\3.simd\simd.1]>
```

Рис. 4.6.1. Сложение серий из восьми целых чисел с использованием MMX – консольный вывод

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

struct VectorB8{
	unsigned char b0;
	unsigned char b1;
	unsigned char b2;
	unsigned char b3;
	unsigned char b4;
	unsigned char b5;
	unsigned char b6;
	unsigned char b7;
};

void MMX_CmpEqB8(struct VectorB8 *Op_A, struct VectorB8 *Op_B){
	__asm
	  {
		MOV EAX, Op_A
		MOV EBX, Op_B

		MOVQ MM0, qword ptr [EAX]
		MOVQ MM1, qword ptr [EBX]

		PCMPEQB MM0, MM1 – векторное сравнение

		MOVQ qword ptr [eax], MM0

		   EMMS– переключение обратно в режим сопроцессора 8087
	}
}

void main(){
	struct VectorB8 a, b;

	  a.b0 = 0;
	  a.b1 = 1;
```

```
    a.b2 = 2;
    a.b3 = 3;
    a.b4 = 4;
    a.b5 = 5;
    a.b6 = 6;
    a.b7 = 7;

    b.b0 = 0;
    b.b1 = 10;
    b.b2 = 20;
    b.b3 = 30;
    b.b4 = 40;
    b.b5 = 50;
    b.b6 = 60;
    b.b7 = 70;

    MMX_CmpEqB8(&a, &b);

    printf("c.b0 = %d, c.b1 = %d, c.b2 = %d, c.b3 = %d "
           "c.b4 = %d, c.b5 = %d, c.b6 = %d, c.b7 = %d\n",
           a.b0, a.b1, a.b2, a.b3,
           a.b4, a.b5, a.b6, a.b7);

    ////////////////////////////////////////////////////

    a.b0 = 0;
    a.b1 = 10;
    a.b2 = 20;
    a.b3 = 30;
    a.b4 = 40;
    a.b5 = 50;
    a.b6 = 60;
    a.b7 = 70;

    b.b0 = 0;
    b.b1 = 1;
    b.b2 = 2;
    b.b3 = 3;
    b.b4 = 4;
```

```
        b.b5 = 5;
        b.b6 = 6;
        b.b7 = 7;

        MMX_CmpEqB8(&a, &b);

        printf("c.b0 = %d, c.b1 = %d, c.b2 = %d, c.b3 = %d "
               "c.b4 = %d, c.b5 = %d, c.b6 = %d, c.b7 = %d\n",
               a.b0, a.b1, a.b2, a.b3,
               a.b4, a.b5, a.b6, a.b7);

}
```

Листинг 4.6.4. Сравнение серий из восьми целых чисел с использованием MMX

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

struct Vector4{
      float x, y, z, w;
};

Vector4 SSE_Divps( const Vector4 &Op_A, const Vector4 &Op_B){
      Vector4 Ret_Vector;

      __asm
        {
            MOV EAX, Op_A
            MOV EBX, Op_B

            MOVUPS XMM0, [EAX]
            MOVUPS XMM1, [EBX]

            DIVPS XMM0, XMM1 – векторное деление четырех чисел float
            MOVUPS [Ret_Vector], XMM0
      }

      return Ret_Vector;
```

```
}

void main(){
	Vector4 a, b, c;

	  a.x = 1;
	  a.y = 2;
	  a.z = 3;
	  a.w = 0;

	  b.x = 10;
	  b.y = 0;
	  b.z = 30;
	  b.w = 0;

	  printf("a.x = %f, a.y = %f, a.z = %f, a.w = %f\n",
		a.x, a.y, a.z, a.w);
	  printf("b.x = %f, b.y = %f, b.z = %f, b.w = %f\n",
		b.x, b.y, b.z, b.w);

	  c = SSE_Divps(a, b);

	  printf("c.x = %f, c.y = %f, c.z = %f, c.w = %f\n",
		c.x, c.y, c.z, c.w);
}
```

Листинг 4.6.5. Деление серий из четырех чисел float с использованием SSE в нотации Intel

```
C:\{spaces}\{finished}\[lecture_slide.1]\3.simd\simd.1]>test1simd1.002.mmx.pcmpeqb.1.exe
c.b0 = 255, c.b1 = 0, c.b2 = 0, c.b3 = 0 c.b4 = 0, c.b5 = 0, c.b6 = 0, c.b7 = 0
c.b0 = 255, c.b1 = 0, c.b2 = 0, c.b3 = 0 c.b4 = 0, c.b5 = 0, c.b6 = 0, c.b7 = 0

C:\{spaces}\{finished}\[lecture_slide.1]\3.simd\simd.1]>test1simd1.100.sse.div.1.exe
a.x = 1.000000, a.y = 2.000000, a.z = 3.000000, a.w = 0.000000
b.x = 10.000000, b.y = 0.000000, b.z = 30.000000, b.w = 0.000000
c.x = 0.100000, c.y = 1.#INF00, c.z = 0.100000, c.w = -1.#IND00

C:\{spaces}\{finished}\[lecture_slide.1]\3.simd\simd.1]>
```

Рис. 4.6.1. Сравнение серий из восьми целых чисел с использованием MMX и деление с использованием SSE – консольный вывод

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

typedef struct{
	float x, y, z, w;
} Vector4;

void SSE_Div(Vector4 *res, Vector4 *a, Vector4 *b){
	asm volatile ("mov %0, %%eax"::"m"(a));
	asm volatile ("movups (%eax), %xmm0");
	asm volatile ("mov %0, %%ebx"::"m"(b));
	asm volatile ("movups (%ebx), %xmm1");
	asm volatile ("divps %xmm1, %xmm0");
	asm volatile ("mov %0, %%eax"::"m"(res));
	asm volatile ("movups %xmm0, (%eax)");
}

int main(void){
	Vector4 a, b, r;

	  a.x = 1.;
	  a.y = 2.;
	  a.z = 3.;
	  a.w = 0.;

	  b.x = 10.;
	  b.y = 0.;
	  b.z = 30.;
	  b.w = 0.;

	  SSE_Div(&r, &a, &b);

	  printf("r.x = %f, r.y = %f, r.z = %f, r.w = %f\n",
		r.x, r.y, r.z, r.w);
	  return 0;
}
```

Листинг 4.6.6. Деление серий из четырех чисел float с использованием SSE в нотации AT&T

Конечно, все приведенные примеры с ассемблерными вставками не обладают переносимостью на разные платформы. Их можно запускать только на x86. Скомпилировать последний пример, например, под Raspberry Pi/Linux Raspbian не получится. На ARM другой набор команд процессора.

6.5. Использование многопоточности.

Как ранее отмечалось, практически все современные микропроцессоры – многоядерные. Для раскрытия их возможностей счетное приложение должно использовать параллелизм на уровне потоков или процессов. Для вычислительного приложения предпочтительнее использовать многопоточность. Первая причина такого выбора – наличие общего виртуального адресного пространства у потоков, что дает возможность работать с одними и теми же данными без их копирования или пересылок. Сравнение пропускной способности при передаче данных в одном узле между MPI процессами и при чтении памяти из потока внутри процесса приведено в табл. 6.5.1. Видно, что пропускная способность при использовании потоков выше примерно на порядок. Вторая причина – меньшие примерно на порядок временные расходы по созданию потоков по сравнению с созданием процессов (см. табл. 6.5.2).

Табл. 6.5.1. Пропускная способность при передаче данными между процессами в одном узле и при доступе к данным из потока.

(источник - https://computing.llnl.gov/tutorials/pthreads/)

Platform	MPI Shared Memory Bandwidth (GB/sec)	Pthreads Worst Case Memory-to-CPU Bandwidth (GB/sec)
Intel 2.6 GHz Xeon E5-2670	4.5	51.2
Intel 2.8 GHz Xeon 5660	5.6	32
AMD 2.3 GHz Opteron	1.8	5.3
AMD 2.4 GHz Opteron	1.2	5.3
IBM 1.9 GHz POWER5 p5-575	4.1	16
IBM 1.5 GHz POWER4	2.1	4
Intel 2.4 GHz Xeon	0.3	4.3
Intel 1.4 GHz Itanium 2	1.8	6.4

Табл. 6.5.2. Время создания 50000 процессов и потоков на компьютерах с различной архитектурой

(источник - https://computing.llnl.gov/tutorials/pthreads/)

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
Intel 2.6 GHz Xeon E5-2670 (16 cores/node)	8.1	0.1	2.9	0.9	0.2	0.3
Intel 2.8 GHz Xeon 5660 (12 cores/node)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16 cores/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8 cores/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8 cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8 cpus/ node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8 cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

Переход к многопоточности в программе позволяет:

1) задействать все доступные ядра процессора при интенсивных вычислениях,

2) устранить ожидания завершения операций ввода/вывода (переключением на другие потоки программы, которые готовы исполняться),

3) обеспечивать высокий приоритет исполнения для частей программы, от которых требуется работа в реальном времени,

4) реализовывать обработку асинхронно возникающих событий в отдельных, выделенных для этого потоках,

5) в приложениях с пользовательским интерфейсом оградить пользователя от ожидания завершения выполнения длительных операций (это операции выполняются в одном потоке, а код пользовательского интерфейса – в другом).

Выделять в отдельные потоки можно, например, процедуры, которые не зависят друг от друга. Взаимодействие между потоками в программе можно организовывать по-разному. Наиболее распространены следующие модели:

1) **Менеджер/работник (manager/worker).** Имеется один поток-менеджер и несколько потоков работников. Менеджер распределяет задания между работниками, а работники их обсчитывают.

2) **Конвейер (pipeline).** Пусть программа обрабатывает поток поступающих запросов. Обработку каждого запроса можно разбить на этапы. Различные этапы выполняются в отдельных потоках.

3) **Равнозначные потоки (peer).** Модель аналогична модели менеджер/работник, но в ней после того, как главный поток распределил работу между потоками, на сам выполняет некоторую часть этой работы.

Рассмотрим основные средства по работе с потоками и приведем для них примеры простых программ. Основные современные ОС, включая GNU Linux, UNIX и Windows, предоставляют функции по работе с потоками. GNU Linux и большинство UNIX реализуют унифицированный интерфейс по работе с потоками - POSIX Threads, что обеспечивает высокую переносимость многопоточных приложений. До появления POSIX Threads разные платформы имели несовместимые между собой средства по работе с потоками, и перенос многопоточных приложений был трудоемким.

ANSI/IEEE POSIX 1003.1 (или сокращенно POSIX Threads) описывает типы данных и функциональный интерфейс по управлению потоками и синхронизации между ними.

Пример программы с использованием POSIX Threads приведен на листинге 6.5.2. В ней в функции main внутри цикла создается и запускается пять потоков. Каждый из них печатает сообщение с указанием переданного номера. У каждого потока свой стек вызовов. Главный поток начинается выполняться с функции main (если не учитывать код стандартной библиотеки). Все остальные потоки выполняют функцию PrintHello, которая вызывает другие функции: printf для печати и pthread_exit для завершения работы потока. Вызов pthread_exit – это завершение функции main, которое отличается от обычного в однопоточной программе путем естественного выхода из функции, вызова оператора return или функции завершения работы процесса, типа _exit. Причина отличия – в том, что другие созданные потоки могут еще выполняться на момент завершения выполнения функции в main в главном потоке. Если при этом вызвать завершение процесса, то исполнение этих других потоков может прекратиться преждевременно, что является ошибкой.

Листинг 6.5.2. Многопоточная программа, использующая POSIX Threads

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

#define NUM_THREADS     5

void *PrintHello(void *threadid){
  long tid;  tid = (long)threadid;

  printf("Hello World! It's me, thread #%ld!\n", tid);
  pthread_exit(NULL);
}

int main (int argc, char *argv[]){
  pthread_t threads[NUM_THREADS];
  int rc; long t;

  for(t=0; t<NUM_THREADS; t++){
    printf("In main: creating thread %ld\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);

    if (rc){
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }
  pthread_exit(NULL);
}
```

Для написания многопоточной программы на POSIX Threads или Windows Threads требуется нетривиальная модификация последовательной программы. Для упрощения построения многопоточных реализаций счетных алгоритмов был предложен стандарт *OpenMP*. Стандарт поддержан во многих современных компиляторах C и Fortran, напрмер, в Visual C, GCC, Intel C. Стандарт описывает набор прагм и функций для распараллеливания программы на несколько потоков

Рассмотрим пример многопоточной программы скалярного произведения двух векторов (листинг 6.5.4.). Последовательная версия

приведена на листинге 6.5.3. Прагма *#pragma omp parallel for* предписывает распараллелить цикл по нескольким потокам. Каждому потоку назначаются на выполнение различные итерации цикла. Можно распараллелить только цикл for (цикл while – нельзя). При этом итерации цикла должны быть независимы друг от друга, их число должно быть известно на момент начала исполнения цикла, приращение индексной переменной может быть только на константу (как положительную, так и отрицательную). Прагма reduction означает редукцию переменной – т.е. сборку всех частично посчитанных значений в разных потоках. Редукция возможна для таких операций, как сложение, умножение, исключающее или.

```
float inner1(float *x, float* y, int n)
{
  float s = 0;
#pragma omp parallel for reduction(+:s)
  for(int i=0; i<n; ++i)
    s += x[i]*y[i];
  return s;
}
```

Листинг 6.5.3. Последовательная версия скалярного произведения двух векторов

```
float inner1(float *x, float* y, int n)
{
  float s = 0;
#pragma omp parallel for reduction(+:s)
  for(int i=0; i<n; ++i)
    s += x[i]*y[i];
  return s;
}
```

Листинг 6.5.4. OpenMP версия скалярного произведения двух векторов

На листинге 6.5.5. приведен пример многопоточной реализации программы, суммирующей поэлементно два вектора. Прагма *shared* указывает переменные, общие для всех потоков. Прагма *private* указывает такие переменные, для которых надо создать отдельный собственный экземпляр в каждом потоке. Прагма *nowait* означает, что если какой-то поток закончил выполнение своей параллельной секции, он может перейти к следующей. Так как в этой программе только одна параллельная секция, эта прагма особого

смысла здесь не имеет. Прагма *schedule* рассматривается далее в разделе о динамической балансировке.

```
#include <omp.h>
void main ()
{       int i;
        float a[N], b[N], c[N];
        for (i=0; i < N; i++)
                a[i] = b[i] = 1.0;

  #pragma omp parallel shared(a,b,c) private(i)
  {
     #pragma omp for schedule(dynamic, 100) nowait
     for (i=0; i < N; i++)
       c[i] = a[i] + b[i];
   }
}
```

Листинг 6.5.5. OpenMP версия поэлементного суммирования двух векторов

На листинге 6.5.6. приведен пример простейшей синхронизации потоков при доступе к общей переменной. Если убрать синхронизацию, при исполнении результат будет неверным. Он всегда будет меньше 1000000. Синронизация здесь производится с помощью прагмы critical, которая описывает критическую секцию для строки x=x+1. Если один поток вошел в выполнение критической секции, другой поток, дойдя до нее, ожидает, пока первый не выйдет из нее.

```
#include <omp.h>
void main()
{   int x;
        x = 0;
        #pragma omp parallel for shared(x)
        for(int I = 0; I < 1000000; i++){
                #pragma omp critical
                x = x + 1;
        }
        printf("x=%d", x)
}
```

Листинг 6.5.6. Простейший пример синхронизации потоков в OpenMP

При построении многопоточного приложения следует аккуратно пользоваться системной приоритетов. Во многих случаях достаточно оставить приоритеты по умолчанию. Если есть некоторые обработчики событий, которые должны срабатывать как можно быстрее, но время работы которых мало, то потоку для их исполнения можно повысить приоритет. При этом после выполнения этих обработчиков поток следует приостановить до следующего срочного запроса. Процедуры с длительными вычислениями никогда не следует запускать в потоках с высоким приоритетом. Если эти вычисления не сопряжены с непосредственным взаимодействием с пользовательским интерфейсом программы, то для потока, наоборот, можно снизить приоритет до фонового уровня.

Выбор числа потоков. Хотя (как видно из табл. 6.5.2) накладные расходы на создание потоков существенно меньше, чем для процессов, они все равно присутствуют. Поэтому, не следует создавать избыточное число потоков. Их оптимальное число определяется несколькими факторами. Перечислим главные из них. Для счетного приложения (алгоритм которого допускает эффективное распараллеливание на общей памяти, т.е., в частности, на потоках) число потоков можно выбрать равным числу ядер или удвоенному числу ядер (если в ядрах есть мультитрединг). Если часто вызываются длительные операции (например, ввод из медленной памяти), то для того, чтобы загрузить ядра, число потоков можно увеличить. Пока одни потоки ожидают завершение операции, ядром пользуются другие потоки. Для приложений с функциональным параллелизмом удобно разбить разные параллельно выполняемые функции по потокам. В этом случае число стадий (наряду с предыдущим фактором) влияет на число создаваемых в приложении потоков. Если в приложении реализуется графический пользовательской интерфейс, обработку пользовательского ввода и отрисовку в окнах также лучше вынести в отдельный поток. Это повышает комфортность использования интерфейса, так как исключаются задержки при выполнении длительных операций. Они могут восприниматься пользователем как кратковременные “подвисания” программы.

Взаимодействие между процессами. В некоторых случаях разрабатываемая система выполняется как набор отдельных процессов, а не потоков отдельного процесса. Причины этого могут быть различными – необходимость разных прав доступа для разных частей системы, несовместимые (в рамках одного процесса) модули, которые сложно модифицировать и т.д. В этом случае, если есть возможность выбора средств

взаимодействия между процессами, с точки зрения производительности целесообразно воспользоваться средствами межпроцессного взаимодействия (Inter-Process Communications, IPC: общими окнами в памяти, очередью сообщений и т.д.), а не средствами сетевого взаимодействия (socket library и библиотеки прикладных протоколов на ее основе). Сетевое взаимодействие имеет больше накладных расходов, и скорость передачи данных при нем будет заметно ниже.

Но и у IPC есть свои накладные расходы. Поэтому, при большом числе взаимодействий между процессами, можно применить следующий прием. Больше число передаваемых фрагментов данных (в текущий момент времени) можно собрать в один пакет и передать его за один вызов системной функции. Эта рекомендация подходит и для файлового ввода/вывода, и для сетевой передачи данных. При разработке параллельной версии системы моделирования WinALT в 2000 г. автор на своем опыте убедился в важности этой рекомендации, т.к. первая построенная параллельная версия системы работала во много (!) раз медленнее обычной последовательной.

Динамическая балансировка. Часто фрагменты вычислений в реализуемой задаче имеют непредсказуемое время вычисления. Если их статически (заранее, до вычисления) распределить по потокам, то часть потоков закончит свою часть вычислений быстрее и будет ждать остальных. Часть ядер компьютера будет простаивать, реальная производительность для такой реализации снизится. Иногда даже при известном заранее времени счета всех фрагментов влияние других задач, выполняемых в системе может приводить к заметной разбалансировке. Для преодоления такой ситуации можно применить динамическую балансировку нагрузки. При таком подходе все фрагменты вычислений распределяются не сразу (до вычислений) по потокам, а во время счета периодически, когда у потока завершаются ранее назначенные ему фрагменты вычислений. При использовании POSIX Threads программист сам должен формировать списки фрагментов и реализовывать передачу фрагментов потокам для счета. В OpenMP при использовании распараллеливания в цикле for можно задать динамическое распределение итераций циклов по потокам с помощью прагм:

```
#pragma omp parallel for schedule(dynamic)
```

или

```
#pragma omp parallel for schedule(guided)
```

Размер фрагмента вычислений (в данном случае – это число выделяемых за один раз итераций) можно задавать как параметр прагмы:

#pragma omp parallel for schedule(dynamic, **100**)

Для dynamic размер блока фиксирован, для guided он постепенно уменьшается к концу вычислений.

Тема 7. Специфика оптимизации ПО в основных проблемных областях.

Повышение точности вычислений. Для многих вычислительных методов точности, даваемой современными архитектурами с длинной слова 64 бита недостаточно. В таком случае используется программная эмуляция арифметики с плавающей запятой для чисел повышенной точности (например double-double) или произвольной точности. Пример свободно распространяемой библиотеки для вычислений с произвольной точностью - GNU Arbitrary Precision Library.

Экономия памяти. Экономия памяти важна как сама по себе, чтобы дать возможность обсчитывать объекты данных как можно юлльших размеров, так и с точки зрения уменьшения времени счета. Пропускная способность шины, соединяющей процессор и оперативную память ограничена, и загрузка объекта данных меньшего размера позволяет уменьшать время выполнения программы. Перечисли некоторые из основных способов экономии памяти:

1) Использование типов данных с меньшим размером. Если не требуется высокая точность при расчетах с плавающей запятой, можно перейти от использования данных с двойной точностью (в C – double, 64 бита) к одинарной (в C – float, 32 бита) или половинной (16 бит, __fp16, _Float16) на тех платформах, где она поддерживается, например, ARM (https://gcc.gnu.org/onlinedocs/gcc/Half-Precision.html). Аналогично, для целых чисел можно перейти от 32-битных целых (unsigned int, int, uint32_t) к 16-

битным (unsigned short, short, uint16_t) или 8-битным (unsigned char, char, uint8_t).

2) Использование битовых полей для работы с элементами данных с небольшим числом состояний (128 и менее). В табл. 7.1. приведены два описания структуры из двух булевых полей. Одна (слева) использует слова, другая (справа) использует битовые поля. Первая на 32-битных платформах занимает 8 байт, вторая – только один.

Табл. 7.1. Примеры определения структуры с полями из машинных слов и с битовыми полями.

`struct {`
`unsigned int fieldBoolean1;`
`unsigned int fieldBoolean2;`
`} status;`
	`struct {`
	`unsigned int fieldBoolean1 : 1;`
	`unsigned int fieldBoolean2 : 1;`
	`} status;`

Если требуется закодировать более двух, но не более четырех состояний, то необходимо выделить под поле два бита. Если не более восьми, то три бита и т.д.

3) Использование не сплошных представлений для матриц (и вообще для массивов), которые можно отнести к разреженным (т.е. к таким, большинство элементов в которых имеют одинаковое значение, обычно нуль). В структуре данных сохраняются только значения элементов, отличных от нуля. Обычно это список (LIL - list of lists, COO – coordinate list), массив списков с парами координат и значений или структура данных для ассоциативного поиска, например, дерево или хэш-таблица (DOK, dictionary of keys). Также применяются форматы со сжатым представлением строки (CSR, CRS, Yale format)

Экономия вычислений. В случае, когда в вычислениях требуется подсчитывать значение функции с большим объемом вычислений и большое число раз, применяют такие техники, как табличное представление и мемоизацию. При табличном задании заранее вычисляются значения функции в требуемом диапазоне и с заданным шагом. Если требуется вычислить значение функции для аргумента между шагами, можно применить интерполяцию. Мемоизация – похожий механизм. Он отличается тем, что таблица знечений функции формируется не до вычислений, а в их процессе.

Если для некоторого аргумента вычисление функции производится впервые, то происходит реальное вычисление значения, оно размещается в таблице и возвращается в виде результата. При последующем запросе вычисления с этим же значением аргумента вычисление исключается, готовое сохраненное значение берется из таблицы. Табличные вычисления и мемоизация подходят только для тех вариантов, когда множество аргументов (требуемое для задачи) не очень велико. Мемоизация работает наиболее эффективно, когда наблюдается локальность по значениям аргумента.

Параллельные вычисления. Если объем вычислений велик, то могут возникать следующие две проблемы. Во-первых, время счета может оказаться неприемлемо большим. Например, вычисление микропогодного прогноза в аэропорте на следующий час лишено всякого смысла, если время отсчета приближается или тем более превышает час. Во-вторых, обем данных для обсчета может оказаться слишком большим для памяти одного компьютера. В подобных ситуациях применяют параллельные вычислительные системы с распределенной памятью (MPP, вычислительные кластеры). Программировать их можно как на низком уровне, создавая сетевые распределенные приложения с собственными протоколами передачи данных и координации работы частей распределенного приложения с использованием TCP или UDP, так и на высоком уровне, с использованием технологий MPI, PVM, DVM, координационных языков Linda, Jada, Pirhana, систем виртуальной общей памяти (OpenSHMEM). Для прикладных счетных программ рекомендуется использование MPI. Основная модель вычислений, применяемая для параллельных вычислительных систем с распределенной памятью, - модель с посылкой сообщений. На ней основан MPI.

Встраиваемые системы. Встраиваемые системы, в отличие от обычных компьютеров, являются частью более большой системы (механической или электрической), которой они управляют. Набор требований (и в частности требования по эффективности) отличаются большим разнообразием, так как системы, которыми надо управлять очень различаются по назначению и стоимости. В одних случаях нужна минимальная цена, в других случаях – надежность и восстановление после ошибок, в-третьих – минимальное время реакции на события.

Требование для систем с минимальной ценой означает, что желательно обойтись максимально простым микроконтроллером. Это значит, что управляющая программа должна иметь минимальный размер. Самые дешевые микроконтроллеры имеют объем программной памяти в диапазоне 2КБ-32Кб.

Для многих автономных систем требуется построение приложений с минимальным потреблением питания. Конечно, это в наибольшей степени проблема аппаратуры, а не программного обеспечения. Но и в программном обеспечении есть приемы повышения энергоэффективности.

В случаях, когда от встраиваемых систем помимо энергоэффективности требуют и высокую производительность, возможностей даже самых высокопроизводительных микроконтроллеров не хватает, применяются ПЛИС или заказные СБИС.

Список литературы

[Касьянов] Касьянов В.Н., Поттосин И.В. Методы построения трансляторов. - Новосибирск. - Наука. – 1986

[Lau] Lau, Edmond. – The Effective Engineer: How to Leverage Your Efforts In Software Engineering to Make a Disproportionate and Meaningful Impact. – The Effective Bookshelf. – Palo Alto, CA. – 2015

[Tarjan] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. ACM Transactions on Programming Languages and Systems, 1(1):115–120, July 1979.

Ссылки в сети

[Carvalho] Carlos Carvalho. The Gap between Process and Memory Speeds. – ICCA’02. - https://pdfs.semanticscholar.org/6ebe/c8701893a6770eb0e19a0d4a732852c86256.pdf

Electric Fence. - https://elinux.org/Electric_Fence

[RTC] Real Time Clock - https://wiki.osdev.org/RTC

[tsc] https://gist.github.com/aprell/2869011

[timers] https://www.softwariness.com/articles/monotonic-clocks-windows-and-posix/

[GCCopti] https://gcc.gnu.org/onlinedocs/gcc-4.9.3/gcc/Optimize-Options.html. – GCC Online Docs. Options That Control Optimization.

[Cooper] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. A Simple, Fast Dominance Algorithm. - http://www.hipersoft.rice.edu/grads/publications/dom14.pdf

[quora_gcc_opti_levels] https://www.quora.com/When-should-you-use-the-different-GCC-optimization-flags-e-g-O2-I-saw-that-OpenCV-uses-the-O2-optimization-instead-of-O3-on-32-bit-Linux-What-are-the-pros-cons-of-each-optimization-level. – Quora. - When should you use the different GCC optimization flags (e.g. -O2)? I saw that OpenCV uses the "-O2" optimization instead of "-O3" on 32-bit Linux. What are the pros/cons of each optimization level?