Федеральное агентство по образованию
Нижегородский государственный университет им. Н.И. Лобачевского
Национальный проект «Образование»
Инновационная образовательная программа ННГУ. Образовательно-научный центр
“Информационно-телекоммуникационные системы: физические основы
и математическое обеспечение”

А.В. Кудин, А.В. Линёв

Архитектура и операционные системы параллельных вычислительных систем

Учебно-методические материалы по программе повышения квалификации «Информационные технологии и компьютерное моделирование в прикладной математике»

Нижний Новгород
2007

Учебно-методические материалы подготовлены в рамках

инновационной образовательной программы ННГУ:

Образовательно-научный центр "Информационно-телекоммуникационные системы: физические основы и математическое обеспечение"

Кудин А.В., Линёв А.В. «Архитектура и операционные системы параллельных вычислительных систем». Учебно-методические материалы по программе повышения квалификации «Технологии высокопроизводительных вычислений для обеспечения учебного процесса и научных исследований». Нижний Новгород, 2007, 73 с.

Учебно-методические материалы содержат краткую информацию о современных архитектурах параллельных вычислительных систем и направлении их развития, о проблеме синхронизации выполнения потоков при разработке параллельных приложений, а также описание стандартных механизмов синхронизации и решения типовых задач.

СОДЕРЖАНИЕ

Архитектура параллельных вычислительных систем

1. Параллелизм в работе ЭВМ

Параллелизм – основа высокопроизводительной работы всех подсистем вычислительных машин. Организация памяти любого уровня иерархии, организация системного ввода/вывода, организация мультиплексирования шин и т.д. базируются на принципах параллельной обработки запросов. Современные операционные системы являются многозадачными и многопользовательскими, имитируя параллельное исполнение программ посредством механизма прерываний.

Развитие процессоростроения также ориентировано на распараллеливание операций, т.е. на выполнение процессором большего числа операций за такт. Ключевыми ступенями развития архитектуры процессоров стали гиперконвейеризация, суперскалярность, неупорядоченная модель обработки, векторное процессирование (технология SIMD), архитектура VLIW. Все ступени были ориентированы на повышение степени параллелизма исполнения.

В настоящее время мощные сервера представляют собой мультипроцессорные системы, а в процессорах активно используется параллелизм уровня потоков. Основное внимание нашего семинара будет уделено многопоточным и многоядерным процессорам.

1.1. Уровни параллелизма

Распараллеливание операций – перспективный путь повышения производительности вычислений. Согласно закону Мура число транзисторов экспоненциально растёт, что позволяет в настоящее время включать в состав CPU большое количество исполнительных устройств самого разного назначения. Прошли времена, когда функционирование ЭВМ подчинялось принципам фон Неймана.

В 70-е годы стал активно применяться принцип конвейеризации вычислений. Сейчас конвейер Intel Pentium 4 состоит из 20 ступеней. Такое распараллеливание на микроуровне – первый шаг на пути эволюции процессоров. На принципах конвейеризации базируются и внешние устройства. Например, динамическая память (организация чередования банков) или внешняя память (организация RAID).

Но число транзисторов на чипе росло. Использование микроуровневого параллелизма позволяло лишь уменьшать CPI (Cycles Per Instruction), так как миллионы

транзисторов при выполнении одиночной инструкции простаивали. На следующем этапе эволюции в 80-е годы стали использовать параллелизм уровня команд посредством размещения в CPU сразу нескольких конвейеров. Такие суперскалярные CPU позволяли достигать CPI<1. Параллелизм уровня инструкций (ILP) породил неупорядоченную модель обработки, динамическое планирование, станции резервации и т.д. От CPI перешли к IPC (Instructions Per Clock). Но ILP ограничен алгоритмом исполняемой программы. Кроме того, при увеличении количества ALU сложность оборудования экспоненциально растёт, увеличивается количество горизонтальных и вертикальных потерь в слотах выдачи. Параллелизм уровня инструкций исчерпал свои резервы, а тенденции Мура позволили процессоростроителям осваивать более высокие уровни параллелизма. Современные методики повышения ILP основаны на использовании процессоров класса SIMD. Это векторное процессирование, матричные процессоры, архитектура VLIW.

Параллелизм уровня потоков и уровня заданий применяется в процессорах класса MIMD. Многопоточные процессоры позволяют снижать вертикальные потери в слотах выдачи, а Simultaneous Multithreading (SMT) процессоры – как вертикальные, так и горизонтальные потери. Закон Мура обусловил также выпуск многоядерных процессоров (CMP). Лучшие современные вычислители – это мультикомпьютерные мультипроцессорные системы.

Параллелизм всех уровней свойственен не только процессорам общего назначения (GPP), но и процессорам специального назначения (ASP (Application-Specific Processor), DSP (Digital Signal Processor)).

Рис. 1 Уровни параллелизма

Иногда классифицируют параллелизм по степени гранулярности как отношение объёма вычислений к объёму коммуникаций. Различают мелкозернистый, среднезернистый и крупнозернистый параллелизм. Мелкозернистый параллелизм обеспечивает сам CPU, но компилятор может и должен ему помочь для обеспечения большего IPC. Среднезернистый параллелизм – прерогатива программиста, которому необходимо разрабатывать многопоточные алгоритмы. Здесь роль компилятора заключается в выборе оптимальной последовательности инструкций (с большим IPC) посредством различных методик (например, символическое разворачивание циклов). Крупнозернистый параллелизм обеспечивает ОС.

1.2. Метрики параллелизма

Допустим, имеется в наличии вычислительная система с неограниченными вычислительными ресурсами (в которой количество процессоров неограниченно). Как поведут себя программы в этой системе? Ускорятся ли они в неограниченное количество раз? Конечно, нет! Скорость работы простейшей однопоточной программы не изменится, так как она не использует параллелизм высокого уровня. Её вычислительная нагрузка ляжет исключительно на один из процессоров. В отличие от микроуровневого параллелизма, который обеспечивается самим процессором, или в отличие от параллелизма уровня инструкций, где участия программиста не требуется (за исключением SIMD), привлечение к вычислениям нескольких процессоров требует существенной модификации алгоритма. Фактически требование дальнейшего повышения производительности вычислений вынуждает применять новые методики программирования, такие как векторное процессирование (например, SSE), многопоточное программирование (например, с использованием технологии OpenMP) и т.д. Отметим, что классические языки программирования высокого уровня (такие, как C++) ориентированы исключительно на класс SISD.

Итак, программист изменил алгоритм и создал многопоточный код (например, при помощи потоков POSIX). Приведёт ли это к неограниченному росту производительности? Введём в рассмотрение степень параллелизма программы – D(t) – число процессоров, участвующих в исполнении программы в момент времени t. При старте программы D(0)=1. Далее программа может создавать независимые потоки и передавать им часть своей нагрузки. Изучив поведение алгоритма на неограниченной машине, мы получим график D(t) – профиль параллелизма программы. Однако степень параллелизма зависит не только от используемого алгоритма программы, но и от

эффективности компиляции и доступных ресурсов при исполнении. Реальное значение D(t) будет иным.

Введём в рассмотрение T(n) – время исполнения программы на n процессорах. T(n)<T(1), если параллельная версия алгоритма эффективна. T(n)>T(1), если накладные расходы (издержки) реализации параллельной версии алгоритма чрезмерно велики. Заметим, что за T(1) взято не время выполнения многопоточной программы на одном процессоре, а время выполнения однопоточной программы.

Тогда ускорение за счёт параллельного выполнения составит S(n) = T(1) / T(n).

С экономической точки зрения интерес представляет ускорение в пересчёте на один процессор – эффективность системы из n процессоров E(n) = S(n) / n.

В идеальном случае ускорение линейно от n. Такой алгоритм обладает свойством масштабируемости (возможностью ускорения вычислений пропорционально числу процессоров). Но на практике масштабируемый алгоритм имеет несколько худшие показатели из-за накладных расходов на поддержку многопроцессорных вычислений. Кроме того, параметр n масштабируемого алгоритма должен быть согласован с количеством процессоров в вычислительной системе. Необходимо также учитывать и текущую загруженность процессоров другими задачами.

Особое внимание заслуживает случай S(n) > n. Изначально может показаться, что такого не может быть. Однако примеры такого успешного распараллеливания есть! Вопрос лишь в том, как квалифицированно разбить исходную задачу на подзадачи. Требования исходной задачи могут превосходить возможности эксплуатируемого процессора по любому из его ресурсов (чаще всего это кеши различных уровней, буфер BTB, буфер TLB). После разбиения на один процессор попадает задача меньшего объёма, и, соответственно, требования к объёму ресурсов процессора сокращаются. При преодолении таких порогов может возникать суперлинейное ускорение. Внимание: организация кода/данных должна обеспечивать максимальную степень локальности кода/данных (иначе ни линейного, ни тем более суперлинейного ускорения достичь не удастся).

1.3. Закон Амдала

Любая параллельная программа содержит последовательную часть. Это ввод/вывод, менеджмент потоков, точки синхронизации и т.п. Обозначим долю последовательной части за f. Тогда доля параллельной части будет 1-f. В 1967 году

Амдал рассмотрел ускорение такой программы на n процессорах, исходя из предположения линейного ускорения параллельной части:

$$S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{f \times T(1) + \frac{(1-f) \times T(1)}{n}} = \frac{n}{1 + (n-1) \times f}$$

При неограниченном числе процессоров ускорение составит всего лишь 1/f.

$$\lim_{n \to \infty} S(n) = \frac{1}{f}$$

Так, например, если доля последовательной части 20%, то теоретически невозможно получить ускорение вычислений более чем в пять раз! Таким образом, превалирующую роль играет доля f, а вовсе не число процессоров!

Ещё пример. Возможно ли ускорение вычислений в два раза при переходе с однопроцессорной машины на четырёхпроцессорную? Правильный ответ – не всегда, так как всё зависит от доли f.

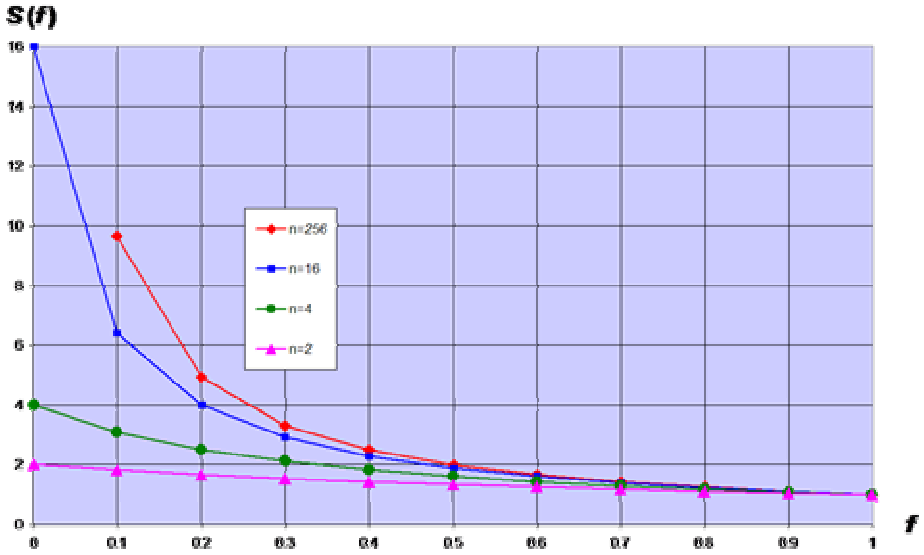

Рис. 2 Закон Амдала

По графикам S(f) можно увидеть, что при f > 50% ни о каком существенном ускорении говорить не приходится. Только если доля f мала, многократное увеличение числа процессоров становится экономически целесообразным.

Более того, имея такую пессимистичную картину в теоретическом плане, на практике не избежать накладных расходов на поддержку многопоточных вычислений, состоящих из алгоритмических издержек (менеджмент потоков и т.п.),

коммуникационных издержек на передачу информации между потоками и издержек в виде дисбаланса загрузки процессоров.

Дисбаланс загрузки процессоров возникает, даже если удалось разбить исходную задачу на равные по сложности подзадачи, так как время их выполнения может существенно различаться по самым разным причинам (от конфликтов в конвейере до планировщика ОС). В точках синхронизации (хотя бы в точке завершения всей задачи) потоки вынуждены ожидать наиболее долго выполняемых, что приводит к простою значительной части процессоров.

1.4. Закон Густафсона

Оптимистичный взгляд на закон Амдала даёт закон Густафсона-Барсиса. Вместо вопроса об ускорении на n процессорах рассмотрим вопрос о замедлении вычислений при переходе на один процессор. Аналогично за f примем долю последовательной части программы. Тогда получим закон масштабируемого ускорения:

$$S(n) = \frac{T(1)}{T(n)} = \frac{f \times T(n) + n \times (1-f) \times T(n)}{f \times T(n) + (1-f) \times T(n)} = n + (1-n) \times f$$

Теперь графики S(f) демонстрируют совершенно иную картину: линейное ускорение в зависимости от числа процессоров. Т.е. законы Амдала и Густафсона в идентичных условиях дают различные значения ускорения. Где же ошибка? Каковы области применения этих законов?

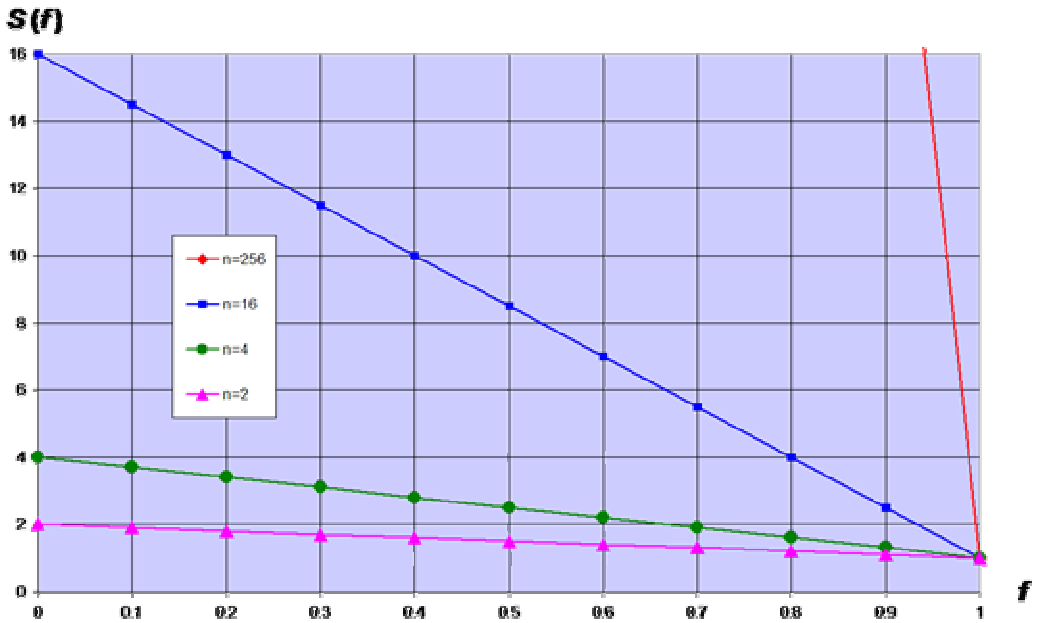

Рис. 3 Закон Густафсона

Густафсон заметил, что, работая на многопроцессорных системах, пользователи склонны к изменению тактики решения задачи. Теперь снижение общего времени исполнения программы уступает объёму решаемой задачи. Такое изменение цели обусловливает переход от закона Амдала к закону Густафсона.

Например, на 100 процессорах программа выполняется 20 минут. При переходе на систему с 1000 процессорами можно достичь времени исполнения порядка двух минут. Однако для получения большей точности решения имеет смысл увеличить на порядок объём решаемой задачи (например, решить систему уравнений в частных производных на более тонкой сетке). Т.е. при сохранении общего времени исполнения пользователи стремятся получить более точный результат.

Увеличение объёма решаемой задачи приводит к увеличению доли параллельной части, так как последовательная часть (ввод/вывод, менеджмент потоков, точки синхронизации и т.п.) не изменяется. Таким образом, уменьшение доли f приводит к перспективным значениям ускорения.

2. Классификации архитектур ВС

Известно более десятка различных классификаций вычислительных систем [4]. Однако ключевой является классификация Флинна, на базе которой возникли новые классификации, детализирующие её исходные классы.

В 1966 году Флинн предложил рассматривать и потоки команд, и потоки данных либо как одиночные, либо как множественные. Так появились четыре класса вычислительных систем: SISD, MISD, SIMD, MIMD.

		Data Stream	
		Single	Multiple
Instruction Stream	Single	**SISD**	**SIMD**
	Multiple	**MISD**	**MIMD**

Рис. 4 Классификация Флинна

Классические фон-неймановские машины попадают в тривиальный класс SISD, в котором одиночный поток инструкций обрабатывает одиночный поток данных. Классические языки высокого уровня (такие, как C++) также ориентированы на программирование в классе SISD. В настоящее время выпуск SISD процессоров почти

прекращён из-за их низкой производительности, которая обусловлена низким уровнем параллелизма вычислений (используется только мелкозернистый параллелизм).

Класс MISD, в котором множественный поток инструкций обрабатывает одиночный поток данных, пуст (бессмысленен).

Все современные передовые процессоры, как общего, так и специального назначения, попадают в класс MIMD. Они одновременно исполняют сразу несколько независимых потоков инструкций, аппаратно обеспечивая крупнозернистый параллелизм. Изъян классификации Флинна заключается в неразличимости современных направлений процессоростроения, что привело к возникновению множества новых классификаций.

Чистых представителей класса SIMD совсем немного. Класс ориентирован на выполнение программ, для которых характерна обработка больших регулярных массивов чисел. Здесь одиночный поток инструкций обрабатывает множественный поток данных. Именно представители класса SIMD впервые достигли производительности порядка GFLOPS.

Наиболее популярная идея класса SIMD – векторное процессирование. Векторный процессор поддерживает обработку не только скалярных, но и векторных операндов. Эффективное декодирование инструкций и удобные данные сказываются на производительности крайне позитивно. Поэтому векторную обработку внедряют и в процессоры классов SISD и MIMD. Например, SIMD расширение IA32 – технологии MMX и SSE.

Другим примером инкапсуляции техники SIMD является процессор STI Cell (альянс Sony, Toshiba и IBM). Процессор является гибридным объединением ведущего процессора (Power Processor Element, PPE) и восьми векторных сопроцессоров (Synergistic Processor Element, SPE). В качестве ведущего процессора используется 64-битный 2-поточный RISC процессор от IBM, традиционно включающий векторное расширение (Vector Multimedia eXtensions, AltiVec). SPE представляют собой параллельно работающие 128-битные SIMD процессоры.

2.1. Класс SIMD

Основные представители класса SIMD: векторные процессоры, матричные процессоры и процессоры с архитектурой VLIW.

У матричных процессоров нет аналогии с векторными. Матричный процессор – это массив процессоров с единым потоком команд. Большой исходный массив данных разделяют на части, подлежащие идентичной обработке. Каждый процессор массива обрабатывает соответствующую часть данных, выполняя единый поток инструкций.

Перспективным представителем класса SIMD является архитектура VLIW (очень длинное командное слово). Одна инструкция в такой системе команд представляет собой кортеж из нескольких RISC инструкций, которые независимы по данным между собой. VLIW процессору не нужно проверять инструкции кортежа на выявление структурных зависимостей, зависимостей по данным или по управлению. Теперь эти функции возложены на компилятор. Процессор сразу может переходить к фазе исполнения. Такие блоки, как динамический планировщик, станции резервации и т.д., здесь упразднены. Высвободившиеся ресурсы (транзисторы) перераспределяются для повышения производительности системы (увеличиваются размеры кешей, буферов BTB и TLB). Таким образом, процессор и компилятор обеспечивают хороший уровень параллелизма команд. Один из самых мощных VLIW процессоров – Intel Itanium 2.

2.2. Класс MIMD

Представителей класса MIMD можно разделить на системы с общей (tightly coupled) или распределённой (loosely coupled) памятью.

Системы с общей памятью (мультипроцессоры) строятся посредством увеличения количества процессоров в машине. Такая мультипроцессорная система симметрична по процессорам. Т.е. задание, вытесненное на одном из предыдущих квантов времени, может быть возобновлено на любом из процессоров. С точки зрения ОС каждый процессор – обычный ресурс, который перераспределяется между заданиями (потоками). Безусловно, ОС стремится к минимизации частоты смены процессора для исполняемых программ. Это позволяет наиболее эффективно использовать ресурсы каждого из процессоров (кешей, буфера BTB, буфера TLB и т.д.). Такой режим симметричного мультипроцессирования (Symmetric Multiprocessor, SMP) поддерживается во всех современных ОС.

Количественные изменения по закону Мура в настоящее время привели к возможности размещения на одном чипе сразу нескольких процессоров (Chip Multiprocessor, CMP). Такой многоядерный (Multi-Core) процессор с точки зрения ОС выглядит как несколько одноядерных и допускает симметричное

мультипроцессирование. Иногда различные ядра имеют общий кеш второго или третьего уровней.

Мультикомпьютерные системы (мультикомпьютеры) – системы с распределённой памятью. Такие системы представляют собой массив мощных серверов, объединённых в единый вычислительный ресурс при помощи высокопроизводительной коммуникационной сети (Massively Parallel Processing, MPP). Кластерные системы – упрощенный вариант MPP. Например, несколько персональных компьютеров в сети Ethernet плюс некоторый механизм распределения вычислительной нагрузки – это уже простейший кластер!

Итак, в настоящее время мощные центры вычислений базируются на крупнозернистом параллелизме и строятся либо как SMP, либо как MPP. Какой подход лучше?

Процессоры в SMP системе представляют собой монолитный ресурс. Программа может быть легко переброшена с перегруженного процессора на недогруженный процессор. Многопоточной программе динамически выделяется то или иное количество процессоров в зависимости от загруженности системы в целом. Всплески вычислительной нагрузки легко перераспределяются планировщиком ОС. Более того, даже зависание одной или нескольких программ не приводит ОС в режим голодания! Но такие позитивные черты обусловлены общей памятью, к которой предъявляются очень высокие требования по производительности. Именно память ограничивает масштабируемость SMP.

В MPP частая передача вычислительной нагрузки между серверами противопоказана, так как приводит к медленным процедурам внешнего ввода/вывода. Фактически, это не единый мощный ресурс. Это объединение ресурсов для решения той или иной задачи, требующее программирования специального вида. Требование специальной организации вычислений вынуждает либо существенно модифицировать имеющееся программное обеспечение, либо разрабатывать его вновь, что, безусловно, затрудняет переход от SMP к MPP. Ключевым преимуществом MPP является отличная масштабируемость, которая ограничивается только предельными параметрами коммуникационной сети.

2.3. Симметричное мультипроцессирование

Архитектура SMP. Система состоит из нескольких однородных процессоров и массива общей памяти. Все процессоры имеют доступ к любой точке памяти с

одинаковой скоростью. Аппаратно поддерживается когерентность кешей. Процессоры подключены к памяти либо с помощью общей шины, либо с помощью crossbar-коммутатора. Коммутатор, базируясь на принципе локальности программ, с целью распараллеливания банков памяти коммутирует процессоры с банками памяти. Фактически, в каждый текущий момент времени каждый процессор имеет доступ только к некоторому фрагменту общей памяти.

Масштабируемость SMP. Наличие общей памяти сильно упрощает взаимодействие процессоров между собой, однако накладывает существенные ограничения на их число. В реальных системах установлено не более 32 ядер (сами ядра, как правило, многопоточные).

Операционная система для SMP. Вся система работает под управлением единой ОС, которая автоматически распределяет процессы по процессорам.

Модель программирования в SMP. Достаточно организовать классические потоки POSIX.

2.4. Массивно-параллельные системы

Архитектура MPP. Система состоит из однородных вычислительных узлов, включающих один или несколько центральных процессоров, локальную память (прямой доступ к памяти других узлов невозможен), коммуникационный процессор. К системе могут быть добавлены специальные узлы ввода/вывода и управляющие узлы. Узлы связаны через некоторую коммуникационную среду.

Масштабируемость MPP. Общее число процессоров в реальных системах достигает нескольких сотен тысяч!

Операционная система для MPP. Полноценная ОС работает только на управляющей машине. На каждом узле работает сильно урезанный вариант ОС, обеспечивающий только работу расположенной в нём ветви параллельного приложения.

Модель программирования в MPP. Программирование производится в рамках модели передачи сообщений (MPI, PVM, BSPlib).

2.5. Кластерные системы

После появления однокристальных микропроцессоров (СБИС технология, четвёртое поколение развития вычислительной техники) строительство суперкомпьютеров ориентировалось на построение специализированных

многопроцессорных систем из массовых микросхем. До середины 1990-х годов в развитии суперкомпьютерных технологий преобладали SMP и MPP подходы.

При симметричном мультипроцессировании процессоры объединяются с использованием общей памяти, что существенно облегчает программирование (достаточно организовать классические потоки POSIX), но предъявляет высокие требования к производительности самой памяти. Более дешевым и хорошо масштабируемым подходом (в сравнении с SMP) является построение мультикомпьютерной системы с распределённой памятью. Такой мультикомпьютер представляет собой массив независимых специализированных вычислительных модулей, объединённых в единый вычислительный ресурс при помощи высокопроизводительной коммуникационной сети со специализированными каналами связи. И вычислительные модули, и каналы связи создаются исключительно под конкретный суперкомпьютер.

Идея создания так называемого кластера рабочих станций явилась развитием MPP подхода. К концу 1990-х годов, благодаря развитию шины PCI (PCI-X, PCI-E), появлению гигабитного Ethetnet и т.п., стандартные персональные компьютеры, объединённые обычной локальной сетью, стали догонять специализированные MPP системы по коммуникационным возможностям. Таким образом, появилась возможность построения полноценной MPP системы из стандартных рабочих станций общего назначения при помощи серийных коммуникационных технологий. Причём стоимость такого кластера оказывалась ниже в среднем на два порядка!

В настоящее время кластер состоит из процессорных элементов (стандартных вычислительных узлов на базе стандартных процессоров), соединённых высокоскоростной сетью передачи данных. Большинство передовых современных кластерных систем (из списка Top500) базируется на процессорах Intel (Quad-Core Intel Xeon, Dual-Core Intel Itanium 2). В качестве процессорных элементов обычно используются SMP сервера 1U в 19-дюймовом форм-факторе. Процессоры других производителей также используются (IBM Power 6, IBM PowerPC, Sun UltraSPARC Niagara II, DEC Alpha (в Cray T3E) и т.д.).

3. Эволюция микропроцессорных архитектур

Рассмотрим основные вехи в эволюции микропроцессорных архитектур. На ранних стадиях эволюции смена поколений ассоциировалась с революционными технологическими прорывами: каждое из первых четырёх поколений имело чётко

выраженные отличительные технологические признаки. Последующие нечёткие деления на поколения основаны на изменениях в архитектуре вычислительных систем, а не на изменениях элементной базы.

Итак, вплоть до 1945 года была “механическая” эра эволюции вычислительной техники (нулевое поколение). Ключевым моментом эволюции является концепция вычислительной машины с хранимой в памяти программой, сформулированная Джоном фон Нейманом в 1945 году. Относительно неё историю развития вычислительной техники можно представить в виде трёх этапов: донеймановкого периода, эры вычислительных машин с фон-неймановской архитектурой и постнеймановской эпохи (эпохи параллельных и распределённых вычислений). Напомним, что сущность фон-неймановской концепции состоит из четырёх принципов: двоичного кодирования, программного управления, однородности памяти (с позиций современного программирования не приветствуется) и адресности.

Первое поколение вычислительной техники (1937–1953) использовало электронные компоненты, которые могли переключаться в тысячу раз быстрее электромеханических аналогов. Программирование осуществлялось в машинном коде, а в пятидесятых годах вместо числовой записи появилась символьная нотация (ассемблер). В простейшем дизайне процессора был только аккумулятор.

Второе поколение (1954–1962) ознаменовалось переходом от электронных ламп к полупроводниковым диодам и транзисторам со временем переключения порядка 0.3 мс. Появление регистрового файла позволило организовать машины либо с регистровой архитектурой, либо со стековой архитектурой. Тогда стековая архитектура превалировала над регистровой, благодаря очень компактному коду. Безоперандные команды брали аргументы из стека и клали результат в стек. Для кодирования такой команды достаточно нескольких бит. В условиях ограниченности объёма доступной памяти это было очень веским преимуществом. В дальнейшем появление полупроводниковых запоминающих устройств привело к резкому увеличению объёма оперативной памяти и, соответственно, к бесперспективности стековой архитектуры.

Третье поколение вычислительной техники (1963–1972) ознаменовалось переходом от дискретных полупроводниковых элементов к интегральным микросхемам, что обусловило скачок производительности. Зарождаются принципы конвейеризации и суперскалярности (принципы мелкозернистого параллелизма). Появляются многозадачные ОС. Архитектура системы команд (ISA) сильно развивается,

предоставляя программистам более развитые команды. Такую ISA впоследствии назовут CISC.

Четвёртое поколение (1972–1984) осуществило переход на СБИС (тысячи транзисторов на одном кристалле). Появление языков высокого уровня привело к резкому снижению востребованности CISC инструкций, так как компиляторы использовали только небольшое подмножество из них. Оказалось целесообразным перейти к архитектуре с сокращённым набором команд (RISC) и использовать высвобожденные ресурсы (транзисторы) для улучшения количественных характеристик CPU.

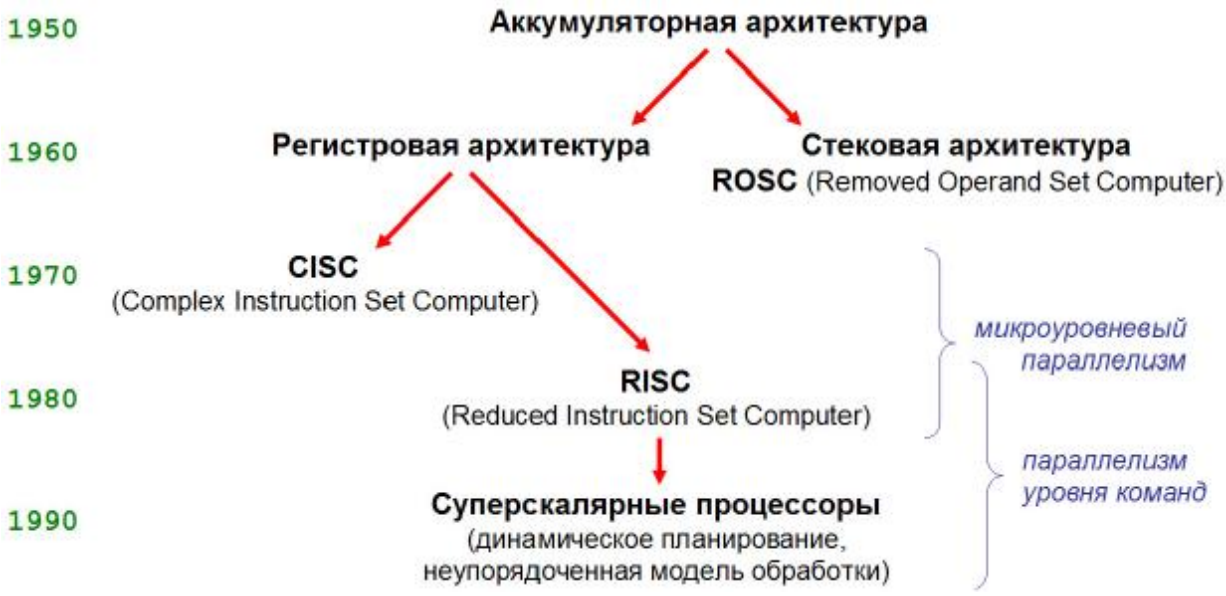

Рис. 5 Эволюция ISA

В девяностых годах доминируют суперскалярные процессоры. Принципы мелкозернистого параллелизма эксплуатируются максимально. Процессор содержит сложную логику динамического планирования в неупорядоченной модели обработки. Увеличение числа конвейеров и станций резервации приводит к экспоненциальному росту взаимосвязей между ними, что ограничивает дальнейшее распараллеливание на уровне команд. Назрела необходимость использования более высоких уровней параллелизма, началась эпоха параллельных и распределённых вычислений (постнеймановская эпоха).

Заметим, что история развития дизайна процессоров (включая эволюцию ISA) связана с агрегированием всё более высоких уровней параллелизма, что в настоящее время привело к мультипроцессорным и мультикомпьютерным системам.

Система команд первых трёх поколений вычислительной техники стремилась удовлетворить растущие потребности развивающихся технологий программирования, включая в состав ISA новые, более развитые команды. Однако угнаться за высоким темпом развития технологий программирования процессоростроители не могли. Образовался семантический разрыв. Так самая сложная в мире CISC система команд x86 не содержит никакой поддержки, например, объектно-ориентированного программирования. Отказ от языковой гонки (переход на RISC) и постоянное совершенствование языков программирования (Java и т.д.) закрепили семантический разрыв окончательно. Тем не менее, стоит отметить, что в последнее время всё активнее осуществляется аппаратная поддержка процессорами общего назначения машинного кода Java (иначе программная интерпретация кода Java имеет низкую производительность) либо в полном объёме, либо частично. Сейчас язык Java и код Java возродили интерес к стековой архитектуре (ROSC). В процессоры всё чаще стали интегрировать виртуальные машины.

Все современные процессоры базируются на RISC. Некоторые могут возразить: например, машинный код Pentium 4 типа CISC, а машинный код сопроцессора Pentium 4 типа ROSC. Тем не менее, Pentium 4 имеет RISC ядро (начиная с i486). Поддержка CISC и ROSC инструкций выполнена с использованием техники Code Morphing для обеспечения преемственности x86 программ. Фактически, CISC и ROSC интерпретируются процессором при помощи декодирования в RISC инструкции (в микрокод).

3.1. Тенденции развития GPP

Итак, назревшая необходимость использования более высоких уровней параллелизма определила перспективные пути развития GPP: VLIW, SMT и CMP. Переход от RISC к VLIW (как в процессорах общего, так и специального назначения) позволил преодолеть ограниченность ILP. Тенденции многопоточного (SMT) и многоядерного процессирования (CMP) привели к аппаратной поддержке крупнозернистого параллелизма.

Рис. 6 Тенденции развития GPP

3.2. Архитектура VLIW

Процессоры с множественной выдачей инструкций (multiple-issue processors) ориентированы на исполнение нескольких инструкций за такт и бывают двух видов: суперскалярные процессоры (superscalar processors) и процессоры с архитектурой VLIW (Very Long Instruction Word).

Первые суперскалярные процессоры работали в режиме упорядоченной выдачи команд. Но упорядоченная выдача команд (in-order issue) неэффективна, так как требуемое функциональное устройство (FU) может оказаться занятым. Упорядоченное завершение команд (in-order completion) также неэффективно, так как остановка продвижения в одном FU приведёт к простою всех FU. Неупорядоченные выдача и завершение – неупорядоченная модель обработки (out-of-order execution) – дополнительный потенциал повышения производительности суперскалярного процессора. Современные суперскалярные процессоры исполняют от 2 до 10 инструкций за такт и используют аппаратную логику анализа ILP перед выдачей команд. Такой аппаратный механизм переупорядочивания исполнения инструкций (out-of-order engine) называется динамическим планированием. Компилятор и динамический планировщик не могут обойти все конфликты (структурные, по данным, по управлению) и задержки доступа к памяти (при кеш-промахах). Таким образом, фактическое число выданных в такте инструкций колеблется от нуля до максимально возможного (загрузка FU колеблется от 0% до 100%).

Если суперскалярные процессоры исполняют переменное число инструкций за такт, используя методы как статического (развёртка кода компилятором), так и динамического (алгоритм Томасуло) планирования, то VLIW процессоры, напротив, исполняют фиксированное число независимых инструкций, сгруппированных в одну

длинную инструкцию. В таком пакете инструкций параллелизм уровня инструкций обеспечивается статически на этапе компиляции. Компания Intel, например, именует такую методику явного распараллеливания инструкций – EPIC – Explicitly Parallel Instruction Computing.

Суть явного параллелизма заключается в том, что распределение команд между исполнительными узлами производится не процессором в ходе выполнения программы (динамически), а компилятором при формировании машинного кода (статически). Таким образом, схемотехника VLIW процессора существенно упрощается (по сложности он сравним с суперскалярным процессором без поддержки неупорядоченного исполнения команд). Кроме того, в компиляторах алгоритмы выбора порядка исполнения команд могут быть существенно сложнее и эффективнее, чем алгоритмы аппаратного планирования инструкций (так как решение необходимо принимать в течение наносекунд).

3.3. Классические многопоточные процессоры

Для однопоточного суперскалярного процессора характерен простой функциональных устройств вследствие имеющихся зависимостей (в первую очередь типа RAW) в исходном алгоритме решения задачи. Потерянные слоты выдачи инструкций можно классифицировать на вертикальные и на горизонтальные потери.

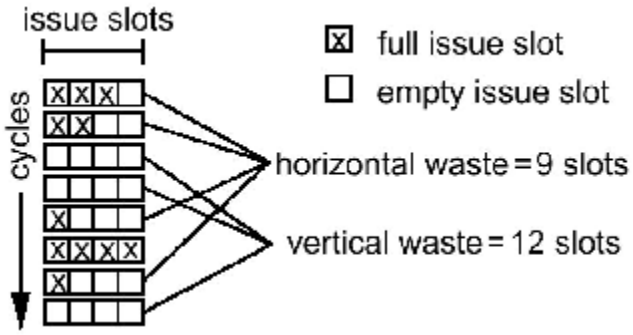

Рис. 7 Ограничение суперскалярных архитектур

Причины возникновения пустых слотов – структурные конфликты, конфликты данных и управления. Так на рис. 7 приведён пример работы однопоточного четырёхконвейерного процессора, для которого идеальное IPC = 4. Реальное IPC будет существенно меньше идеального (пикового) IPC.

Классическое многопоточное процессирование снижает вертикальные потери слотов выдачи за счёт использования параллелизма более высокого уровня – параллелизма уровня потоков (TLP). Вместо простоев при исполнении единственного

потока команд происходят переключения на исполнение других потоков команд (в порядке кольцевой очереди). При этом усложнение аппаратуры минимально – необходимо поддерживать несколько контекстов исполнения (в основном несколько регистровых файлов). Только один поток (один контекст) выдаёт инструкции каждый такт. С точки зрения ОС множество аппаратных контекстов – это несколько логических процессоров (SMP система). Такая аппаратная многопоточность не снижает горизонтальные потери, так как параллелизм уровня инструкций в каждом отдельном потоке остаётся ограниченным.

В [5] под нагрузкой SPEC92 собрана статистика причин простоев восьмипоточного процессора: реальное IPC ≈ 1.5, потерян 81% слотов выдачи, из них вертикальных – 61% (рис. 8).

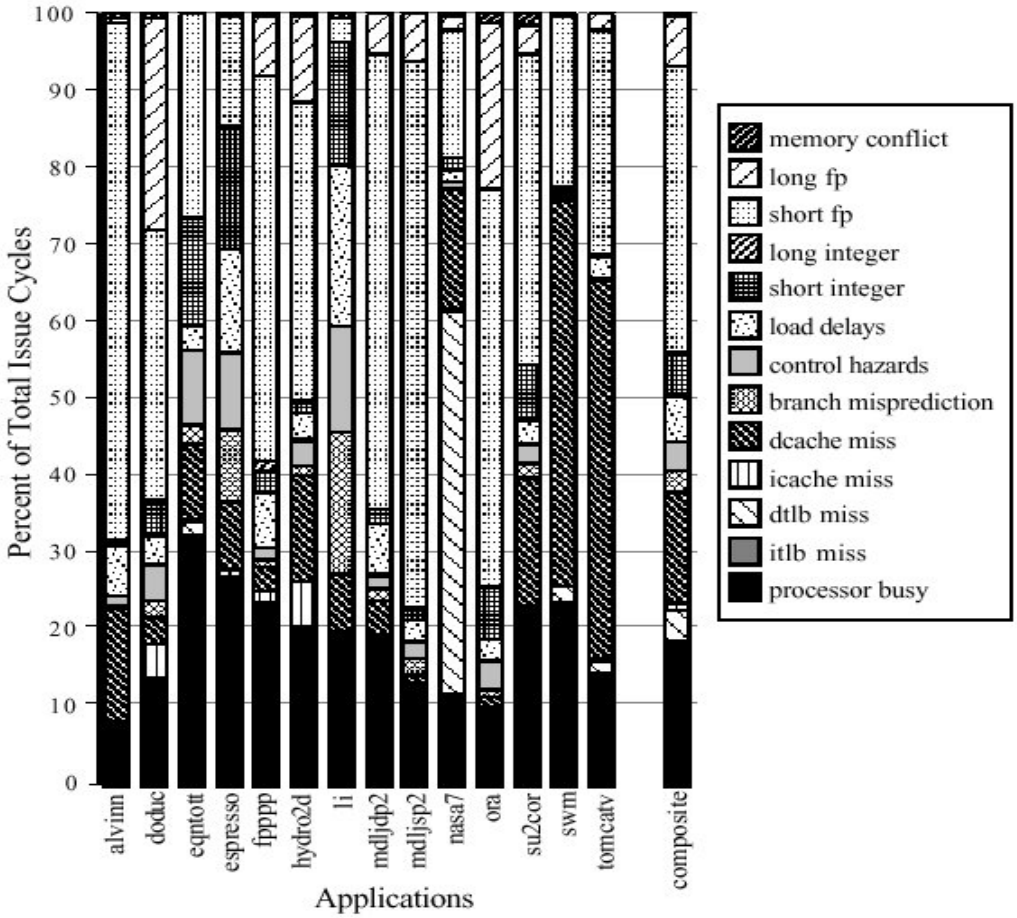

Рис. 8 Статистика причин простоев

3.4. Конвейерная многопоточность

Simultaneous Multithreading (SMT) – эволюционная микропроцессорная архитектура, впервые представленная в 1995 году в университете Вашингтона Дином

Тулсеном (Dean Tullsen) и предназначенная для повышения эффективности использования аппаратных ресурсов в многопоточных суперскалярных микропроцессорах.

SMT использует параллелизм уровня потоков (TLP) на одном вычислительном ядре, позволяющий производить одновременную выдачу, исполнение и завершение инструкций из различных потоков в течение одного и того же такта. Один физический SMT процессор действует как несколько логических процессоров, каждый из которых исполняет отдельный поток инструкций. SMT предоставляет эффективный механизм скрытия длительных простоев конвейера, снижая как вертикальные, так и горизонтальные потери.

Экономическая целесообразность технологии SMT: рост производительности значительно больше, чем рост площади чипа и энергопотребления.

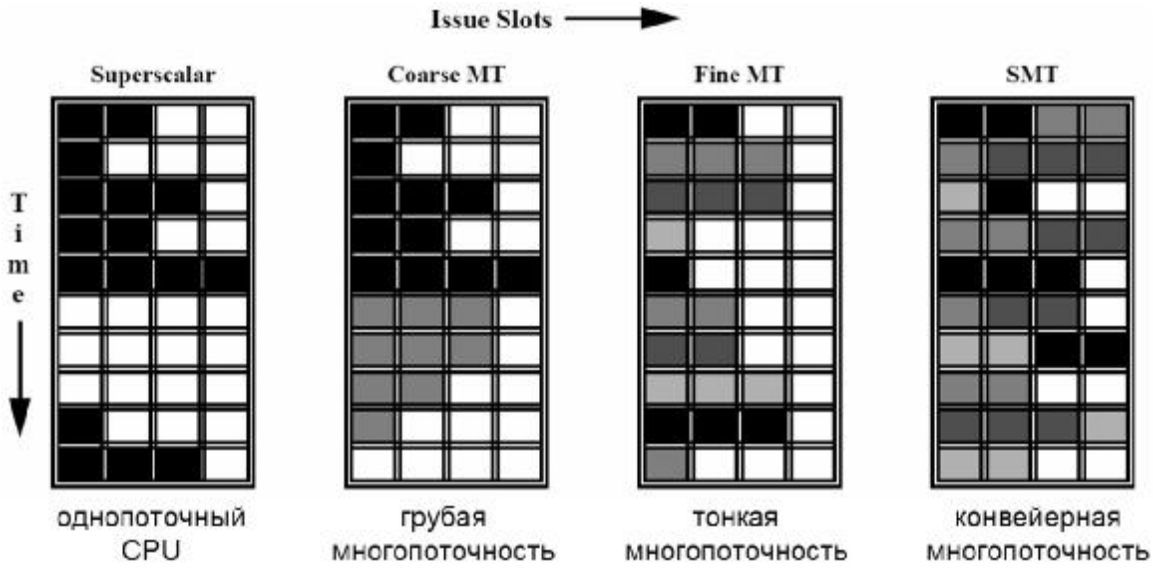

Рис. 9 Сравнение эффективности

При множестве исполняющихся потоков в случаях промахов кеша, неверно предсказанных переходов и т.п. скрываются даже длительные штрафы (с большими задержками). Снижение и горизонтальных, и вертикальных потерь слотов выдачи ведёт к увеличению скорости выдачи инструкций (к увеличению IPC). Функциональные устройства совместно используются всеми контекстами в каждом такте. Выдача инструкций для функциональных устройств множеством потоков повышает использование ресурсов процессора. Однако с целью поддержки множества исполняющихся потоков накладываются более жёсткие требования к размерам ресурсов процессора (кешей, буфера BTB, буфера TLB, буфера переименования и т.д.).

Необходимые изменения для поддержки SMT: множество контекстов исполнения и привязка каждой инструкции к своему контексту при прохождении всего конвейера; механизм выборки инструкций из множества потоков; отдельные для каждого потока механизмы завершения инструкций, менеджмента очереди инструкций и обработки исключений.

Изменения для повышения производительности SMT: больший аппаратный регистровый файл для поддержки переименования регистров во всех потоках; большая пропускная способность доступа к памяти; большие кеши для компенсации снижения производительности из-за совместного использования несколькими потоками (из-за снижения локальности); больший BTB; больший TLB.

Преимущества SMT: преодоление ограничений производительности суперскалярной обработки, связанных с низким параллелизмом уровня инструкций, посредством применения параллелизма уровня потоков; повышение эффективности использования аппаратных ресурсов; предоставление механизма скрытия длительных простоев конвейера. Недостатки SMT: повышенные требования к производительности иерархии памяти; повышенные требования к размерам ресурсов процессора.

3.5. Модели многопоточных процессоров

Модели многопоточного процессора, который может выдавать до, например, восьми инструкций за такт, отличаются способами использования слотов выдачи и функциональных устройств.

Модель **Fine-Grain Multithreading** – классическая тонкая многопоточность, скрывающая все источники вертикальных потерь, но не горизонтальных. Только один поток выдаёт инструкции каждый такт, зато может использовать всю ширину выдачи процессора.

Модель **SM: Limited Connection** – ограниченные связи в конвейере. Каждый аппаратный контекст напрямую соединен только с некоторыми функциональными устройствами. Например, если аппаратура поддерживает восемь потоков и имеет четыре целочисленных устройства, каждое целочисленное устройство может получать инструкции в точности от двух потоков. Разбиение функциональных устройств по потокам в результате менее динамично, чем в других SMT моделях, но каждое функциональное устройство разделяемо (критический фактор при достижении высокой утилизации ресурсов).

Модели **SM: Single/Dual/Four Issue**. В этих трёх моделях ограничивается количество инструкций, которое каждый поток может выдать или иметь активными в окне планирования на каждом такте. Например, в SM: Dual Issue каждый поток может выдать максимум две инструкции за такт, и потребуется минимум четыре потока для заполнения восьми слотов выдачи в одном такте.

Модель **SM: Full Simultaneous Issue** – полностью одновременная выдача – самая гибкая модель суперскалярного процессора с конвейерной многопоточностью. Все восемь потоков конкурируют за каждый из восьми слотов выдачи каждый такт. Наивысшая сложность аппаратной реализации.

В [5] произведено количественное сравнение производительности этих моделей (рис.10).

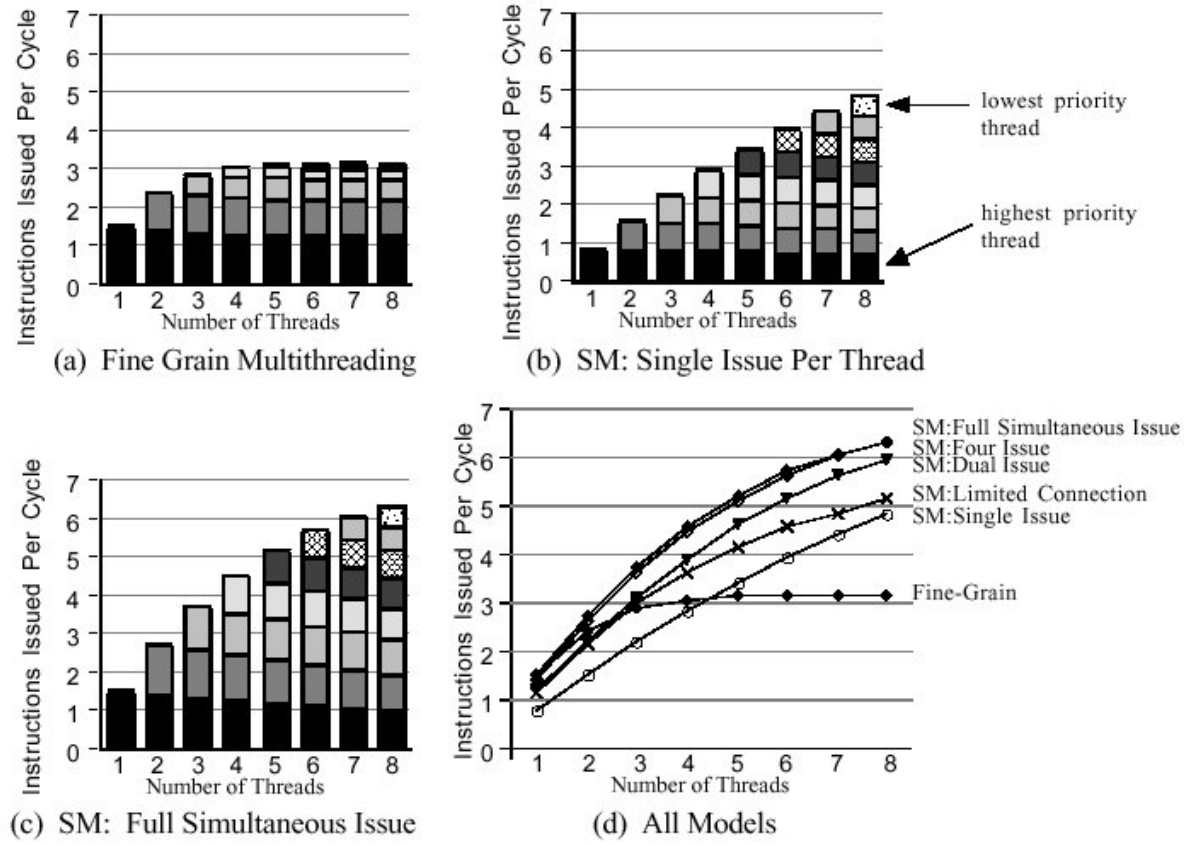

Рис. 10 Сравнение производительности моделей

3.5.1. Стратегии планирования выборки инструкций

Стратегия **Round Robin** – выборка по регулярному графику. Например, в RR 1.8 каждый такт из одного потока выбирается до восьми инструкций, а в RR 2.4 каждый такт из двух потоков выбирается до четырёх инструкций.

Стратегия **BR-Count**. Наивысший приоритет у потоков с наименьшей вероятностью ложной спекуляции исполнения (с наименьшим числом невычисленных переходов).

Стратегия **MISS-Count**. Наивысший приоритет у потоков с наименьшей частотой промахов в кеше данных.

Стратегия **I-Count**. Наивысший приоритет у потоков с наименьшим числом инструкций в статической части конвейера (в очереди декодированных инструкций).

В [6] произведено количественное сравнение производительности этих стратегий (рис. 11 и 12). Так, например, ICOUNT 2.8 увеличивает производительность по сравнению с RR 2.8 на 23% за счёт уменьшения помех в потоке инструкций, выбирая лучшую смесь инструкций.

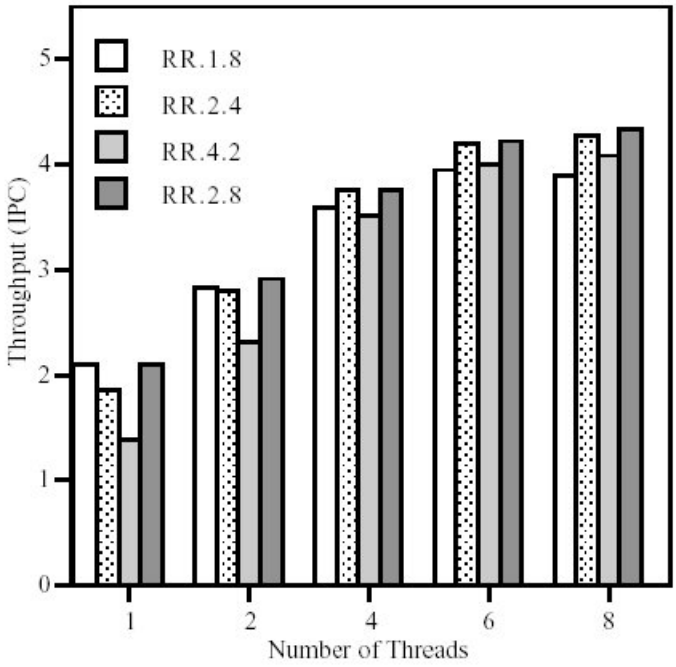

Рис. 11 Производительность Round Robin

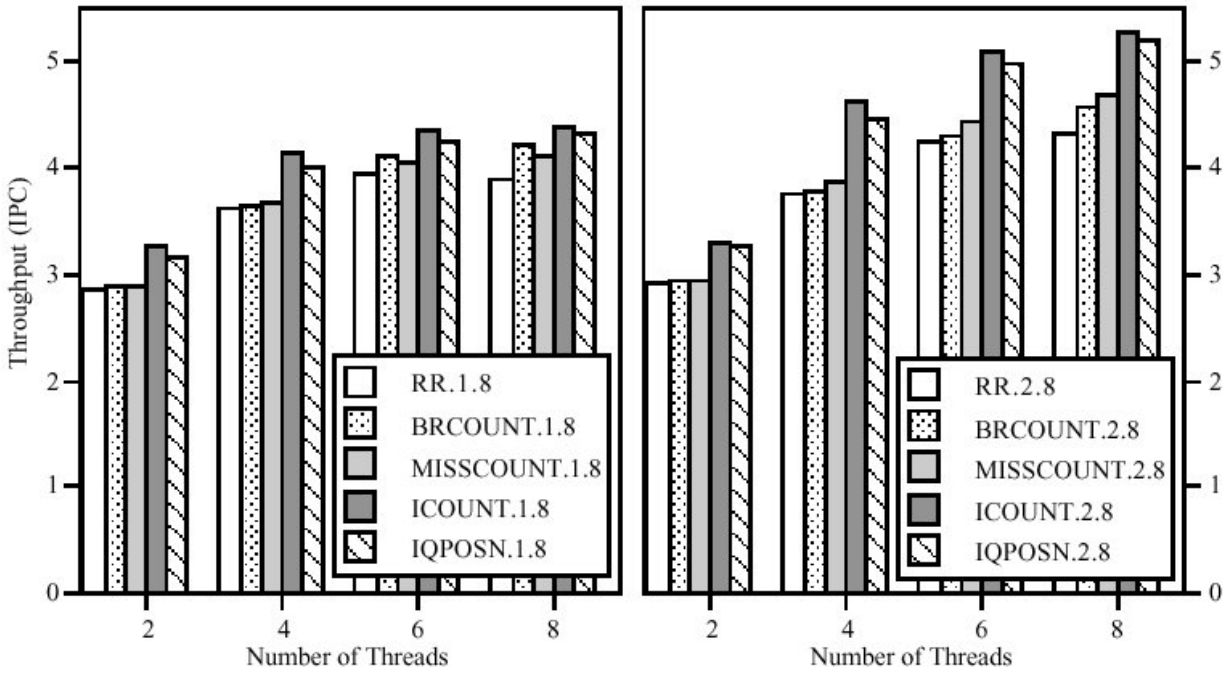

Рис. 12 Сравнение эвристик выборки инструкций

Пример многопоточного процессора: Sun UltraSPARC Niagara II

– 8 ядер × 8 потоков

– кеш I уровня кода 16K, данных 8K

– кеш II уровня 4M для каждого ядра

– ядро: 1.4 ГГц, 8 SMT, 65 нм, 11 слоёв, 342 мм^2

Пример многопоточного процессора: IBM Power 6

– 32 ядра × 2 потока

– кеш I уровня кода 64K, данных 64K

– кеш II уровня 8M для каждого ядра

– кеш III уровня 32M (16-входовый)

– ядро: 4.7 ГГц, 2 SMT, 65 нм, 10 слоёв, 341 мм^2

Пример MPP системы: IBM BlueGene/L

– всего 131072 процессора

– производительность 400 TFLOPS

– узлы IBM PowerPC 440 – 2 × 0.7 ГГц, 5.6 GFLOPS

– 3D Torus network: 2.1 Gb/s

– площадь: 1225 м^2

– энергопотребление: 1.7 МВатт

Пример MPP системы: IBM BlueGene/P

– всего 294912 процессора

– производительность 1 PFLOPS

– узлы IBM PowerPC 450 – 4 × 0.85 ГГц, 13.9 GFLOPS

– 3D Torus network: 5.1 Gb/s

– площадь: 1728 м^2

– энергопотребление: 2.3 МВатт

3.6. Эра Multicore и коррекция тенденций развития

2007 год оказался революционным в процессоростроении: теперь на рынке нет серверов с одноядерными микропроцессорами. Началась эра многоядерного процессоростроения. Процессоры штурмуют TeraFLOP-ный рубеж, а MPP системы – PetaFLOP-ный рубеж [8].

До эры Multicore вычислительная техника развивалась согласно известным трендам, называемыми законами Мура [1]:

– удвоение плотности элементов процессорных СБИС каждые два года,

– удвоение частоты процессорных СБИС каждые два года,

– удвоение количества элементов запоминающих СБИС каждые полтора года,

– двукратное уменьшение времени доступа запоминающих СБИС каждые десять лет,

– удвоение ёмкости внешней памяти каждый год,

– двукратное уменьшение времени доступа во внешней памяти каждые десять лет,

– удвоение скорости передачи данных в глобальных сетях каждый год.

Однако 15 лет экспоненциального роста тактовой частоты закончились [7]. Энергопотребление удалось стабилизировать. ILP достиг предела. Хотя экспоненциальный рост числа транзисторов сохраняется (рис. 13).

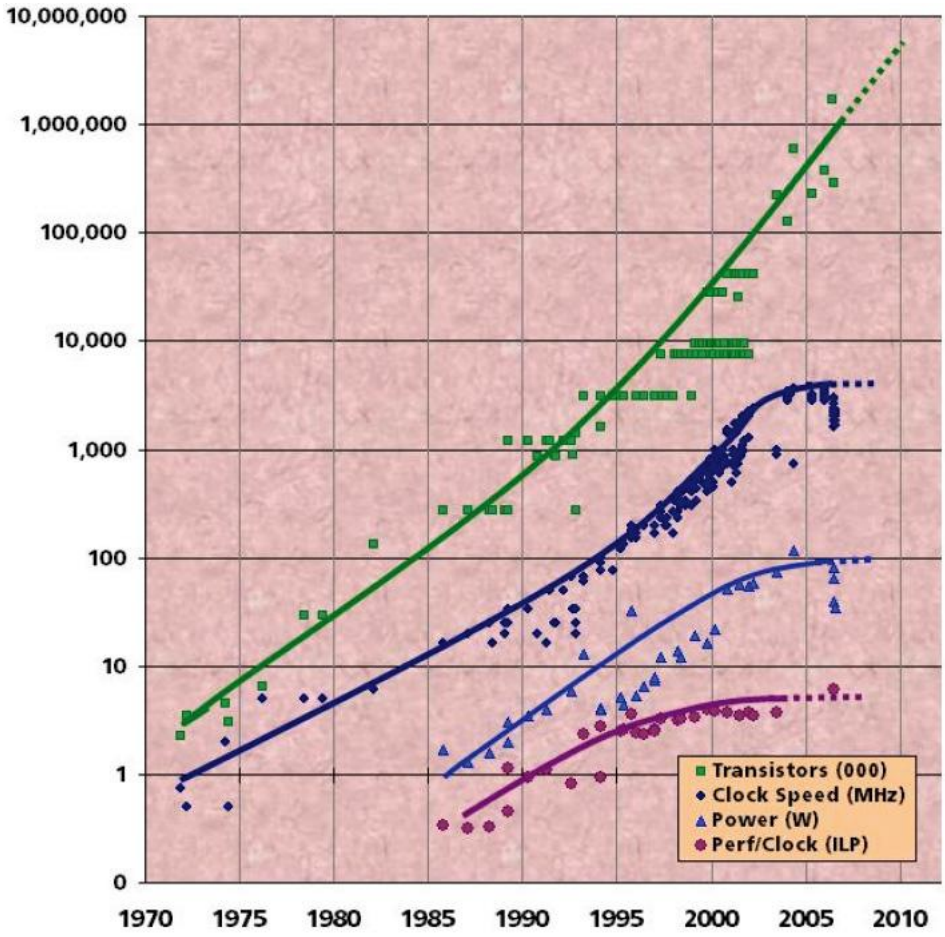

Рис. 13 Современные тренды развития

Таким образом, традиционные источники повышения производительности (тактовая частота, IPC) стабилизировались. Теперь работает новый закон Мура: число ядер удваивается каждые 18 месяцев (вместо удвоения частоты). То есть, по сути, процессор стал новым транзистором!

Энергопотребление мультипроцессоров тесно связано с их производительностью [9] (рис. 14).

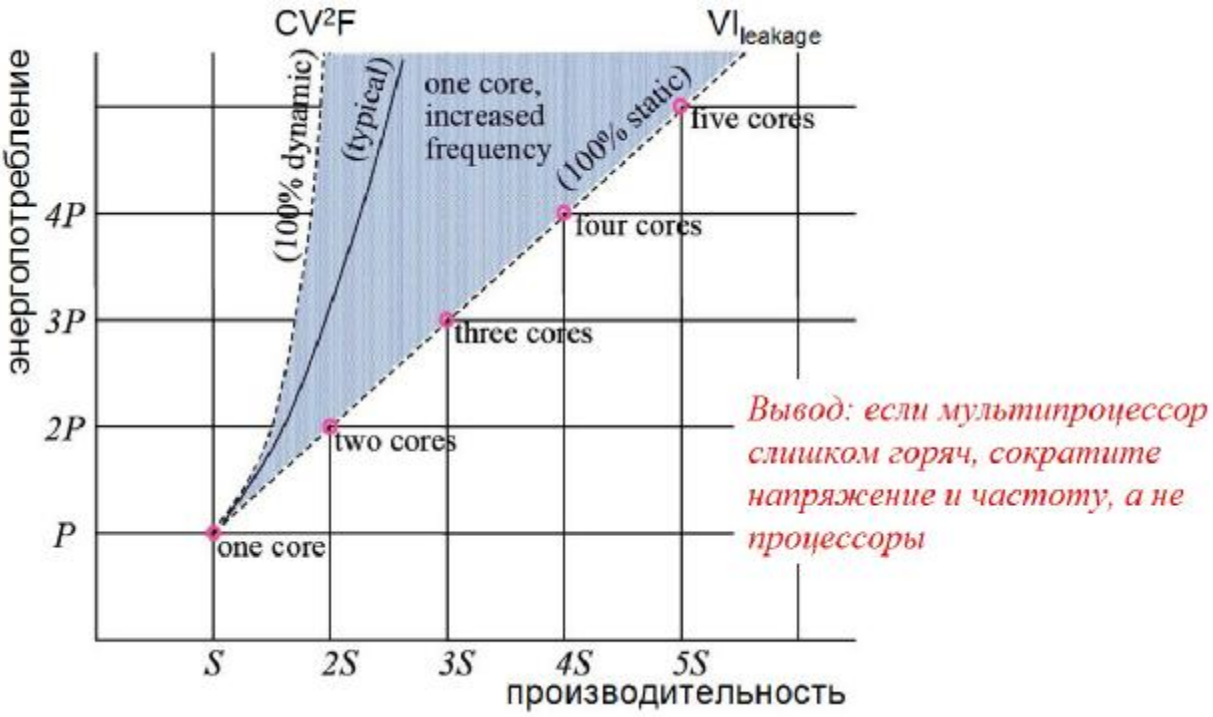

Рис. 14 Энергопотребление мультипроцессоров

3.6.1. Малые ядра

В настоящее время у передовых процессоров со сложными ядрами сложилось несколько ключевых проблем: жёсткие ограничения по энергопотреблению, драматичное падение темпов роста производительности, крайне сложное тестирование работоспособности.

Назовём малыми ядрами упрощенные ядра с короткими конвейерами и упорядоченной обработкой на небольших частотах.

Оказывается, малые ядра избавлены от проблем передовых ядер: они имеют меньшее энергопотребление и их легче проверять. Кроме того, малые ядра не на много медленнее больших, а на чипе их может быть существенно больше.

Power5 (Server)
– 389 мм^2
– 120 Ватт, 1900 МГц
Intel Core2 (Laptop)
– 130 мм^2
– 15 Ватт, 1000 МГц
ARM Cortex A8 (BMW Auto)
– 5 мм^2
– 0.8 Ватт, 800 МГц
Tensilica DP (Cell phones)
– 0.8 мм^2
– 0.09 Ватт, 600 МГц
Tensilica Xtensa (ClSCO router)
– 0.32 мм^2 для трёх ядер
– 0.05 Ватт, 600 МГц

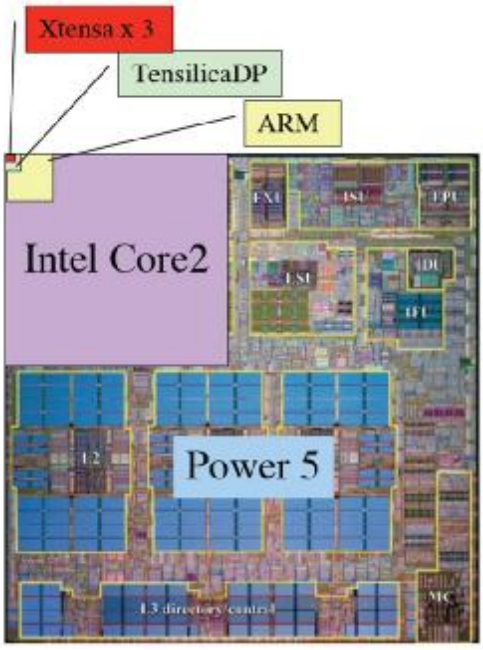

Рис. 15 Сравнение ядер

3.6.2. Новое понятие вычислительной эффективности

Передовые MPP системы потребляют мегаватты энергии! Стоимость энергии больше стоимости оборудования! Энергопотребление ограничивает будущий рост систем. Поэтому актуально введение нового понятия вычислительной эффективности: **performance/watt** (вместо пиковой производительности).

Processor	Number of Cores	Clock Speed	Peak Gflops	Gflops/W
AMD Opteron	2	2.8 GHz	11.2	0.093
ClearSpeed CSX600	96	250 MHz	48	4.8
IBM BlueGene/L ASIC	2	700 MHz	5.6	0.43
Intel Clovertown	4	2.66 GHz	42.56	0.35
Intel Tera-scale Processor	80	3.16 GHz	1010	16
STI Cell	8+1	3.2 GHz	204.8	2.4

Рис. 16 Сравнение ядер

Сравнительный анализ современных многоядерных процессоров различных производителей (рис. 16) демонстрирует преимущество процессоров, построенных на

малых ядрах. Именно такие процессоры достигают большей вычислительной эффективности в пересчёте на один ватт затраченной энергии.

Таким образом, сравнение процессоров по шкале FLOPS уходит в прошлое (так же как и по MIPS и тем более по тактовой частоте). Например, компания Intel в линейке многоядерных процессоров Xeon ориентируется на новую шкалу вычислительной эффективности (рис. 17 и 18).

Рис. 17 Эффективность Multicore Intel Xeon

Процессоры Intel® Xeon®	Архитектура Intel®	Количество ядер в процессоре	Количество операций за тактовый цикл	Объем кэш-памяти	Тактовые частоты	Максимальная частота системной шины	Энергоэффективность
Четырехъядерный процессор серии 5300	Микроархитектура Intel® Core™	4	4	8 МБ	3,00 ГГц 2,66 ГГц 2,33 ГГц 2,00 ГГц 1,86 ГГц 1,60 ГГц	1333	20-30 Вт на 1 ядро
Двухъядерный процессор серии 5100	Микроархитектура Intel® Core™	2	4	4 МБ	3,00 ГГц 2,66 ГГц 2,33 ГГц 2,00 ГГц 1,86 ГГц 1,60 ГГц	1333	32,5 Вт на 1 ядро
Одноядерный процессор	Микроархитектура Intel NetBurst®	1	3	2 МБ	3,80 ГГц 3,60 ГГц 3,40 ГГц 3,20 ГГц 3,00 ГГц	800	110 Вт на 1 ядро

Рис. 18 Эффективность Multicore Intel Xeon

3.6.3. Переход от Multicore к Manycore

Итак, Multicore процессоры (например, Intel Core2 Duo) используют прогрессивные ядра (архитектуры) и ориентированы на типичную вычислительную нагрузку (workload). Согласно новому закону Мура число ядер на таких процессорах будет удваиваться каждые 18 месяцев. Имея хорошую характеристику пиковой производительности, Multicore процессоры уступают Manycore процессорам с точки зрения энергоэффективности.

Manycore процессоры (например, Nvidia G80 (128 cores), Intel Polaris (80 cores), Cisco/Tensilica Metro (188 cores)), напротив, используют упрощенные ядра (короткие конвейеры, небольшие частоты, in-order обработка). Согласно новому закону Мура число ядер на таких процессорах также будет удваиваться каждые 18 месяцев. Преимущества Manycore процессоров: наилучшая вычислительная эффективность на ватт, простое тестирование, низкая вероятность дефектов производства, малая стоимость разработки.

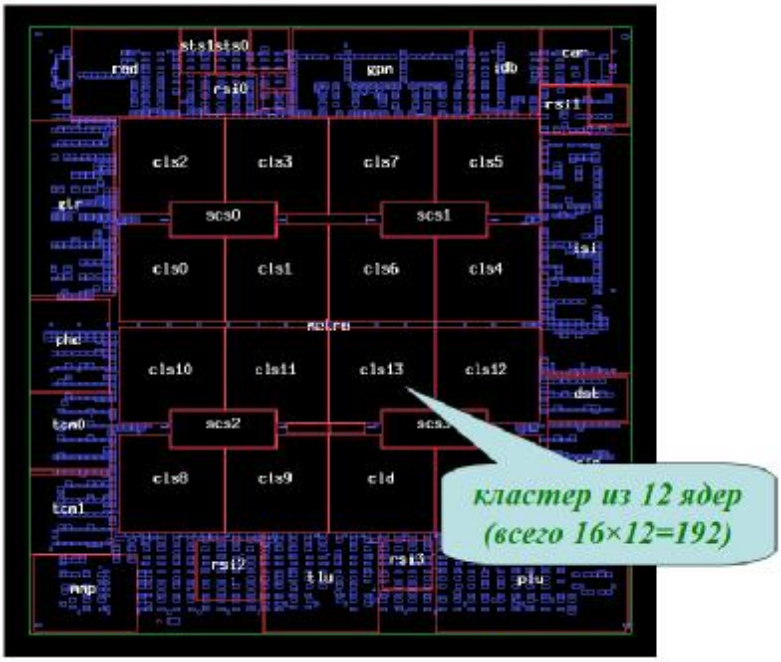

Рис. 19 Tensilica Xtensa

На рис. 19 приведён пример 192-х ядерного процессора Tensilica Xtensa, используемого в Cisco CRS-1 Terabit Router. 188 ядер общего назначения доступны для программного обеспечения. На случай выхода их из строя аппаратно зарезервированы ещё четыре ядра.

3.7. Препятствия на пути многоядерных и многопроцессорных систем

Согласно [7] совсем скоро будет преодолён рубеж в один миллион ядер в системе. Однако на пути развития многоядерных и многопроцессорных систем имеется ряд проблем.

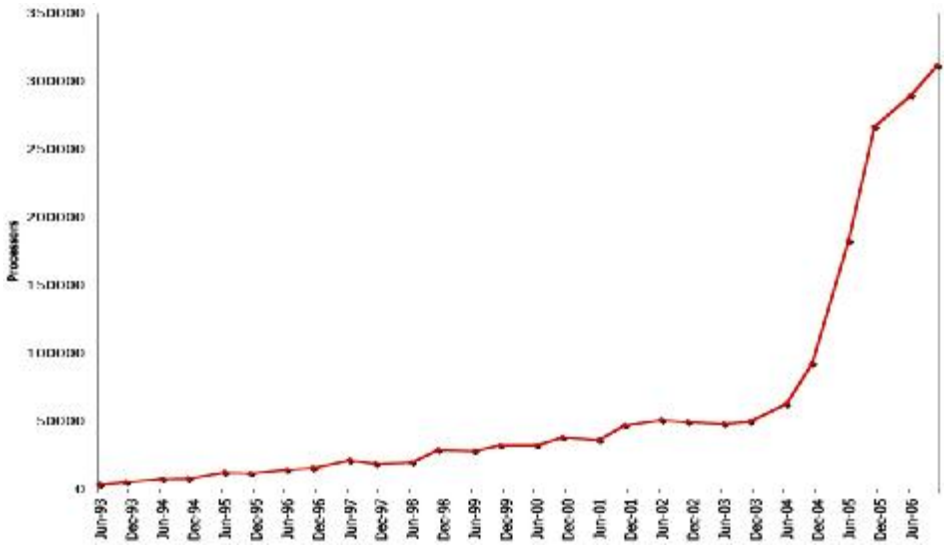

Рис. 20 Число процессоров в системах из TOP15

Во-первых, обеспечение баланса системы. Традиционная проблема SMP систем – доступ к RAM. Полоса пропускания RAM – ключевой ограничивающий фактор числа ядер в SMP серверах. Эффективность же массивно-параллельных и кластерных систем ограничена пропускной способностью сети передачи данных.

Во-вторых, обеспечение надёжности системы. Чем больше ядер, тем больше вероятность отказов (надёжность пропорциональна числу чипов в системе).

В-третьих, обеспечение программной поддержки. Software отстал от hardware. В настоящее время для системы с миллионом ядер нет ни эффективных операционных систем, ни эффективных языков программирования. Прикладного программного обеспечения крайне мало.

Необходимо создать ОС нового типа: не с временным, а с пространственным мультиплексированием; с симметричным доступом не только к памяти, но и к устройствам ввода/вывода; с новым многоядерным механизмом прерываний; с новым механизмом обеспечения отказоустойчивости по ядрам.

Последовательные программы теперь стали медленными программами! Ни SMP, ни OpenMP модели программирования не поддерживают потенциал сцеплённых ядер. Программирование необходимо изобрести заново.

Проблемы и средства параллельного программирования

1. Проблемы параллельного программирования

Традиционное последовательное программирование предполагает пошаговое выполнение команд программы, реализующей некоторый алгоритм. Выполнение программы начинается запуском определенной функции, например main(), с последующим выполнением четко определенной последовательности операций. Если программа поддерживает пользовательский ввод – для каждого события пользовательского ввода определяется своя последовательность обработки. Такие последовательные программы предсказуемы и относительно просты в разработке и отладке.

При создании параллельной программы можно использовать различные формы параллельной декомпозиции (декомпозиция задачи, декомпозиция данных, декомпозиция потока данных и т.д.) и различные шаблоны проектирования («параллелизм задач», «разделяй и властвуй», «геометрическая декомпозиция», «конвейерная обработка», «волновой фронт» и т.д.), но на уровне операционной системы все будет сведено к организации параллельного выполнения нескольких потоков в одном или нескольких процессах. Проблемы, возникающие при разработке параллельных приложений, относятся преимущественно к следующим типам.

- Передача данных между потоками. Основные способы: передача через разделяемую память, средства потоковой передачи, средства передачи сообщений.

- Синхронизация выполнения потоков: своевременный старт этапов алгоритма, корректный доступ к разделяемым ресурсам и т.д.

- Масштабируемость и распределение нагрузки: многопоточная программа может запускаться на вычислительных системах различных конфигураций, например, с одним центральным процессором (ЦП) или шестнадцатью. Сможет ли программа эффективно использовать имеющиеся 16 процессоров?

В данном разделе мы рассмотрим вопросы, связанные с синхронизацией: необходимость синхронизации, аппаратные возможности, позволяющие решать задачи синхронизации, типовые задачи синхронизации и их решения, часто встречающиеся проблемы и способы борьбы с ними.

2. Необходимость синхронизации

Предположим, имеется многопоточная программа, работающая с базой данных (БД). Функции, использующие БД построены одинаково и используют однотипные последовательности, включающие следующие три шага:

1. Считать запись из базы данных в локальный буфер потока.

2. Изменить значение записи.

3. Записать модифицированную запись из локального буфера потока в базу данных.

В случае последовательного обращения потоков к записям базы данных результат операции предсказуем. Однако если два потока попытаются одновременно обратиться к одной записи базы данных, мы можем получить непредсказуемый результат.

Сначала рассмотрим ситуацию, когда вычислительная система имеет один ЦП; предполагается, что операционная система (ОС) использует вытесняющий алгоритм планирования. В этом случае основа проблемы заключается в возможности передачи ЦП от одного потока к другому до завершения им операции с записью БД. На рис. 21 представлены три возможных варианта диаграммы выполнения потоков.

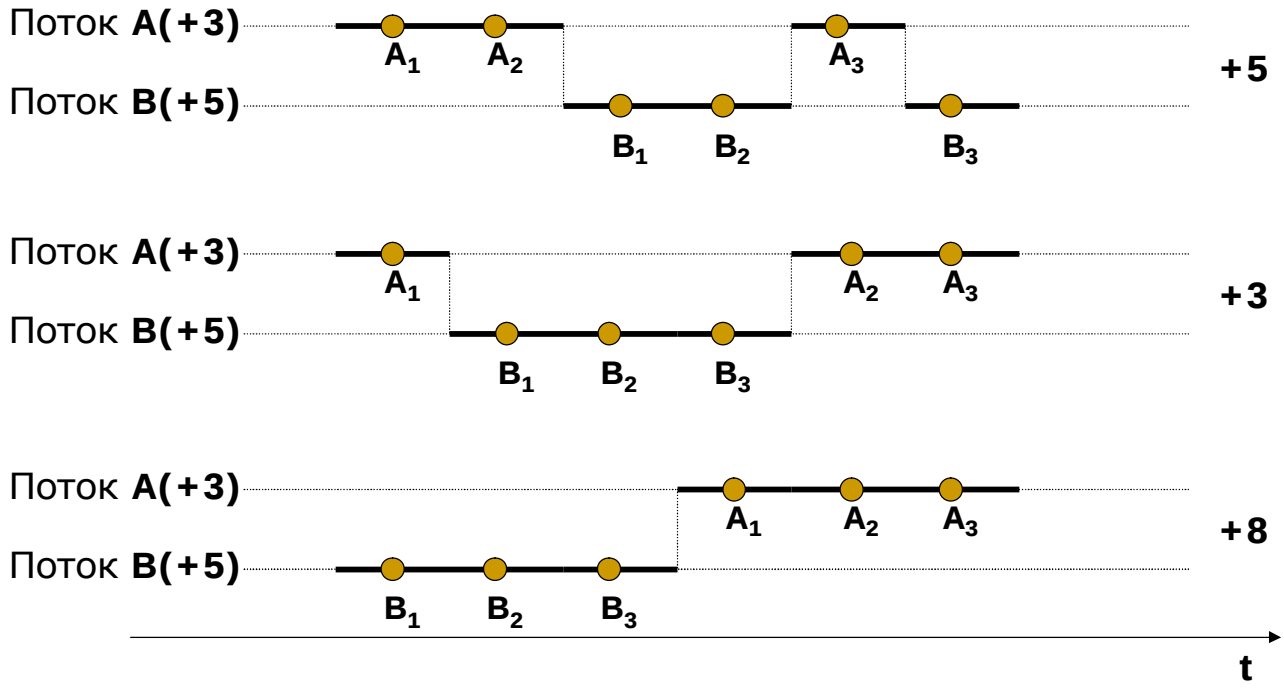

Рис. 21 Варианты диаграммы выполнения потоков на однопроцессорной системе

Если предположить, что поток A предполагает увеличить значение некоторого поля записи БД на 3, а поток B предполагает увеличить значение той же записи на 5, то мы имеем три возможных конечных значения данного поля записи. Какой именно

вариант реализуется при конкретном запуске программы, зависит от взаимных скоростей потоков и моментов передачи ЦП от одного потока к другому. Отметим, что в большинстве случаев поток не может повлиять на данные факторы.

В случае многопроцессорной системы также возможна реализация различных диаграмм выполнения потоков (см. рис. 22).

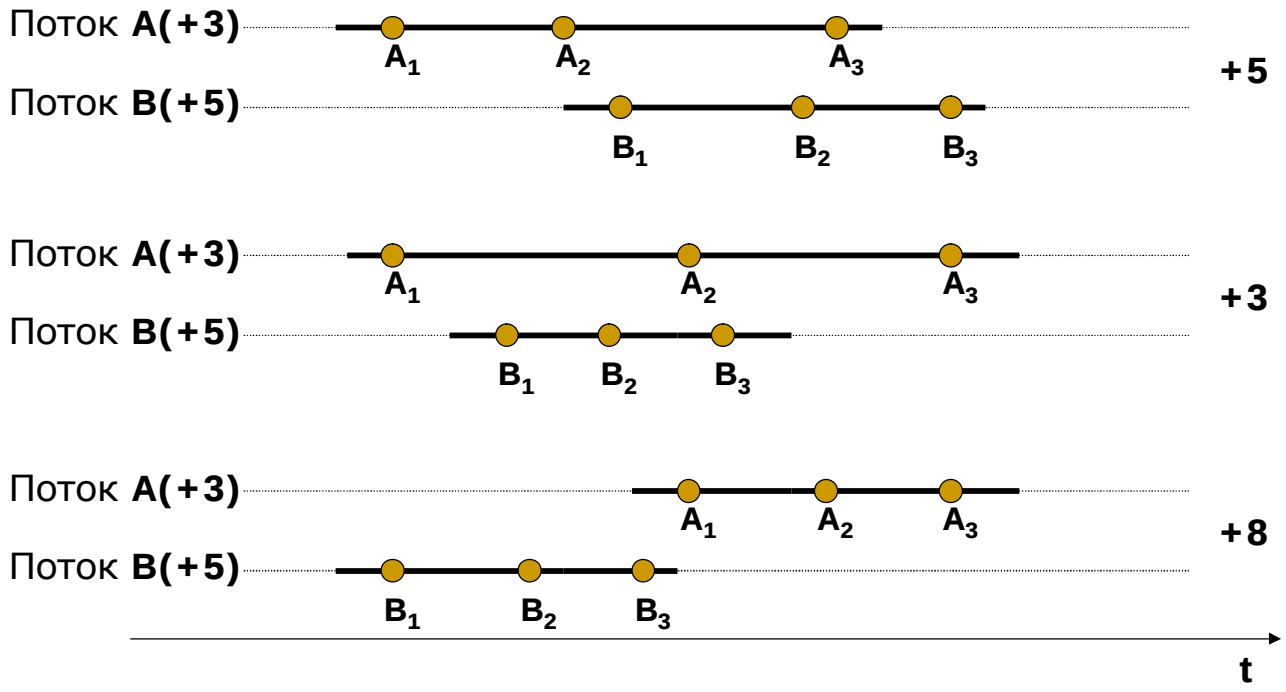

Рис. 22 Варианты диаграммы выполнения потоков на многопроцессорной системе

На данной диаграмме мы предполагаем, что потоки не вытеснялись планировщиком. В этом случае результат выполнения операции зависит от относительных скоростей потоков. Поток обычно никак не может предсказать или повлиять на свою относительную скорость: она зависит от состояния вычислительной системы (например, находятся ли его код или данные в кэш ЦП или нет) и ОС и процесса-владельца потока (например, часть страниц, используемых потоком, может находиться на жестком диске).

Таким образом, результат вычислений в рассмотренном примере не однозначен – все зависит от условий выполнения потоков, и разные запуски программы могут привести к разным результатам! Такая ситуация имеет место при совместном использовании любых ресурсов, в частности, при использовании "обычных" общих (глобальных и статических) переменных. Ситуация, когда два или более потоков используют разделяемый ресурс и конечный результат зависит от соотношения скоростей потоков, называется **состязанием** или **гонками** (race conditions).

Критическая секция (КС, critical section) – часть программы, результат выполнения которой может непредсказуемо меняться, если в ходе ее выполнения состояние ресурсов, используемых в этой части программы, изменяется другими потоками.

Критическая секция всегда определяется по отношению к определенным **критическим ресурсам** (например, критическим данным), при несогласованном доступе к которым могут возникнуть нежелательные эффекты.

Одна программа может содержать несколько критических секций, относящихся к одним и тем же данным. В нашем примере над записью БД могут быть определены операции пополнения счета и списания со счета (если это БД банковских счетов). При параллельном выполнении данных операций мы можем получить различные результаты.

Одна критическая секция может работать с различными критическими данными. В примере БД может содержать большое количество записей, каждая из которых может являться критическими данными. Отметим, что если несколько потоков одновременно работают с разными записями, никаких проблем нет – они возникают только при обращении нескольких потоков к одной и той же записи.

Для разрешения проблемы согласованного доступа к ресурсу необходимо использовать синхронизацию. Например, очевидно, что для исключения эффекта гонок по отношению к критическим ресурсам, необходимо обеспечить, чтобы в каждый момент времени в критической секции, связанной с этими ресурсами, находился только один поток. Все остальные потоки должны блокироваться на входе в критическую секцию. Когда один поток покидает критическую секцию, один из ожидающих потоков может в нее войти. Подобное требование обычно называется **взаимоисключением** (**mutual exclusion**).

Ниже приводится полная постановка задачи организации согласованного доступа к ресурсам.

3. Задача взаимного исключения

Постановка задачи взаимного исключения выглядит следующим образом: необходимо согласовать работу n>1 параллельных потоков при использовании некоторого критического ресурса таким образом, чтобы удовлетворить следующим требованиям:

1. одновременно внутри критической секции должно находиться не более одного потока;

2. относительные скорости развития потоков неизвестны и произвольны;

3. критические секции не должны иметь приоритета в отношении друг друга;

4. остановка какого-либо потока вне его критической секции не должна влиять на дальнейшую работу потоков по использованию критического ресурса;

5. любой поток может переходить в любое состояние, отличное от активного, вне пределов своей критической секции;

6. решение о вхождении потоков в их критические секции при одинаковом времени поступления запросов на такое вхождение и равноприоритетности потоков не откладывается на неопределенный срок, а является конечным во времени;

7. освобождение критического ресурса и выход из критической секции должны быть произведены потоком, использующим критический ресурс, за конечное время.

Решением задачи взаимного исключения является программа, которая удовлетворяет всем поставленным условиям при любой реализовавшейся последовательности выполнения потоков (на любой диаграмме выполнения потоков).

Необходимость выполнения условия (1) обсуждалось выше. Условие (2) определяет особенность среды выполнения потоков. Условие (3) говорит о том, что если имеются несколько критических секций, работающих с одним и тем же критическим ресурсом, то при поступлении от нескольких потоков запросов о входе в разные критические секции решение не должно основываться на том, в какую именно КС хочет войти тот или иной поток.

Условия (4) и (5) определяют, что код синхронизации должен быть локализован непосредственно около критической секции и используемые им ресурсы не должны использоваться в других частях программы.

Условия (6) и (7) указывают на типичные ошибки в решениях задачи взаимного исключения – для ряда предложенных решений можно предложить диаграмму выполнения потоков, при реализации которой принятие решения о входе в КС или осуществление выхода из КС могут выполняться в течение бесконечного времени.

Для решения задачи взаимного исключения можно использовать различные имеющиеся в распоряжении аппаратные возможности. Мы рассмотрим следующие

возможности: запрещение прерываний, использование разделяемых переменных, использование специальных команд ЦП.

4. Решения задачи взаимного исключения

4.1. Использование запрещения прерываний

На однопроцессорных системах проблема совместного использования ресурсов возникает по причине непредсказуемости момента передачи планировщиком ЦП от одного потока другому. Планировщик принимает решения о передачи в четко определенных случаях (создание и завершение потока или процесса и т.д.), при этом вытесняющая многозадачность, как правило, реализуется посредством принятия решения планирования при прерываниях таймера. Таким образом, можно предупредить вытеснение потока, запретив вытеснение по таймеру, например, запретив само прерывание от таймера. В таком случае, решение задачи взаимного исключения будет выглядеть следующим образом (внешний бесконечный цикл символизирует многократное использование критической секции потоком).

```
while( true ){
   DisableInterrupts();
   CS();                /*   Critical Section   */
   EnableInterrupts();
   NCS();               /* Non-Critical Section */
}
```

Несмотря на то, что в частном случае однопроцессорной машины данное решение является корректным, оно имеет несколько недостатков.

1. Прикладные программы, как правило, не могут запрещать прерывания. Эта операция разрешена только на максимальном уровне привилегий, на котором обычно выполняются только компоненты операционной системы и приравненные к ним модули (например, драйверы устройств).

2. На многопроцессорных системах прерывания запрещаются только на текущем процессоре, соответственно, несколько потоков, запущенных на различных ЦП, могут одновременно выполнять критическую секцию.

3. При запрещении прерываний на длительное время могут возникнуть сложности с функционированием устройств, не получавших должного внимания.

Таким образом, данное решение может применяться в очень ограниченном наборе ситуаций, в частности, прикладные программы в подавляющем большинстве современных ОС не могут его использовать.

4.2. Использование разделяемых переменных

Основная идея данного метода – использование некоторого набора разделяемых данных для согласования доступа к критическому ресурсу. Сначала рассмотрим несколько неправильных решений.

Решение 1.

```
// Разделяемые данные
bool flag0 = false;
bool flag1 = false;
```

```
// Код потока 0
while( true ){
  while( flag1 ) ;
  flag0 = true;
  CS0(); //Critical Section
  flag0 = false;
  NCS0();//NonCriticalSection
}
```

```
// Код потока 1
while( true ){
  while( flag0 ) ;
  flag1 = true;
  CS1(); //Critical Section
  flag1 = false;
  NCS1();//NonCriticalSection
}
```

`flag0` – `false`, если поток 0 выполняется вне критической секции, `true` – если поток 0 выполняет критическую секцию. `flag1` имеет тот же смысл для потока 1. Идея решения заключается в ожидании выхода потока-партнера из критической секции.

Для доказательства ошибочности решения необходимо привести контрпример, который в данном случае должен представлять такую диаграмму выполнения потоков, при реализации которой нарушается одно из условий задачи взаимного исключения. Контрпример для решения 1 приведен на рис. 23 (поток 0 обозначен A, поток 1 – B; числа означают номера строк кода внутри внешнего цикла while).

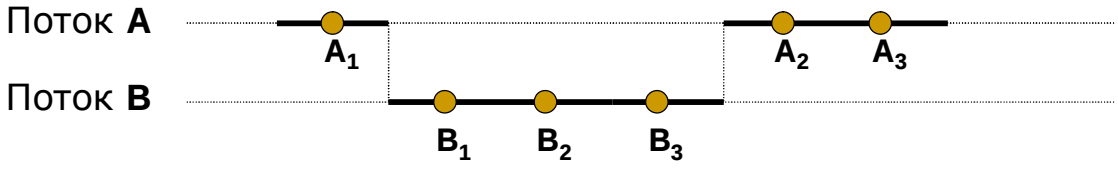

Рис. 23 Контрпример для решения 1 задачи взаимного исключения

B_3 соответствует выполнению потоком 1 его критической секции. Если считать, что выполнение B_3 было прервано до завершения КС, то в момент, когда поток 0 (Поток A) приступит к выполнению своей КС (A_3) мы получаем ситуацию, в которой два потока одновременно выполняются в критической секции, что противоречит условию 1 задачи взаимного исключения. Таким образом, мы доказали неверность данного решения и можем перейти к следующему.

Решение 2.

```
// Разделяемые данные
bool flag0 = false;
bool flag1 = false;
// Код потока 0
while( true ){
  flag0 = true;
  while( flag1 ) ;
  CS0();
  flag0 = false;
  NCS0();
}
```

```
// Код потока 1
while( true ){
  flag1 = true;
  while( flag0 ) ;
  CS1();
  flag1 = false;
  NCS1();
}
```

`flag0` – `false`, если поток 0 выполняется вне критической секции и не желает начать ее выполнение, `true` – если поток 0 либо выполняет критическую секцию, либо ожидает возможности в нее войти. `flag1` имеет тот же смысл для потока 1. Идея решения: поток заявляет о своем желании войти в критическую секцию, после чего ждет момента, когда это будет возможно.

Контрпример для данного решения приведен на Рис. 24

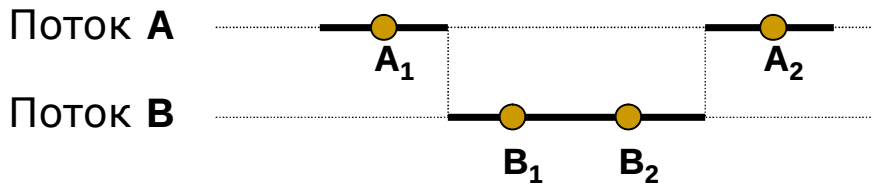

Рис. 24 Контрпример для решения 2 задачи взаимного исключения

К моменту, когда поток 0 начал выполнение строки A_2, значение обоих флагов установлено в `true`, поэтому потоки будут бесконечно выполнять циклы A_2 и B_2 соответственно. Таким образом, решение 2 неверно, поскольку не выполняется условие

(6) задачи взаимного исключения (решение о вхождении потоков в их критические секции ... является конечным во времени).

Решение 3.

```
// Разделяемые данные
bool flag0 = true;
bool flag1 = false;
// Код потока 0                    // Код потока 1
while( true ){                     while( true ){
  while( ! flag0 ) ;                 while( ! flag1 ) ;
  CS0();                             CS1();
  flag0 = false;                     flag1 = false;
  flag1 = true;                      flag0 = true;
  NCS0();                            NCS1();
}                                  }
```

`flag0` – `false`, если поток 1 разрешил потоку 0 выполнять критическую секцию. `flag1` имеет симметричный смысл для потока 1. Идея решения: поток выполняет критическую секцию, после чего разрешает выполнить критическую секцию потоку-партнеру.

Для данного решения нельзя построить диаграмму выполнения потоков, приводящую к недопустимому результату, однако в нем подразумевается поочередное выполнение критических секций потоками (что обычно не выполняется). Поток 0 не сможет выполнить свою критическую секцию в очередной раз, если до этого не выполнил КС поток 1. Таким образом, если потоку 0 необходимо дважды выполнить критическую секцию, а выполнение потока 1 было приостановлено вне критической секции, поток 0 будет ожидать повторной возможности входа в КС неопределенно долго, что нарушает условие (4) задачи взаимного исключения (остановка какого-либо потока вне его критической секции не должна влиять на дальнейшую работу потоков по использованию критического ресурса).

4.3. Алгоритмы Деккера и Петерсона

Решение Деккера основано на решении 2, но использует дополнительную переменную, упорядочивающую вход в критическую секцию при одновременном поступлении запросов на вхождение. Алгоритм Петерсона использует тот же самый принцип. Мы приведем именно его, поскольку его запись несколько проще.

Алгоритм Петерсона.

```
/* i – номер потока (0 или 1) */
int ready[2] = {0, 0};  /* Намерение войти в КС */
int turn = 0;           /* Приоритетный поток */
/* Код потоков */
while( true ) {
   ready[i] = 1;
   turn = 1 - i;
   while( ready[1-i] && turn == 1-i )
      ;
   CSi();
   ready[i] = 0;
   NCSi();
}
```

Для доказательства корректности данного решения необходимо рассмотреть все возможные диаграммы исполнения потоков и показать, что алгоритм в каждом случае работает правильно.

Существует обобщение алгоритма Деккера для N потоков – алгоритм булочной.

4.4. Использование специальных команд ЦП

Задача взаимного исключения была поставлено достаточно давно, и очень скоро была осознана ее важность и необходимость эффективного решения. Для поддержки решения были добавлены специальные команды в набор инструкций ЦП (различные в различны архитектурах). Мы рассмотрим решения, основанные на использовании команд Test&Set и Swap.

Принцип работы команды Test&Set может быть описан в виде следующей функции:

```
int Test_and_Set (int *target){
   int tmp = *target;
   *target = 1;
   return tmp;
}
```

Однако, поскольку Test&Set является одной командой ЦП, ее выполнение не может быть прервано, то есть управление не может быть передано другому потоку до ее

завершения. Благодаря этому, можно предложить следующее решение задачи взаимного исключения.

```
int lock = 0; /* Признак блокировки критического ресурса */
while( true ) {
   while( Test_and_Set( &lock ) )
      ;
   CSi();
   lock = 0;
   NCSi();
}
```

Решение основано на использовании для каждого критического ресурса специального признака блокировки (в примере – переменная `lock`). Значение lock = 0 указывает, что ресурс в настоящий момент не используется, значение lock = 1 – что ресурс свободен. Проверка условия работает корректно, поскольку невозможно прервать выполнение инструкции.

Решение с использованием команды swap использует тот же подход.

Описание работы команды swap в виде функции.

```
void Swap( int *a, int *b ){
   int tmp = *a;
   *a = *b;
   *b = tmp;
}
```

Решение задачи взаимного исключения с использованием команды swap.

```
int lock = 0; /* Признак блокировки критического ресурса */
int key;      /* Вспомогательная переменная */
while( true ) {
   key = 1;
   do{
      Swap( &lock, &key );
   }while (key);
   CSi();
   lock = 0;
   NCSi();
}
```

Использование других команд ЦП, добавленных для поддержки решения задачи взаимного исключения, основано на том же самом принципе: создается дополнительная переменная, выполняющая роль признака блокировки, и специализированная команда позволяет одновременно изменить значение данной переменной и проанализировать предыдущее значение.

Таким образом, решение задачи взаимного исключения принимает следующий вид (см. рис. 25).

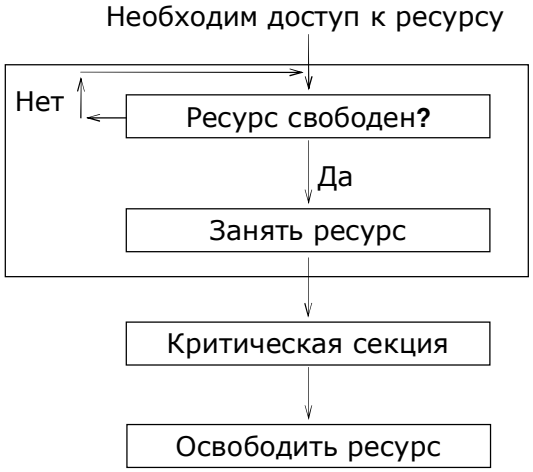

Рис. 25 Решение задачи взаимного исключения с использованием специализированных команд ЦП

При работе с признаком занятости (блокировки) ресурса поток использует "активное ожидания", то есть выполняет цикл, ожидая освобождения признака блокировки. Очевидно, существует множество случаев, когда данное решение неэффективно. Например, если при наличии одного процессора какой-то поток находится в состоянии активного ожидания, поток, захвативший признак блокировки, не может продолжать свое исполнение и никакой другой поток тоже не может. Таким образом, поток выполняет бесполезную работу до того момента, пока планировщик не вытеснит его с ЦП.

Данной проблемы можно избежать, если после выполнения каждой итерации цикла добровольно отдавать ЦП системе для передачи другому потоку (используя вызов `yield()` в UNIX, `Sleep()` в Win32); в этом случае при каждом получении ЦП поток выполняет только одну итерацию. Еще более эффективное решение инкапсулирует все операции над признаками блокировки в функции, и очередная итерация выполняется только при вызове функции, которая освобождает соответствующий признак блокировки.

Еще один важный момент заключается в том, что на уровне ЦП специализированные инструкции выполняются в виде последовательных операций на различных стадиях конвейера, то есть на уровне ЦП данные операции не являются атомарными. В случае однопроцессорной системы это не имеет значения, поскольку нельзя прервать выполнение потока в середине инструкции, а в многопроцессорной системе с общей памятью или в системе с NUMA-архитектурой (Non-Uniform Memory Access) при одновременном выполнении на двух процессорах специализированной команды для одного и того же признака блокировки мы получим тот же самый эффект гонок, который ранее наблюдали для потоков. Для решения данной проблемы существуют специальные команды или префиксы команд, которые позволяют временно блокировать доступ других процессоров к общей памяти.

Несмотря на затратность активного ожидание существуют ситуации, когда его использование в изначальном виде наиболее эффективно.

- Предположим, несколько потоков выполняются на многопроцессорной системе, и один из них установил признак блокировки и работает в критической секции. Если другому потоку, выполняющемуся на другом процессоре, потребовался тот же самый критический ресурс, и известно, что критические секции невелики, то вызов yield()/Sleep() может обслуживаться дольше, чем активное ожидание освобождения ресурса.

- Можно использовать активное ожидание в случае, если ожидается асинхронное изменение состояние признака блокировки. Например, если мы ожидаем прихода данных от внешнего устройства (сетевого контроллера, звукового контроллера и т.д.) и признак блокировки изменяется модулем, отвечающим за обслуживание событий от соответствующего устройства.

В прикладной программе можно самостоятельно реализовать механизм решения задачи взаимного исключения, используя один из приведенных алгоритмов. Однако обычно в операционной системе или специализированной библиотеки на основе низкоуровневых алгоритмов реализуются высокоуровневые средства синхронизации, которые и предоставляются прикладным программам.

5. Высокоуровневые механизмы синхронизации

Существует три высокоуровневых механизма синхронизации:

- Семафоры

- Мониторы

- Синхронные сообщения

Известно, что он взаимозаменяемые, то есть с помощью любого из них можно реализовать оба остальных.

5.1. Семафоры

Семафоры – примитивы синхронизации более высокого уровня абстракции, чем признаки блокировки; предложены Дийкстрой (Dijkstra) в 1968 г. в качестве компонента операционной системы THE.

Семафор – это целая неотрицательная переменная **sem**, для которой определены 2 атомарные операции: **P(sem)** и **V(sem)**.

- **P(sem)** (wait/down) – ожидает выполнения условия sem > 0, затем уменьшает sem на 1 и возвращает управление

- **V(sem)** (signal/up) увеличивает sem на 1

Атомарность операций обеспечивается на уровне реализации (посредством использования одного из алгоритмов, описанных выше). Возможны различные реализации семафоров, ниже мы предложим одну из них, в которой каждый семафор имеет очередь потоков, ожидающих увеличения его значения. В этом случае операции P и Vмогут быть реализованы следующим образом.

Определение структуры семафора.

```
typedef struct{
   int value;
   list<thread> L;
} semaphore;
```

При вызове потоком P(sem) если sem>0 (семафор "свободен"), его значение уменьшается на 1, и выполнение потока продолжается; если sem<=0 (семафор "занят"), поток переводится в состояние ожидания, помещается в очередь, соответствующую данному семафору, и запускается какой-либо другой готовый к выполнению поток.

```
P(S){
   S.value = S.value - 1;
   if( S.value < 0 ){
      add this thread to S.L;
      block;
```

```
}
```

При вызове потоком V(sem) если очередь потоков, ассоциированная с данным семафором, не пуста – один из потоков, ожидающих увеличения значения семафора, разблокируется и переводится в состояние "готов к выполнению"; поток, вызвавший V (sem), продолжает свое выполнение. В случае если нет потоков, ожидающих освобождения семафора, значение семафора увеличивается.

```
V(S){
   S.value = S.value + 1;
   if( S.value <= 0 ){
      remove a thread T from S.L;
      wakeup T;
}
```

Значение поля структуры семафора S.value в данной реализации может принимать отрицательные значения, их нужно интерпретировать следующим образом: значение семафора равно 0, число потоков, ожидающих увеличения значения семафора, равно |S|.

Семафоры обычно используются для организации согласованного доступа к критическим ресурсам. Можно выделить два вида семафоров.

1. Двоичный семафор может принимать значения 0 и 1, инициализируется значением 1. Используется для обеспечения эксклюзивного доступа к ресурсу (например, при работе в критической секции).

2. Счетный семафор может принимать значения от 0 до N и представляет ресурсы, состоящие из нескольких однородных элементов, где N – число единиц ресурса. Обычно инициализируется значением N и позволяет потокам исполняться, пока есть неиспользуемые элементы.

Семафоры первого вида часто оформляются в виде специального механизма синхронизации – мьютексов (mutex, от **mu**tual **exc**lusion). **Мьютекс** – двоичный семафор, обычно используемый для организации согласованного доступа к неделимому общему ресурсу. Мьютекс может принимать значения 1 (свободен) и 0 (занят). Над мьютексами определены следующие операции:

- acquire(mutex) – уменьшить (занять) мьютекс;

- release(mutex) – увеличить (освободить) мьютекс;

- tryacquire(mutex) – часто реализуемая неблокирующая операция, выполняющая попытку уменьшить (занять) мьютекс, и всегда сразу возвращающая управление.

Мьютексы в конкретных реализациях могут иметь дополнительные полезные свойства.

1. Запоминание потока-владельца.

Мьютекс, запоминающий владельца (то есть поток, успешно выполнивший операции acquire() или tryacquire()), освобождается только после вызова операции release() потоком-владельцем. Такие мьютексы удобно использовать для классической организации эксклюзивного доступа к разделяемому ресурсу, включающей следующие шаги: (1) захватить мьютекс, защищающий ресурс; (2) использовать ресурс; (3) освободить мьютекс, защищающий ресурс.

Мьютекс, не запоминающий поток-владелец, может быть освобожден другим потоком – это позволяет установить блокировку критического ресурса в одной части программы, для того чтобы использовать занятый ресурс позднее, возможно, в другом потоке. Использование таких мьютексов необходимо, например, при выполнении асинхронных операций, когда один поток подготавливает ресурсы для выполнения операции и инициирует ее, а обработку завершения операции осуществляет другой поток.

2. Рекурсивность.

Поток может многократно захватить рекурсивный мьютекс (вызывать aquire()); для освобождения мьютекса поток должен соответствующее число раз вызвать release(). Рекурсивные мьютексы удобны в ситуациях, когда имеется множество функций, работающих с одним и тем же критическим ресурсом, каждая функция выполняет последовательность операций "захватить мьютекс, защищающий ресурс", "использовать ресурс", "освободить мьютекс", и функции вызывают друг друга.

3. Наследование приоритета.

Предположим, имеются три потока: высокоприоритетный, среднеприоритетный и низкоприоритетный, и некоторый мьютекс захвачен низкоприоритетным потоком. Если высокоприоритетному потоку потребуется данный мьютекс, он выполнит вызов aquire() и перейдет в состояние ожидания. Центральный процессор перейдет среднеприоритетному потоку, который может выполняться в течение неограниченного времени, не предоставляя низкоприоритетному потоку возможности освободить занятый мьютескс. Таким образом, выполнение высокоприоритетного потока будет

зависеть от поведения среднеприоритетного потока – такая ситуация называется «инверсия приоритетов». Один из способов борьбы с данным явлением – наследование приоритета – поток, захвативший мьютекс, временно наследует максимальный из приоритетов потоков, ждущих освобождения данного мьютекса.

Позже мы рассмотрим примеры использования семафоров и мьютексов, а сейчас отметим, что семафоры и мьютексы представляют собой специальный вид разделяемых переменных, для которых отсутствует связь между ними и защищаемым критическим ресурсом. Это делает невозможным автоматическую поверку корректности использования семафоров и мьютексов (например, компилятором).

Объединение механизмов синхронизации с защищаемым ресурсом выполнение в мониторах.

5.2. Мониторы

Монитор – это конструкция языка программирования, поддерживающая управляемый доступ к разделяемым данным. Монитор инкапсулирует:

- разделяемые критические данные;

- функции, использующие разделяемые данные;

- синхронизацию выполнения параллельных потоков, вызывающих указанные функции.

Доступ к данным, расположенным в мониторе, реализуется только посредством вызова предоставленных функций. Только один поток может находиться в мониторе в любой момент времени, если второй поток пытается вызвать метод монитора, он переходит в состояние ожидания до выхода из монитора первого потока. Код синхронизации добавляется компилятором (см. рис. 26).

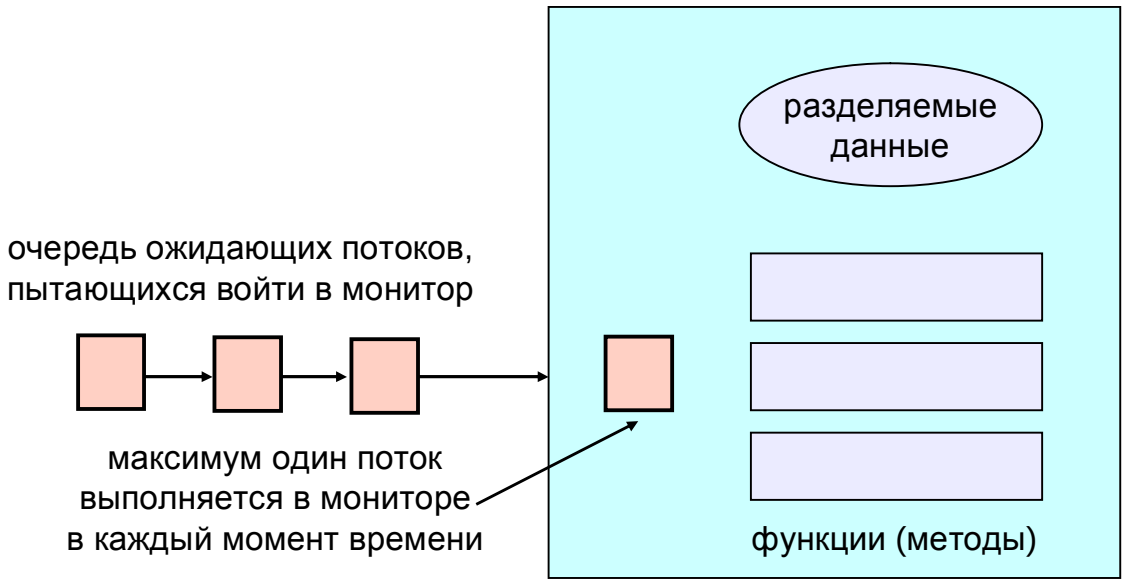

Рис. 26 Монитор

Таким образом, монитор по определению решает задачу взаимного исключения – достаточно для работы с критическими данными использовать только методы монитора.

Однако, возможны ситуации, в которых использование монитора приводит к некорректным результатам. Например, предположим, что разделяемыми данными является некоторый буфер, над которым определены операции добавления элемента и изъятия элемента. Изначально буфер пуст. Что произойдет, если первым будет выполнен запрос на изъятия элемента?

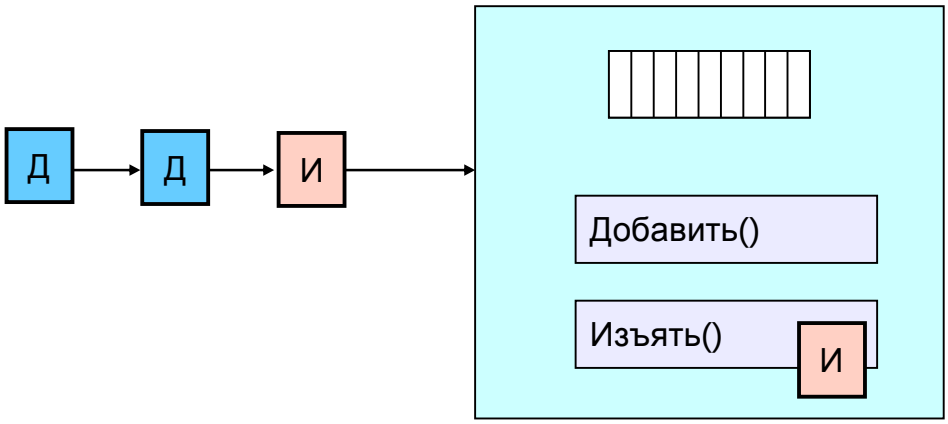

Рис. 27 Тупиковая ситуация при использовании монитора

Поток, выполнивший функцию изъятия, будет бесконечно ожидать появления данных, поскольку поток, добавляющий данные, не сможет войти в монитор.

Для преодоления подобных ситуаций вводятся условные переменные (conditional variable). Условная переменная символизируют ожидание потоком, выполняющим

метод монитора, наступления некоторого события (суть события не определяется средствами языка программирования). Над условными переменными определены три операции.

- wait(cv) – выполняется потоком, который хочет подождать наступления события. Снимает блокировку монитора, после чего другой поток может войти в него; ожидает, пока какой-либо другой поток не подаст условный сигнал.

- signal(cv) – выполняется потоком, сигнализирующем о наступлении события. Пробуждает максимум один ожидающий поток; если отсутствуют потоки, ожидающие события, информация о приходе сигнала теряется.

- broadcast(cv) – также выполняется потоком, сигнализирующем о наступлении события. Пробуждает все ожидающие события потоки.

В реализациях для условных переменных часто организуются очереди ожидающих их потоков. Кроме того, условные переменные часто реализуются как самостоятельный механизм синхронизации, а не составляющий элемент мониторов (Event в Win32, conditional variable библиотеки POSIX и т.д.)

Существует два типа мониторов, различным образом обрабатывающих поступление сигнала о произошедшем событии: мониторы Хоара и мониторы Меса.

Мониторы Хоара обрабатывают вызов signal(cv) следующим образом:

- немедленно запускается поток, ожидавший сигнала;

- поток, пославший сигнал, блокируется и остается блокированным все время, пока выполняется поток, которого он вывел из состояния ожидания.

Таким образом, поток, посылающий сигнал, должен предварительно перевести монитор в корректное и непротиворечивое состояние, а после отправления сигнала иметь в виду, что состояние монитора могло быть изменено потоком, разблокированным по его сигналу.

Мониторы Меса обрабатывают вызов signal(cv) несколько другим способом:

- ожидающий поток переводится в состояние "готов к выполнению", а поток, пославший сигнал, продолжает исполнение;

- ожидавший поток запускается при выходе потока, пославшего сигнал, из монитора или его перехода в состояние ожидания.

В мониторах Меса поток, пославший сигнал, может не восстанавливать состояние монитора вплоть до выхода из него. Однако если поток вышел из состояния ожидания

– это всего лишь означает, что произошли какие-то изменения: поскольку сигнализирующий поток продолжил исполнение после отправление сигнала, он мог изменить состояние монитора и нарушить условие ожидания. Таким образом, ожидавший поток должен опять проверить выполнение ожидаемых условий и при необходимости продолжить ожидание.

Если рассматривать участок кода, связанный с ожиданием выполнения события, то для разных типов мониторов он будет выглядеть следующим образом.

Монитор Хоара:

```
if( not Ready )
   wait(c);
```

Монитор Меса:

```
while( not Ready )
   wait(c);
```

Несмотря на то, что мониторы Хоара обеспечивают немедленный запуск потока, ожидающего наступления события и тем самым гарантируют выполнение ожидаемого условия, обычно реализуются мониторы Меса, поскольку они используют меньшее количество переключений контекста, проще в использовании и естественным образом расширяются на случай операции broadcast().

5.3. Синхронные сообщения

Использование сообщений – один из способов межпроцессного и межпоточного взаимодействия, позволяющий потокам подавать сигналы друг другу и обмениваться данными. Важным свойством механизма сообщений является возможность его использования для организации взаимодействия потоков, выполняющихся на различных узлах сети.

Обычно реализуются два типа доставки сообщений: синхронная и асинхронная. При асинхронной доставке поток, посылающий сообщение, инициирует процесс доставки сообщения, после чего продолжает свою работу (сообщение доставляется операционной системой параллельно деятельности потока). При синхронной доставке поток, пославший сообщение, дожидается подтверждения его получения принимающим потоком.

Сообщения позволяют организовать синхронизацию выполнения потоков. Например, пусть имеются разделяемые данные, над которыми требуется многократно

выполнить некоторую операцию и есть два потока, которые выполняют данную операцию над частями разделяемых данных. Предположим также, что каждое следующее выполнение операции требует, чтобы предыдущее было полностью завершено обоими потоками. В этом случае мы можем синхронизировать работу потоков следующим образом.

```
/* Код i-ого обработчика */
while( обработка_не_завершена ){
   <обработка части данных i-ого потока >
   SendMessage(1-i);   /*Отправить сообщение потоку 1-i*/
   ReceiveMessage(1-i);/*Получить сообщение от потока 1-i*/
}
```

В данном решении можно использовать как синхронные, так и асинхронные сообщения, а вот в следующем – только синхронные.

```
// Код потока 0
while(обработка_не_завершена){
   <обработка части данных 0>
   SendMessage(1);
}
```

```
// Код потока 1
while(обработка_не_завершена){
   <обработка части данных 1>
   ReceiveMessage(0);
}
```

Мы не будем рассматривать дополнительные примеры, в которых используется доставка сообщений. Несмотря на то, что использование данного механизма представляется интуитивно понятным, он обычно не используется при решении типовых задач синхронизации, рассмотрению которых посвящен следующий блок.

6. Типовые задачи синхронизации

В программировании не существует идеальных решений и алгоритмов на все случаи жизни, но имеются типовые задачи и варианты их решения. Мы рассмотрим несколько типовых задач синхронизации и их решения:

- «Производители-Потребители»
- «Читатели-Писатели»
- «Обедающие философы»
- «Спящий брадобрей»

6.1. Задача «Производители-потребители»

Одной из типовых задач, требующих синхронизации, является задача producer-consumer (производитель-потребитель). Пусть два потока обмениваются информацией через буфер ограниченного размера. Производитель добавляет информацию в буфер, а потребитель извлекает ее оттуда (см. рис. 28).

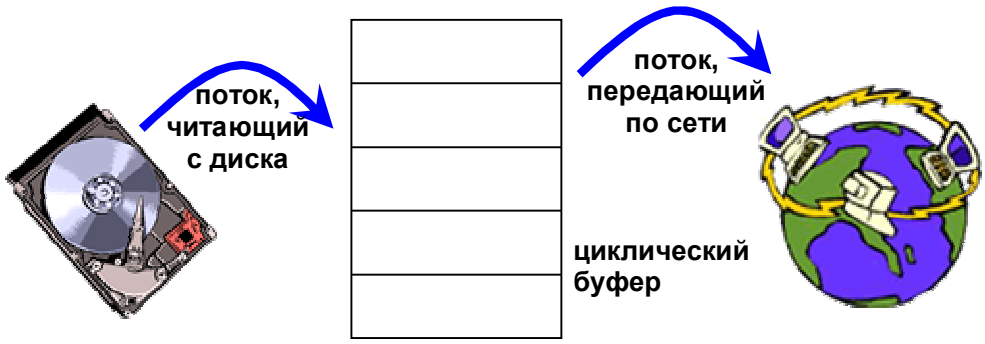

Рис. 28 Два потока, обменивающихся информацией через циклический буфер

Функциональность потоков производителя и потребителя можно записать следующим образом.

```
Producer:
  while(true) {
    PrepareData(); // Подготовить данные
    Put(Data);     // Поместить данные в циклический буфер
  }
Consumer:
  while(true) {
    Get(&Data);    // Считать данные из циклического буфера
    UseData();     // Использовать данные
  }
```

Сразу необходимо отметить, что если буфер пуст, то потребитель должен ждать, пока в нем появятся данные, а если буфер полон, то производитель должен ждать появления свободного элемента. В данном случае реализация производителя и потребителя будет иметь следующий вид.

```
Producer:
  while(true) {
    PrepareData();          // Подготовить данные
    while( ! Put(Data) )    // Разместить данные
```

```
      ;
  }
Consumer:
  while(true) {
    while( ! Get(&Data) )   // Считать данные
      ;
    UseData();              // Использовать данные
  }
```

Задача «Производители-Потребители» заключается в обеспечении согласованного доступа нескольких потоков к разделяемому циклическому буферу. Корректное решение должно удовлетворять следующим условиям:

- потоки выполняются параллельно;

- одновременно в критической секции, связанной с каждым критическим ресурсом, должно находиться не более одного потока;

- потоки должны завершить работу в течение конечного времени;

- потоки должны корректно использовать операции с циклическим буфером.

Предположим, что решение задачи использует реализацию циклического буфера, в которой Start указывает на ячейку, в которую будет положен следующий элемент, End – на ячейку с элементом, который будет выдан по следующему запросу, и всегда имеется, по крайней мере, одна неиспользуемая ячейка. Это позволяет избежать гонок между производителями и потребителями: например, производитель изменяет значение только переменной Start, причем если в момент изменения потребитель находился в своей критической секции (т.е. принимал решение о возможности изъять очередной элемент или изымал элемент), то изменение значения переменной Start производителем не может привести к принятию неверного решения потребителем или ошибке при изъятии элемента. В случае нескольких потоков, для потоков-производителей критическим ресурсом будет переменная Start, для потоков-потребителей – переменная End.

Решение данной задачи должно, во-первых, обеспечивать согласованный доступ к критическим данным, во-вторых, оповещение потребителей, ожидающих поступления данных, и производителей, ожидающих появления незаполненных записей в буфере.

Решение задачи "Производитель-Потребитель", использующее семафоры

```
Semaphore Access = 1;//управляет доступом к разделяемым данным
```

```
Semaphore Empty = n;  // количество пустых записей
Semaphore Full  = 0;  // количество заполненных записей
Producer(){
   P(Empty);  // ждем появления свободной записи
   P(Access);  // получаем доступ к указателям
   <записываем значение в запись>
   V(Access); // завершили работу с указателями
   V(Full);  // сигнализируем о появлении заполненной записи
}
Consumer(){
   P(Full);  // ждем появления заполненной записи
   P(Access); // получаем доступ к указателям
   <извлекаем данные из записи>
   V(Access); // завершили работу с указателями
   V(Empty); // сигнализируем о появлении свободной записи
   <используем данные из записи>
}
```

Двоичный семафор Access используется для организации согласованного доступа к критическим данным, счетные семафоры Full и Empty используются для сигнализации ожидающим потокам о наступлении ожидаемого события (появлении заполненной или пустой записи соответственно).

Решение задачи "Производитель-Потребитель", использующие мониторы

```
Monitor Bounded_buffer {
   buffer Resources[N];
   condition not_full, not_empty;
   Produce(resource x) {
      while( array “resources” is full )
         wait(not_full);
      <записываем значение “x” в запись массива “Resources”>
      signal(not_empty);
   }
   Consume(resource *x) {
      while( array “resources” is empty )
         wait(not_empty);
```

```
        *x = <считываем значение из массива “Resources”>
        signal(not_full);
    }
}
```

В данном решении для сигнализации используются условные переменные, а защита критических данных производится автоматически, поскольку они находятся внутри монитора.

6.2. Задача «Читатели-Писатели»

Вторая типовая задача – проблема читателей и писателей (Readers-Writers problem). Предположим, у нас есть хранилище данных, с которым работают несколько потоков. Потоки могут выполнять операции чтения и записи данных.

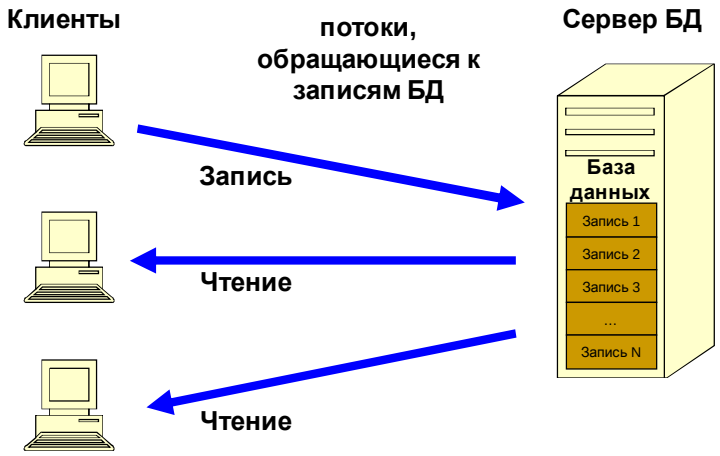

Рис. 29 Несколько потоков, обращающихся к базе данных

Функциональность потоков читателя и писателя можно записать следующим образом.

```
Writer:
  while(true) {
    PrepareData();  // Подготовить данные
    Write(Data);    // Записать данные
  }
Reader:
  while(true) {
    Read(&Data);    // Прочитать данные
    UseData();      // Использовать данные
  }
```

Если несколько потоков одновременно обращаются к разным записям базы данных, то никаких проблем не возникает. Проблема синхронизации доступа к данным возникает при обращении нескольких потоков к одной и той записи. К тому же, организация согласованного доступа для потоков-читателей и потоков писателей может отличаться, поскольку одновременное выполнение операции чтения несколькими потоками не приводит к возникновению проблем, в то время как операция записи требует эксклюзивного доступа.

Задача «Читатели-Писатели» заключается в обеспечении согласованного доступа нескольких потоков к разделяемым данным. Корректное решение должно удовлетворять следующим условиям:

- потоки выполняются параллельно;

- во время выполнения потоком операции записи, данные не могут использоваться другими потоками;

- во время выполнения потоком операции чтения, другие потоки также могут выполнять операцию чтения;

- потоки должны завершить работу в течение конечного времени.

Решение задачи "Писатели-Читатели", использующее семафоры

```
Semaphore RC = 1; //управляет доступом к переменной ReadCount
Semaphore Access = 1; //управляет доступом к данным
int ReadCount = 0; //количество "активных"(читающих) читателей
Writer(){
   P(Access);      // захватываем доступ к критическим данным
   <выполняем операцию записи>
   V(Access);      // освобождаем доступ к критическим данным
}
Reader(){
   P(RC);//получаем эксклюзивный доступ к переменной ReadCount
   ReadCount++;    // увеличиваем число активных читателей
   if( ReadCount == 1 )
      P(Access);   // захватываем доступ к критическим данным
   V(RC);// освобождаем доступ к переменной ReadCount
   <выполняем операцию чтения>
   P(RC);//получаем эксклюзивный доступ к переменной ReadCount
```

```
   ReadCount--;    // уменьшаем число активных читателей
   if( ReadCount == 0 )
      V(Access);   // освобождаем доступ к критическим данным
   V(RC);// освобождаем доступ к переменной ReadCount
}
```

Для защиты разделяемых критических данных используется двоичный семафор Access. Функция писателя просто получает доступ к критическим данным (уменьшая семафор Access) и использует их.

Функция читателя должна обеспечить одновременный доступ к критическим данным нескольких читателей, поэтому она использует дополнительную переменную ReadCount – количество читателей, выполняющих операцию чтения в настоящий момент. Данная переменная также требует защиты для корректного использования несколькими читателями (для этого введен двоичный семафор RC), но ее использование позволяет выполнять запрос на доступ к критическим данным только первому читателю (остальные будут заблокированы при попытке уменьшить семафор RC), а освобождение данных оставить последнему читателю, завершившему операцию чтения. Обратим внимание, что вследствие подобного подхода функция читателя содержит 3 критических секции: одну, связанную с чтением критических данных, и две, связанных с изменением переменной ReadCount и организации процесса входа и выхода из первой критической секции.

Отметим также, что если потоки-читатели постоянно входят в критическую секцию, возможно голодание (starvation) потоков-писателей. (Голодание – ситуация, когда поток не может приступить к выполнению своей задачи в течение неограниченного периода времени). Для решения этой проблемы можно использовать еще один семафор, с помощью которого организовать запрет доступа новых читателей в критическую секцию при наличии ожидающих писателей (мы не будем приводить данное решение).

6.3. Задача «Обедающие философы»

Предположим, имеется некоторая совокупность ресурсов, которые могут использоваться потоками, причем для решения конкретной задачи потоку требуется использовать несколько ресурсов одновременно. В этом случае мы имеем задачу обеспечения согласованного доступа к некоторому подмножеству имеющихся ресурсов. Задача об обедающих философах представляет частную постановку такой

задачи, имеющую ограничениями относительно общего случая, но допускающая расширения своего решения.

Оригинальная постановка задачи, предложенная Э. Дийкстрой, звучит следующим образом. На круглом столе расставлены тарелки, по одной на каждого философа. В центре стола – большое блюдо спагетти, а на столе лежат пять вилок — каждая между двумя соседними тарелками. Каждый философ находится только в двух состояниях — либо он размышляет, либо ест спагетти. Начать думать после еды философу ничто не мешает. Но чтобы начать есть, необходимо выполнить ряд условий. Предполагается, что любой философ, прежде чем начать есть, должен положить из общего блюда спагетти себе в тарелку. Для этого он одновременно должен держать в левой и правой руках по вилке, набрать спагетти в тарелку с их помощью и, не выпуская вилок из рук, начать есть. Закончив еду, философ кладет вилки слева и справа от своей тарелки и опять начинает размышлять до тех пор, пока снова не проголодается.

В качестве совместно используемых ресурсов в данном случае выступают вилки, а ограничение состоит в том, что каждая вилка используется только двумя сидящими рядом с ней философами.

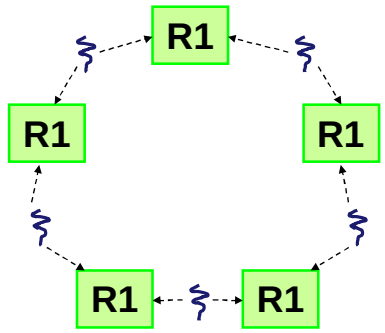

Рис. 30 Потоки, конкурирующие за множества ресурсов

Функциональность потоков-философов можно записать следующим образом.

```
Philosopher:
  while(true) {
    think();      // Размышляем
    take_forks(); // Берем 2 вилки
    eat();        // Обедаем
    put_forks(i); // Кладем на стол обе вилки
  }
```

Задача «Обедающие философы» заключается в обеспечении согласованного доступа нескольких потоков к разделяемым ресурсам. Корректное решение должно удовлетворять следующим условиям:

- потоки выполняются параллельно;

- во время использования «вилок» потоком данные «вилки» не должны использоваться его соседями;

- потоки должны завершить работу в течение конечного времени.

Решение задачи "Обедающие философы", использующее семафоры (предполагается количество философов, равное 5)

```
#define LEFT     (i+4)%5
#define RIGHT    (i+1)%5
#define THINKING 0    // Состояния философов
#define HUNGRY   1
#define EATING   2
int State[5];       //Массив состояний философов
Semaphore Access=1; //Управление доступом к критическим данным
Semaphore S[5]={0,0,0,0,0};//Для ожидания освобождения ресурсов
void Philosopher( int i ){  // i – номер философа от 0 до 4
  while( true ){
    think();       // Размышляем
    take_forks(i); // Берем 2 вилки или блокируемся
    eat();         // Обедаем
    put_forks(i);  // Кладем на стол обе вилки
  }
}
void take_forks( int i){
  P(Access); // Захватываем доступ к критическим данным
  State[i] = HUNGRY; // Изменяем свое состояние
  test(i);   // Пытаемся взять две вилки
  V(Access); // Освобождаем доступ к критическим данным
  P(S[i]);   // Блокируемся, если не удалось взять вилки
}
```

```
void put_forks( int i){
  P(Access);   // Захватываем доступ к критическим данным
  State[i] = THINKING; // Изменяем свое состояние
  test(LEFT);  // Пытаемся накормить соседа слева
  test(RIGHT); // Пытаемся накормить соседа справа
  V(Access);   // Освобождаем доступ к критическим данным
}
void test( int i ){
 if(state[i]==HUNGRY&&state[LEFT]!=EATING&&state[RIGHT]!=EATING){
   State[i] = EATING;  // Изменяем состояние
   V(S[i]);            // Разрешаем перейти к еде
 }
}
```

Семафор Access используется для защиты массива состояний философов State[5]. Семафоры S[5] используются для оповещения философа о возможности взять две вилки и начать обед. Отметим, что момент начала обеда определяется не философом, желающим обедать, а философом, завершившим обедать.

Можно предположить следующие типичные недостатки, присущие различным решениям задачи об обедающих философах:

- если все философы проголодались, взяли левую вилку и не желают ее отдавать, то они все будут ожидать правые вилки в течение бесконечного времени; предложенное решение не допускает возникновения подобной ситуации;

- если имеется голодный философ, то его непосредственные соседи могут по очереди блокировать его левую и правую вилку таким образом, что в каждый момент времени хотя бы одна из необходимых ему вилок занята – в результате возможно голодание отдельного философа; предложенное решение допускает возможность такого голодания.

Имеется простое расширение предложенного решения на случай, когда каждый поток может использовать произвольное множество ресурсов. Для этого, во-первых, необходимо добавить массив признаков занятости ресурсов и описания множеств ресурсов, затребованных каждым потоком, во-вторых, при освобождении ресурсов поток должен просматривать все потоки, ожидающие освобождения ресурсов, и в

случае, если запросы одного или нескольких потоков могут быть удовлетворены, резервировать за ними ресурсы и разрешать им продолжить выполнение.

Подобное решение будет лишено первого недостатка (по причине того, что все потоки требуют все необходимые им ресурсы сразу, а впоследствии их все возвращают), но будет подвержено второму, как и оригинальное решение.

7. Взаимоблокировка

Операционная система управляет многими видами ресурсов. Если процессам предоставляются исключительные права доступа к ресурсам, то процесс, получивший в свое распоряжение один ресурс и затребовавший другой, может ожидать предоставления второго ресурса в течение неопределенного времени.

Предположим, что работа процесса с ресурсами включает три стадии: получить/захватить ресурс (если ресурс свободен, он предоставляется процессу; если ресурс занят, процесс блокируется), использовать ресурс, освободить ресурс. Тогда при наличии двух процессов (A и B) и двух ресурсов (1-й и 2-й) возможна следующая нежелательная ситуация: процесс A захватил ресурс 1 и ожидает предоставления ему ресурса 2, процесс B захватил ресурс 2 и ожидает предоставления ему ресурса 1. Такая ситуация называется взаимоблокировкой или тупиком (deadlock).

Пример кода, который может привести к взаимоблокировке

```
Semaphore  Sem1 = 1;
Semaphore  Sem2 = 1;
Код процесса A:                  Код процесса B:
   P(Sem1);                         P(Sem2);
   P(Sem2);                         P(Sem1);
   /*использование ресурсов*/       /*использование ресурсов*/
   V(Sem2);                         V(Sem2);
   V(Sem1);                         V(Sem1);
```

Если после уменьшения семафора Sem1 потоком процесса A центральный процессор будет передан потоку процесса B, который уменьшит семафор Sem2, мы получим взаимоблокировку.

В общем случае, множество процессов находится в тупиковой ситуации, если каждый процесс ожидает некоторого события, которое может быть вызвано только действиями другого процесса из данного множества.

Процессы могут ожидать событий различного типа, например: завершение другого процесса, приход сообщения из другого процесса, освобождение ресурса. Мы будем рассматривать взаимоблокировки ресурсного типа как наиболее часто встречающиеся, к тому же рассмотренные подходы могут быть распространены на взаимоблокировки других типов.

Далее изложены основные аспекты проблемы взаимоблокировки: обнаружение, предотвращение, избегание и восстановление после возникновения.

7.1. Обнаружение взаимоблокировки

Для обнаружения блокировки ресурсного типа можно использовать граф «Процесс-ресурс». Направленный граф "Процесс-ресурс" включает:

- Множество вершин $V = P \cup R$, где $P=\{P_1,P_2,\ldots,P_N\}$ – множество процессов, $R=\{R_1,R_2,\ldots,R_M\}$ – множество ресурсов;

- Дуги запросов – направлены от процессов к ресурсам, дуга $P_i \rightarrow Rj$ означает, что процесс P_i запросил ресурс R_j;

- Дуги распределения – направлены от ресурсов к процессам, дуга $R_j \rightarrow P_i$ означает, что ресурс R_j выделен процессу P_i.

Если граф "процесс-ресурс" не имеет циклов, тупиков нет. Если граф "процесс-ресурс" имеет цикл, возможно, тупик существует

Над графом «Процесс ресурс» можно определить операцию редукции: если все запросы какого-либо процесса могут быть удовлетворены, он может быть редуцирован. При редукции все ресурсы, выделенные процессу, освобождаются.

Для операции редукции доказаны следующие утверждения:

- если граф полностью редуцируем, взаимоблокировка отсутствует;

- порядок выполнения редукции не имеет значения,

что позволяет разработать простой алгоритм обнаружения взаимоблокировки.

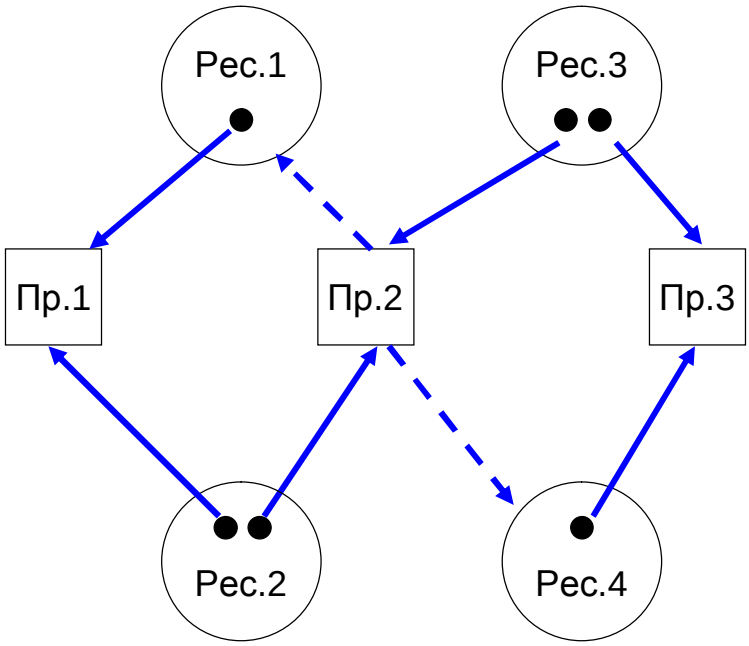

Рис. 31 Пример графа «Процесс-ресурс»

На рис. 31 представлен пример полностью редуцируемого графа «Процесс-ресурс». Последовательность редукции: Пр.3, Пр.2, Пр.1.

7.2. Предотвращение взаимоблокировок

В 1971 г. Е. Коффман, М. Элфик и А. Шошани сформулировали необходимые условия возникновения взаимоблокировки.

1. Mutual Exclusion (Взаимное исключение) – по крайней мере один из запрашиваемых ресурсов является неделимым (то есть должен захватываться в эксклюзивное использование).

2. Hold and wait (Удержание ресурсов при ожидании) – существует процесс, владеющий некоторым ресурсом и ожидающий освобождения другого ресурса.

3. No preemption (Неперераспределяемость ресурсов) – ресурсы не могут быть отобраны у процесса без его желания.

4. Circular wait (Циклическое ожидание) – существует такое множество процессов $\{P_1, P_2,\ldots, P_N\}$, в котором P_1 ждет освобождения ресурса процессом P_2, P_2 ждет $P_3,\ldots, P_N$ ждет P_1.

Способы предотвращения тупиков основаны на атаке одного из этих условий.

7.2.1. Устранения условия "Mutual exclusion"

Устранение условия взаимного исключения можно достигнуть, построив над ресурсом абстракцию, позволяющую использовать ресурс нескольким процессам одновременно. Например, над ресурсом «принтер» часто строится абстракция "очередь печати", позволяющая выполнять печать нескольким процессам одновременно.

К сожалению, это далеко не всегда возможно и требует дополнительных ресурсов.

7.2.2. Устранения условия "No preemption"

Для устранения условия неперераспределяемости ресурсов требуется обеспечить работу с состояниями ресурсов и контекстами работы процесса с ресурсом, поскольку при перераспределении ресурса придется выполнять следующую последовательность операций:

- запоминание контекста работы процесса с ресурсом и состояния ресурса;

- передача ресурса другому процессу;

- возврат ресурса процессу, восстановление состояния ресурса и контекста работы процесса с ресурсом.

Для каких-то ресурсов это возможно (например, для ЦП при передаче его от потока потоку), для других – нет (например, для принтера). К тому же процесс может использовать несколько ресурсов совместно, рассчитывая на то, что эксклюзивное владение одним из ресурсов автоматически означает эксклюзивное владение другими ресурсами. Например, если один из потоков процесса захватил мьютекс, он считает, что может эксклюзивно пользоваться ресурсом, защищаемым данным мьютексом. Поскольку соответствие между мьютексом и защищаемым им объектом заключено исключительно в логике программы и не поддается автоматическому определению, передача мьютекса от одного потока/процесса другому не имеет смысла, поскольку в момент передачи защищаемый объект может находиться в некорректном/неустойчивом состоянии и не может быть использован другим процессов/потоком.

7.2.3. Устранение условия Hold&Wait

Исключение условия удержания и ожидания – задача прикладных программистов. Со стороны операционной системы требуется только поддержка специальных модификаций или специальных параметров для функций запроса ресурсов, который

позволяют не ожидать освобождения ресурса бесконечно, а завершить ожидание через некоторое время (либо сразу) при невозможности получения ресурса.

Существует несколько возможных подходов к устранению условия Hold&Wait:

1. Можно запрашивать все необходимые ресурсы до начала выполнения.

Во-первых, обычно процессы точно не знают, какие ресурсы им понадобятся в ходе работы, во-вторых, возможно голодание при ожидании популярных ресурсов. К тому же, возможно неэффективное использование ресурсов, если часть ресурсов нужна процессу на короткое время.

2. При возникновении потребности в дополнительных ресурсах – освобождаем все уже имеющиеся, затем запрашиваем те, что необходимы в настоящий момент.

При использовании данного подхода также возможно голодание и имеет место неэффективное использование ресурсов.

7.2.4. Устранение условия Circular wait

Для устранения условия циклического ожидания можно использовать специальные правила выделения ресурсов.

1. Можно позволить процессам владеть только одним ресурсом в каждый момент времени.

В принципе, такой подход решает проблему тупиков, но на практике его вряд ли возможно использовать.

2. Можно ввести для ресурсов нумерацию и обязать процессы запрашивать ресурсы строго в порядке возрастания их номеров.

Данный подход основан на том факте, что в цикле графа «Процесс-ресурс» всегда есть процесс, у которого номер имеющегося ресурса больше номера запрошенного. Однако, нумерация не всегда возможна (например, если имеются счетные ресурсы и число ресурсов изменяется), к тому же мы опять имеем неэффективное использование ресурсов

7.3. Избегание взаимоблокировок

Избегание основано на использовании специальных алгоритмов распределения ресурсов, не допускающих возникновения взаимоблокировки. Подобные алгоритмы требуют информации о максимальном количестве ресурсов, которое может потребоваться каждому процессу. Мы рассмотрим алгоритм банкира.

Состояние называется **безопасным**, если для него имеется такая последовательность процессов {P_1, P_2,…, P_n}, что для каждого P_i ресурсы, которые затребовал P_i, могут быть предоставлены за счет имеющихся незанятых ресурсов и ресурсов, выделенных всем процессам P_j, где $j < i$.

Состояние безопасно, поскольку ОС может гарантированно избежать тупика посредством блокирования любых новых запросов, пока не выполнится безопасная последовательность.

Алгоритм банкира формулируется следующим образом.

При поступлении запроса на выделение ресурса проверяем, является ли состояние, в которое мы перейдем после выделения, безопасным. Если новое состояние безопасно – выделяем ресурс, если новое состояние небезопасно – ресурс не выделяем, блокируем процесс, выполнивший запрос.

Рассмотрим пример.

Предположим, имеются 12 устройств хранения.

Процесс	MAX потребность	Владеет	Может запросить
p0	10	5	5
p1	4	2	2
p2	9	2	7

3 устройства свободны

Текущее состояние безопасно, поскольку существует безопасная последовательность: <p_1,p_0,p_2>: p1 может завершиться, используя только свободные ресурсы, p_0 может завершиться, используя свободные ресурсы и ресурсы, которые сейчас выделены процессу p_1, p_2 может завершиться, используя свободные ресурсы и ресурсы, которые сейчас выделены процессам p_1 и p_0.

Если p_1 запросит еще 1 устройство, удовлетворение запроса будет удовлетворено, поскольку после система останется в безопасном состоянии.

Если p_2 запросит еще 1 устройство, удовлетворение запроса будет отложено, поскольку оно переведет систему в небезопасное состояние.

Алгоритм банкира можно проиллюстрировать, используя граф "Процесс-ресурс". При выполнении запроса:

- представляем, что мы его удовлетворили;

- представляем, что все остальные возможные запросы выполнены;

- проверяем, может ли граф "процесс-ресурс" быть полностью редуцирован; если да – выделяем запрошенный ресурс, если нет – блокируем процесс, выполнивший запрос.

7.3.1. Восстановление после взаимоблокировки

Для восстановления после тупика его сначала необходимо обнаружить. Для этого нужно отслеживать выделение ресурсов и поступающие запросы. Поскольку операция проверки наличия взаимоблокировки является достаточно длительной, ее выполнение при каждом запросе потребует слишком большого количества ресурсов. Поэтому алгоритм обнаружения запускают при каждом N-ом запросе или через некоторый временной интервал.

Можно использовать следующие способы восстановления после взаимоблокировки:

1. Уничтожение одного/всех процессов, участвующих в тупике.

Обычно процессы уничтожаются до тех пор, пока взаимоблокировка не распадется. Все вычисления уничтоженных процессов придется повторить. Это грубый, но эффективный метод.

Еще одна модификация данного подхода может использоваться для малых систем: если система не имеет существенного собственного состояния (то есть состояние системы может быть восстановлено после перезагрузки без необходимости сохранять какие-то данные), то при обнаружении взаимоблокировки система перезагружается.

2. Откат выбранного процесса к некоторой контрольной точке или к началу (partial or total rollback)

Если программа предназначена для выполнения длительных вычислений, то разработчик может предусмотреть возможность периодического сохранения состояния программы (в контрольных точках). В этом случае при возникновении взаимоблокировки с участием процесса, выполняющего данную программу, ее исполнение может быть прервано и начато с последнего сохраненного состояния.

Данный подход не гарантирует, что мы не попадем в ситуацию взаимоблокировки при повторном исполнении, однако непредсказуемость относительных скоростей выполнения потоков позволяет предположить такую возможность.

8. Заключение

Широкое распространение многопроцессорных и многоядерных систем требует от программистов перехода от разработки последовательных программ к разработке параллельных, что ставит перед ними множество новых проблем. Ключевой момент в разработке и отладке параллельных программ – понимание реализации параллельного выполнения потоков и определения участков программы, требующих синхронизации.

Можно использовать стандартные средства синхронизации, предоставляемые операционной системой или некоторой библиотекой (семафоры, мониторы, сообщения), можно выполнить собственные реализации механизмов реализации для достижения максимальной производительности. Но программирование параллельных приложений должно быть максимально аккуратным, поскольку неумелое использование средств синхронизации – источник трудно находимых ошибок и потерь производительности.

Литература

1. John L. Hennessy and David A. Patterson, Computer Architecture: A Quantitative Approach, 3rd Ed., Morgan Kaufmann, 2003

2. Randal E. Bryant and David R. O'Hallaron, Computer Systems: A Programmer's Perspective, Prentice Hall, Pearson Education, 2003

3. William Stallings, Computer Organization and Architecture: Designing and Performance, 5th Ed., Prentice Hall, 2000

4. Воеводин В.В., Воеводин Вл.В., Параллельные вычисления, СПб: БХВ-Петербург, 2004

5. Tullsen D.M., Eggers S.J., Levy H.M., Simultaneous multithreading: Maximizing on-chip parallelism, Proc. 22nd International Symposium on Computer Architecture, 1995

6. Tullsen D.M., Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor, Proc. 23rd International Symposium on Computer Architecture, 1996

7. Shalf J., The Landscape of Computer Architecture, ISC2007, Dresden, 2007

8. Sterling T., HPC Achievement and Impact, ISC2007, Dresden, 2007

9. Burton S., Reinventing Computing, ISC2007, Dresden, 2007

10. Sh. Akhter, J. Roberts. Multi-Core Programming. Increasing Performance through Software Multithreading. – IntelPress, 2006.

11. A. Silberschatz, P. B. Galvin, G. Gagne. Operating Systems Concepts. Seventh edition. – John Wiley & Sons, Inc. 2005.

12. Таненбаум Э. Современные операционные системы. 2-е изд. – СПб.: Питер, 2002.

13. Карпов В.Е., Коньков К.А. Введение в операционные системы. Курс лекций. 2-е изд. – М.: ИНТУИТ.РУ, 2005.

14. Рихтер Дж. Windows для профессионалов (Создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows). 4-е изд. – М.: Русская Редакция; пер. с англ. – СПб.: Питер, 2001.