Операционные системы для многопроцессорных и многомашинных систем. Мультипроцессоры. Часть 1

Лекция

Ревизия: 0.2

Последняя модификация: Vlad Kovtun

Дата последней модификации: 03.04.2015 14:45:00

История изменений

17.04.2010 – Версия 0.1. Первичный документ. Ковтун В.Ю.

10.08.2014 – Версия 0.2. Изменены рисунки. Ковтун В.Ю.

Содержание

Лекция 8. Операционные системы для многопроцессорных и многомашинных систем. Мультипроцессоры. Часть 1

Вопросы

1. Введение.
2. Многопроцессорные системы.
3. OpenMP API

Введение

Постоянная потребность человечества ко все большим вычислительным ресурсам, подвигает инженеров и ученых к различным подходам к увеличении производительности у современных компьютеров, а также искать нестандартные решения.

Далее будет рассматриваться несколько подходов (стандартных) к построению высокопроизводительных вычислительных систем посредством распараллеливания:

- Создание многопроцессорных вычислительных систем.
- Создание многомашинных вычислительных систем (кластеров).
- Локальные и глобальные распределенные вычислительные системы.

Весь обмен информацией между электронными (или оптическими) компонентами сводится, в конечном итоге, к отправке и приему сообщений, представляющих собой строго определенные последовательности битов. Различия состоят во временных параметрах, пространственных масштабах и логической организации. Одну крайность составляют мультипроцессорные системы с общей оперативной памятью и с числом процессоров от двух и более.

В этой модели каждый CPU обладает равным доступом ко всей физической памяти и может читать и писать отдельные слова. Время доступа к памяти обычно составляет от 10 до 50 не. Хотя такая система, показанная на Рис. 1(а), может показаться простой, ее реализация представляет собой далеко не простую задачу и обычно включает, большое количество скрытно передаваемых сообщений.

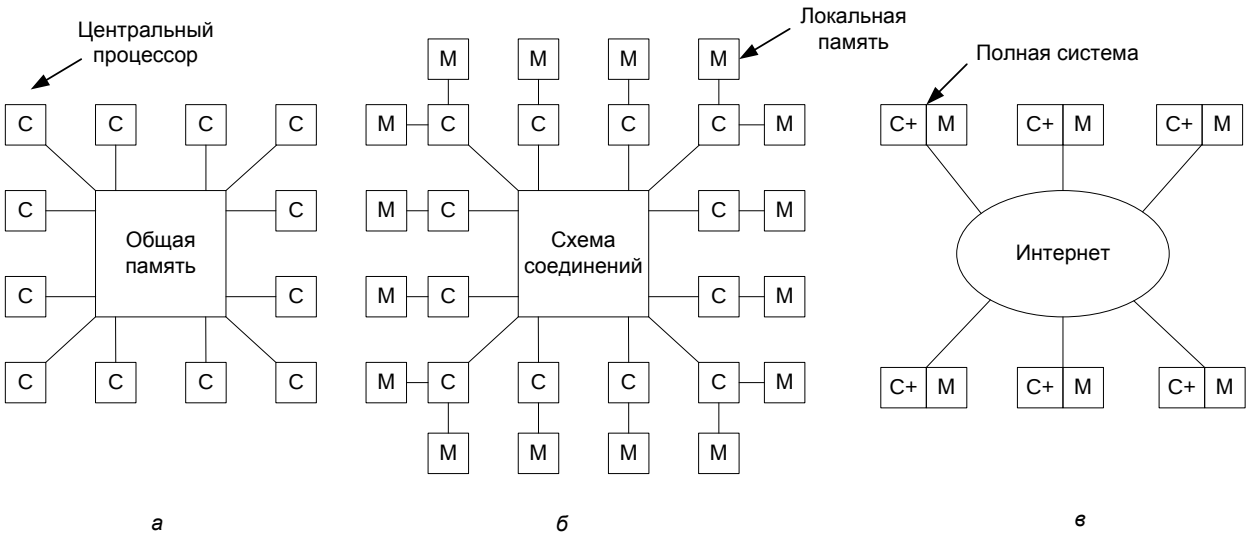


Рис. 1. Мультипроцессорная система с общей памятью (а); мультипроцессорная система с передачей сообщений (б); глобальная распределенная система (а)

Следом идут системы, Рис. 1(б), в которых пары, состоящие из CPU и памяти, соединены высокоскоростной соединительной схемой. Такая разновидность системы называется **мультипроцессорной системой с передачей сообщений**. Каждый блок памяти является локальным для одного CPU, и доступ к ней может получить только этот CPU. CPU общаются, обмениваясь сообщениями посредством системы передачи данных. При хорошем соединении, для передачи сообщения может потребоваться от 10 до 50 мкс, что все же значительно больше, чем время доступа к памяти в схеме на Рис. 1(а). В этой схеме нет общей глобальной памяти. **Мультикомпьютеры** - системы с передачей сообщений гораздо легче создать, чем мультипроцессоры - системы с общей

памятью, но писать программы для них значительно труднее. Поэтому у каждого подхода есть свои поклонники.

Третья модель, показанная на Рис. 1(в), представляет большое количество полноценных компьютеров, соединенных глобальной сетью, такой как Интернет, и образующих вместе **распределенную систему**. У каждого компьютера есть своя собственная память. Компьютеры в распределенной системе общаются, обмениваясь друг с другом сообщениями. Основное различие схем на рис. 1(б) и рис. 1(в) заключается в том, что во втором случае используются полноценные компьютеры, а время передачи сообщений составляет от 10 до 50 мс, то есть примерно еще в 1000 раз больше. Из-за большей величины задержки применение этих **слабосвязанных систем** отличается от использования **сильносвязанных систем**, показанных на Рис. 1(б). Величина задержки у всех трех типов систем отличается друг от друга на три десятичных порядка.

Мультипроцессоры

Мультипроцессор с общей памятью (или просто **мультипроцессор**) представляет собой вычислительную систему, в которой два или более CPU делят полный доступ к общей ОЗУ. Программа, работающая на любом CPU, видит нормальное (обычно разбитое на страницы) виртуальное адресное пространство. Единственное **необычное свойство** такой системы заключается в том, что CPU может записать какое-либо значение в память, а затем, считав это слово снова, получить другое значение (потому что другой CPU изменил его). При правильной организации это свойство формирует **основу межпроцессорного обмена информацией**: один CPU пишет данные в память, а другой считывает их оттуда. По большей части мультипроцессорные ОС представляют собой просто обычные ОС. Они обрабатывают системные вызовы, управляют памятью, предоставляют службы файловой системы и управляют устройствами ввода-вывода. Тем не менее, есть области, в которых они обладают уникальными свойствами. К этим областям относятся:

- синхронизация процессов;
- управление ресурсами;
- планирование.

Мультипроцессорное аппаратное обеспечение

У всех мультипроцессоров каждый CPU может адресоваться ко всей памяти. Однако по характеру доступа к памяти эти машины делятся на два класса:

- Мультипроцессоры, у которых каждое слово данных может быть считано с одинаковой скоростью, называются **UMA-мультипроцессорами** (Uniform Memory Access — однородный доступ к памяти).
- В противоположность им **NUMA-мультипроцессоры** (NonUniform Memory Access неоднородный доступ к памяти) этим свойством не обладают.

Почему существует такое различие, станет ясно позднее. Сначала будут описаны мультипроцессоры UMA, а затем — мультипроцессоры NUMA.

Архитектура симметричных мультипроцессоров UMA с общей шиной

В основе простейшей архитектуры мультипроцессоров **лежит идея общей шины**, Рис. 2(а). Несколько CPU и несколько модулей памяти одновременно используют одну и ту же шину для общения друг с другом. Когда CPU хочет прочитать слово в памяти, он сначала проверяет, свободна ли шина. Если шина свободна, CPU выставляет на нее адрес нужного ему слова, подает несколько управляющих сигналов и ждет, пока память не выставит нужное слово на шину данных.

Если шина занята, CPU просто ждет, пока она не освободится. В этом заключается проблема данной архитектуры. При двух или трех CPU состязанием за шину можно управлять. При 32 или 64 CPU шина будет постоянно занята, а производительность системы будет полностью ограничена пропускной способностью шины. При этом большую часть времени CPU будут простаивать.

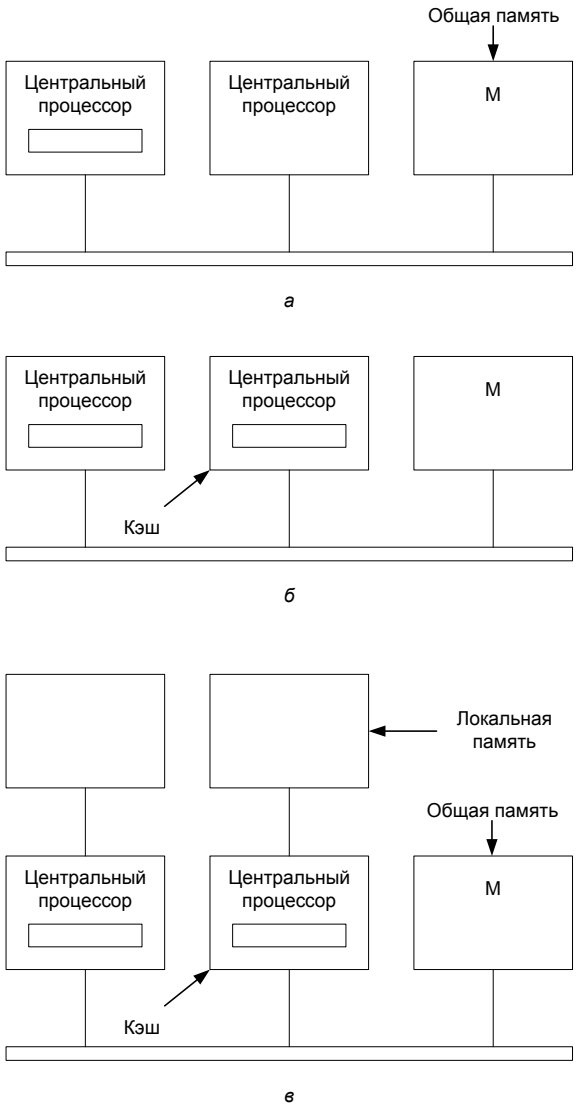


Рис. 2. Три варианта архитектуры мультипроцессоров с общей шиной: без кэша (а); с кэшем (б); с кэшем и собственной памятью (в)

Решение этой проблемы состоит в том, чтобы **добавить каждому CPU кэш**, как показано на Рис. 2(б). Кэш может располагаться внутри микросхемы CPU или рядом с CPU, на процессорной плате. Поскольку большое количество обращений к памяти теперь может быть удовлетворено прямо из кэша, обращений к шине будет существенно меньше, и система сможет поддерживать большее число CPU. Как правило, кэширование выполняется не для отдельных слов, а для блоков по 32 или по 64 байта. При обращении к слову весь блок считывается в кэш CPU, обратившегося к слову.

Для каждого блока кэша устанавливается режим доступа: либо для него разрешается только чтение (в этом случае этот блок может одновременно присутствовать в нескольких кэшах), либо разрешается и чтение, и запись (в этом случае этот блок не может одновременно присутствовать в нескольких кэшах). При попытке записи CPU слова, находящегося в одном или нескольких удаленных кэшах, аппаратура шины выставляет на шину специальный сигнал, информирующий остальные кэши о записи. Если в остальных кэшах соответствующие блоки «чистые», то есть модифицированные точные копии блока, находящегося в памяти, тогда они могут просто отбросить свои копии и позволить пишущему CPU получить этот блок из памяти. Если же в каком-либо кэше содержится «грязная» (то есть модифицированная) копия, она должна быть либо записана в память, прежде чем операция записи может быть продолжена, либо передана напрямую пишущему CPU по шине. Существует много протоколов обмена данных между кэшами и памятью.

Еще один вариант архитектуры мультипроцессоров представлен на Рис. 2(в). В этом случае у каждого CPU имеется не только кэш, но также и локальная собственная память, с которой он соединен по выделенной (индивидуальной) шине. Для оптимального использования подобной конфигурации компилятор должен поместить

текст программы, константы, стеки (то есть все неизменяемые данные), а также локальные переменные в локальные модули памяти. При этом общая память используется только для общих модифицируемых переменных. В большинстве случаев такая схема использования памяти сильно снижает трафик по шине, но для ее реализации требуются специальные действия со стороны компилятора.

Мультипроцессоры UMA с координатными коммутаторами

Даже при оптимальном использовании кэша наличие всего одной общей шины ограничивает число UMA-мультипроцессоров тридцатью двумя или шестьюдесятью четырьмя. Чтобы преодолеть это ограничение, требуется другая схема соединительной сети. Довольно простая схема соединения n CPU и k модулей памяти представляет собой **координатный коммутатор**, Рис. 3.

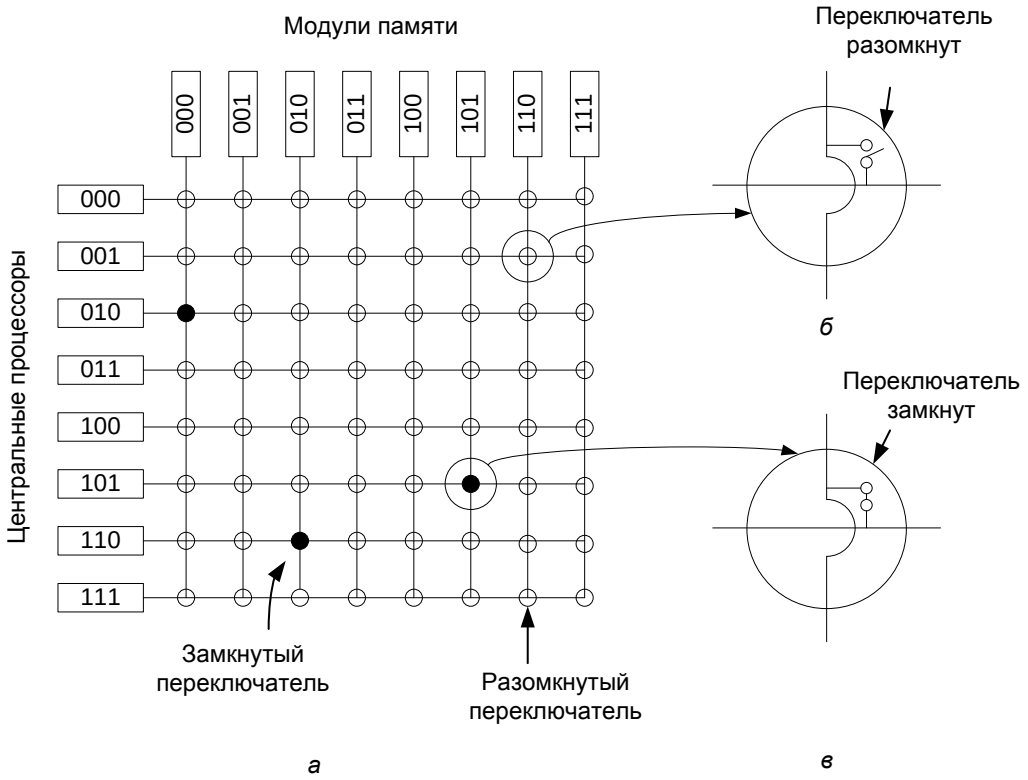


Рис. 3. Координатный коммутатор 8x8 (а); разомкнутый переключатель (б); замкнутый переключатель (е)

На каждом пересечении горизонтальной (входной) и вертикальной (выходной) линий располагается координатный переключатель. Он представляет собой небольшой переключатель, который может быть открыт или закрыт, в зависимости от того, должны быть соединены вертикальная и горизонтальная линии или нет. На Рис. 3(а) изображены три одновременно замкнутых переключателя, что позволяет одновременно соединить пары (CPU, блок памяти) (010, 000), (101, 101) и (110, 010).

Одно из самых **замечательных свойств координатного коммутатора** заключается в том, что **он представляет собой неблокирующую сеть**: **ни один CPU не получает отказа соединения по причине занятости какого-либо переключателя** (при условии, что сам требующийся модуль памяти свободен). При такой схеме не требуется планирования доступа к памяти. Даже если семь любых соединений уже установлены, всегда можно соединить оставшийся CPU с оставшимся модулем памяти.

Основной недостаток координатного коммутатора состоит в том, что число переключателей растет пропорционально квадрату от числа CPU. При 1000 CPU и 1000 модулях памяти потребуется миллион переключателей. Такой огромный координатный коммутатор просто не реализуем. Тем не менее, для систем среднего размера архитектура координатного коммутатора является применимой.

Мультипроцессоры UMA с многоступенчатые коммутаторными сетями

Принципиально другая архитектура мультипроцессоров базируется на простых коммутаторах 2x2, Рис. 4(а). У такого коммутатора два входа и два выхода. Сообщения, поступающие по любой из входных линий, могут переключаться на любую выходную линию. Сообщения в рассматриваемом нами мультипроцессоре будут состоять из четырех частей, Рис. 4(б):

- **Поле Module (модуль)** указывает модуль памяти.
- **Поле Address (адрес)** указывает адрес внутри модуля.
- **Поле Opcode (код операции)** указывает операцию, то есть READ (чтение) или WRITE (запись).
- Необязательное поле **Value (значение)** может содержать операнд, например 32-разрядное слово, которое должно быть записано операцией WRITE.

По значению поля Module коммутатор определяет, по какой из двух выходных линий следует отправить сообщение.

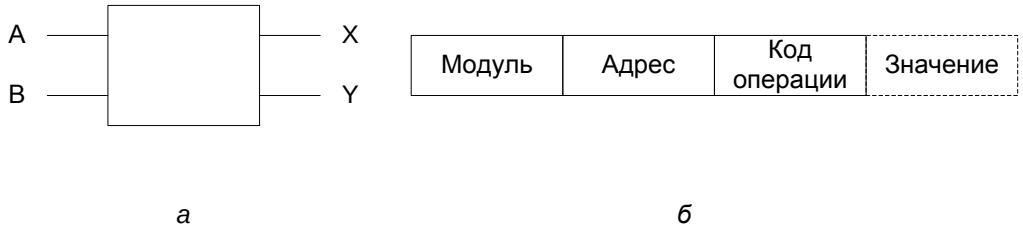


Рис. 4. Коммутатор 2x2 (а); формат сообщения (б)

С помощью данного коммутатора 2x2 можно построить самые различные **многоступенчатые коммутаторные сети**. Один из вариантов таких сетей представляет собой сеть омега, показанная на Рис. 5. Это сеть без излишеств, так сказать, экономический класс сетей. В данном примере она соединяет восемь CPU с восемью модулями памяти всего 12 коммутаторами. В более общем случае для n CPU и n модулей памяти потребуется $\log_2 n$ ступеней с $n/2$ коммутаторами в каждой ступени. Таким образом, в целом для сети омега потребуется «$\frac{n}{2}\log_2 n$» коммутаторов, что значительно меньше, чем n^2 коммутаторов, особенно для больших n.

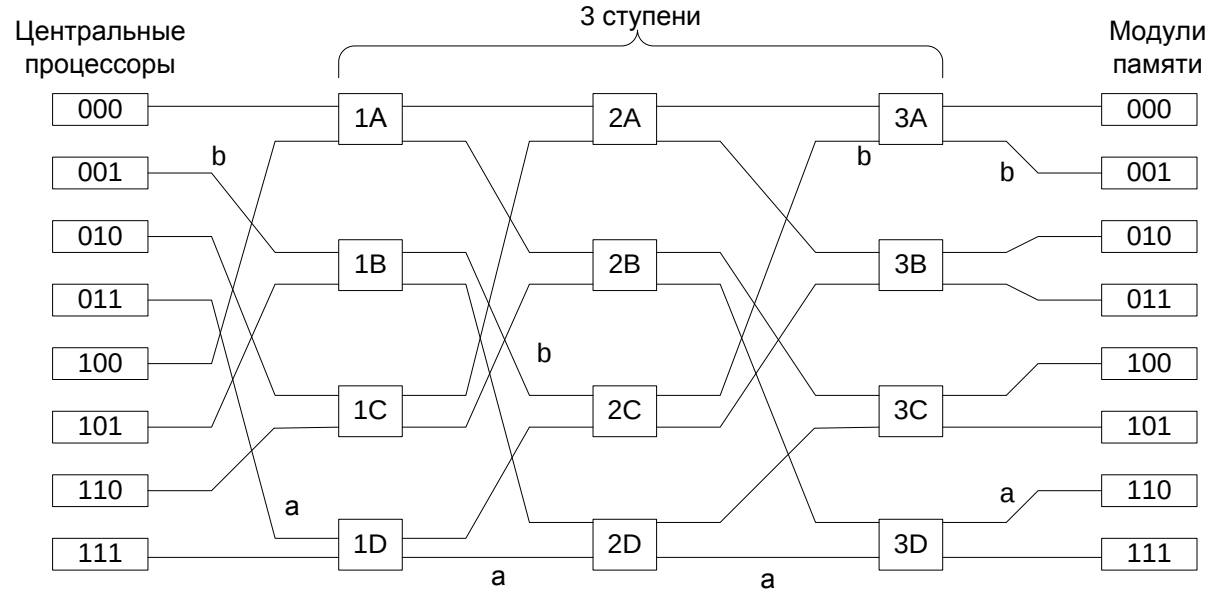


Рис. 5. Коммутирующая сеть омега

Схему соединений в сети омега часто называют **идеальным тасованием**, поскольку на каждой ступени перемешивание сигналов напоминает тасование колоды карт. Чтобы понять принципы работы сети омега, предположим, что CPU 011 нужно прочитать слово

из модуля памяти 110. CPU посылает коммутатору 1D сообщение READ|module=110. Старший (то есть самый левый) бит этого поля коммутатор использует для выбора маршрута, 0 означает выбор верхнего выхода, а 1 — нижнего. Поскольку бит равен 1, сообщение направляется по нижнему выходу коммутатору 2D.

Все коммутаторы второй ступени, включая коммутатор 2D, используют для маршрутизации второй бит. Он также равен 1, поэтому сообщение передается по нижнему выходу коммутатору 3D. Он проверяет младший бит, и поскольку бит равен 0, то сообщение передается по верхнему выходу и попадает, как и требовалось, к модулю памяти 110. Путь этого сообщения помечен символом a.

По мере продвижения по коммутирующей сети, левые биты номера модуля оказываются более не нужными. Они могут использоваться для запоминания входных линий, чтобы ответ мог найти обратный путь. Для пути a, входные линии имеют номера 0 (верхний вход ID), 1 (нижний вход 2D) и 1 (нижний вход 3D). Ответ направляется по адресу 011, обработка которого производится справа налево.

В то же самое время CPU 001 хочет записать слово в модуль памяти 001. Этот процесс происходит аналогично описанному выше. Сообщение направляется по верхнему, верхнему и нижнему выходу (путь отмечен символом b). Когда сообщение доходит до модуля памяти, поле Module=001, то есть путь, пройденный сообщением. Поскольку эти два запроса не используют общих коммутаторов, линий и модулей памяти, они могут выполняться параллельно.

Теперь посмотрим, что произойдет, если CPU 000 одновременно с этим захочет обратиться к модулю памяти 000. Его запрос войдет в конфликт с запросом CPU 001 на коммутаторе 3A. Одному из них придется подождать. В отличие от координатного коммутатора, сеть омега представляет собой **блокирующую сеть** - не все наборы запросов могут быть обработаны одновременно. Возникают конфликты из-за использования линии или коммутатора как между запросами к памяти, так и между ответами памяти на эти запросы.

Было бы желательно распределить запросы к памяти более равномерно между модулями. Один из распространенных методов заключается в использовании младших разрядов в качестве номеров модулей. Представьте, например, байт-ориентированное адресное пространство компьютера, обращающегося к памяти, в основном с 32-разрядными словами. Два младших разряда при этом обычно будут равны 00, но следующие три бита будут распределены равномерно. Если использовать эти три бита в качестве номера модуля, последовательно адресуемые слова окажутся в последовательных модулях. Система памяти, в которой соседние слова хранятся в различных модулях памяти, называется **чередующейся**. Чередующаяся память позволяет добиться максимального распараллеливания, так как большинство обращений к памяти представляют собой запросы по идущим подряд адресам. Возможно создание неблокирующих коммутирующих сетей, предоставляющих каждому CPU несколько путей к каждому модулю памяти для лучшего распределения трафика.

Мультипроцессоры NUMA

Для мультипроцессоров UMA с единственной общей шиной пределом является несколько десятков CPU, в то время как мультипроцессорам с координатным коммутатором или коммутирующей сетью требуется большое количество (дорогого) аппаратного обеспечения, и количество CPU в них не намного больше. Чтобы создать мультипроцессор с числом CPU, превосходящем 100, нужно чем-то пожертвовать. **Обычно в жертву приносится идея одинакового времени доступа ко всем модулям памяти.** Таким образом, получается концепция мультипроцессоров NUMA (**NonUniform Memory Access** — неоднородный доступ к памяти). Как и UMA, мультипроцессоры NUMA предоставляют единое адресное пространство для всех CPU, но в отличие от UMA-машин доступ к локальной памяти у них быстрее, чем к удаленным модулям. Таким образом, все программы, написанные для UMA, будут работать и на мультипроцессорах NUMA, но их производительность будет ниже, чем на машинах UMA при той же тактовой частоте CPU.

У машин NUMA есть три ключевые характеристики, которые, взятые вместе, отличают их от других мультипроцессоров.

1. Для всех нейтральных CPU имеется единое адресное пространство.

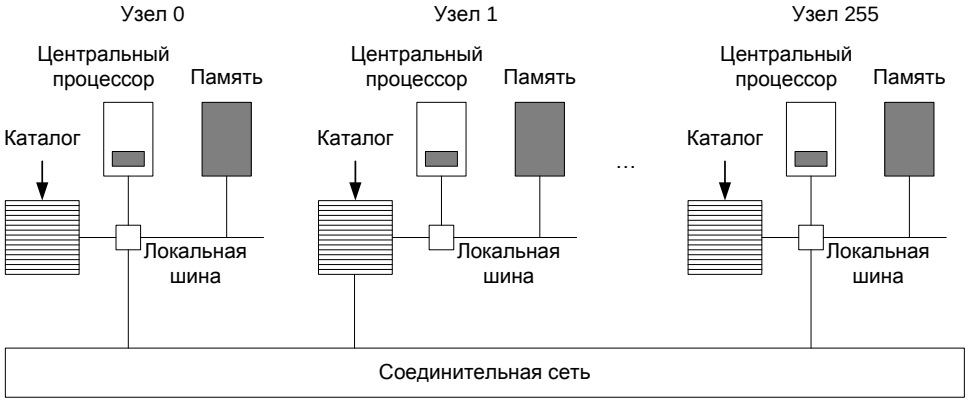
2. Доступ к удаленным модулям памяти осуществляется при помощи специальных команд CPU.

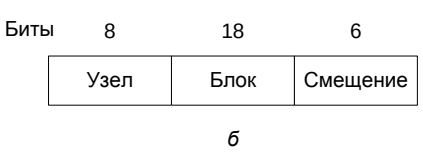
3. Доступ к удаленным модулям памяти медленнее, чем к локальной памяти.

В том случае, если доступ к удаленной памяти не является скрытым (то есть кэширование не применяется), система называется **NC-NUMA** (No Caching NUMA — система NUMA без кэширования). При наличии когерентных кэш-модулей система называется **CC-NUMA** (Cache-Coherent NUMA — система NUMA с когерентным кэшированием).

Наиболее популярным подходом при построении больших мультипроцессоров CC-NUMA в настоящий момент является **каталоговый мультипроцессор** - поддержание БД, в которой содержится информация о том, где располагается каждая строка кэша и ее состояние. При обращении к строке кэша БД получает запрос на поиск этой строки и выдает ее состояние («чистая» или «грязная»). Поскольку запросы этой БД направляются на каждой команде CPU, обращающейся к памяти, эта база должна храниться в крайне быстром специальном аппаратном устройстве, способном выдавать ответ за долю цикла шины.

Рассмотрим простой (гипотетический) пример системы, состоящей из 256 узлов, каждый из которых содержит один CPU и 16 Мбайт RAM, соединенной с CPU локальной шиной. Общий объем ОЗУ составляет 232 байт (4 Гбайт), разделенных на 226 линий кэша по 64 байт каждый. Эта память статически распределяется между узлами, с адресами от 0 до 16 M в узле 0, от 16 до 32 M в узле 1 и т.д. Узлы соединяются сетью Рис. 6(а). В каждом узле также располагаются записи каталога для 218 64-байтовых линий кэша, составляющих 2й байт памяти. Пока будем предполагать, что каждая строка может содержаться максимум в одном кэше.





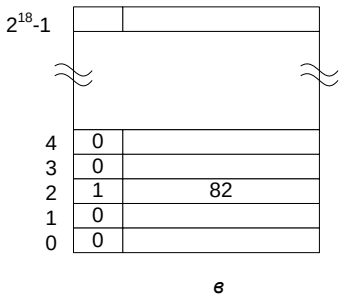


Рис. 6. Каталоговый мультипроцессор с 256 узлами (а); разделение 32-разрядного адреса на поля (б); каталог в узле 36 (в)

Чтобы понять, как работает каталог, рассмотрим выполнение команды LOAD CPU 20 с обращением к строке кэша. Сначала CPU, издающий команду LOAD, передает ее аргумент своему диспетчеру памяти (MMU), который преобразует его в физический адрес, например 0x24000108. MMU расщепляет этот адрес на три части, показанные на Рис. 6(б). В десятичном виде эти части выглядят как узел 36, строка 4 и смещение 8. MMU видит, что CPU обращается к слову памяти в узле 36, а не в узле 20, поэтому он посылает узлу 36 по соединительной сети сообщение с запросом, находится ли эта строка в кэше, и если да, то где.

Когда запрос приходит по соединительной сети на узел 36, он направляется к аппаратуре каталога. Это аппаратное обеспечение обращается по индексу в свою таблицу, состоящую из 2^{18} записей, по одной для каждой строки кэша, и достает из нее запись 4. Как видно на Рис. 6(в), эта строка не является кэшированной, поэтому аппаратное обеспечение достает ее из локальной памяти, отправляет узлу 20 и отмечает в таблице, что теперь эта строка кэширована в узле 20.

Теперь рассмотрим пример другого запроса. На этот раз у узла 36 запрашивается строка 2. Как показано на Рис. 6(в), эта строка кэширована в узле 82. При этом аппаратное обеспечение изменяет запись, отмечая, что теперь эта строка находится в узле 20, и посылает узлу 82 команду переслать эту строку узлу 20, а также пометить свою строку кэша как недействительную. Обратите внимание, что даже так называемый «мультипроцессор с общей памятью» вынужден пересылать большое количество сообщений незаметно для верхнего уровня.

Посчитаем, сколько памяти занимают каталоги, У каждого узла есть 16 Мбайт ОЗУ и 2^{18} 9-битовых записей для учета этой памяти. Таким образом, накладные расходы на содержание каталога составляют около $9x2^{18}$ бит, что составляет около 1,76 % от 16 Мбайт. Эта величина не так уж велика и должна быть приемлемой (хотя для каталога должна использоваться высокоскоростная память, что увеличивает ее стоимость). Даже при 32-байтовых строках кэша накладные расходы на содержание каталога будут около 4 %. При 128-байтовых строках кэша накладные расходы не будут превышать 1 %.

Ограничение такой схемы заключается в том, что строка может быть кэширована только в одном узле. Чтобы позволить кэшировать строку одновременно в нескольких узлах, нам потребуется какой-то способ обнаружения всех этих строк чтобы, например, пометить их все как недействительные или чтобы обновить их при записи.

Типы мультипроцессорных ОС

Возможны различные варианты организации данных мультипроцессорных ОС. Ниже будут рассмотрены три из них.

Каждому CPU — свою ОС

Простейший способ организации мультипроцессорных ОС состоит в том, чтобы статически разделить ОЗУ по числу CPU и дать каждому CPU свою собственную память с собственной копией ОС. В результате n CPU будут работать как n независимых компьютеров. В качестве очевидного варианта оптимизации можно позволить всем CPU совместно использовать код ОС и хранить только индивидуальные копии данных, Рис. 7. Квадратики, помеченные словом Data, означают персональные данные ОС для каждого CPU.

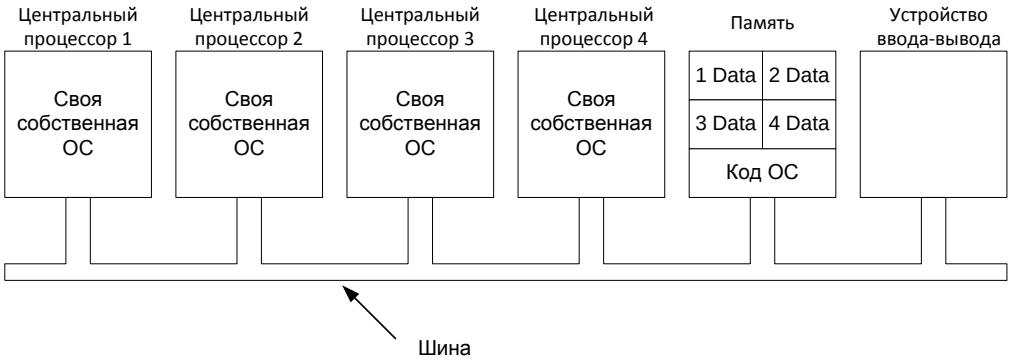


Рис. 7. Разделение памяти мультипроцессора между четырьмя CPU с общей копией кода ОС

Тем не менее, такая схема лучше, чем n независимых компьютеров, так как она позволяет всем машинам совместно использовать набор дисков и других устройств ввода-вывода, а также обеспечивает гибкое совместное использование памяти. **Например**, если требуется запустить большую программу, одному из CPU может быть выделена большая порция памяти на время выполнения этой программы. Кроме того, процессы могут эффективно общаться друг с другом, если одному процессу будет позволено писать данные в память, а другой процесс будет их считывать в этом месте.

Но с точки зрения ОС наличие ОС у каждого CPU является крайне примитивным подходом.

Следует отметить четыре аспекта данной схемы, возможно, не являющихся очевидными:

1. Когда процесс обращается к системному вызову (СВ), СВ перехватывается и обрабатывается его собственным CPU при помощи структур данных в таблицах ОС.

2. Поскольку у каждой ОС есть свои собственные таблицы, у нее есть также и свой набор процессов, которые она сама планирует. Совместного использования процессов нет. Если пользователь регистрируется на CPU 1, то все его процессы работают на CPU 1. В результате может случиться так, что CPU 1 окажется загружен работой, тогда как CPU 2 будет простаивать.

3. Совместного использования страниц также нет. Может случиться так, что у CPU 2 много свободных страниц, в то время как CPU 1 будет постоянно заниматься свопингом. И нет никакого способа занять свободные страницы у соседнего CPU, так как выделение памяти статически фиксировано.

4. Это хуже всего, если ОС поддерживает буферный кэш недавно использованных дисковых блоков, то каждая ОС будет выполнять это независимо от остальных. Таким образом, может случиться так, что некоторый блок диска будет присутствовать в нескольких буферах одновременно, причем в нескольких буферах сразу он может оказаться модифицированным, что приведет к порче данных на диске. Единственный способ избежать этого заключается в полном отказе от блочного кэша, что значительно снизит производительность всей системы.

Мультипроцессоры типа «хозяин-подчиненный»

По причине приведенных выше соображений такая модель теперь используется редко, хотя она применялась на заре эпохи мультипроцессоров, когда ставилась цель просто перенести существующие ОС на какой-либо новый мультипроцессор как можно быстрее. Эта модель показана на Рис. 8. Здесь используется всего одна копия ОС, находящаяся на CPU 1 и отсутствующая на других CPU. Все системные вызовы перенаправляются для обработки на CPU 1. CPU 1 может также выполнять процессы пользователя, если у него будет оставаться для этого время. Такая схема называется «хозяин-подчинённый», так как CPU 1 является «хозяином», то есть ведущим, а все остальные CPU — подчиненными, или ведомыми, см. Рис. 8.

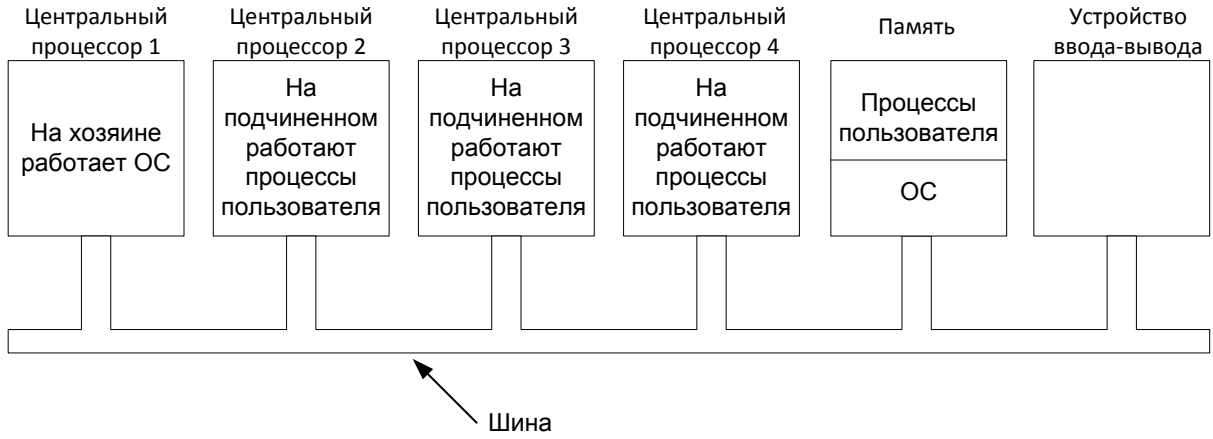


Рис. 8. Модель мультипроцессора «хозяин-подчиненный»

Модель мультипроцессора «хозяин-подчиненный» позволяет решить большинство проблем первой модели.

В этой модели используется единая структура данных (например, один общий список или набор приоритетных списков), учитывающая готовые процессы. Когда CPU переходит в состояние простоя, он запрашивает у ОС процесс, который можно обрабатывать, и при наличии готовых процессов ОС назначает этому CPU процесс. Поэтому при такой организации: никогда не может случиться так, что один CPU будет простаивать, в то время как другой CPU перегружен. Страницы памяти могут динамически предоставляться всем процессам. Кроме того, в такой системе есть всего

один общий буферный кэш блочных устройств, поэтому дискам не грозит порча данных, как в предыдущей модели при попытке использования блочного кэша.

Недостаток: при большом количестве CPU хозяин может стать узким местом системы. Ведь ему приходится обрабатывать все СВ от всех CPU. **Например**, если обработка СВ занимает 10 % времени, тогда 10 CPU завалят хозяина работой, а при 20 CPU хозяин уже не будет успевать их обрабатывать, и система начнет простаивать. Следовательно, такая модель проста и работоспособна для небольших мультипроцессоров, но на больших она работать не может.

Симметричные мультипроцессоры

Другая модель, представляющая собой симметричные мультипроцессоры (SMP, Symmetric Multiprocessor), позволяет устранить перекос предыдущей модели. Как и в предыдущей схеме, в памяти находится всего одна копия ОС, но выполнять ее может любой CPU. При системном вызове на CPU, обратившемся к системе с СВ, происходит прерывание с переходом в режим ядра и обработкой СВ. Модель симметричного мультипроцессора показана на Рис. 9.

Эта модель обеспечивает динамический баланс процессов и памяти, поскольку в ней имеется всего один набор таблиц ОС. Она также позволяет избежать простоя системы, связанного с перегрузкой ведущего CPU, так как в ней нет ведущего CPU. И все же данная модель имеет собственные проблемы. В частности, если код ОС будет выполняться одновременно на двух или более CPU, произойдет катастрофа. Представьте себе два CPU, одновременно берущих один и тот же процесс для запуска или запрашивающих одну и ту же свободную страницу памяти. Простейший способ разрешения подобных проблем заключается в связывании мьютекса (то есть блокировки) с ОС, в результате чего вся система превращается в одну большую критическую область. Когда CPU хочет выполнять код ОС, он должен сначала получить мьютекс. Если мьютекс заблокирован, CPU вынужден ждать. Таким образом, любой CPU может выполнить код ОС, но в каждый момент времени только один из них будет делать это.

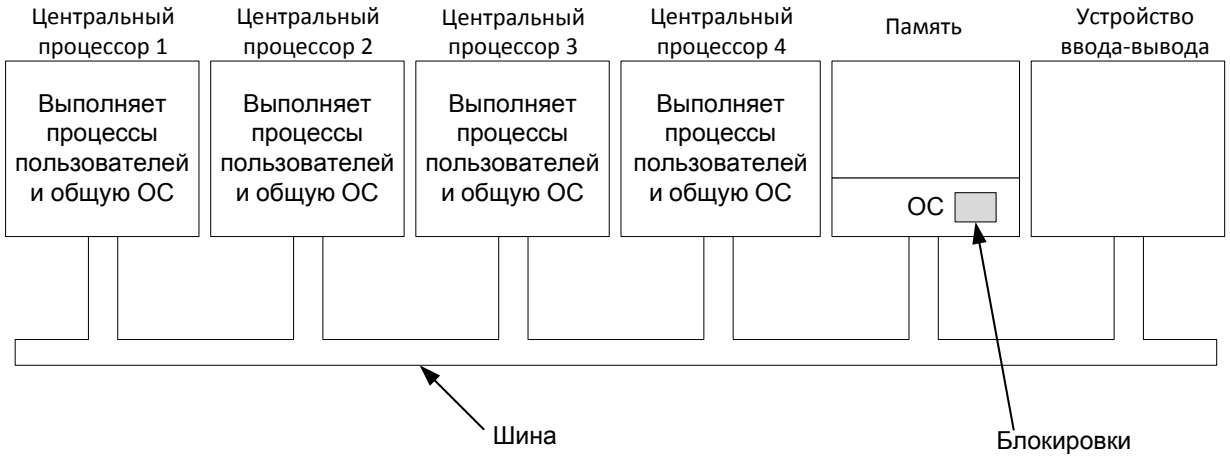


Рис. 9. Модель симметричного мультипроцессора

Такая модель работает, но она практически так же плоха, как и модель «хозяин-подчиненный». Опять же предположим, что 10 % всего времени CPU расходуется на выполнение самой ОС. При 20 CPU большинство CPU будут подолгу стоять в длинных очередях, ожидая разращения доступа к мьютексу. К счастью, такую ситуацию легко исправить. Многие части ОС независимы друг от друга. Например, один CPU может заниматься планированием, в то время как другой CPU будет выполнять обращение к файловой системе, а третий обрабатывать страничное прерывание.

Такое наблюдение приводит к расщеплению ОС на независимые критические области, не взаимодействующие друг с другом. У каждой критической области есть свой мьютекс, поэтому только один CPU может выполнять ее в каждый момент времени. Таким образом, можно достичь большей степени распараллеливания. Однако может случиться, что некоторые таблицы, например таблица процессов, используются в нескольких критических областях. Так, таблица процессов требуется для планирования, но также для выполнения системного вызова `CreateProcess` и для обработки сигналов. Для каждой таблицы, использующейся несколькими критическими

областями, требуется свой собственный мьютекс. Итак, в каждый момент времени каждая критическая область может выполняться только одним CPU, а также к каждой критической таблице может быть предоставлен доступ только одному CPU.

Подобная организация используется в большинстве современных мультипроцессоров. Сложность написания ОС для такой машины заключается не в том, что программный код сильно отличается от обычной операционной системы. Нет. Самым сложным является расщепление операционной системы на критические области, которые могут выполняться параллельно на разных CPU, не мешая друг другу даже косвенно. Кроме того, каждая таблица, используемая двумя и более критическими областями, должна быть отдельно защищена мьютексом, а все программы, пользующиеся этой таблицей, должны корректно использовать мьютекс.

Кроме того, следует уделить особое внимание вопросу избегания взаимоблокировок. Если двум критическим областям одновременно потребуются таблица A и таблица B и они затребуют эти таблицы в разном порядке, то рано или поздно возникнет взаимоблокировка, причину появления которой будет очень трудно определить. Теоретически всем таблицам можно поставить в соответствие целые числа и потребовать от всех критических областей запрашивать эти таблицы по порядку номеров. Такая стратегия позволяет избежать взаимоблокировок, но она требует от программиста детального исследования того, какие таблицы потребуются каждой критической области, чтобы запросить их в правильном порядке.

При изменении программы со временем критической области может потребоваться новая таблица, которая не была нужна ранее. Если изменением программы занимается новый программист, не понимающий всей логики системы, он может поддаться искушению просто захватить мьютекс в тот момент, когда требуется таблица, и отпустить его, когда таблица более не нужна. Однако именно такие простые и кажущиеся разумными действия могут привести к взаимоблокировке, что с точки зрения пользователя выглядит как зависание системы. Очень не просто грамотно спроектировать систему, но поддерживать ее в правильном состоянии в течение нескольких лет и при этом совершенствовать систему меняющимся штатом программистов крайне трудно.

Синхронизация в мультипроцессорах

Если процесс на однопроцессорной системе обращается к СВ, который требует доступа к некой критической таблице, программа ядра может просто запретить прерывания, прежде чем обратиться к таблице. Однако на мультипроцессоре запрет прерывания повлияет только на один CPU, выполнивший команду запрета прерываний. Остальные CPU продолжат свою работу и смогут получить доступ к критической таблице. Требуется специальный **мьютекс-протокол**, который будет выполняться всеми CPU, чтобы гарантировать работу взаимного исключения.

Сердцем любого практического мьютекс-протокола является команда CPU, позволяющая исследовать и изменить слово в памяти за одну операцию. Пример использования команды TSL (Test and Set Lock — проверить и установить блокировку) для реализации критических областей был рассмотрен в **соответствующей лекции**. Как уже было сказано, эта команда считывает слово памяти в регистр CPU. Одновременно она записывает 1 (или другое ненулевое значение) в слово памяти. Конечно, для выполнения операций чтения и записи памяти требуется два цикла шины. На однопроцессорной машине, поскольку одна команда CPU не может быть прервана на полпути, команда TSL работает должным образом.

Теперь представьте себе, что может произойти на мультипроцессоре. На Рис. 10 показан наихудший случай совпадения по времени обращений к слову памяти по адресу 1000, начальное состояние которого 0. На шаге 1 CPU 1 считывает слово и получает 0. На шаге 2, прежде чем CPU 1 успевает изменить это слово, CPU 2 также считывает слово и тоже получает 0. На шаге 3 CPU 1 записывает в это слово 1. На шаге 4 CPU 2 также записывает в это слово 1. Оба CPU получили от команды TSL 0, поэтому оба считают себя вправе получить доступ к критической области. **Таким образом, взаимное исключение не срабатывает.**

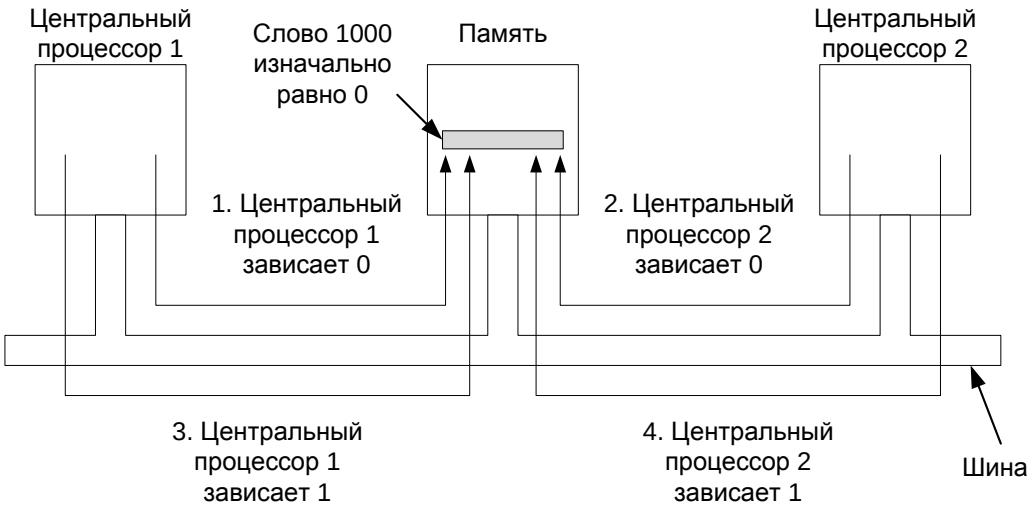


Рис. 10. Команда TSL может работать неверно, если не заблокировать шину

Для разрешения этой проблемы команда TSL сначала должна блокировать шину посредством команды `LOCK`, не допуская обращения к ней других CPU, затем выполнить оба обращения к памяти, после чего разблокировать шину. Как правило, посреством этой команды сначала выполняется обычный запрос шины по стандартному протоколу, затем устанавливается в 1 некая специальная линия шины. Пока эта линия шины установлена в 1, никакой другой CPU не может получить к ней доступ. Такая команда может быть выполнена только на шине, у которой есть необходимые специальные линии и (аппаратный) протокол для их использования. Для чисто программной синхронизации был разработан протокол Петерсона.

Если команда TSL корректно реализована и применяется, она гарантирует правильную работу взаимного исключения. Однако подобный способ реализации взаимного исключения использует **спин-блокировку**, так как запрашивающий CPU находится в цикле опроса блокировки с максимальной скоростью. Этот метод является не только тратой времени запрашивающего CPU, но он также может накладывать значительную нагрузку на шину или память, существенно снижая скорость работы всех остальных CPU, пытающихся выполнять свою обычную работу.

На первый взгляд может показаться, что наличие кэширования должно устранить проблему конкуренции за шину, но это не так. Теоретически, как только запрашивающий CPU прочитал слово блокировки, он должен получить его копию в свой кэш. Пока ни один другой CPU не предпринимает попыток использовать это слово, запрашивающий CPU может работать с ним в своем кэше. Когда CPU, владеющий словом блокировки, записывает в него 1, протокол кэша автоматически помечает как недействительные все копии этого слова в удаленных кэшах, требуя получения правильных значений.

Проблема заключается в том, что кэш оперирует блоками по 32 или 64 байт. Обычно слова, окружающие слово блокировки, нужны CPU, удерживающему это слово. Поскольку команда TSL представляет собой запись (так как она модифицирует слово блокировки), ей требуется монопольный доступ к блоку кэша, содержащему слово блокировки. Таким образом, каждая команда TSL помечает блок кэша владельца блокировки как недействительный и получает приватную, эксклюзивную копию для запрашивающего CPU. Как только владелец блокировки изменит слово, соседнее с блокировкой, блок кэша перемещается на его машину. В результате весь блок кэша, содержащий слово блокировки, постоянно как челнок мотается взад-вперед от CPU, удерживающего блокировку, к CPU, пытающемуся ее получить. Все это создает довольно значительный и совершенно излишний трафик шины.

Проблему можно было бы решить, если бы удалось избавиться от всех операций записи, вызванных командой TSL запрашивающей стороны:

1. Это можно сделать, если запрашивающая сторона сначала будет **выполнять простую итерацию чтения, чтобы убедиться, что мьютекс свободен**. Только убедившись, что он свободен, CPU выполняет команду TSL, чтобы захватить его. В результате большинство операций опроса представляют собой операции чтения, а не операции записи. Когда мьютекс считан, владелец выполняет операцию записи, для

чего требуется монопольный доступ. При этом все остальные копии этого блока кэша объявляются недействительными. При следующей операции чтения запрашивающий CPU перезагрузит этот блок кэша. Обратите внимание, что при одновременном споре двух CPU за один и тот же мьютекс может случиться, что они одновременно увидят, что мьютекс свободен, и одновременно выполнят команду TSL. Ничего страшного в этом случае не произойдет, так как выполнена будет только одна команда TSL и такая ситуация не приведет к состоянию состязания. Если вы видите, что мьютекс свободен, и тут же пытаетесь его схватить командой TSL, то нет никакой гарантии успеха данного предприятия. Мьютекс может успеть захватить кто-либо другой.

2. Другой способ снижения шинного трафика заключается в использовании **алгоритма двоичного экспоненциального отката**, применяемого в сети Ethernet. Вместо постоянного опроса, между опросами может быть вставлен цикл задержки. Вначале задержка представляет собой одну команду.

Если мьютекс занят, задержка удваивается, учетверяется и т. д. до некоторого максимального уровня. При низком уровне получим быструю реакцию при освобождении мьютекса, зато потребуется больше обращений к шине для чтения блока кэша. Высокий максимальный уровень позволит уменьшить число лишних обращений к шине за счет более медленной реакции программы на освобождение мьютекса. Использование алгоритма двоичного экспоненциального отката не зависит от применения операций чистого чтения перед командой TSL

3. Еще более удачная идея заключается в том, чтобы каждому CPU, желающему заполучить мьютекс, **позволить опрашивать свою собственную переменную блокировки**, Рис. 11. Во избежание конфликтов переменная должна располагаться в не используемом для других целей блоке кэша. Работа алгоритма состоит в том, что CPU, которому не удается заполучить мьютекс, захватывает переменную блокировки и присоединяется к концу списка CPU, ожидающих освобождения мьютекса. Когда CPU, удерживающий мьютекс, покидает критическую область, он освобождает персональную переменную, проверяемую первым CPU в списке (в его собственном кэше). Тем самым следующий CPU получает возможность войти в критическую область. Покинув критическую область, этот CPU освобождает переменную блокировки, тестируемую следующим CPU и т.д. Хотя такой протокол несколько сложен (это необходимо, чтобы не допустить одновременного присоединения двух CPU к концу списка), он эффективен и не страдает от проблемы голодания.

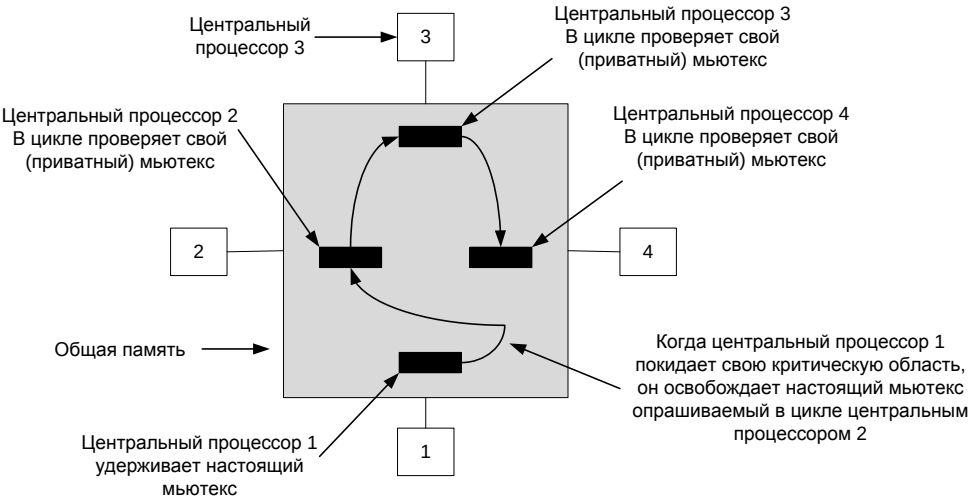


Рис. 11. Использование нескольких мьютексов во избежание пробуксовки кэша

Ожидание в цикле или переключения?

До сих пор предполагалось, что CPU, которому требуется мьютекс, просто ждет, пока тот не освободится, опрашивая его состояние постоянно или периодически, либо присоединяясь к списку ожидающих CPU.

Однако в некоторых случаях у CPU может быть выбор. Например, если какому-либо потоку на CPU потребуется доступ к блочному кэшу ФС, заблокированному в данный момент, CPU может решить не ждать, а переключиться на другой поток. Вопрос принятия решения о том, ждать или переключиться на другой поток, являлся предметом многочисленных исследований, некоторые из них будут обсуждаться ниже.

Если допустить, что оба варианта (опрос в цикле и переключение потоков) выполнимы, возникает следующая ситуация. Циклический опрос напрямую расходует время CPU. Периодическая проверка состояния мьютекса не является продуктивной работой. Переключение процессов, однако, также расходует циклы CPU, так как для этого требуется сохранить текущее состояние потока, получить мьютекс списка свободных процессов, выбрать из этого списка поток, загрузить его состояние и передать ему управление. Более того, кэш CPU будет содержать не те блоки, поэтому после переключения потоков возможно много промахов кэша. Также вероятны ошибки TLB. Наконец, потребуется переключение обратно на исходный поток, результатом которого снова будут промахи кэша. Циклы CPU, потраченные на эти два переключения контекста плюс промахи кэша, представляют собой существенные накладные расходы.

Если известно, что мьютексы удерживаются, как правило, в течение 50 мкс, а переключение с одного потока на другой занимает 1 мс, а также 1 мс требует обратное переключение, то более эффективное решение состоит в **простом ожидании освобождения мьютекса в цикле**. С другой стороны, если средний мьютекс удерживается на 10 мс, то два переключения контекста являются вполне оправданными. Проблема в том, что длительность нахождения процессов в критических областях может варьироваться в широких пределах, поэтому выбор верного решения непрост:

1. Всегда использовать циклический опрос.

2. Использование только переключений.

3. Каждый раз принимать отдельное решение.

В момент принятия решения не известно, что лучше — переключаться или ждать, но в любой системе можно отследить активность процессов и затем проанализировать ее в автономном режиме. При этом можно сказать, какое решение было бы лучшим в том или ином случае и сколько времени CPU было потрачено. Таким образом, ретроспективный алгоритм становится эталоном, относительно которого может измеряться эффективность реальных алгоритмов.

На практике используется модель, в которой поток, не заполучивший мьютекс, какое-то время опрашивает состояние мьютекса в цикле (**спин-блокировка**). Если пороговое значение времени ожидания превышается, он переключается. В некоторых случаях пороговое значение фиксировано, как правило, оно равно известному времени, требующемуся на переключение на другой поток и обратно. В других случаях эта величина меняется динамически, в зависимости от истории состояния ожидаемого мьютекса.

Планирование мультипроцессора

На мультипроцессоре планирование двумерно. **Планировщик должен решить, какой процесс и на котором CPU запустить.** Это дополнительное измерение существенно усложняет планирование на мультипроцессорах.

Другой усложняющий фактор состоит в том, что в некоторых системах все процессы являются независимыми, тогда как в других системах они формируют группы.

Примером **первой ситуации** является система реального времени, в которой независимые пользователи запускают независимые процессы. Эти процессы не связаны друг с другом, и планирование каждого из них не зависит от остальных процессов.

Пример **второй ситуации** часто встречается в среде разработки программ. Большие системы, как правило, состоят из некоторого количества заголовочных файлов, содержащих макросы, определения типов и объявления переменных, используемых в собственно файлах программы.

Разделение времени

Рассмотрим сначала случай планирования независимых процессов. Простейший алгоритм планирования независимых процессов (или потоков) состоит в поддержании единой структуры данных для готовых процессов, возможно, просто списка, но скорее всего множества списков для процессов с различными приоритетами, Рис. 12(а). Здесь все 16 CPU в данный момент заняты, а 14 процессов с различными приоритетами ожидают запуска. Первым заканчивает работу (или его процесс блокируется) CPU 4. При этом он блокирует очередь планирования и выбирает из нее процесс с наивысшим

приоритетом, то есть процесс A, Рис. 12(б). Затем освобождает CPU 12 и выбирает процесс B, Рис. 12(в). Пока эти процессы независимы, подобное планирование представляет собой разумный выбор.

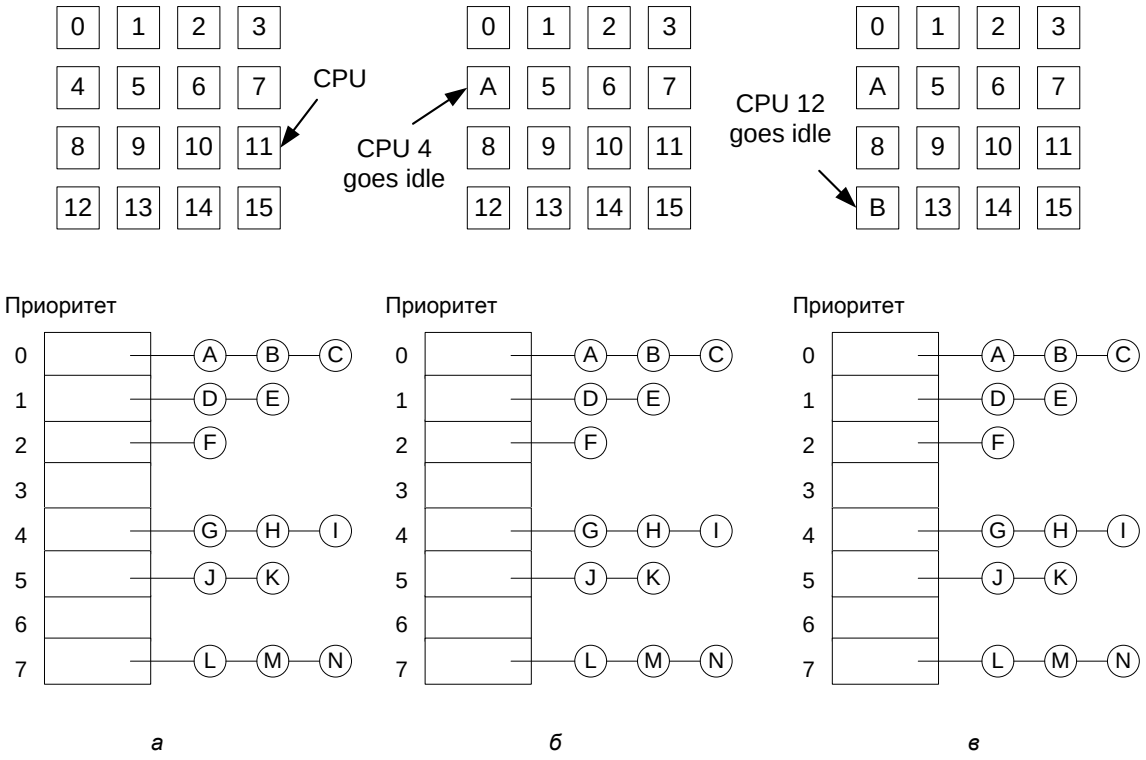


Рис. 12. Использование единой структуры данных для планирования мультипроцессора

Наличие единой структуры данных планирования, используемой всеми CPU, обеспечивает CPU режим разделения времени подобно тому, как это выполняется на однопроцессорной системе. Кроме того, такая организация позволяет автоматически балансировать нагрузку, то есть она исключает ситуацию, при которой один CPU простаивает, в то время как другие CPU перегружены.

Два недостатка такой схемы представляют собой:

- потенциальный рост конкуренции за структуру данных планирования по мере увеличения числа CPU;
- обычные накладные расходы на выполнение переключения контекста, когда процесс блокируется, ожидая выполнения операции ввода-вывода.

Переключение контекста также может случиться, когда истекает квант процесса. Мультипроцессор обладает определенными свойствами, которыми не обладают однопроцессорные системы. Предположим, что процесс удерживает спин-блокировку, что, как уже говорилось выше, часто встречается на мультипроцессорах. Другие CPU, ждущие освобождения блокировки, просто теряют время в циклах ожидания, пока этот процесс не будет запущен снова и не отпустит блокировку. На однопроцессорных системах спин-блокировка применяется редко. Поэтому если процесс, удерживающий мьютекс, блокируется и запускается другой процесс, то при попытке заполучить мьютекс второй процесс будет тут же блокирован, и много времени потеряно не будет.

Чтобы решить данную проблему, в некоторых системах применяется **умное планирование**, в котором процесс, захватывающий спин-блокировку, устанавливает флаг, демонстрирующий, что он в данный момент обладает спин-блокировкой. Когда процесс освобождает блокировку, он также очищает и флаг. Таким образом, планировщик не останавливает процесс, удерживающий спин-блокировку, а, напротив, дает ему еще немного времени, чтобы тот завершил выполнение критической области и отпустил мьютекс.

Другая проблема, играющая важную роль в планировании, заключается в том факте, **что хотя все CPU равны, но некоторые CPU равнее других**. В частности, когда процесс A достаточно долго работает на CPU k, то его кэш будет полон блоками процесса A. Если процесс A должен быть снова вскоре запущен, его производительность может оказаться выше, если он будет запущен снова на CPU k, так

как кэш CPU k может все еще содержать некоторые блоки процесса A. Наличие блоков в кэше увеличит частоту попаданий кэша и, таким образом, увеличит скорость выполнения процесса. Кроме того, TLB также может содержать правильные страницы, что снизит количество прерываний из-за ошибок TLB.

Некоторые мультипроцессоры учитывают данные соображения и используют так называемое **родственное планирование**. Основная идея данного метода заключается в приложении серьезных усилий для того, чтобы процесс был запущен на том же CPU, что и в прошлый раз. Один из способов реализации этого метода состоит в использовании **двухуровневого алгоритма планирования**. В момент создания процесс назначается конкретному CPU, например наименее загруженному в данный момент. Это назначение процессов CPU представляет собой верхний уровень алгоритма. В результате каждый CPU получает свой набор процессов.

Действительное планирование процессов находится на нижнем уровне алгоритма. Оно выполняется отдельно каждым CPU при помощи приоритетов или других средств. Старания удерживать процессы на одном и том же CPU максимизируют родственность кэша. Однако если у какого-либо CPU нет работы, у загруженного работой CPU отнимается процесс и отдается ему.

Двухуровневое планирование обладает тремя преимуществами:

1. Оно довольно равномерно распределяет нагрузку среди имеющихся CPU.

2. Двухуровневое планирование по возможности использует преимущество родственности кэша.

3. Поскольку у каждого CPU при таком варианте планирования есть свой собственный список свободных процессов, конкуренция за списки свободных процессов минимизируется, так как попытки использования списка другого CPU происходят относительно нечасто.

Совместное использование пространства

Другой подход к планированию мультипроцессоров может быть использован, если процессы связаны друг с другом каким-либо способом. Для нашей задачи задание, состоящее из нескольких связанных процессов или процесс, состоящий из нескольких потоков ядра, представляют собой, по сути, одно и то же. Будем называть планируемые объекты **потоками**, но все сказанное здесь в равной мере справедливо и для процессов. Планирование нескольких потоков на нескольких CPU называется **совместным использованием пространства** или **разделением пространства.**

Простейший алгоритм разделения пространства работает следующим образом. Предположим, что сразу создается целая группа связанных потоков. В момент их создания планировщик проверяет, есть ли свободные CPU по количеству создаваемых потоков. **Если свободных CPU достаточно, каждому потоку выделяется собственный** (то есть работающий в однозадачном режиме) CPU и все потоки запускаются. **Если CPU недостаточно, ни один из потоков не запускается, пока не освободится достаточное количество CPU**. Каждый поток выполняется на сроем CPU вплоть до завершения, после чего все CPU возвращаются в пул свободных CPU. Если поток оказывается заблокированным операцией ввода-вывода, он продолжает удерживать CPU, который простаивает до тех пор, пока поток не сможет продолжать свою работу. При появлении следующего пакета потоков применяется тот же алгоритм.

В любой момент времени множество CPU статически разделяется на несколько подмножеств, на каждом из которых выполняются потоки одного процесса. На Рис. 13 показаны подмножества из 4, 6, 8 и 12 CPU, и 2 CPU остались не включенными в подмножества. Со временем, по мере завершения работы одних процессов и появления новых процессов, количество и размеры групп CPU изменяются.

Рис. 13. Набор из 32 CPU, разделенный на четыре группы, плюс два CPU свободны

Периодически должны приниматься решения о планировании процессов. В однопроцессорных системах для пакетного планирования применяется хорошо известный алгоритм **«кратчайшее задание первое»**. Подобный алгоритм для мультипроцессора представляет собой выбор процесса, для которого требуется наименьшее количество циклов CPU, то есть процесса, чье произведение числа CPU на время работы минимально. Однако на практике эта информация редко бывает доступна, поэтому применение такого алгоритма затруднительно.

В этой простой модели разбиения CPU на группы процесс просто запрашивает определенное количество CPU и либо сразу получает их, либо ждет, пока они не освободятся.

Другой подход состоит в том, чтобы активно управлять степенью распараллеливания процессов:

Один из способов управления степенью распараллеливания заключается в наличии центрального сервера, ведущего **учет работающих и желающих работать процессов**, а также минимального и максимального количества требующихся для них CPU. Периодически каждый CPU опрашивает центральный сервер, чтобы узнать, сколько CPU он может использовать. Затем он увеличивает или уменьшает количество процессов или потоков, стараясь добиться соответствия числу доступных CPU.

Бригадное планирование

Очевидное преимущество разделения пространства заключается в устранении многозадачности, что снижает накладные расходы по переключению контекста. Однако ее недостаток состоит в потерях времени при блокировке CPU.

Чтобы понять, какие возможны проблемы при независимом планировании потоков процесса (или процессов задания), рассмотрим систему с потоками A_0 и A_1, принадлежащими процессу A, и потоками B_0 и B_1 принадлежащими процессу B. Потоки A_0 и B_0 работают в режиме разделения времени на CPU 0, а потоки A_1 и B_1 — на CPU 1. Потокам A_0 и A_1, нужно часто обмениваться информацией. Общение потоков выглядит следующим образом. Поток A_0 посылает потоку A_1 сообщение, после чего поток A_1 отправляет потоку A_0 ответ и т. д. Предположим, что потоки A_0 и B_1 начали выполняться первыми, как показано на Рис. 14.

В интервале времени 0 поток A_0 посылает потоку A_1 запрос, но поток A_1 не получает его до тех пор, пока не будет запущен в интервале времени 1, начинающемся через 100 мс. Он немедленно отправляет ответ, но поток A_0 не получает ответа, пока его снова не запустят в момент времени 200 мс. В результате за 200 мс получается всего одна пара запрос-ответ, что не слишком хорошо.

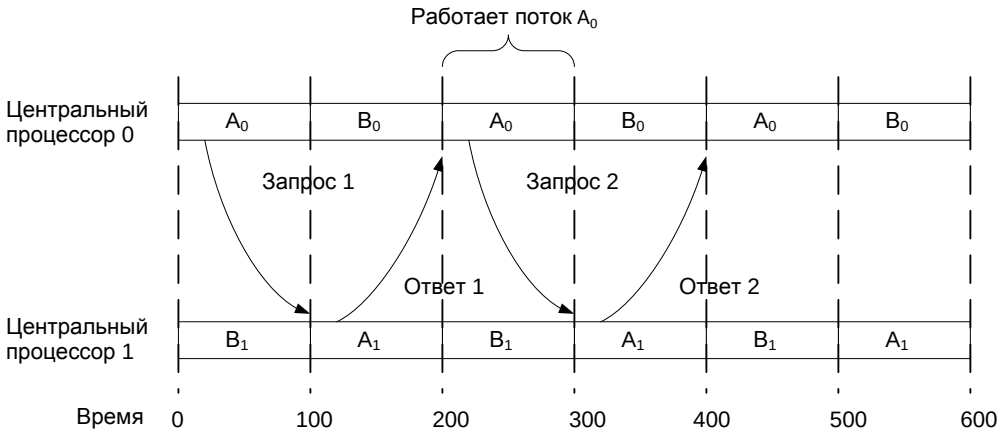


Рис. 14. Общение двух работающих в противофазе потоков, принадлежащих процессу A

Решением данной проблемы является так называемое **бригадное планирование**, представляющее собой развитие идеи совместного планирования. Бригадное планирование состоит из трех частей:

1. Группы связанных потоков планируются как одно целое, бригада.

2. Все члены бригады запускаются одновременно, на разных CPU с разделением времени.

3. Все члены бригады начинают и завершают свои временные интервалы вместе.

Бригадное планирование работает благодаря синхронности работы всех CPU. Это значит, что время разделяется на дискретные кванты, как было показано на Рис. 14. В начале каждого нового кванта все CPU перепланируются заново, и на каждом CPU запускается новый поток. В начале следующего кванта опять принимается решение о планировании. В середине кванта планирование не выполняется. Если какой-либо поток блокируется, его CPU простаивает до конца кванта времени.

Пример работы бригадного планирования приведен на Рис. 15. Здесь показан мультипроцессор с шестью CPU, на которых работают пять процессов, от A до E, с общим числом потоков, равным 24. В течение временного интервала 0 потоки от A_0 до A_6 планируются и выполняются. Во время интервала 1 планируются и выполняются потоки B_0-B_2 и C_0-C_2. В течение временного интервала 2 планируются и выполняются пять потоков процесса D и поток E_0. Оставшиеся шесть потоков процесса E работают во время интервала 3. Затем цикл повторяется, так что временной интервал 4 повторяет интервал 0 и т. д.

Идея бригадного планирования состоит в том, чтобы все потоки процесса работали по возможности вместе, так, что если один из них посылает сообщение другому потоку, то второй поток получает сообщение практически мгновенно и может так же быстро на него ответить. На Рис. 15, поскольку все потоки процесса A работают вместе в течение одного кванта времени, они могут отправлять и принимать большое количество сообщений за один квант времени, устраняя, таким образом, проблему Рис. 14.

Рис. 15. Бригадное планирование

OpenMP API

Введение

Коротко остановимся на программном интерфейсе OpenMP C/C++ [6], который поддерживается большинством современных компиляторов, и позволяющий использовать внутренний параллелизм приложений.

Для использования всей мощи данной библиотеки, для Microsoft Visual C++ следует подключить VCOMP.LIB (VCOMPD.LIB – debug version). Более подробную информацию об использовании и подключении OpenMP к компилятору, можно в [5].

Директивы

Директива	Описание
`Atomic`	Указывает, что область памяти будет обновлена автоматически.
`Barrier`	Синхронизирует все потоки в одну группу; все потоки замораживаются на барьере, до тех пор, пока все потоки не выполнят барьер.
`Critical`	Указывает, код следует выполнять лишь на одном потоке в один и тот же момент времени.
`flush (OpenMP)`	Указывает, что все потоки обладают одинаковым видом памяти для всех общих объектов.
`for (OpenMP)`	Указывает, что следует выполнять все действия внутри цикла в параллельных потоках.
`Master`	Указывает, что только ведущий поток может выполнить данную секцию программы.
`ordered (OpenMP)`	Указывает, что код в параллельном цикле следует выполнять как последовательный цикл.
`Parallel`	Определяет параллельную область, которая является кодом исполняющимся параллельно множеством потоков.
`sections (OpenMP)`	Идентифицирует секцию кода разделенную среди всех потоков.
`Single`	Позволяет определить, что секцию кода следует выполнить в одном потоке, не обязательно ведущим потоком.

`threadprivate`	Указывает, что переменная является персональной (личной) для потока.

Пример

Ниже приведем пример исходного кода C++ использующего директивы и функции OpenMP.

```
// omp_init_lock.cpp
// compile with: /openmp
#include <stdio.h>
#include <omp.h>
omp_lock_t my_lock;
int main() {
   omp_init_lock(&my_lock);
   #pragma omp parallel num_threads(4)
   {
      int tid = omp_get_thread_num( );
      int i, j;
      for (i = 0; i < 5; ++i) {
         omp_set_lock(&my_lock);
         printf_s("Thread %d - starting locked region\n", tid);
         printf_s("Thread %d - ending locked region\n", tid);
         omp_unset_lock(&my_lock);
      }
   }
   omp_destroy_lock(&my_lock);
}
```

Литература

1. Э. Таненбаум. Современные операционные системы. 2-ое изд. –СПб.: Питер, 2002. – 1040 с.

2. А. Шоу. Логическое проектирование операционных систем. Пер. с англ. –М.: Мир, 1981. –360 с.

3. С. Кейслер. Проектирование операционных систем для малых ЭВМ: Пер. с англ. –М.: Мир, 1986. –680 с.

4. Э. Таненбаум, А. Вудхалл. Операционные системы: разработка и реализация. Классика CS. –СПб.: Питер, 2006. –576 с.

5. Microsoft Development Network. URL: http://msdn.com

6. OpenMP API. URL: http://openmp.org