2.6

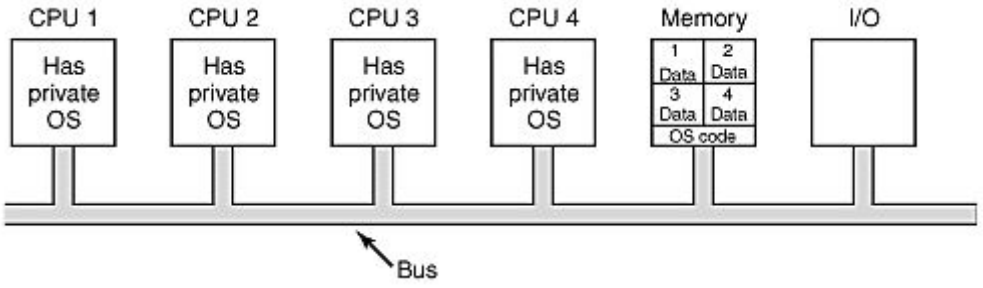
Многоядерные/многопроцессорные системы, одновременная многопоточность (SMT/HT). Потоковые процессоры.

1. Многопроцессорность

Рассмотрим несколько подходов к реализации ОС, использующих несколько процессоров.

1.1. У каждого CPU своя ОС

Наиболее тривиальный способ – разделить память на сколько частей, сколько имеется процессоров, назначить каждому из них свою приватную часть памяти и копию ОС. Фактически, в такой ситуации CPU работают как независимые компьютеры.
Одна из очевидных оптимизаций – позволить CPU расшарить код ОС между собой, и иметь приватные копии только данных (пример на рисунке).



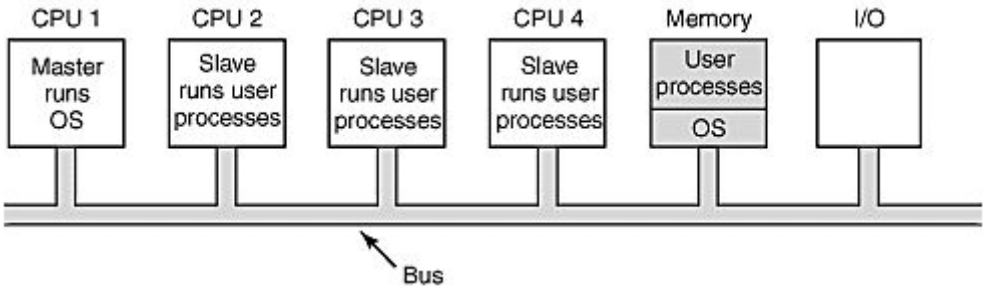
Такая схема все равно лучше, чем несколько независимых компьютеров, поскольку позволяет CPU использовать общие устройства ввода/вывода, а также выдать одному из CPU увеличенный объем памяти на определенное время.
Еще процессы могут эффективно взаимодействовать между собой, используя общую память (например, можно назначить определенный участок памяти, куда процесс A запишет данные, и процесс B считает их с того же места).

Минусы:

1. Нельзя расшарить процессы между CPU. Поэтому если пользователь запустил программу на CPU1, то остальные CPU будут простаивать.
2. Нельзя динамически распределять память. Если у CPU1 закончилась память, а у CPU2 есть свободная, очень жаль.
3. *(главный)* Каждая ОС кеширует блоки недавно использованного дискового пространства, независимо от остальных ОС. Из-за этого в буферах ОС могут храниться конфликтующие версии измененных данных. Для исправления такой проблемы нужно избавиться от буферов кеширования. Это не сложно, но сильно вредит производительности.

1.2. Master-slave

При таком подходе CPU1 назначается главным. На нем запускается ОС, никакие другие CPU не работают с копиями ОС. Все вызовы системных функций перенаправляются к CPU1. Также CPU1 может запускать некоторые пользовательские процессы, если осталось свободное процессорное время.

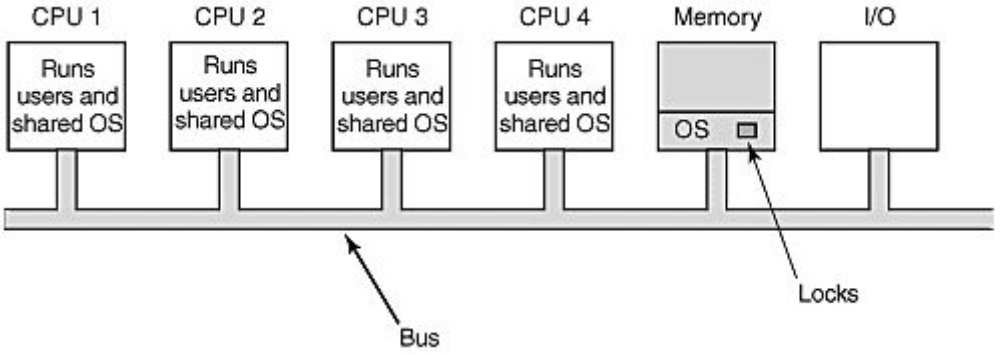


Такой подход решает почти все проблемы предыдущего: список процессов хранится централизованно, и когда CPU не занят, он может запросить у ОС выдать ему процесс. Динамическое выделение памяти позволяет оптимально распределять память между CPU, и поскольку теперь есть только один кеш, конфликты никогда не возникнут.

Минус:
При использовании многих CPU, master начинает тормозить, поскольку он должен обрабатывать системные вызовы со всех CPU. Поэтому для большого количества CPU такой подход неприменим.

1.3. Симметричные процессоры (Symmetric MultiProcessor)

Этот подход избавляется от несимметричности. В памяти хранится одна копия ОС, и каждый из CPU может запускать код ОС. Когда на CPU происходит системный вызов, этот CPU обрабатывает вызов.



Эта модель динамически распределяет память и процессы, поскольку есть всего одна память ОС. Она также решает проблему с пропускной способностью одного процессора, поскольку больше нет master.

Минус:
Если два CPU одновременно запустят код ОС, то все упадет: например, они одновременно запустят один и тот же процесс или попытаются выделить один и тот же свободный кусок памяти. Это решается введением mutex для ОС, но ОС доступной только для одного CPU в определенный момент времени. Иными словами, каждый CPU может запускать ОС, но они не могут делать это одновременно. Из-за этого когда ОС недоступна какому-либо CPU, он простаивает. Такая модель работает почти так же плохо, как предыдущая.

Но ее легко усовершенствовать: многие части ОС не зависят друг от друга. Это позволяет разделить ОС на независимые части, и каждую из них защитить отдельным mutex. Но некоторые части ОС, например, работа с процессами, используются несколькими другими частями. Каждая такая часть должна иметь свой mutex. Это является наибольшей трудностью в разработке таких систем.
Также необходимо специальным образом бороться с взаимными блокировками *(deadlock)*. Если две части ОС зависят от компонентов A и B, и первая сначала запросит A, а вторая B, то возникнет deadlock. Это решается определением специального порядка, в котором компоненты ОС будут вызывать mutex-ы, но это создает трудности для программиста.

2) Многоядерность

2.1) Для чего нужна многоядерность?

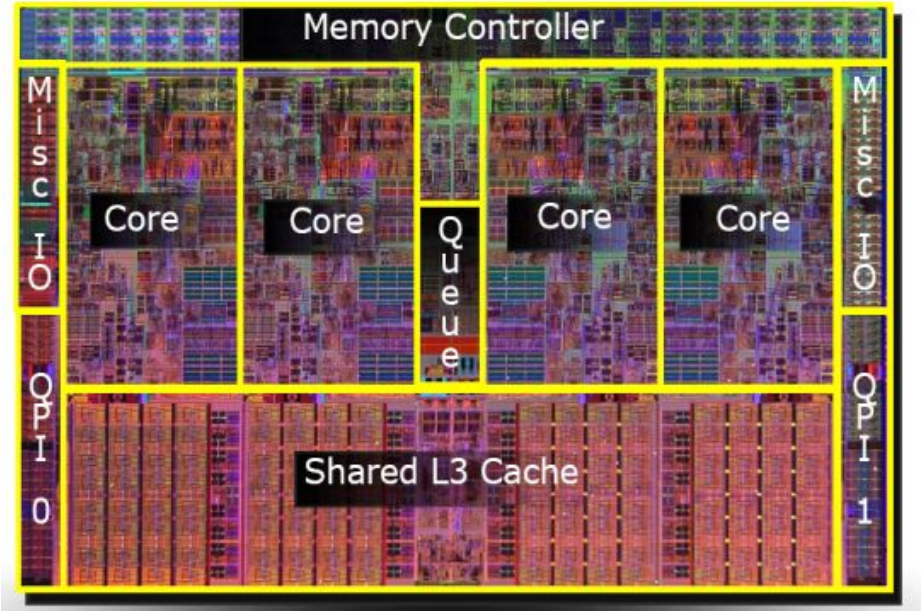
Долгое время повышение производительности традиционных одноядерных процессоров в основном происходило за счет последовательного увеличения тактовой частоты (около 80% производительности процессора определяла именно тактовая частота) с одновременным увеличением количества транзисторов на одном кристалле. Однако дальнейшее повышение тактовой частоты (при тактовой частоте более 3,8 ГГц чипы попросту перегреваются!) упирается в ряд фундаментальных физических барьеров:

1. во-первых, с уменьшением размеров кристалла и с повышением тактовой частоты возрастает ток утечки транзисторов. Это ведет к повышению потребляемой мощности и увеличению выброса тепла
2. преимущества более высокой тактовой частоты частично сводятся на нет из-за задержек при обращении к памяти, так как время доступа к памяти не соответствует возрастающим тактовым частотам

2.2) Какая идея?

Идея весьма проста – вместо разгона одного ядра с 2 до 4 ГГц с напряжением 1.5 В и соответствующим тепловыделением 200 ватт (цифры взяты относительно “с потолка” - к реальности нулевых близки, но никаких точных вычислений я не делал), возьмем два ядра по 2 ГГц с напругой 1 В и тепловыделением 75*2=150 ватт. Как можно заметить, данная схема действительно экономичнее. А работать будет с той же скоростью. Но это уже не точно. Положим, мы имеем некую программку. В первом случае (повышение частоты) скорость работы программы вырастет ~в 2 раза без всяких действий со стороны программиста. Во втором случае - процентов на 25. И то лишь за счёт того, что система будет творить свой

шабаш на втором ядре, а первое отдаст полностью на откуп софтине. Если мы продолжим масштабировать нашу схему до 4/ 6/8 ядер - наша программа быстрее работать не станет. Для ускорения ее работы ленивому программисту нужно самому взять и разделить свой код на треды (потоки) - желательно, чтобы они были относительно независимы. Ибо синхронизация тредов - занятие не из самых простых. К сожалению, нормально пользоваться множеством тредов начинают только сейчас. Собственно, необходимость разбиения кода на треды – лишь одна из немногих проблем мультипроцессорных/ядерных систем. Но остальные были описаны выше, повторять их мы не станем (когерентность кэшей, NUMA).



Микроснимок четырехъядерного процессора Intel с кодовым именем Nehalem. Выделены отдельные ядра, общий кэш третьего уровня, а также линки QPI к другим процессорам и общий контроллер памяти.

2.3) Основные проблемы создания многоядерных процессоров

- каждое ядро процессора должно быть независимым, – с независимым энергопотреблением и управляемой мощностью;

2.4) Преимущества многоядерных процессоров

- возможность распределять работу программ, например, основных задач приложений и фоновых задач операционной системы, по нескольким ядрам;
- увеличение скорости работы программ, которые умеют использовать несколько ядер;
- более эффективное использование требовательных к вычислительным ресурсам мультимедийных приложений (например, видеоредакторов);
- снижение энергопотребления за счет возможности отключить некоторые ядра

2.5) Недостатки многоядерных процессоров

- возросшая себестоимость производства многоядерных процессоров (по сравнению с одноядерными) заставляет чипмейкеров увеличивать их стоимость, а это отчасти сдерживает спрос;
- так как с оперативной памятью одновременно работают сразу два и более ядра, необходимо «научить» их работать без конфликтов;
- возросшее энергопотребление (в случае полной загрузки всех ядер) требует применения мощных схем питания;
- требуется более мощная система охлаждения;
- количество оптимизированного под многоядерность программного обеспечения ничтожно мало (большинство программ рассчитаны на работу в классическом одноядерном режиме, поэтому они просто не могут задействовать вычислительную мощь дополнительных ядер);

3) SMT\HT

3.1) Что такое SMT?

Вспомним прекрасную схему суперскаляра. В соответствии с ней у нас на одно ядро приходится несколько конвейеров. Но что если у нас все команды одного из тредов зависят друг от друга и их не раскидать по разным конвейерам? Не оставлять же их простаивать? Для решения этой проблемы была придумана штука, называющаяся Simultaneous MultiThreading (*частная реализация Intel - HyperThreading*). *И*дея проста - мы берём и говорим, что одно физическое ядро есть два (на самом - любое число, но на практике применяется двойка) логических. Всё, теперь мы можем на одном суперскаляре выполнять два потока команд (т.е. в вышеупомянутой ситуации, когда загружен только один конвейер, мы можем как минимум загрузить ещё один из конвейеров - очевидно, что совершенно другой тред не будет зависим от команд первого). На практике такой хак зачастую дает ~25-30% прироста производительности. Но работает он лишь в тех случаях, когда тредов больше, чем физ. ядер и когда операционная система адекватно эти треды распределяет (т.е. не кидает два треда на одно ядро во время простаивания второго ядра). О-очень редко (на практике - лишь в бенчмарке Linpack) SMT даёт замедление - происходит это из-за “драк за кэш”. Это ситуация, когда у нас изначальный тред еле-еле, но умещался в кэш (не важно какого уровня), а докинув ещё один тред - данные умещаться перестали. Вот и получится, что сначала второй тред выпихнет из кэша данные первого, потом - первый пихнет данные второго и т.д.

3.2) Подробнее про HT (она же HTT)

В недрах компании Intel родилась *Hyper-Threading Technology (HTT)* – технология сверхпоточной обработки данных, которая позволяет процессору выполнять в одноядерном процессоре параллельно два программных потоков одновременно. Hyper-threading значительно

повышает эффективность выполнения ресурсоемких приложений (например, связанных с аудио- и видео редактированием, 3D-моделированием), а также работу ОС в многозадачном режиме. Процессор Pentium 4 с включенным Hyper-threading имеет одно физическое ядро, которое разделено на два логических, поэтому операционная система определяет его, как два разных процессора (вместо одного). Заметим, что добавление HT очень дешевое - 3% на кристалле, в то время когда добавить +1 ядро = много ресурсов.

3.3) Принцип работы HT

Процессор, поддерживающий технологию hyper-threading:

1. может хранить состояние сразу двух потоков;
2. содержит по одному набору регистров и по одному контроллеру прерываний (APIC) на каждый логический процессор.

Остальные элементы физического процессора являются общими для всех логических процессоров.

Рассмотрим пример. Физический процессор выполняет поток команд первого логического процессора. Выполнение потока команд приостанавливается по одной из следующих причин:

- произошел промах при обращении к кэшу процессора;
- выполнено неверное предсказание ветвления;
- ожидается результат предыдущей инструкции.

Физический процессор не будет бездействовать, а передаст управление потоку команд второго логического процессора. Таким образом, пока один логический процессор ожидает, например, данные из памяти, вычислительные ресурсы физического процессора будут использоваться вторым логическим процессором.

4) Закон Амдала

Пусть необходимо решить некоторую вычислительную задачу. Предположим, что её алгоритм таков, что доля α от общего объёма вычислений может быть получена только последовательными расчетами, а, соответственно, доля 1−α может быть распараллелена идеально (то есть время вычисления будет обратно пропорционально

числу задействованных узлов p). Тогда ускорение, которое может быть получено на вычислительной системе из p процессоров, по сравнению с однопроцессорным решением не будет превышать величины:

$$S_p = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

5) Потоковые процессоры

В потоковых процессорах используется иная компенсация латентности памяти, называемая SuperHyperThreading. На современных процессорах площадь кристалла занятая выч устройствами оч маленькая, кучу места занимает вспомогательные блоки: кеш , планировщик, менеджеры разные - но они не вычисляют. Если выкинем кеш и всякие вспомогательные блоки и наставим вычислителей, то скорость вычислений будет соответственно больше. Используем параллельности уровня 1к - 10к тредов. То внутри что-то происходит выч-эффективное. К примеру обработка изображений. С каждой точкой что-то нужно сделать. В итоге можно сделать параллельность = # точек в картинке. Скорость обработки одной точки не интересует, а интересует за сколько обработаем всю картинку. то есть нужно специфическое железо.

Как это работает?

- много ядер (сотни)
- хитрый способ эффективной работы с памятью

Как без кеша обходимся?

Кеш помогает в скорость доступа к памяти. Есть другой вариант с долгим откликом памяти.

Можно сделать высокий уровень CMT. на ядро отправим 40 тредов. Будет не совсем смт, так как в смт параллельно выполняются. А теперьу нас будет в очереди 40 тредов но теперь берем первый

тред и выполняем его пока можем выполнять, пока не найдем команду обращения к памяти. затем выполняем запрос к памяти и прерываем тред. затем повторяем со вторым тредом из очереди. и когда придем в первому треду по кругу, то уже запрос из памяти пришел. где храним? много регистров. В итоге можем не делать кеш, а для нашего кода не будет задержки от отклика памяти. по сути прячем время отклика памяти за счет большого кол-ва вычислений. просто переходим к другим вычислениям пока ждали бы отклик памяти. Много тредов значит нужно много каналов. Если не успеваем, значит не успеваем данные передавать. То есть мы упираемся в скорость передачи.
Пример потокового процессора: видеокарточка (??? спросить у скакова). В видеокарточках есть кеш, но его задача: совмещение запросов памяти.
Затем появились различные API для работы с видеокарточками: OpenCL, CUDA.

6) FAQ

- В чем различие между потоками и процессами?
  Ответ: поток - есть составляющая процесса. У потоков внутри процесса одна и та же память, в то время как один процесс не имеет доступ к памяти другого процесса
- Почему раньше никому не нужна была многопоточность?
  Ответ: так как отсутствовали многопоточные приложения
- Почему раньше никто не создавал многопоточность?
  Ответ: так как люди могли улучшать железку просто поднятием частоты
- Как надо изменить ядро, чтобы оно могло распараллеливать в HT?

Ответ: Так как 2 исполнителя, значит у нас 2 ветки кода, значит надо распараллелить регистры. Также надо допилить планировщик, что он понимал, что команды из разных тредов относятся к разным регистрам. Это нужно, чтобы заполнить конвейер.

- Что должно быть выполнено, чтобы HT давало ускорение?

Ответ: 1) суперскаляр не должен быть нагружен; 2) Потоков должно быть больше, чем ядер

- Какое ускорение дает HT?

Ответ: Логически - зависит от софта, физически - от ядер. Максимум в два раза, минимум в отрицательное. Пример с отрицательным ускорением: есть поток который хорошо загружает, использует 20 кб кеша, в итоге поток умещается в кеше и ему хорошо. если 2 таких потока на ядре, то и обращаются к данным каждый к своим и кеша нужно больше. то есть 40 кб кеша, а у нас только 32. то есть не все помещаются. то есть нам придется часто ходить в RAM, проиграем в уровне кешировании, но не проиграем на уровне вычислений.

- Чья задача распределять потоки?

  Ответ: операционной системы.
- Что если мы хотим экономить энергию?

  Ответ: запихаем треды на включенные ядра и отключим другие.