РАЗГОН И ТОРМОЖЕНИЕ WINDOWS NT

i486C-ядру посвящается…

КРИС КАСПЕРСКИ

Разработчики ядра исполнительной системы говорят, что оно дает пищу всему остальному. И это отнюдь не преувеличение! На плохом фундаменте ничего хорошего не построишь, и качество ядра в значительной мере определяет производительность всей операционной системы в целом. В комплект поставки Windows NT входит большое количество разнообразных ядер (в том числе и нашумевшее ядро i486C, по слухам значительно увеличивающее быстродействие системы). Как оценить их производительность? Обычные тестирующие пакеты для этого не подходят, и адекватную методику измерений приходится разрабатывать самостоятельно с учетом архитектуры ядра Windows и специфической направленности возложенных на него задач.

Большинство пользователей и системных администраторов живут с ядром, назначенным операционной системой при ее инсталляции, совершенно не задумываясь о том, что его можно заменить другим. В штатный комплект поставки Windows NT входит более десятка различных ядер, и еще большее их количество может быть найдено на просторах Интернета. Некоторые производители аппаратного обеспечения оптимизируют ядра под свои машины, и зачастую эта оптимальность сохраняется на большинстве остальных. Существует мнение, что древние ядра (все еще совместимые с новомодными версиями Windows) намного производительнее современных, хотя и уступают им по функциональности.

Отдельного разговора заслуживает история с ядром i486C, оптимизированным под 80486-машины. Оно входило во все версии Windows вплоть до NT 4.0, но затем неожиданно исчезло, и Windows 2000 вышла уже без него. Однако с Windows XP все вернулось вновь. Говорят, что оно здорово увеличивает производительность системы. Автор, знакомый с ним еще со времен Windows NT 4.0, подтверждает – да, увеличивает, особенно на медленных машинах. Для теоретического обоснования данного факта и была написана эта статья.

Разумеется, само по себе ядро i486C не настолько интересно, чтобы уделять ему 13 журнальных полос. Давайте мыслить более глобально. Ядер много, а инструментов для оценки их производительности не существует (во всяком случае в открытом доступе нет ни одного). Тем не менее такой инструмент при желании можно разработать и самостоятельно, о чем я, собственно, и собираюсь рассказать.

Структура ядра

Ядро Windows NT состоит из двух ключевых компонентов: executive system – исполнительной системы (далее по тексту KERNEL), реализованной в файле ntoskrnl.exe, и библиотеки аппаратных абстракций – Hardware Abstraction Layer (сокращенно HAL), представленной файлом HAL.DLL. На самом деле имена файлов могут быть любыми, и в зависимости от типа ядра они варьируются в довольно широких пределах.

Исходная концепция построения Windows NT предполагала сосредоточить весь системно-зависимый код в HAL, используя его как фундамент для воздвижения системно-независимой исполнительной системы. Тогда для переноса ядра на новую платформу было бы достаточно переписать один HAL, не трогая ничего остального (по крайней мере теоретически). В действительности же это требование так и не было выполнено, и большое количество системно-зависимого кода просочилось в исполнительную систему, а HAL превратился в сплошное нагромождение неклассифицируемых функций, тесно переплетенных с исполнительной системой, так что двухуровневая схема организации ядра в настоящее время выглядит довольно условной.

Исполнительная система Windows NT реализует высокоуровневые функции управления основными ресурсами (как то: памятью, файлами, процессами и потоками), в определенном смысле являясь операционной системой в миниатюре. Большинство этих функций слабо связаны с конструктивными особенностями конкретной аппаратуры. Они практически не меняются от одной исполнительной системе к другой и одинаково производительны (или непроизводительны) во всех ядрах.

Обособленная часть исполнительной системы, реализующая наиболее низкоуровневые операции и тесно взаимодействующая с библиотекой аппаратных абстракций, называется ядром (KERNEL). Большинство ядерных процедур предназначены для сугубо внутреннего использования и не экспортируются (хотя присутствуют в отладочных символах), а те, что экспортируются, обычно начинаются с префикса Ke (подпрограммы ядра) или Ki (обработка прерываний в ядре).

Это уже третье упоминание ядра, которое мы встречаем, что создает определенную путаницу. Давайте попробуем разложить образовавшийся терминологический кавардак по полочкам. На самом верхнем уровне абстракций ядром принято называть совокупность компонентов операционной системы, работающих на привилегированном кольце нулевого уровня. Спустившись пониже, мы увидим, что ядро отнюдь не монолитно и состоит как минимум из двух частей: собственно самого ядра и загружаемых драйверов. Ядро Windows NT реализовано в двух файлах: библиотеке аппаратных абстракций (по сути дела являющееся набором первичных драйверов) и исполнительной системе. Выбором исполнительной системы руководит ключ KERNEL

файла boot.ini, поэтому многие ассоциируют ее с ядром, хотя это и не совсем верно, но не будем докапываться до столба, ведь фундамент исполнительной системы – тоже ядро. И это еще далеко не все! Подсистемы окружения (win32, POSIX, OS/2) имеют свои собственные ядра, сосредоточенные в прикладных библиотеках непривилегированного режима третьего кольца, и общаются с ядром Windows NT через специальную «прослойку», реализованную в файле NTDLL.DLL. Ядра подсистем окружения представляют собой сквозные переходники к ядру Windows NT и практически полностью абстрагированы от оборудования. Практически, но не совсем! Некоторая порция системно-зависимого кода присутствует и здесь. Многопроцессорные версии файлов NTDLL.DLL и KERNEL32.DLL для синхронизации потоков используют машинную команду LOCK. В однопроцессорных версиях она теряет смысл и заменяется более быстродействующей командой NOP. Наверняка существуют и другие различия, но я такие специально не искал, поскольку их влияние на производительность системы незначительно.

Из всего этого зоопарка нас в первую очередь будет интересовать ядро исполнительной системы и HAL (рис. 1).

Типы ядер

Тип выбираемого ядра определяется как архитектурными особенностями конкретной аппаратной платформы, так и личными предпочтениями пользователя системы, обусловленными спецификой решаемых задач. Существует по меньшей мере пять основных критериев классификации ядер:

- тип платформы (Intel Pentium/Intel Itanium, Compaq SystemPro, AST Manhattan);
- количество процессоров (однопроцессорные и многопроцессорные ядра);
- количество установленной памяти (до 4 Гб, свыше 4 Гб);
- тип контроллера прерываний (APIC- и PIC-ядра);
- тип корневого перечислителя (ACPI- и не ACPI-ядра).

Очевидно, что ядро должно быть совместимо с целевым процессором на уровне двоичного кода и работать в наиболее естественном для него режиме (64-разрядный процессор, поддерживающий архитектуру IA32, сможет работать и со стандартным 32-разрядным ядром, но разумным такое решение не назовешь). Данная статья обсуждает вопросы оценки сравнительной производительности ядер

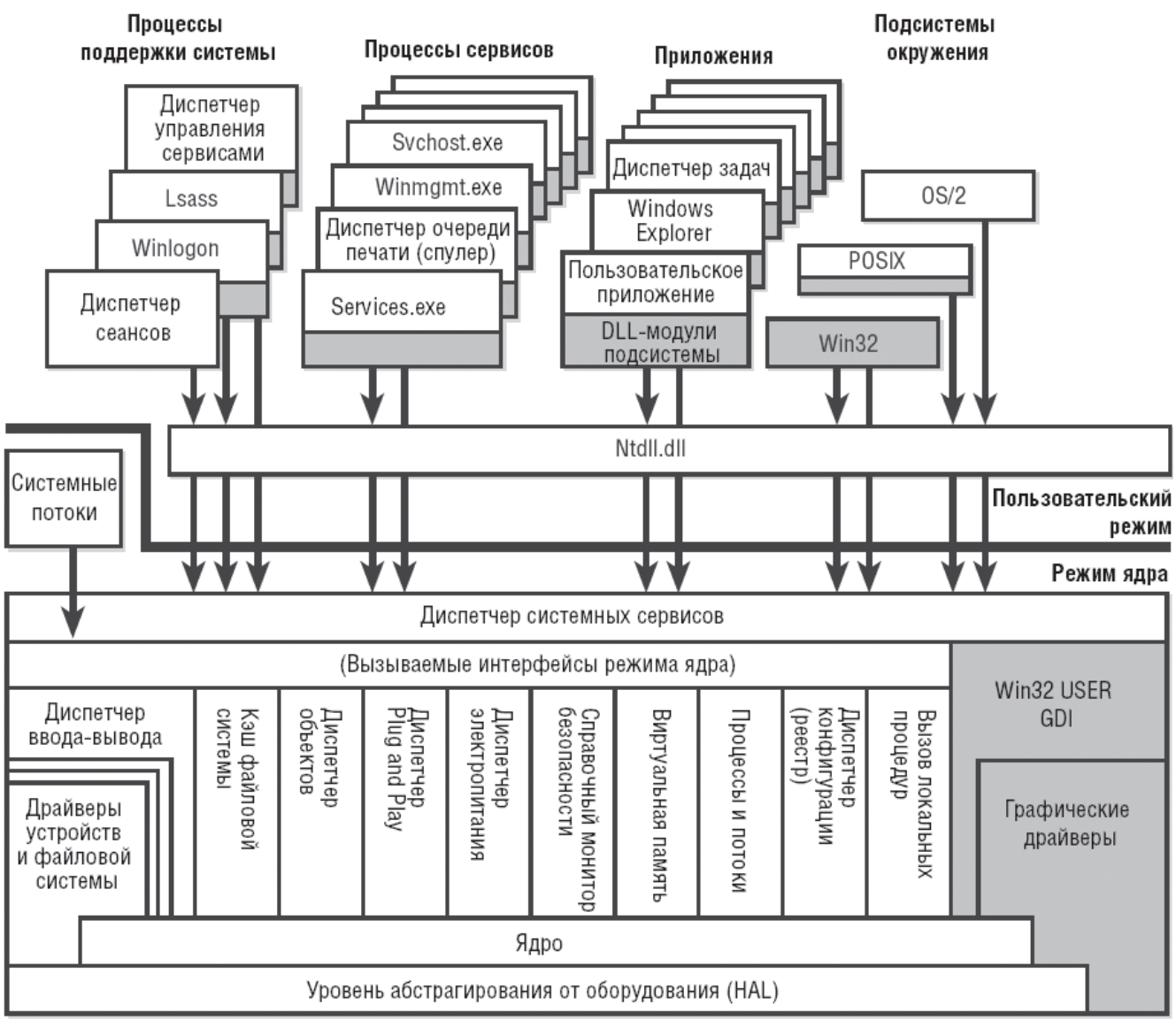

Рисунок 1. Архитектура ядра Windows NT

в рамках одной аппаратной платформы, и тема выбора процессора здесь не затрагивается.

Многопроцессорные ядра отличаются от монопроцессорных прежде всего тем, что они «видят» все установленные процессоры и умеют с ними взаимодействовать, возлагая эту задачу на специальный драйвер, встроенный в HAL. Также в них кардинально переработаны механизмы синхронизации. Если в монопроцессорных ядрах для предотвращения прерывания критичного участка кода достаточно всего лишь подтянуть IRQL к верхнему уровню или заблокировать прерывания командной CLI, то в многопроцессорных ядрах такая стратегия уже не срабатывает – ведь на всех остальных процессорах прерывания разрешены – и приходится прибегать к спинлокам (от английского spin lock – взаимоблокировка).

Для защиты участка кода от вмешательства извне система взводит специальный флаг, складывая поступающие запросы в специальную очередь. Естественно, это требует некоторого количества процессорного времени, что негативно сказывается на производительности, но другого выхода у нас нет. Значительно усложняется и схема диспетчеризации прерываний, ведь теперь один набор IRQ приходится делить между несколькими процессорами, а таблицы обработчиков аппаратных/программных прерываний поддерживать в согласованном состоянии. Изменения коснулись и планировщика, а точнее самой стратегии планирования потоков, которая может быть реализована как по симметричной, так и по асимметричной схеме. Симметричные ядра (а их большинство) допускают выполнение каждого из потоков на любом свободном процессоре, асимметричные же жестко закрепляют системные потоки за одним из процессоров, выполняя пользовательские потоки на всех остальных. Асимметричные ядра не входят в стандартный комплект поставки Windows NT и обычно предоставляются поставщиками соответствующего оборудования. Асимметричные ядра несколько менее производительны, чем симметричные (один из процессоров большую часть своего времени простаивает), однако их намного сложнее «затормозить», и сколько бы прожорливых потоков ни запустил злобный хакер, администратор всегда может обезоружить их, ведь системные потоки выполняются на отдельном процессоре! Многопроцессорные ядра следует использовать только на многопроцессорных системах, в противном случае мы значительно проиграем в производительности, причем многопроцессорные материнские платы с одним процессором на борту требуют специального унипроцессорного ядра (uniprocessor kernel), а работоспособность однопроцессорных ядер в такой конфигурации никем не гарантирована (хотя обычно они все-таки работают) (рис. 2).

Разрядность внешней адресной шины младших моделей процессоров Intel Pentium составляет 32 бита, и потому они не могут адресовать более 4 Гб физической памяти. Поскольку для серьезных серверов и мощных рабочих станций этого оказалось недостаточно, начиная с Pentium Pro, ширина шины была увеличена до 36 бит, в результате чего

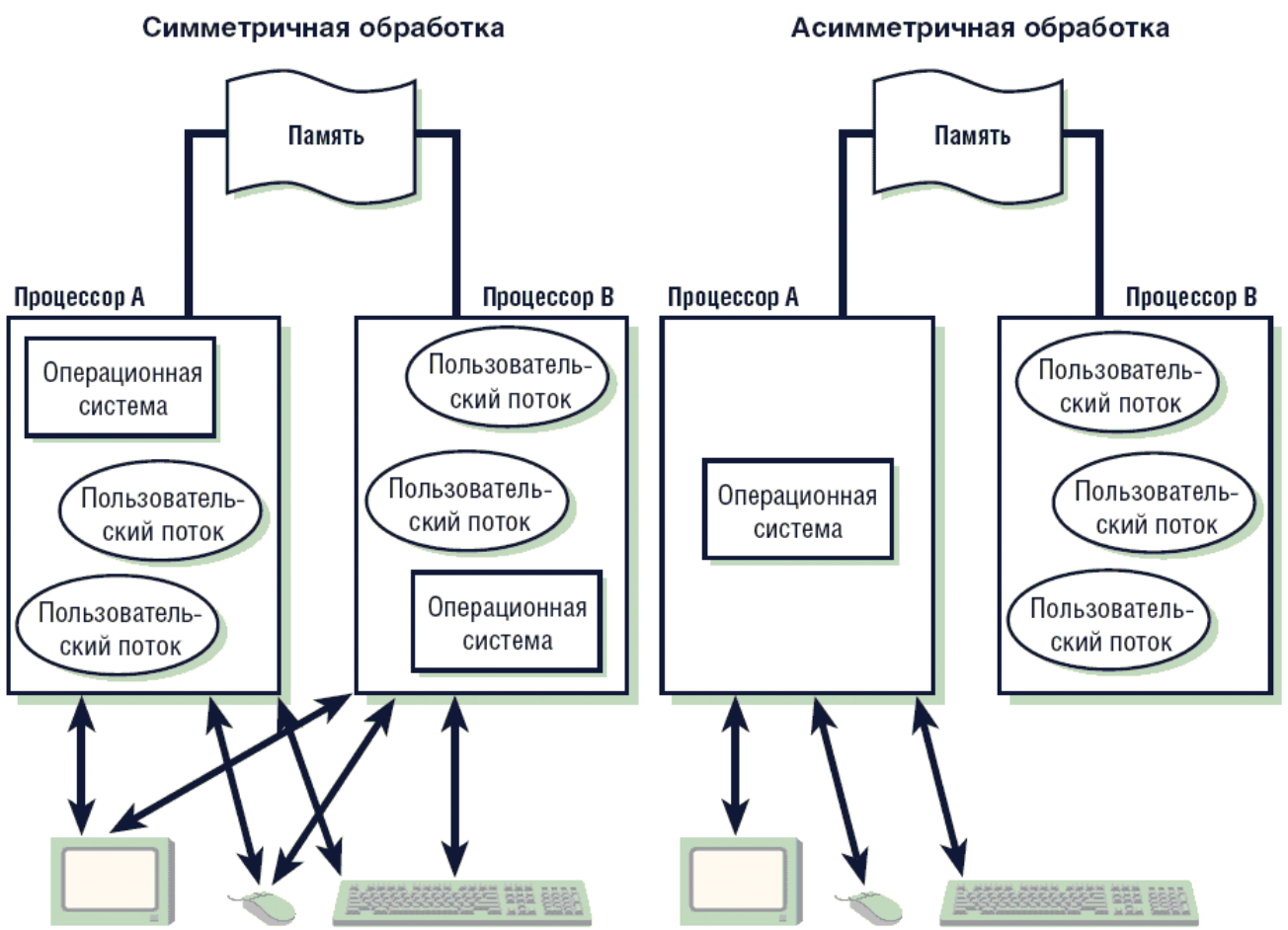

Рисунок 2. Симметричная и асимметричная обработка

мы получили возможность адресовать вплоть до 64 Гб физической памяти. При работе в обычном режиме страничной адресации четыре старших бита адресной шины обнуляются, и чтобы их задействовать, необходимо перевести процессор в режим PAE (Physical Address Extensions), отличающийся структурой таблиц страничной переадресации и поддерживающий 2 Мб страницы памяти. PAE-ядра несколько производительнее обычных ядер, поскольку засовывают старшие 2 Мб адресного пространства процесса в одну страницу, сокращая тем самым издержки на переключение контекста между процессами. Вы можете использовать PAE-ядра, даже если у вас установлено менее 4 Гб физической памяти, однако выигрыш в производительности при этом будет не очень существенен.

В зависимости от типа контроллера прерываний, установленного на материнской плате, следует выбирать либо PIC-, либо APIC-ядро. PIC-контроллеры поддерживают 15 IRQ и встречаются только на многопроцессорных материнских платах.

APIC-контроллеры поддерживают до 256 IRQ и многопроцессорную обработку. На программном уровне PIC- и APIC-контроллеры взаимно совместимы, поэтому PIC-ядро должно работать и с APIC-контроллером, однако, во-первых, при этом оно увидит всего лишь 15 IRQ, а во-вторых, такая конфигурация не тестировалась Microsoft, и потому никаких гарантий, что система не зависнет при загрузке, у нас нет.

Материнские платы с поддержкой технологии ACPI могут работать как с ACPI, так и с не ACPI-ядрами, при этом не ACPI-ядра самостоятельно распределяют системные ресурсы компьютера и взаимодействуют с устройствами напрямую, а ACPI-ядра во всем полагаются на ACPI-контроллер, фактически являющийся корневым перечислителем, т.е. самой главной шиной компьютера, к которой подключены все остальные шины и устройства. И хотя эта шина виртуальна, производительность системы значительно падает, поскольку ACPI-контроллер имеет тенденцию вешать все PCI-устройства на одно прерывание со всеми вытекающими отсюда последствиями.

Подробнее обо всем этом и многом другом я расскажу по ходу углубления внутрь статьи, а пока же сосредоточим свое внимание на ядре как таковом, приоткрыв черный ящик.

Почему не пригодны тестовые пакеты

Среди изобилия тестовых программ, представленных на рынке, можно найти множество утилит для оценки быстродействия центрального процессора, оперативной памяти, видеокарты или подсистемы ввода/вывода, но мне неизвестны инструменты, пригодные для определения производительности ядра операционной системы.

А почему бы не воспользоваться WINSTONE или SPECweb96? Первый имитирует запуск реальных офисных приложений, второй – веб-сервера. Разве их показания не будут отражать объективное влияние конкретного ядра на производительность всей операционной системы в целом? Нет, не будут. И вот почему. WINSTONE (как и большинство его соплеменников) прогоняет тесты в идеализированных условиях при минимальном количестве потоков, поэтому накладные расходы на переключение контекста не учитываются. К тому же степень «отзывчивости» системы (подсознательно ассоциируемая с ее производительностью) обусловливается отнюдь не скоростью выполнения кода, а интеллектуальностью планировщика, повышающего приоритет потока в тот момент, когда он больше всех остальных нуждается в процессорном времени (например, при захвате фокуса, завершении ожидаемой операции ввода/вывода и т. д.). Но тесты этого не учитывают.

Допустим, пересчет миллиона ячеек электронной таблицы, независимо от версии ядра, длится ровно один час. Означает ли это, что все ядра равноценны? Вовсе нет! За кадром остались такие жизненно важные аспекты, как «подвижность» фоновых потоков системы, «чувствительность» к прерываниям и т. д. Одно ядро достойно обрабатывает клавиатурный ввод одновременно с расчетом, а другое жутко тормозит, реагируя на нажатия с задержкой в несколько секунд. Ну и с каким из них вам будет приятнее работать? А ведь тестирующей программе все равно…

Зайдем с другой стороны. Обычно тестовое задание состоит из серии повторяющихся замеров, время выполнения которых усредняется. Замеры со значительными отклонениями от средневзвешенного значения отбрасываются (ну мало ли, может, в этот момент началась отгрузка кэша на диск). Если продолжительность одного замера составляет не более 20 мс (целая вечность для процессора!), за это время может вообще не произойти ни одного переключения контекста, а если оно и произойдет, то будет безжалостно отбраковано при обработке результатов, в итоге мы получим «чистую» производительность машины за вычетом вклада операционной системы. Можно ли этого избежать? Увы! В противном случае результаты тестирования будут варьироваться от прогона к прогону, и пользователь растеряется – какое же значение ему выбрать? Ведь на коротком промежутке времени (порядка 10 – 20 мс) издержки от побочных эффектов крайне непостоянны и неоднородны, поэтому выдавать неочищенный результат нельзя.

Если же продолжительность замера превышает 20 мс, планировщик Windows автоматически перераспределяет процессорное время так, чтобы переключение контекста основного потока (и этим потоком будет тестовый поток!) происходило как можно реже, а его накладные расходы стремились к нулю. Естественно, остальные потоки системы сажаются на голодный паек, работая рывками (и, как мы увидим далее, даже при умеренной загрузке системы на «плохих» ядрах потоки получают управление не чаще чем один раз… в десять секунд. Не «тиков», а именно секунд! И хотя интегральная производительность системы не только не уменьшается, но даже возрастает, работать с ней становится невозможно).

То же самое относится и к имитатору веб-сервера. Допустим, одно ядро обрабатывает сто тысяч запросов за минуту, а другое – сто пятьдесят. Какое из них производительнее? А если первое обслуживает всех своих клиентов плавно, а второе отдает информацию «плевками» с паузами по десять-пятнадцать секунд? К сожалению, известные мне тесты этого обстоятельства не учитывают и потому их показания при всей своей объективности толкают на выбор неправильного ядра. Помните анекдот про машину, которая

ездит быстро, но тормозит медленно? Производительность – это не только отношение количества проделанной работы к затраченному времени, это еще и качество предоставляемого сервиса!

Сформулируем главное требование, предъявляемое нами к системе: планировщик должен исхитриться перераспределить процессорное время между потоками так, чтобы обеспечить наилучший баланс между производительностью и плавностью работы даже при большом количестве одновременно выполняющихся потоков. Теперь для достижения полного счастья остается лишь найти ядро своей мечты.

Обсуждение методик тестирования

Скажите, какую физическую величину измеряет обыкновенный ртутный или спиртовой термометр. Температуру? Нет, объем. Каждый тест по-своему объективен и что-то измеряет, но результат измерений в значительной мере определяется его интерпретацией. Объем ртутного шарика прямо пропорционален его температуре. Это хорошо. Но производительность не температура. Это комплексная величина, и в индексах производительности столько же смысла, сколько и в коэффициенте интеллектуальности отдельного индивидуума. Кто умнее – Гейзенберг или Галуа? Кто сильнее – кит или слон? Какое из всех ядер самое производительное? Можно спорить до хрипоты, но в такой формулировке вопрос не имеет ответа. Как минимум мы должны выявить факторы, в наибольшей степени влияющие на совокупное быстродействие системы, и разработать адекватные методики тестирования, разлагающие векторное понятие производительности на скалярные величины.

Никакая методика тестирования не безупречна и отражает лишь часть реальной действительности, поэтому к полученным результатам следует относиться с большой осторожностью. Если Формула-1 ездит быстрее, чем КамАЗ, отсюда еще не следует, что она сохранит свое преимущество при перевозке двухсот тонн кирпичей.

Методика тестирования тесно связана со спецификой возлагаемых на систему задач (у автогонщика свои требования к машине, у дальнобойщика – свои), поэтому никаких конкретных решений здесь не приводится, напротив, эта статья показывает, как разрабатывать тестовые методики самостоятельно и какие проблемы поджидают вас на этом пути. А проблем будет много.

Разность таймеров

Современные материнские платы несут на борту несколько независимых таймеров, которые никто не калибровал и каждый из которых врет слегка по-своему. Различные ядра используют различные таймеры для подсчета времени и системного планирования, поэтому отталкиваться от API-функций типа GetTickCount или QueryPerfomanceCounter для сравнения производительности ядер категорически недопустимо! Ну во всяком случае без их предварительной калибровки.

Первым (а на IBM XT еще и единственным) возник программируемый таймер интервалов – Programmable Interval Timer, или сокращенно PIT, базирующийся на микросхеме Intel 8254 (сейчас – 82C54) и тактируемый частотой 1.19318 МГц (сейчас – либо 1.19318, либо 14.31818 МГц, причем последняя встречается намного чаще), что обеспечивало ему превосходную по тем временам точность измерений – порядка 0.84 мс (~1 мс с учетом накладных расходов). В Windows продолжительность одного тика таймера составляет 10 мс. Каждые 10 мс таймер дергает прерыванием, и закрепленный за ним обработчик увеличивает системное время на эту же величину. Если обработчик по каким-либо причинам проморгает таймерное прерывание (аппаратные прерывания запрещены инструкцией CLI или перепрограммированием PIC-контроллера), системное время начнет отставать от реального, существенно снижая точность измерений. Хуже того, размеренность хода PIT далеко не идеальна и варьируется в довольно широких пределах. Windows использует PIC-таймер в основном для планирования потоков, а для измерения времени стремится использовать другие, более точные таймеры, и переходит на PIT только тогда, когда ни один из них не доступен.

Таймер часов реального времени (Real Time Clock или сокращенно RTC), впервые появившийся в IBM AT, обычно тактируется частотой 32.768 кГц, автоматически обновляя счетчик времени в CMOS, что не требует наличия программного обработчика прерываний. Частота обновления по умолчанию составляет 100 Гц, но при желании RTC-таймер может быть перепрограммирован, и тогда часы будут идти или медленнее, или быстрее. Точность показаний зависит как от состояния питающей батарейки (которая вообще-то никакая не батарейка, а настоящий литиевый аккумулятор, но это уже не важно), так и от добротности реализации микросхемы RTC со всеми обслуживающими ее компонентами. По заверениям производителей среднее время ухода за день составляет порядка 1-2 сек, однако имеющиеся у меня материнские платы врут намного сильнее, к тому же некоторые из них обнаруживают значительные «биения» на временных интервалах порядка десятых долей секунды, что отнюдь не способствует точности измерений. К тому же некоторые версии Windows периодически синхронизируют системное время с часами реального времени, что еще больше подрывает доверие к системному времени. Используя его для измерения продолжительности тех или иных процессов, вы можете получить очень странный результат. Часы реального времени используют многие тестовые программы и перепрограммирование RTC-таймера позволяет фальсифицировать результат. Windows 2000 использует RTC только для периодической синхронизации с системными часами.

Усовершенствованный контроллер прерываний (Advanced Programmable Interrupt Controller, или сокращенно APIC), в основном использующийся в многопроцессорных системах, помимо прочей оснастки включает в себя и некоторую пародию на таймер, предназначенный для планирования потоков и не пригодный ни для каких измерений ввиду своей невысокой точности. Однако по непонятным причинам APIC-ядра используют APIC-таймер в качестве основного таймера системы, при этом величина одного «тика» составляет уже не 10 мс, а 15 мс. Естественно, часы идут с прежней скоростью, но политика планирования существенно изменяется – с увеличением кванта со-

кращаются накладные расходы на переключение контекстов, но ухудшается плавность коммутации между ними. Использовать показания счетчика системного времени для сравнения производительности APIC-ядер с другими ядрами недопустимо, т.к. полученный результат будет заведомо ложным.

Материнские платы, поддерживающие ACPI (Advanced Configuration and Power Interface), имеют специальный таймер, обычно управляемый менеджером электропитания и потому называющийся Power Management Timer, или сокращенно PM. Еще его называют ACPI-таймером. Штатно он тактируется частой 3.579545 МГц (тактовая частота PIT, разделенная на четыре), что обеспечивает точность измерений порядка ~0.3 мс. ACPI-ядра используют PM-таймер в качестве основного таймера системы, чему сами не рады. Чипсеты от VIA, SIS, ALI, RCC не вполне корректно реализуют PM-таймер, что приводит к обвальному падению производительности операционной системы и снижению надежности ее работы. Проблема лечится установкой соответствующего пакета обновления, подробнее о котором можно прочитать в технической заметке Q266344. Разумеется, исправить аппаратную проблему (а в данном случае мы имеем дело именно с ней) программными средствами невозможно, и ее можно лишь обойти. Но даже на правильном чипсете при высокой загрузке PCI-шины PM-таймер не успевает своевременно передавать свои тики, и хотя они при этом не пропадают, обновление счетчика времени происходит «рывками», для преодоления которых Microsoft рекомендует сверять показания PM с показаниями PIC/APIC или RTC.

И если PM неожиданно прыгнет вперед (jump forward), обогнав своих соплеменников, этот замер должен аннулироваться как недействительный.

Современные чипсеты (и, в частности, Intel 845) содержат специальный высокоточный таймер (High Precision Event Timers, или сокращенно HPET), тактируемый частотой от 10 МГц, при которой время одного тика составляет от 0.1 мс при точности порядка ±0.2% на интервалах от 1 мс до 100 мс. Это действительно рекордно высокая точность, по крайней мере на порядок прерывающая точность всех остальных таймеров, однако HPET все еще остается завидной экзотикой, и чипсеты с его поддержкой пока еще не очень широко распространены.

Помимо этого на материнской плате можно найти множество таймеров, например, PCI Latency Timer или десятки таймеров, обслуживающих чипсет, шины, память и прочие системные устройства. Многие из них тактируются частотами PCI- или AGP-шины, что обеспечивает достаточно высокую точность измерений (ниже, чем у HPET, но существенно выше, чем у PM). К сожалению, они в своей массе не стандартизированы и на каждой материнской плате реализуются по-своему, если вообще реализуются.

Некоторые используют в качестве таймера команду RDTSC, считывающую показания внутреннего счетчика процессора, каждый такт увеличивающегося на постоянную величину (как правило единицу). Для профилировки машинного кода она подходит на ура, но вот на роль беспристрастного метронома уже не тянет. Некоторые ACPI-контроллеры динамически изменяют частоту процессора или усыпляют его в паузах между работой для лучшего охлаждения. Как следствие – непосредственное преобразование процессорных тактов в истинное время оказывается невозможным.

Утилиту для оценки разности хода нескольких таймеров можно скачать, например, отсюда: http://www.overclockers.ru/cgi-bin/files/download.cgi?file=320&filename=timertest.rar. И если выяснится, что ваши таймеры идут неодинаково, для сравнения производительности различных ядер будет необходимо ограничиться лишь одним из них. Надежнее всего использовать для снятия показаний свой собственный драйвер, поскольку стратегия выбора таймеров ядром системы в общем случае непредсказуема и может отличаться от вышеописанной.

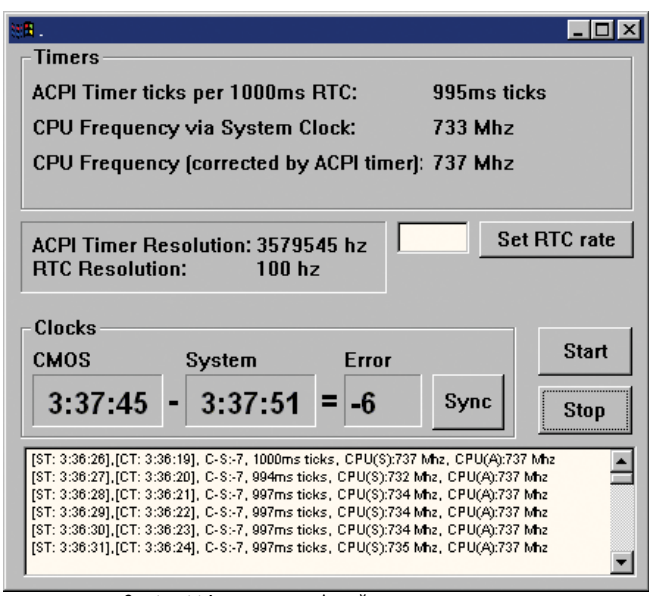

Рисунок 3. testtimer за работой

Рассмотрим некоторые API-функции, используемые тестовыми программами для измерения интервалов времени.

- GetTickCount. Самая популярная функция. Возвращает количество миллисекунд, прошедших со времени последнего старта системы. В зависимости от типа установленного ядра использует либо PIT-, либо APIC-таймеры, в соответствии с чем ее разрешение составляет либо 10 мс, либо 15 мс, причем некоторые тики таймера могут быть пропущены (т.е. за 30 мс не произойдет ни одного увеличения счетчика). Не рекомендуется к употреблению.
- GetSystemTime. Возвращает истинное время. То есть функция думает, что оно истинное, а в действительности – производное от системного счетчика, инкрементируемого каждые 10 мс или 15 мс за вычетом «съеденных» тиков и врожденной неравномерности хода PIT- и APIC-таймеров. Периодически синхронизирует себя с часами реального времени, а если запущена специальная интернет-служба, то еще и с атомными часами, т.е. системное время продвигается траекторией пьяного гонщика, едущего по пересеченной местности. Для измерений временных промежутков непригодна.
- timeGetTime. То же самое, что и GetTickCount. Документация утверждает, что разрешающая способность этой функции составляет 1 мс, в действительности же – 10 мс. Синхронные изменения timeGetTime и GetTickCount подтверждают, что они питаются от одного источника.

- Sleep. Усыпляет поток на указанное количество миллисекунд, задерживая на время управление. Теоретически может использоваться для калибровки других таймеров и вычисления коэффициента перевода таковой процессора в секунды. Практически же… Время ожидания принудительно округляется до величины, кратной «тику» основного системного таймера, причем на момент выхода из сна данный поток должен находиться в самом начале очереди потоков, ожидающих выполнения, в противном случае ему придется подождать. Может быть, доли секунды, а может, несколько минут – все зависит от размеров очереди, а она непостоянна. На хорошо загруженной системе поток, планирующий вздремнуть 100 мс, рискует проснуться… через минуту!
- QueryPerformanceCounter. Возвращает количество тиков наиболее точного таймера, прошедших с момента старта системы или… его последнего переполнения. Является переходником к функции KeQueryPerformance Counter, реализованной в HAL. Windows XP поддерживает HPET, более ранние системы – нет. Если HPET аппаратно доступен и программно поддерживается, используется он, в противном случае ACPI-ядра будут использовать PM, а не ACPI – либо PIT, либо возвратят ноль, сигнализируя об ошибке. Для определения продолжительности одного тика можно использовать функцию QueryPerformanceFrequency, возвращающую его частоту в герцах. Это самое лучшее средство для профилировки и хронометража времени из всех имеющихся, однако, как уже говорилось, PM-таймеры могут идти неточно или совершать неожиданные прыжки вперед, поэтому, показания, возвращенные QueryPerformance Counter, требуют некоторой обработки.

Синхронизация

На машинном уровне межпроцессорная синхронизация обеспечивается префиксом LOCK, предоставляющим команде монопольный доступ к памяти. Все посягательства со стороны остальных процессоров (если они есть) блокируются. Обычно LOCK используется совместно с командами, устанавливающими или снимающими флаги, сигнализирующие о том, что указанная структура данных в настоящий момент модифицируется процессором, и потому лучше ее не трогать.

Многопроцессорные ядра содержат множество LOCK, встречающихся в самых неожиданных местах, и съедающих вполне ощутимый процент производительности, поэтому многопроцессорные ядра всегда медленнее. В однопроцессорных ядрах часть LOCK убрана полностью, вместе с примыкающими к ним флагами, а часть заменена более быстродействующими NOP. Общая же структура ядра сохранена в более или менее неизменном виде (рис. 4).

ACPI и IRQ

Прерывания – их мифическая разводка, неиссякаемый источник слухов, сплетен, легенд и суеверий. Попробуем с ними разобраться.

Все x86-процессоры (за исключением старших моделей Intel Pentium-4) управляют прерываниями через специальный интерфейсный вывод, обычно обозначаемый как INTR, высокий уровень сигнала на котором свидетельствует о поступлении запроса на прерывание. Процессор прекращает текущую работу, вырабатывает сигнал подтверждения прерывания (Interrupt Acknowledge) и считывает с шины данных 8-битный номер вектора перекрывания (INT n), переданный контроллером прерываний, обычно реализуемым на правнучатых племянниках микросхемы i8259 и территориально находящийся в южном мосту чипсета.

За вычетом 32 прерываний, зарезервированных разработчиками процессора, мы имеем 224 прерывания, пригодных для обработки сигналов от периферийных устройств. Означает ли это, что верхнее ограничение на максимальное количество одновременно поддерживаемых устройств равно 224? Нет! Некоторые из древних процессоров имели всего лишь один вектор прерывания, но это ничуть не мешало им управлять десятком устройств одновременно. Как? Да очень просто. Что такое прерывание? Всего лишь способ устройства обратить на себя внимание. При наличии достаточного количества свободных векторов за каждым из устройств может быть закреплено свое персональное прерывание, однозначно указывающее на источник сигнала. Это существенно упрощает как проектирование самих устройств, так и разработку обслуживающих их драйверов, однако отнюдь не является необходимым. Получив сигнал прерывания, процессор может опросить все устройства по очереди, выясняя, кто из них затребовал внимания. Естественно, это достаточно медленная операция, выливающаяся в десятки дополнительных операций ввода/вывода и к тому же потенциально небезопасная, т.к. малейшая ошибка разработчика оборачивается серьезными конфликтами. Короче говоря, для достижения наивысшей стабильности и производительности системы каждое прерывание должно использоваться не более чем одним устройством.

Контроллер прерываний, используемый в IBM XT, поддерживал восемь аппаратных прерываний, обозначенных IRQ (Interrupt Request) и пронумерованных от 0 до 7. Номер IRQ соответствует приоритету прерывания: чем больше номер – тем ниже приоритет. Во время обработки более приоритетных прерываний генерация менее приоритетных подавляется и соответственно, наоборот, менее приоритетные прерывания вытесняются более приоритетными. Считается, чем больше ресурсов требует устройство, тем выше должен быть приоритет его IRQ. Это неверно. Выбор предпочтительного IRQ, определяется отнюдь не «прожорливостью» устройства, а критичностью потери прерывания. Допустим, сетевая карта, видя, что входной буфер практически полон, а данные по витой паре продолжают поступать, сгенерировала прерывание, которое было вытеснено прерыванием звуковой карты, имеющей более высокий приоритет и сигнализирующей об опустошении выходного буфера. Если драйвер звуковой карты ненароком замешкается и удержит обработчик прерывания дольше положенного, входной буфер сетевой карты переполнится, и часть пакетов окажется безвозвратно утеряна, и данные придется передавать вновь, что несколько снизит быстродействие сети. Является ли эта ситуация нормальной? Для кого-то да, а для кого-то и нет! Потерянных пакетов, конечно, жаль, но если поменять приоритеты местами, звуковая карта отреагирует на опустошение входного буфера суровым иска-

жением воспроизводимого сигнала, чего в некоторых ситуациях ни в коем случае допускать нельзя. Никакого другого влияния на производительность выбор приоритетов не оказывает. Независимо от номера IRQ обработка прерывания занимает одно и то же время. От обработки остальных прерываний она также не освобождает. При условии, что аппаратные устройства и обслуживающие их драйверы реализованы правильно (т.е. более или менее безболезненно переживают потерю IRQ и «отпускают» прерывание практически сразу же после его возникновения), выбор приоритетов никакой роли не играет.

Контроллер прерываний позволяет отображать аппаратные IRQ0-IRQ7 на 8 любых смежных векторов прерываний, например, на INT 30h – INT 37h. Тогда, при возбуждении IRQ0, процессор сгенерирует прерывание INT 30h, а при возбуждении IRQ3 – INT 33h. В IBM AT количество контроллеров было увеличено до двух, причем второй был подключен на вход первого, в результате чего количество аппаратных прерываний возросло до 15. Почему не 16? Так ведь одна из восьми линий прерываний была израсходована на каскадирование с другим контроллером!

Некоторое количество прерываний разошлось по системным устройствам, некоторое было выделено шине ISA – тогдашнему индустриальному стандарту. Генерация прерываний осуществлялась изменением уровня сигнала на линии соответствующего IRQ. Могут ли два или более устройств висеть на одном IRQ? Ну вообще-то могут, но если они одновременно сгенерируют сигнал прерывания, то до контроллера дойдет лишь один из них, а остальные будут потеряны, но ни контроллер, ни устройства об этом не догадаются. Такая ситуация получила название конфликта, и ее последствия всем хорошо известны. Впрочем, если прерывания возникают не слишком часто, то оба устройства вполне уживаются друг с другом (в свое время автор держал на одном прерывании и мышь, и модем).

Шина PCI, пришедшая на смену ISA, работает всего с четырьмя линиями равно приоритетных прерываний, условно обозначаемых как INTA, INTB, INTC и INTD. На каждый слот подведены все четыре прерывания, и устройство может использовать любое подмножество из них, хотя обычно ограничиваются только одним. Линии прерываний одного слота соединяются с линиями остальных слотов (а в некоторых дешевых платах все INTA, INTB, INTC и INTD вешаются на одну линию прерывания). Для равномерного распределения прерывания по устройствам на каждом слоту происходит ротация прерываний (рис. 5). Допустим, у нас есть два слота: в первом слоте прерывание INTA (со стороны устройства) соответствует прерыванию INTA (со стороны шины), прерывание INTB → INTB и т. д. Во втором слоте прерыванию INTA (с стороны устройства) соответствует прерывание INTB (со стороны шины), INTB → INTC, INTC → INTD и INTD → INTA, в результате чего устройства, исполь-

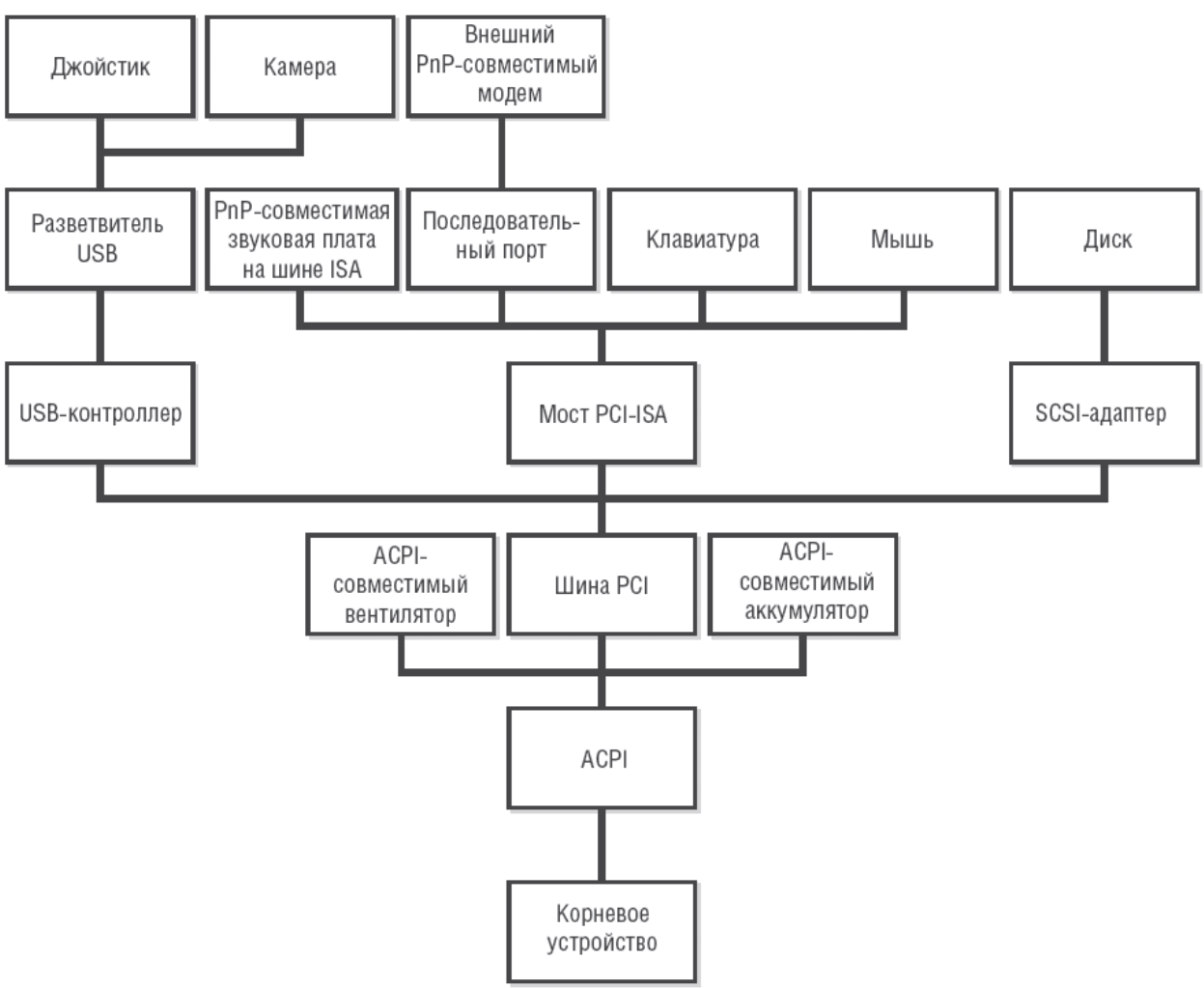

Рисунок 4. ACPI как корневой перечислитель

зующие прерывание INTA, оказываются развешаны по прерываниям INTA и INTB.

	PIRQA#	PIRQB#	PIRQC#	PIRQD#
PCI Slot 1	INTA#	INTB#	INTC#	INTD#
PCI Slot 2	INTB#	INTC#	INTD#	INTA#
PCI Slot 3	INTC#	INTD#	INTA#	INTB#
PCI Slot 4	INTD#	INTA#	INTB#	INTC#

Рисунок 5. Ротация аппаратных прерываний PCI-шины

Линии прерываний INTA – INTB соединяются с выводами PIRQ0 – PIRQ3 контроллера PCI-шины, а оттуда через роутер (PCI Interrupt Router) попадают в контроллер прерываний, тем или иным образом отображаясь на четыре линии IRQ, не занятые никакими ISA-устройствами. Поскольку количество установленных PCI-устройств обычно много больше четырех (мы считаем также и внутренние устройства, такие, например, как интегрированный контроллер USB, чаще всего повешенный на INTD), несколько устройств вынуждены делить одно прерывание между собой. В отличие от ISA, в PCI-шине совместное использование прерываний является ее нормальным состояниям. Генерация прерываний осуществляется не по переходу, а по состоянию, и устройство может удерживать линию прерывания в соответствующем состоянии до тех пор, пока его запрос не будет обработан. Теоретически это легко. Практически же… Даже поверхностное тестирование обнаруживает большое количество устройств и драйверов, не вполне соответствующих спецификациям и не желающих делить свое PIRQ с другими (или делающих это настолько неумело, что производительность падает в разы). Следование спецификациям предотвращает конфликты, но оставляет проблему падения производительности в силе. При совместном использовании прерываний драйвера получают сигналы не только от своих, но и от чужих устройств, заставляя их обращаться к своему устройству за подтверждением, и если выяснится, что прерывание сгенерировало не оно, запрос передается следующему драйверу в цепочке. А теперь представьте, что произойдет, если на одном прерывании висит десяток устройств и драйвер наиболее «беспокойного» из них попадет в самый хвост очереди?

Для достижения наивысшей производительности следует, во-первых, оптимально распределить PCI-карты по слотам (например, если у вас на шесть PCI-слотов приходятся две PCI-карты, то, втыкая устройства в первый и пятый слот, вы вешаете их на одно PIRQ), по возможности совмещая на одном PIRQ только наименее темпераментные устройства, т.е. такие, которые генерируют прерывания реже всего. Во-вторых, каждое PIRQ должно отображаться на свое IRQ. Какое – не суть важно (ведь приоритет PCI-прерываний одинаков), но только свое. Совместное использование одного IRQ несколькими PIRQ обычно не приводит к конфликтам, но негативно сказывается на производительности, ведь драйвера работают не с PIRQ, а с IRQ!

ACPI-ядра, работающие с PCI-шиной через ACPI-контроллер, лишены возможности управлять отображением PIRQ на IRQ по своему усмотрению. Не может управлять этим и BIOS (во всяком случае легальными средствами). Сам же ACPI стремится повесить все PIRQ на одно IRQ (обычно IRQ9), и помешать ему очень трудно. Если количество установленных PCI-устройств намного больше четырех, то разница в производительности между ACPI- и не ACPI-ядрами несущественна, поскольку, даже отказавшись от ACPI, вы все равно будете вынуждены разделять одно PIRQ между несколькими устройствами. Другое дело, если количество PCI-устройств невелико и наиболее темпераментные из них висят на своих прерываниях, тогда при переходе с ACPI- на не ACPI-ядро разница в быстродействии системы может оказаться очень значительной (то же самое относится и к неудачно спроектированным устройствам, не умеющим делить прерывания с другими и не имеющих достойной замены, например, дорогой видеоускоритель, RAID-контроллер и т. д.).

К сожалению, просто взять и отключить ACPI нельзя, поскольку он является не только менеджером питания, распределителем ресурсов, но еще и корневым перечислите-

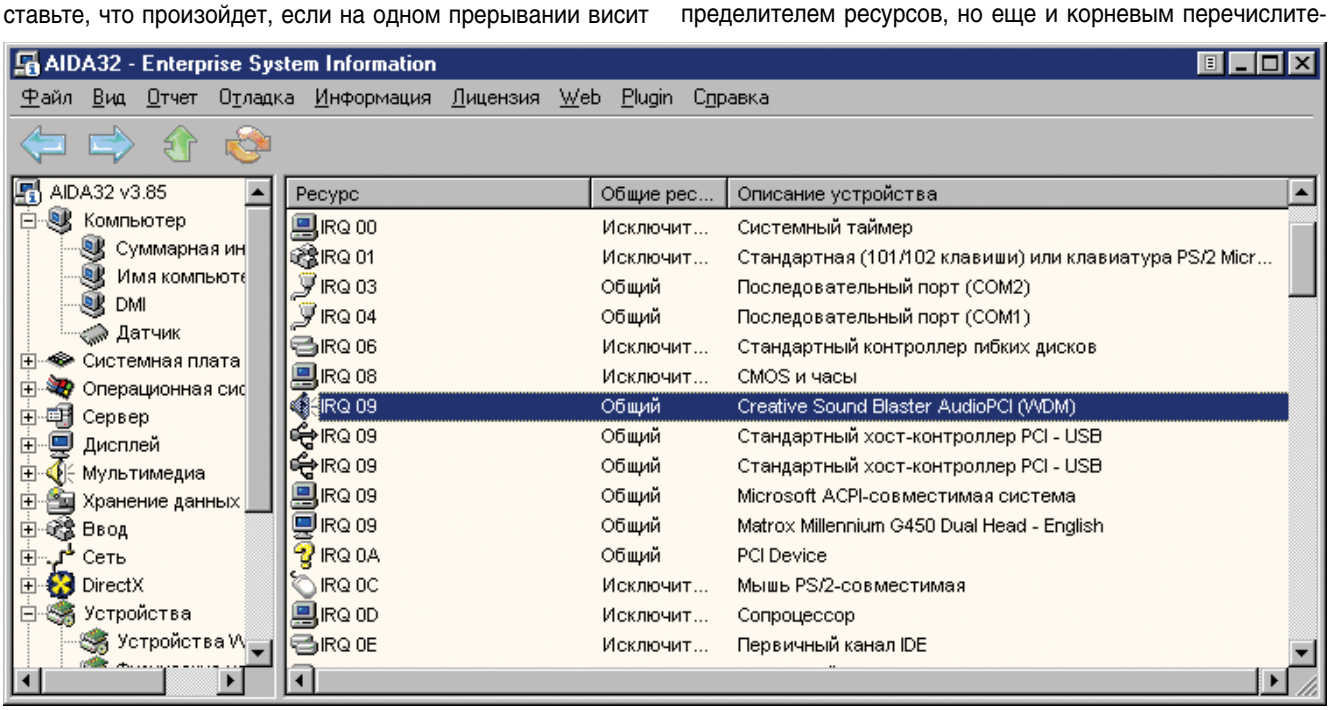

Рисунок 6. Все PCI-устройства на одном прерывании

лем. ACPI- и не ACPI-ядра используют различные деревья устройств и потому взаимно несовместимы. Смена ядра в обязательном порядке требует переустановки системы, в противном случае та откажется загружаться. Это существенно затрудняет сравнение быстродействия ACPI- и не ACPI-ядер, поскольку переустановка системы радикальным и непредсказуемым образом изменяет ее производительность (рис. 6).

Продвинутые материнские платы используют усовершенствованный контроллер прерываний (Advanced PIC или сокращенно APIC), поддерживающий 256 IRQ и способный работать в многопроцессорных системах. Однако в монопроцессорных конфигурациях он не обеспечивает никаких дополнительных преимуществ, т.к. количество свободных прерываний ограничивается не контроллером, а PCI-шиной. К тому же APIC-ядра не вполне корректно работают с таймером, что сводит на нет все их преимущества.

Переключение контекста

Под «многозадачностью» большинство пользователей подразумевает возможность параллельного выполнения нескольких приложений: чтобы в фоне играл WinAmp, скачивался mp3 с Интернета, принималась почта, редактировалась электронная таблица и т. д. Минимальной единицей исполнения в Windows является поток. Потоки объединяются в процессы, а процессы – в задания (jobs). Каждый поток обладает собственным стеком и набором регистров, но все потоки одного процесса выполняются в едином адресном пространстве и обладают идентичными квотами.

В любой момент времени на данном процессоре может выполняться только один поток, и если количество потоков превышает количество установленных процессоров, потоки вынуждены сражаться за процессорное время. Распределением процессорного времени между потоками занимается ядро. Вытесняющая многозадачность, реализованная в Windows NT, устроена приблизительно так: каждому потоку выдается определенная порция машинного времени, называемая квантом (quantum), по истечении которой планировщик (dispatcher) принудительно переключает процессор на другой поток. Учет процессорного времени обеспечивается за счет таймера. Периодически (раз в 10 мс или 15 мс) таймер генерирует аппаратное прерывание, приказывающее процессору временно приостановить выполнение текущего потока и передать бразды правления диспетчеру. Диспетчер уменьшает квант потока на некоторую величину (обычно равную двум) и либо возобновляет выполнение потока, либо (если квант обратился в нуль) сохраняет регистры потока в специальной области памяти, называемой контекстом (context), находит поток, больше всего нуждающийся в процессорном времени, восстанавливает его контекст вместе с контекстом процесса (если этот поток принадлежит другому процессу) и передает ему управление.

Потоки обрабатываются по очереди в соответствии с их приоритетом и принятой стратегией планирования. Планировщик сложным образом манипулирует с очередью, повышая приоритеты потоков, которые слишком долго ждут процессорного времени, только что получили фокус управления или дождались завершения операции ввода/вывода. Алгоритм планирования непрерывно совершенствуется, однако, не все усовершенствования оказывают благоприятное влияние на производительность. В общем случае многопоточные приложения должны исполняться на тех ядрах, под стратегию планирования которых они оптимизировались, в противном случае можно нарваться на неожиданное падение производительности.

При небольшом количестве потоков накладные расходы на их переключения довольно невелики, и ими можно пренебречь, но по мере насыщения системы они стремительно растут! На что же расходуется процессорное время? Прежде всего на служебные нужды самого планировщика (анализ очереди, ротацию приоритетов и т. д.), затем на сохранение/восстановление контекста потоков и процессов. Посмотрим, как все это устроено изнутри?

Дизассемблирование показывает, что планировщик как бы размазан по всему ядру. Код, прямо или косвенно связанный с планированием, рассредоточен по десяткам функций, большинство из которых не документированы и не экспортируются. Это существенно затрудняет сравнение различных ядер друг с другом, но не делает его невозможным. Чуть позже мы покажем, как можно выделить подпрограммы профилировщика из ядра, пока же сосредоточимся на переключении и сохранении/восстановлении контекста.

Процессоры семейства x86 поддерживают аппаратный механизм управления контекстами, автоматически сохраняя/восстанавливая все регистры при переключении на другую задачу, но Windows не использует его, предпочитая обрабатывать каждый из регистров вручную. Какое-то время автор думал, что i486C-ядро, ничего не знающее о MMX/SSE-регистрах современных процессоров и не включающее их в контекст, будет выигрывать в скорости, однако параллельная работа двух и более мультимедийных приложений окажется невозможной. В действительности же оказалось, что за сохранение/восстановление регистров сопроцессора (если его можно так назвать) отвечают машинные команды FXSAVE/FXSTOR, обрабатывающие и MMX/SSE-регистры тоже, но чтобы выяснить это, пришлось перерыть все ядро – от HAL до исполнительной системы!

Переключение контекста осуществляется служебной функцией SwapContext, реализованной в ntoskrnl.exe. Это чисто внутренняя функция, и ядро ее не экспортирует. Тем не менее она присутствует в символьных файлах (symbol file), бесплатно распространяемых фирмой Microsoft. Полный комплект занимает порядка 150 Мб и неподъемно тяжел для модемного скачивания. Ряд утилит, таких, например, как Symbol Retriever, от NuMega, позволяют выборочно скачивать необходимые символьные файлы вручную, значительно сокращая время перекачки, однако по непонятным причинам они то работают, то нет (Microsoft блокирует доступ?), поэтому необходимо уметь находить точку входа в SwapContext самостоятельно. Это легко. SwapContext – единственная, кто может приводить к синему экрану смерти с надгробной надписью «ATTEMPTED_SWICH_FROM_DPC», которой соответствует BugCheck код B8h. Загрузив ntoskrnl.exe в ИДУ (или любой другой дизассемблер), перечислим все перекрестные ссылки, ведущие к функциям KeBugCheck и KeBugCheckEx. В какой-то из них мы найдем PUSH B8h/CALL KeBugCheck или что-то в этом роде. Она-то

и будет функцией SwapContex. Прокручивая экран дизассемблера вверх, мы увидим вызов функции HalRequestSoftwareInterrupt, которая, собственно, и переключает контекст, а в многопроцессорной версии ядра еще и машинную команду FXSAVE, которая тут совсем ни к чему и которая отсутствует в монопроцессорной версии. К тому же многопроцессорные версии намного щепетильнее относятся к вопросам синхронизации и потому оказываются несколько менее производительными.

Функция HalRequestSoftwareInterrupt, реализованная в HAL, через короткий патрубок соединяется с функциями _HalpDispatchInterrupt/_HalpDispatchInterrupt, сохраняющими/восстанавливающими регистры в своих локальных переменных (не в контексте потока!) и на определенном этапе передающих управление на KiDispatchInterrupt, вновь возвращающую нас в ntoskrnl.exe и рекурсивно вызывающую SwapContext. Кто же тогда сохраняет/восстанавливает контексты? Оказывается – аппаратные обработчики. Список указателей на предустановленные обработчики находится в ntoskrnl.exe и содержится в переменной IDT (не путать с IDT-таблицей процессора!), которая, как и следовало ожидать, не экспортируется ядром, но присутствует в символьных файлах. При их отсутствии найти переменную IDT можно так: просматривая таблицу прерываний любых из ядерных отладчиков (Soft-Ice, Microsoft Kernel Debugger), определите адреса нескольких непереназначенных обработчиков прерываний (т.е. таких, которые указывают на ntoskrnl.exe, а не к драйверу) и, загрузив ntoskrnl.exe в дизассемблер, восстановите перекрестные ссылки, ведущие к ним. Это и будет структурой IDT.

Другие функции также могут сохранять/восстанавливать текущий контекст (это, в частности, делает Kei386EoiHelper, расположенная в ntoskrnl.exe), поэтому накладные расходы на переключение между потоками оказываются достаточно велики и выливаются в тысячи и тысячи команд машинного кода, причем каждое ядро имеет свои особенности реализации. Как оценить, насколько одно из них производительнее другого?

Логично, если мы уговорим ядро переключать контексты так быстро, как только это возможно, то количество переключений в единицу времени и определит долю накладных расходов в общем быстродействии ядра.

Сказано – сделано. Создаем большое количество потоков (по меньшей мере сто или даже триста) и каждый из них заставляем циклически вызывать функцию Sleep(0), приводящую к отдаче квантов времени и как следствие – немедленному переключению на другой поток. Количество переключений контекста можно определить по содержимому специального счетчика производительности, отображаемого Системным Монитором, утилитой CPUMon Марка Руссиновича, отладчиком Microsoft Kernel Debugger и многими другими программами.

Листинг 1. Измеритель скорости переключения контекста

```
thread()
{
    // отдаем процессорное время в бесконечном цикле
    while(1) Sleep(0);
}

#define defNthr   300
#define argNthr  ((argc > 1)?atol(argv[1]):defNthr)

main(int argc, char **argv)
{
    int a, zzz;

    printf("creating %d threads...", argNthr);

    // создаем argNthr потоков
    for (a = 0; a < argNthr; a++) ↵
          CreateThread(0, 0, (void*)thread, 0,0, &zzz);
    printf("OK\n"); thread();
    return 0;
}
```

Сравнение ACPI-ядра с i486C-ядром на машине автора (Intel Pentium-III 733 МГц, 256 Мб SDR-RAM-133) обнаруживает значительное расхождение в их производительности. i486C-ядро переключает контекст намного быстрее, особенно при работе с большим количеством потоков. В общем случае – количество переключений контекста обратно пропорционально количеству потоков, т.к. контексты надо где-то хранить, а кэш-память не резиновая. Если ядро делает много лишних сохранений (о которых мы уже говорили), оно существенно проигрывает в скорости. Тем не менее, все ядра спроектированы достаточно грамотно и сохраняют отличную подвижность даже при работе с тысячами потоков.

Переключение процессов требует дополнительных накладных расходов и потребляет намного больше памяти, попутно вызывая сброс буфера ассоциативной трансляции, поскольку каждый из процессов обладает своим адресным пространством. Выделить код, ответственный за переключение контекстов, несложно – он выдает себя обращением к регистру CR3, загружая в него указатель на каталог страниц (Page Directory Physical Address).

Давайте немного модернизируем нашу тестовую программу, заменив потоки процессами. Один из вариантов реализации может выглядеть так:

Листинг 2. Измеритель скорости переключения процессов

```
thread()
{
    while(1) Sleep(0);
}

#define defNthr   3
#define argNthr  ((argc > 1)?atol(argv[1]):defNthr)
#define argProc   "-666"

main(int argc, char **argv)
{
    int a, zzz;
    char buf[1000];
    STARTUPINFO st;
    PROCESS_INFORMATION  pi;

    memset(&st, 0, sizeof(st)); st.cb = sizeof(st);

    if ((argc > 1) && !strcmp(argv[1], argProc)) thread();

    sprintf(buf,"%s %s",argv[0], argProc);
    printf("creating %d proc...", argNthr);
    for (a = 0; a < argNthr; a++)
          CreateProcess(0, buf, 0,0,0, ↵
                NORMAL_PRIORITY_CLASS,0, 0, &st, &pi);
    printf("OK\n"); thread();
    return 0;
}
```

Даже при небольшом количестве процессов система значительно «проседает» под нагрузкой и начинает ощутимо притормаживать, а количество переключений контекстов сокращаются приблизительно вдвое. i486C-ядро по-прежне-

му держится впереди, что не может не радовать, к тому же с ним система намного более оживленно реагирует на внешние раздражители (в частности, клавиатурный ввод). Быстродействие подсистемы ввода/вывода специально не тестировалось, но по субъективным ощущениям i486C и с этим справляется намного быстрее.

Желающие подкрепить экспериментальные данные доброй порцией дизассемблерных листингов могут самостоятельно проанализировать функции планировщика, если, конечно, ухитрятся выдрать их из ядра! Далеко не всем исследователям недр Windows удалось это сделать…

Задумайтесь: если ядро львиную долю процессорного времени тратит на переключение контекстов, не означает ли это, что наиболее интенсивно вызываемыми функциями окажутся функции, принадлежащие планировщику? Запускаем нашу тестовую программу, подключаем Microsoft Kernel Profiler (или любой другой профилировщик ядра по вкусу) и дав ему на сбор статистики порядка десяти секунд, внимательно изучим полученный результат:

Листинг 3. Функции ядра, прямо или косвенно относящиеся к планированию

```
4484       ntoskrnl.exe  ExReleaseResourceForThread
4362       ntoskrnl.exe  KiDispatchInterrupt
4333       ntoskrnl.exe  SeTokenImpersonationLevel
2908       ntoskrnl.exe  KeDelayExecutionThread
2815       ntoskrnl.exe  KiIpiServiceRoutine
 300       ntoskrnl.exe  RtlPrefetchMemoryNonTemporal
  73       ntoskrnl.exe  KeDisconnectInterrupt
  41       ntoskrnl.exe  ExFreePoolWithTag
```

Длительность квантов

Частые переключения контекстов отрицательно сказываются на производительности системы, поэтому Windows NT подбирает продолжительность одного кванта с таким расчетом, чтобы они происходили как можно реже, теряя при этом подвижность и способность достаточно быстро реагировать.

Допустим, у нас имеется 100 потоков, каждому из которых выделяется 100 мс процессорного времени, причем все потоки используют отведенное им время полностью. Тогда между двумя переключениями одного и того же потока пройдет 10 сек! Вот тебе, бабушка, и многозадачный день… Разве это нормально, когда нажатая клавиша отображается на экране только через 10 сек? Когда сетевые клиенты получают в час по чайной ложке? А ведь если сервер обрабатывает каждого из клиентов в отдельном потоке (что является типичной стратегией программирования в Windows NT), он должен умело распределять процессорное время между тысячами потоков!

Разработчики UNIX, программирующие не ради денег, а в силу исторической неизбежности, стремятся выбирать величину кванта так, чтобы сервер не терял отзывчивости даже при пиковой нагрузке. Разработчики Windows NT, напротив, оптимизировали свою систему под максимальную производительность, меняя величину кванта от версии к версии так, чтобы совокупное количество обработанных запросов в единицу времени было максимальным. Ведь производительность – это сила, а комфортабельность и уют – понятия растяжимые. Поднимите компьютерные журналы, полазьте по Интернету – везде лежат только сравнительные тесты производительности, но нигде – отношение времени простоя клиента ко времени работы. Ладно, оставим лирику и перейдем к делу.

Windows NT поддерживает два типа квантов – длинные и короткие. Независимо от своего типа кванты могут быть как фиксированной, так и переменной длины (кванты переменной длины еще называют динамическими). Величина кванта выражается в условных единицах, официально называемых quantum units. Три квантовых единицы обычно равны одному тику таймера.

Управление типом и продолжительностью кванта осуществляется через следующий параметр реестра: HKLM\SYSTEM\CurrentControlSet\Control\PriorityControl\Win32PrioritySeparation.

Если 4-й и 5-й биты, считая от нуля, равны 10, система использует короткие кванты. То же самое происходит при оптимизации параметров быстродействия под исполнение приложений (Панель Управления → Система → Дополнительно → Параметры быстродействия). 01 – указывает на длинные кванты (они же используются при оптимизации системы под выполнение служб в фоновом режиме). Любое другое значение выбирает продолжительность кванта по умолчанию (короткие – в Windows 2000 Professional, длинные – в Windows 2000 Server).

Если 2-й и 3-й биты равны 10 – длина квантов фиксированна; 01 – позволяет планировщику динамически варьировать продолжительность кванта в заранее оговоренных пределах. Любое другое значение выбирает тип квантов по умолчанию (переменные – в Windows 2000 Professional, фиксированные – в Windows 2000 Server). При использовании динамических квантов планировщик пытается автоматически увеличивать продолжительность квантов некоторых потоков, тех, которым процессорное время нужнее всего. Во всяком случае, планировщик думает, что оно им нужнее, а думает он приблизительно так: если поток обслуживает GUI-окно и это окно находится в фокусе, продолжительность кванта увеличивается. Если поток полностью использует весь отведенный ему квант, его продолжительность увеличивается. Если… Конкретный алгоритм планирования зависит от выбранного ядра, и потому одни ядра могут оказаться намного предпочтительнее других.

Два младших бита задают индекс в двухэлементном массиве PsPrioritySeparation, расположенном внутри ntoskrnl.exe и используемым планировщиком для расчета продолжительности квантов активного процесса. Эта переменная не экспортируется ядром, но упоминается в символьных файлах. Если же они отсутствуют, обратитесь к функции PsSetProcessPriorityByClass, которая использует первый элемент этого массива как указатель на другой массив.

Таблица 1. Модельный ряд квантов в Windows 2000 Professional

	Короткие			Длинные		
Переменные	6	12	18	12	24	36
Фиксированные	18	18	18	36	36	36

Для экспериментов с квантами слегка модернизируем нашу тестовую утилиту, заставляя потоки (процессы) использовать отведенный им квант времени целиком. А в первичный поток встроим счетчик времени, вычисляющий продолжительность интервала между двумя соседними переключениями:

Листинг 4. Измеритель продолжительности квантов

```
thread()
{
    int a, b;
    while(!f) Sleep(0);
    while (f != 2);
    while(1)
    {
           for (a = 1; a< 100; a++) b = b + (b % a);
    }
}

#define defNthr   300
#define argNthr ((argc > 1)?atol(argv[1]):defNthr)

main(int argc, char **argv)
{
    int           a, zzz;
    SYSTEMTIME    st;

    printf("creating %d threads...", argNthr);
    for (a = 0; a < argNthr; a++)
           CreateThread(0, 0, (void*)thread, 0,0, &zzz);

    f = 1;  printf("OK\n");

    Sleep(0); f = 2;

    while(1)
    {
           GetSystemTime(&st);
           printf("* %02d:%02d:%02d\n",st.wHour, ↵
                  st.wMinute, st.wSecond);
           Sleep(0);
    }
    return 0;
}
```

Под Windows 2000 Professional уже при 100 потоках время прогона очереди составляет 10 сек, в то же время под Windows 2000 Server и того больше. Выглядит это, скажу я вам, очень удручающе, и работать в таких условиях становится крайне дискомфортно. Причем приобретение более быстродействующего процессора не ускоряет обработку очереди, ведь потоки по-прежнему используют отведенные им кванты целиком, и хотя успевают переработать намного больше данных, каждый из них, как и раньше, получает управление раз в десять секунд. Переход на двухпроцессорную машину повышает отзывчивость системы приблизительно в 1.3 раза (два процессора уменьшают длину очереди в два раза, но за счет перехода на APIC продолжительность одного тика таймера увеличивается с 10 мс до 15 мс, итого 2/15/10 ~ 1.3). Однако есть и другой, менее дорогостоящий и намного более радикальный способ решения проблемы. Один щелчок мыши увеличит реакционность системы в пять, а то и более раз. Не верите? Ну так щелкните по рабочему столу, чтобы окно нашего тестового приложения потеряло фокус. Взгляните на таблицу временных замеров, которую мне удалось получить.

Листинг 5. Время обработки очереди из 100 потоков при исполнении на переднем плане (слева) и в фоновом режиме (справа) на Windows 2000 Professional

```
00:14:48            00:23:10
00:15:02            00:23:12
00:15:16            00:23:14
00:15:30            00:23:16
00:15:46            00:23:18
00:15:59            00:23:20
00:16:11            00:23:22
00:16:27            00:23:24
00:16:41            00:23:27
00:16:55            00:23:29
```

Что произошло? Потоки ушли в фон, их приоритет понизился, а величина кванта сократилась до минимума. И хотя накладные расходы на переключение контекста возросли, время обработки очереди сократилось до 2 сек, с которыми вполне можно жить!

А теперь закройте все окна с несохраненными документами, которые вам жалко потерять, и увеличьте приоритет тестового приложения хотя бы на одну ступень. Висим? А то! Потоки тестового приложения отбирают процессорное время у всех остальных потоков (включая и некоторые системные), и они оказываются нефункциональны. То есть функциональны, но раз за 10 сек, чего для обработки клавиатурного и мышиного ввода более чем недостаточно. Забавно, но i486C-ядро при этом продолжает работать более или менее нормально.

Обсуждение полученных результатов

Результаты тестирования i486C-ядра, полученные на машине автора, приведены ниже.

Таблица 2. Скорость переключения контекста потоков на Windows 2000 Professional с отдачей квантов времени (потоки используют ничтожную часть отведенного им процессорного времени)

Количество потоков	Количество переключений контекстов за 10 секунд	
	ACPI-ядро	i486C-ядро
+50	7.701.161	8.002.734
+300	2.864.962	4.828.723

Таблица 3. Скорость переключения контекста процессов на Windows 2000 Professional с отдачей квантов времени (потоки используют ничтожную часть отведенного им процессорного времени)

Количество процессоров	Количество переключений контекстов за 10 секунд	
	ACPI-ядро	i486C-ядро
+50	2.923.719	9.638.651
+300	1.945.529	5.038.837

Таблица 4. Скорость переключения контекста потоков на Windows 2000 Professional без отдачи квантов времени (потоки используют отведенное им процессорное время целиком)

Количество потоков	Количество переключений контекстов за 10 секунд	
	ACPI-ядро	i486C-ядро
+50	3.033	6.481
+300	2.086	4.166

Таблица 5. Время обработки очереди на Windows 2000 Professional без отдачи квантов времени (потоки используют отведенное им процессорное время целиком)

Количество потоков	Время обработки очереди, сек	
	ACPI-ядро	i486C-ядро
+50	8	1
+300	15	2

Как видно, это ядро имеет множество преимуществ перед стандартным ACPI-ядром. Какое из них использовать – каждый должен решать сам. i486C-ядро не поддерживает ACPI и поэтому не способно в полной мере управлять энергопитанием компьютера, однако отрицать его сильные стороны, право же, не стоит. Оно действительно увеличивает производительность системы, и ничего мифического в этом разгоне нет. Не верите? Испытайте его сами. Для этого в процессе установки (переустановки) операционной системы дождитесь, когда на экране появится сообщение «Press F6 if you need to install a third party SCSI or RAID driver» (Нажмите F6, если вам необходимо загрузить SCSI- или RAID-драйвер стороннего производителя), нажмите F5 и выберите из списка имеющихся ядер «Standart PC with C-Step i486» (Стандартный компьютер i486 степпинг-C). После чего продолжите установку в обычном режиме.