многоядерные процессоры и проблемы ими порождаемые

крис касперски

многоядерные и Hyper-Threading процессоры не только увеличивают производительность, но и порождают многочисленные проблемы — некоторые приложения (драйвера) начинают работать нестабильно, выбрасывая критические ошибки или обрушивая систему в голубой экран смерти. в чем причина такого поведения и как его избежать?

введение

Многопроцессорные системы имеют свою специфику с которой программисты, работающие на IBM PC, долгое время оставались совершенно незнакомы. Поначалу это не создавало никаких проблем, поскольку большинство людей видели многопроцессорные системы только на картинках, и только единицы могли позволить себе иметь такую штучку на рабочем столе.

Теперь же все изменилось. Исчерпав резервы тактовой частоты, производители процессоров сначала предложили нам Hyper-Threading (два виртуальных процессора в одном), а затем и многоядерные процессоры (несколько полноценных процессоров на одном кристалле). И хотя до "эмуляции" настоящей многопроцессорной системы им еще далеко (многоядерные процессоры имеют одну шину, один контроллер прерываний и т. д.), дефекты программного обеспечения уже начинают проявляться.

Проблема на самом деле _очень_ серьезна и относится не только к программистам-самоучкам, клепающим мелкие утилиты, но и затрагивает весьма именитые корпорации, в том числе специализирующиеся и на мобильных устройствах. Вот что пишет AMD в руководстве по программированию под многоядерные процессоры: "...the primary issues that the mobile industry typically faced involved maximizing performance in a battery-operated environment, handling sleep states and non-standard display and I/O subsystems, and low-voltage considerations. Thus, many device drivers were tuned to maximize reliability and performance in those singleprocessor mobile environments: and many haven't even been tested in a multiprocessor system, even in the manufacturer's own test lab" ("*...основная проблема в том, что мобильная индустрия в основном сосредоточена на максимизации производительность в условиях питания от батарей, обработке "спящего" состояния, нестандартных дисплеев, подсистемы ввода/вывода и уменьшении питающего напряжения. Поэтому, большинство драйверов нацелено на максимальную надежность и производительность в однопроцессорном окружении. Многие из них не были протестированы на многопроцессорных машинах, даже в производственных тестовых лабораториях*" (http://developer.amd.com/assets/16_Interrupts.pdf).

Основной "удар" различий одно- и многопроцессорных машин операционная система и BIOS берут на себя (***примечание:*** *здесь и далее по тексту под термином "многопроцессорные машины" мы будем понимать как истинно многопроцессорные системы, так и компьютеры, построенные на базе многоядерных процессоров или процессоров с Hyper-Threading*). Прикладное приложение или драйвер устройства, спроектированный для однопроцессорной системы, не требует _никакой_ адоптации для переноса на многопроцессорную систему, если, конечно, он спроектирован правильно. Многие типы ошибок (и, в особенности, ошибки синхронизации) могут годами не проявляться в однопроцессорных конфигурациях, но заваливают многопроцессорную машину каждые десять минут, а то и чаще.

Исправление ошибок требует переделки исходных текстов (иногда очень значительной), но… что делать, если все, что у нас есть — это двоичный файл? Хорошо, если дефекты исправлены в новой версии (которая, между прочим, денег стоит), а если нет?

Эта статья адресована как самим разработчикам, так и продвинутым пользователям, умеющих держать hiew в руках и не шарахающихся в сторону от дизассемблера. Мы постараемся рассмотреть как можно больше способов решения проблем, а вопросы правомерности модификации двоичного кода пускай решают юристы и… моралисты.

прикладной уровень

Минимальной единицей исполнения в Windows является поток (thread), который в каждый момент времени может исполняться только на одном процессоре. Несмотря на то, что в большинстве случаев этот процессор не является жестко закрепленным и планировщик может запускать поток на любом свободном процессоре, поток остается неделимым (как атом), и различные части потока никогда не выполняются более, чем на одном процессоре одновременно. То есть, если в системе запущено только одно однопоточное приложение, на многопроцессорной машине оно будет выполняться с той же самой скоростью, что и на однопроцессорной (или даже чуть медленнее, за счет накладных расходов на поддержку многопроцессорности).

Рисунок 1 в симметричной многопроцессорной системе (которой является Windows NT и ее потомки), каждый поток может исполняться на любом процессоре

Процесс — более крупная исполнительная единица. Грубо говоря, это "коробок", в котором находятся принадлежащие ему потоки, исполняющиеся в едином адресном пространстве. Каждый поток обладает своим собственным стеком и набором регистров, но вынужден разделять глобальные переменные и динамическую память вместе с другими потоками процесса, что порождает проблему синхронизации. Допустим, один поток выводит ASCIIZ строку на экран, а другой — в это же самое время выполняет над этой строкой функцию `strcat()`, удаляющую символ нуля до завершения операции копирования. Как следствие — первый поток "вылетит" за пределы строки и пойдет чесать напаханную область памяти до тех пор пока не встретит посторонний нуль или не нарвется на исключение типа access violation.

Предотвратить такую ситуацию можно двояко. Либо переписать `strcat()` так, чтобы она сначала дублировала символ нуля, а только потом замещала его символом копируемой строки, либо воспользоваться один из средств синхронизации, например, критической секций, фактически представляющий собой флаг занятости. Поток, копирующий строку, взводит этот флаг перед вызовом `strcat()`, а поток, выводящий ее на экран, проверяет состояние флага и при необходимости ждет пока тот не освобождается и тут же взводит его вновь, чтобы во время вывода строки никто другой не вздумал ее модифицировать.

В первом случае, требуется всего лишь переделать `strcat()`, а во втором — скоординировать действие нескольких потоков, малейшая небрежность в синхронизации которых оборачивается либо неполной синхронизацией (например, поток, выводящий строку на экран не взводит перед этим флаг занятости), либо взаимоблокировкой (когда два или более потоков ждут освобождения друг друга, но никак не могут дождаться, поскольку один из них взвел флаг занятости и забыл его сбросить). К сожалению, при работе со сложными структурами данных без механизмов синхронизации обойтись уже не получается. Синхронизующий код как бы "размазывается" по всей программе и проверить его работоспособность становится очень трудно. Отсюда и ошибки.

С Linux/BSD в этом плане дела обстоят намного лучше. Основной единицей выполнения там является процесс (поддержка потоков уже появилась, но так и не сыскала большой популярности). Процессы исполняются в раздельных адресных пространствах и могут обмениваться данными только через явные средства межпроцессорного взаимодействия, значительно упрощая задачу синхронизации.

Теперь поговорим о том, почему на однопроцессорных машинах ошибки синхронизации проявляются значительно реже, чем на многопроцессорных. Дело в том, что при наличии только одного процессора, потоки выполняются последовательно, а не параллельно. Иллюзия одновременного выполнения создается лишь за счет того, что каждый поток работает в течении очень короткого (с человеческой точки зрения) промежутка времени, называемого квантом, а потом системный планировщик передает управление другому потоку. Длительность кванта варьируется в зависимости от множества обстоятельств (подробнее этот вопрос рассмотрен в статье "разгон и торможение Windows NT"), но как бы там ни было, квант — это целая вечность для процессора, за которую он очень многое успевает сделать.

Рассмотрим следующую (кстати, вполне типичную) ситуацию. Поток вызывает какую-нибудь функцию из стандартной библиотеки Си, а затем считывает глобальную переменную `errno`, в которую функция поместила код ошибки. В многопоточной программе, выполняющийся на однопроцессорной машине такая стратегия работает довольно уверенно, хотя и является порочной. Существует угроза, что поток будет прерван планировщиком после завершения Си-функции, но до обращения к переменной `errno`, и управление получит другой

поток, вызывающему "свою" Си-функцию, затирающую прежнее содержимое `errno`. И, когда первый поток вновь получит управление, он увидит там совсем не то, что ожидал! Однако, вероятность этого события на _однопроцессорной_ машине крайне мала. Тело потока состоит из тысяч машинных команд и переключение контекста может произойти где угодно. Чтобы попасть между вызовом Си-функции и обращением к `errno` это надо очень сильно "постараться".

А вот на многопроцессорной системе, где несколько потоков выполняются _параллельно_ вероятность одновременного вызова Си-функций значительно повышается и тщательно протестированная (на однопроцессорной машине), проверенная и отлаженная программа начинает регулярно падать без всяких видимых причин!

уровень драйверов

Драйвера обычно не создают своих собственных потоков, довольствуясь уже существующими, но проблем с синхронизацией у них даже больше, чем у приложений. Хуже всего то, что на многопроцессорной системе _одни_ и _те же_ части драйвера могут _одновременно_ выполняться на _различных_ процессорах! Чтобы понять причины такого беспредела, нам необходимо разобраться с базовыми понятиями ядра: IRQL и ISR.

Планировка драйверов осуществляется совсем не так, как потоков прикладного режима. Если прикладной поток может быть прерван в любое время безо всякого вреда, прервать работу драйвера можно только с его явного разрешения, иначе нормальное функционирование системы станет невозможным. Драйвера, обрабатывающие асинхронные события, должны быть либо полностью реентерабельными (т. е. корректно "подхватывать" новое событие во время обработки предыдущего), либо каким-то образом задерживать поступление новых событий, пока они не разберутся с текущим. Первый механизм гораздо более сложен в реализации. Программисты, писавшие резидентов под MS-DOS, должно быть помнят как часто им приходилось пользоваться командой CLI, запрещающий прерывания на время перестройки критических структур данных. Допустим, наш русификатор устанавливает новый обработчик клавиатурного прерывания. Он записал в таблицу векторов свое смещение и только собирался записать сегмент, как пользователь вдруг нажал на клавишу и процессор передал управление по адресу со старым сегментом и новым смещением.

Программируемый контроллер прерываний (Programmable Interrupt Controller, или, сокращенно PIC) оригинального IBM PC был построен на микросхеме i8259A, сейчас же контроллер прерываний встроен непосредственно в южный мост чипсета и эмулирует i8259A лишь в целях обратной совместимости. PIC имеет 15 линий прерываний, а каждая линия — свой приоритет. Во время обработки прерываний, прерывания с равным или более низким приоритетом маскируются, так сказать, откладываясь на потом. Иногда это помогает, иногда нет. Например, если замаскировать прерывания от таймера более чем на один "тик", системные часы начнут отставать. А если проигнорировать прерывания от звуковой карты и вовремя не "скормить" ей очередную порцию данных, она начнет "булькать", заставляя пользователя рыдать от счастья и биться головой о монитор. Прерывания с более высоким приоритетом прерывают менее приоритетные прерывания, возвращая им управление после того, как они будут обработаны. Усовершенствованные клоны PIC'a (Advanced Programmable Interrupt Controller или, сокращенно, APIC) обеспечивают 256 линий прерываний и, в отличии от обычного PIC'a, способны работать в многопроцессорных системах.

Рисунок 2 архитектура контроллера прерываний на двухпроцессорной машине

Операционная система Windows поддерживает PIC и APIC контроллеры, но использует свою собственную систему приоритетов прерываний, известную под аббревиатурой IRQL, которая расшифровывается как Interrupt Request Levels (Уровни Запроса Прерываний). Всего существует 32 уровня, пронумерованных целыми числами от 0 до 31. Уровень 0 имеет минимальный приоритет, 31 — максимальный. Нормальное выполнение потока происходит на нулевом уровне, называемого пассивным (PASSIVE) и его может прерывать любое асинхронное событие, возникающее в системе. При этом операционная система повышает текущий IRQL до уровня возникшего прерывания и передает управление его ISR (Interrupt Service Routine – процедура обработки прерывания), предварительно сохранив состояние текущего обработчика.

Приоритеты с номерами 1 и 2 отданы под программные прерывания (например, возникающие при ошибке обращения к странице памяти, вытесненной на диск), а все

остальные — обслуживают аппаратные прерывания от периферийных устройств, причем, прерывания от таймера имеет приоритет 28.

Рисунок 3 уровни запросов прерываний и их назначение

Чтобы замаскировать прерывания на время выполнения ISR многие программисты просто повышают уровень IRQL ядерной API-функций `KeRaiseIrql()`, а при выходе из ISR восстанавливают его вызовом `KeLowerIrql()`. Даже если они не делают этого явно, за них это делает система. Рассмотрим происходящие события более подробно.

Допустим, поток A работает на уровне IRQL равном PASSIVE_LEVEL (см. рис.4). Устройство Device 1 возбуждает аппаратное прерывание с уровнем DIRQL (т. е. с номером 3 до 31 включительно). Операционная система прерывает выполнение Потока A, повышает IRQL до DIRQL и передает управление на ISR устройства Device 1. Обработчик прерывания обращается к устройству Device 1, делает с ним все, что оно требует, ставит в очередь отложенную процедуру `DpcForISR()` для дальнейшей обработки и понижает IRQL до прежнего уровня. Отложенные процедуры (Deferred Procedure Calls или, сокращено, DPCs) выполняются на IRQL равном 2 (DISPATCH_LEVEL) и потому не могут начать свою работу вплоть до выхода из ISR.

Если во время выполнения ISR возникнет прерывания, то оно будет замаскировано. Если прерывание возникнет во время выполнения `DpcForISR()`, операционная система прервет ее работу, передаст управление ISR, который поставит в очередь еще одну отложенную процедуру, и вновь возвратится в `DpcForISR()`. Таким образом, сколько бы прерываний ни возникало, отложенные процедуры обрабатываются последовательно, в порядке очереди.

Рисунок 4 обработка аппаратных прерываний на машине с одним процессором

На однопроцессорных системах такая схема работает вполне нормально, но вот на многопроцессорных… каждый процессор имеет свой IRQL, независимый от остальных. Повышение IRQL на одном процессоре никак не затрагивает все остальные и генерация прерываний продолжается (см. рис. 5).

Рисунок 5 маскировка прерываний драйвером на двухпроцессорной машине

Допустим, поток A выполняется на процессоре 1 с IRQL=PASSIVE_LEVEL, в то время как поток B выполняется на процессоре 1 с тем же самым IRQL (см. рис. 6). Устройство Device 1 посылал процессору 0 сигнал прерывания. Операционная система "ловит" его, повышает IRQL процессора 0 до значения DIRQL и передает управление ISR устройства Device 1, которое делает с устройством что положено и ставит в очередь отложенную процедуру `DpcForIsr()` для дальнейшей обработки. По умолчанию, функция добавляется в очередь того процессора, на котором запущена ISR (в данном случае процессора 0).

Устройство Device 1 вновь генерирует сигнал прерывания, который на этот раз посылается процессору 1, поскольку процессор 0 еще не успел завершить обработку ISR и не понизил IRQL. Система повышает IRQL процессора 1 до DIRQL и передает управление IRQ устройства Device 1, который делает с устройством все что нужно и ставит отложенную процедуру `DpcForIsr()` в очередь на процессоре 1.

Рисунок 6 обработка аппаратных прерываний драйвером на двухпроцессорной машине

Затем ISR на обоих процессорах завершаются, система понижает IRQL и начинается выполнение отложенной процедуры `DpcForIsr()`, стоящей как в очереди процессора 0, так и в очереди процессора 1 Да! Вы не ошиблись! Процедура `DpcForIsr()` будет исполняться сразу на обоих процессорах одновременно, отвечая за обработку двух прерываний от одного устройства! Как вам это нравится?! В такой ситуации очень легко превратить совместно используемые данные в мешанину, возвратив неожиданный результат или завесив систему (см. рис 7).

Рисунок 7 отсутствие синхронизации при обработке прерываний на двухпроцессорной машине приводит к порче разделяемых данных

Чтобы упорядочить выполнение отложенных процедур, необходимо использовать спинлуки (spin-lock) или другие средства синхронизации, работающие по принципу флагов занятости (см. рис. 8).

Рисунок 8 защита разделяемых данных спин-блокировками

Другим источником ошибок являются модификация кода ядра системы или загружаемых драйверов. Многие программы, такие как брандмауэры, антивирусы, защиты или вирусы перехватывают некоторые функции для управления трафиком, автоматической проверки открываемых файлов и т. д. Модификация потенциально опасна даже на однопроцессорных машинах, а о многопроцессорных и говорить не стоит! Это отвратительный прием программирования, которого настоятельно рекомендуется избегать, но… он есть! И это факт!

Большинство программистов просто внедряют в начало функции jump на свой перехватчик (предварительно скопировав оригинальные байты в свой же собственный буфер). При завершении работы обработчик выполняет сохраненные инструкции, после чего передает управление на первую машинную инструкцию перехваченной функции, следующую за jump'ом. Поскольку, на x86 процессорах длина команд непостоянна, перехватчику приходится тащить за собой целый дизассемблер (называемый дизассемблером длин). Однако, это не самое страшное.

Во-первых, посторонний отладчик мог внедрить в начало (или середину функции) программную точку останова, представляющую собой однобайтовую команду с опкодом CCh, сохранив оригинальный байт где-то в памяти. В этом случае, вставлять jump поверх CCh ни в коем случае нельзя, поскольку отладчик может заметить, что точка останова исчезла и поставить CCh еще раз, забыв обновить оригинальное содержимое, оставшееся от старой команды. Корректный перехват в этом случае практически невозможен. Теоретически, можно внедрить jump во вторую инструкцию, но для этого нам необходимо определить где заканчивается первая, а поскольку ее начало искажено программной точкой останова, для ее декодирования придется прибегнуть к эвристическим методам, а они ненадежны. К счастью, большинство функций начинаются со стандартного пролога `PUSH EBP/MOV EBP,ESP` (55h/8Bh ECh), поэтому, встретив последовательность CCh/8Bh ECh мы вполне уверенно можем внедрять свой jump, начиная с `MOV EBP,ESP`.

Вот только тут есть один нюанс. Команда ближнего перехода в 32-битном режиме занимает целых 5 байт, поэтому, для ее записи необходимо воспользоваться командой `MOVQ`, иначе модификация будет представлять неатомарную операцию. Задумайтесь, что произойдет, если мы записали 4 первых байта команды `JMP NEAR TARGET` командой `MOV` и только собрались дописать последний байт, как внезапно пробудившийся поток захотел вызвать эту функцию? Правильно — произойдет крах!

Но даже атомарность не спасает от всех проблем. Допустим, мы записываем 5ти байтовую команду `JMP NEAR TARGET` поверх 2х байтовой команды `MOV EBP,ESP`, естественно, затрагивая следующую за ней команду. Даже на однопроцессорных машинах существует вероятность, что какой-то из потоков был ранее прерван сразу же после выполнения `MOV EBP,ESP` и когда он возобновит свое выполнение, то… окажется _посередине_ команды `JMP NEAR TARGET`, что повлечет за собой непредсказуемое поведение системы.

Алгоритм безопасной модификации выглядит так: перехватываем `INT 03h`, запоминая адрес прежнего обработчика, внедряем в начало перехватываемой функции CCh (если только программная точка уже не установлена). При возникновении прерывания `INT 03h` мы сравниваем полученный адрес со списком адресов перехваченных функций и, если это, действительно, "наш" адрес, выполняем ранее сохраненную машинную инструкцию в своем буфере и передаем управление на вторую инструкцию перехваченной функции. Снимать CCh ни в коем случае нельзя! Поскольку в этот момент функцию может вызывать кто-то еще, но наш перехватчик "прозевает" этот факт!

Если же полученный адрес не "наш", мы передаем управление предыдущему обработчику `INT 03h`. Тоже самое мы делаем, если программная точка останова была установлена еще до перехвата. Тогда мы позволяем предыдущему обработчику `INT 03h` восстановить ее содержимое, а сами ставим CCh на следующую инструкцию. Конечно, такой способ перехвата _намного_ сложнее "общепринятого", зато он на 100% надежен и работает в любых конфигурациях — как одно- так и многопроцессорных.

пути решения проблем

Самое простое (и самое радикальное) решение — указать ключ /NUMPROC=1 (или /ONECPU) в файле boot.ini, одним росчерком пера превратив многопроцессорную систему в однопроцессорную. Правда, о производительности после этого можно забыть, поэтому прибегать к такому "варварскому" методу стоит только в самых крайних случаях, когда система регулярно сбоит, а времени на поиски неисправности и капитальный ремонт у нас нет.

Кстати, поиск неисправностей – самое сложное дело. Некорректная синхронизация потоков приводит к порче данных и критические ошибки возникают (если они вообще возникают, хуже когда программа делает из обрабатываемых данных "винегрет") довольно далеко от места "аварии". То же самое относится и к голубым экранам смерти. Изучение дампов памяти дает довольно скудную информацию, особенно если разрушены структуры данных, хранящиеся в динамической памяти, которая каждый раз выделяется по разным адресам, что затрудняет воспроизведение ошибки.

При наличии исходных текстов в первую очередь проверьте: не используется ли во многопоточной программе однопоточные версии библиотек? В частности, компилятор Microsoft Visual C++ поставляется с двумя версиями статических Си-библиотек: LIBC.LIB – для однопоточных и LIBCMT.LIB – для многопоточных программ. Динамически компонуемая библиотека MSVCRT.LIB используется как в одно- так и во многопоточных проектах. Так же поищите прямые вызовы `CreateThread()`. Со стандартной Си-библиотекой они _не_ совместимы и потому должны быть в обязательном порядке заменены на `_beginthread()` или `_beginthreadex()`.

Все глобальные переменные (кроме тех, что используются для обмена данных между потоками) поместите в TLS (Thread Local Storage – Локальная Память Потока). На уровне исходных текстов это делается так: "`__declspec (thread) int my_var;`", при этом компилятор создает в PE-файле специальную секцию .tls, куда и помещает `my_var`, автоматически создавая отдельный экземпляр для каждого из потоков.

В отсутствии исходных текстов эту затею осуществить труднее, но все-таки возможно. Сначала необходимо найти переменные, к которым идет обращение из нескольких потоков. Это делается так: ищутся все вызовы `CreateThread()/_beginthread()`, определяется стартовый адрес функции потока и создается дерево функций, вызываемых этим потоков (для этого удобно использовать скрипт func_tree.idc от mammon'a, который можно скачать с www.idapro.com). Перечисляем глобальные переменные, упомянутые в этих функциях и если одна и та же переменная встречается в деревьях двух разных потоков — смотрим на нее пристальным взглядом, пытаясь ответить на вопрос: может ли она быть источником проблем или нет? Если переменная не используется для обмена данными между потоками — замещаем все обращения к ней на переходник к нашему обработчику, размещенному в свободном месте файла, который, используя вызовы `TslSetValue()/TslGetValue()`, записывает/считывает ее содержимое. Если же переменная используется для обмена данными между потоками — окружаем ее критическими секциями или другими механизмами синхронизации.

Естественно, все это требует правки исполняемого файла (и при том довольно значительной). Без соответствующих знаний и навыков за такую задачу не стоит и браться! Правда, есть шанс, что проблему удастся разрешить и без правки — поменяв приоритеты потоков. Если один из двух (или более) потоков, использующий разделяемые данные без синхронизации, получит больший приоритет, чем остальные, "расстановка сил" немедленно изменится и, возможно, она изменится так, что порча данных станет происходить не так часто, как прежде. Нужные значения приоритетов подбираются экспериментально, а задаются API-функций `SetThreadPriority()`, принимающий дескриптор потока. Вот тут-то и начинаются проблемы. Мы можем легко узнать идентификатор потока через функции TOOLHELP32: `CreateToolhelp32Snapshot()`,`Thread32First()/Thread32Next()`, остается "всего лишь" преобразовать его в дескриптор. Долгое время это приходилось делать весьма извращенным путем через недокументированные функции типа `NtOpenThread` (см. http://hi-tech.nsys.by/11/), но в Windows 2000 наконец-то появилась легальная API-функция `OpenThread()`, принимающая идентификатор потока и возвращающая его дескриптор (разумеется, при условии, что все необходимые права у нас есть). Виват, Microsoft!

Разобравшись с прикладными приложениями, перейдем к драйверам. При наличии исходных текстов достаточно использовать спин-блокировку во всех отложенных процедурах, однако, в большинстве случаев исходных текстов у нас нет, а править драйвер в hiew'e удовольствие не из приятных. К счастью, существует и другой путь — достаточно исправить таблицу диспетчеризации прерываний (IDT – Interrupt Dispatch Table), разрешив каждому

процессору обрабатывать прерывания только от "своих" устройств. Это практически не снижает производительности (особенно если быстрые и медленные устройства между процессорами распределяются по честному, то есть равномерно) и ликвидирует ошибки синхронизации вместе с голубыми экранами смерти.

заключение

Многопроцессорные системы создают гораздо больше проблем, чем мы здесь описали и далеко не все из них разрешимы в рамках простой переделки программ. Получив возможность создавать потоки, программисты далеко не сразу осознали, что отлаживать многопоточные программы на порядок сложнее, чем однопоточные. С другой стороны, уже сейчас мы приходим к распределенным системам и распределенному программированию. Разбив цикл с большим количеством итераций на два цикла, исполняющихся в разных потоках/процессах, на двухпроцессорной машине мы удвоим производительностью! Это слишком значительный выигрыш, чтобы позволить себе пренебрегать им, поэтому осваивать азы распределенного программирования нужно уже сейчас.

>>> врезка ссылки по теме

- **Scheduling, Thread Context, and IRQL**:
  - статья, сжато, но доходчиво, описывающая механизмы диспетчеризации потоков, IRQL-уровни и обработку прерываний на однопроцессорных и многопроцессорных машинах под Windows NT (на английском языке): http://www.microsoft.com/whdc/driver/kernel/IRQL.mspx;
- **Locks, Deadlocks, and Synchronization:**
  - статья, описывающая механизмы синхронизации Windows NT и проблемы, возникающие при их некорректном использовании (на английском языке): http://www.microsoft.com/whdc/driver/kernel/locks.mspx;
- **Principles of Concurrent and Distributed Programming**:
  - книга, посвященная основам распределенного программирования (на английском языке): http://www.amazon.com/gp/product/013711821X/002-0912814-0689662?v=glance&n=283155;