Стеки ядра

Хотя размер пользовательского стека обычно составляет 1 Мбайт, объем памяти, предназначенной стеку ядра, значительно меньше: 12 Кбайт для платформы x86 и 16 Кбайт для платформы x64, далее следует сторожевая страница (что в целом составляет 16 или 20 Кбайт виртуального адресного пространства). Считается, что код, запускаемый в режиме ядра, имеет меньше рекурсий, чем пользовательский код, а также более эффективно задействует переменные и сохраняет размеры буфера стека на низком уровне. Поскольку стеки ядра находятся в системном адресном пространстве (которое совместно используется всеми процессами), расходование ими памяти имеет более существенное влияние на систему.

Хотя код ядра обычно не рекурсивен, взаимодействия между графическими системными вызовами обрабатываются драйвером Win32k.sys, и его последующие обратные вызовы кода, выполняемого в пользовательском режиме, могут стать причиной рекурсивных повторных вхождений в ядро в том же самом стеке ядра. По этой причине Windows предоставляет механизм динамического расширения и сокращения стека ядра от его начального размера. Поскольку каждый дополнительный вызов графики выполняется из одного и того же потока, выделяется еще один стек ядра размером 16 Кбайт (где-нибудь в системном адресном пространстве; диспетчер памяти позволяет переходить между стеками при приближении к сторожевой странице). Как показано на рис. 5.28, когда каждый вызов возвращает управление вызывавшему коду (раскрутка стека вызовов), диспетчер памяти освобождает дополнительно выделенный стек ядра. Этот механизм позволяет надежно поддерживать рекурсивные системные вызовы и эффективно использовать системное адресное пространство, а также, если необходимо, его могут применять разработчики драйверов при выполнении рекурсивных внешних вызовов с помощью API-функции `KeExpandKernelStackAndCallout(Ex)`.

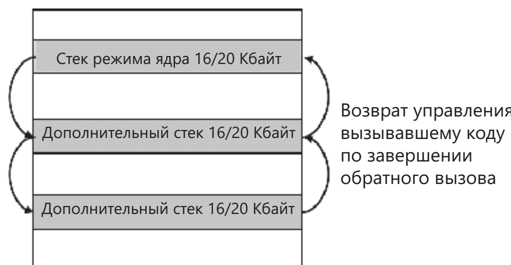

Рис. 5.28. Переходы между стеками ядра

ЭКСПЕРИМЕНТ: ПРОСМОТР ПОКАЗАТЕЛЯ ИСПОЛЬЗОВАНИЯ СТЕКА ЯДРА

Для вывода на экран сведений о текущей занятости физической памяти стеками ядра можно воспользоваться программой RamMap из пакета Sysinternals. Следующий снимок экрана сделан на вкладке Use Counts:

Driver Locked	3,419,852 K	3,419,852 K	
Kernel Stack	47,496 K	41,380 K	6,116 K
Unused	10,773,740 K	77,504 K	

Чтобы просмотреть данные об использовании стека режима ядра, выполните следующие действия.

1. Повторите предыдущий эксперимент с TestLimit, но пока не завершайте TestLimit.
2. Переключитесь на RamMap.
3. Откройте меню File и выберите команду Refresh (или нажмите F5). Размер стека ядра должен увеличиться:

Driver Locked	3,419,852 K	3,419,852 K	
Kernel Stack	346,076 K	339,916 K	6,160 K
Unused	10,318,232 K	77,680 K	

Запуск утилиты TestLimit еще несколько раз (без закрытия предыдущих экземпляров) приведет на 32-разрядной системе к быстрому истощению физической памяти. Это ограничение ведет к одному из основных общесистемных ограничений на количество 32-разрядных потоков.

DPC-стек

Для каждого процесса Windows поддерживает DPC-стек, доступный для использования системой при выполнении отложенных вызовов процедур (DPC). При таком подходе DPC-код изолируется от стека ядра текущего программного потока (не имеющего отношения к фактической операции DPC-вызова, поскольку DPC-вызовы выполняются в контексте произвольного потока; см. главу 6). DPC-стек настраивается в качестве исходного стека для обслуживания в ходе системного вызова инструкции `sysenter`(x86), `svc` (ARM) или `syscall`(x64). За переключение стека при выполнении этих инструкций отвечает центральный процессор, который действует на основании состояния одного из зависящих от модели регистров (Model-Specific Register, MSR на платформах x86/x64). Тем не менее Windows не в состоянии перепрограммировать MSR для каждого контекстного переключения, поскольку это очень затратная операция. Поэтому Windows задает в MSR указатель на DPC-стек каждого процессора.

Дескрипторы виртуальных адресов

Чтобы узнать, когда загружать страницы в память, диспетчер памяти использует алгоритм *подкачки страниц по требованию* (demand-paging algorithm), ожидая перед извлечением страницы с диска ссылки потока на адрес и получение им ошибки отсутствия страницы. По аналогии с копированием при записи, подкачка страниц по требованию является одной из форм *отложенного вычисления* (lazy

evaluation), при котором решение задачи откладывается до момента, когда в нем возникнет прямая необходимость.

Диспетчер памяти выполняет отложенное вычисление не только при внесении страниц в память, но и при создании таблиц страниц, необходимых для описания новых страниц. Например, когда поток подтверждает большую область виртуальной памяти с помощью функции `VirtualAlloc` или `VirtualAllocExNuma`, диспетчер памяти может тут же создать таблицы страниц, отвечающие за доступ ко всему диапазону выделенной памяти. А что, если к части диапазона вообще не будет обращений? Создание таблиц страниц для всего диапазона обернется напрасной тратой ресурсов. Вместо этого диспетчер памяти откладывает создание таблицы страниц до тех пор, пока поток не получит ошибку страницы, и только потом создает таблицу страниц для этой страницы. Такой метод существенно повышает производительность процессов, выполняющих резервирование и/или подтверждение больших объемов памяти, к которой редко обращаются.

Виртуальное адресное пространство, которое будет занято такими еще не существующими таблицами страниц, входит в квоту процесса на страничный файл и в системный показатель подтверждения. Тем самым гарантируется, что пространство будет доступно для этих таблиц при их фактическом создании. Благодаря алгоритму отложенного вычисления выделение даже больших блоков памяти является довольно быстрой операцией. Когда поток выделяет память, диспетчер памяти должен нести ответственность за диапазон адресов, используемых потоком. Для этого диспетчер памяти содержит еще один набор структур данных, призванных отслеживать, какие виртуальные адреса зарезервированы в адресном пространстве процесса, а какие нет. Эти структуры данных известны как *дескрипторы виртуальных адресов* (VAD, Virtual Address Descriptors). Память для VAD-дескрипторов выделяется в невыгружаемом пуле.

Дескрипторы виртуальных адресов процесса

Для каждого процесса диспетчер памяти поддерживает набор VAD-дескрипторов, описывающий состояние адресного пространства процесса. VAD-дескрипторы сведены в самобалансирующееся АВЛ-дерево (названное по первым буквам фамилий его создателей — Адельсона-Вельского и Ландиса), которое обладает оптимальной сбалансированностью. Это приводит к меньшему среднему количеству сравнений при поиске VAD-дескриптора, соответствующего виртуальному адресу. Один дескриптор виртуального адреса выделяется каждому виртуально непрерывному диапазону несвободных виртуальных адресов, имеющих одинаковые характеристики (при этом зарезервированные адреса отличаются и от подтвержденных, и от отображаемых, учитывается также защита памяти и т. д.). Схематически VAD-дерево показано на рис. 5.29.

Когда процесс резервирует адресное пространство или отображает представление раздела, диспетчер памяти создает VAD-дескриптор для хранения любой информа-

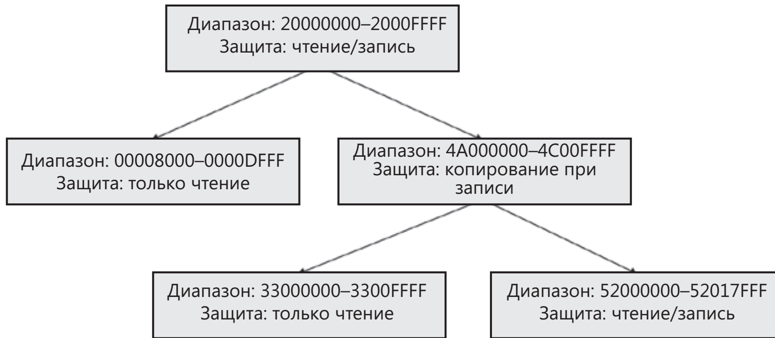

Рис. 5.29. Дескрипторы виртуальных адресов

ции, предоставленной запросом на выделение памяти: диапазона зарезервированных адресов, признаков того, является ли этот диапазон совместно используемым или закрытым, может ли дочерний процесс унаследовать содержимое диапазона, применима ли к страницам диапазона защита страниц.

При первом обращении потока к адресу диспетчер памяти должен создать PTE-запись для страницы, содержащейся по этому адресу. Для этого он находит VAD-дескриптор, в адресный диапазон которого входит запрашиваемый адрес, и использует найденную информацию для заполнения PTE-записи. Если адрес не входит в диапазон, перекрываемый VAD-дескриптором, или в диапазон зарезервированных, но не подтвержденных адресов, диспетчер памяти узнает о том, что поток не выделил память перед попыткой ее использования, и поэтому генерирует нарушение прав доступа.

ЭКСПЕРИМЕНТ: ПРОСМОТР ДЕСКРИПТОРОВ ВИРТУАЛЬНЫХ АДРЕСОВ

Для просмотра VAD-дескрипторов заданного процесса можно воспользоваться командой `!vad` отладчика ядра. Но сначала с помощью команды `!process` нужно найти адрес корневого элемента VAD-дерева, а затем указать этот адрес в команде `!vad`, как в следующем примере просмотра VAD-дерева для процесса, в котором выполняется Explorer.exe:

```
lkd> !process 0 1 explorer.exe
PROCESS ffffc8069382e080
    SessionId: 1  Cid: 43e0    Peb: 00bc5000  ParentCid: 0338
    DirBase: 554ab7000  ObjectTable: ffffda8f62811d80  HandleCount: 823.
    Image: explorer.exe
    VadRoot ffffc806912337f0 Vads 505 Clone 0 Private 5088. Modified 2146.
            Locked 0.
...
lkd> !vad ffffc8068ae1e470
VAD             Level     Start       End Commit
ffffc80689bc52b0  9       640       64f      0 Mapped       READWRITE
Pagefile section, shared commit 0x10
ffffc80689be6900  8       650       651      0 Mapped       READONLY
Pagefile section, shared commit 0x2
```

```
ffffc80689bc4290  9       660       675       0 Mapped       READONLY
Pagefile section, shared commit 0x16
ffffc8068ae1f320  7       680       6ff      32 Private      READWRITE
ffffc80689b290b0  9       700       701       2 Private      READWRITE
ffffc80688da04f0  8       710       711       2 Private      READWRITE
ffffc80682795760  6       720       723       0 Mapped       READONLY
Pagefile section, shared commit 0x4
ffffc80688d85670 10       730       731       0 Mapped       READONLY
Pagefile section, shared commit 0x2
ffffc80689bdd9e0  9       740       741       2 Private      READWRITE
ffffc80688da57b0  8       750       755       0 Mapped       READONLY
\Windows\en-US\explorer.exe.mui
...
Total VADs: 574, average level: 8, maximum depth: 10
Total private commit: 0x3420 pages (53376 KB)
Total shared commit:  0x478 pages (4576 KB)
```

Чередование дескрипторов виртуальных адресов

Как правило, драйверу видеокарты нужно копировать данные из графического приложения пользовательского режима в другую системную память разного происхождения, включая память видеокарты и память AGP-порта, причем обе имеют разные атрибуты кэширования и разные адреса. Чтобы дать этим разным представлениям памяти возможность быстро отображаться на процесс и поддерживать разные атрибуты кэширования, диспетчер памяти реализует чередующиеся дескрипторы виртуальных адресов (rotate virtual address descriptors), позволяющие видеодрайверам осуществлять прямую передачу данных с использованием графического процессора (Graphical Processing Unit, GPU) и по мере надобности менять для страниц представления процесса ненужную память на нужную. Пример того, как один и тот же виртуальный адрес может поочередно переходить между видеопамятью и виртуальной памятью, показан на рис. 5.30.

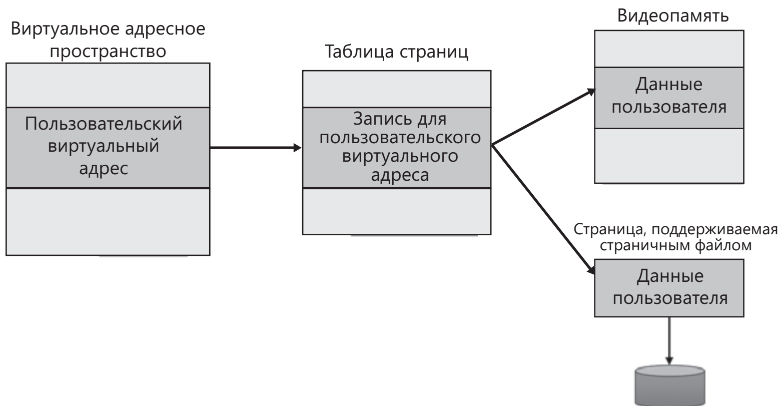

Рис. 5.30. Чередующиеся дескрипторы виртуальных адресов

NUMA

Каждый новый выпуск Windows предлагает новые варианты совершенствования диспетчера памяти для наилучшего использования машин с архитектурой неоднородного доступа к памяти (Non Uniform Memory Architecture, NUMA), например больших серверных систем (а также рабочих SMP-станций на процессорах Intel i7 и AMD Opteron). Поддержка технологии NUMA в диспетчере памяти наделяет его средствами получения информации об узлах, их местоположении, топологии и издержках доступа, что позволяет приложениям и драйверам пользоваться возможностями технологии NUMA, абстрагируясь от базовых деталей оборудования.

При инициализации диспетчера памяти он вызывает функцию `MiComputeNumaCosts` для выполнения различных операций, касающихся страниц и кэша на различных узлах с последующим вычислением затрачиваемого на завершение этих операций времени. На основании этой информации диспетчер памяти создает граф издержек доступа к узлу (расстояния между узлом и другими узлами системы). Когда системе потребуются страницы для заданной операции, она обращается к графу, чтобы выбрать наиболее оптимальный узел (т. е. ближайший). Если на этом узле нет доступной памяти, выбирается следующий ближайший узел и т. д.

Хотя диспетчер памяти гарантирует, что память по возможности будет выделяться процессорным узлом того программного потока, который реализует выделение, — так называемым *идеальным узлом* (ideal node), он также предоставляет API-функции `VirtualAllocExNuma`, `CreateFileMappingNuma`, `MapViewOfFileExNuma` и `AllocateUserPhysicalPagesNuma`, позволяющие приложениям выбирать собственный узел.

Идеальный узел используется не только при выделении памяти приложением, но и в ходе операций ядра и ошибок страницы. Например, когда поток выполняется на неидеальном процессоре и сталкивается с ошибкой отсутствия страницы, диспетчер памяти не станет трогать текущий узел, а вместо этого выделит память из идеального узла потока. Хотя это может привести к увеличению времени доступа, пока поток по-прежнему выполняется на данном центральном процессоре, в общем и целом, как только поток вернется на свой идеальный узел, доступ к памяти станет оптимальным. В любом случае, если идеальный узел не располагает свободными ресурсами, в качестве идеального будет выбран не какой-то случайный узел, а ближайший узел. Впрочем, драйверы, как и пользовательские приложения, могут указать собственный узел при помощи API-функции `MmAllocatePagesforMdlEx` или `MmAllocateContiguousMemorySpecifyCacheNode`.

Различные пулы и структуры данных диспетчера памяти также оптимизируются с целью получения преимуществ NUMA-узлов. Для поддержки невыгружаемого пула диспетчер памяти старается равномерно использовать физическую память всех узлов системы. После выделения памяти для невыгружаемого пула диспетчер памяти использует идеальный узел в качестве индекса при выборе диапазона адресов виртуальной памяти внутри невыгружаемого пула, который соответствует физической памяти, принадлежащей этому узлу. Кроме того, для эффективного

использования этих вариантов конфигурации памяти для каждого NUMA-узла создаются списки свободных пулов узлов. Кроме невыгружаемого пула точно так же между всеми узлами распределяются системный кэш и системные PTE-записи, а также ассоциативные списки диспетчера памяти.

И наконец, когда система удовлетворяет свои потребности в страницах, заполненных нулями, она делает это параллельно на разных NUMA-узлах, создавая потоки на родственных по типу оборудования NUMA-узлах, соответствующих узлам, на которых находится физическая память. Механизмы предварительной выборки и супервыборки (см. далее) также используют идеальный узел целевого процесса, а разрешимые ошибки страницы заставляют страницы мигрировать на идеальный узел, на котором выполняется поток, столкнувшийся с такой ошибкой.

Объекты разделов

Ранее при описании общей памяти уже отмечалось, что *объект раздела*, который в подсистеме Windows называется *объектом отображения файла* (file mapping object), представляет собой блок памяти, который может совместно использоваться двумя и более процессами. Объект раздела может быть отображен на страничный файл или на какой-нибудь другой файл на диске.

Исполнительная система применяет разделы для загрузки в память исполняемых образов, а диспетчер кэша использует их для доступа к данным в кэшированном файле. (Дополнительные сведения о том, как диспетчер кэша задействует объекты разделов, можно найти в главе 14 части 2.) Объекты разделов можно также использовать для отображения файла внутри адресного пространства процесса. После этого к файлу можно будет обращаться как к большому массиву, отображая различные представления объекта раздела и читая их из памяти и записывая в память, а не из файла и в файл (такие действия называются *вводом/выводом отображаемого файла*). При обращении программы к недостоверной странице (к странице, отсутствующей в физической памяти) происходит ошибка страницы, и диспетчер памяти автоматически помещает страницу в память из отображаемого (или страничного) файла. Если приложение вносит в страницу изменения, диспетчер памяти записывает изменения в файл в ходе обычных операций подкачки (или же приложение может сбросить представление, воспользовавшись Windows-функцией `FlushViewOfFile`).

Объекты разделов, так же как и другие объекты, выделяются и освобождаются диспетчером объектов. Диспетчер объектов создает и инициализирует заголовок объекта, который используется им для управления объектами, а диспетчер памяти определяет тело объекта раздела. Диспетчер памяти также реализует службы, которые могут вызываться потоками пользовательского режима для извлечения и изменения атрибутов, хранящихся в теле объектов разделов. Структура объекта раздела показана на рис. 5.31, а уникальные атрибуты, хранящиеся в объектах разделов, перечислены в табл. 5.14.