ГЛАВА 8

Системные механизмы

Операционная система Windows предоставляет ряд основных механизмов, используемых такими компонентами режима ядра, как исполнительная система, ядро и драйверы устройств. В данной главе рассматриваются следующие системные механизмы и описывается порядок их использования.

- Модель выполнения кода процессором, включая понятие уровней колец, сегментация, состояния задач, диспетчеризация системных прерываний, отложенный вызов процедур (deferred procedure call, DPC), асинхронный вызов процедур (asynchronous procedure call, APC), таймеры, рабочие потоки системы, диспетчеризация исключений и диспетчеризация системных служб.
- Барьеры спекулятивного выполнения и другие средства защиты от уязвимостей по линии приложений.
- Диспетчер объектов.
- Синхронизация, включая спин-блокировки, объекты диспетчера ядра, порядок реализации ожиданий, а также примитивы синхронизации, относящиеся к пользовательскому режиму, такие как адресованные ожидания, условные переменные, тонкие блокировки «чтение — запись» (slim reader-writer, SRW).
- Продвинутый локальный вызов процедур (Advanced Local Procedure Call, ALPC).
- Средство уведомлений Windows (Windows Notification Facility, WNF).
- WoW64.
- Фреймворк отладки для пользовательского режима.

Дополнительно в главе подробно описываются универсальная платформа Windows (Universal Windows Platform, UWP) и набор служб как пользовательского режима, так и режима ядра, на работу которых она опирается, в частности:

- пакетных приложений и службы развертывания AppX;
- приложений Centennial и Windows Desktop Bridge;
- диспетчера состояний процессов (Process State Management, PSM) и диспетчера жизненного цикла процессов (Process Lifetime Manager, PLM);
- модератора активности хоста (Host Activity Moderator, HAM) и модератора фоновой активности (Background Activity Moderator, BAM).

МОДЕЛЬ ВЫПОЛНЕНИЯ КОДА ПРОЦЕССОРОМ

В этом разделе мы рассмотрим внутренние механизмы архитектуры процессоров i386 от Intel и ее расширения — архитектуры AMD64, фигурирующей в современных системах. Хотя обе компании создавали эти технологии самостоятельно, стоит отметить, что сегодня они вместе реализуют решения друг друга. Поэтому, несмотря на присутствие вышеупомянутых терминов в именах файлов и ключей реестра Windows, названия x86 (32 бита) и x64 (64 бита) сегодня употребляются более часто.

Речь пойдет о сегментации, задачах, уровнях привилегий, критических механизмах. Наконец, мы рассмотрим понятия диспетчеризации системных прерываний и системных вызовов.

Сегментация

Высокоуровневые языки программирования, такие как C/C++ или Rust, преобразуются компилятором в *машинный код*, часто называемый *ассемблером*. Этот низкоуровневый язык позволяет обращаться к регистрам процессора напрямую. Зачастую программам доступны три основных вида регистров (из тех, что можно увидеть в отладчике):

- программный счетчик (Program Counter, PC), в архитектурах x86/x64 называемый *указателем команд* (Instruction Pointer, IP), где он представлен регистрами EIP (для x86) и RIP (для x64). Его значение всегда указывает на команду в машинном коде, исполняемую в данный момент, кроме некоторых 32-разрядных архитектур класса ARM;
- указатель стека (Stack Pointer, SP), представленный регистрами ESP (для x86) и RSP (для x64). Его значение указывает на позицию в памяти, соответствующую текущей вершине стека;
- все остальные, известные как регистры общего назначения (General Purpose Registers, GPR), среди которых EAX/RAX, ECX/RCX, EDX/RDX, ESI/RSI, а также R8, R14 и множество других.

Хотя перечисленные регистры могут содержать значения адресов в памяти, существует еще несколько, используемых в условиях *сегментации в безопасном режиме*. Последняя предполагает проверки для различных *сегментных регистров*, также называемых *селекторами*.

- Все попытки доступа к программному счетчику предварительно оцениваются относительно регистра сегмента кода (Code Segment, CS).
- Все попытки доступа к указателю стека предварительно оцениваются относительно регистра сегмента стека (Stack Segment, SS).
- Обращения к прочим регистрам определяются *переопределением сегмента*, в ходе чего кодирование позволяло задать проверку сегмента данных (Data Segment, DS), дополнительных сегментов ES или FS.

Эти селекторы размещаются в 16-разрядных регистрах, а их значения хранятся в структуре данных, которая называется *глобальной таблицей дескрипторов* (Global

Descriptor Table, GDT). Чтобы обращаться к ней, процессор хранит ее адрес в еще одном особом регистре — GDTR. Структура такого селектора показана на рис. 8.1.

Рис. 8.1. Структура селектора сегмента в системе x86

Тем самым смещение, находящееся в селекторе сегмента, берется из GDT, если только не установлен флаг TI (Table Indicator). В последнем случае происходит обращение к другой структуре данных — *локальной таблице дескрипторов* (Local Descriptor Table, LDT). Ее адрес хранится в регистре LDTR, и она не используется в современных системах Windows. Результатом поиска становится запись о сегменте, а в случае неуспеха — некорректная запись, что приведет к общей ошибке безопасности (#GP) или исключению ошибки сегмента.

Эта запись, называемая в современных ОС *дескриптором сегмента*, служит для двух важнейших целей.

- Для сегментов кода дескриптор содержит *код уровня привилегий* (Code Privilege Level, CPL). Эти уровни привилегий часто описываются как *защитные кольца*, на которых код будет выполняться. Данный уровень (в диапазоне от 0 до 3) затем кэшируется в двух старших битах самого селектора (см. рис. 8.1). Операционные системы, такие как Windows, применяют уровень (кольцо) 0 для запуска компонентов ядра и драйверов, а уровень 3 — для приложений и служб.

  Кроме того, в системах типа x64 сегмент кода имеет признак того, используется он в *длинном режиме* (long mode) или в *режиме совместимости*. Первый применяется для прямого выполнения кода под x64, а второй обеспечивает совместимость с кодом под x86. Аналогичный механизм существует и в x86, где сегменты могут помечаться как 16- или 32-разрядные.
- Для прочих сегментов в дескрипторе указывается *уровень привилегий дескриптора* (Descriptor Privilege Level, DPL), который необходим для доступа к сегменту. Хотя в современных системах подобная проверка уже анахронизм, процессоры все еще требуют (а приложения все еще ожидают) ее правильной настройки.

Наконец, в системах типа x86 дескрипторы сегмента могут дополнительно хранить 32-разрядный *базовый адрес*, в случае чего любое значение, уже помещенное в регистр в результате переопределения сегмента, будет считаться смещением от этого адреса. Чтобы смещение не превысило некоторого предела, оно сверяется с соответствующим *пределом сегмента*. Поскольку в большинстве ОС базовый адрес приравнен к 0 (а предел — к 0xFFFFFFFF), в архитектуре x64 этот концепт реализован везде, за исключением селекторов FS и GS, поведение которых несколько отличается.

- Если сегмент кода работает в длинном режиме, базовый адрес для сегмента FS берется из моделезависимого регистра FS_BASE (Model Specific Register, MSR) — 0C0000100h. Для сегмента GS нужно загрузить GS_BASE MSR-0C0000101h или GS_SWAP MSR-0C0000102h в зависимости от *состояния свопа (замещения)*, задаваемого инструкцией swapgs.

Если в селекторе сегмента FS или GS установлен бит TI, тогда значение берется из записи в LDT с соответствующим смещением, которое ограничено только 32-разрядными базовыми адресами. Так сделано для обеспечения совместимости с некоторыми операционными системами, а сам предел игнорируется.

- Если сегмент кода работает в режиме совместимости, тогда базовый адрес обычным образом считывается из записи в GDT (или LDT, если установлен бит TI). Предел устанавливается и проверяется по смещению в регистре, заданному после переопределения сегмента.

Такое поведение сегментов FS и GS позволяет операционным системам вроде Windows реализовать *эффект локальности регистров в рамках потока*, где к отдельным структурам данных можно обращаться по базовому адресу сегмента, что дает возможность получить простой доступ к конкретным смещениям/полям внутри него. К примеру, согласно описанию, приведенному главе 3 тома 1, Windows хранит адрес блока окружения потока (Thread Environment Block, TEB) в сегментном регистре FS в системах x86 и в GS (замещенном) в системах x64. В таком случае при выполнении кода уровня ядра в x86 FS вручную переключается на другой сегмент, который соответствует адресу Kernel Processor Control Region (KPCR). В свою очередь, для архитектуры x64 этот адрес сразу помещается в регистр GS (не замещенный).

Таким образом, сегментация позволяет реализовать в Windows следующие эффекты: закодировать и установить уровень привилегий, который будет иметь исполняемый фрагмент кода на конкретном уровне процессора, обеспечить прямой доступ к структурам данных TEB и KPCR в пользовательском режиме и/или режиме уровня ядра соответственно. Заметим, что поскольку на глобальную таблицу дескрипторов ссылается регистр процессора GDTR, у каждого процессора она может быть своя. На самом деле именно этим Windows пользуется, чтобы обеспечить каждому процессору загрузку правильного KPCR для каждой GDT и то, что для текущего исполняемого потока на текущем процессоре его TEB находится в нужном сегменте.

ЭКСПЕРИМЕНТ. Просмотр GDT в x64

При проведении удаленной отладки или анализе аварийного дампа (в том числе при использовании LiveKD) вы сможете просмотреть содержимое GDT, включая состояние всех сегментов и их базовые адреса (где это применимо), с помощью команды отладчика dg. Она принимает на вход стартовый и конечный сегменты, в приведенном примере это 10h и 50h:

```
0: kd> dg 10 50

                                                    P Si Gr Pr Lo
Sel        Base              Limit          Type    l ze an es ng Flags
---- ----------------- ----------------- ---------- - -- -- -- -- --------
0010 00000000`00000000 00000000`00000000 Code RE Ac 0 Nb By P  Lo 0000029b
0018 00000000`00000000 00000000`00000000 Data RW Ac 0 Bg By P  Nl 00000493
0020 00000000`00000000 00000000`ffffffff Code RE Ac 3 Bg Pg P  Nl 00000cfb
0028 00000000`00000000 00000000`ffffffff Data RW Ac 3 Bg Pg P  Nl 00000cf3
0030 00000000`00000000 00000000`00000000 Code RE Ac 3 Nb By P  Lo 000002fb
0050 00000000`00000000 00000000`00003c00 Data RW Ac 3 Bg By P  Nl 000004f3
```

Ключевые сегменты здесь 10h, 18h, 20h, 28h, 30h и 50h. (Приведенный вывод несколько сокращен: удалена информация, не относящаяся к данной теме.)

В 10h (KGDT64_R0_CODE) вы можете наблюдать сегмент кода в длинном режиме на уровне нулевого кольца, что видно по `0` в столбце `Pl`, коду `Lo` в столбце `Long` и типу `Code RE`. Аналогично в 20h (KGDT64_R3_CMCODE) вы видите `Nl`-сегмент в третьем кольце (not long, то есть в режиме совместимости), где сейчас выполняется код для x86 в подсистеме WOW64. В 30h (KGDT64_R3_CODE), в свою очередь, находится аналогичный сегмент в длинном режиме. Затем обратите внимание на 18h (KGDT64_R0_DATA) и 28h (KGDT64_R3_DATA), которые соответствуют сегментам стека, данных и дополнительному сегменту.

Наконец, последний сегмент по 50h (KGDT_R3_CMTEB), как правило, имеет нулевой базовый адрес, если только вы не запускаете код под x86 в WOW64 при выводе дампа GDT. Как было сказано ранее, здесь будет храниться адрес TEB при активном режиме совместимости.

Чтобы посмотреть сегменты TEB и KPCR в 64-разрядном режиме, вам вместо этого потребуется выгрузить (dump) соответствующие MSR, что можно сделать следующими командами (актуально при удаленной и локальной отладке ядра, но не при анализе дампа):

```
lkd> rdmsr c0000101
msr[c0000101] = ffffb401`a3b80000

lkd> rdmsr c0000102
msr[c0000102] = 000000e5`6dbe9000
```

Вы можете сравнить эти значения с содержимым `@$pcr` и `@$teb`, где будут находиться те же данные:

```
lkd> dx -r0 @$pcr
@$pcr                 : 0xffffb401a3b80000 [Type: _KPCR *]
lkd> dx -r0 @$teb
@$teb                 : 0xe56dbe9000 [Type: _TEB *]
```

ЭКСПЕРИМЕНТ. Просмотр GDT в x86

В x86-системах GDT оформляется похожими сегментами, но с другими селекторами. Кроме того, из-за двойного назначения сегментного регистра FS вместо функциональности swapgs, а также из-за отсутствия длинного режима количество селекторов несколько отличается, что видно в примере:

```
kd> dg 8 38

                                 P Si Gr Pr Lo
Sel    Base     Limit     Type    l ze an es ng Flags
---- -------- -------- ---------- - -- -- -- -- --------
0008 00000000 ffffffff Code RE Ac 0 Bg Pg P  Nl 00000c9b
0010 00000000 ffffffff Data RW Ac 0 Bg Pg P  Nl 00000c93
```

```
0018 00000000 ffffffff Code RE    3 Bg Pg P  Nl 00000cfa
0020 00000000 ffffffff Data RW Ac 3 Bg Pg P  Nl 00000cf3
0030 80a9e000 00006020 Data RW Ac 3 Bg By P  Nl 00000493
0038 00000000 00000fff Data RW    3 Bg By P  Nl 000004f2
```

Ключевые сегменты здесь 8h, 10h, 18h, 20h, 30h и 38h. По смещению 08h (KGDT_R0_CODE) можно наблюдать сегмент кода на уровне нулевого кольца. Аналогично по смещению 18h (KGDT_R3_CODE) находится сегмент на уровне третьего кольца. Наконец, в 10h (KGDT_R0_DATA) и 20h (KGDT_R3_DATA) будут сегменты стека, данных и дополнительный сегмент.

В системах x86 в сегментах 30h (KGDT_R0_PCR) и 38h (KGDT_R3_TEB) будут базовые адреса KPCR и TEB текущего потока соответственно. В данной архитектуре никаких MSR для сегментации не используется.

Отложенная загрузка сегментов

С учетом описанного ранее поведения сегментов исследование значений сегментных регистров DS и ES в системах x86 и/или x64 может дать неожиданные результаты: они не всегда соответствуют значениям определенных для них уровней кольца. К примеру, какой-нибудь поток в архитектуре x86 в пользовательском режиме будет иметь следующие сегменты:

```
CS = 1Bh (18h | 3)
ES, DS = 23 (20h | 3)
FS = 3Bh (38h | 3)
```

Но во время системного вызова на уровне нулевого кольца обнаружатся другие сегменты:

```
CS = 08h (08h | 0)
ES, DS = 23 (20h | 3)
FS = 30h (30h | 0)
```

Похожим образом поток уровня ядра для x64 будет устанавливать свои сегментные регистры ES и DS в 2Bh (28h | 3). Это несоответствие сопутствует функциональности, называемой *отложенной загрузкой сегментов*, и отражает бессмысленность уровня привилегий дескриптора (DPL) для сегмента данных в ситуации, когда текущий уровень привилегий кода (CPL) равен 0 в условиях системы с плоской моделью памяти. Поскольку более высокий CPL всегда имеет доступ к данным более низких DPL (но не наоборот), установка DS и/или ES в «правильное» значение при входе в ядро потребовала бы их восстановления при возвращении в пользовательский режим.

Хотя инструкция MOV DS, 10h может показаться банальной, при встрече с ней микрокоду процессора потребуется выполнить несколько проверок на корректность селектора, что существенно повысит стоимость системных вызовов и обработки прерываний. Чтобы избежать этого, Windows всегда использует значения для регистров сегментов данных с уровня кольца 3.

Сегменты состояния задач

Кроме сегментных регистров кода и данных в системах архитектур x86 и x64, существует дополнительный особый регистр — регистр задач (Task Register, TR), который работает как еще один 16-битный селектор, где хранится смещение в пределах GDT. Однако в данном случае сегмент ассоциирован не с кодом или данными, а скорее с *задачей*. С точки зрения внутреннего состояния процессора он характеризует исполняемый фрагмент кода, называемый *Task State*, в случае Windows — текущий поток. Эти состояния задач, представленные сегментами (сегмент состояния задачи — Task State Segment, TSS), используются в современных операционных системах под x86 для решения ряда задач, связанных с критическими перехватами в процессоре (подробнее о них — в следующем разделе). Как минимум TSS характеризует каталог страниц (через регистр CR3), например PML4 в системах x64 (подробнее о страницах см. в главе 5 тома 1), сегмент кода, сегмент стека, указатель инструкций (IP) и до четырех указателей на стек (по одному на каждый уровень кольца). Подобные TSS используются в следующих целях.

- Они характеризуют текущее состояние исполнения кода в моменты, когда не происходит каких-либо системных прерываний. Впоследствии процессор пользуется этим для корректной обработки исключений и прерываний, загружая стек нулевого кольца из TSS, если перед этим он работал на уровне третьего кольца.
- Для разрешения ситуаций архитектурной гонки при работе с отказами отладки (#DB), для чего требуется выделенный TSS с отдельным обработчиком отказов отладки и стеком ядра.
- Характеризуют состояние исполнения кода, которое следует применить в случае срабатывания ловушки двойного отказа (#DF). С их помощью выполняется переход в обработчик двойного отказа с безопасным (запасным) стеком ядра вместо стека текущего потока, который и мог вызвать отказ.
- Характеризуют состояние исполнения кода, которое следует применить для немаскируемого прерывания (Non Maskable Interrupt, NMI). Используются для перехода в обработчик NMI с безопасным стеком ядра.
- Наконец, в похожих ситуациях при исключении машинных проверок (#MCE), которые по тем же причинам могут исполняться в условиях отдельного, безопасного стека ядра.

В системах x86 главный (текущий) TSS окажется в GDT по селектору 028h, что объясняет наличие в регистре TR аналогичного значения в ходе нормального выполнения кода в Windows. Кроме того, #DF TSS будет по 58h, NMI TSS — по 50h, а #MCE TSS — по 0A0h. Наконец, #DB TSS будет по 0A8h.

В системах x86 нет возможности содержать несколько TSS, так как эта функциональность была сведена в основном к единственной необходимости исполнять обработчики системных прерываний, выполняющиеся на выделенном стеке ядра. Таким образом, теперь используется лишь один TSS (в случае с Windows — по 040h), где содержится массив из восьми указателей стека, известный как таблица стеков прерываний (Interrupt Stack Table, IST). Каждое из предшествующих системных прерываний здесь ассоциируется с позицией в IST, а не с отдельным TSS. В следующем разделе, где мы разберем несколько записей IDT, вы сможете увидеть разницу в поведении систем x86 и x64 и в том, как они обрабатывают системные прерывания.

ЭКСПЕРИМЕНТ. Просмотр структур TSS в системах x86

В системе x86 мы можем заглянуть в общесистемный TSS по селектору 28h, воспользовавшись уже знакомой нам командой dg:

```
kd> dg 28 28
                                  P Si Gr Pr Lo
Sel    Base     Limit     Type    l ze an es ng Flags
---- -------- -------- ---------- - -- -- -- -- --------
0028 8116e400 000020ab TSS32 Busy 0 Nb By P  Nl 0000008b
```

На выходе появится виртуальный адрес структуры данных KTSS, которую можно выгрузить с помощью команд dx или dt:

```
kd> dx (nt!_KTSS*)0x8116e400
(nt!_KTSS*)0x8116e400                : 0x8116e400 [Type: _KTSS *]
    [+0x000] Backlink        : 0x0 [Type: unsigned short]
    [+0x002] Reserved0       : 0x0 [Type: unsigned short]
    [+0x004] Esp0            : 0x81174000 [Type: unsigned long]
    [+0x008] Ss0             : 0x10 [Type: unsigned short]
```

Важно отметить, что только двум полям, Esp0 и Ss0, присвоены значения. Это связано с тем, что Windows никогда не прибегает к аппаратному переключению задач, кроме случаев с системными прерываниями, описанных ранее. Таким образом, основным назначением данного конкретного TSS является загрузка подходящего стека ядра во время обработки аппаратных прерываний.

Как вы узнаете из раздела «Диспетчеризация системных прерываний», в системах, не подверженных архитектурной уязвимости Meltdown в процессоре, данный указатель стека будет ссылаться на стек ядра текущего потока (согласно структуре KTHREAD, описанной в главе 5 тома 1). В системах, где такая уязвимость есть, он будет указывать на переходный стек из зоны дескриптора процессора (Processor Descriptor Area, PDA). В то же время сегмент стека всегда в селекторе 10h, или KGDT_R0_DATA.

Как говорилось ранее, еще один TSS используется для исключения машинных проверок (#MC). На него можно посмотреть с помощью команды dg:

```
kd> dg a0 a0
                                  P Si Gr Pr Lo
Sel    Base     Limit     Type    l ze an es ng Flags
---- -------- -------- ---------- - -- -- -- -- --------
00A0 81170590 00000067 TSS32 Avl  0 Nb By P  Nl 00000089
```

Однако на этот раз вместо dx мы используем команду .tss, которая отформатирует некоторые поля из KTSS и отобразит задачу так, будто это поток, исполняемый в текущий момент. В данном случае входным параметром будет селектор задач (A0h):

```
kd> .tss a0

eax=00000000 ebx=00000000 ecx=00000000 edx=00000000 esi=00000000 edi=00000000
eip=81e1a718 esp=820f5470 ebp=00000000 iopl=0       nv up di  pl nz na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000000

hal!HalpMcaExceptionHandlerWrapper:
81e1a718 fa              cli
```

Обратите внимание на то, что сегментные регистры настраиваются так, как было описано в разделе про отложенную загрузку сегментов, а программный счетчик (EIP) ссылается на обработчик для #MC. Кроме того, стек настроен на безопасный стек из кода ядра, который не должен быть затронут повреждениями памяти. Наконец, пусть этого и не видно по команде `.tss`, CR3 указывает на системный каталог страниц. В разделе «Диспетчеризация системных прерываний» мы вернемся к данному TSS при рассмотрении команды `!idt`.

ЭКСПЕРИМЕНТ. Просмотр TSS и IST в системе x64

К сожалению, в системах x64 команда `dg` содержит баг, не позволяющий корректно отображать 64-битные базовые адреса сегментов, в силу чего для получения базового адреса сегмента TSS (по 40h) надо выгрузить как бы два сегмента и совместить старший, средний и младший байты:

```
0: kd> dg 40 48
                                                    P  Si  Gr  Pr  Lo
Sel        Base              Limit          Type    l ze an es ng Flags
---- ----------------- ----------------- ---------- - -- -- -- -- --------
0040 00000000`7074d000 00000000`00000067 TSS32 Busy 0 Nb By P  Nl 0000008b
0048 00000000`0000ffff 00000000`0000f802 <Reserved> 0 Nb By Np Nl 00000000
```

Таким образом, в приведенном примере KTSS64 оказывается по адресу `0xFFFFF8027074D000`. Есть и другой способ его получить: KPCR каждого процессора содержит поле `TssBase`, где также хранится указатель на KTSS64:

```
0: kd> dx @$pcr->TssBase
@$pcr->TssBase                : 0xfffff8027074d000 [Type: _KTSS64 *]
    [+0x000] Reserved0        : 0x0 [Type: unsigned long]
    [+0x004] Rsp0             : 0xfffff80270757c90 [Type: unsigned __int64]
```

Заметьте, что виртуальный адрес совпадает с тем, который можно увидеть в GDT. А также что все поля, кроме `RSP0`, обнулены. Заполненное поле, как и в системе x86, содержит либо адрес стека ядра для текущего потока (для систем без аппаратной уязвимости Meltdown), либо адрес переходного стека и зоны дескриптора процессора (PDA).

В системе, где проводился эксперимент, использовался процессор Intel десятого поколения, следовательно, `RSP0` указывает на текущий стек ядра:

```
0: kd> dx @$thread->Tcb.InitialStack
@$thread->Tcb.InitialStack : 0xfffff80270757c90 [Type: void *]
```

Наконец, взглянув на таблицу стеков прерываний, мы можем наблюдать различные стеки, связанные с системными прерываниями #DF, #MC, #DB и NMI.

В разделе «Диспетчеризация системных прерываний» мы рассмотрим, как таблица обработки прерываний (IDT) ссылается на них:

```
0: kd> dx @$pcr->TssBase->Ist
@$pcr->TssBase->Ist     [Type: unsigned __int64 [8]]
    [0]              : 0x0 [Type: unsigned __int64]
    [1]              : 0xfffff80270768000 [Type: unsigned __int64]
    [2]              : 0xfffff8027076c000 [Type: unsigned __int64]
    [3]              : 0xfffff8027076a000 [Type: unsigned __int64]
    [4]              : 0xfffff8027076e000 [Type: unsigned __int64]
```

Теперь, когда связь между уровнем кольца, исполнением кода и некоторыми ключевыми сегментами GDT прояснена, можно взглянуть на реальные переходы между различными сегментами кода и их уровнями кольца в разделе о диспетчеризации системных прерываний.

Прежде чем перейти к этой теме, проанализируем, как меняется конфигурация TSS в системах, уязвимых к атакам с аппаратной стороны типа Meltdown.

АППАРАТНЫЕ УЯЗВИМОСТИ К АТАКАМ ПО СТОРОННИМ КАНАЛАМ

Современные микропроцессоры способны вычислять и перемещать данные между своими внутренними регистрами очень быстро (счет идет на пикосекунды). Но сами эти регистры — очень ограниченный ресурс. В связи с этим операционной системе и приложениям приходится постоянно требовать от процессора перемещать данные из регистров в память и обратно. Существуют различные типы памяти, к которым он может обратиться. Память, находящаяся на самом процессоре и доступная непосредственно из блока исполнения, называется «кэш» и известна своей скоростью и дороговизной. Память, доступная по внешней шине, обычно называется RAM (Random Access Memory) и характеризуется низкой скоростью, малой стоимостью и большим объемом. Близость памяти к процессору помещает ее в некоторую иерархию, положение в которой зависит от скорости и физических размеров (чем ближе память к процессору, тем она быстрее и миниатюрнее). Как показано на рис. 8.2, в современных компьютерах процессоры часто имеют три уровня быстрой кэш-памяти, доступной каждому физическому ядру: L1, L2 и L3. Первые два ближе всех, и у каждого ядра процессора они свои. Кэш L3 дальше всех и всегда общий для всех ядер (заметим, что у встраиваемых процессоров кэш L3, как правило, отсутствует).

Одной из важнейших характеристик кэша является время доступа к нему, сравнимое с временем обращения к регистрам процессора (однако еще медленнее). Тем не менее обращение к основной памяти длится в сотни раз дольше. Это значит, что, если процессор будет исполнять все инструкции по порядку, он столкнется с большими паузами, вызванными попытками достучаться туда. Для решения этой проблемы в современных архитектурах прибегают к различным стратегиям. Исторически они привели к изобретению атак по сторонним каналам (также известных как спекулятивные атаки), очень эффективных при преодолении средств защиты пользовательских систем.

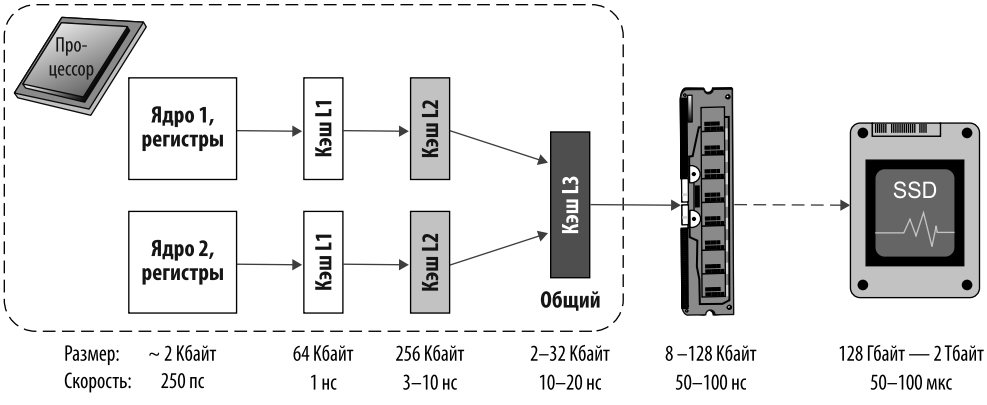

Рис. 8.2. Кэши и память современных процессоров, их средние размеры и время отклика

Чтобы правильно описать природу аппаратных атак по сторонним каналам и способы, которыми Windows борется с ними, следует рассмотреть несколько основных принципов работы внутренних механизмов процессора.

Внеочередное выполнение команд

Современный микропроцессор исполняет машинный код с помощью своего вычислительного конвейера. Тот включает в себя множество стадий, в числе которых — прочтение команды, декодирование, переименование и распределение регистров, переупорядочение, исполнение и фиксация результата выполнения. Когда процессору требуется обойти проблему медленной памяти, типичной стратегией является придание исполнительному устройству способности выполнять команды не по порядку, а в зависимости от готовности необходимых ресурсов. Таким образом, процессор обрабатывает код нелинейно, что позволяет максимально продуктивно задействовать все исполнительные устройства его ядра. Современный процессор способен исполнять сотни команд спекулятивно, пока однозначно не наступит момент, когда они потребуются, а их результат будет зафиксирован.

Одну из проблем описанного подхода создают команды условного перехода. Вычисление условия определяет два возможных пути дальнейшего продвижения по коду. Какой из них верен, зависит от команд, исполненных ранее. Когда этим командам требуется доступ к медленной памяти, могут возникать простои. В таком случае исполнитель ожидает, пока результат выполнения команд, определяющих условие, не будет зафиксирован (а именно, ожидание шины памяти, пока та обеспечивает доступ), чтобы потом продолжить внеочередное выполнение команд по корректному пути. Похожая проблема возникает при непрямом переходе. В этом случае исполнительное устройство не в курсе того, в какое место в коде произойдет переход (обычно это прыжок или вызов), поскольку целевой адрес еще нужно извлечь из основной памяти. В такой ситуации под *спекулятивным выполнением* подразумевается, что конвейер процессора декодирует и исполняет по нескольку команд параллельно, или, иными словами, вне очереди. Тем не менее результаты их исполнения не помещаются в регистры процессора, а запись данных в память откладывается по тех пор, пока условие перехода не будет окончательно вычислено.