Министерство образования и науки Российской Федерации

МОСКОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ (государственный университет)

ФАКУЛЬТЕТ УПРАВЛЕНИЯ И ПРИКЛАДНОЙ МАТЕМАТИКИ

КАФЕДРА ТЕОРЕТИЧЕСКОЙ И ПРИКЛАДНОЙ ИНФОРМАТИКИ

(Специальность 03.04.01 «Прикладная математика и физика»)

ДЕДУПЛИКАЦИЯ СТРАНИЦ ИСПОЛНЯЕМОГО КОДА ДРАЙВЕРОВ ОС WINDOWS.

Выпускная квалификационная работа

студентки 973 группы

Кудиновой Марины Викторовны _____

Научный руководитель

Тормасов А. Г. _____

Научный консультант

Костюшко А. В. _____

г. Долгопрудный

2015

Оглавление

Аннотация 4
1. Введение 5
1.1 Постановка проблемы 5
1.2 Цель работы 5
1.3. Обзор существующих методов ее решения 6
1.3.1. KVM 6
1.3.2. VMware 6
1.3.3. Недостатки существующих методов 6
1.3.4. Особенности данной ситуации 6
2. Решение 15
2. 1. Предлагаемый метод решения 15
2. 1.1. Плюсы и минусы относительно других методов 15
2.1.2. Жизненный цикл драйвера 15
2.1.2.1. Начальная ситуация. 15
2.1.2.2. Начало работы 16
2.1.2.3. Работа системы 17
2.1.2.4. Завершение работы одного из драйверов 19
2.1.2.5. Необходимость использования нулевого набора 20
2.1.2.6. Итоговый алгоритм решения 20
2.2. Реализация 24
2.2.1. Ограничения 24
2.2.1.1. Рассматриваемые типы драйверов 24
2.2.1.2. Рассматриваемые типы страниц 24
2.2.1.3. Copy On Write в Windows 26
2.2.1.4. Итог 27
2.2.3. Имплементация механизма копирования при записи (COW) 35
2.2.3.1. Цель 35
2.2.3.2. Предлагаемый метод решения 35
2.2.3.3. Трансляция виртуального адреса в физический 35
2.2.3.4. Объединение страниц 36
2.2.3.5. Разъединение страниц 38
2.2.3.6. Обработка Page Fault 38
2.2.3.7. COW и высокие IRQL и связанные с ними проблемы 40
2.3. Математическая модель работы 41

2.3.1. Основные положения 41
2.3.1.1. Различные типы страниц памяти 41
2.3.1.2. NonPageable read-only страницы 42
2.3.1.3. Pageable readonly страницы 43
2.3.1.4. Pageable writable страницы 43
2.3.1.5. NonPageable writable страницы 45
2.3.1.6. Итог 46
2.3.3. Математическая модель разделения страниц при записи (Модель №2) 47
2.3.3.1. Основные положения 47
2.3.3.2. Простейший случай: 1 драйвер и 1 страница. Обоснование применимости. 47
2.3.3.3. Время до первой записи как случайная величина 50
2.3.3.4. Пуассоновский процесс 53
2.3.3.5. Модификация Пуассоновского процесса. 55
2.3.3.6. Итоги. 56
2.4. Тестирование и сравнение результатов с математической моделью 58
2.4.1. Сбор статистики 58
2.4.2. Нахождение λ и сравнение результатов с математической моделью 59
2.4.3. Время загрузки драйвера 62
2.4.4. Тестирование Pageable страниц. 62
2.4.5. Подсчет выигрышей. 63
3. Заключение 65
3.1. Результаты 65
3.2. Перспективы развития 66
3.3. Возможные применения 67
3.3.1. Возможные применения метода дедупликации одинаковых страниц. 67
3.3.2. Возможные применения Copy On Write в NonPaged Memory 67
4. Список литературы 69

Аннотация

Данной работе исследовались способы дедупликации исполняемого кода драйверов в ОС Windows, был предложен и реализован метод дедупликации, который был протестирован и доказал свою эффективность. В процессе работы над этим методом был имплементирован ранее отсутствовавший в ОС Windows механизм копирования при записи (Copy On Write или COW) для неподкачиваемого пула памяти. Этот механизм сам по себе может быть иметь очень широкое применение, в следствие чего, о нем в данной работе говорится отдельно.

1. Введение

Рассмотрим ситуацию, когда в ОС Windows одновременно исполняются несколько идентичных драйверов. Такая ситуация возникает, например, при контейнерной виртуализации, где для каждого контейнера приходится устанавливать свои драйвера, которые очень часто совпадают для разных контейнеров вплоть до версии [17, 18]. Исполняемый код каждого из этих по сути идентичных экземпляров одного драйвера загружается в память машины. В случае большого количества контейнеров размер занимаемой им памяти исчисляется уже десятками Мб.

Таким образом в системе находятся наборы идентичных страниц, которые в случае нахождения в оперативной памяти занимают место, которое могло бы в противном случае быть использовано с пользой, например, для запуска еще одного контейнера. А в случае, когда эти страницы принадлежат подкачиваемому пулу памяти, они создают дополнительную нагрузку на подсистему подкачки страниц. Например, если в системе заканчивается память, и драйверы попеременно обращаются к нескольким идентичным страницам, то менеджеру памяти приходится постоянно то подгружать эти страницы с диска, то сбрасывать их обратно в файл подкачки. Этого можно было бы избежать, имея всего одну физическую страницу вместо многих одинаковых. Таким образом приходим к постановке проблемы.

1.1 Постановка проблемы

Когда в систему загружено много идентичных драйверов. Страницы их кода, во-первых, идентичны, во-вторых, каждая из них занимает память, а в-третьих, если это подкачиваемые страниц, то они создают дополнительную нагрузку на подсистему подкачки страниц.

1.2 Цель работы

Цель работы - создание механизма дедупликации страниц исполняемого кода идентичных драйверов.

Все драйверы с идентичным бинарным кодом должны использовать один общий набор страниц, а не работать каждый со своим, как раньше.

Таким образом необходимо создать метод: алгоритм и его рабочую реализацию, которые бы могли заменить в памяти несколько идентичных наборов страниц кода драйвера одним единственным экземпляром.

1.3. Обзор существующих методов ее решения

1.3.1. **KVM**

Same Page Merging (KSM), которая применяется в Linux и KVM. Впервые появилась в 2008г. [19]. Она позволяет ядру объединять одинаковые страницы памяти различных процессов или виртуальных гостевых систем в одну для совместного использования [19 - 22].

1.3.2. VMware

Transparent page sharing (TPS) от VMware - это техника, в которой идентичные блоки виртуальной памяти, используемые в гостевых ОС в виртуальных машинах, хранятся только в одном экземпляре в физической памяти хостовой ОС [23 - 25].

1.3.3. Недостатки существующих методов

Эти два метода: KSM и TPS - достаточно похожи и с учетом того, что один объединяет отдельные страницы, а второй - блоки страниц.

Однако, они имеют свои недостатки. Например, им требуется периодическое сканирование памяти в поисках одинаковых страниц, или же, они могут анализировать страницы, только тогда, когда они загружаются в память. Однако, при этом все равно увеличивается нагрузка на процессор.

А главное, так как данная задача изначально относится к контейнерной виртуализации для Windows, то оба метода неприменимы.

1.3.4. Особенности данной ситуации

Основная задача – не поиск страниц, а их объединение.

Данная ситуация вообще достаточно необычна, поскольку основная задача здесь не поиск страниц, а их объединение. Не нужно искать одинаковые страницы, так как знаем название и версию драйвера и можем отследить его загрузку.

Работа с NonPaged Memory в Windows, где нет механизма Copy On Write.

Еще одно отличие - работа с NonPaged памятью в Windows, где тем механизма Copy On Write. Придется его сделать.

1.4. Обзор литературы

В этом пункте будут описаны все необходимые понятия, определения и предметная область.

1.4.1. Виртуализация

Обзор технологий виртуализации.

Задача и методы ее решения, излагаемые в данной работе, тесно связаны с виртуализацией. Следовательно, необходимо сначала провести небольшой обзор этой темы, и дать определения используемым терминам.

Виртуализация.

У термина "виртуализация" есть несколько значений [10, 26]:

- Виртуализация в вычислениях – процесс представления набора вычислительных ресурсов, или их логического объединения, который дает какие-либо преимущества перед оригинальной конфигурацией**.**

- Виртуализация – методология разделения ресурсов компьютера в среды исполнения по средствам таких концепций или технологий, как разбиение на партиции, разделение времени, симуляция, эмуляция, качество обслуживания и другие.

- Виртуализация – это способ абстрагирования реального используемого объекта от концептуального представления о нем у его пользователя.

В данной работе речь пойдет о последнем значении.

Виртуальная машина.

Виртуальная машина — это окружение, которое представляется для «гостевой» операционной системы, как аппаратное. Однако на самом деле это программное окружение, которое эмулируется программным обеспечением хостовой системы. Эта эмуляция должна быть достаточно надёжной, чтобы драйверы гостевой системы могли стабильно работать. При использовании паравиртуализации, виртуальная машина не эмулирует аппаратное обеспечение, а, вместо этого, предлагает использовать специальное API. [7]

Гипервизор.

Гипервизор, также называемый монитором виртуальных машин, это программа, позволяющая на одном компьютере (хосте) запускать несколько операционных систем (называемых гостевыми). Название это возникло из того, что, фактически, гипервизор находится на уровень выше супервизора (системы, управляющей запуском обычных пользовательских приложений). Гипервизор предоставляет гостевым операционным системам виртуальную аппаратную платформу, отвечает за изоляцию гостевых систем друг от друга, разделение и упраление ресурсами. Гипервизор также может предоставлять

средства связи между гостевыми операционными системами по типу, например, внутренней компьютерной сети.

Гипервизор предоставляет возможности по отдельному отключению или включению машин, подключению к ним различного эмулированного оборудования, а также и «проброс» реального оборудования в виртуальную машину (например, клавиатуры и мыши, или даже сетевой карты и cd-привода).

Гипервизоры разделяют на два типа: запускаемые на аппаратуре как операционные системы (на “bare metal”) и запускаемые под обычной операционной системой.

«прозрачность» при управлении ресурсами. Также такой подход позволяет добиться большей производительности. Из недостатков можно отметить необходимость включения в гипервизор драйверов для поддержки различного оборудования. [11, 12]

Гипервизоры второго типа работают под некоторой операционной системой, такой как Windows или Linux. Такой вариант более доступен обычным пользователям, кроме того исчезает необходимость в драйверах устройств, так как низкоуровневая работа с устройствами в данном случае ложится на плечи хостовой операционной системы.

Контейнерная виртуализация.

Контейнерная виртуализация - это подвид виртуализации, в котором виртуализуется только пространство пользователя, называкемое контейнером, а ядро остается общим для всех контейнеров. Плюсы относительно виртуальных машин - экономия места, можно развернуть гораздо больше контейнеров, по сравнению с числом виртуальных машин, на одном и том же оборудовании. Минусы - все эти контейнеры будут иметь одинаковую операционную систему, т.к. имеют общее ядро. В каждый контейнер грузится собственный набор драйверов. А так как они часто совпадают, встает задача дедупликации их кода [17, 18].

Технологии виртуализации.

Существуют различные технологии виртуализации, имеющие свои достоинства и недостатки. И выбор конкретной технологии зависит от поставленной задачи [26].

Эмуляция.

Эмулируется каждая команда процессора. То есть на одну ассемблерную команду эмулируемого кода приходится насколько команд физического процессора. Этот способ надежен (обеспечивает полную изоляцию), универсален (позволяет исполнять на одной архитектуре процессора программы, написанные для другой), но чересчур медленный.

Бинарная (динамическая) трансляция.

В данном случае, только "проблемные" (то есть те, исполнение которых различается в зависимости от того, в каком кольце защиты процессора они исполняются) команды гостевой ОС перехватываются гипервизором или монитором виртуальной машины. После того как эти команды заменяются на безопасные, происходит возврат управления гостевой ОС.

Стаббирование.

Более быстрый способ, чем бинарная трансляция. Очень похож на нее, но с созданием "стабов", то есть если в бинарной трансляции анализируется каждый кусок кода, то тут некоторые участки кода сразу вызывают переключение в гипервизор.

Паравиртуализация.

Паравиртуализация — техника виртуализации, при которой гостевые операционные системы подготавливаются для исполнения в виртуализированной среде, для чего их ядро незначительно модифицируется. Операционная система взаимодействует с программой гипервизора, который предоставляет ей гостевой API, вместо использования напрямую таких ресурсов, как таблица страниц памяти. Метод паравиртуализации позволяет добиться более высокой производительности, чем метод динамической трансляции.

Аппаратная виртуализация.

Аппаратная виртуализация — виртуализация с поддержкой специальной процессорной архитектуры. В отличие от программной виртуализации, с помощью данной техники возможно использование изолированных гостевых систем, управляемых гипервизором напрямую. Гостевая система не зависит от архитектуры хостовой платформы и реализации платформы виртуализации. Например, с помощью технологий аппаратной виртуализации возможен запуск 64-битных гостевых систем на 32-битных хостовых системах.

Аппаратная виртуализация обеспечивает производительность, сравнимую с производительностью невиртуализованной машины, что дает виртуализации возможность практического использования и влечет её широкое распространение. Наиболее распространены технологии виртуализации Intel-VT и AMD-V. [2, 7, 8, 27].

Особенности аппаратной виртуализации []:

- Монитор виртуальной машины(BMM) исполняется как vmroot.
- Гость исполняется со своими обычным полномочиями.
- Большая часть системных данных процессора описана в vmcs.
- Vmexit-ы в случае доступа к невиртуализованным данным.

1.4.2. Драйверы в ОС Windows

Ядро.

Термин "ядро" а ОС Windows имеет несколько значений. В данной работе во "Введении" и в "Архитектуре решения" под ядром ОС будет чаще всего пониматься к коду, работающему в режиме ядра процессора. В части "Метод решения и реализация" под термином "ядро" будет пониматься ntoskrnl.exe.

Драйверы.

В современных ОС для управления каждым подключенным к компьютеру устройством ввода-вывода требуется специальная программа, учитывающая его особенности. Она называется **драйвером устройства**. Каждый драйвер устройства управляет обычно одним типом (или как максимум одним классом) устройств. Большинство современных ОС исполняют драйвера в контексте ядра ОС. Функции драйверов устройств: принятие абстрактных запросов на чтение-запись от абстрагированного от оборудования ПО и остлеживание порядка их выполнения; инициализация устройств; управление устройством. Кроме того, все драйвера в системе имеют одинаковый интерфейс. [5]

Рассмотрим, как дело обстоит в ОС Windows. Драйверы устройств являются загружаемыми модулями режима ядра (как правило это файлы с расширением .sys). Они образуют интерфейс между диспетчером ввода-вывода и соответствующим оборудованием. Эти драйверы выполняются в режиме ядра в одном из трех контекстов:

- в контексте пользовательского потока, инициировавшего функцию ввода-вывода;
- в контексте системного потока режима ядра;
- как результат прерывания (а значит, не в контексте какого-либо процесса или потока, который был текущим на момент прерывания). [1]

На Рис.1. изображено внутреннее устройство ядра ОС Windows. Как видим, драйверы играют в ней очень важную роль.

Рис. 1 Организация режима ядра Windows

В Windows для управления устройством загружается драйвер устройства и создается объект драйвера (**Driver Object**), в котором представлены свойства устройства и даны указатели на реализованный в нем функции для обработки запросов ввода-вывода.

1.4.3. Трансляция виртуального адреса в физический

Определения

Page Table – структура, позволяющая производить трансляцию виртуальных адресов в физические.

Page Table Entry (PTE) – элементы данной структуры, содержащие PFN и атрибуты физической страницы.

Page Frame Number (PFN) – номер элемента базы данных PFN, которая содержит описания всех физических страниц в системе. [1, 2].

Структуры трансляции адресов в x-64 системах.

Пусть есть виртуальный адрес, он состоит из указателей на каталоги страниц, на карту страниц, на таблицы страниц, указатель на PTE (см. пункт 1.4. "Обзор литературы. Определения".) внутри таблицы страниц и на байт внутри страниц. См. Рис. 5.

Для перевода виртуального адреса в физический, необходимо пройти по всем каталогам, находим нужный PTE, из которого получаем информацию о том, какая же физическая страница, описываемая в базе данных PFN, содержит нужную информацию [1, 2].

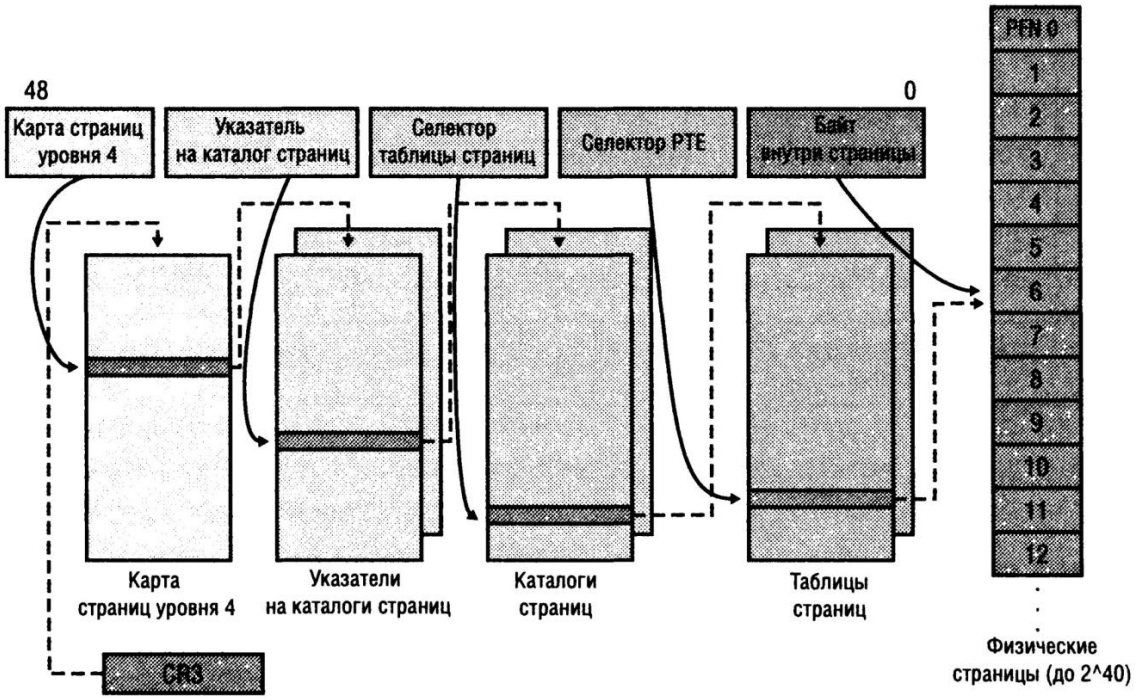


Рис. 5. Трансляция виртуального адреса в физический на платформа x64

Структура PTE (Page Table Entry)

Структура Page Table Entry нуждается в более детальном рассмотрении.

Она состоит из двух основных частей, первая это номер PFN, то есть номер элемента в базе данных PFN, который отвечает за данную физическую страницу, и некоторых атрибутов, из которых нам важны только два - это Writable и Copy On Write [1, 2]. См. Рис. 6.

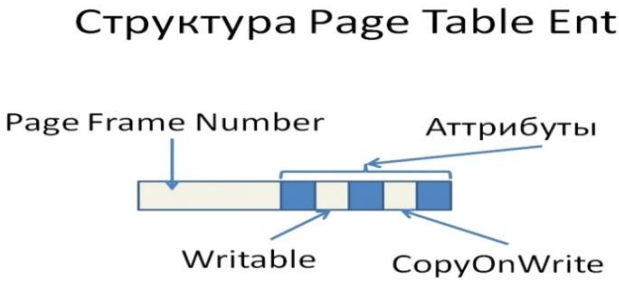


Рис. 6. Структура Page Table Entry.

1.4.4. Уровни прерываний в ОС Windows

Прерывания и исключения - ситуации, в которых прерывается нормальный поток выполнения кода процессором. Прерывание (interrupt) - асинхронное событие, например, генерируемое внешним (по отношению к процессору) устройством. Оно может произойти в любой момент независимо от текущих команд, выполняемых процессором. Исключение (exception) - синхронное событие, результат выполнения определенной команды. Например, ошибки доступа или деление на нуль.

Уровни запросов программных прерываний (interrupt request levels, IRQL) необходимы для реализации приоритетов прерываний. Чем выше номер уровня, тем приоритетнее прерывание. Прерывания обслуживаются в порядке их приоритета, и прерывания с более высоким IRQL вытесняют обработку прерываний с меньшим [1].

На нулевом (PASSIVE LEVEL) уровне IRQL работают пользовательские процессы и часть кода операционной системы. Программа, работающая на этом уровне, может быть вытеснена почти любым событием, случившимся в системе. Большинство процедур режима ядра старается удерживать IRQL уровень процессора как можно более низким. IRQL уровни 1 (APC LEVEL) и 2 (DISPATCH LEVEL) предназначены для так называемых программных (в терминологии Microsoft) прерываний соответственно: асинхронный вызов процедуры - APC (asynchronous procedure call) и отложенный вызов процедуры - DPC (deferred procedure call). Если ядро принимает решение выполнить некоторую системную процедуру, но нет необходимости делать это немедленно, оно ставит ее в очередь DPC и генерирует DPC прерывание. Когда IRQL процессора станет достаточно низким, эта процедура выполняется. IRQL уровни 3-26 относятся к обычным прерываниям от устройств. [1]

1.4.5. Механизм Copy On Write

Определение:

Копирование при записи (Copy On Write) – оптимизация, используемая диспетчером памяти для экономии физической памяти.

Главная идея: при копировании областей данных создавать реальную копию только когда ОС обращается к этим данным с целью записи. Таким образом, это один из примеров отложенных вычислений (lazy evaluation) [1, 5].

Существующие методы использования:

Copy On Write в Linux – используется, например, при вызове fork() или при работе KSM.

Copy On Write в Windows для Paged Memory – используется менеджером памяти.

Но в Windows нет Copy On Write для NonPaged Memory - для неподкачиваемого пула памяти. Это связано с тем, при высоких уровнях прерываний (IRQL) нельзя выделять страницы.

2. Решение

В данной создается метод: алгоритм и его рабочая реализация, которые заменяют в памяти несколько идентичных наборов страниц кода драйвера одним единственным экземпляром.

2. 1. Предлагаемый метод решения

Предлагаемый метод решения состоит из следующих ключевых пунктов:

- Обнаружение драйверов и поиск страниц, пригодных для «слияния»;
- Создание общего набора физических страниц;
- Подмена страниц драйвера страницами из общего набора;
- Возвращение неиспользуемых страниц в систему;
- При попытке записи на страницу создание ее копии, доступной только этому драйверу (механизм COW);
- При отгрузке корректное возвращение всех занимаемых ресурсов.

Каждый из этих этапов будет более подробно описан в пункте 2.1.2. "Жизненный цикл драйвера".

2. 1.1. Плюсы и минусы относительно других методов

Не требуется сканирование памяти. В данной системе можно заранее определить обслуживаемые имена драйверов и типы страниц. Так как известно имя драйвера, с которым предстоит работа, можно не искать его страницы, сканируя всю память системы, а отслеживать момент его загрузки [21, 23].

2.1.2. Жизненный цикл драйвера

В этом пункте более подробно рассматривается предлагаемый метод решения задачи. Он иллюстрируется жизненным циклом одного драйвера. Сначала описывается та ситуация, которая была в системе изначально, а после показывается, как будет работать система в при применении разработанного метода.

2.1.2.1. Начальная ситуация.

Было раньше.

Раньше, в системе была конкуренция за ресурсы, это видно на Рис.1. Пусть существуют два драйвера в системе: драйвер1 и драйвер2, каждого из которых есть набор используемых им страниц. Однако, свободной памяти в системе не хватает сразу на два драйвера и возникают проблемы.

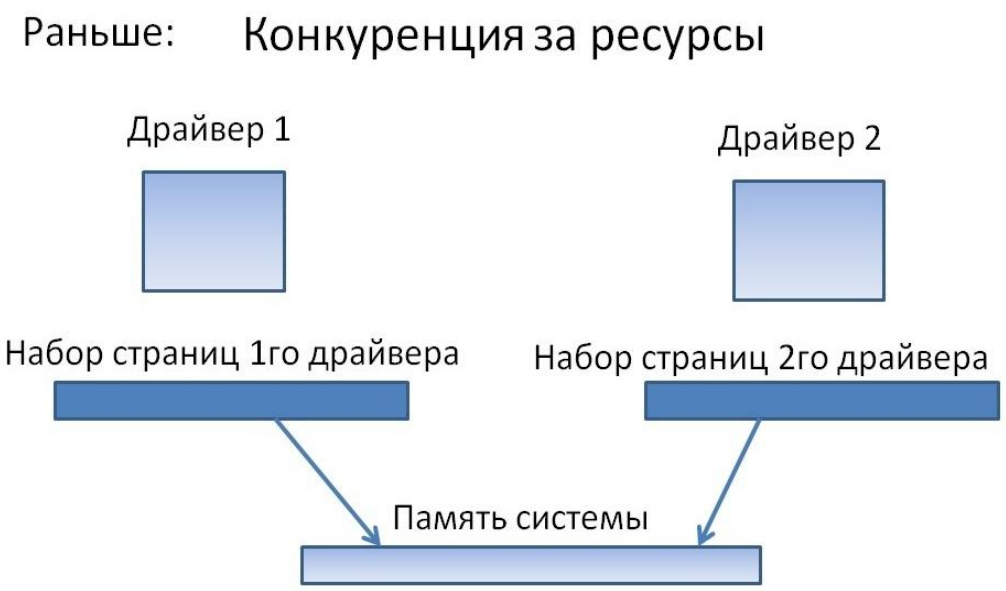


Рис1. Было раньше.

2.1.2.2. Начало работы

Рассмотрим процесс загрузки и начала работы нового экземпляра драйвера в случае дедупликации.

Пусть в системе так же как и в предыдущем случае существуют 2 драйвера. У каждого из них есть свой набор страниц. Теперь же, эти два драйвера перестают ссылаться на свои наборы страниц, и начинают ссылаться на общий набор страниц, который идентичен каждому из их наборов страниц. Этот общий - "нулевой" набор страниц создается при загрузке в систему первого драйвера, после того как происходит сканирование и выбор подходящий его страниц.

Таким образом, сначала нужно объединить наборы страниц. См. Рис. 2.

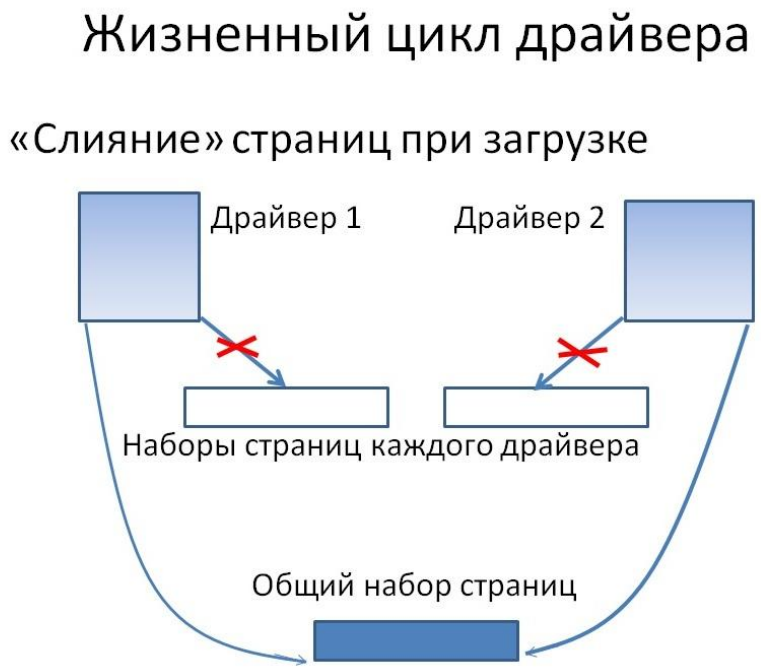


Рис2. Слияние страниц при загрузке.

Таким образом, драйвер перестает ссылаться на свои собственные страницы, которые возвращаются в систему, и начинает ссылаться на другие, представленные страницами из общего набора. То есть происходит подмена страниц. Более подробно механизм подмены страниц драйвера будет описан в пункте про Реализацию и Копирование при записи. Однако, уже здесь, следует уточнить, что речь идет о физических страницах памяти, а не об из виртуальных. Виртуальные адреса страниц драйвера не изменяются. При подмене страниц нарушается связь между виртуальными адресами страниц драйвера и связанными с ними физическими страницами. Эти физические страницы возвращаются позже в систему, а виртуальные адреса начинают ссылаться на новые физические страницы. Таким образом, сам драйвер не замечает произошедшую подмену.

2.1.2.3. Работа системы

Рассмотрим, как теперь происходит работа с драйвером.

Поскольку драйвер не замечает произошедшую подмену страниц, он может свободно исполнять или читать их содержимое. Однако, поскольку теперь все экземпляры драйвера ссылаются на один и тот же набор физических страниц, писать в эти страницы

нельзя. Потому что, запись сделанная одним экземпляром повлияет на работу всех остальных, что может привести к непредсказуемым последствиям.

Следовательно, необходимо запретить драйверам писать в страницы из общего набора. Поскольку, иногда им все-таки может понадобиться совершить запись в свои страницы кода, необходимо использовать механизм копирования при записи.

Рассмотрим на примере, см. Рис. 3. Итак, в системе находятся 2 драйвера, которые ссылаются на общий набор физических страниц. Когда драйвер2 хочет произвести запись в какую-то из своих страниц, эта виртуальная страница, то есть ее виртуальный адрес, перестает ссылаться на физическую страницу из общего набора. Вместе этого для этого драйвера создается отдельная копия этой страницы, то есть новая физическая страница с идентичным содержимым, доступная только этому экземпляру драйвера. Остальные же экземпляры не замечают изменений.

В процессе работы многие из экземпляров могут пытаться записать что-то свои страницы. В результате этого им будут выделяться отдельные копии. В итоге, в системе будет находиться один общий набор страниц для всех драйверов, и еще наборы страниц, "свои" для каждого драйвера - наборы тех страниц, в которые он совершил запись.

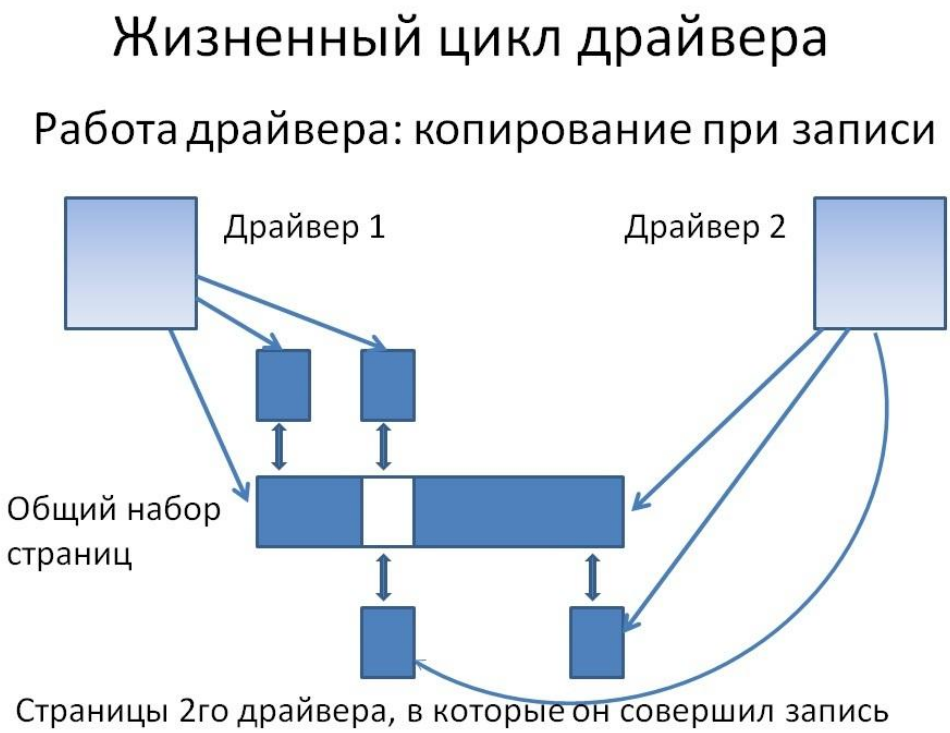


Рис3. "Отделение" страниц от общего набора при записи.

2.1.2.4. Завершение работы одного из драйверов

Очень важный момент это отгрузка драйвера. Ведь, если в данной ситуации ничего не сделать, то когда отгрузится первый драйвер, упадут все остальные (вся система), так как их страниц будут выгружены. Поэтому необходимо отдельно проработать отгрузку одного из драйверов.

Вот как предлагается это делать. Рассмотрим ситуацию, когда 2 драйвера в системе, ссылаются на общий набор страниц. См. Рис 4. Допустим, один из них хочет отгрузиться - происходит "разъединение" страниц при отгрузке. Это значит, что первый драйвер, который хотят отгрузить больше не будет ссылаться на общий набор страниц, он будет ссылаться на другой набор страниц, то есть наборы разъединяются. и он может быть спокойно отгружен.

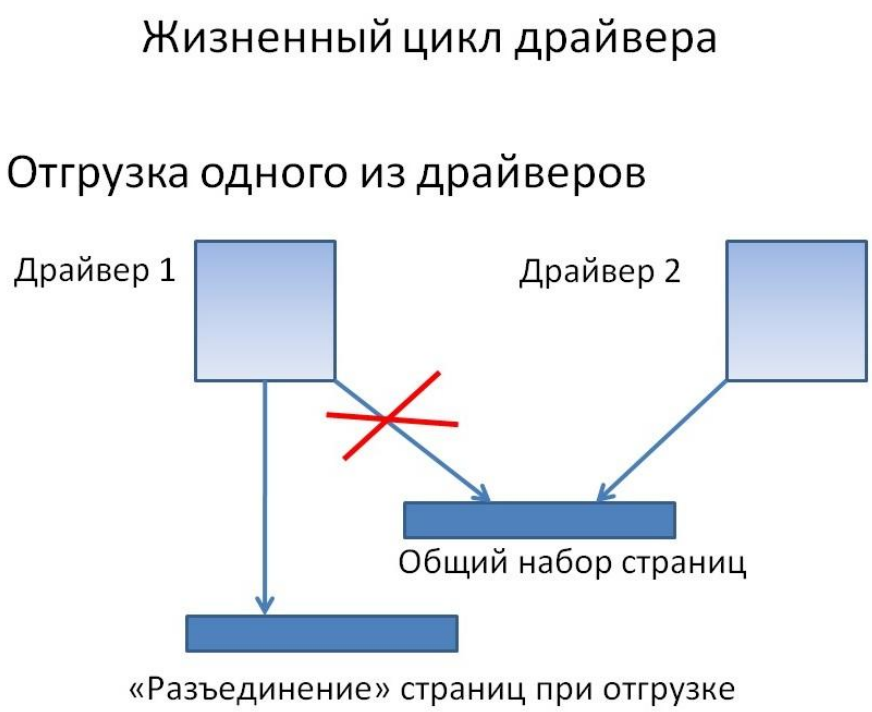


Рис.4. Отгрузка одного из драйверов.

2.1.2.5. Необходимость использования нулевого набора

Обоснование необходимости создания общего набора страниц, не принадлежащего ни одному из драйверов.

Именно в момент отгрузки одного из драйверов становится ясно, зачем было нужно выделять дополнительно память под общий набор, и почему он не создавался на основе набора страниц какого-либо одного из драйверов.

Это связано с тем, что если сделать его, например, на основе набора страниц первого драйвер, то этот драйвер можно будет отгружать лишь последним, (а кроме того, возникнут проблемы, если он захочет что-то записать в свои страницы). Или же, если он отгрузится придется перемапировать ВСЕ остальные драйверы на какой-то из оставшихся. И так каждый раз при отгрузке такого "главного" драйвера. (Опять же остается проблема, если у этого "следующего" часть страниц уже с записями. В этом случае можно потерять неизмененные страницы.)

А в случае с никому не принадлежащим общим набором страниц какой бы из драйверов мы не попытались отгрузить, это не повлияет на работу остальных.

2.1.2.6. Итоговый алгоритм решения

Теперь, когда рассмотрены основополагающие моменты решения, можно представить итоговый алгоритм решения данной задачи. Более подробное описание каждого из приведенных ниже пунктов будет в пункте 2.2. "Реализация".

Алгоритм:

- Отслеживание загрузки и отгрузки каждого экземпляра драйвера.
- Загрузка первого экземпляра драйвера:
  - Анализ секций исполняемого файла, поиск «executable» страниц.
  - Создание общего набора страниц.
  - Встройка в обработчик исключения Page Fault.
- Загрузка второго и последующих экземпляров:

- Проверка идентичности экземпляров
- Подмена страниц драйвера страницами общего набора.
- Возвращение в систему неиспользуемых страниц драйверов.
- Writable страницы сделать Copy On Write.

• Работа системы:

- При попытке записи - исключение Page Fault и «разъединение» страниц.

• Выгрузка одного из экземпляров:

- «Разъединение» страниц.

Итак, вот как это работает:

1) Отслеживание загрузки и отгрузки каждого экземпляра драйвера.

2) Загрузка первого экземпляра драйвера:

Во время загрузки первого экземпляра драйвера, происходит:

Анализ секций исполняемого файла, поиск «executable» страниц. То есть, страниц кода. Это связано с тем, что все-таки операции записи в страницы кода встречаются гораздо реже, чем операции записи в страницы с данными; а значит, данный механизм слияния страниц может быть полезен потому, что, если каждый драйвер будет писать в свои страниц данных, например, различные значения, то получится, что нам просто нечего объединять.

Создание общего набора страниц. Как было уже сказано выше, это набор страниц должен создаваться именно отдельно, сам по себе, и не принадлежать ни одному из драйверов, чтобы не было проблем с отгрузкой.

Встройка в обработчик исключения Page Fault. Это необходимо для того, чтобы отслеживать попытки записи в Copy On Write страницы и правильно их обрабатывать. Изначально в Windows нет такого механизма для пула неподкачиваемой памяти. Однако, он был создан в процессе работы.

На данном этапе мы получаем уже практически готовую работающую систему.

3) Во время загрузки второго и последующих экземпляров, происходит:

Проверка идентичности экземпляров. Это нужно для того, чтобы ни в коем случае не попал драйвер, например, с этим же названием, но другой версии, который будет иметь другие страницы кода.

Подмена страниц драйвера страницами общего набора. В пункте про реализацию будет пояснено, как это сделано. На данном этапе можно считать, что страницы драйвера подменяются страницами из общего набора, и драйвер ссылается на страницы из общего набора, а про свои страницы он "забывает", хотя они все еще находятся в системе.

Возвращение в систему неиспользуемых страниц драйверов.

Кроме того, те страницы из общего набора, которые изначально были помечены Writable , необходимо сделать Copy On Write.

4) Когда все загружено, объединено и готово к работе - рассмотрим работу системы:

При попытке записи происходит исключение Page Fault, (так как, запись в страницы помеченные Copy On Write запрещена,) и, поскольку мы встроились в обработчик исключения Page Fault и сделали там свой обработчик, происходит «разъединение» страниц. "Разъединение=отделение" от общего набора той единственной страницы, в которую велась запись. Драйвер перестает ссылать на такую страниц из общего набора, для него создается копия этой страниц, уже доступная для записи. Дальше уже исключения не происходят, потому что это его страница.

Система работает таким образом, некоторые страницы разъединяются, некоторые нет. У каждого драйвер есть свой набор "разъединенных" страниц, а так же он обращается ко всем остальным своим страниц, которые находятся в общем наборе.

5) При выгрузке одного из экземпляров, происходит:

«Разъединение» наборов страниц. Это означает, что те страницы драйвера, которые до сих пор ссылались на страницы из общего набора, "отделяются" от них, и больше эти виртуальные адреса на них не ссылаются. Теперь, отгрузка этого экземпляра драйвера обычным системным способом не повлияет на все остальные загруженные экземпляры.

Для корректной отгрузки драйвера, этого пока недостаточно, ведь виртуальные адреса его страниц вообще ни на что не ссылаются. Можно было бы выделить ему новые

страницы, специально для отгрузки. Но это бы привело к слишком большой нагрузке на подсистему памяти (а возможно, и дефициту страниц) при массовой отгрузке таких драйверов, например, при резком выключении. В связи с этим, в данной работе этот способ не используется. Вместо него используется способ аналогичный работе с Discardable секциями драйверов. Страницы (их виртуальные адреса) драйвера, ранее ссылавшиеся на физические страницы из общего набора помечаются "уже отгруженными", и после этого, драйвер может быть корректно отгружен системными средствами самой ОС. Более подробно см. в пункте про Реализацию.

2.2. Реализация

Этот пункт посвящен реализации представленного выше алгоритма. Вначале будет уточнено, для каких именно драйверов и страниц производилась реализация, а после изложены основные моменты реализации, не касающиеся работы механизма Copy On Write. Имплементация Copy On Write была вынесена в отдельный пункт, поскольку является основополагающей частью работы и может иметь более широкое применение, чем дедупликация кода драйверов в целом.

2.2.1. Ограничения

Прежде чем рассматривать реализацию этого алгоритма, уточним, с какими именно типами драйверов и с какими их страницами будем работать. А так же опишем основные ограничения, с которыми пришлось столкнуться в процессе реализации этого алгоритма.

2.2.1.1. Рассматриваемые типы драйверов

Ядро или User Space

Следует уточнить, что исследовалась только ядерная часть.

Сессионные или Не сессионные драйверы

В данной работе, не рассматриваются сессионные драйверы. Так как они не нуждаются в дедупликации кода, поскольку для них механизм Copy On Write изначально используется менеджером памяти. (См. пункт 1.4. Обзор литературы.)

2.2.1.2. Рассматриваемые типы страниц

В данной работе речь идет только о целиковых страницах, а не о отдельных записях на страницу.

Секции

Прежде всего необходимо отметить, что в данном пункте речь идет скорее о секциях, так как загрузчик исполняемых образов в Windows во время загрузки драйвера в память устанавливает атрибуты его страниц в соответствии с теми, которые имеет секция, содержащая эту страницу. Этот момент будет далее рассмотрен подробнее в пункте 2.2.2. про поиск подходящих для слияния страниц в драйвере.

Code

В данной работе рассматриваются только страницы кода драйверов, так как очень редко драйвер изменяет свой код, и эти страницы имеет смысл сливать в одну, поскольку они совпадают с большой вероятностью. Страницы данных было решено не задействовать, так как данные часто подвергаются изменениям, и, следовательно, не приносят выгоды при данной оптимизации.

Pageable и NonPageable

Как было сказано во введении (См. пункт 1.4. Обзор литературы), в ОС Windows существуют два пула памяти подкачиваемый и неподкачиваемый. Проблема дедупликации кода драйверов особенно остро стоит для неподкачиваемого. Страницы кода драйверов из неподкачиваемого пула всегда находятся в оперативной памяти, а значит, занимают драгоценное место, и их нельзя заменить на что-то более полезное, пока эти драйверы не выгрузятся. Чем больше кол-во одинаковых драйверов, тем большее кол-во памяти заполнено по сути не нужными экземплярами кода, то есть тратится зря. Таким образом, имея конечный размер и будучи заполненной по сути не нужными экземплярами кода, эта область памяти начинает накладывать ограничения по памяти на работу системы. В случае с подкачиваемым пулом такой проблемы нет, и большое кол-во одинаковых страниц влияет только на частоту подкачки и сброса страниц на диск.

В связи с этим, было решено в первую очередь решить задачу дедупликации для неподкачиваемого пула памяти. Поскольку аналогичная задача для подкачиваемого пула очень сложна, она в данной работе не рассматривается.

Итак, алгоритм дедупликации кода идентичных драйверов был реализован для пула неподкачиваемой памяти. Это значит, что работа механизма Copy On Write ведется со страницами из невыгружаемого пула памяти. Однако, чтобы иметь возможность использовать данный алгоритм и для выгружаемых страниц, такие страницы делаются невыгружаемыми (NonPageable), то есть блокируются в памяти. Для этого в драйвере запрещается отгрузка страниц на диск (то есть свопинг. См. пункт 1.4. Обзор литературы). А после, с ними можно работать так же, как с остальными.

Таким образом, в системе в место большого кол-ва наборов Pageable страниц существует один набор в неподкачиваемой памяти.

Такой подход при большом кол-ве драйверов значительно снижает нагрузку на подсистему подкачки страниц, и, следовательно, хорошо влияет на производительность.

Кроме того, он эффективен и по памяти. Рассмотрим ситуацию, когда несколько одинаковых наборов страниц находятся в подкачиваемом пуле памяти, и с каждым из них работает свой экземпляр драйвера. Тогда, усредненное по времени кол-во страниц, находящихся в памяти, будет зависеть от кол-ва драйверов и интенсивности их работы. Однако, оно с очень большой вероятностью будет больше, чем страниц в одном наборе. Что означает, что объединение и перенос страниц в неподкачиваемый пул памяти были выгодны, ведь в этом случае, в системе в любой момент времени ровно один набор (для read-only страниц). Такой подход хорошо подходит для драйверов, которые часто обращаются к своим страницам. Однако, он не эффективен (а даже, наоборот, вреден) для

малоактивных драйверов, которые длительное время бездействуют, и все страницы которых как правило большую часть времени выгружены на диск (то есть среднее по времени кол-во страниц в памяти близко к 0).

В пункте 2.3. "Математическая модель" подробнее говорится о целесообразности включения pageable секций драйвера в рассмотрение. Однако, решение должно приниматься в каждом случае отдельно и основываться на характере поведения драйвера и кол-ве драйверов в системе. Например, в зависимости от типа драйвера, и от его "активности" его Pageable страницы могут быть как все подгружены в память, так и все отгружены на диск [3].

Writable/Nonwritable

Объединение страниц в одну особенно эффективно для nonwritable страниц. Однако, было решено так же объединять и Writable страницы. Это связано с тем, что во время компиляции драйвера секция кода помечается как Writable, даже если запись происходит всего в несколько страниц из этой секции. Таким образом в процессе работы разъединяться будут не все страницы помеченные, как Writable, а лишь небольшая их часть. (Подробнее см. в пункте 2.3."Математическая модель" и 2.4."Тестирование"). Следовательно, объединение остается эффективным и выгодным с точки зрения кол-ва занимаемой памяти.

Large

В данной работе Large Pages не рассматривались.

Поскольку в данной работе речь идет только о целиковых страницах, а не о отдельных записях на страницу, то Large Pages не рассматривались, из-за того, что практически невозможно найти две одинаковые большие страницы. Хотя конечно, существует алгоритм работы и с ними, когда большая страница делится на несколько маленьких, которые уже объединяются. и т.д.

2.2.1.3. Copy On Write в Windows

Как уже было сказано выше, в данной работе рассматривается механизм дедупликации страниц, с помощью копирования при записи, только в неподкачиваемом пуле памяти. Однако, в нем в Windows не существует механизма Copy On Write. Данный механизм, в ОС Windows работает только для неподкачиваемой памяти. Причина состоит в том, что страницы неподкачиваемой памяти могут вызываться на высоких IRQL (уровнях прерываний, см. 1.4. Обзор литературы), где нельзя выделять страницы памяти. А механизм копирования при записи подразумевает выделение новой страницы - копии для передачи вызвавшему потоку.

В данном случае, такой механизм необходим хотя бы для read-only страниц, в которые не будут производиться записи, и, значит, не понадобятся дополнительные страницы. Следовательно, он был создан. А так же модифицирован так, чтобы мог обслуживать и writable страницы тоже.

2.2.1.4. Итог

Итог:

- драйверы - не сессионные
- страницы - целиком, только код, nonpage и page; writable и nonwritable.

Для реализации всего этого, пришлось имплементировать Copy On Write в NonPageable памяти, вклиниться в обработчик Page Fault, создать дополнительный резервный буфер страниц для высоких IRQL и т.д. Обо всем этом и пойдет речь в следующих двух пунктах.

2.2.2. Основные этапы реализации

В данном пункте рассматриваются тонкости реализации предложенного метода в ядре Windows.

2.2.2.1. Отслеживание загрузки и отгрузки каждого экземпляра драйвера

Отслеживание загрузки

Отслеживание загрузки реализуется с помощью системных средств. В ОС Windows существует системный вызов PsSetLoadImageNotifyRoutine, устанавливающий функцию NotifyRoutine, которая срабатывает, когда в систему грузится новый драйвер. На этапе загрузки нового драйвера, когда страницы с его кодом уже загружены в память, но перед передачей управления его функции инициализации DriverInit, вызывается наш обработчик (NotifyRoutine), который может выбрать с драйверы с определенным именем и начать работать с ними, в рамках дедупликации страниц их кода.

Отслеживание отгрузки

Отслеживание отгрузки драйвера является гораздо более сложной задачей в силу отсутствия аналогичной функции. Существует несколько способов поймать этом момент. Из-за своей простоты и наименьшего количества изменений в коде ядра и драйверов был выбран следующий.

Поскольку при отгрузке драйверов в ОС Windows сначала вызывается установленная драйвером функция DriverUnload, и только потом подсистема памяти начинает выгружать его страницы из памяти (а именно до этого момента нужно успеть "разъединить" все ранее "объединенные" страницы), то нужно поймать момент вызова функции DriverUnload драйвера и "вклиниться" сразу после нее.

В тот момент, когда драйвер только что был загружен в память, но еще не приступил к инициализации (см. выше), происходит "встройка" в его код. То есть внутри функции NotifyRoutine нужно "встроиться" в функию DriverInit инициализации драйвера. То есть снять запрет на запись в страницы драйвера; в первые несколько байт функии DriverInit инициализации драйвера поставить безусловный переход на мою новую функцию, MyNewDrEntry, которая будучи вызванной, вернет DriverInit в ее исходное состояние, вызовет ее, и получит от нее в соответствующем поле структуры DriverObject адрес функции DriverUnload драйвера, который, прежде чем передать эту структуру

менеджеру памяти, она подменит на адрес моей новой функции MyNewDrUnload, которая теперь будет вызываться вместо страной функции DriverUnload драйвера.

Таким образом, способ состоит в следующем: необходимо заменить родную функцию отгрузки драйвера DriverUnload на мою новую функцию MyNewDrUnload, которая теперь будет вызываться вместо нее. Для этого необходимо "встроиться" в функию DriverInit инициализации драйвера и подменить значение в соответствующем поле структуры DriverObject.

Теперь, при отгрузки драйвера вместо DriverUnload запустится MyNewDrUnload, которая сначала позволит отработать родной функции отгрузки драйвера, а потом выполнить "разделение" страниц.

2.2.2.2. Анализ секций исполняемого файла

Тут речь идет о секциях, так как загрузчик исполняемых образов в Windows во время загрузки драйвера в память устанавливает атрибуты его страниц в соответствии с теми, которые имеет секция, содержащая эту страницу.

Когда в систему грузится первый экземпляр драйвера, необходимо найти среди всех его страниц страницы, подходящие для слияния, то есть страницы кода - исполняемые.

Для этого в образе драйвера, загруженном в память, нужно найти все секции, с подходящими свойствами (атрибутами), и "запомнить"(например, записать в массив) все страницы из этих секций. Именно страницы из массива и будут потом "объединяться" с аналогичными страницами других экземпляров.

2.2.2.3. Создание общего набора страниц

Необходимость создания нулевого набора описана в пункте 2.1.2.5. Здесь же рассмотрим, как он реализуется. На этом этапе уже известно количество и содержимое всех страниц, которые подлежат объединению.

Выделение страниц

Сначала необходимо выделить нужное количество страниц. Документированного способа сделать это нет. Поэтому было произведено детальное исследование работы подсистемы памяти в ядре ОС Windows, с упором на механизмы выделения страниц при загрузке драйвера в систему. Был обнаружен набор функций, решающий эту задачу: MiReserveSystemPtes - для того, чтобы зарезервировать системные PTE (то есть получить

непрерывный набор адресов); MiAllocatePfn - выделить физическую страницу для данного PTE; MiGetVirtualAddressMappedByPte - узнать, какой виртуальный адрес соответствует данной странице.

Атрибуты и содержимое страниц

После этого для каждой страницы записать в нее соответствующее содержимое. А так же выставить атрибуты и права доступа так, чтобы они были аналогичны тем, которые имеют страницы драйвера изначально. Кроме того writable страницы нужно превратить в Copy on Write.

Удаление нулевого набора

В конце работы, после того как все экземпляры были выгружены, необходимо удалить из памяти страницы нулевого набора. Эта задача так же не решается документированными средствами ОС Windows. Было произведено исследование, в ходе которого были найдены необходимые функции: MiDeleteSystemPageableVm - удаляет область в подкачиваемой памяти (pageable system address space (paged pool or driver pageable sections)); MiReleaseSystemPtes - освобождает PTE в неподкачиваемом пуле памяти (non paged portion of system space).

Эти же функции могут быть использованы и для возвращения в систему "освободженных" страниц, то есть тех, с которыми драйвер больше не работает из-за того, что они были подменены страницами из общего набора.

2.2.2.4. Встройка в обработчик прерывания Page Fault

Первым делом, необходимо найти, по какому адресу ядро загружено в память - документированная функция ZwQuerySystemInformation.

Работа обработчика прерывания выглядит так: сначала вызывается функция KiTrap0E, которая выполняет некоторые проверки и вызывает функцию MmAccessFault, которая обрабатывает прерывание. Т.к. это 64 битная система, нельзя просто подменить вызов функции MmAccessFault - на вызов своей новой функции. Но можно найти "пустое" пространство в коде ядра (несколько байт, заполненных нулями в следствие выравнивания на страницу), и туда записать небольшой "stub", то есть код, который будет вызывать мой обработчик. И подменить вызов функции MmAccessFault - на передачу управления в этот "stub".

При попытке записи - исключение Page Fault и «разъединение» страниц.

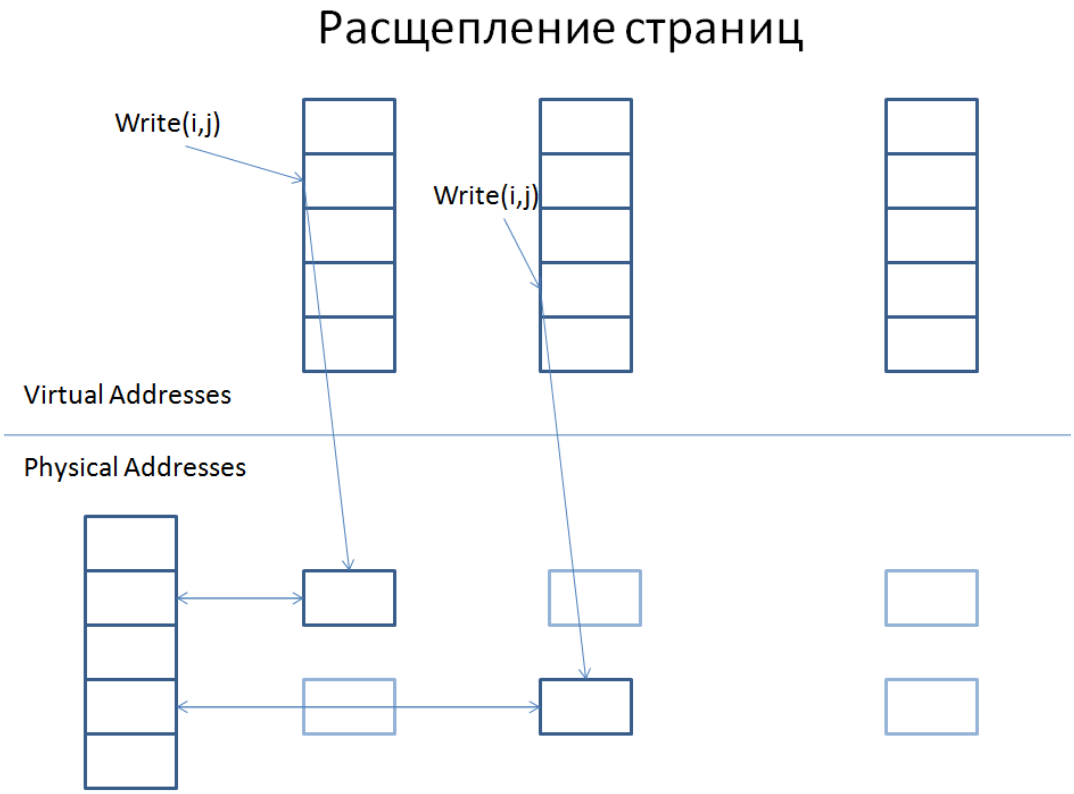


Рис. 9.

Таким образом, теперь, при возникновении исключения ошибки страницы (Page Fault) будет вызываться мой новый обработчик, который должен будет "разъединять" ранее "объединенные" страницы при попытки записи в них (если, конечно, изначально драйвер имел на это право).

Внутри новой функции обработчика

Внутри новой функции обработчика необходимо поставить поверку, и если адрес, на котором произошла ошибка страницы не является адресом (виртуальным) какой-либо из "объединенных" страниц, то вызывать старую функцию MmAccessFault. А если является и ошибка произошла из-за того, что один из экземпляров драйвера захотел что-то записать в свою страницу (которая была уже объединена с другими и помечена копируемой при записи (Copy on Write)), то необходимо обработать эту ситуацию.

Выделить новую физическую страницу; записать на нее содержимое соответствующей страницы из нулевого набора; выставить для нее "правильные" атрибуты, и сделать ее доступной для записи; "разъединить" страницы, то есть сделать так, чтобы этот виртуальный адрес ссылался теперь на эту вновь предоставленную страницу, а не на страницу из нулевого набора. (Подробнее см. пункт 2.2.3. "Имплементация механизма копирования при записи".)

2.2.2.5. Проверка идентичности экземпляров

В тот момент, когда новый экземпляр грузится в систему, необходимо проверить его идентичность всем остальным, т.к. любое несоответствие в коде может привести к непредсказуемым последствиям. Основным критерием служит имя драйвера, его версия, однако, можно использовать и более сложные методы, например, как в KSM - подсчитывать функцию ..... и сравнивать значения, полученные для двух "одинаковых" страниц.

2.2.2.6. Подмена страниц драйвера страницами общего набора

Когда создан общий набор необходим подменить "родные" страницы драйвра на идентичные им страницы из общего набора страниц. Драйвер обращается к своим страницам по их виртуальным адресам. Дальше для перевода виртуального адреса в физический используется структура PTE (Page Table Enry, см. пункт 1.4. "Озор литературы"), которая содержит номер нужной страницы в базе данных PFN (Page Frame Number, см. пункт 1.4. "Озор литературы"). Нам необходимо сделать так, чтобы драйвер обращался к странице из общего набора, то есть виртуальный адрес страницы должен остаться старым, а вот номер физической страницы, на которую он ссылается должен измениться. Следовательно, необходимо изменить значение PTE, соответствующей этому виртуальному адресу. И прописать в него новый номер физической страницы и ее атрибуты. Подробнее, как это делается, см. пункт 2.2.3. "Имплементация копирования при записи".

2.2.2.7. Writable страницы сделать Copy On Write.

Для этого достаточно сбросить бит "Writable" и взвести бит "Copy On Write" в атрибутах в PTE страницы. (Подробнее см. пункт 1.4. "Обзор литературы" и 2.2.3. "Имплементация копирования при записи" про PTE и ее строение.)

2.2.2.8. Создание и работа с резервным буфером; подкачивающая нить

При "разъединении" страниц при записи необходимо предоставить новую физическую страницу, копию той страницы из обзего набора, на которую драйвер хотел что-то записать. Так как это происходит в обработчике прерывания Page Fault, то сделать это необходимо быстро. Однако, не всегда есть возможность выделить новую страницу, то есть запросить ее у системы. В случае работы на высоких уровнях прерываний (IRQL) сделать это невозможно, так как выставлена блокировка базы данных PFN.

В этом случае предлагается создать резервный буфер заранее выделенных страниц, и при необходимости (когда нельзя выделить новую) брать сттраницу из него.

Поскольку во время работы, буфер будет опустошаться, необходимо завести дополнительную нить, которая будет добавлять в него страницы. Механизм работы нити может быть различным, она может просыпаться раз в некоторое время, или начинать работать по какому-то событию. Она должна проверять количество страниц в резервном буфере, и довавлять в него новые взамен использованных.

Размер буфера и частота добавления новых страниц - очень важные параметры, так как, если буфер будет пуст в тот момент, когда потребуется страница, то зависнет по меньшей мере один экземпляр драйвера на одном из процессоров. Это не страшно, если подкачивающая нить на другом процессоре добавит страницу в буфер. Однако, в худшем случае может привести в зависанию всей системы.

С другой стороны, драйверы довольно редко изменяют собственный код (а именно это означает в одну из "объединенных" страниц), а значит, можно подобрать эти значения такими, что с одной стороны они будут гарантировать надежность, а с другой стороны все еще будут давать значительный выигрыш по количеству страниц. Следует отметить, что эти параметры разные для каждого драйвера. Однако, если встретится драйвер, который слишком часто изменяет собственный код, то имеет смысл не объединять его writable страницы вовсе.

В пункте 2.3. "Математическая модель" производится расчет минимального количества страниц, которое должно быть в резервном буфере, и описывается интесивность работы с ним. Несмотря на довольно точный расчет, этого объема может в какой-то ситуации не хватить, и на этот случай, у пользователя есть возможность изменять их в ручную.

2.2.2.9. Процесс "разъединения" страниц при отгрузке одного экземпляра драйвера

При разъединении страниц происходит следующее. У драйвера есть виртуальные адреса страниц в памяти, где находится его код. Каждому виртуальные адресу соответствует определенный физические адреса той, страниц, на который хранятся эти данные. Это механизм трансляции виртуального адреса в физический. Для трансляции используются PTE и PFN страницы (См. пункт 1.4. Обзор литературы). При выгрузке 1го экземпляра драйвера, виртуальные страниц перестают ссылаться на физические страницы, которые принадлежат общему набору. Windows существуют секции помеченные как INIT, то есть Discardable, которые выгружаются сразу же после загрузки драйвер, тем не менее, этот диапазон адресов, остается системе, хотя физические страниц уже отгружены, PTE,

соответствующие этим виртуальным адресам зануляются, и при отгрузке драйвера система понимает, что эти страницы уже были отгружены ранее. То есть они помечаются так, как будто страниц, связанные с ними уже были выгружены. Это действительно так, так как эти страниц были отгружены в систему еще на этапе загрузки драйвер. Этот же метод мы будем использовать здесь.

Таким образом, при разъединении страниц при выгрузке, не будет выделяться новая страница, но будут "зануляться" PTE соответствующих виртуальных страниц.

Завершение работы

В случае, когда происходит выгрузка моего управляющего драйвера, он должен сначала дождаться пока завершат свою работу все работающие экземпляры дедуплицируемого драйвера, так как иначе, из-за особенностей метода отслеживания момента их отгрузки, они просто не смогут быть выгружены без него.

2.2.3. Имплементация механизма копирования при записи (COW)

Механизм Copy On Write для неподкачиваемого (NonPaged Pool) пула памяти необходим, когда заходит речь о максимально полном использовании ресурсов.

2.2.3.1. Цель

В данном разделе основной целью является использование механизма Copy On Write в страницах NonPaged Memory. Но не при копировании страниц, а для объединения уже существующих в системе идентичных страниц в одну и экономии памяти, и далее при разъединении страниц при попытке записи.

2.2.3.2. Предлагаемый метод решения

Предлагаемый метод решения состоит в том, что если в системе есть несколько одинаковых страниц, необходимо:

- Объединить страницы в одну;
- Вернуть «освобожденную» страницу в систему;

При записи необходимо:

– Предоставить новую страницу тому потоку, который собирался выполнить запись;

– Разъединить страницы.

2.2.3.3. Трансляция виртуального адреса в физический

Про трансляцию виртуального адреса в физический см. пункт 1.4. "Обзор литературы".

Во-первых, следует уточнить различия и связи между виртуальными и физическими страницами в системе. Потому что в данной работе с виртуальными страниц не происходит ничего, а изменяются, подгружаются и отгружаются только физические.

2.2.3.4. Объединение страниц

Рассмотрим две идентичные страницы, находящиеся в пуле неподкачиваемых страниц.

Что было Раньше:

На Рис.7 есть 2 различные, но с идентичным одержимым виртуальные страницы, с разными PTE, и которые ссылаются на две различные, хоть и с идентичным одержимым, физические страниц. То есть в PTE одной из страниц записано PFN_1, а в PTE другой - PFN_2.

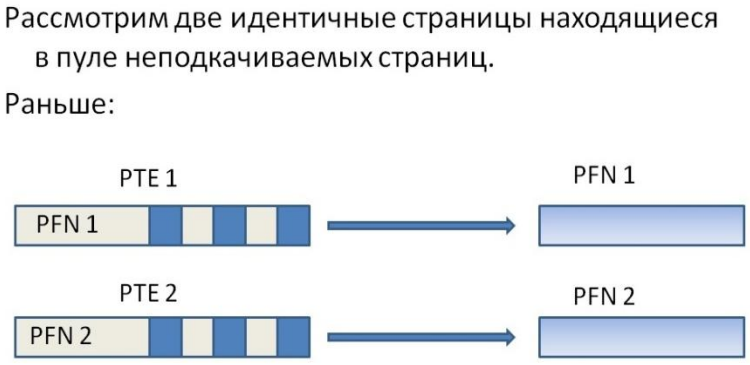


Рис. 7. Было раньше.

Объединение двух страниц, путем изменения их PTE

- 1-я страница из Writable становится Copy on Write;
- 2-я страница (так же) становится Copy on Write;

(Это только для тех страниц, которые были Writable. Если же она не была Writable, то ее атрибуты не изменяются вовсе.)

- PTE 2-й страницы начинает ссылаться на первую.

То есть в PTE 2й страницы прописывается PFN номер 1й физической страниц. См. Рис. 8.

- 1-я страница из Writable становится Copy on Write
- 2-я страница становится Copy on Write
- PTE 2-й страницы начинает ссылаться на первую

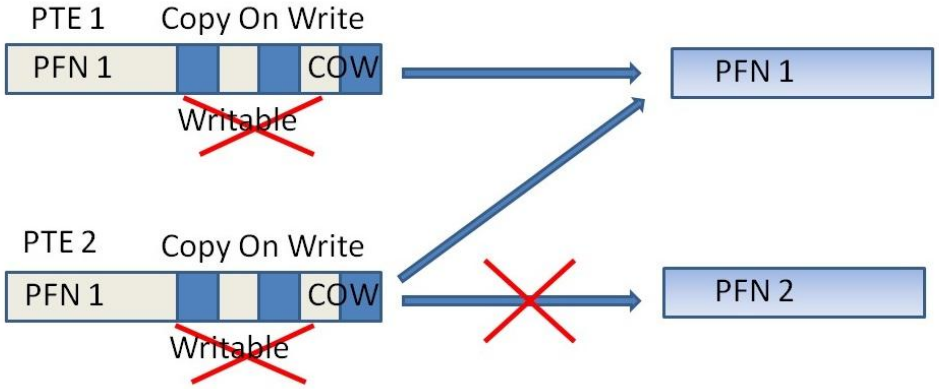


Рис. 8. Объединение двух страниц, путем изменения их PTE.

Возврат "освобожденной" страницы в систему

После этого, PTE_2 будет ссылаться на новую физическую страницу. PFN 2й страниц остается "забытым", то есть на эту физическую страницу больше никто не ссылается, и ее можно вернуть в систему. См. Рис. 9.

Это делается вызовом соответствующей функции (..mmdelete.. mmrelease..).

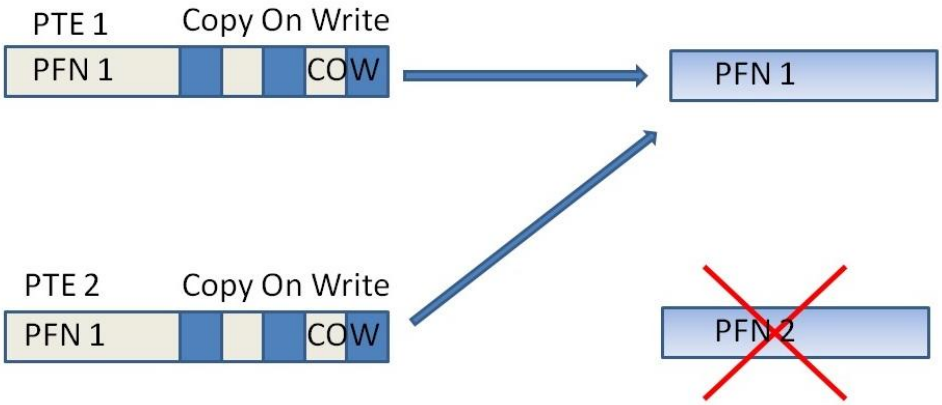


Рис. 9. Возврат в систему "освобожденной" страницы.

Следует отметить, что в элементе базы данных PFN есть поле, отвечающее за то, какая PTE была раньше у этой страницы, но это поле рассматривается диспетчером памяти, только в тот момент когда страниц была выгружена из памяти и ее пытаются

подгрузить обратно. Поскольку в данной работе рассматривается только неподкачиваемый пул, то содержимое этого поля не влияет в работу систему.

2.2.3.5. Разъединение страниц

Теперь система готова к работе и будет так работать до первой попытки записи на одну из страниц.

В случае попытки записи на одну из страниц происходит исключение ошибки страницы Page Fault, связанное с нарушением прав доступа. Поскольку нельзя писать в страницы помеченные как NonWritable и Copy On Write (то есть сброшен бит Writable в взведен бит Copy On Write).

Page Fault в случае запроса на запись

Когда поступит запрос на запись:

- Произойдет Page Fault
- Вызовется обработчик этого исключения
- Будет BSOD (Page Fault in NonPaged Memory) с извещением о том, что произошел Page Fault in NonPaged Memory (исключение ошибки страницы в неподкачиваемом пуле памяти).

=> Необходимо встроиться в обработчик Page Fault и там:

- Предоставить новую страницу
- Разъединить страницы, изменив их PTE так, чтобы виртуальный адрес, по которому пытались произвести запись, ссылался теперь на выделенную физическую страниц.

2.2.3.6. Обработка Page Fault

Рассмотрим обработку исключения Page Fault для рассматриваемых страниц более подробно.

2.2.3.6.1. Предоставление новой страницы

Прежде всего необходимо предоставить новую физическую страницу, копию общей, чтобы поток инициировавший запись, получил себе в пользование копию нужной ему страницы и мог совершить запись. См. Рис. 10.

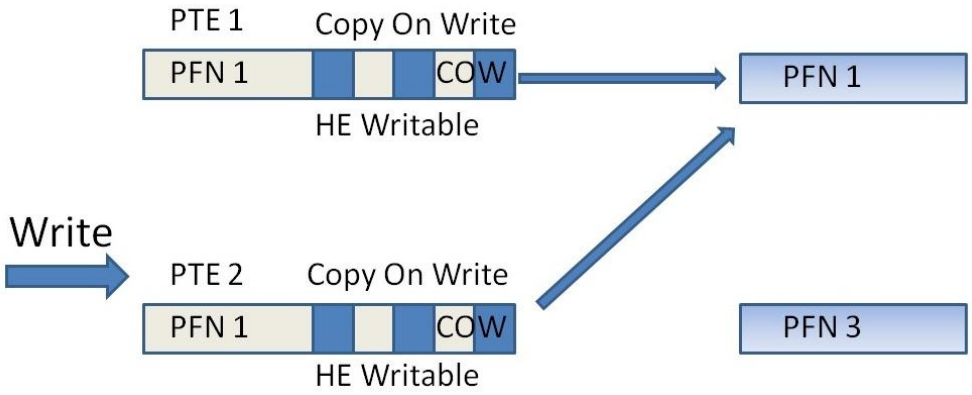


Рис. 10. Предоставление новой страницы.

2.2.3.6.2. Разъединение страниц , путем изменения их PTE

Итак, когда новая страница выделена, наступает черед "разделения" страниц. Происходит "перемапирование" PTE с одной страницы на другую с изменением атрибутов. См. Рис. 11.

- PTE 2-й страницы начинает ссылаться на предоставленную новую страницу
- 2-я страница становится Writable
- 1-я страница становится Writable, если на нее больше никто не ссылается. То есть если объединены были всего две страницы, или не становится, если на нее ссылается еще какая-нибудь страницы. Однако, что поскольку мы сами их на нее замапировали, то мы знаем точное кол-во ссылающихся на нее страниц.

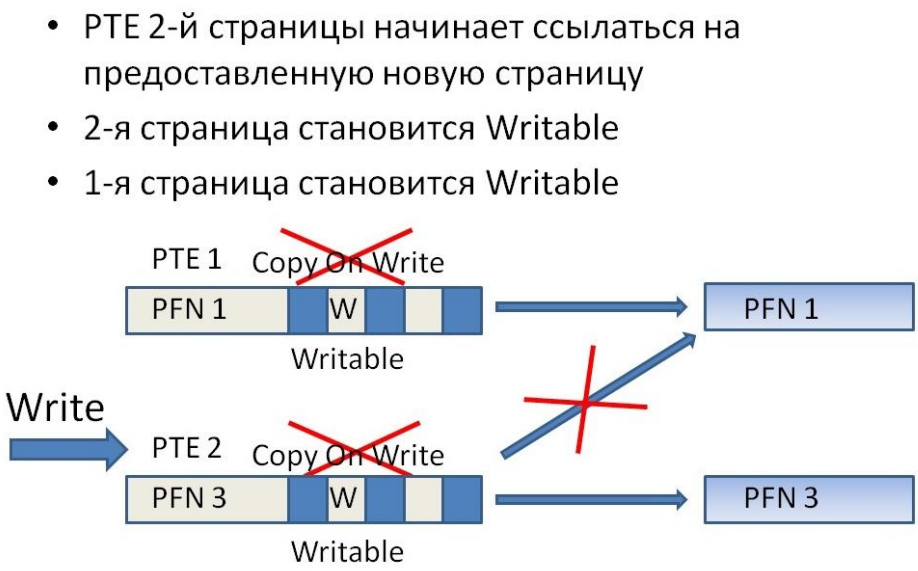


Рис. 11. Разъединение страниц, путем изменения их PTE.

2.2.3.7. COW и высокие IRQL и связанные с ними проблемы

При предоставление новой страницы необходимо упомянуть об уровнях прерываний. В зависимости от того, при каком IRQL работает система, изменяется поведение обработчика Page Fault:

- Низкие IRQL – новая страница запрашивается у системы;
- Высокие IRQL – новая страница берется из заранее подготовленного набора страниц, который пополняется так часто, насколько это возможно.

Размер этого заранее подготовленного набора страниц, называемого "резервным буфером страниц" определяется исходя из поведения драйвера и расчетов в математической модели. См. пункт "Математическая модель разъединения страниц".

2.3. Математическая модель работы

2.3.1. Основные положения

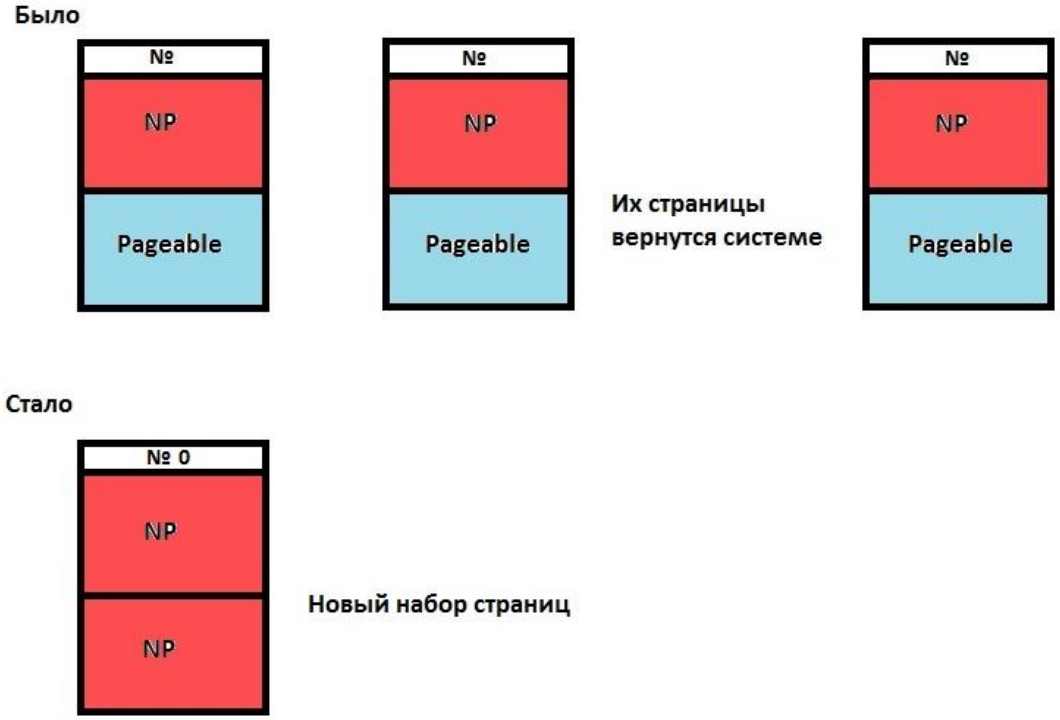
Поскольку целью работы является дедупликация исполняемого кода драйверов, то есть уменьшение количества памяти занимаемой этими страницами, то, во-первых, необходимо подсчитать:

- **Изменения кол-ва занимаемой памяти** (минимальные, максимальные и вероятностные).

2.3.1.1. Различные типы страниц памяти

Рассмотрим, какие типы страниц бывают в драйверах, и в какие типы они переходят потом в результате работы моей программы.

Было N одинаковых драйверов с одинаковым кодом. Теперь, создается новый набор страниц, полностью в невыгружаемой памяти, драйверы виртуальные страницы драйверов перемапируются на него, а их собственные страницы возвращаются в систему.

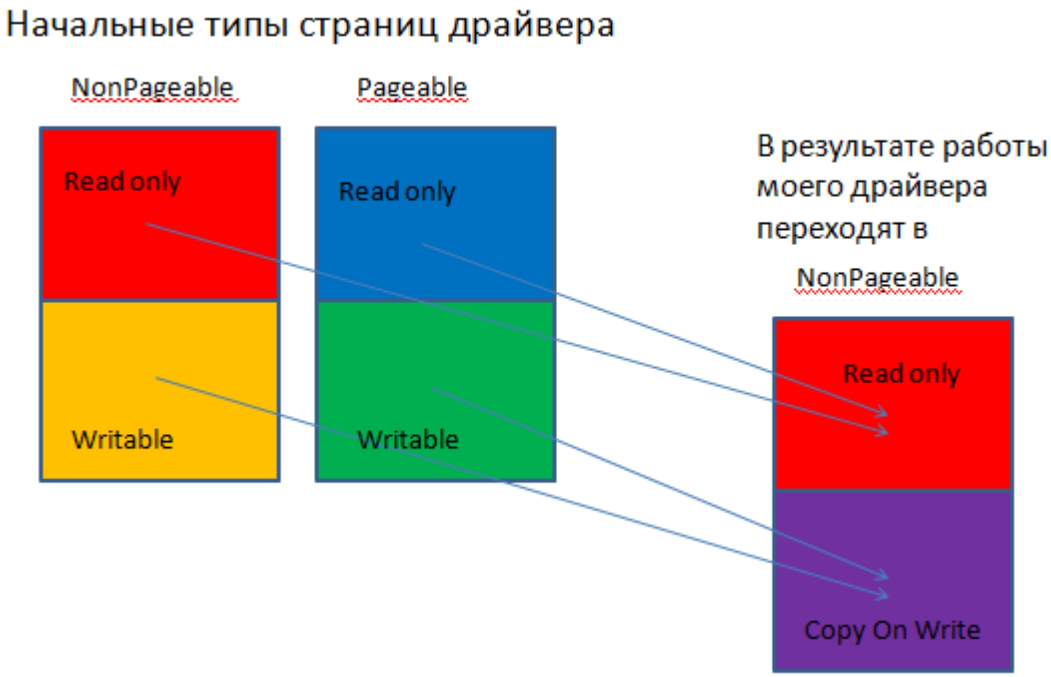


Поделим все страницы на 4 типа (executable будем относить к тому же типу, что и read-only):

- NonPageable, read-only
- NonPageable, writeable
- Pageable, read-only

- Pageable, writable

В результате работы моей программы все страницы становятся NP, read-only не изменяются, а writeable переходят в COW.



Рассмотрим, поподробнее, кол-во памяти занимаемое страницами каждого типа. Для них посчитаем минимальное, максимальное количество, а так же выведем требования к вероятностной модели.

А также, для каждого типа страниц подсчитаем изменение нагрузки на процессор, вызванной страницами этого типа, при работе моей программы.

2.3.1.2. NonPageable read-only страницы

Неподкачиваемая память, только чтение. Это означает, во-первых, что не будет никаких проблем с высокими IRQL. А во-вторых, что поскольку, тут только чтение в страницах, резидентных в памяти, то их кол-во со временем не меняется. Следовательно, можно точно посчитать выгоду по памяти.

Кроме того, следует отметить, что нагрузка на процессор не меняется, так как подкачки/отгрузки страниц нет. Следует учитывать только работу моего драйвера. И то только в моменты загрузки в систему нового драйвера. ...

Было: N экземпляров по x. стр. Стало: 1 экземпляр из x страниц. Где N, это кол-во драйверов, а x - кол-во страниц одного экземпляра, имеющих тип NP, read-only.

Вывод: Nx -> x.

(Кроме того, следует еще учесть размер моего драйвера.)

2.3.1.3. Pageable readonly страницы

Было: N экземпляров по x страниц, находившихся в подкачиваемом пуле памяти. Значит, в зависимости от типа драйвера и интенсивности его работы кол-во страниц в оперативной памяти могло меняться от 0 до Nx. (и только низкие IRQL)

Стало: (тоже, что и в предыдущем случае) 1 экземпляр из x страниц, но находящийся в неподкачиваемой памяти. Опять только чтение, значит, кол-во этих страниц не меняется. То есть, их всегда будет ровно x.

Мин оценка производительности моей программы (не нагруженный случай): 0 ->x.

Макс: Nx -> x.

Чтобы сделать вывод о полезности данного преобразования, нужно использовать мат модель подкачки страниц. И по ней сделать выводы о том,

- Сколько страниц их Nx находятся в памяти в среднем. (оценка занимаемой памяти)
- Как часто происходит подкачка и отгрузка страниц. (нагрузка на процессор)

При этом следует учитывать, что N наборов страниц независимые и одинаковые, а страница в них - x штук - могут быть зависимы (например, потому что нужны почти одновременно, или из-за того, что оптимальный алгоритм подкачки страниц подкачивает страницы блоками).

Вывод: необходимо найти мат модель подкачки страниц, определить среднее число страниц в системе и уже их него делать выводы.

2.3.1.4. Pageable writable страницы

Было: тоже, что и в предыдущем случае. => min=0, max=Nx, и необходимо использовать мат модель подкачки страниц.

Стало: 1 экземпляров из x страниц, и при запросах на запись выделяются новые страницы.

(Их всегда можно выделить, потому что данный код работает только при низких IRQL. Это связано с тем, что изначально эти страницы были Pageable, а значит, создатели драйвера позаботились о том, чтобы этот код не вызывался при высоких IRQL.)

В начале, страниц было ровно x, но во время работы драйверов при попытках записи кол-во страниц увеличивается и может дойти до (N+1)x, если каждый драйвер выполнит запись во все свои страницы (все страницы каждого драйвера и плюс еще нулевой набор).

Кол-во страниц с записями в системе можно уменьшить, если:

- анализировать, одинаковые ли изменения драйверы в них производят и объединять страницы с одинаковыми изменениями;
- измененные страницы обратно делать Pageable.

Вывод:

- Минимальная выгода: 0 -> (N+1)x, то есть V = -(N+1)x. (Когда драйвер обращается к своим страницам очень редко, но при этом успел сделать запись в каждую страницу).
- Макс: Nx->x, V= +(N-1)x (когда драйвер постоянно использует свои страниц, но не пишет в них).

Следует отметить, что, как правило, драйвер реально делает запись только в небольшое кол-во из своих x страниц кода. А остальные только читает и исполняет. Это с вязано с тем, что если даже запись происходит в одну страницу из секции, то как writeable помечается все секция целиком. (это касается только страниц кода)

То есть : (0 ÷ N)x -> (1 ÷ N+1)x

Следует создать и использовать мат модель разделения страниц с одной стороны и мат модель подкачки/отгрузки страниц с другой стороны. И найти такие параметры N - кол-во драйверов и x - кол-во Pageable writable страниц в них, при которых объединение таких страниц будет выгодно.

Что касается мат модели разделения страниц, то в данном случае важно только общее кол-во занимаемых страниц неподкачиваемой памяти, в то время как временные показатели (например, среднее время между разъединением страниц) отходят на второй план.

Нагрузка на процессор (во время работы системы) в этом случае существенно снижается, так как не нужно подгружать страницы с диска для операции чтения или исполнения, которые происходят чаще чем операции записи. А те страницы, в которые происходит запись подгружаются лишь 1 раз. Мин выгода: за все время - 0 (вообще не используют эти страницы). Макс: t/(время подгрузки с диска + отгрузки 1й страницы) - 0 (не пишут, но постоянно читают). Средняя же полезность определяется средним кол-вом страниц, принадлежащих этим драйверам, подгружаемых с диска в минуту, из модель подкачки страниц.

Нужно еще провести эксперименты с разными драйверами и выяснить в какое кол-во страниц из x на самом деле идет запись.

2.3.1.5. NonPageable writable страницы

Было: ровно Nx страниц в неподкачиваемой памяти.

Стало: (аналогично предыдущему случаю) сначала бал 1 набор из x страниц, но с течением времени, когда приходят запросы на запись страницы появляются новые страницы, пока не разделятся все, которые только могут. Максимальное число страниц в памяти в этом случае (N+1)x, где x - кол-во страниц этого типа в одном драйвере.

Во-первых, это неподкачиваемый пул памяти, значит могут быть высокие IRQL. Это значит, что при попытке записи на страницу Copy On Write может оказаться невозможно выделить новую страницу для данного экземпляра драйвера.

Поэтому рассмотрим 2 случая: отдельно низкие и высокие IRQL.

2.3.1.5.1. Низкие IRQL

Всегда можно выделять новые страницы памяти. Тут все просто. Очень похоже на предыдущий случай. Мин: Nx->(N+1)x. Макс: Nx->x.

Вывод: Собрать статистику кол-ва страниц (из NP writable), в которые на самом деле происходит запись. Nx->(1 ÷ N+1)x. И мат модель разделения страниц из предыдущего случая.

2.3.1.5.2. Высокие (IRQL ≥ Dispatch)

С одной стороны нельзя выделять новые страницы, с другой стороны нельзя изменять код других драйверов или проигнорировать запрос на запись, что может привести к BSOD. Эта проблема решится, если заранее создать буфер страниц, и при необходимости брать странички оттуда, пополняя его когда это возможно и необходимо. Итак, в этом случае в памяти находится еще и резервный буфер страниц.

Мат модель для этого случая строится на основе мат модели разделения страниц из предыдущего случая с учетом нового буфера и с акцентом на время между двумя попытками записи. Последнее особенно важно, так как именно тут оценивается размер буфера, а если его в какой-то момент не хватит, это приведет к временному зависанию потоков на высоких IRQL, что плохо отражается на надежности системы.

Из данной мат модели будут получены оценки времени между событиями и исходя их них будет подкорректирован и размер и алгоритм пополнения резервного буфера страниц буфера.

Кроме того, необходимо провести эксперименты с разными драйверами и выяснить в какое кол-во страниц из x на самом деле идет запись. Кроме того, можно сделать так,

чтобы мой драйвер при первом включении собирал статистику о кол-ве и частоте использования страниц, и исходя из этого корректировал размер буфера.

2.3.1.6. Итог

Итак, рассмотрены все 4 типа страниц и их основные свойства и особенности, сведем информацию в одну таблицу для наглядности.

	NP (любые IRQL)	P (low IRQL)
read only	1 тип. $Nx \rightarrow x$ Известно точное кол-во страниц.	2 тип. $(0 \div N)x \rightarrow x$ Мат модель подкачки страниц (№1).
writeable	4 тип. $Nx \rightarrow (1 \div N+1)x - M$ Дополнительный буфер памяти из M страниц. Мат модель разделения страниц при записи (№2) с учетом времени между событиями.	3 тип. $(0 \div N)x \rightarrow (1 \div N+1)x$ Мат модель подкачки страниц (№1) и мат модель разделения страниц при записи (№2) .

Пример.

Для иллюстрации полученных значений, приведем численные показатели. Допустим, в систему загружается 100 драйверов, пусть они имеют по 50, 30, 10 и 10 страниц каждого типа, соответственно. И пусть, изначально из всех подкачиваемых страниц в памяти находилось 50%, размер подкачиваемого буфера выбирается из расчета 6 страниц на драйвер. Тогда, можем рассчитать выигрыш. Для первого типа: x*(N-1) = 50*99 = 4950 страниц; для второго: x*50%*N - x = 30*50 - 30 = 1470; для третьего 7+(7-3)*100 = 407; для четвертого: 207. Итого, из всего 10 тысяч страниц, удалось сэкономить четь больше 7 тысяч. А это **70%** и 27 Мб.

Итак, рассмотрев 4 типа памяти приходим к необходимости учета двух различных мат моделей. Рассмотрим их подробнее.

2.3.3. Математическая модель разделения страниц при записи (Модель №2).

Математическая модель разделения страниц при записи (№2) с учетом времени между событиями.

2.3.3.1. Основные положения

Пусть в начальный момент времени в виртуальной памяти было N экземпляров драйвера по L станиц, которые отображались на один "нулевой" набор из L страниц в физической памяти. Экземпляры считаем одинаковыми и независимыми, а страницы из одного набора - различными и, возможно, не независимыми друг от друга. Это связано с тем, что драйвер может в одной своей функции сделать записи в несколько своих страниц, например, идущих подряд. Для моей системы это будет выглядеть как несколько последовательных, но очень близких по времени, запросов на запись (то есть вызовов MyMmAccessFault). Таким образом запись в первую из этих страниц будет автоматически означать и запись во все остальные.

Пусть в некоторый момент времени t приходит запрос на запись в j-ю страницу i-того набора - write(i, j). Тогда вызывается моя функция MyMmAccessFault, происходит выделение новой страницы (или взятие из буфера при высоких IRQL), и расщепление страниц. Таким образом, для i-го набора j-ая виртуальная страница ссылается теперь не на аналогичную копируемую при записи страницу из "нулевого" набора, а на свою отдельную доступную для записи физическую страницу. Дальнейшие запросы на запись в эту страницу никак не влияют на работу остальной системы и не будут приводить к нарушениям прав доступа, и вызовам MmAccessFault, следовательно, в данной модели могут не учитываться.

Краткий итог - характеристика модели: существуют N одинаковых независимых экземпляров, содержащих L различных не независимых страниц; кроме того, важен только первый вызов write(i,j) для каждого $i \in [1,N]$ и $j \in [1,L]$.

2.3.3.2. Простейший случай: 1 драйвер и 1 страница. Обоснование применимости.

Раз экземпляры одинаковы и независимы, можно рассмотреть сначала поведение одного из них, а потом обобщить это на N штук.

Поскольку страницы не независимы, и связь между ними (их корреляция) изначально не известна, построить математическую модель довольно сложно. Однако, из этого затруднения можно выйти, построив программу специфическим образом:

а) для случая Pageable Writable страниц, где важно среднее кол-во страниц в системе, можно провести ряд экспериментов для каждого драйвера и найти m = выборочное среднее кол-ва страниц, в которые ведется запись за один вызов функции драйвера, то есть которые требуется разъединить почти одновременно. Потом построить мат модель для L' независимых страниц и умножить ее на это выборочное среднее. L' определяется из формулы L/m. Кроме того, надо проверить данную гипотезу; то есть построить модель и подставить, возможно, для случая где важно лишь общее кол-во страниц это не важно.

б) для случая NonPageable Writable страниц при высоких IRQL, где наоборот, важнее время между вызовами, можно просто сделать отдельный буфер для каждой странички и рассматривать их отдельно, а пополнение сделать раз в определенное время, например, равное мат ожиданию времени между двумя соседними вызовами. или просто увеличить размер буфера страниц в m раз, где m= выборочное среднее кол-ва страниц, в которые ведется запись за один вызов функции драйвера, то есть которые требуется разъединить почти одновременно.

Страницы драйвера не являются независимыми. Покажем это на примере. Пусть в драйвере есть функция, которая выполняет последовательно операции записи на две различные страницы его кода. Тогда, с точки зрения моего драйвера, этот драйвер при вызове этой функции будет посылать запрос на запись в эти две страницы почти одновременно. И если в какой-то момент времени будет приходить запрос на запись в первую из этих страниц, то с довольно большой вероятностью (зависящей от того, из каких еще мест этого драйвера будут выполняться операции записи в эти страницы) ко второй странице тоже будет приходить запрос на запись в этот же момент времени. Таким образом эти две страницы драйвера не могут считаться независимыми. Поскольку, изначально о драйвере нет никакой информации, то нельзя считать его страницы независимыми. Однако, можно считать и независимыми в совокупности.

Утверждение, что страницы не независимы, привело бы к тому, что Пуассоновский процесс в данной модели необходимо было бы заменить на процесс восстановления. Что сделало бы вычисления слишком громоздкими.

Однако, если предположить, что вызовы функций драйвера системой можно рассматривать как независимые, то можно пересчитать математическую модель для кол-

ва различных вызовов, и умножать получившееся кол-во страниц на выборочное среднее кол-ва страниц, в который драйвер делает запись при одном вызове.

2.3.3.3. Время до первой записи как случайная величина

Рассмотрим случай, когда у каждого из N одинаковых независимых драйверов всего одна станица: L=1.

Изначально (в момент времени $t_0 = 0$) в памяти находится одна общая для всех страница, копируемая при записи. Когда от i-го драйвера к ней приходит запрос на запись (write(i)), странички разделяются и в памяти появляется новая страница, доступная для записи, но видимая только i-му драйверу. Теперь он будет обращаться к ней напрямую. Процесс заканчивается, когда все N драйверов будут иметь свои копии этой страницы.

Событием будем считать поступление первого запроса на запись от i-го драйвера, всего за все время работы произойдет N событий. Кол-во событий произошедших за время t обозначим K(t).

Обозначим ξ_i - время, в которое пришел запрос на запись от i-го драйвера, то есть момент наступления события. Для каждого драйвера ξ - случайная величина, мат ожидание $M\xi$ и функцию распределения $F(\xi)$, которой можно приближенно найти из статистических данных, собранных за время нескольких запусков системы.

Получается, что есть N случайных величин ξ, независимых, одинаково распределенных, с функцией распределения $F(\xi)$.

Нужно найти:

1. Сколько событий произойдут к определенному моменту времени? Это нужно для того, чтобы принять решение о целесообразности объединения страниц в случае Pageable Writable.

Ищем мат ожидание кол-ва событий $MK(t)$.

2. Найти вероятность того, что несколько событий произойдут почти одновременно. То есть интервалы между ними будут настолько малы, что резервный буфер не будет успевать заполняться. Это нужно для случая NonPageable, Writable для корректировки размеров буфера.

Ищем вероятность того, что более чем k событий произойдут на интервале меньшем ε, где ε определяется исходя из времени, которое нужно системе, чтобы заполнить резервный буфер.

$P(\geq k \text{ событий произойдут на отрезке } [t_1; t_2], \text{где } t_2 - t_1 < \varepsilon)$.

1) Вероятность того, что ровно k из N событий произойдут за время t, обозначим P(K=k,t).

$$\begin{aligned} P(K=k,t) &= P(\xi_1 < t, \dots, \xi_k < t, \xi_{k+1} \geq t, \dots, \xi_n \geq t) \cdot C_n^k \\ &= \langle \textit{учтем независимость случайных величин} \rangle \\ &= P(\xi_1 < t) \cdot \dots \cdot P(\xi_k < t) \cdot P(\xi_{k+1} \geq t) \cdot \dots \cdot P(\xi_n \geq t) \cdot C_n^k \\ &= \langle \textit{поскольку } F(\xi) = P(\xi < t), \textit{по опр. функции распределения} \rangle \\ &= F_1(t) \cdot \dots \cdot F_k(t) \cdot \big(1 - F_{k+1}(t)\big) \cdot \dots \cdot \big(1 - F_n(t)\big) \cdot C_n^k \\ &= \langle \textit{учтем, что случайные величины } \xi \textit{ одинаково распределены} \rangle \\ &= F(t)^k \cdot \big(1 - F(t)\big)^{n-k} \cdot C_n^k. \end{aligned}$$

Таким образом $P(K=k,t) = C_n^k \cdot F(t)^k \cdot \big(1 - F(t)\big)^{n-k}$, где $C_n^k = \frac{n!}{k!(n-k)!}$ - число сочетаний из n по k.

Мат ожидание $MK(t) = \sum_k k \cdot P(K=k,t) = \sum_{k=0}^n k \cdot C_n^k \cdot F(t)^k \cdot \big(1 - F(t)\big)^{n-k}$.

$$\mathbf{MK(t) = \sum_{k=0}^{n} k \cdot C_n^k \cdot F(t)^k \cdot \big(1 - F(t)\big)^{n-k}.}$$

2) Вероятность того, что ровно k из N событий произойдет на временном отрезке $[t_1; t_2]$, обозначим P(K=k, $[t_1; t_2]$).

$$\begin{aligned} P(K=k,[t_1;t_2]) &= P(\xi_1, \dots, \xi_k \in [t_1;t_2];\ \xi_{k+1}, \dots, \xi_n \notin [t_1;t_2]) \cdot C_n^k \\ &= \langle \textit{учтем, что случайные величины } \xi \textit{ независимы и одинаково распределены} \rangle \\ &= C_n^k \cdot P(t_1 \leq \xi_1 < t_2)^k \cdot P(\xi_n \notin [t_1;t_2])^{n-k}, \end{aligned}$$

$$P(t_1 \leq \xi_1 < t_2) = F(t_2) - F(t_1),$$

$$P(\xi_n \notin [t_1;t_2]) = P(\xi_n < t_1 \textit{ или } \xi_n \geq t_2) = F(t_1) + \big(1 - F(t_2)\big) = 1 - \big(F(t_2) - F(t_1)\big).$$

Следовательно,

$$P(K=k,[t_1;t_2]) = C_n^k \cdot \big(F(t_2) - F(t_1)\big)^k \cdot \Big(1 - \big(F(t_2) - F(t_1)\big)\Big)^{n-k}.$$

Обозначим $A = F(t_2) - F(t_1)$. При n>100 и nA(1-A)>20 можно воспользоваться теоремой Муавра-Лапласа.

3) Вероятность того, что больше или ровно k из N событий произойдет на временном отрезке $[t_1; t_2]$, обозначим P(K≥k, $[t_1; t_2]$).

$$P(K \geq k,[t_1;t_2]) = \sum_{s=k}^{n} C_n^s \cdot \big(F(t_2) - F(t_1)\big)^s \cdot \Big(1 - \big(F(t_2) - F(t_1)\big)\Big)^{n-s}.$$

4) Коллизией из k событий назовем момент, когда k событий попали на временной отрезок длиной < ε.

P(∃ хоть одна коллизия из k событий) ≅

$\sum_{\text{по всем непересекающимся отрезкам длины } \varepsilon} P(K \geq k, [t_1; t_1 + \varepsilon])$

5) P(∃ коллизия из 2-х событий, когда N=2) =

$P(|\xi_1 - \xi_2| < \varepsilon) = \int_0^\infty f(t)dt \cdot \big(F(t+\varepsilon) - F(t-\varepsilon)\big)$. Здесь учтено, что случайная величина ξ непрерывна (т.к. это временя), и формула полной вероятности. (Для дискретной случайной величины эта формула выглядела бы так:

$P\{|\xi_1 - \xi_2| < \varepsilon\} = \sum_i P_1(x_i) * P_2(|x_i - \xi_2| < \varepsilon) = \sum_i P_1(x_i) * P_2(x_i - \varepsilon < \xi_2 < x_1 + \varepsilon) = \sum_i P_1(x_i) * \big(F_2(x_i + \varepsilon) - F_2(x_i - \varepsilon)\big)$

). Можно обобщить на случай коллизии k из N событий. Опуская длинные выкладки, приходим к формуле:

$$\boldsymbol{P}(\exists \text{ коллизия } \boldsymbol{k} \text{ из } \boldsymbol{N}) = \int_0^\infty \boldsymbol{f(t)dt} \cdot \boldsymbol{P}(\xi_2, \dots, \xi_k \in [\boldsymbol{t-\varepsilon};\ \boldsymbol{t+\varepsilon}];\ \xi_{k+1}, \dots, \xi_n \notin [\boldsymbol{t-\varepsilon};\ \boldsymbol{t+\varepsilon}]) \cdot \sum_{i=1}^{n-k+1} \boldsymbol{C_{n-i}^{k-1}}$$

$$= \int_0^\infty \boldsymbol{f(t)dt} \cdot \big(\boldsymbol{F(t+\varepsilon) - F(t-\varepsilon)}\big)^{\boldsymbol{k-1}} \cdot \Big(\boldsymbol{1} - \big(\boldsymbol{F(t+\varepsilon) - F(t-\varepsilon)}\big)\Big)^{\boldsymbol{n-k-1}} \cdot \sum_{i=1}^{n-k+1} \boldsymbol{C_{n-i}^{k-1}}.$$

2.3.3.4. Пуассоновский процесс

Функцию распределения F(ξ) можно получать либо чисто из экспериментальных данных (статистическая функция распределения), либо можно получить ее вид из теоретических соображений с точностью до неизвестных параметров. Этот параграф посвящен именно этому.

Как и в предыдущем случае, рассматривается 1 драйвер обращающийся к своей единственной странице. Но в отличие от предыдущего случая, не будем ограничиваться только его первым обращением. Рассмотрим его обращения как случайный процесс. То есть событие - это запись в станицу. События происходят через случайные промежутки времени, и они независимы.

Как видим, обращения драйвера к своей странице это Пуассоновский процесс, о есть процесс с независимыми приращениями. В данном случае интенсивность Пуассоновского процесса λ не известна, и берется из экспериментальных данных. Она окажется разной для разных драйверов, и вообще говоря, может зависеть от времени.

1) Пуассоновский процесс

$$P(K(t) = k) = \frac{(\lambda t)^k e^{-\lambda t}}{k!}, t \in [0, \infty), k = 0,1, \dots \Leftrightarrow K(t) \in Po(\lambda t).$$

λ - интенсивность - мат ожидание числа событий в единицу времени.

MK(t)=λt.

2) Распределение интервала времени T между событиями. Для Пуассоновского процесса оно имеет показательный закон распределения:

$$F_T(t) = P(T < t) = 1 - P(K(t) = 0) = 1 - e^{-\lambda t},$$

где T - время между двумя событиями.

Функция плотности распределения имеет вид:

$$f_T(t) = \lambda e^{-\lambda t}, t \geq 0.$$

MT=1/λ, DT=$1/\lambda^2$

$$P(t_1 \leq T < t_2) = F_T(t_2) - F_T(t_1) = e^{-\lambda t_1} - e^{-\lambda t_2}.$$

Таким образом, выражение

$$\boldsymbol{F_T(t) = P(T < t) = 1 - P(K(t) = 0) = 1 - e^{-\lambda t}},$$

можно использовать в качестве теоретической функции распределения времени до первого вызова.

2.3.3.5. Модификация Пуассоновского процесса.

Пуассоновский процесс довольно неплохо описывает механизм обращения драйвера к своим страницам, но в данном случае важно только первое обращение. Другими словами, на самом деле, до первого события общий процесс представляет собой сумму N пуассоновских процессов с одинаковыми интенсивностями (если по 1 странице на драйвер), после наступления первого события, тот простейший процесс в котором оно произошло исключается из рассмотрения, так как больше не сможет сгенерировать событие, то есть это уже будет сумма N-1 процессов и т.д. пока все страницы не разъединятся.

Пусть одновременно и независимо происходят N простейших потоков $K_{1(t)}, \dots, K\_n(t)$ с $\lambda_i, i = \overline{1,n}$. Тогда общая сумма числа событий в этих потоках $K(t) = \sum_{i=1}^{n} K_i(t)$ является Po(λt) с $\lambda = \sum_{i=1}^{n} \lambda_i$.

Пуассоновский процесс с переменной интенсивностью. В общем случае это процесс с заданной (неслучайной) λ(t). Здесь же рассмотрим только случай с кусочно постоянной λ. Пусть N простейших пуассоновских процессов независимы и происходят последовательно на непересекающихся отрезках. Тогда суммарное число событий $K(t) \in Po(\lambda t)$, где $\lambda = \frac{1}{t} \cdot \sum_{i=1}^{n} \lambda_i \cdot (t_i - t_{i-1}), [0,t) = \bigcup_{i=1}^{n} [t_{i-1}, t_i)$. Д-во:

$M(\xi + \eta) = M\xi + M\eta, MK = \lambda t; MK_i = \lambda_i(t_i - t_{i-1}), \lambda t = \sum_{i=1}^{n} \lambda_i(t_i - t_{i-1})$.

Таким образом, реальный процесс в системе может быть представлен по аналогии с пуассоновским процессом с переменной интенсивностью с кусочно постоянными λ. Но он не будет уже являться пуассоновским процессом, так как λ_i зависит не от времени, а от кол-ва наступивших событий.

Такое представление оно дает возможность посчитать вероятность коллизий. Например, коллизии из двух элементов на k-м шаге. Итак, считаем, что у каждого из простейших процессов, составляющих общий процесс, интенсивность = λ_0, в самом начале она = $\lambda_0 n$, потом $\lambda_0(n-1)$, и т.д., а в самом конце $= 0$. Тогда,

$$P(T < \varepsilon \mid K(t) = k) = 1 - e^{-\lambda_k \varepsilon} = 1 - e^{-\lambda_0 (n-k)\varepsilon}.$$

2.3.3.6. Итоги.

В рамках математической модели "разделения" страниц при записи были получены следующие формулы.

Математическое ожидание количества событий, которые произойдут к определенному моменту времени:

$$MK(t) = \sum_{k=0}^{n} k \cdot C_n^k \cdot F(t)^k \cdot \left(1 - F(t)\right)^{n-k},$$

где функция распределения считается по формуле $F_T(t) = P(T < t) = 1 - P(K(t) = 0) = 1 - e^{-\lambda t}$.

Оно необходимо для того, чтобы в случае Pageable страниц принять решение об их "объединении", а так же для для того, чтобы правильно расчитывать размер резервного буфера страниц.

При реализации учитывалось, что формула для распределения получена исходя из представления данного процесса Пуасоновским, то есть имеет приближенный вид. Был создан метод, при котором, если размер буфера приближается к значению, предсказанному в рамках математической модели, например, становится разным 75%, то необходимо увеличить буфер в 2 раза.

Вероятность того, что несколько событий произойдут почти одновременно. То есть интервалы между ними будут настолько малы, что резервный буфер не будет успевать заполняться. То есть вероятность того, что более чем k событий произойдут на интервале меньшем ε, где ε определяется исходя из времени, которое нужно системе, чтобы заполнить резервный буфер.
$P(\geq k \text{ событий произойдут на отрезке } [t_1; t_2], \text{где } t_2 - t_1 < \varepsilon)$.

$$P(\exists \text{ коллизия } k \text{ из } N)$$
$$= \int_0^{\infty} f(t)dt \cdot P(\xi_2, \dots, \xi_k \in [t-\varepsilon;\ t+\varepsilon];\ \xi_{k+1}, \dots, \xi_n \notin [t-\varepsilon;\ t+\varepsilon])$$
$$\cdot \sum_{i=1}^{n-k+1} C_{n-i}^{k-1}$$
$$= \int_0^{\infty} f(t)dt \cdot \big(F(t+\varepsilon) - F(t-\varepsilon)\big)^{k-1} \cdot \Big(1 - \big(F(t+\varepsilon) - F(t-\varepsilon)\big)\Big)^{n-k-1}$$
$$\cdot \sum_{i=1}^{n-k+1} C_{n-i}^{k-1}.$$

Этот результат необходим для случая NonPageable, Writable для корректировки размеров буфера и частоты работы подкачивающей нити, добавляющей в него страницы. В реализации опять же учтен приближенный характер выражений. Если эта вероятность становится больше определенного значения, то увеличивается размер буфера, и нить начинает чаще проверять, не нужно ли добавить страниц.

Следует отдельно отметить, что несмотря на то, что для приближенных вычислений используется формула $F_T(t) = P(T < t) = 1 - P(K(t) = 0) = 1 - e^{-\lambda t}$, полученная в рамках приближения реального процесса Пуассоновским, сами формулы для математического ожидания и вероятности коллизии были выведены без привязки к определенному типу распределения случайной величины - времени до первого обращения к странице. Следовательно, в качестве F(t) можно подставить статистическую формулу распределения, полученную в ходе большого количество экспериментов.

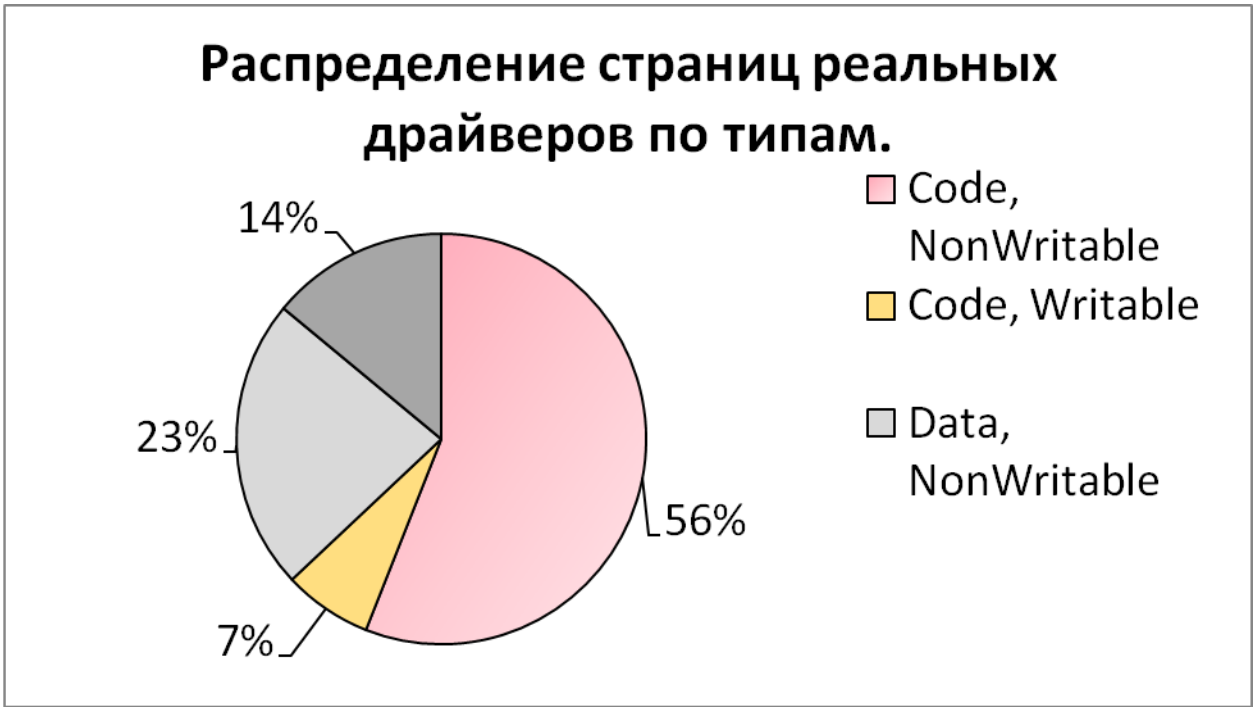
Однако, поскольку математическая модель используется лишь для приближенной оценки, а получение статистических функций для каждой страницы каждого драйвера весьма затруднительно, было решено ограничиться приближением с Пуассоновским процессом.

2.4. Тестирование и сравнение результатов с математической моделью

2.4.1. Сбор статистики

Во-первых, проанализируем, как распределяются страницы по типам. Это дает возможность судить об итоговых выигрышах, а так же еще раз доказывает актуальность исследования.

Были проанализированы ~300 драйверов, установленных на моем рабочем компьютере. На диаграмме – усредненные данные.



Были получены следующие результаты:

Всего драйверов = 320, всего их страниц = 22480, то есть 87Мб памяти, из них: Code = 13220, Data = 9260, что равняется примерно 59% и 41% соответственно. Распределение страниц по типам:

Тип	cnpr	cnpw	cpr	cpw	dnpr	dnpw	dpr	dpw
Страницы	9567 43%	453 2%	3200 14%	20 0%	5170 23%	3883 17%	149 1%	38 0%

где c - код, d - данные, p - pageable, np - nonpageable, w - writable, r - nonwritable.

Максимальное число доступных для записи NonPageable страниц было = 7, а именно, в драйверах http.sys и ntfs.sys. Всего Writable страниц кода - 2%, данных - 17% от общего числа страниц драйвера. Всего NonWritable страниц (и кода и данных) = 18106, что составляет 81% всех страниц: 57% - это код и 24% - это данные.

2.4.2. Нахождение λ и сравнение результатов с математической моделью

Некоторые драйверы были протестированы более подробно. Среди них как простейшие тестовые драйверы, так и более сложные (например, http.sys).

Во-первых, Было протестировано согласование поведения драйвера с математической моделью. Для этого было проведено большое количество экспериментов, в каждом из которых измерялось время до первого обращения драйвера к определенной странице памяти. Как уже было показано ранее, обращения драйвера к странице на запись фиксируется с помощью встройки в обработчик исключения ошибки страницы MmAccessFault.

Порядок проведение эксперимента: выбиралась определенная страничка драйвера и отслеживались записи только в нее. Драйвер запускался N раз, где N варьировалось от 100 до 10 000. При каждом запуске измерялось t_i - время до Page Fault, то есть время до первого обращения в страницу на запись.

После сбора данных, во-первых, строилась статистическая функция распределения, а во-вторых, рассчитывалась интенсивность Пуассоновского потока в математической модели λ. Поскольку математическое ожидание T = 1/λ, то λ рассчитывалось как обратная величина от T – среднего значения измеренных величин t_i – время до первой записи в страницу.

$$= \frac{1}{T}, \text{где } T = \frac{1}{N}\sum_{i=0}^{N} t_i .$$

После этого строились и сравнивались графики для теоретической и экспериментальной (статистической) функции распределения.

Статистическая функция распределения оказалась не непрерывной, а дискретной. По результатам 10 000 экспериментов для одной из страниц, величина t – время до первой записи на страницу, принимала лишь 11 различных значений.

Были получены результаты для всех NonPageable страниц драйвера http.sys, в которые он выполняет запись, даже для страниц данных. Однако, здесь приводится лишь один график, так как графики статистических функций распределения и полученные для разных страниц λ оказались очень похожими.

Экспериментальные данные приведены в таблице и на графике:

Значение времени	Как часто встречается
15	2209
31	558
46	1407
62	1553
78	1500
93	1245
109	1211
125	297
140	16
156	3
281	1

Таблица 3.

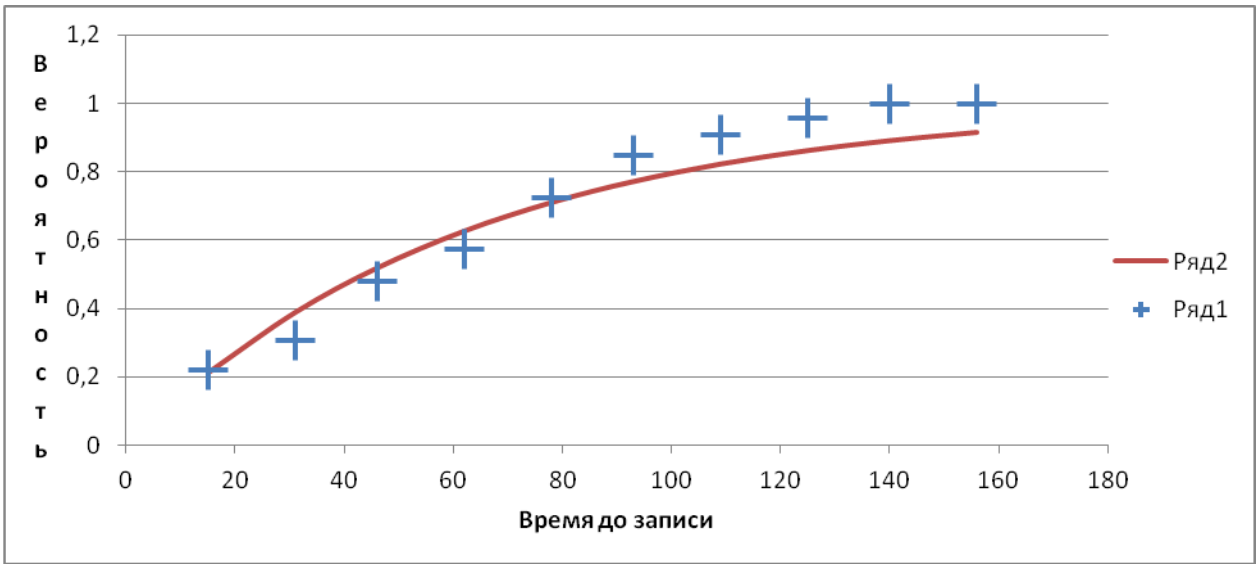


График 1 – сравнение функций распределения. Ряд 1 – экспериментальные данные, Ряд 2 – теоретическая функция распределения с интенсивностью ƛ, полученной из эксперимента.

На графике 2 показан графический расчет ƛ. В данном эксперименте параметр ƛ расчитывался двумя способами, как 1/T, т из графика. Поскольку $F_T(t) = 1 - e^{-\lambda t}$, то построив зависимость $-\ln(1-p)$ от t, где p – доля от общего количества значений времени, соответствующая этому t, мы получим прямую, проходящую через ноль, и имеющую угол наклона ƛ.

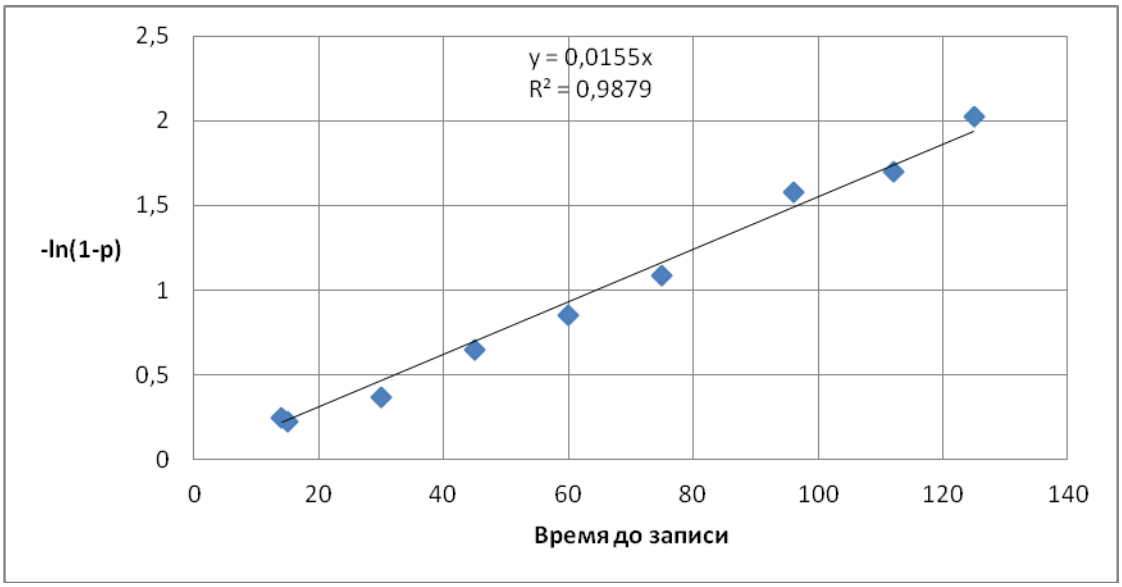


График 2. Графический расчет ƛ.

Как видно из графиков, экспериментальные данные довольно хорошо согласуются с математической моделью, а значит, можно сделать вывод, что поведение драйверов в целом хорошо согласуется с моделью Пуассоновского процесса.

Кроме того, были протестированы различные драйверы, они так же показали хорошее согласование с математической моделью, а ƛ для них оказалась в пределах [0, 0.8].

Кроме этого, в процессе эксперимента было установлено, что процент доступных для записи (writable) страниц, в которые действительно велась запись оказался в промежутке [0, 60%] для различных драйверов.

2.4.3. Время загрузки драйвера

В работе так же считались накладные расходы. В данном случае, поскольку цель работы – оптимизация работы с памятью, наблюдается незначительное снижение производительности, а именно немного увеличивается время загрузки драйвера в систему.

Были произведены измерения времени работы системы, времени обработки исключения ошибки страницы Page Fault, дополнительного времени, которое теперь тратится на загрузку одного из драйверов, будь то первый экземпляр, или последующие, а так же выявлены этапы работы, которые вносят наибольшую задержку.

Для тестирования использовался драйвер http.sys, так как это драйвер с наибольшим количеством страниц в системе, а значит, и время загрузки у него максимальное.

Был произведен ряд экспериментов, по результатам которых получилось:

Время, на которое увеличивается время загрузки первого экземпляра ~ 40 ms.

Время, на которое увеличивается время загрузки второго и последующих экземпляров ~ 5 ms.

Время обработки Page Fault ~ 15s.

Таким образом получается, что если в контейнере 100 драйверов, то время старта первого контейнера увеличивается на ~ 2-3 сек (потому, что большая часть драйверов гораздо меньше, чем http.sys). А время старта второго и последующих контейнеров увеличивается меньше, чем на секунду! Причем, это время можно еще уменьшить, потому что основную задержку дает выделение и высвобождение страниц, а высвобождение страниц можно сделать отложенным, что сэкономит еще 5-10 ms.

2.4.4. Тестирование Pageable страниц.

В работе не строится математическая модель подкачки страниц, однако, для подсчета выигрыша необходимо знать количество Pageable страниц, которые были в памяти до начала работы.

Был произведен ряд тестов, в которых для разных драйверов считалось количество подкачиваемых страниц, находящихся в памяти. Приведем тут значения для драйвера http.sys. Всего у этого драйвера 44 Pageable страницы, из них в определенный момент в

памяти находилось 15, в то время как 29 были сброщены на диск. Естественно, эти параметры меняются во время работы системы. А так же, они очень зависят от типа самого драйвера. Однако, общее правило для драйваеров на моем компюьтере было таким:

60% - откачено на диск

40% - находится в памяти.

2.4.5. Подсчет выигрышей.

Для того, чтобы подсчитать выигрыш, необходимо знать количество страниц драйвера, относящихся к разным типам.

Приведем эти данные для драйвера http.sys.

Тип	cnpr	cnpw	cpr	cpw	dnpr	dnpw	dpr	dpw
Страницы	45	8	44	0	40	13	0	0

Таблица 4.

Всего в драйвере 150 страниц, каждая страница – 4 Кб, следовательно всего 600 Кб памяти.

Расчет для случая неподкачиваемой памяти: NP, r = 45 *4 = 180 Кб; NP,w = (8-3) *4 = 20 Кб (Так как в резервном буфере было по 3 страницы для каждого драйвера).

Расчет для случая подкачиваемой памяти: P, w = 0.

Для P, r: Всего 44 страницы, из них 15 было в памяти для каждого драйвера раньше, а теперь для всех драйверов будет ровно 44 страницы. Тогда выигрыш будет (15*N - 44) * 4 Кб, то есть это выгодно, когда N>3.

Пусть N = 10. Тогда выигрыш: 200 б (от NP) * 9 + (15*10 - 44) * 4 б (от P) = 1800 + 424 Кб = 2224 Кб = 2,2 Мб. А всего, без нашей системы было бы 5б8 Мб. Получаем экономию: 38% .

Пусть N = 100. Тогда 200*99 + (15*100 - 44)*4 = 19 800 + 5 824 = 25 624 б = 25,0 Мб.А всего было бы 58.6 Мб. Экономия: 43%.

Таким образом в зависимости, от количества драйверов в системе можно получить экономию памяти порядка 50% ! А выигрыш исчисляется десятками мегабайт.

Все выше сказанное доказывает эффективность и необходимость созданной системы.

3. Заключение

3.1. Результаты

Данный алгоритм реализован для случая NonPaged Memory.

В процессе работы над реализацией драйвера, дедуплицирующего код других драйверов, был разработан и реализован ранее отсутствовавший в Windows механизм Copy On Write в NonPaged Memory. Данный механизм реализован и для случая работы при низких, и при высоких IRQL. Однако, следует заметить, что для случая работы на высоких IRQL с Writable страницами при активных попытках дедуплицируемого драйвера писать в свои страница кода этот механизм не может гарантировать стопроцентную безопасность и работоспособность системы. Вследствие чего, все драйверы должны предварительно тестироваться на наличие таких ситуаций, несмотря даже на то, что они предельно редки.

Кроме этого, была детально изучена работа менеджера памяти в ОС Windows с неподкачиваемым пулом памяти. При работе над реализацией алгоритма были с самого начала учтены особенности работы его структур, и при выборе способов реализации учитывались в том числе и будущие затраты, необходимые для переноса данной реализации на следующие выпуски ОС Windows.

Была разработана математическая модель, достаточно полно описывающая поведение системы и хорошо предсказывающая кол-во сэкономленных страниц. А так же математическая модель разделения страниц общего набора при попытках записи в них различными экземплярами драйвера, по которой можно определить размер заранее подготовленного буфера страниц, используемого в высоких IRQL, и кроме того определить математическое ожидание кол-ва разъединенных страниц в определенный момент времени.

Был выполнен ряд тестов системы на различных драйверах, результаты подставлены в математическую модель, проверено их соответствие.

3.2. Перспективы развития

Реализовать этот же алгоритм и для Paged Memory. Это означает, не делать те страниц, которые были Pageable не подкачиваемыми, а использовать существующий в Windows механизм Copy On Write для Paged памяти.

Сессионные драйвера. Ведь, в данной работе, речь шла только о не сессионных драйверах.

Добавление поддержки Large Pages. Поскольку многие экземпляры могут производить одинаковые записи в свои страницы, то можно не просто разъединять измененные страницы, как это происходит сейчас, а можно среди измененных страниц находить одинаковые и объединять их в одну, до следующей записи на одну их этих страниц. Это может дать еще довольно существенную экономию памяти для некоторых типов драйверов. А именно, для тех, которые делают в свои страницы много однотипных записей.

Кроме того, можно сделать более быстрое пополнение резервного буфера страниц для работы на высоких IRQL. Это бы увеличило надежность.

3.3. Возможные применения

3.3.1. Возможные применения метода дедупликации одинаковых страниц.

Контейнерная виртуализация для Windows.

Проблема, решаемая в данной работе, возникла из контейнерной виртуализации для Windows. И именно там находится ее главная область применения.

Предлагаемый метод решения может оказаться полезным и во многих других ситуациях, когда в системе Windows загружено одновременно несколько экземпляров одного драйвера.

Как оптимизация памяти в виртуальных машин Windows.

Однако, некоторые описываемые здесь аспекты, касающиеся поиска одинаковых страниц, могут оказаться полезными для виртуальных машин как оптимизация распределения памяти между машинами на ОС Windows, если вклиниться внутрь ядра ОС и отслеживать загрузку отгрузку драйверов, а остальную работы производить на хосте. Таким образом, если раньше приходилось анализировать память, чтобы найти идентичные страницы, то теперь можно получать часть информации от самой ОС.

На пользовательских компьютерах.

На пользовательских компьютерах данная технология тоже может быть полезна для драйверов, несколько экземпляров которых работают одновременно. (...подробнее...)

Для данных.

Тоже самое можно сделать и для виртуальных страниц с данными, если заранее известно, что записи в них происходят редко.

3.3.2. Возможные применения Copy On Write в NonPaged Memory

Кроме того, реализованная технология Copy on Write в NonPaged пуле памяти в Windows не имеет аналогов и может сама по себе иметь гораздо более широкое применение. Только следует помнить, что неверная конфигурация параметров (а именно размера резервного буфера) может приводить к сбоям в работе на высоких IRQL в случае, если попытки записи будут происходить слишком часто.

Контейнерная виртуализация для Windows.

Опять таки это контейнерная виртуализация для Windows, но можно объединять не только код драйверов, но и любые страницы в различных контейнерах. Важно только, чтобы они были идентичны и записи в них производились редко.

Как аналог KSM для ядра Windows.

То есть для периодического сканирования памяти системы, нахождения одинаковых страниц, и объединения их в одну с помощью описанного здесь механизма Copy on Write в NonPaged Memory.

4. Список литературы

1. Руссинович М., Соломон Д. «Внутреннее устройство Microsoft Windows». – М.: Русская Редакция; СПб.: Питер, 2008. – 992 с. Главы 1-4, 7, 9.
2. Intel 64 and IA-32 Architecture Software Developer's Manual, volume "System Programming Guide, Part 1"– Intel, 2012. – 1333 с.
3. Oney W. "Programming the Microsoft Windows Driver Model" – Redmond, Washington: Microsoft Press, 2003 - 846 с. Главы 1, 2.
4. Nebbett G. "Windows NT/2000 Native API Reference"– Indianapolis: Pearson Education (US) , 2000. – 512 с.
5. Таненбаум Э., "Современные операционные системы"– СПб.: Питер, 2010. – 1120 с. Главы 1, 4, 5, 11.
6. М. Тим Джонс, «Виртуальный Linux. Обзор методов виртуализации, архитектур и реализаций», 2007
7. Сергей Озеров, Александр Карабуто «Технологии виртуализации: вчера, сегодня, завтра»
8. Keith Adams, Ole Agesen A Comparison of Software and Hardware Techniques for x86 Virtualization
9. Crowley Charles, "Operating Systems, A Design-Oriented Approach".
10. Smith J., Nair R., "Virtual Machines, versatile Platforms for Systems and Processes".
11. Мендель Розенблюм, Тэл Гарфинкель Мониторы виртуальных машин: современность и тенденции (рус.). Издательство «Открытые системы»
12. Андрей Колесов Вернемся к нашим гипервизорам, PC Week/RE № 16 — 17 (670—671).
13. Горбачев О.Г., Гуз С.А., Натан А.А. "Основы теории случайных процессов.", Главы 1-3, Москва 2002.
14. Гнеденко Б.В. "Курс теории вероятностей." –М.: Наука, 1988.
15. Гнеденко Б.В.,Беляев Ю.К.,Соловьев А.Д. "Математические методы в теории надежности." –М.: Наука, 1965.
16. Archana Ganapathi, Viji Ganapathi, David Patterson " Windows XP Kernel Crash Analysis"– University of California, Berkeley.
17. http://habrahabr.ru/company/parallels/blog/245533/
18. https://infoboxcloud.ru/community/blog/iaas/220.html
19. http://www.linux-kvm.com/sites/default/files/KvmForum2008_KSM.pdf

20. http://www.linux-kvm.org/page/KSM
21. https://www.kernel.org/doc/Documentation/vm/ksm.txt
22. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Virtualization_Administration_Guide/chap-KSM.html
23. http://www.vmware.com/files/pdf/mem_mgmt_perf_vsphere5.pdf
24. http://www.vmgu.ru/search/ESX%7C%20/24
25. http://www.vmgu.ru/articles/vmware-transparent-page-sharing
26. http://hosting101.ru/articles/virtualization.html
27. http://www.ixbt.com/cm/virtualization-h.shtml
28. http://www.intel.com/cd/corporate/europe/emea/rus/update/360260.htm
29. http://msdn.microsoft.com/en-us/library/windows/hardware/ff545448(v=vs.85).aspx
30. http://support.microsoft.com/kb/244617/ru