ТЕМА 3.2. ОРГАНИЗАЦИЯ ВИРТУАЛЬНОЙ ПАМЯТИ

В данной теме рассматриваются следующие вопросы:

- Виртуальная память.
- Структура адресного пространства виртуальной памяти.
- Задачи управления виртуальной памятью: задача размещения, задача перемещения, задача преобразования адресов, задача замещения.
- Подкачка.
- Алгоритмы замещения страниц.
- Куча (heap).
- Стек

Лекции – 2 часа, лабораторные занятия – 2 часа, самостоятельная работа – 2 часа.

Экзаменационные вопросы по теме:

- Виртуальная память. Структуризация адресного пространства виртуальной памяти. Задачи управления виртуальной памятью.
- Подкачка. Алгоритмы замещения страниц. Куча (heap). Стек.

3.2.1. Виртуальная память

Упрощённо говоря, процессор обращается к памяти через шину. Адресами памяти, которыми обмениваются в шине, являются физические адреса, то есть сырые числа от нуля до верхней границы доступной физической памяти (например, до 2^{33}, если у вас установлено 8 ГБ оперативки). Ранее между процессором и микросхемами памяти располагался северный мост — отдельный чип, но в реализации Intel начиная с микроархитектуры Sandy Bridge он интегрирован на кристалл процессора [1].

Физические адреса являются конкретными и окончательными — без трансляции, без подкачки, без проверки привилегий. Вы выставляете их на шину и всё: выполняется чтение или запись.

Однако в современной операционной системе программы используют абстракцию — виртуальное адресное пространство. Каждая программа пишется в такой модели, что она выполняется одна, всё пространство принадлежит ей, код использует адреса логической памяти, которые должны быть преобразованы в физические адреса до того, как будет выполнен доступ к памяти. Концептуально преобразование показано на рис.3.2.1.

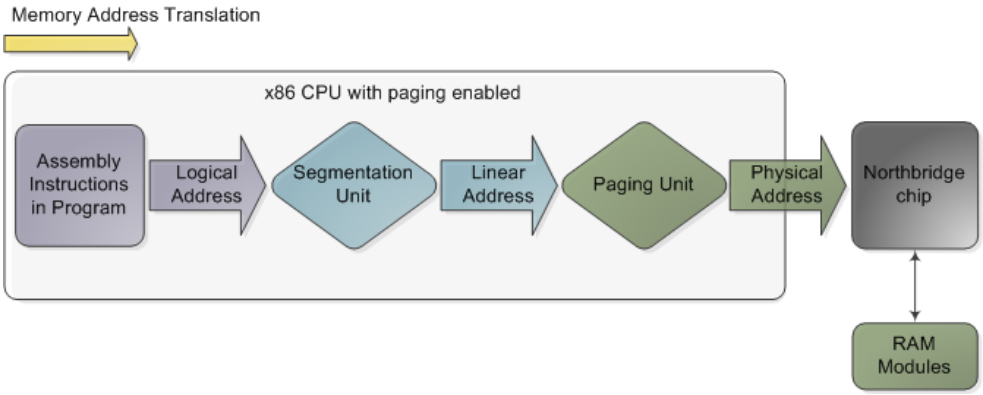

Рис. 3.2.1. Преобразование адресов памяти

Это не физическая схема, а только описание процесса преобразования адресов. Такая трансляция осуществляется всякий раз, когда CPU выполняет инструкцию, которая ссылается на адрес памяти.

Логический адрес на x86 состоит из двух частей: селектора сегмента и смещения внутри сегмента. Процесс трансляции включает два шага:

- учёт сегментного селектора и переход от смещения внутри сегмента к некоторому линейному адресу;
- перевод линейного адреса в физический.

Спрашивается, зачем нужен первый шаг и зачем нужны эти сегменты, почему бы напрямую не использовать линейные адреса в программе? Это результат эволюции. Чтобы действительно понять смысл сегментации x86, нам нужно вернуться в 1978 год.

Реальный режим

16-битный процессор 8086 использовал 16-битные регистры и мог напрямую адресовать только 2^{16} байт памяти. Инженеры придумывали, как же можно заставить его работать с большим объёмом памяти, не расширяя разрядность регистров.

Были придуманы сегментные регистры, которые должны были задавать, к какому именно 64-килобайтному куску памяти относится данный 16-битный адрес.

Решение выглядит логичным: сначала вы устанавливаете сегментный регистр, по сути говоря «так, я хочу работать с куском памяти начиная с адреса X»; затем 16-битный адрес уже используется как смещение в рамках этого куска.

Всего предусматривалось сначала четыре 16-битных сегментных регистра, потом добавили ещё два:

- CS = Code Segment
- DS = Data Segment
- ES = Extra (или Destination) Segment
- SS = Stack Segment
- FS
- GS

Названия этих регистров связаны с назначением. При выполнении инструкций они загружаются из сегмента кода. При обращении к стеку (инструкции push/pop) неявно используется сегмент стека (при работе с регистрами SP и BP). Некоторые инструкции (так называемые «строковые») используют фиксированные сегменты, например инструкция `movs` копирует из DS:(E)SI в ES:(E)DI.

Для вычисления линейного адреса ячейки памяти процессор вычисляет физический адрес начала сегмента — умножает сегментную часть виртуального адреса на число 16 (или, что то же самое, сдвигает её влево на 4 бита), а затем складывает полученное число со смещением от начала сегмента. Таким образом, сегменты частично перекрывались, и всего можно было адресовать около 1 МБ физической памяти. На рис. 3.2.2 адрес перехода складывается с содержимым сегментного регистра кода CS, умноженным на 16, чтобы получить реальный адрес памяти.

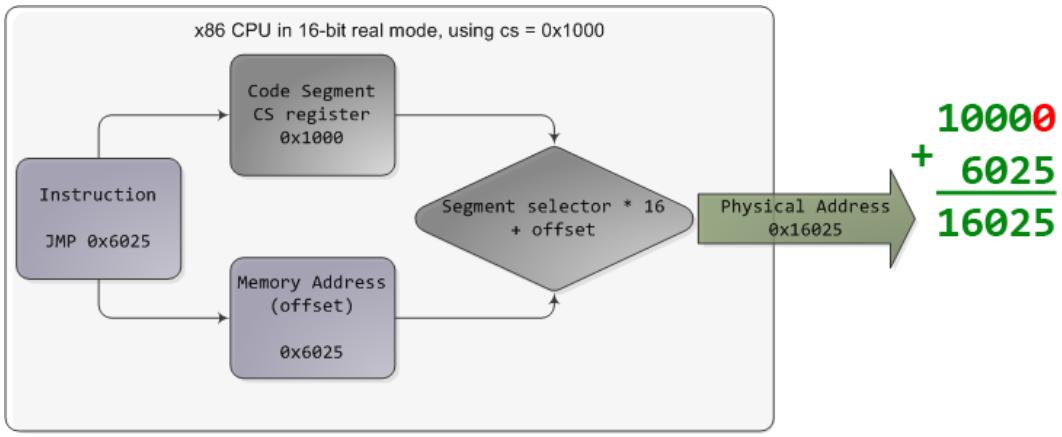

Рис. 3.2.2. Вычисление физического адреса в реальном режиме

В реальном режиме отсутствует защита памяти и разграничение прав доступа.

Программы были маленькие, поэтому их стек и код полностью помещались в 64 КБ, не было проблем. В языке C тех древних времён обычный указатель был 16-битный и указывал относительно сегмента по умолчанию, однако существовали также far-указатели, которые включали в себя значение сегментного регистра. Призраки этих far-указателей преследуют нас в названиях типов в WinAPI (например, LPVOID — long (far) pointer to void).

Защищённый режим

В 32-битном защищенном режиме также используется сегментированная модель памяти, однако уже организованная по другому принципу: расположение сегментов описывается специальными структурами (таблицами дескрипторов), расположенными в оперативной памяти.

Сегменты памяти также выбираются все теми же сегментными регистрами. Значение сегментного регистра (сегментный селектор) больше не является сырым адресом, но вместо этого представляет собой структуру такого вида:

Рис. 3.2.3. Структура селектора сегмента

Существует два типа дескрипторных таблиц: глобальная (GDT) и локальная (LDT). Глобальная таблица описывает сегменты операционной системы и разделяемых структур данных, у каждого ядра своя. Локальная таблица может быть определена для каждой конкретной задачи (процесса). Бит TI равен 0 для GDT и 1 для LDT. Индекс задаёт номер дескриптора в таблице дескрипторов сегмента. Поле RPL расшифровывается как Requested Privilege Level.

Сама таблица представляет собой просто массив, содержащий 8-байтные записи (дескрипторы сегмента), где каждая запись описывает один сегмент и выглядит так:

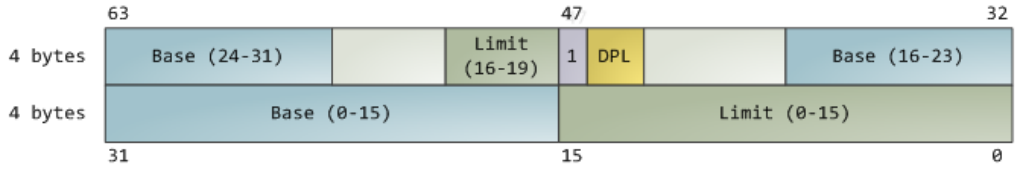

Рис. 3.2.4. Структура дескриптора сегмента

Помимо базового адреса сегмента дескрипторы содержат размер сегмента (точнее, максимально доступное смещение) и различные атрибуты сегментов, использующиеся для защиты памяти и определения прав доступа к сегменту для различных программных модулей. Базовый адрес представляет собой 32-битный линейный адрес, указывающий на начало сегмента, а лимит определяет, насколько большой сегмент. Добавление базового адреса к адресу логической памяти дает линейный адрес (никакого умножения на 16 уже нет). DPL (Descriptor Privilege Level) — уровень привилегий дескриптора; это число от 0 (наиболее привилегированный, режим ядра) до 3 (наименее привилегированный, пользовательский режим), которое контролирует доступ к сегменту.

Когда CPU находится в 32-битных режимах, регистры и инструкции могут в любом случае адресовать всё линейное адресное пространство. Итак, почему бы не установить базовый адрес в ноль и позволить логическим адресам совпадать с линейными адресами? Intel называет это «плоской моделью», и это именно то, что делают современные ядра операционных систем под x86. Это эквивалентно отключению сегментации.

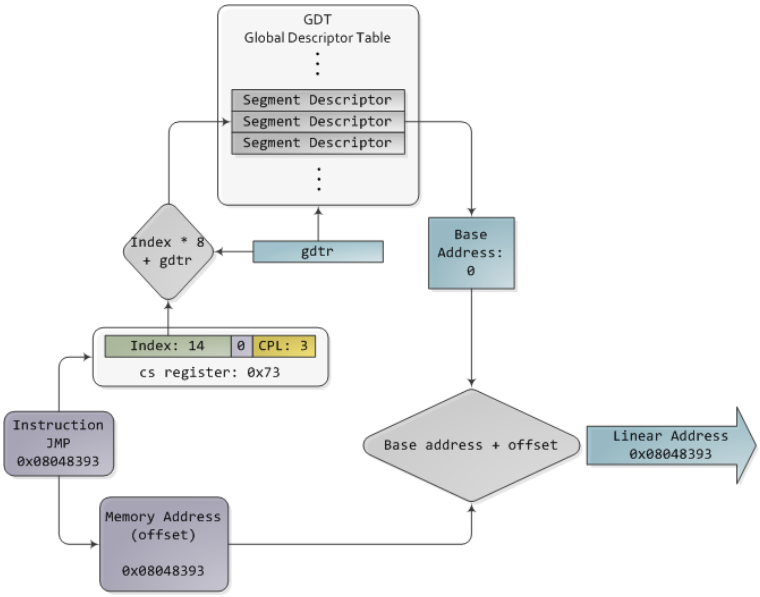

Рис. 3.2.5. Вычисление линейного адреса в защищенном режиме

Понятно, что раз таблицы GDT и LDT лежат в памяти, каждый раз ходить в них за базовым адресом долго. Поэтому сегментные дескрипторы кешируются в специальных регистрах в момент загрузки (в тот момент, когда происходит запись в сегментный селектор).

Местоположение GDT в памяти указывается процессору посредством инструкции **`lgdt`**.

Длинный режим

На архитектуре x86-64 в длинном (64-битном) режиме сегментация не используется. Для четырёх сегментных регистров (CS, SS, DS и ES) базовый адрес принудительно выставляются в 0. Сегментные регистры FS и GS по-прежнему могут иметь ненулевой базовый адрес (но он стал 64-битным и может быть установлен через отдельные моделезависимые регистры (MSR)). Это позволяет ОС использовать их для служебных целей.

Например, Microsoft Windows на x86-64 использует GS для указания на Thread Environment Block, маленькую структуру для каждого потока, которая содержит информацию об обработке исключений, thread-local-переменных и прочих per-thread-сведений. Аналогично, ядро Linux использует GS-сегмент для хранения данных per-CPU.

Посмотрим на таблицы сегментных дескрипторов. Таблица LDT на самом деле вышла из употребления и сейчас не используется. В таблице GDT в современных системах есть как минимум пять записей:

- Null — первая ячейка не используется (сделано, чтобы нулевое значение селектора было зарезервированным [1]);
- Kernel Code;
- Kernel Data;
- User Code;
- User Data.

Кольца защиты

Уровни привилегий x86 — механизм, с помощью которого ОС и ЦП ограничивают возможности программ пользовательского режима.

Существует четыре уровня привилегий: от 0 (наиболее привилегированных) до 3 (наименее привилегированных). В любой момент времени процессор x86 работает на определенном уровне привилегий, который определяет, что код может и не может сделать. Эти уровни привилегий часто описываются как защитные кольца, причем самое внутреннее кольцо соответствует самым высоким привилегиям.

Большинство современных ОС на x86 используют только Ring 0 и Ring 3. Кольцо 1 изначально планировались для использования в драйверах. Некоторые гипервизоры используют кольцо 1 для гостевых операционных систем. Кольца 1–2 не могут выполнять привилегированные инструкции, но это единственное реальное ограничение; в противном случае они имеют такие же привилегии, как и кольцо 0.

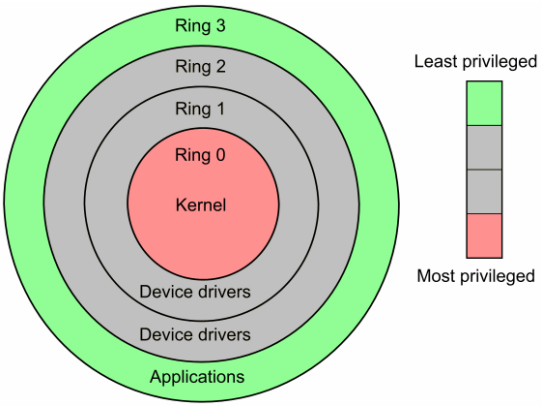

Рис. 3.1.6. Кольца защиты процессора x86

Кольца управляют доступом к памяти. Код с уровнем привилегий i может смотреть только данные уровня i и выше (менее привилегированных).

На кольце 0 можно делать всё. На кольце 3, например, нельзя:

изменить текущее кольцо защиты (иначе весь механизм был бы бесполезен);

- изменить таблицу страниц;
- зарегистрировать обработчик прерываний;
- выполнить ввод-вывод инструкциями in и out;
- ...

Текущий уровень привилегий (CPL) определяется сегментным дескриптором кода. Если сейчас исполняется сегмент кода с уровнем привилегий 3, значит, исполняется пользовательский код. Если исполняется код с уровнем привилегий 0 — исполняется код ядра.

При обращении к памяти проверяется неравенство

$$\max(\text{CPL}, \text{RPL}) \leq \text{DPL},$$

где

CPL — текущий уровень привилегий,

RPL — записан в сегментном регистре (селекторе),

DPL — записан в сегментном дескрипторе.

Если неравенство ложно, генерируется ошибка general protection fault (GPF).

Системные вызовы позволяют менять текущий уровень привилегий, поэтому эта операция достаточно тяжёлая.

В **ARM** вместо этого кольца называются **уровнями исключений** (Exception level), но основные идеи остаются прежними [2].

В ARMv8 существует 4 уровня исключений, которые обычно используются как:

EL0: область пользователя

EL1: ядро («супервизор» в терминологии ARM).

Для перехода на уровень EL1 служит инструкция **`svc`** (SuperVisor Call), ранее известной как **`swi`**, которая является инструкцией, используемой для выполнения системных вызовов Linux.

EL2: гипервизоры, например Xen.

Для перехода на уровень EL2 служит инструкция **`hvc`** (HyperVisor Call).

> *Гипервизор для ОС — то же самое, что ОС для пользовательской среды.*
>
> *Например, Xen позволяет одновременно запускать несколько операционных систем, таких как Linux или Windows, в одной системе и изолирует операционные системы друг от друга для обеспечения безопасности и простоты отладки, точно так же, как Linux делает это для пользовательских программ.*
>
> *Гипервизоры являются ключевой частью современной облачной инфраструктуры: они позволяют нескольким серверам работать на одном оборудовании, сохраняя использование оборудования всегда близким к 100% и экономя много денег.*
>
> *Например, AWS использовала Xen до 2017 года, когда о его переходе на KVM стало известно в новостях.*

EL3: еще один уровень.

Для перехода на уровень EL3 служит инструкции **`smc`** (Secure Mode Call).

В эталонной модели архитектуры ARMv8 уровни исключений описаны, как показано на рис. 3.2.7.

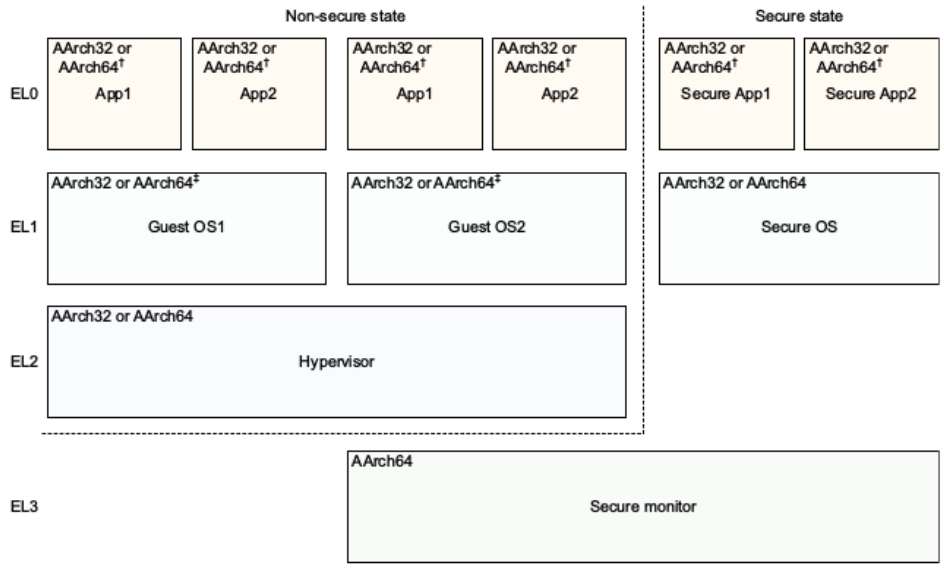

Рис.3.2.7. Уровни исключений согласно эталонной модели архитектуры ARMv8

Ситуация с ARM немного изменилась с появлением расширений хоста виртуализации ARMv8.1 (VHE, Virtualization Host Extensions). Это расширение позволяет ядру эффективно работать в EL2 (рис. 3.2.8)

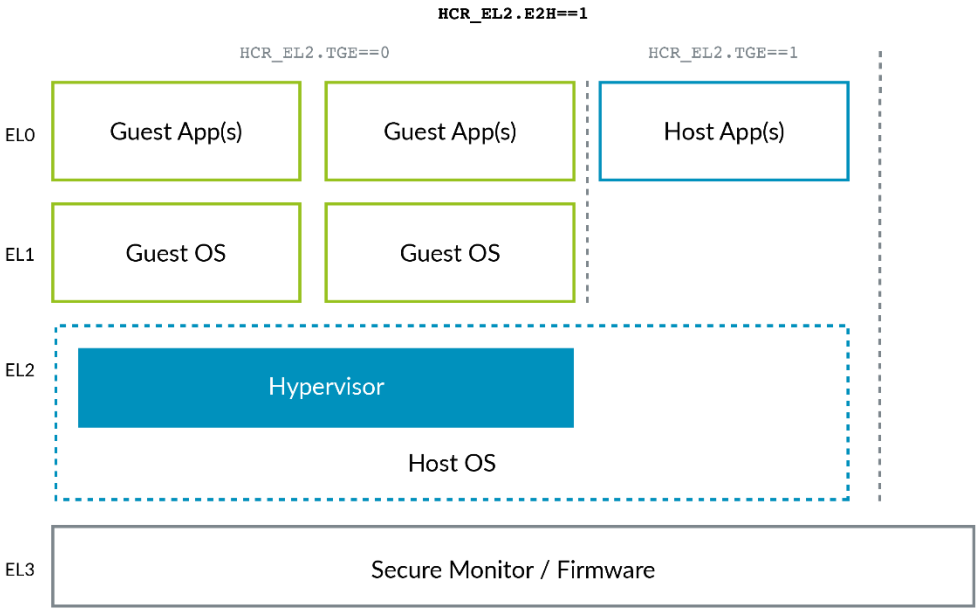

Рис.3.2.8. Уровни исключений для Virtualization Host Extensions ARMv8.1

VHE был создан потому, что решения виртуализации в ядре Linux, такие как KVM, получили преимущество над Xen. Когда большинству клиентов нужны только виртуальные машины Linux, то KVM проще и потенциально более эффективен, чем Xen. Итак, теперь ядро Linux хоста выступает в этих случаях в качестве гипервизора.

На рис. 3.2.8.в мы видим, что когда бит E2H регистра HCR_EL2 равен 1, VHE включен, и:

- ядро Linux работает в EL2 вместо EL1
- когда HCR_EL2.TGE == 1, мы являемся обычной программой пользовательского пространства хоста. Использование sudo может уничтожить хост, как обычно.
- когда HCR_EL2.TGE == 0, мы являемся гостевой ОС (например, когда вы запускаете ОС Ubuntu внутри QEMU KVM внутри хоста Ubuntu. Выполнение sudo не может уничтожить хост, если нет ошибки ядра QEMU/хоста.

Кольца с отрицательными номерами

Итак, первоначальная цель кольца 0–3 заключалась в том, чтобы изолировать привилегии между кодом пользовательского режима и ядром и остановить перемещение кода пользовательского режима по структурам управления системой [3].

Затем виртуализация стала популярной в x86, и Intel/AMD решили добавить для нее аппаратную поддержку. Для этого требуется фрагмент кода супервизора (гипервизора) для настройки некоторых структур управления (называемых VMCS), определяющих виртуальные машины, а затем вызова `vmenter` и обработки `vmexit`, то есть условий, при которых виртуальной машине требуется помощь со стороны гипервизора.

Этот фрагмент кода называется «**кольцо -1**». Такого фактического уровня привилегий не существует, но поскольку на нем могут размещаться несколько ядер, каждое из которых считает, что у них есть доступ к системе по кольцу 0, это имеет смысл.

Режим управления системой (System Management Mode) — еще один зверь со специальными инструкциями. Прошивка (ваш BIOS) настраивает обработчик SMM для обработки прерываний управления системой в зависимости от того, о чем микропрограмма хочет получать уведомления. При срабатывании этих событий работа ОС (или даже гипервизора) приостанавливается и вводится специальное адресное пространство. Эта область должна быть невидима для самой ОС при выполнении на том же процессоре. Отсюда «**кольцо-2**», поскольку оно более привилегировано, чем гипервизор.

Вы также услышите упоминание «**кольца-3**» здесь и там в отношении Intel ME или AMD PSP. Это второй процессор с отдельной прошивкой (полагаю, Intel использует процессоры ARC SoC), способный делать с основной системой все, что угодно. Якобы это сделано для обеспечения IPMI/удаленного управления функциональностью аппаратного типа. Он может работать всякий раз, когда на оборудование подается питание, независимо от того, включена основная система или нет — его целью, как я уже сказал, будет включение основной системы.

С точки зрения безопасности, чем в более низкий круг вы попадете, тем более незаметным вы себя сможете сделать. Целью исследования Bluepill было сокрытие от ОС того факта, что она действительно работает на виртуальной машине. Позже были проведены исследования устойчивости SMM. Например, сохранение SMM потенциально позволит вам переустановить вредоносное ПО даже после полной очистки жесткого диска и повторной установки. Intel ME потенциально открывает постоянно работающий сетевой чип для установки вредоносного ПО на основную цель.

Здесь мы рассматривали в основном чипы Intel, но вы должны знать, что другие платформы работают по-другому. Например, чипы ARM, среди прочего, имеют режимы «супервизор» и «пользователь».

Обратите внимание, что ARM, возможно, благодаря ретроспективе, имеет лучшее соглашение об именах уровней привилегий, чем x86, без необходимости использования отрицательных уровней: 0 — самый низкий, а 3 — самый высокий. Более высокие уровни, как правило, создаются чаще, чем более низкие.

3.2.2. Структура адресного пространства виртуальной памяти

Как уже упоминалось, процессоры архитектуры x86-64 поддерживают два основных режима работы: Long mode («длинный» режим) и Legacy mode («унаследованный», режим совместимости с 32-битным x86) [1]. На рис. 3.2.9 показано, в каких процессорах впервые появился каждый режим).

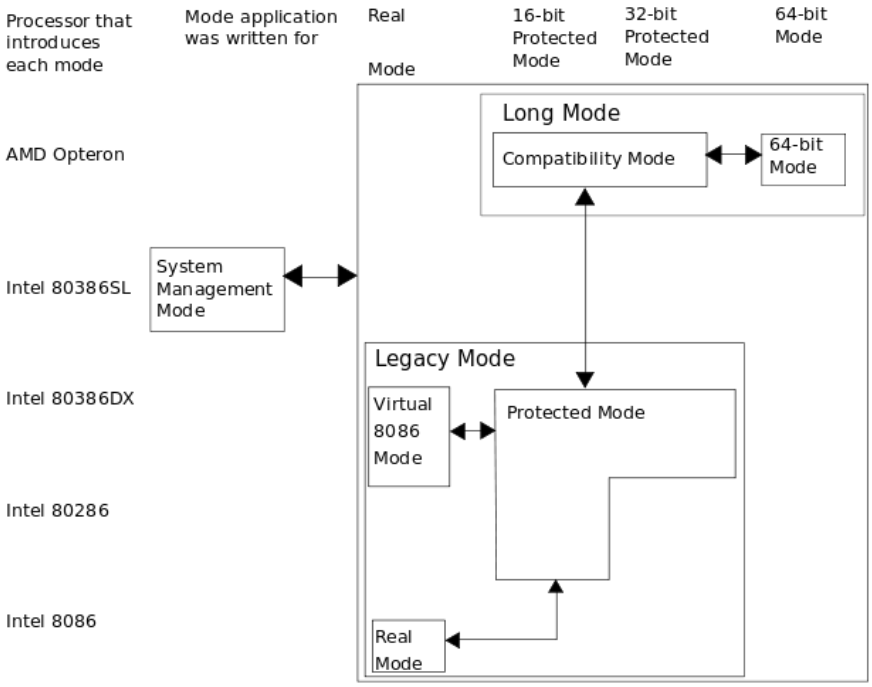

Рис.3.2.9. Доступность длинного и унаследованного режимов в разных процессорах

«Длинный» режим — «родной» для процессоров x86-64. Этот режим даёт возможность воспользоваться всеми дополнительными преимуществами, предоставляемыми архитектурой. Для использования этого режима необходима 64-битная операционная система.

Этот режим позволяет выполнять 64-битные программы; также (для обратной совместимости) предоставляется поддержка выполнения 32-битного кода, например, 32-битных приложений, хотя 32-битные программы не смогут использовать 64-битные системные библиотеки, и наоборот. Чтобы справиться с этой проблемой, большинство 64-разрядных операционных систем предоставляют два набора необходимых системных файлов: один — для родных 64-битных приложений, и другой — для 32-битных программ.

Когда вы используете 64-битную операционную систему (Windows, Linux или какую-либо другую), ваш процессор работает в длинном режиме. 32-битные ОС теряют популярность и используются всё реже, так как не позволяют использовать весь потенциал современного аппаратного обеспечения.

> *Так, дистрибутив Ubuntu Linux уже начиная с версии 17.10 не выпускается в 32-битном исполнении. Windows Server 2008 стала последней серверной ОС от Microsoft, которая имела 32-битную версию, и Windows Server 2008 R2 и все последующие существует только в 64-битной версии. Минимальное требование для Windows 11 — двухъядерный 64-битный процессор.*
>
> *Следует отметить, что Intel и AMD не производили 32-битный процессор более 19 лет. Процессор Intel Pentium 4 2,8 ГГц был последним 32-разрядным процессором. Эти машины могли работать только с 32-битными операционными системами, такими как Windows 95, 98 и XP. [4]*

Данный «унаследованный» режим позволяет процессору выполнять инструкции, рассчитанные для процессоров x86, и предоставляет полную совместимость с 32-битным кодом и операционными системами. В этом режиме процессор ведёт себя точно так же, как x86-процессор, например, Athlon или Pentium III, и дополнительные функции, предоставляемые архитектурой x86-64 (например, дополнительные регистры),

недоступны. В этом режиме 64-битные программы и операционные системы работать не будут.

Этот режим включает в себя подрежимы:

- Реальный режим (real mode)
- Защищённый режим (protected mode)
- Режим виртуального 8086 (virtual 8086 mode)

Реальный режим использовался в MS-DOS, в реальном режиме выполнялся код BIOS при загрузке компьютера.

Защищённый режим используется в 32-битных версиях современных многозадачных операционных систем (например, обычная 32-битная Windows XP работает в защищённом режиме, как и 32-битная версия Ubuntu 16.04).

Режим виртуального 8086 — подрежим защищённого, предназначался главным образом для создания т. н. «виртуальных DOS-машин». Если из 32-битной версии Windows вы запускаете 16-битное DOS-приложение, то работает эмулятор NTVDM (NT Virtual DOS Machine), который использует этот режим процессора. Другой эмулятор, DOSBox, не использует этот режим V86, а выполняет полную эмуляцию. Заметим, что в 64-битных версиях Windows эмулятор NTVDM был исключён, поэтому напрямую запустить на выполнение 16-битный com- или exe-файл стало невозможно (тем не менее, можно использовать тот же DOSBox или другой гипервизор для полной эмуляции реального режима).

Из длинного режима нельзя перейти в реальный или режим виртуального 8086 без перезагрузки. Поэтому, как уже отмечено, в 64-битных версиях Windows не работает NTVDM и нельзя запускать 16-битные программы.

Самый современный процессор x86-64 полностью поддерживает реальный режим. Если загрузка выполняется через BIOS, то код загрузчика (из сектора #0) исполняется в реальном режиме. Однако если вместо BIOS используется UEFI, то переход в Long mode происходит ещё раньше, и никакого кода в реальном режиме уже не выполняется. Можно считать, что современный компьютер сразу начинает работать в 64-битном длинном режиме.

Страничная организация памяти

Страничная память — способ организации виртуальной памяти, при котором виртуальные адреса отображаются на физические постранично.

В семействе x86 поддержка появилась с поколения 386, оно же первое 32-битное поколение.

Если сегментация сейчас практически не используется, то таблицы страниц, наоборот, используются вовсю во всех современных операционных системах. Его важно понимать, так как с особенностями страничной организации можно прямо или косвенно столкнуться при решении прикладных задач.

Страницы

Виртуальная память делится на страницы. Размер размера страницы задается процессором и обычно на x86-64 составляет 4 KiB. Это означает, что управление памятью в ядре выполняется с точностью до страницы. Когда вам понадобится новая память, ядро предоставит вам одну или несколько страниц. При освобождении памяти вы вернёте одну или несколько страниц... Каждый более гранулярный API (например, **`malloc`**) реализуется в пространстве пользователя.

Физическая память также поделена на страницы.

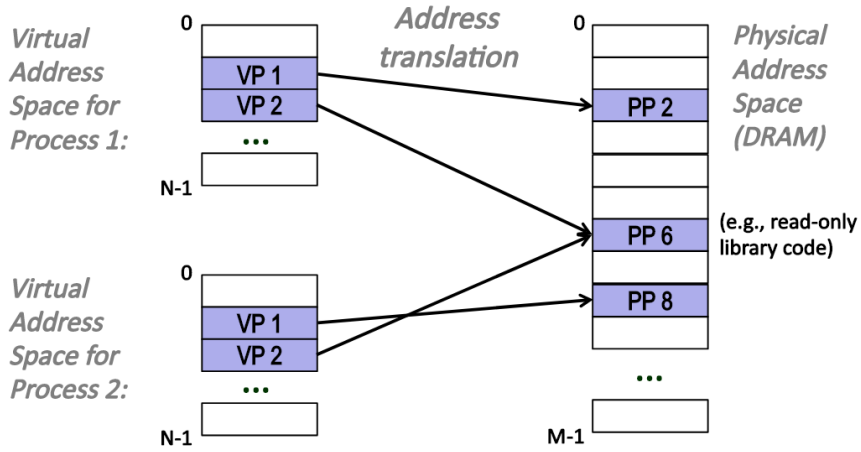

Рис. 3.2.10. Преобразование виртуальных адресов в физические

Хотя виртуальные адреса имеют разрядность в 64 бита, текущие реализации (и все чипы, которые находятся на стадии проектирования) не позволяют использовать всё виртуальное адресное пространство из 2^{64} байт (16 экзабайт). Это будет примерно в четыре миллиарда раз больше виртуального адресного пространства на 32-битных машинах. В обозримом будущем большинству операционных систем и приложений не потребуется такое большое адресное пространство, поэтому внедрение таких широких виртуальных адресов просто увеличит сложность и расходы на трансляцию адреса без реальной выгоды. Поэтому AMD решила, что в первых реализациях архитектуры фактически при трансляции адресов будут использоваться только младшие 48 бит виртуального адреса.

Кроме того, спецификация AMD требует, что старшие 16 бит любого виртуального адреса, биты с 48-го по 63-й, должны быть копиями бита 47 (по принципу sign extension). Если это требование не выполняется, процессор будет вызывать исключение. Адреса, соответствующие этому правилу, называются «канонической формой» (рис. 3.2.11). Канонические адреса в общей сложности составляют 256 терабайт полезного виртуального адресного пространства. Это по-прежнему в 65536 раз больше, чем 4 ГБ виртуального адресного пространства 32-битных машин.

Рис. 3.2.11. Канонические адреса в адресном пространстве

Это соглашение допускает при необходимости масштабируемость до истинной 64-разрядной адресации. Многие операционные системы (включая семейство Windows NT и GNU/Linux) берут себе старшую половину адресного пространства (пространство ядра) и оставляют младшую половину (пользовательское пространство) для кода приложения, стека пользовательского режима, кучи и других областей данных. Конструкция «канонического адреса» гарантирует, что каждая совместимая с AMD64 реализация имеет, по сути, две половины памяти: нижняя половина «растет вверх» по мере того, как

становится доступнее больше виртуальных битов адреса, а верхняя половина — наоборот, вверху адресного пространства и растет вниз.

Первые версии Windows для x64 даже не использовали все 256 ТБ; они были ограничены только 8 ТБ пользовательского пространства и 8 ТБ пространства ядра. Всё 48-битное адресное пространство стало поддерживаться в Windows 8.1, которая была выпущена в октябре 2013 года.

Структура таблицы страниц

Ставится задача транслировать 48-битный виртуальный адрес в физический. Она решается аппаратным обеспечением — блоком управления памятью (memory management unit, MMU). Этот блок является частью процессора. Чтобы транслировать адреса, он использует структуры данных в оперативной памяти, называемые таблицами страниц.

Вместо двухуровневой системы таблиц страниц, используемой системами с 32-битной архитектурой x86, системы, работающие в длинном режиме, используют четыре уровня таблицы страниц.

Возможные размеры страниц:

- 4 КБ (2^{12} байт) — наиболее часто используется (как и в x86)
- 2 МБ (2^{21} байт)
- 1 ГБ (2^{30} байт)

Пусть для определённости размер страницы равен 4 КБ. Значит, младшие 12 битов адреса кодируют смещение внутри страницы и не изменяются, а старшие биты используются для определения адреса начала страницы.

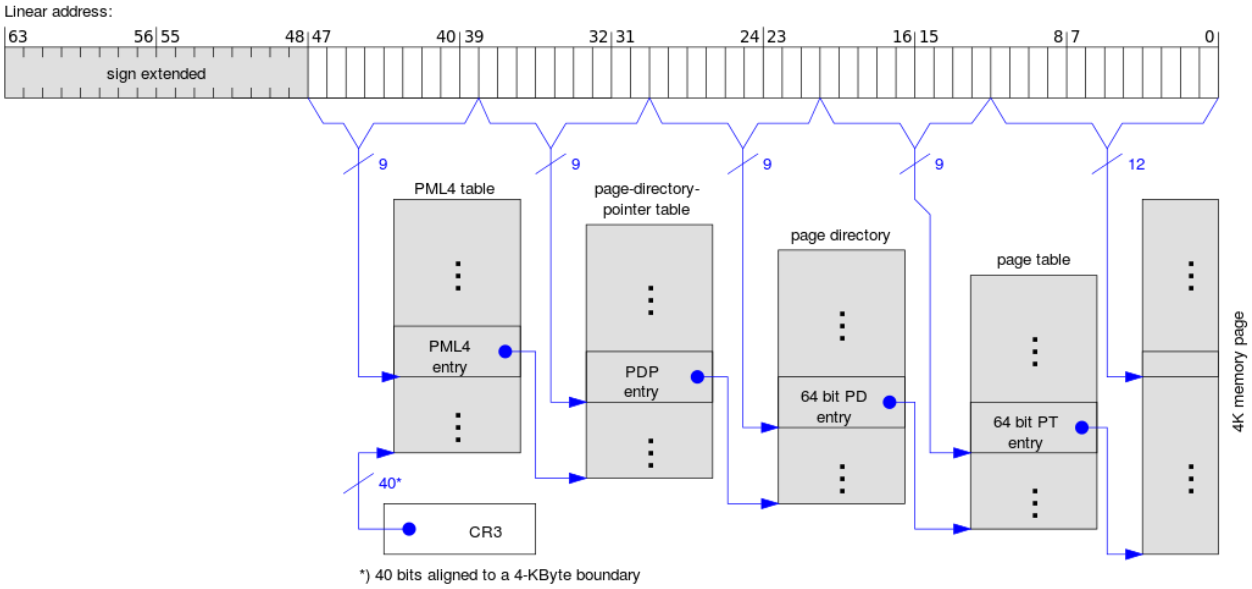

Рис. 3.2.12. четыре уровня таблицы страниц в длинном режиме

CR3 — это специальный регистр процессора. В записях каждой таблицы лежит физический адрес начала таблицы следующего уровня.

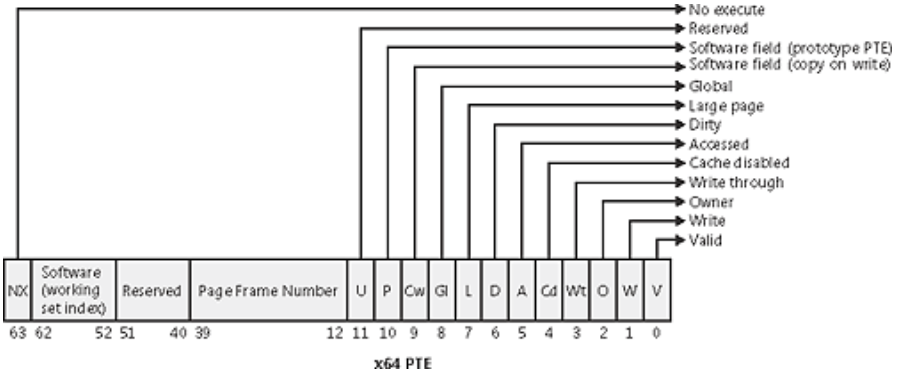

Рис. 3.2.13. Структура записи таблицы страниц

Полная иерархия сопоставления страниц размером 4 КБ для всего 48-битного пространства займет немногим больше 512 ГБ ОЗУ (около 0.195% от виртуального пространства 256 ТБ).

Кеширование

Таблицы страниц хранятся в оперативной памяти. Если при каждом обращении по виртуальному адресу выполнять полностью трансляцию адресов, это будет работать очень медленно. Поэтому в процессоре реализуется специальный кеш под названием «буфер ассоциативной трансляции» (Translation lookaside buffer, TLB).

На практике вероятность промаха TLB невысока и составляет в среднем от 0,01% до 1%.

Управление памятью в Linux

У каждого процесса в системе Linux есть адресное пространство, состоящее из трех логических сегментов: текста, данных и стека. Пример адресного пространства процесса изображен на рис. 3.2.14 (процесс A). Текстовый сегмент (text segment) содержит машинные команды, образующие исполняемый код программы. Он создается компилятором и ассемблером при трансляции программы (написанной на языке высокого уровня, например, C или C++) в машинный код. Как правило, текстовый сегмент доступен только для чтения. Таким образом, не изменяются ни размеры, ни содержание текстового сегмента.

Сегмент данных (data segment) содержит переменные, строки, массивы и другие данные программы. Он состоит из двух частей: инициализированных и неинициализированных данных. По историческим причинам вторая часть называется BSS (Block Started by Symbol). Инициализированная часть сегмента данных содержит переменные и константы компилятора, значения которых должны быть заданы при запуске программы. Все переменные в BSS должны быть инициализированы в нуль после загрузки.

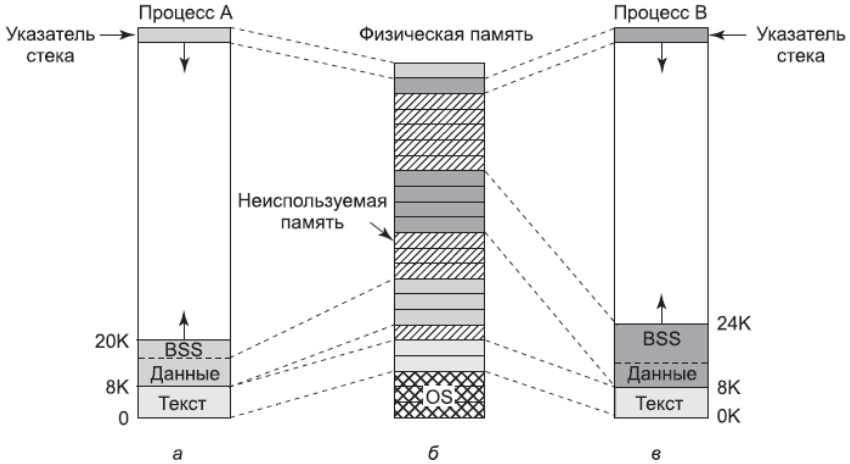

Рис. 3.2.14. а — виртуальное адресное пространство процесса A; б — физическая память; в — виртуальное адресное пространство процесса B

В отличие от текстового сегмента, который не может изменяться, сегмент данных изменяться может. Программы все время модифицируют свои переменные. Более того, многим программам требуется динамическое выделение памяти во время выполнения. Для этого операционная система Linux разрешает сегменту данных расти при выделении памяти и уменьшаться при освобождении памяти. Программа может установить размер своего сегмента данных при помощи системного вызова **brk**. Таким образом, чтобы выделить больше памяти, программа может увеличить размер своего сегмента данных. Этим системным вызовом активно пользуется библиотечная процедура **malloc** языка С, используемая для выделения памяти. Дескриптор адресного пространства процесса содержит информацию о диапазоне динамически выделенных областей памяти процесса (который обычно называется кучей — heap).

Третий сегмент — это сегмент стека (stack segment). На большинстве компьютеров он начинается около старших адресов виртуального адресного пространства и растет вниз к 0. Например, на 32-битной платформе x86 стек начинается с адреса 0xC0000000, который соответствует предельному виртуальному адресу, видимому процессам пользовательского режима. Если указатель стека оказывается ниже нижней границы сегмента стека, то происходит аппаратное прерывание, при котором операционная система понижает границу сегмента стека на одну страницу. Программы не управляют явно размером сегмента стека.

Когда программа запускается, ее стек не пуст. Напротив, он содержит все переменные окружения (оболочки), а также командную строку, введенную в оболочке для вызова этой программы. Таким образом, программа может узнать параметры, с которыми она была запущена.

Когда два пользователя запускают одну и ту же программу (например, текстовый редактор), то в памяти можно было бы хранить две копии программы редактора. Однако такой подход неэффективен. Вместо этого большинством систем Linux поддерживаются текстовые сегменты совместного использования (shared text segments). На рис. 3.2.14, *а* и *в* мы видим два процесса, A и B, совместно использующие общий текстовый сегмент. На рис. 3.2.14, *б* мы видим возможную компоновку физической памяти, где оба процесса совместно используют один и тот же фрагмент текста. Отображение выполняется аппаратным обеспечением виртуальной памяти.

Сегменты данных и стека никогда не бывают общими, кроме как после выполнения системного вызова **fork**, и то только те страницы, которые не модифицируются. Если размер одного из сегментов должен быть увеличен, то отсутствие свободного места в соседних страницах памяти не является проблемой, поскольку соседние виртуальные страницы памяти не обязаны отображаться на соседние физические страницы.

В дополнение к динамическому выделению памяти процессы в Linux могут обращаться к данным файлов при помощи отображения файлов на адресное пространство памяти (memory-mapped files). Эта функция позволяет отображать файл на часть адресного пространства процесса, чтобы можно было читать из файла и писать в файл так, как если бы это был массив байтов, хранящийся в памяти. Отображение файла на адресное пространство памяти делает произвольный доступ к нему существенно более легким, нежели при использовании таких системных вызовов, как **read** и **write**. Совместный доступ к библиотекам предоставляется именно при помощи этого механизма. На рис. 3.2.15 показан файл, одновременно отображенный на адресные пространства двух процессов по различным виртуальным адресам.

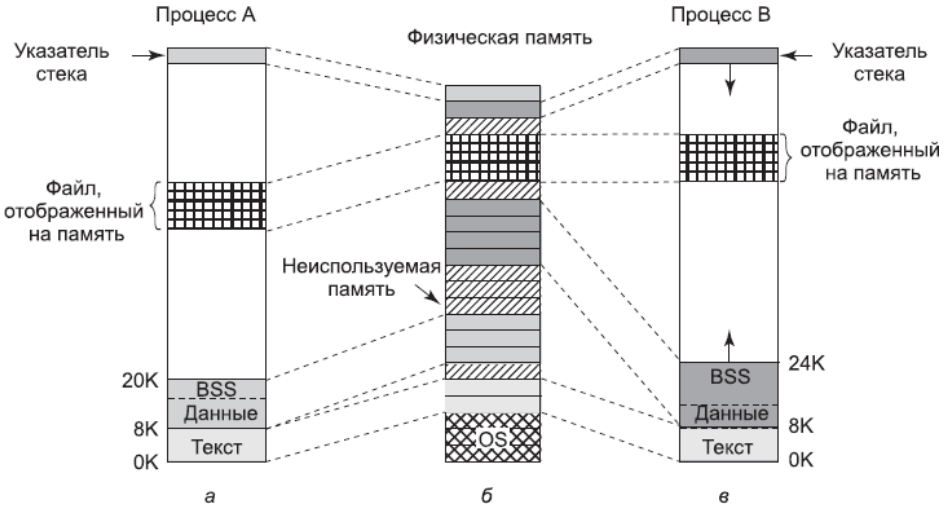

Рис. 3.2.15. Два процесса совместно используют один отображенный на память файл

Дополнительное преимущество отображения файла на память заключается в том, что два или более процесса могут одновременно отобразить на свое адресное пространство один и тот же файл. Запись в этот файл одним из процессов мгновенно становится видимой всем остальным. Таким образом, отображение на адресное пространство памяти временного файла (который будет удален после завершения работы процессов) представляет собой механизм реализации общей памяти (с высокой пропускной способностью) для нескольких процессов. В предельном случае два или более процесса могут отобразить на память файл, покрывающий все адресное пространство, получая тем самым такую форму совместного использования памяти, которая является чем-то средним между процессами и потоками. В этом случае (как и у потоков) все адресное пространство используется совместно, но каждый процесс обслуживает, например, свои собственные открытые файлы и сигналы, что отличает этот вариант от потоков. Однако на практике такой способ никогда не применяется.

Виртуальное адресное пространство делится на однородные, непрерывные и выровненные по границам страниц области. То есть каждая область состоит из участка смежных страниц с одинаковой защитой и страничной организацией. Текстовый сегмент и отображенные файлы являются примерами таких областей. Между областями виртуального адресного пространства могут быть дыры. Любая ссылка на дыру приводит к фатальной страничной ошибке. Размер страницы фиксирован: например, для Pentium он равен 4 Кбайт, а для Alpha — 8 Кбайт. Начиная с Pentium была добавлена поддержка страничных блоков размером 4 Мбайт. На последних 64-разрядных архитектурах Linux умеет поддерживать большие страницы (huge pages) размером по 2 Мбайт или 1 Гбайт каждая. Кроме того, в режиме расширения физических адресов (Physical Address Extension (PAE)), который используется на некоторых 32-битных архитектурах для увеличения адресного пространства процессов сверх 4 Гбайт, поддерживается также размер страниц 2 Мбайт.

Управление памятью в Windows

В Windows каждый пользовательский процесс имеет собственное виртуальное адресное пространство [5]. Для компьютеров x86 виртуальные адреса имеют длину 32 бита, так что каждый процесс имеет 4 Гбайт виртуального адресного пространства, по 2 Гбайта пользователю и ядру. На машинах x64 и пользователь и ядро получают больше виртуальных адресов, чем они смогут разумно использовать в обозримом будущем. И в компьютерах x86, и в компьютерах x64 виртуальное адресное пространство имеет замещение страниц по требованию со страницей фиксированного размера — 4 Кбайт, но в некоторых случаях, как мы скоро увидим, применяются также большие страницы

размером по 2 Мбайт (за счет использования каталога страниц и обхода соответствующей таблицы страниц).

Виртуальное адресное пространство для трех пользовательских процессов на компьютере x86 показано в упрощенной форме на рис. 3.2.16. Нижние и верхние 64 КБ виртуального адресного пространства каждого процесса обычно никуда не отображаются. Так было сделано специально, чтобы помочь отлавливать программные ошибки и смягчить уязвимости определенного типа.

Рис. 3.2.16. Виртуальное адресное пространство для трех пользовательских процессов на компьютере x86. Белым цветом показаны закрытые области каждого процесса. Заштрихованные области являются общими для всех процессов

С отметки 64 Кбайт начинаются пользовательские закрытые код и данные. Эта область простирается почти до 2 Гбайт. Верхние 2 Гбайт содержат операционную систему (включая код, данные, а также резидентный и нерезидентный пулы). Верхние 2 Гбайт — это виртуальная память ядра, которая совместно используется всеми пользовательскими процессами (кроме данных виртуальной памяти, таких как таблицы страниц и списки рабочих наборов, которые у каждого процесса свои). Виртуальная память ядра доступна только из режима ядра. Причина совместного использования виртуальной памяти процессов ядром состоит в том, что когда поток делает системный вызов, то он захватывается в режим ядра и может продолжать выполнение без изменения карты памяти. Все, что нужно сделать, — это переключиться на стек ядра потока.

Страница виртуальных адресов может быть в одном из трех состояний: недействительная, зарезервированная или зафиксированная. Недействительная страница (invalid page) не отображается на объект раздела памяти, и ссылка на нее вызывает страничную ошибку, которая приводит к нарушению доступа. После того как код или данные отображаются на виртуальную страницу, страница называется зафиксированной (commited). Страничная ошибка на зафиксированной странице приводит к отображению страницы с вызвавшим ошибку виртуальным адресом на одну из страниц, представленных объектом сегмента или сохраненных в файле подкачки. Часто для этого необходимо бывает выделить физическую страницу, а также выполнить ввод-вывод для файла, представленного объектом сегмента (чтобы прочитать данные с диска). Однако страничные ошибки могут возникать и потому, что нужно обновить элемент таблицы страниц, поскольку та физическая страница памяти, на которую он ссылается, все еще кэширована в памяти (в этом случае ввод-вывод не нужен). Это называется мягкой ошибкой (soft faults).

Виртуальная страница может быть также в зарезервированном (reserved) состоянии. Зарезервированная виртуальная страница недействительна, но эти виртуальные адреса никогда не будут выделяться диспетчером памяти для других целей. Например, когда создается новый поток, то многие страницы пространства стека пользовательского режима резервируются в пространстве виртуальных адресов процесса, но только одна страница фиксируется. По мере роста стека диспетчер виртуальной памяти будет автоматически фиксировать дополнительные страницы (до тех пор, пока зарезервированный объем практически истощится). Зарезервированные страницы действуют как страницы защиты — они предохраняют от слишком большого роста стека и перезаписи данных других процессов. Резервирование всех виртуальных страниц означает, что стек может в итоге разрастись до максимального размера (не рискуя, что некоторые необходимые для стека последовательные страницы виртуального адресного пространства могут быть отданы для других целей). Кроме атрибутов «недействительная», «зарезервированная» и «зафиксированная» страницы имеют и другие атрибуты, такие как «читаемая», «записываемая» и «исполняемая».

Интерфейс прикладного программирования Win32 имеет несколько функций, которые позволяют процессу явно управлять своей виртуальной памятью. Самые важные из этих функций перечислены в табл. 3.2.1.

Таблица 3.2.1. Основные функции Win32 для управления виртуальной памятью в Windows

Функция Win32	**Описание**
VirtualAlloc	Зарезервировать или зафиксировать область
VirtualFree	Освободить или отменить фиксирование области
VirtualProtect	Изменить защиту (чтение/запись/выполнение) для области
VirtualQuery	Сделать запрос о статусе области
VirtualLock	Сделать область резидентной (отключить для нее подкачку)
VirtualUnlock	Сделать область подкачиваемой
CreateFileMapping	Создать отображающий файл объект и (необязательно) присвоить ему имя
MapViewOfFile	Отобразить файл (или его часть) на адресное пространство
UnmapViewOfFile	Удалить отображенный файл из адресного пространства
OpenFileMapping	Открыть ранее созданный объект отображения файла

Первые четыре функции API используются для выделения, освобождения, защиты и запроса областей виртуального адресного пространства. Выделенные области всегда начинаются на границе 64 Кбайт (для минимизации проблем при переносе на архитектуры будущего, страницы которых будут больше нынешних). Реальный размер выделенного адресного пространства может быть меньше 64 Кбайт, но он должен быть кратен размеру страницы. Следующие два вызова API дают процессу возможность зафиксировать страницы в памяти (чтобы они не вытеснялись) и отменить эту фиксацию. Такие страницы могут потребоваться программе реального времени для того, чтобы избежать страничных ошибок во время критичных операций. Операционная система устанавливает предел, для того чтобы процессы не становились слишком прожорливыми. Страницы могут удаляться из памяти, но только в случае вытеснения всего процесса. Когда он возвращается назад, то все заблокированные страницы загружаются снова (после чего потоки могут начинать работу). Windows имеет также функции собственного API, которые позволяют процессу обращаться к виртуальной памяти другого процесса, для которого у него есть описатель. Последние четыре функции API предназначены для

управления отображенными в память файлами. Для отображения файла необходимо сначала создать объект отображения файла при помощи `CreateFileMapping`. Эта функция возвращает описатель для объекта отображения файла (то есть объекта сегмента) и (необязательно) вводит его имя в пространство имен Win32, чтобы другие процессы также могли его использовать. Следующие две функции отображают и снимают отображение представлений для объектов сегментов (из виртуального адресного пространства процесса). Последний вызов API процесс может использовать для совместного использования отображения, которое было создано другим процессом (при помощи `CreateFileMapping`), — обычно это делается для отображения анонимной памяти. Таким образом, два или более процесса могут совместно использовать области своих адресных пространств. Эта методика позволяет им писать в некоторые области виртуальной памяти друг друга.

3.2.3. Задачи управления виртуальной памятью: задача размещения, задача перемещения, задача преобразования адресов, задача замещения

Память является важнейшим ресурсом, требующим тщательного управления со стороны мультипрограммной операционной системы. Особая роль памяти объясняется тем, что процессор может выполнять инструкции протравы только в том случае, если они находятся в памяти. Память распределяется как между модулями прикладных программ, так и между модулями самой операционной системы [6].

В ранних ОС управление памятью сводилось просто к загрузке программы и ее данных из некоторого внешнего накопителя (перфоленты, магнитной ленты или магнитного диска) в память. С появлением мультипрограммирования перед ОС были поставлены новые задачи, связанные с распределением имеющейся памяти между несколькими одновременно выполняющимися программами.

Современные компьютерные системы используют концепцию виртуальной памяти. Виртуальная память - это совокупность программно-аппаратных средств, позволяющих пользователям писать программы, размер которых превосходит имеющуюся оперативную память.

Функциями ОС по управлению памятью в мультипрограммной системе являются:

- отслеживание свободной и занятой памяти;
- выделение памяти процессам и освобождение памяти по завершении процессов;
- вытеснение кодов и данных процессов из оперативной памяти на диск (полное или частичное), когда размеры основной памяти не достаточны для размещения в ней всех процессов, и возвращение их в оперативную память, когда в ней освобождается место;
- настройка адресов программы на конкретную область физической памяти.

Помимо первоначального выделения памяти процессам при их создании ОС должна также заниматься динамическим распределением памяти, то есть выполнять запросы приложений на выделение им дополнительной памяти во время выполнения. После того как приложение перестает нуждаться в дополнительной памяти, оно может возвратить ее системе. Выделение памяти случайной длины в случайные моменты времени из общего пула памяти приводит к фрагментации и, вследствие этого, к неэффективному ее использованию. Дефрагментация памяти тоже является функцией операционной системы.

Во время работы операционной системы ей часто приходится создавать новые служебные информационные структуры, такие как описатели процессов и потоков, различные таблицы распределения ресурсов, буферы, используемые процессами для обмена данными, синхронизирующие объекты и т. п. Все эти системные объекты требуют памяти. В некоторых ОС заранее (во время установки) резервируется некоторый фиксированный

объем памяти для системных нужд. В других же ОС используется более гибкий подход, при котором память для системных целей выделяется динамически. В таком случае разные подсистемы ОС при создании своих таблиц, объектов, структур и т. п. обращаются к подсистеме управления памятью с запросами.

Защита памяти — это еще одна важная задача операционной системы, которая состоит в том, чтобы не позволить выполняемому процессу записывать или читать данные из памяти, назначенной другому процессу. Эта функция, как правило, реализуется программными модулями ОС в тесном взаимодействии с аппаратными средствами.

Виртуальная память решает следующие **задачи**:

- размещает данные в запоминающих устройствах разного типа, например, часть программы в оперативной памяти, а часть на диске;
- перемещает по мере необходимости данные между запоминающими устройствами разного типа, например, подгружает нужную часть программы с диска в оперативную память;
- преобразует виртуальные адреса в физические.

Все эти действия выполняются автоматически, без участия программиста, то есть механизм виртуальной памяти является прозрачным по отношению к пользователю.

Как уже обсуждалось ранее, наиболее распространенными реализациями виртуальной памяти является страничное, сегментное и странично-сегментное распределение памяти, а также свопинг (подкачка).

3.2.4. Подкачка

Файл подкачки или виртуальная память — это способ системы виртуальной памяти увеличить оперативную память, когда ее не хватает для совершения операций. Система автоматически задействует файл подкачки, когда приложениям не хватит системной памяти ОЗУ. Хотя система сама регулирует объем файла подкачки иногда может понадобиться вручную увеличить виртуальную память.

Подкачка в Linux

В ранних системах UNIX использовался процесс подкачки (swapper process), который перемещал процессы целиком между памятью и диском (когда все активные процессы не помещались в физической памяти). Linux (подобно другим современным версиям UNIX) больше не перемещает процессы целиком. Единицей управления памятью является страница, и почти все компоненты управления памятью работают с точностью до страниц. Подсистема подкачки также работает с точностью до страниц и тесно связана с алгоритмом Page Frame Reclaiming Algorithm.

Основная идея подкачки в Linux проста: процессу не обязательно находиться целиком в памяти для того, чтобы выполняться. Все, что нужно, — это пользовательская структура и таблицы страниц. Если они подкачаны в память, то процесс считается находящимся в памяти и может планироваться для выполнения. Страницы сегментов текста, данных и стека подкачиваются динамически (по одной) по мере появления ссылок на них. Если пользовательская структура и таблица страниц не находятся в памяти, то процесс не может выполняться до тех пор, пока процесс подкачки не доставит их в память.

Подкачка реализована частично ядром, а частично новым процессом, называемым демоном страниц (page daemon). Демон страниц — это процесс 2 (процесс 0 — это процесс idle, традиционно называемый своппером, а процесс 1 — это init). Как и все демоны, демон страниц работает периодически. После пробуждения он осматривается, есть ли для него работа. Если он видит, что количество страниц в списке свободных слишком мало, то он начинает освобождать страницы.

Операционная система Linux является системой с подкачкой страниц по требованию (без упреждающей подкачки) и без концепции рабочего набора (хотя в ней есть системный вызов для указания пользователем страницы, которая ему может скоро понадобиться). Текстовые сегменты и отображаемые на адресное пространство памяти файлы подгружаются из соответствующих им файлов на диске. Все остальное выгружается либо в раздел подкачки (если он присутствует), либо в один из файлов подкачки (фиксированной длины), которые называются областью подкачки (swap area). Файлы подкачки могут динамически добавляться и удаляться, и у каждого есть свой приоритет. Подкачка страниц из отдельного раздела диска, доступ к которому осуществляется как к отдельному устройству, не содержащему файловой системы, более эффективна, чем подкачка из файла, по нескольким причинам. Во-первых, не требуется отображение блоков файла в блоки диска. Во-вторых, физическая запись может иметь любой размер, а не только размер блока файла. В-третьих, страница всегда пишется на диск в виде единого непрерывного участка, а при записи в файл подкачки это может быть и не так.

Linux различает четыре разных типа страниц: **неиспользуемые** (unreclaimable), **подкачиваемые** (swappable), **синхронизируемые** (syncable) и **отбрасываемые** (discardable). Неиспользуемые страницы (которые включают зарезервированные или заблокированные страницы, стеки режима ядра и т. п.) не могут вытесняться в подкачку. Подкачиваемые страницы должны быть записаны обратно в область подкачки (или в раздел подкачки) перед тем, как их можно будет вновь использовать. Синхронизируемые страницы должны быть записаны на диск в том случае, если они были помечены как «грязные». И наконец, отбрасываемые страницы могут быть использованы немедленно.

Если доступной памяти в одной из зон становится меньше нижнего предела, то страничный демон kswapd инициирует алгоритм востребования страниц PFRA. При каждом выполнении PFRA сначала пытается востребовать легкодоступные страницы, после чего переходит к труднодоступным. Отбрасываемые страницы и страницы, на которые нет ссылок, могут быть востребованы немедленно, для этого их необходимо перенести в список свободных страниц зоны. Затем он ищет такие страницы с резервным хранением, на которые не было ссылок в последнее время (при помощи временного алгоритма). Затем следуют совместно используемые страницы, которые не используются активно пользователями. PFRA при выборе (в данной категории) старых страниц для вытеснения использует алгоритм, подобный алгоритму часов. В основе этого алгоритма лежит цикл, который сканирует список активных и неактивных страниц каждой зоны, пытаясь востребовать страницы различных типов (с разной срочностью). Значение срочности передается как параметр, который сообщает процедуре о том, сколько усилий нужно потратить для востребования страниц. Обычно он указывает, сколько страниц нужно обследовать до того, как прекратить работу.

Второй демон системы управления памятью pdflush фактически является набором фоновых потоков демона. Либо потоки pdflush пробуждаются периодически (обычно каждые 500 мс) для записи на диск очень старых «грязных» страниц, либо их явным образом будит ядро (когда уровень доступной памяти падает ниже определенного порога) для записи «грязных» страниц из кэша страниц на диск.

Подкачка в Windows

Интересный компромисс получается при назначении резервного хранилища в зафиксированных страницах, которые не имеют соответствия конкретным файлам. Эти страницы используют файл подкачки (pagefile). Вопрос состоит в том, как и когда отображать виртуальную страницу на конкретное местоположение в файле подкачки. Простая стратегия могла быть такой: надо назначить каждой виртуальной странице страницу в одном из файлов подкачки на диске (во время фиксации виртуальной

страницы). Это гарантирует, что всегда будет известно место для записи каждой фиксируемой страницы (если ее нужно будет удалить из памяти).

Windows использует синхронную (just-in-time) стратегию. Зафиксированным страницам (поддерживаемым файлом подкачки) не выделяется место в файле подкачки до того момента, когда их необходимо вытеснить в файл подкачки. Для тех страниц, которые никогда не вытесняются, дисковое пространство не выделяется. Если суммарная виртуальная память меньше, чем имеющаяся физическая память, то файл подкачки не нужен совсем.

Когда хранящиеся в файле подкачки страницы считываются в память, они сохраняют свое место в файле подкачки (до первой модификации). Если страница не модифицируется, она попадает в специальный список свободных физических страниц, называемый списком резервирования (standby list), из которого она может быть взята для повторного использования (без необходимости записывать ее обратно на диск). Если же она модифицируется, то диспетчер памяти освобождает страницу в файле подкачки и теперь единственный экземпляр страницы находится в памяти. Диспетчер памяти делает это путем маркировки страницы «только для чтения» (после ее загрузки). В первый раз, когда поток пытается записать в страницу, диспетчер памяти обнаружит эту ситуацию и освободит страницу в файле подкачки, обеспечит доступ на запись к странице, а затем даст потоку возможность повторить попытку.

Windows поддерживает до 16 файлов подкачки. Обычно они распределены по нескольким дискам (для повышения производительности ввода-вывода). Все файлы имеют некий начальный размер и максимальный размер (до которого они могут увеличиться при необходимости), однако лучше создавать эти файлы максимального размера (при инсталляции системы). Если позднее (когда диски уже будут более заполнены) возникнет необходимость увеличения файла подкачки, то, скорее всего, новое пространство файла подкачки будет иметь высокую степень фрагментации (что снижает производительность).

hiberfile.sys - файл для сохранения памяти в режиме «сон» (гибернация);

pagefile.sys - файл подкачки;

swapfile.sys - файл подкачки отдельных (предварительно скаченных из магазина приложений UWP) для быстрого применения (в случае надобности).

Гибернация - энергосберегающее состояние компьютера, предназначенное в первую очередь для ноутбуков. Если в режиме «Сна» данные о состоянии системы и программ хранятся в оперативной памяти, потребляющей энергию, то при гибернации эта информация сохраняется на системном жестком диске в скрытом фале **hiberfil.sys**, после чего ноутбук выключается. При включении, эти данные считываются, и вы можете продолжить работу с компьютером с того момента, на котором закончили.

Самый простой способ включения или отключения режима гибернации — использовать командную строку (потребуется запустить ее от имени администратора).

Чтобы отключить гибернацию, в командной строке введите powercfg -h off и нажмите Enter. Это отключит данный режим, удалит файл hiberfil.sys с жесткого диска, а также отключит опцию быстрого запуска Windows 10 (которая также задействует данную технологию и без гибернации не работает).

```
C:\>powercfg -h off
```

3.2.5. Алгоритмы замещения страниц

При возникновении ошибки отсутствия страницы операционная система должна выбрать выселяемую (удаляемую из памяти) страницу, чтобы освободить место для загружаемой страницы. Если предназначенная для удаления страница за время своего нахождения в памяти претерпела изменения, она должна быть переписана на диске, чтобы привести

дисковую копию в актуальное состояние. Но если страница не изменялась (например, она содержала текст программы), дисковая копия не утратила своей актуальности и перезапись не требуется. Тогда считываемая страница просто пишется поверх выселяемой [5].

Если бы при каждой ошибке отсутствия страницы можно было выбирать для выселения произвольную страницу, то производительность системы была бы намного выше, если бы выбор падал на редко запрашиваемую страницу. При удалении интенсивно используемой страницы высока вероятность того, что она в скором времени будет загружена опять, что приведет к лишним издержкам. На выработку алгоритмов замещения страниц было потрачено множество усилий как в теоретической, так и в экспериментальной областях. Далее мы рассмотрим некоторые из наиболее важных алгоритмов.

Во всех рассматриваемых далее алгоритмах замещения страниц ставится вполне определенный вопрос: когда возникает необходимость удаления страницы из памяти, должна ли эта страница быть одной из тех, что принадлежат процессу, в работе которого произошла ошибка отсутствия страницы, или это может быть страница, принадлежащая другому процессу? В первом случае мы четко ограничиваем каждый процесс фиксированным количеством используемых страниц, а во втором таких ограничений не накладываем. Возможны оба варианта, а к этому вопросу мы еще вернемся.

Оптимальный алгоритм замещения страниц

Наилучший алгоритм замещения страниц несложно описать, но совершенно невозможно реализовать. В нем все происходит следующим образом. На момент возникновения ошибки отсутствия страницы в памяти находится определенный набор страниц. К некоторым из этих страниц будет осуществляться обращение буквально из следующих команд (эти команды содержатся на странице). К другим страницам обращения может не быть и через 10, 100 или, возможно, даже 1000 команд. Каждая страница может быть помечена количеством команд, которые должны быть выполнены до первого обращения к странице.

Оптимальный алгоритм замещения страниц гласит, что должна быть удалена страница, имеющая пометку с наибольшим значением. Если какая-то страница не будет использоваться на протяжении 8 млн команд, а другая какая-нибудь страница не будет использоваться на протяжении 6 млн команд, то удаление первой из них приведет к ошибке отсутствия страницы, в результате которой она будет снова выбрана с диска в самом отдаленном будущем. Компьютеры, как и люди, пытаются по возможности максимально отсрочить неприятные события.

Единственной проблемой такого алгоритма является невозможность его реализации. К тому времени, когда произойдет ошибка отсутствия страницы, у операционной системы не будет способа узнать, когда каждая из страниц будет востребована в следующий раз. Далее мы будем рассматривать те алгоритмы, которые действительно полезны для реальных систем.

Использование битов состояния в алгоритмах исключения страниц

Чтобы позволить операционной системе осуществить сбор полезной статистики востребованности страниц, большинство компьютеров, использующих виртуальную память, имеют два бита состояния, **R** и **M**, связанных с каждой страницей. Бит **R** устанавливается при каждом обращении к странице (при чтении или записи). Бит **M** устанавливается, когда в страницу ведется запись (то есть когда она модифицируется). Эти биты присутствуют в каждой записи таблицы страниц. Важно усвоить, что эти биты должны обновляться при каждом обращении к памяти, поэтому необходимо, чтобы их

значения устанавливались аппаратной частью. После установки бита в 1 он сохраняет это значение до тех пор, пока не будет сброшен операционной системой.

Биты **R** и **M** могут использоваться для создания следующего простого алгоритма замещения страниц. При запуске процесса оба страничных бита для всех его страниц устанавливаются операционной системой в 0. Время от времени (например, при каждом прерывании по таймеру) бит R сбрасывается, чтобы отличить те страницы, к которым в последнее время не было обращений, от тех, к которым такие обращения были.

При возникновении ошибки отсутствия страницы операционная система просматривает все страницы и на основе текущих значений принадлежащих им битов **R** и **M** делит их на четыре категории:

1. Класс 0: в последнее время не было ни обращений, ни модификаций.
2. Класс 1: обращений в последнее время не было, но страница модифицирована.
3. Класс 2: в последнее время были обращения, но модификаций не было.
4. Класс 3: в последнее время были и обращения, и модификации.

Алгоритм исключения давно использовавшейся страницы

Алгоритм исключения давно использовавшейся страницы (Not Recently Used (NRU)) удаляет произвольную страницу, относящуюся к самому низкому непустому классу. В этот алгоритм заложена идея, суть которой в том, что лучше удалить модифицированную страницу, к которой не было обращений по крайней мере за последний такт системных часов (обычно это время составляет около 20 мс), чем удалить интенсивно используемую страницу. Главная привлекательность алгоритма NRU в том, что его нетрудно понять, сравнительно просто реализовать и добиться от него производительности, которая, конечно, не оптимальна, но может быть вполне приемлема.

Алгоритм «первой пришла, первой и ушла»

Другим низкозатратным алгоритмом замещения страниц является алгоритм FIFO (First In, First Out — «первым пришел, первым ушел»). Операционная система ведет список всех страниц, находящихся на данный момент в памяти, причем совсем недавно поступившие находятся в хвосте, поступившие раньше всех — в голове списка. При возникновении ошибки отсутствия страницы удаляется страница, находящаяся в голове списка, а к его хвосту добавляется новая страница. Принцип FIFO в чистом виде используется довольно редко.

Алгоритм «второй шанс»

Простой модификацией алгоритма FIFO, исключающей проблему удаления часто запрашиваемой страницы, может стать проверка бита R самой старой страницы. Если его значение равно нулю, значит, страница не только старая, но и невостребованная, поэтому она тут же удаляется. Если бит **R** имеет значение 1, он сбрасывается, а страница помещается в конец списка страниц и время ее загрузки обновляется, как будто она только что поступила в память. Затем поиск продолжается.

Действие этого алгоритма, названного «второй шанс», показано на рис. 3.2.17. Страницы с A по H содержатся в связанном списке отсортированными по времени их поступления в память.

Предположим, что ошибка отсутствия страницы возникла на отметке времени 20. Самой старой является страница A, время поступления которой соответствует началу процесса и равно 0. Если бит **R** для страницы A сброшен, страница либо удаляется из памяти с записью на диск (если она измененная), либо просто удаляется (если она неизмененная). Но если бит **R** установлен, то A помещается в конец списка и ее «время загрузки» переключается на текущее (20). Также при этом сбрасывается бит **R**. А поиск подходящей страницы продолжается со страницы B.

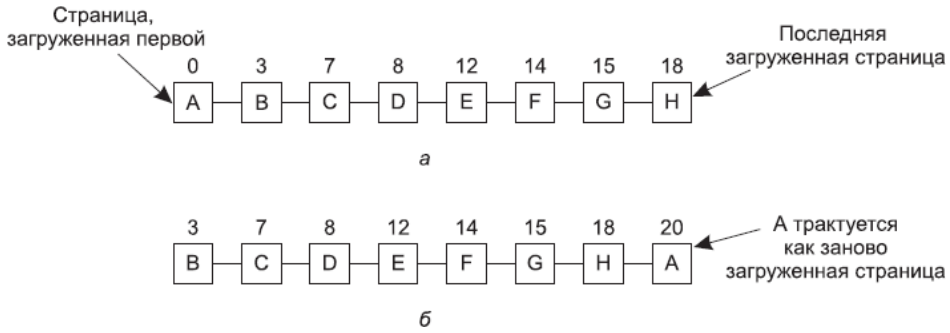

Рис. 3.2.17. Действие алгоритма «второй шанс»: а — страницы, отсортированные в порядке FIFO; б — список страниц при возникновении ошибки отсутствия страницы, показателе времени 20 и установленном в странице А бите R; числа над страницами — это время, когда они были загружены

Алгоритм «второй шанс» занимается поиском ранее загруженной в память страницы, к которой за только что прошедший интервал времени таймера не было обращений. Если обращения были ко всем страницам, то алгоритм «второй шанс» превращается в простой алгоритм FIFO. Представим, в частности, что у всех страниц на рис. 3.2.17, *а* бит **R** установлен. Операционная система поочередно перемещает страницы в конец списка, очищая бит R при каждом добавлении страницы к концу списка. В конце концов она возвращается к странице А, у которой бит **R** теперь уже сброшен. И тогда страница А выселяется. Таким образом, алгоритм всегда завершает свою работу.

Алгоритм «часы»

При всей своей логичности алгоритм «второй шанс» слишком неэффективен, поскольку он постоянно перемещает страницы в своем списке. Лучше содержать все страничные блоки в циклическом списке в виде часов (рис. 3.2.18). Стрелка указывает на самую старую страницу.

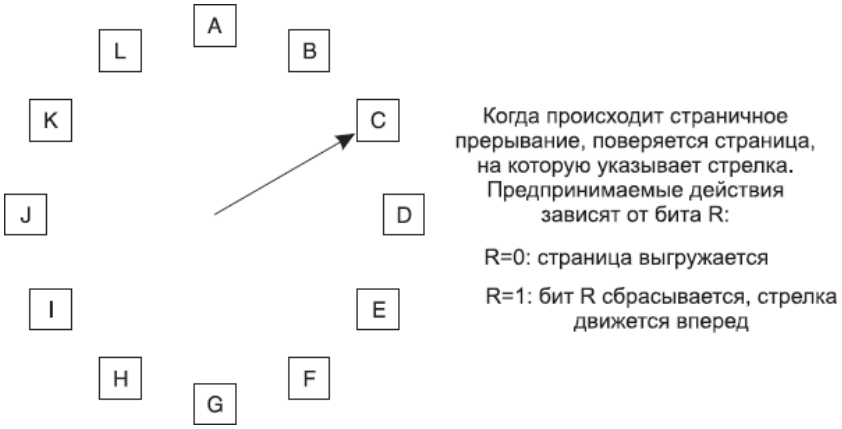

Рис. 3.2.18. Алгоритм «часы»

При возникновении ошибки отсутствия страницы проверяется та страница, на которую указывает стрелка. Если ее бит R имеет значение 0, страница выселяется, на ее место в «циферблате» вставляется новая страница и стрелка передвигается вперед на одну позицию. Если значение бита **R** равно 1, то он сбрасывается и стрелка перемещается на следующую страницу. Этот процесс повторяется до тех пор, пока не будет найдена страница с **R** = 0. Неудивительно, что этот алгоритм называется «часы».

Алгоритм замещения наименее востребованной страницы

В основе неплохого приближения к оптимальному алгоритму лежит наблюдение, что страницы, интенсивно используемые несколькими последними командами, будут, скорее всего, снова востребованы следующими несколькими командами. И наоборот, долгое

время не востребованные страницы наверняка еще долго так и останутся невостребованными. Эта мысль наталкивает на вполне реализуемый алгоритм: при возникновении ошибки отсутствия страницы нужно избавиться от той страницы, которая длительное время не была востребована. Эта стратегия называется замещением наименее востребованной страницы (Least Recently Used (LRU)).

Теоретически реализовать алгоритм LRU вполне возможно, но его практическая реализация дается нелегко. Для поиска страницы в списке, ее удаления из него и последующего перемещения этой страницы вперед потребуется довольно много времени, даже если это будет возложено на аппаратное обеспечение (если предположить, что такое оборудование можно создать). Так как вряд ли будет создано оборудование с такой функциональностью, нам понадобится решение, которое может быть реализовано в программном обеспечении.

Алгоритм нечастого востребования

Одно из возможных решений называется алгоритмом нечастого востребования (Not Frequently Used (NFU)). Для его реализации потребуется программный счетчик с начальным нулевым значением, связанный с каждой страницей. При каждом прерывании от таймера операционная система сканирует все находящиеся в памяти страницы. Для каждой страницы к счетчику добавляется значение бита R, равное 0 или 1. Счетчики позволяют приблизительно отследить частоту обращений к каждой странице. При возникновении ошибки отсутствия страницы для замещения выбирается та страница, чей счетчик имеет наименьшее значение.

Основная проблема при использовании алгоритма NFU заключается в том, что он никогда ничего не забывает. К примеру, при многопроходной компиляции те страницы, которые интенсивно использовались при первом проходе, могут сохранять высокие значения счетчиков и при последующих проходах. Фактически если на первый проход затрачивается больше времени, чем на все остальные проходы, то страницы, содержащие код для последующих проходов, могут всегда иметь более низкие показатели счетчиков, чем страницы, использовавшиеся при первом проходе. Вследствие этого операционная система будет замещать нужные страницы вместо тех, надобность в которых уже отпала.

Алгоритм старения

К счастью, небольшая модификация алгоритма NFU позволяет довольно близко подойти к имитации алгоритма LRU. Модификация состоит из двух частей. Во-первых, перед добавлением к счетчикам значения бита **R** их значение сдвигается на один разряд вправо. Во-вторых, значение бита **R** добавляется к самому левому, а не к самому правому биту.

На рис. 3.2.19 показано, как работает модифицированный алгоритм, известный как алгоритм старения. Предположим, что после первого прерывания от таймера бит **R** для страниц от 0-й до 5-й имеет значения соответственно 1, 0, 1, 0, 1 и 1 (для страницы 0 оно равно 1, для страницы 1 — 0, для страницы 2 — 1 и т. д.). Иными словами, между прерываниями от таймера, соответствующими тактам 0 и 1, было обращение к страницам 0, 2, 4 и 5, в результате которого их биты R были установлены в 1, а у остальных страниц их значение осталось равным 0. После того как были смещены значения шести соответствующих счетчиков и слева вставлено значение бита **R**, они приобрели значения, показанные на рис. 3.2.19, *а*. В четырех оставшихся столбцах показаны состояния шести счетчиков после следующих четырех прерываний от таймера.

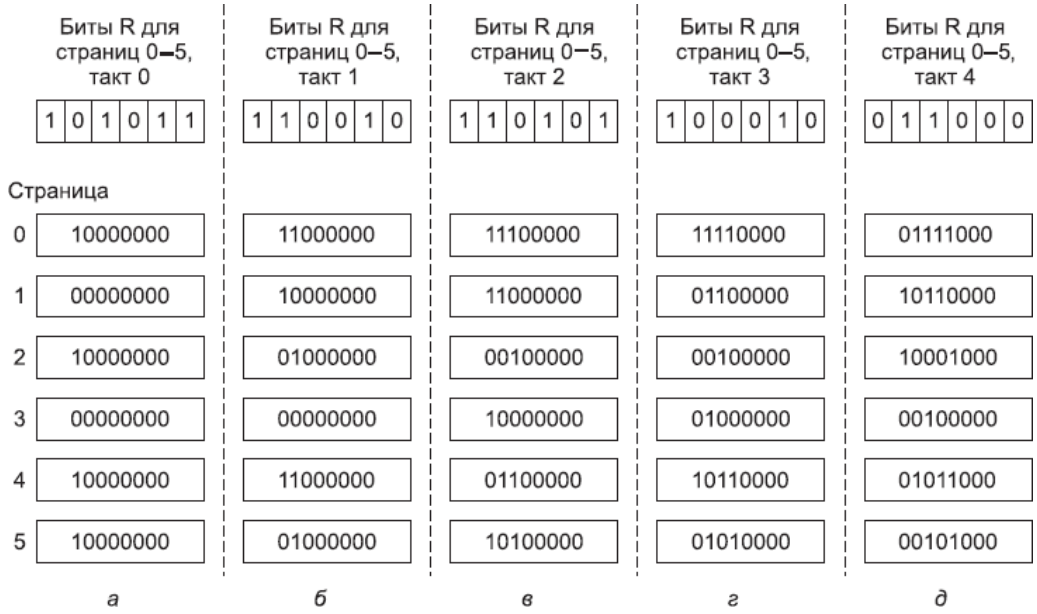

Рис. 3.2.19. Алгоритм старения является программной моделью алгоритма LRU. Здесь показаны шесть страниц в моменты пяти таймерных прерываний, обозначенных буквами от *а* до *д*

При возникновении ошибки отсутствия страницы удаляется та страница, чей счетчик имеет наименьшее значение. Очевидно, что в счетчике страницы, к которой не было обращений за, скажем, четыре прерывания от таймера, будет четыре ведущих нуля, и поэтому значение ее счетчика будет меньшим, чем счетчика страницы, к которой не было обращений в течение трех прерываний от таймера.

Отличия алгоритма старения от полноценного LRU: неразличимость во времени обращений к страницам внутри интервала между двумя последовательными прерывания таймера и ограниченный горизонт (учитываются только несколько последних прерываний таймера)

Алгоритм «рабочий набор»

При использовании замещения страниц в простейшей форме процессы начинают свою работу, не имея в памяти вообще никаких страниц. Как только центральный процессор попытается извлечь первую команду, он получает ошибку отсутствия страницы, заставляющую операционную систему ввести в память страницу, содержащую первую команду. Обычно вскоре за этим следуют ошибки отсутствия страниц с глобальными переменными и стеком. Через некоторое время процесс располагает большинством необходимых ему страниц и приступает к работе, сталкиваясь с ошибками отсутствия страниц относительно редко. Эта стратегия называется замещением страниц по требованию (demand paging), поскольку страницы загружаются только по мере надобности, а не заранее.

Но, как правило, процессы в конкретный момент работают только с небольшой частью памяти. Если обеспечить ее наличие в памяти, количество страничных ошибок резко уменьшится.

Набор страниц, который процесс использует в данный момент, известен как рабочий набор. Если в памяти находится весь рабочий набор, процесс будет работать, не вызывая многочисленных ошибок отсутствия страниц, пока не перейдет к другой фазе выполнения. Если объем доступной памяти слишком мал для размещения всего рабочего набора, процесс вызовет множество ошибок отсутствия страниц и будет работать медленно, поскольку выполнение команды занимает всего несколько наносекунд, а считывание страницы с диска — обычно 10 мс. Если он будет выполнять одну или две команды за 10 мс, то завершение его работы займет целую вечность. О программе,

вызывающей ошибку отсутствия страницы через каждые несколько команд, говорят, что она пробуксовывает.

В многозадачных системах процессы довольно часто сбрасываются на диск (то есть все их страницы удаляются из памяти), чтобы дать возможность другим процессам воспользоваться своей очередью доступа к центральному процессору. Многие системы замещения страниц пытаются отслеживать рабочий набор каждого процесса и обеспечивать его присутствие в памяти, перед тем как позволить процессу возобновить работу. Такой подход называется моделью рабочего набора. Он был разработан для существенного сокращения количества ошибок отсутствия страниц. Загрузка страниц до того, как процессу будет позволено возобновить работу, называется также опережающей подкачкой страниц (prepaging).

Следует заметить, что со временем рабочий набор изменяется. Но обычно он изменяется медленно, поэтому появляется возможность выстроить разумные предположения о том, какие страницы понадобятся при возобновлении работы программы, основываясь на том, каков был ее рабочий набор при последней приостановке ее работы. Опережающая подкачка страниц как раз и заключается в загрузке этих страниц перед возобновлением процесса.

Для реализации модели рабочего набора необходимо, чтобы операционная система отслеживала, какие именно страницы входят в рабочий набор. При наличии такой информации тут же напрашивается возможный алгоритм замещения страниц: при возникновении ошибки отсутствия страницы нужно выселить ту страницу, которая не относится к рабочему набору. Для реализации подобного алгоритма нам необходим четкий способ определения того, какие именно страницы относятся к рабочему набору. По определению рабочий набор — это набор страниц, используемых в k самых последних обращений (некоторые авторы используют термин «k самых последних страничных обращений», но это дело вкуса). Для реализации любого алгоритма рабочего набора некоторые значения k должны быть выбраны заранее. Затем после каждого обращения к памяти однозначно определяется и набор страниц, используемый при самых последних k обращениях к памяти.

На практике вместо вычисления k обращений к памяти используют время выполнения. Например, вместо определения рабочего набора в качестве страниц, использовавшихся в течение предыдущих 10 млн обращений к памяти, мы можем определить его как набор страниц, используемых в течение последних 100 мс времени выполнения. На практике с таким определением работать гораздо лучше и проще. Следует заметить, что для каждого процесса вычисляется только его собственное время выполнения. Таким образом, если процесс запускается во время T и получает 40 мс времени центрального процессора за время T + 100 мс, то для определения рабочего набора берется время 40 мс. Интервал времени центрального процессора, реально занимаемый процессом с момента его запуска, часто называют текущим виртуальным временем. При этом приближении рабочим набором процесса является набор страниц, к которым он обращался в течение последних t секунд виртуального времени.

Основной замысел алгоритма замещения страниц, основанного на рабочем наборе, состоит в том, чтобы найти страницу, не принадлежащую рабочему набору, и удалить ее из памяти. На рис. 3.2.20 показана часть таблицы страниц, используемой в некой машине. Поскольку в качестве кандидатов на выселение рассматриваются только страницы, находящиеся в памяти, страницы, в ней отсутствующие, этим алгоритмом игнорируются. Каждая запись состоит (как минимум) из двух ключевых элементов информации: времени (приблизительного) последнего использования страницы и бита **R** (Referenced — бита обращения).

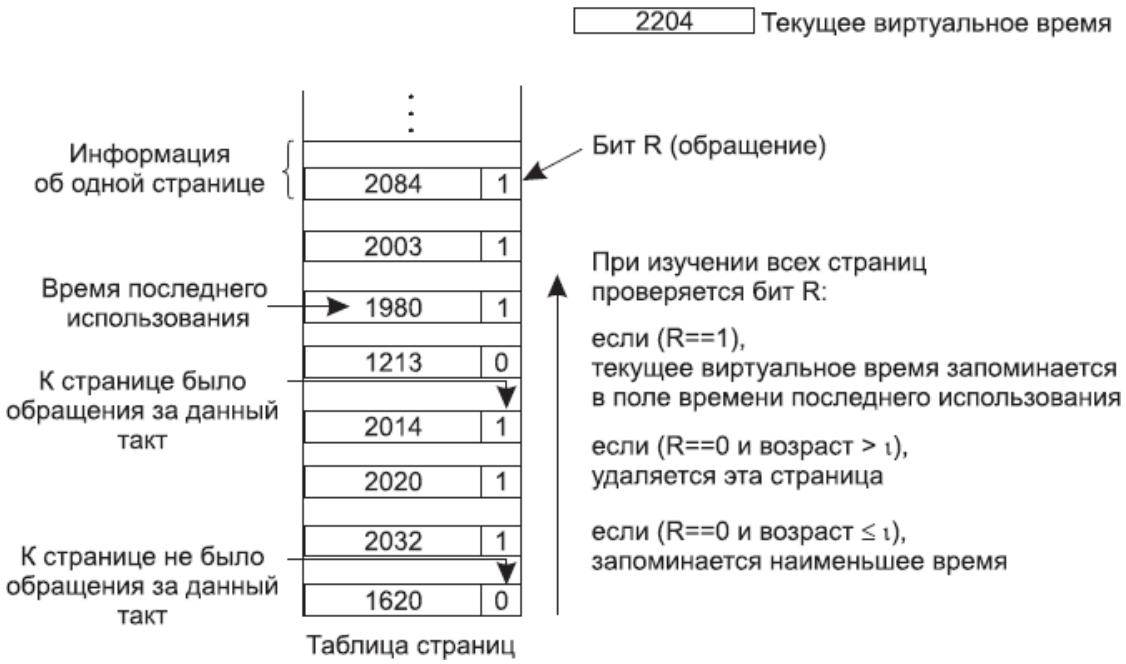

Рис. 3.2.20. Алгоритм рабочего набора

Рассмотрим работу алгоритма. Предполагается, что аппаратура, как было рассмотрено ранее, устанавливает биты **R** и **M**. Также предполагается, что периодические прерывания от таймера запускают программу, очищающую бит обращения **R**. При каждой ошибке отсутствия страницы происходит сканирование таблицы страниц с целью найти страницу, пригодную для удаления.

При каждой обработке записи проверяется состояние бита **R**. Если его значение равно 1, текущее виртуальное время записывается в поле времени последнего использования таблицы страниц, показывая, что страница была использована при возникновении ошибки отсутствия страницы. Если обращение к странице происходит в течение текущего такта времени, становится понятно, что она принадлежит рабочему набору и не является кандидатом на удаление (предполагается, что t охватывает несколько системных тактов).

Если значение **R** равно 0, значит, за текущий такт времени обращений к странице не было и она может быть кандидатом на удаление. Чтобы понять, должна ли она быть удалена или нет, вычисляется ее возраст (текущее виртуальное время за вычетом времени последнего использования), который сравнивается со значением t. Если возраст превышает значение t, то страница уже не относится к рабочему набору и заменяется новой страницей. Сканирование продолжается, и происходит обновление всех остальных записей.

Но если значение **R** равно 0, но возраст меньше или равен t, то страница все еще относится к рабочему набору. Страница временно избегает удаления, но страница с наибольшим возрастом (наименьшим значением времени последнего использования) берется на заметку. Если будет просканирована вся таблица страниц и не будет найдена страница — кандидат на удаление, значит, к рабочему набору относятся все страницы.

В таком случае, если найдена одна и более страниц с **R** = 0, удаляется одна из них, имеющая наибольший возраст. В худшем случае в течение текущего такта было обращение ко всем страницам (и поэтому у всех страниц **R** = 1), поэтому для удаления одна из них выбирается случайным образом, при этом предпочтение отдается неизмененной странице, если таковая имеется.

Алгоритм WSClock

Базовый алгоритм рабочего набора слишком трудоемок, поскольку при возникновении ошибки отсутствия страницы для определения местонахождения подходящего кандидата на удаление необходимо просканировать всю таблицу страниц. Усовершенствованный алгоритм, основанный на алгоритме «часы», но также использующий информацию о рабочем наборе, называется WSClock. Благодаря простоте реализации и хорошей производительности он довольно широко используется на практике.

Необходимая структура данных сводится к циклическому списку страничных блоков, как в алгоритме «часы» и как показано на рис. 3.2.21, ***а***. Изначально этот список пуст. При загрузке первой страницы она добавляется к списку. По мере загрузки следующих страниц они попадают в список, формируя замкнутое кольцо. В каждой записи содержится поле времени последнего использования из базового алгоритма рабочего набора, а также бит **R** (показанный на рисунке) и бит **M** (не показанный на рисунке).

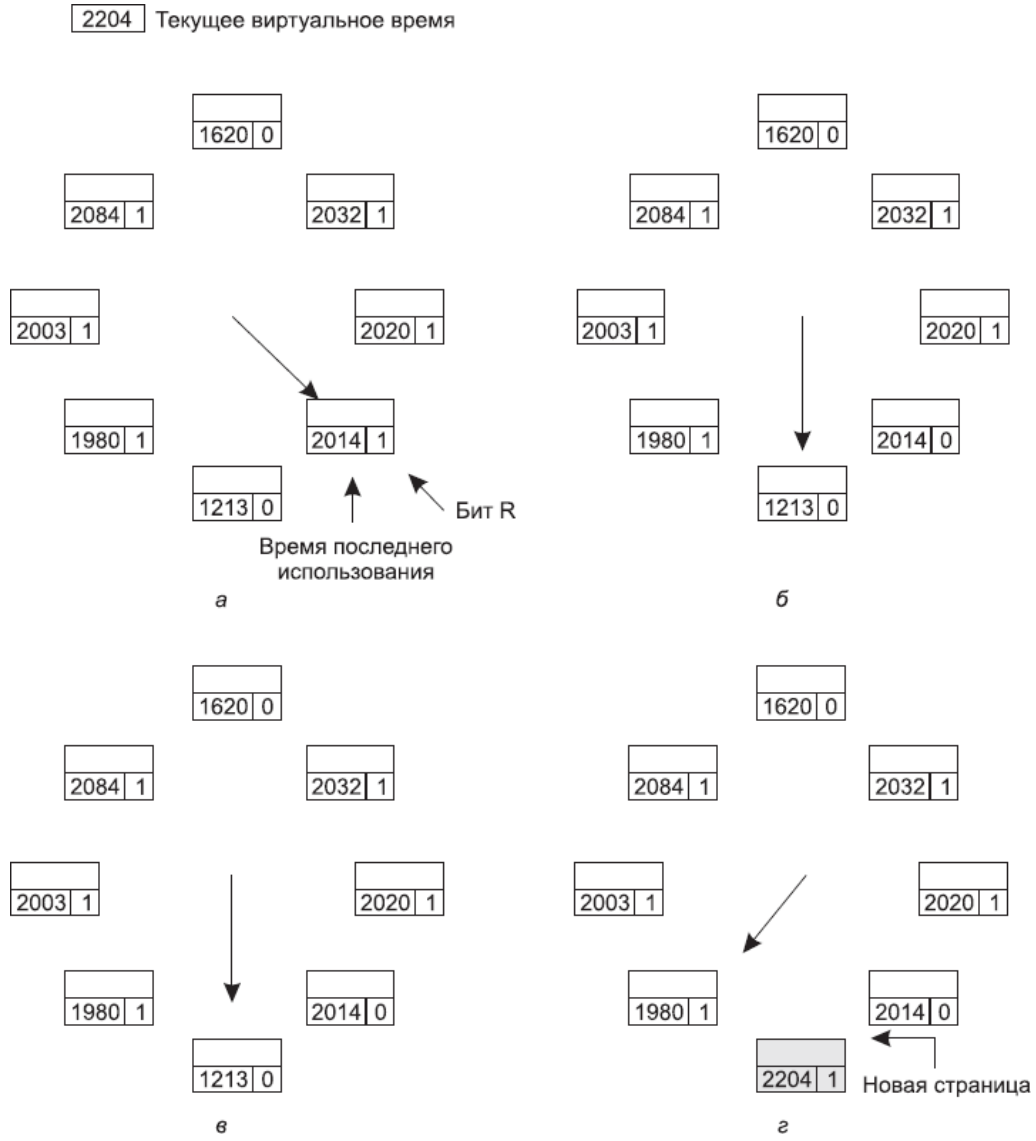

Рис. 3.2.21. Работа алгоритма WSClock: а и б — пример того, что происходит, когда R = 1; в и г — пример того, что происходит, когда R = 0

Как и в алгоритме «часы», при каждой ошибке отсутствия страницы сначала проверяется страница, на которую указывает стрелка. Если бит **R** установлен в 1, значит, страница была использована в течение текущего такта, поэтому она не является идеальным кандидатом на удаление. Затем бит **R** устанавливается в 0, стрелка перемещается на следующую страницу, и алгоритм повторяется уже для нее. Состояние, получившееся после этой последовательности событий, показано на рис. 3.2.21, *б*.

Теперь посмотрим, что получится, если у страницы, на которую указывает стрелка, бит **R** = 0 (рис. 3.2.21, *в*). Если ее возраст превышает значение t и страница не изменена, она не относится к рабочему набору и ее точная копия присутствует на диске. Тогда страничный блок просто истребуется и в него помещается новая страница (рис. 3.2.21, *г*). Но если страница изменена, ее блок не может быть тотчас же истребован, поскольку на диске нет ее точной копии. Чтобы избежать переключения процесса, запись на диск планируется, а стрелка перемещается дальше и алгоритм продолжает свою работу на следующей странице. В конце концов должна попасться старая, неизмененная страница, которой можно будет тут же и воспользоваться.

В принципе, за один оборот часовой стрелки может быть запланирована операция дискового ввода-вывода для всех страниц. Для уменьшения потока обмена данными с диском может быть установлен лимит, позволяющий сбрасывать на диск максимум n страниц. По достижении этого лимита новые записи на диск планироваться уже не должны.

А что делать, если стрелка пройдет полный круг и вернется в начальную позицию?

Тогда следует рассмотреть два варианта.

1. Была запланирована хотя бы одна запись на диск.

2. Не было запланировано ни одной записи на диск.

В первом случае стрелка просто продолжит движение, выискивая неизмененную страницу. Поскольку была запланирована одна или более записей на диск, со временем одна из записей завершится, и задействованная в ней страница будет помечена неизмененной. Первая же неизмененная страница и будет удалена. Эта страница не обязательно должна быть первой запланированной, поскольку драйвер диска может изменить порядок записи, чтобы оптимизировать производительность его работы.

Во втором случае все страницы относятся к рабочему набору, иначе должна была быть запланирована хотя бы одна запись. При недостатке дополнительной информации простейшее, что можно сделать, — истребовать любую неизмененную страницу и воспользоваться ею. Расположение неизмененной страницы может быть отслежено в процессе оборота стрелки. Если неизмененных страниц не имеется, то в качестве жертвы выбирается текущая страница, которая и сбрасывается на диск.

Краткая сравнительная характеристика алгоритмов замещения страниц

Алгоритм	Особенности
Оптимальный	Не может быть реализован, но полезен в качестве оценочного критерия
NRU (Not Recently Used) — алгоритм исключения давно использовавшейся страницы	Является довольно грубым приближением к алгоритму LRU
FIFO (First In, First Out) — алгоритм «первой пришла, первой и ушла»	Может выгрузить важные страницы
Алгоритм «второй шанс»	Является существенным усовершенствованием алгоритма FIFO
Алгоритм «часы»	Вполне реализуемый алгоритм
LRU (Least Recently Used) — алгоритм замещения наименее востребованной страницы	Очень хороший, но труднореализуемый во всех тонкостях алгоритм
NFU (Not Frequently Used) — алгоритм нечастого востребования	Является довольно грубым приближением к алгоритму LRU

Алгоритм старения	Вполне эффективный алгоритм, являющийся неплохим приближением к алгоритму LRU
Алгоритм рабочего набора	Весьма затратный для реализации алгоритм
WSClock	Вполне эффективный алгоритм

Оптимальный алгоритм удаляет страницу с самым отдаленным предстоящим обращением. К сожалению, у нас нет способа определения, какая это будет страница, поэтому на практике этот алгоритм использоваться не может. Но он полезен в качестве оценочного критерия при рассмотрении других алгоритмов.

Алгоритм исключения недавно использованной страницы (NRU) делит страницы на четыре класса в зависимости от состояния битов **R** и **M**. Затем выбирает произвольную страницу из класса с самым низким номером. Этот алгоритм нетрудно реализовать, но он слишком примитивен. Есть более подходящие алгоритмы.

Алгоритм FIFO предполагает отслеживание порядка, в котором страницы были загружены в память, путем сохранения сведений об этих страницах в связанном списке. Это упрощает удаление самой старой страницы, но она-то как раз и может все еще использоваться, поэтому FIFO — неподходящий выбор.

Алгоритм «второй шанс» является модификацией алгоритма FIFO и перед удалением страницы проверяет, не используется ли она в данный момент. Если страница все еще используется, она остается в памяти. Эта модификация существенно повышает производительность. Алгоритм «часы» является простой разновидностью алгоритма «второй шанс». Он имеет такой же показатель производительности, но требует несколько меньшего времени на свое выполнение.

Алгоритм LRU превосходен во всех отношениях, но не может быть реализован без специального оборудования. Если такое оборудование недоступно, то он не может быть использован. Алгоритм NFU является грубой попыткой приблизиться к алгоритму LRU. Его нельзя признать удачным. А вот алгоритм старения — куда более удачное приближение к алгоритму LRU, которое к тому же может быть эффективно реализовано и считается хорошим выбором.

В двух последних алгоритмах используется рабочий набор. Алгоритм рабочего набора обеспечивает приемлемую производительность, но его реализация обходится слишком дорого. Алгоритм WSClock является вариантом, который не только обеспечивает неплохую производительность, но и может быть эффективно реализован.

В итоге наиболее приемлемыми алгоритмами являются алгоритм старения и алгоритм WSClock. Они основаны на LRU и рабочем наборе соответственно. Оба обеспечивают неплохую производительность страничной организации памяти и могут быть эффективно реализованы. Существует также ряд других хороших алгоритмов, но эти два, наверное, имеют наибольшее практическое значение.

3.2.6. Куча (heap)

Куча — это сегмент памяти, для которого не устанавливается постоянный размер перед компиляцией и который может динамически управляться программистом. Думайте о куче как о «свободном пуле» памяти, который вы можете использовать при запуске приложения. Размер кучи приложения определяется физическими ограничениями вашей оперативной памяти (оперативной памяти) и обычно намного больше размера стека [7].

Мы используем память из кучи, когда не знаем, сколько места займет структура данных в нашей программе, когда нам нужно выделить больше памяти, чем доступно в стеке, или когда нам нужно создать переменные, сохраняющиеся в течение всего времени. нашего приложения. Мы можем сделать это на языке программирования C, используя **`malloc`**, **`realloc`**, **`calloc`** и/или **`free`**. Посмотрите пример ниже.

```c
#include <stdlib.h>
int main(void)
{
	int *ptr;
	ptr = (int*) malloc(sizeof(int)* 100);
	// использование выделенной памяти
	free(ptr);
	return(0);
}
```

Выделяем нашей программе 4000 байт памяти, а затем освобождаем ее.

Мы выделяем память из кучи с помощью функции **malloc()**. Аргумент, который мы хотим включить в **malloc**, — это объем памяти, которую мы хотим выделить нашему приложению, в байтах. **malloc** возвращает указатель `void`, тип которого приведен к целочисленному указателю, который теперь указывает на первый адрес в памяти для нашей памяти длиной 4000 байт. Теперь мы можем хранить информацию в этих адресах памяти и делать с этой информацией все, что захотим, на протяжении всей нашей программы или на протяжении всей нашей функции, потому что у нас есть указатель, который ссылается на первый адрес памяти из недавно выделенной кучи.

Если вы намеренно не создаете из кучи переменные, сохраняющиеся на протяжении всего времени работы вашего приложения, вы всегда захотите освободить память обратно машине с помощью функции **free()**. Если вы не освободите память с помощью функции **free()**, у вас останется память, которая сохранится на протяжении всей вашей программы. Если мы не освободим память из нашей программы перед завершением работы приложения, в нашем приложении возникнут утечки памяти. Если в вашем приложении имеется достаточно утечек памяти, оно может потреблять больше памяти, чем физически доступно, и может привести к сбою программ.

Управление кучей в Windows

Каждый процесс имеет **кучу по умолчанию**, предоставляемую системой. Приложения, которые часто выделяют ресурсы из кучи, могут повысить производительность с помощью частных кучи [8].

Частная куча — это блок одной или нескольких страниц в адресном пространстве вызывающего процесса. После создания частной кучи процесс использует такие функции, как **HeapAlloc** и **HeapFree**, для управления памятью в этой куче.

Функции кучи также можно использовать для управления памятью в куче процесса по умолчанию с помощью дескриптора, возвращаемого функцией **GetProcessHeap**. Для этой цели новые приложения должны использовать функции кучи вместо глобальных и локальных функций.

Нет различий между памятью, выделенной из частной кучи, и памятью, выделенной с помощью других функций выделения памяти.

Функция **HeapCreate** создает частный объект кучи, из которого вызывающий процесс может выделять блоки памяти с помощью функции **HeapAlloc**. **HeapCreate** указывает как начальный, так и максимальный размер кучи. Начальный размер определяет количество зафиксированных (commited) страниц для чтения и записи, изначально выделенных для кучи. Максимальный размер определяет общее количество зарезервированных страниц. Эти страницы создают непрерывный блок в виртуальном адресном пространстве процесса, в который может расти куча. Дополнительные страницы автоматически фиксируются из этого зарезервированного пространства, если запросы **HeapAlloc** превышают текущий размер зафиксированных страниц, при условии, что физическое хранилище для него доступно. После фиксации страницы не удаляются до завершения процесса или до тех пор, пока куча не будет уничтожена путем вызова функции **HeapDestroy**.

Память частного объекта кучи доступна только процессу, создав его. Если библиотека динамической компоновки (DLL) создает частную кучу, она делает это в адресном пространстве процесса, который вызвал библиотеку DLL. Onf доступен только для этого процесса.

Функция **HeapAlloc** выделяет указанное количество байтов из частной кучи и возвращает указатель на выделенный блок. Этот указатель можно использовать в функциях **HeapFree**, **HeapReAlloc**, **HeapSize** и **HeapValidate**.

Память, выделенная **HeapAlloc**, не перемещается. Адрес, возвращаемый **HeapAlloc**, действителен до освобождения или перераспределения блока памяти; Блок памяти не нужно блокировать.

Так как система не может сжать частную кучу, она может стать фрагментированной. Приложения, выделяющие большие объемы памяти с различными размерами выделения, могут использовать кучу с низким уровнем фрагментации для уменьшения фрагментации кучи.

Функции кучи можно использовать для создания частной кучи при запуске процесса, указав начальный размер, достаточный для удовлетворения требований процесса к памяти. Если вызов функции **HeapCreate** завершается сбоем, процесс может завершиться или уведомить пользователя о нехватке памяти; Однако, если он будет успешным, процесс будет гарантированно иметь необходимую память.

Память, запрашиваемая методом **HeapCreate**, может быть непрерывной и может не быть непрерывной. Память, выделенная в куче **HeapAlloc**, является непрерывной. Не следует выполнять запись или чтение из памяти в куче, за пределами диапазона памяти, выделенной **HeapAlloc**, и не следует предполагать связь между двумя областями памяти, выделенными **HeapAlloc**.

Вы не должны каким-либо образом ссылаться на память, которая была освобождена **HeapFree**. После освобождения памяти любая информация, которая могла быть в ней, исчезает навсегда. Если требуется информация, не освобождайте память, содержащую эти сведения. Вызовы функций, возвращающие сведения о памяти (например, **HeapSize**), не могут использоваться с освобожденной памятью, так как они могут возвращать фиктивные данные.

Функция **HeapDestroy** уничтожает частный объект кучи. Он удаляет и освобождает все страницы объекта кучи, а также делает дескриптор кучи недействительным.

Глобальные и локальные функции

Глобальные и локальные функции поддерживаются для переноса из 16-разрядного кода или обеспечения совместимости исходного кода с 16-разрядной версией Windows. Начиная с 32-разрядной версии Windows глобальные и локальные функции реализуются как функции-оболочки, которые вызывают соответствующие функции кучи с помощью дескриптора кучи процесса по умолчанию. Таким образом, глобальные и локальные функции имеют большую нагрузку, чем другие функции управления памятью.

Функции кучи предоставляют больше возможностей и управления, чем глобальные и локальные функции. Новые приложения должны использовать функции кучи, если в документации не указано, что следует использовать глобальную или локальную функцию. Например, некоторые функции Windows выделяют память, которую необходимо освободить с помощью **LocalFree**, а глобальные функции по-прежнему используются с динамическим обменом данными (DDE), функциями буфера обмена и объектами данных OLE.

Управление памятью Windows не предоставляет отдельную локальную кучу и глобальную кучу, как это делает 16-разрядная версия Windows. В результате глобальные и локальные семейства функций эквивалентны, и выбор между ними является вопросом

личного предпочтения. Обратите внимание, что переход с 16-разрядной модели сегментированной памяти на 32-разрядную модель виртуальной памяти сделал некоторые связанные глобальные и локальные функции и их параметры ненужными или бессмысленными. Например, больше нет указателей ближнего и дальнего, так как локальные и глобальные выделения возвращают 32-разрядные виртуальные адреса.

Объекты памяти, выделенные **GlobalAlloc** и **LocalAlloc**, находятся в закрытых зафиксированных страницах с доступом на чтение и запись, к которым другие процессы не могут получить доступ. Память, выделенная с помощью **GlobalAlloc** с `GMEM_DDESHARE`, фактически не используется глобально, как в 16-разрядной версии Windows. Это значение не действует и доступно только для обеспечения совместимости. Приложения, которым требуется общая память для других целей, должны использовать объекты сопоставления файлов. Несколько процессов могут сопоставить представление одного и того же объекта сопоставления файлов для предоставления именованной общей памяти.

Выделение памяти ограничивается только доступной физической памятью, включая хранилище в файле подкачки на диске. При выделении фиксированной памяти **GlobalAlloc** и **LocalAlloc** возвращают указатель, который вызывающий процесс может немедленно использовать для доступа к памяти. При выделении перемещаемой памяти возвращаемое значение является дескриптором. Чтобы получить указатель на перемещаемый объект памяти, используйте функции **GlobalLock** и **LocalLock**.

Фактический размер выделенной памяти может быть больше запрошенного размера. Чтобы определить фактическое количество выделенных байтов, используйте функцию **GlobalSize** или **LocalSize**. Если выделенная сумма больше запрошенной суммы, процесс может использовать всю сумму.

Функции **GlobalReAlloc** и **LocalReAlloc** изменяют размер или атрибуты объекта памяти, выделенного **GlobalAlloc** и **LocalAlloc**. Размер может увеличиваться или уменьшаться.

Функции **GlobalFree** и **LocalFree** освобождают память, выделенную **GlobalAlloc**, **LocalAlloc**, **GlobalReAlloc** или **LocalReAlloc**. Чтобы удалить указанный объект памяти без аннулирования дескриптора, используйте функцию **GlobalDiscard** или **LocalDiscard**. Дескриптор может быть позже использован **GlobalReAlloc** или **LocalReAlloc** для выделения нового блока памяти, связанного с тем же дескриптором.

Чтобы вернуть сведения об указанном объекте памяти, используйте функцию **GlobalFlags** или **LocalFlags**. Эти сведения включают количество блокировок объекта и указывают, является ли объект отменяемым или уже удален. Чтобы вернуть дескриптор объекту памяти, связанному с указанным указателем, используйте функцию **GlobalHandle** или **LocalHandle**.

3.2.7. Стек

Стек — это сегмент памяти, в котором данные, такие как локальные переменные и вызовы функций, добавляются и/или удаляются по принципу «последним пришел — первым вышел» (LIFO).

Вообще говоря, стек — это структура данных, которая хранит значения данных в памяти последовательно. Однако, в отличие от массива, вы получаете доступ (чтение или запись) к данным только на «верхней части» стека. Чтение из стека называется «извлечение» (pop), а запись в стек — «вталкивание» (push). Стек также известен как очередь LIFO (последним пришел — первым ушел), поскольку значения извлекаются из стека в порядке, обратном тому, в котором они помещаются в него. Извлеченные данные исчезают из стека [9].

Все архитектуры x86 используют стек в качестве временной области хранения в оперативной памяти, которая позволяет процессору быстро сохранять и извлекать

данные из памяти. На текущую вершину стека указывает регистр esp. Стек «растет» вниз, от верхних адресов памяти к младшим, поэтому значения, недавно помещенные в стек, располагаются по адресам памяти над указателем esp. Ни один регистр специально не указывает на нижнюю часть стека, хотя большинство операционных систем отслеживают границы стека, чтобы обнаружить как состояния «недополнения» (выталкивание пустого стека), так и состояния «переполнения» (помещение слишком большого количества информации в стек).

Когда значение извлекается из стека, оно остается в памяти до тех пор, пока не будет перезаписано. Однако никогда не следует полагаться на содержимое адресов памяти ниже **esp**, поскольку другие функции могут перезаписать эти значения без вашего ведома.

Пользователи ранних версий Windows могут вспомнить печально известный «синий экран смерти», который иногда вызывался исключением переполнения стека. Это происходит, когда в стек записывается слишком много данных, и стек «вырастает» за свои пределы. Современные операционные системы используют улучшенную проверку границ и восстановление ошибок, чтобы уменьшить вероятность переполнения стека и поддерживать стабильность системы после того, как оно произошло.

Следующие строки кода функционально эквивалентны:

```
push eax                    sub esp, 4
                            mov DWORD PTR SS:[esp], eax

pop eax                     mov eax, DWORD PTR SS:[esp]
                            add esp, 4
```

но команды **push** и **pop** выполняются намного быстрее, чем альтернативный код.

Организация стека в архитектуре x86-64

В x86-64 по сравнению с x86 заметно выросло число регистров общего назначения (вдвое), что не может не радовать. Поэтому больше регистров можно использовать для передачи аргументов в функции, при этом меньше использовать стек в памяти, тем самым вызов функции ускоряется.

Архитектура x86-64 имеет:

- 16 целочисленных 64-битных регистров общего назначения (RAX, RBX, RCX, RDX, RBP, RSI, RDI, **RSP**, R8 — R15);
- 8 80-битных регистров с плавающей точкой (ST0 — ST7);
- 8 64-битных регистров MMX (MM0 — MM7, имеют общее пространство с регистрами ST0 — ST7);
- 16 128-битных регистров SSE (XMM0 — XMM15);
- 64-битный указатель RIP и 64-битный регистр флагов RFLAGS.

Понятие регистров процессора виртуализируется. Эта технология называется переименованием регистров (register renaming). Декодер команд выделяет регистр из большого банка регистров. Когда инструкция завершается, значение этого динамически распределенного регистра записывается обратно в любой регистр, в котором в настоящее время хранится значение, скажем, **rax**.

Системные вызовы Linux

В процессорах архитектуры x86 для явного вызова синхронного прерывания имеется инструкция int, аргументом которой является номер прерывания (от 0 до 255). В защищённом и длинном режиме обычные программы не могут обслуживать прерывания, эта функция доступна только системному коду (операционной системе).

В ОС Linux номер прерывания 0x80 используется для выполнения системных вызовов. Обработчиком прерывания 0x80 является ядро Linux. Программа перед выполнением

прерывания помещает в регистр **eax** номер системного вызова, который нужно выполнить. Когда управление переходит в ring 0, то ядро считывает этот номер и вызывает нужную функцию.

Метод этот широко применялся на 32-битных системах, на 64-битных он считается устаревшим и не применяется, но тоже работает, хотя с целым рядом ограничений (например, нельзя в качестве параметра передать 64-битный указатель).

- Поместить номер системного вызова в **eax**.
- Поместить аргументы в регистры **ebx**, **ecx**, **edx**, **esi**, **edi**, **ebp**.
- Вызвать инструкцию int 0x80.
- Получить результат из **eax**.

Пример реализации вызова getpid() на ассемблере

```
.intel_syntax noprefix
 .globl mygetpid
 .text
mygetpid:
    mov eax, 20
    int 0x80
    ret
```

Инструкция syscall

Это более современный метод, используется в 64-битном Linux, работает быстрее.

- Номер системного вызова помещается в **rax**.
- Аргументы записываются в **rdi**, **rsi**, **rdx**, **r10**, **r8** и **r9**.
- Затем вызывается **syscall**.
- Результат в **rax**.

Пример (для системного вызова getpid по таблице используется номер 39):

```
.intel_syntax noprefix
 .globl mygetpid
 .text
mygetpid:
    mov rax, 39
    syscall
    ret
```

Список использованных источников

1. Unix2019b/Организация памяти на x86-64
https://acm.bsu.by/wiki/Unix2019b/Организация памяти на x86-64

2. What are Ring 0 and Ring 3 in the context of operating systems?
https://stackoverflow.com/questions/18717016/what-are-ring-0-and-ring-3-in-the-context-of-operating-systems

3. What is protection ring -1?
https://security.stackexchange.com/questions/129098/what-is-protection-ring-1

4. Может ли Windows 11 работать на 32-битном процессоре?
https://novinkiit.com/mozhet-li-windows-11-rabotat-na-32-bitnom-protsessore/

5. Таненбаум, Э. Современные операционные системы. / Э. Таненбаум, Х. Бос. — 4-е изд. — СПб.: Питер, 2015. — 1120 с.

6. Функции ОС по управлению памятью
https://karpov-k.me/computernaya-nauka/os/340-func-os-po-upravleniu-pamyatiu

7. Stack vs Heap. What's the Difference and Why Should I Care?
https://www.linux.com/training-tutorials/stack-vs-heap-whats-difference-and-why-should-i-care/

8. Функции кучи
https://learn.microsoft.com/ru-ru/windows/win32/memory/heap-functions

9. x86 Disassembly/The Stack
https://en.wikibooks.org/wiki/X86_Disassembly/The_Stack

10. Архитектура x86-64
https://acm.bsu.by/wiki/C2017/Архитектура_x86-64

Windows Programming/Memory Subsystem
https://en.wikibooks.org/wiki/Windows_Programming/Memory_Subsystem