1. Модель программного интерфейса операционной системы Windows. Нотация программного интерфейса. Понятие объекта ядра и описателя объекта ядра операционной системы Windows. Модель архитектуры ОС Windows.

2. Понятие пользовательского режима и режима ядра операционной системы Windows. Модель виртуальной памяти процесса в пользовательском режиме и в режиме ядра операционной системы Windows. Архитектура приложения в пользовательском режиме работы и в режиме ядра ОС Windows. Основные модули ОС Windows.

3. Системный реестр операционной системы Windows. Структура и главные разделы. Точки автозапуска программ. Средства редактирования реестра Windows. Функции работы с реестром из приложения.

4. Понятие окна в ОС Windows. Основные элементы окна. Понятие родительского и дочернего окна. Структура программы с событийным управлением. Минимальная программа для ОС Windows с окном на экране. Создание и отображение окна.

5. Структура программы с событийным управлением. Структура события – оконного сообщения Windows. Очередь сообщений. Цикл приема и обработки сообщений. Процедура обработки сообщений. Процедуры посылки сообщений. Синхронные и асинхронные сообщения.(Сытов П)

6. Ввод данных с манипулятора «мышь». Обработка сообщений мыши. Ввод данных с клавиатуры. Понятие фокуса ввода. Обработка сообщений от клавиатуры.

7. Вывод информации в окно. Механизм перерисовки окна. Понятие области обновления окна. Операции с областью обновления окна.

8. Принципы построения графической подсистемы ОС Windows. Понятие контекста устройства. Вывод графической информации на физическое устройство. Управление цветом. Палитры цветов. Графические инструменты. Рисование геометрических фигур.

9. Растровые изображения. Виды растровых изображений. Значки и курсоры. Способ вывода растровых изображений с эффектом прозрачного фона. Аппаратно-зависимые и аппаратно-независимые растровые изображения. Операции с растровыми изображениями. Вывод растровых изображений

10. Библиотека работы с двумерной графикой Direct2D. Инициализация библиотеки. Фабрика графических объектов библиотеки Direct2D. Вывод графики средствами библиотеки Direct2D.

11. Вывод текста в ОС Windows. Понятие шрифта. Характеристики шрифта. Понятия физического и логического шрифта. Операции с физическими шрифтами. Операции с логическими шрифтами. Параметры ширины и высоты логического шрифта

12. Системы координат. Трансформации. Матрица трансформаций. Виды трансформаций и их представление в матрице трансформаций. Преобразования в страничной системе координат. Режимы масштабирования

13. Понятие ресурсов программ Windows. Виды ресурсов. Операции с ресурсами.

14. Понятие динамически-загружаемой библиотеки. Создание DLL-библиотеки. Использование DLL-библиотеки в программе методом статического импорта процедур. Соглашения о вызовах процедур DLL-библиотеки. Точка входа-выхода DLL-библиотеки.

15. Понятие динамически-загружаемой библиотеки. Создание DLL-библиотеки. Использование DLL-библиотеки в программе методом динамический импорта процедур.

16. Понятие динамически-загружаемой библиотеки. Создание в DLL-библиотеке разделяемых между приложениями глобальных данных. Разделы импорта и экспорта DLL-библиотеки. Переадресация вызовов процедур DLL-библиотек к другим DLL-библиотекам. Исключение конфликта версий DLL

17. Понятие объекта ядра ОС Windows. Виды объектов ядра. Атрибуты защиты объекта ядра. Дескриптор защиты объекта ядра. Создание и удаление объектов ядра.

18. Проецирование файлов в память. Отличие в механизме проецирования файлов в память в ОС Windows и UNIX/Linux. Действия по проецированию файла в память.

19. Современные многопроцессорные архитектуры SMP и NUMA. Многоуровневое кэширование памяти в современных процессорах. Проблема перестановки операций чтения и записи в архитектурах с ослабленной моделью памяти. Способы решения проблемы перестановки операций чтения и записи.

20. Средства распараллеливания вычислений в ОС Windows. Понятия процесса и потока. Достоинства и недостатки процессов и потоков. Создание и завершение процесса. Запуск процессов по цепочке.

21. Средства распараллеливания вычислений в ОС Windows. Понятия процесса и потока. Создание и завершение потока. Приостановка и возобновление потока. Контекст потока.

22. Понятие пула потоков. Архитектура пула потоков. Операции с потоками при работе с пулом потоков.

23. Распределение процессорного времени между потоками ОС Windows. Механизм приоритетов. Класс приоритета процесса. Относительный уровень приоритета потока. Базовый и динамический приоритеты потока. Операции с приоритетами.

24. Механизмы синхронизации потоков одного и разных процессов в ОС Windows. Обзор и сравнительная характеристика механизмов синхронизации.

25. Синхронизация потоков в пределах одного процесса ОС Windows. Критическая секция. Операции с критической секцией. Атомарные операции.

26. Синхронизация потоков в пределах одного процесса ОС Windows. Ожидаемое условие (монитор Хора). Операции с ожидаемым условием. Пример использования ожидаемого условия для синхронизации потоков.

27. Синхронизация потоков разных процессов с помощью объектов ядра. Понятие свободного и занятого состояния объекта ядра. Процедуры ожидания освобождения объекта ядра. Перевод объекта ядра в свободное состояние. Объекты синхронизации: блокировки, семафоры, события.

28. Синхронизация потоков разных процессов с помощью объектов ядра. Понятие свободного и занятого состояния объекта ядра. Процедуры ожидания освобождения объекта ядра. Ожидаемые таймеры. Оконные таймеры.

29. Структура системного программного интерфейса ОС Windows (Native API). Nt-функции и Zw-функции в пользовательском режиме и режиме ядра ОС Windows.

30. Системный вызов ОС Windows. Алгоритм системного вызова. Особенность системного вызова из режима ядра.

31. Отладка драйверов ОС Windows. Средства отладки драйверов. Посмертный анализ. Живая отладка.

32. Структуры данных общего назначения в режиме ядра ОС Windows. Представление строк стандарта Unicode. Представление двусвязных списков.

33. Механизм прерываний ОС Windows. Аппаратные и программные прерывания. Понятие прерывания, исключения и системного вызова. Таблица векторов прерываний (IDT).

34. Аппаратные прерывания. Программируемый контроллер прерываний. Механизм вызова прерываний. Обработка аппаратных прерываний. Понятие приоритета прерываний (IRQL). Понятие процедуры обработки прерываний (ISR).

35. Понятие приоритета прерываний (IRQL). Приоритеты прерываний для процессора x86 или x64. Процедура обработки прерываний (ISR). Схема обработки аппаратных прерываний.

36. Программные прерывания. Понятие отложенной процедуры (DPC). Назначение отложенных процедур. Механизм обслуживания отложенных процедур. Операции с отложенными процедурами.

37. Понятие асинхронной процедуры (APC). Назначение асинхронных процедур. Типы асинхронных процедур. Операции с асинхронными процедурами.

38. Понятие асинхронной процедуры (APC). Асинхронные процедуры режима ядра: специальная и нормальная APC-процедуры. Асинхронные процедуры пользовательского режима.

39. Понятие элемента работы (Work Item). Назначение элементов работы. Операции с элементами работы. Очереди элементов работы. Обслуживание элементов работы.

40. Управление памятью в ОС Windows. Менеджер памяти. Виртуальная память процесса. Управление памятью в пользовательском режиме. Страничная виртуальная память. Куча (свалка, heap). Проецирование файлов в память.

41. Управление памятью в пользовательском режиме ОС Windows. Оптимизация работы кучи с помощью списков предыстории (Look-aside Lists) и низко-фрагментированной кучи (Low Fragmentation Heap).

42. Структура виртуальной памяти в ОС Windows. Виды страниц. Состояния страниц. Структура виртуального адреса. Трансляция виртуального адреса в физический. Кэширование виртуальных адресов

43. Управление памятью в режиме ядра ОС Windows. Пулы памяти. Выделение и освобождение памяти в пулах памяти. Структура описателя пула памяти. Доступ к описателям пулов памяти на однопроцессорной и многопроцессорной системах.

44. Пулы памяти ОС Windows. Пул подкачиваемой памяти, пул неподкачиваемой памяти, пул сессии, особый пул. Тегирование пулов. Структура данных пула. Выделение и освобождение памяти в пулах памяти. Организация списков свободных блоков в пуле памяти. Заголовок блока пула памяти.

45. Управление памятью в режиме ядра ОС Windows. Оптимизация использования оперативной памяти с помощью списков предыстории – Look-aside Lists.

46. Представление объекта ядра в памяти. Менеджер объектов.

47. Фиксация данных в физической памяти ОС Windows. Таблица описания памяти (MDL) и ее использование.

48. Понятие драйвера ОС Windows. Виды драйверов. Типы драйверов в режиме ядра. Точки входа в драйвер.

49. Объект, описывающий драйвер. Объект, описывающий устройство. Объект, описывающий файл. Структура и взаимосвязь объектов.

50. Понятие пакета ввода-вывода (IRP). Структура пакета ввода-вывода. Схема обработки пакета ввода-вывода при открытии файла.

51. Понятие пакета ввода-вывода (IRP). Структура пакета ввода-вывода. Схема обработки пакета ввода-вывода при выполнении чтения-записи файла.

52. Перехват API-вызовов ОС Windows в пользовательском режиме. Внедрение DLL с помощью реестра. Внедрение DLL с помощью ловушек. Внедрение DLL с помощью дистанционного потока.

53. Перехват API-вызовов ОС Windows в пользовательском режиме. Замена адреса в таблице импорта. Перехват в точке входа в процедуру с помощью подмены начальных инструкций (Microsoft Detours).

54. Перехват API-вызовов ОС Windows в режиме ядра. Таблица системных функций KeServiceDescriptorTable. Таблица системных функций KeServiceDescriptorTableShadow. Понятие UI-потока. Защита от перехвата (Kernel Patch Protection) в 64-разрядной ОС Windows.

55. Перехват API-вызовов менеджера объектов ОС Windows в режиме ядра.

56. Перехват API-вызовов создания и уничтожения процессов и потоков ОС Windows в режиме ядра.

57. Перехват операций с реестром в ОС Windows в режиме ядра.

58. Перехват операций с файлами в ОС Windows в режиме ядра. Мини-фильтры файловой системы.

#	*Фамилия Имя*
1	Бигвава Лариса (DONE)
2	Бигвава Лариса (DONE)
3	Костюкова Анастасия(DONE)
4	Бигвава Лариса (DONE)
5	Сытов Павел(DONE)
6	Сытов Павел(DONE)
7	Видничук Вадим(DONE)
8	Сытов Павел(DONE)
9	Видничук Вадим(DONE)
10	Кулуев Павел(DONE)
11	Видничук Вадим(DONE)
12	Кучумова Мария(DONE)
13	Клещиков Алексей (DONE)
14	Кулуев Павел (DONE)
15	Кулуев Павел(DONE)
16	Кулуев Павел (DONE)
17	Клещиков Алексей (DONE)
18	Дударев Максим (DONE)
19	Дударев Максим (DONE)
20	Костюкова Анастасия(DONE)
21	Костюкова Анастасия(DONE)
22	Чигир Вероника(DONE)
23	Видничук Вадим (DONE)
24	Костюкова Анастасия (DONE)
25	Костюкова Анастасия(DONE)
26	Рылеев Евгений (DONE)
27	Рылеев Евгений (DONE)
28	Рылеев Евгений (DONE)
29	Житницкий Александр (DONE)
30	Житницкий Александр (DONE)
31	Козорез Станислав(DONE)

32	Чигир Вероника(DONE)
33	Казак Виктор (DONE)
34	Казак Виктор (DONE)
35	Чигир Вероника(DONE)
36	Гривачевский Андрей(DONE)
37	Гривачевский Андрей(DONE)
38	Гривачевский Андрей(DONE)
39	Чигир Вероника(DONE)
40	Кучумова Мария (DONE)
41	Кучумова Мария (DONE)
42	Кучумова Мария (DONE)
43	Левко Сергей (DONE)
44	Левко Сергей (DONE)
45	Левко Сергей (DONE)
46	Тесейко Мария (DONE)
47	Козорез Станислав (DONE)
48	Карпеш Таня(DONE)
49	Карпеш Таня(DONE)
50	Ильюкевич Андрей (DONE)
51	Ильюкевич Андрей (DONE)
52	Клещиков Алексей (DONE)
53	Клещиков Алексей (DONE)
54	Клещиков Алексей (DONE)
55	Видничук Вадим(DONE)
56	Пушнов Никита ; Поплавскис Паша(PROCESSING)
57	Пушнов Никита ; Поплавскис Паша(DONE)
58	Пушнов Никита ; Поплавскис Паша(PROCESSING)

1. Модель программного интерфейса операционной системы Windows. Нотация программного интерфейса. Понятие объекта ядра и описателя объекта ядра операционной системы Windows. Модель архитектуры ОС Windows.
Модель программного интерфейса операционной системы Windows. Нотация программного интерфейса.

Интерфейс прикладного программирования Windows API (application programming interface) является интерфейсом системного программирования в пользовательском режиме для семейства операционных систем Windows. Windows API состоит из нескольких тысяч вызываемых функций, которые разбиты на следующие основные категории:

1. Базовые службы (Base Services).
2. Службы компонентов (Component Services).
3. Службы пользовательского интерфейса (User Interface Services).
4. Графические и мультимедийные службы (Graphics and Multimedia Services).
5. Обмен сообщениями и совместная работа (Messaging and Collaboration).

Нотация Windows API (Win32, Win64):

1. Имена функций – «глагол–существительное»: CreateWindow, ReadFile, SendMessage.
2. Имена переменных – префикс (венгерская нотация, Charles Simonyi).

API (Application Programming Interface).

API - набор готовых классов, процедур, функций, структур и констант, предоставляемых приложением (библиотекой, сервисом) для использования во внешних программных продуктах.

API определяет функциональность, которую предоставляет программа (модуль, библиотека), при этом API позволяет абстрагироваться от того, как именно эта функциональность реализована.

Программные компоненты взаимодействуют друг с другом посредством API. При этом обычно компоненты образуют иерархию — высокоуровневые компоненты используют API низкоуровневых компонентов, а те, в свою очередь, используют API ещё более низкоуровневых компонентов.

n **Процедурный API.** Единая точка доступа к службе – за вызовом процедуры стоит программное прерывание.
n **Объектный подход**. Отсутствие указателей на внутренние структуры данных ОС. Применение описателей (дескрипторов) вместо указателей.
n **«Венгерская» нотация в идентификаторах.**

Суть венгерской нотации сводится к тому, что имена идентификаторов предваряются заранее оговорёнными префиксами, состоящими из одного или нескольких символов. При этом, как правило, ни само наличие префиксов, ни их написание не являются требованием языков программирования, и у каждого программиста (или коллектива программистов) они могут быть своими.

s	string	строка	sClientName	префикс задает тип
d	delta	Разница между значениями	int a, b; ... dc = b - a;	префикс задает смысл

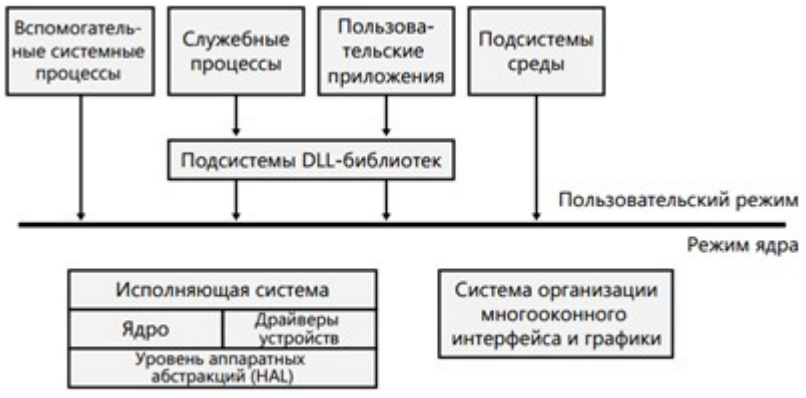
Понятие объекта ядра и описателя объекта ядра операционной системы Windows.
Система позволяет создавать и оперировать с несколькими типами объектов ядра, в том числе: маркерами доступа (access token objects), файлами (file objects), проекциями файлов (file-mapping objects), портами завершения ввода-вывода (I/O completion port objects), заданиями (job objects), почтовыми ящиками (mailslot objects), мьютексами (mutex objects), каналами (pipe objects), процессами (process objects), семафорами (semaphore objects), потоками (thread objects) и ожидаемыми таймерами (waitable timer objects). Эти объекты создаются Windows-функциями. Каждый **объект ядра** — на самом деле просто блок памяти, выделенный ядром и доступный только ему. Этот блок представляет собой структуру данных, в элементах которой содержится информация об объекте. Некоторые элементы (дескриптор защиты, счетчик числа пользователей и др.) присутствуют во всех объектах, но большая их часть специфична для объектов конкретного типа. Например, у объекта «процесс» есть идентификатор, базовый приоритет и

код завершения, а у объекта «файл» — смещение в байтах, режим разделения и режим открытия. Поскольку структуры объектов ядра доступны только ядру, приложение не может самостоятельно найти эти структуры в памяти и напрямую модифицировать их содержимое. Такое ограничение Microsoft ввела намеренно, чтобы ни одна программа не нарушила целостность структур объектов ядра. Это же ограничение позволяет Microsoft вводить, убирать или изменять элементы структур, не нарушая работы каких-либо приложений. Но вот вопрос: если мы не можем напрямую модифицировать эти структуры, то как же наши приложения оперируют с объектами ядра? Ответ в том, что в Windows предусмотрен набор функций, обрабатывающих структуры объектов ядра по строго определенным правилам. Мы получаем доступ к объектам ядра только через эти функции. Когда Вы вызываете функцию, создающую объект ядра, она возвращает описатель, идентифицирующий созданный объект. **Описатель** следует рассматривать как «непрозрачное» значение, которое может быть использовано любым потоком Вашего процесса. Этот описатель Вы передаете Windows-функциям, сообщая системе, какой объект ядра Вас интересует.

Создание объекта ядра:
При инициализации процесса система создает в нем таблицу описателей, используемую только для объектов ядра. Когда процесс инициализируется в первый раз, таблица описателей еще пуста. Но стоит одному из его потоков вызвать функцию, создающую объект ядра (например, CreateFileMapping), как ядро выделяет для этого объекта блок памяти и инициализирует его; далее ядро просматривает таблицу описателей, принадлежащую данному процессу, и отыскивает свободную запись. Поскольку таблица еще пуста, ядро обнаруживает структуру с индексом 1 и инициализирует ее. Указатель устанавливается на внутренний адрес структуры данных объекта, маска доступа — на доступ без ограничений, и, наконец, определяется последний компонент – флаги.
Все функции, создающие объекты ядра, возвращают описатели, которые привязаны к конкретному процессу и могут быть использованы в любом потоке данного процесса. Значение описателя представляет собой индекс в таблице описателей, принадлежащей процессу, и таким образом идентифицирует место, где хранится информация, связанная с объектом ядра.

Закрытие объекта ядра:
Независимо от того, как именно Вы создали объект ядра, по окончании работы с ним его нужно закрыть вызовом *CloseHandle*:
BOOL CloseHandle(HANDLE hobj);
Эта функция сначала проверяет таблицу описателей, принадлежащую вызывающему процессу, чтобы убедиться, идентифицирует ли переданный ей индекс (описатель) объект, к которому этот процесс действительно имеет доступ. Если переданный индекс правилен, система получает адрес структуры данных объекта и уменьшает в этой структуре счетчик числа пользователей; как только счетчик обнулится, ядро удалит объект из памяти.

Модель архитектуры ОС Windows.



Упрощенное представление архитектуры Windows

У вспомогательных системных процессов, у процессов служб, у пользовательских приложений и у подсистем среды окружения, — у всех есть свое собственное закрытое адресное пространство.
Четырем основным процессам пользовательского режима можно дать следующие описания:
1. **Фиксированные** (или реализованные на аппаратном уровне) **вспомогательные системные процессы**, такие как процесс входа в систему и администратор сеансов — Session Manager, которые не входят в службы Windows (они не запускаются диспетчером управления службами).
2. **Служебные процессы**, реализующие такие службы Windows, как Диспетчер задач (Task Scheduler) и спулер печати (Print Spooler). Как правило, от служб требуется, чтобы они работали независимо от входов пользователей в систему.
3. **Пользовательские приложения**, которые могут относиться к одному из следующих типов: для 32- или 64-разрядной версии Windows, для 16-разрядной версии Windows 3.1, для 16-разрядной версии MS-DOS

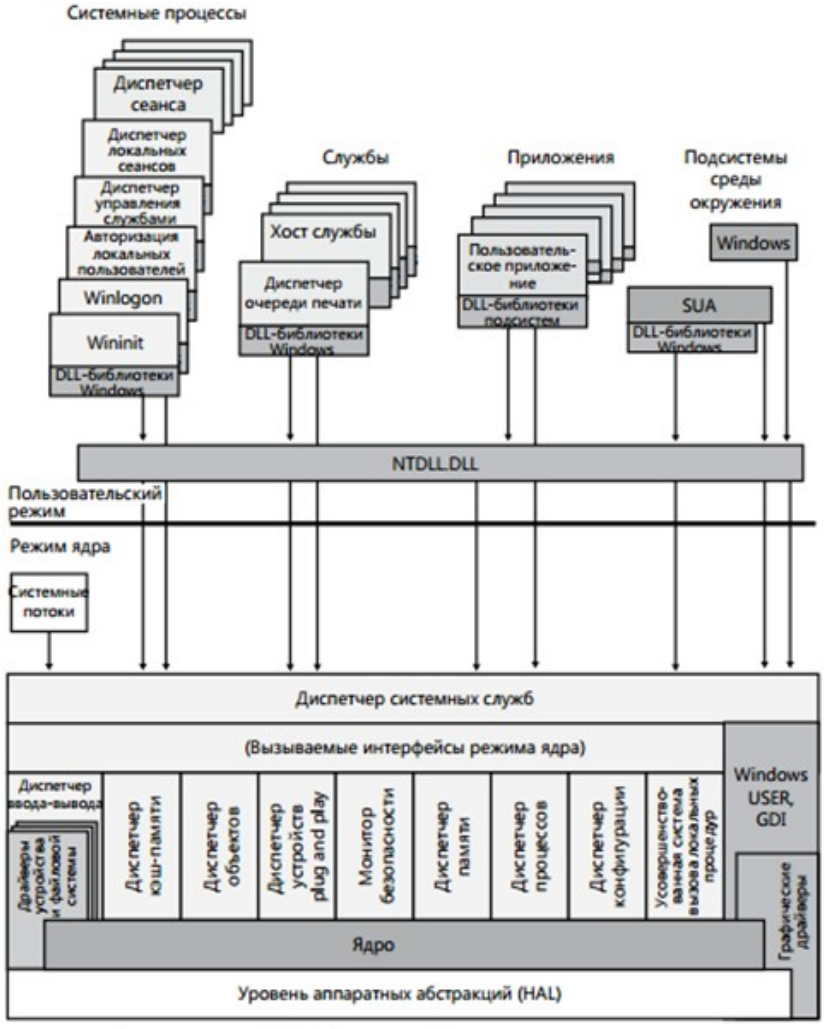
или для 32- или 64-разрядной версии POSIX. Следует учесть, что 16-разрядные приложения могут запускаться только на 32-разрядной версии Windows.

4. **Серверные процессы подсистемы окружения**, которые реализуют часть поддержки среды операционной системы или специализированную часть, представляемую пользователю и программисту. Изначально Windows NT поставляется тремя подсистемами среды: Windows, POSIX и OS/2. Но подсистемы POSIX и OS/2 последний раз поставлялись с Windows 2000. Выпуски клиентской версии Windows Ultimate и Enterprise, а также все серверные версии включают поддержку для усовершенствованной подсистемы POSIX, которая называетсяподсистемой для приложений на основе Unix (Unix-based Applications, SUA).

При выполнении под управлением Windows пользовательские приложения не вызывают имеющиеся в операционной системе Windows службы напрямую, а проходят через одну или несколько подсистем динамически подключаемых библиотек (dynamic-link libraries, DLL).

Подсистемы DLL-библиотек предназначены для перевода документированной функции в соответствующий внутренний (и зачастую недокументированный) вызов системной службы. Этот перевод может включать в себя (или не включать) отправку сообщения процессу подсистемы среды, обслуживающему пользовательское приложение.

В Windows входят следующие компоненты, работающие в режиме ядра:



Аппаратные интерфейсы (шины, устройства ввода-вывода, прерывания, интервальные таймеры, DMA, управление кэш-памятью и т. д.)

Архитектура Windows

1. **Исполняющая система Windows** содержит основные службы операционной системы, такие как управление памятью, управление процессами и потоками, безопасность, ввод-вывод, сеть и связь между процессами.

2. **Ядро Windows** состоит из низкоуровневых функций операционной системы, таких как диспетчеризация потоков, диспетчеризация прерываний и исключений и мультипроцессорная синхронизация. Оно также предоставляет набор подпрограмм и базовых объектов, используемых остальной исполняющей системой для реализации высокоуровневых конструктивных элементов. »

3. К **драйверам устройств** относятся как аппаратные драйверы устройств, которые переводят вызовы функций ввода-вывода в запросы ввода-вывода конкретного аппаратного устройства, так и неаппаратные драйверы устройств, такие как драйверы файловой системы и сети.

4. **Уровень аппаратных абстракций** (hardware abstraction layer, HAL), являющийся уровнем кода, который изолирует ядро, драйверы устройств и остальную исполняющую систему Windows от аппаратных различий конкретных платформ (таких как различия между материнскими платами).

5. **Система организации многооконного интерфейса и графики**, реализующая функции графического пользовательского интерфейса (graphical user interface, GUI), более известные как имеющиеся в Windows USER- и GDI-функции, предназначенные для работы с окнами, элементами управления пользовательского интерфейса и графикой.

2. Понятие пользовательского режима и режима ядра операционной системы Windows. Модель виртуальной памяти процесса в пользовательском режиме и в режиме ядра операционной системы Windows. Архитектура приложения в пользовательском режиме работы и в режиме ядра ОС Windows. Основные модули ОС Windows.

Понятие пользовательского режима и режима ядра операционной системы Windows. Модель виртуальной памяти процесса в пользовательском режиме и в режиме ядра операционной системы Windows.

Чтобы защитить жизненно важные системные данные от доступа и (или) внесения изменений со стороны пользовательских приложений, в Windows используются два процессорных режима доступа (даже если процессор, на котором работает Windows, поддерживает более двух режимов): пользовательский режим и режим ядра. Код пользовательского приложения запускается в **пользовательском режиме**, а код операционной системы (например, системные службы и драйверы устройств) запускается в режиме ядра. **Режим ядра** — такой режим работы процессора, в котором предоставляется доступ ко всей системной памяти и ко всем инструкциям центрального процессора. Предоставляя программному обеспечению операционной системы более высокий уровень привилегий, нежели прикладному программному обеспечению, процессор гарантирует, что приложения с неправильным поведением не смогут в целом нарушить стабильность работы системы. Хотя у каждого Windows-процесса есть свое собственное закрытое адресное пространство, код операционной системы и код драйвера устройства, используют одно и то же общее виртуальное адресное пространство. Каждая страница **в виртуальной памяти** имеет пометку, показывающую, в каком режиме доступа должен быть процессор для чтения и (или) записи страницы. Доступ к страницам в системном пространстве может быть осуществлен только из режима ядра, тогда как доступ ко всем страницам в пользовательском адресном пространстве может быть осуществлен из пользовательского режима. Станицы, предназначенные только для чтения (например, те страницы, которые содержат статические данные), недоступны для записи из любого режима. Кроме того, при работе на процессорах, поддерживающих защиту той памяти, которая не содержит исполняемого кода (no-execute memory protection), Windows помечает страницы, содержащие данные, как неисполняемые, предотвращая тем самым неумышленное или злонамеренное выполнение кода из областей данных.

32-разрядные версии Windows не защищают закрытую системную память чтения-записи, используемую компонентами операционной системы, запущенными в режиме ядра. Иными словами, в режиме ядра код операционной системы и драйвера устройства имеют полный доступ к системному пространству памяти и могут обойти систему защиты Windows, получив доступ к объектам. Поскольку основная часть кода операционной системы Windows работает в режиме ядра, очень важно, чтобы компоненты, работающие в этом режиме, были тщательно проработаны и протестированы, чтобы не нарушать безопасность системы или не становиться причиной нестабильной работы системы. Отсутствие защиты также подчеркивает необходимость проявлять особую осторожность при загрузке драйвера устройства стороннего производителя, потому что программное обеспечение, работающее в режиме ядра, имеет полный доступ ко всем данным операционной системы. Этот недостаток стал одной из причин введения в Windows механизма подписи драйверов, который выводит предупреждение пользователю при попытке добавления автоматически настраиваемого (Plug and Play) драйвера, не имеющего подписи (или, при определенной настройке, блокирует добавление такого драйвера). Помимо этого верификатор драйверов — Driver Verifier — помогает создателям драйверов выискивать просчеты (например, переполнение буферов или допущение утечек памяти), способные повлиять на безопасность или стабильность работы системы.

Пользовательские приложения осуществляют переключение из пользовательского режима в режим ядра. Переход из режима пользователя в режим ядра осуществляется за счет использования специальной инструкции процессора, которая заставляет процессор переключиться в режим ядра и войти в код диспетчеризации системных служб, вызывающий соответствующую внутреннюю функцию в Ntoskrnl.exe или в Win32k.sys. Перед тем как вернуть управление пользовательскому потоку, процессор переключается в прежний, пользовательский режим работы.

Таким образом, пользовательский поток вполне может выполняться часть времени в пользовательском режиме, а другую часть времени — в режиме ядра. Фактически, из-за того что основная масса графики и оконная система также работают в режиме ядра, приложения, интенсивно использующие графику, проводят большую часть своего времени в режиме ядра, нежели в пользовательском режиме. Более сложные приложения могут использовать такие новые технологии, как Direct2D и создание составных изображений (compositing), которые проводят основной объем вычислений в пользовательском режиме и отправляют ядру только исходные данные поверхностей, сокращая время, затрачиваемое на переходы между пользовательскими режимами и режимами ядра.

Виртуальная память

В Windows реализована система виртуальной памяти, которая образует плоское (линейное) адресное пространство. Она создает каждому процессу иллюзию того, что у него есть достаточно большое и закрытое от других процессов адресное пространство. Виртуальная память дает логическое представление, которое не обязательно соответствует структуре физической памяти. В период выполнения диспетчер памяти, используя аппаратную поддержку, транслирует, или **проецирует** (maps), виртуальные адреса на физические, по которым реально хранятся данные. Управляя проектированием и защитой страниц памяти, операционная система гарантирует, что ни один процесс не помешает другому и не сможет повредить данные самой операционной системы.

Поскольку в большинстве компьютеров объем физической памяти намного меньше от общего объема виртуальной памяти, задействованной процессами, диспетчер памяти перемещает (подкачивает) часть содержимого памяти на диск. Подкачки данных на диск освобождает физическую память для других процессов или операционной системы. Когда поток обращается к сброшенной на диск страницы виртуальной памяти, диспетчер памяти загружает эту информацию с диска обратно в память. Для использования преимуществ подкачки в программах никакого дополнительного кода не нужно, потому что диспетчер памяти опирается на аппаратную поддержку этого механизма.

Размер виртуального адресного пространства зависит от конкретной аппаратной платформы. На 32-разрядных системах теоретический максимум для общего виртуального адресного пространства составляет 4 Гб. По умолчанию Windows выделяет нижнюю половину этого пространства (в диапазоне адресов от x00000000 к x7FFFFFFF) процессам, а вторую половину (в диапазоне адресов от x80000000 к xFFFFFFFF) использует в своих целях

Виртуальная память процесса:

От 2 ГБ до 2 ТБ. Кратна 64 КБ – гранулярность памяти пользовательского режима. Информацию о гранулярности можно получить с помощью GetSystemInfo().

Часть виртуальной памяти процесса, которая находится резидентно в физической памяти, называется рабочим набором – Working Set. Диапазон рабочего набора устанавливается функцией SetProcessWorkingSetSize(). Стандартный минимальный рабочий набор – 50 страниц по 4 КБ (200 КБ), стандартный максимальный рабочий набор – 345 страниц по 4 КБ (1380 КБ).

1. n Управление памятью в пользовательском режиме
a. Страничная виртуальная память:
b. Выделение: VirtualAlloc(), VirtualAllocEx(), VirtualAllocExNuma(), VirtualFree(), VirtualFreeEx(). Гранулярность в user mode – 64 КБ.
c. Защита страниц: VirtualProtect(), VirtualProtectEx().
d. Фиксация страниц в физической памяти: VirtualLock(), VirtualUnlock().
e. Информация: VirtualQuery(), VirtualQueryEx().
2. Куча (свалка) – Heap:
a. Создание: HeapCreate(), HeapDestroy().
b. Выделение: HeapAlloc(), HeapReAlloc(), HeapSize(), HeapFree(). Гранулярность – 8 байтов на x86, 16 байтов на x64.
c. Информация: HeapValidate(), HeapWalk(), HeapQueryInformation(), HeapSetInformation().
d. Кучи процесса: GetProcessHeap() – стандартная куча равная 1 MB, GetProcessHeaps() – все кучи процесса.
3. Отображение файлов в память – File Mapping:
a. Объект ядра, описывающий отображение фрагмента файла в диапазон виртуальных адресов, называется разделом (Section Object).
4. Управление памятью в режиме ядра
a. Пулы памяти – Memory Pools
b. Списки предыстории – Look-aside Lists
c. Представление объектов ядра в памяти
d. Фиксация данных в физической памяти
e. Таблицы описания памяти – Memory Descriptor Lists

Архитектура приложения в пользовательском режиме работы и в режиме ядра ОС Windows. Основные модули ОС Windows.

Пользовательские приложения не вызывают напрямую системные службы Windows. Вместо этого ими используется одна или несколько DLL-библиотек подсистемы. Эти библиотеки экспортируют документированный интерфейс, который может быть использован программами, связанными с данной подсистемой. Например, API-функции Windows реализованы в DLL-библиотеках подсистемы Windows, таких, как **Kernel32.dll, Advapi32.dll, User32.dll и Gdi32.dll.**

Ntdll.dll является специальной библиотекой системной поддержки, предназначенной, главным образом, для использования DLL-библиотек подсистем. В ней содержатся функции двух типов:

1. функции-заглушки, обеспечивающие переходы от диспетчера системных служб к системным службам исполняющей системы Windows;

Код внутри функции содержит зависящую от конкретной архитектуры инструкцию, осуществляющую переход в режим ядра для вызова диспетчера системных служб, который после проверки ряда параметров вызывает настоящую системную службу режима ядра, реальный код которой содержится в файле **Ntoskrnl.exe.**

2. вспомогательные внутренние функции, используемые подсистемами, DLL-библиотеками подсистем и другими исходными образами.

Основные системные файлы Windows

Имя файла	Компоненты
Ntoskrnl.exe	Исполняющая система и ядро
Ntkrnlpa.exe (только в 32-разрядных системах)	Исполняющая система и ядро с поддержкой расширения физического адреса — Physical Address Extension (PAE), позволяющего 32-разрядным системам осуществлять адресацию вплоть до 64 Гб физической памяти и помечать память как не содержащую исполняемый код
Hal.dll	Уровень аппаратных абстракций
Win32k.sys	Часть подсистемы Windows, работающей в режиме ядра
Ntdll.dll	Внутренние вспомогательные функции и заглушки диспетчера системных служб к исполняющим функциям
Kernel32.dll, Advapi32.dll, User32.dll, Gdi32.dll	Основные Windows-подсистемы DLL-библиотек

3. Системный реестр операционной системы Windows. Структура и главные разделы. Точки автозапуска программ. Средства редактирования реестра Windows. Функции работы с реестром из приложения.

Реестр Windows— иерархически построенная база данных параметров и настроек, состоящая из ульев.В Windows элементы реестра хранятся в виде раздельных структур. Реестр подразделяется на составные части, которые разработчики этой операционной системы назвали ульями (hives) по аналогии с ячеистой структурой пчелиного улья. Улей представляет собой совокупность вложенных ключей и параметров, берущую начало в вершине иерархии реестра. Отличие ульев от других групп ключей состоит в том, что они являются постоянными компонентами реестра. *Ульи не создаются динамически при загрузке операционной системы и не удаляются при ее остановке.*

На втором уровне располагаются разделы или ключи реестра (Registry Keys), на третьем – подразделы (Subkeys) и на четвертом и далее – параметры (Values).

Реестр содержит данные, к которым Windows постоянно обращается во время загрузки, работы и её завершения, а именно:

- профили всех пользователей, то есть их настройки;
- конфигурация оборудования, установленного в операционной системе.
- данные об установленных программах и типах документов, создаваемых каждой программой;
- свойства папок и значков программ;
- данные об используемых портах.

Корневой раздел	Аббревиатура	Ссылка	Описание
HKEY_CURRENT_USER	HKCU	Подраздел в разделе HKEY_USERS, соответствующий текущему, вошедшему в систему пользователю	Хранит данные, связанные с текущим пользователем, вошедшим в систему.
HKEY_USERS	HKU	Не является ссылкой	Хранит информацию обо всех учетных записях, имеющихся на машине
HKEY_CLASSES_ROOT	HKCR	Не является прямой ссылкой, а представляет собой объединенное представление о разделе HKLM\SOFTWARE\Classes и о разделе HKEY_USERS\<*SID*>\SOFTWARE\Classes	Хранит файловые связи и информацию о регистрации объектов, относящихся к модели компонентных объектов — Component Object Model (COM)
HKEY_LOCAL_MACHINE	HKLM	Не является ссылкой	Хранит информацию, связанную с системой
HKEY_PERFORMANCE_DATA	HKPD	Не является ссылкой	Хранит информацию о производительности
HKEY_CURRENT_CONFIG	HKCC	HKLM\SYSTEM\ CurrentControlSet\ Hardware Profiles\Current	Хранит определенную информацию о текущем профиле оборудования

Корневой раздел **HKCU** содержит данные, относящиеся к персональным настройкам и программной конфигурации локально вошедшего в систему пользователя. Он указывает на пользовательский профиль текущего вошедшего в систему пользователя, находящийся на жестком диске в файле \Users\ <*имя_пользователя*>\Ntuser.dat

Раздел **HKU** содержит подраздел для каждого загруженного профиля пользователя и использует базу данных классов, зарегистрированных в системе. Он также содержит подраздел HKU\.DEFAULT, связанный с профилем системы. Место, в котором система хранит профили, определяется параметром реестра HKLM\ Software\Microsoft\Windows NT\CurrentVersion\ProfileList\ProfilesDirectory, значение которого по умолчанию установлено в %SystemDrive%\Users. В разделе ProfileList также хранится список профилей, имеющихся в

системе. Информация для каждого профиля размещена в подразделе, имя которого отображает идентификатор безопасности security identifier (SID), — той учетной записи, которой соответствует профиль.

Таблица 4.4. Подразделы HKEY_CURRENT_USER

Подраздел	Описание
AppEvents	Связи между событиями и звуками
Console	Настройки окна командной строки (например, ширина, высота и цветовые решения)
Control Panel	Заставка экрана, схема рабочего стола, настройки клавиатуры и мыши, а также доступность и региональные настройки
Environment	Определения переменных среды окружения
EUDC	Информация о символах, определенных конечным пользователем
Identities	Информация об учетной записи почты Windows
Keyboard Layout	Настройки раскладки клавиатуры (например, US. или UK.)
Network	Настройки и отображения сетевого драйвера
Printers	Настройки подключения принтера
Software	Предпочтения пользователя в отношении программного обеспечения
Volatile Environment	Временные определения переменных среды окружения

Раздел **HKEY_CLASSES_ROOT** состоит из трех типов информации:

1) ассоциации с расширениями файлов;
2) регистрации COM-классов;
3) виртуализированный корневой раздел реестра системы для управления учетными записями пользователей — User Account Control (UAC)

Данные, хранящиеся в разделе **HKEY_CLASSES_ROOT**, поступают из двух источников:

1) из данных о регистрации классов для отдельного пользователя, находящихся в разделе HKCU\SOFTWARE\Classes (который отображается на файл на жестком диске \Users\<*имя_пользователя*>\AppData\Local\Microsoft\Windows\Usrclass.dat);
2) из данных о регистрации классов в масштабе всей системы, находящихся в разделе HKLM\SOFTWARE\Classes.

Причина отделения регистрационных данных, относящихся к каждому пользователю, от регистрационных данных, распространяемых на всю систему, объясняется тем, что эти настройки могут содержаться в перемещаемых профилях. Тем самым также закрывается дыра в системе безопасности: непривилегированный пользователь не может изменить или удалить разделы в общесистемной версии HKEY_CLASSES_ROOT и не может повлиять на функционирование приложений в системе.

HKLM является корневым разделом, в котором содержатся общесистемные подразделы конфигурации: BCD00000000, COMPONENTS (загружаемый в динамическом режиме по мере необходимости), HARDWARE, SAM, SECURITY, SOFTWARE и SYSTEM.

Подраздел HKLM\BCD00000000 содержит информацию из базы данных загрузочной конфигурации — Boot Configuration Database (BCD), — загружаемую в качестве куста реестра.

Подраздел **HKEY_CURRENT_CONFIG** является всего лишь ссылкой на текущий профиль оборудования, сохраненный в разделе HKLM\SYSTEM\CurrentControlSet\Hardware Profiles\Current. Профили оборудования в Windows больше не поддерживаются, но раздел по-прежнему существует для поддержки старых версий.

HKEY_PErForMANcE_DATA

Реестр является механизмом, используемым в Windows для доступа к значениям счетчиков производительности, относящихся либо к операционной системе, либо к серверным приложениям. Одним из сопутствующих преимуществ предоставления доступа к счетчикам производительности через реестр является то, что удаленное отслеживание производительности работает «бесплатно», поскольку удаленный доступ к реестру легко обеспечивается с помощью обычных API-функций реестра.

Можно получить непосредственный доступ к информации счетчиков производительности в реестре, открыв специальный раздел HKEY_PERFORMANCE_DATA и запросив находящиеся в нем параметры.

Просматривая реестр в редакторе реестра, вы этот раздел не найдете, он доступен только программным способом, через имеющиеся в Windows функции реестра, например через RegQueryValueEx.

Информация о производительности на самом деле в реестре не хранится, функции реестра используют этот раздел для поиска информации у поставщиков данных о производительности.

Внутреннее устройство реестра

Кусты

На диске реестр не является обычным большим файлом, а представляет собой набор отдельных файлов, которые называются кустами. Каждый куст содержит дерево реестра, у которого есть раздел, служащий ему корнем или отправной точкой дерева. Подразделы и их параметры находятся ниже корня. Можно подумать, что корневые разделы, отображаемые в редакторе реестра, соответствуют корневым разделам в кустах, но так

бывает не всегда. В табл. 4.5 перечислены кусты реестра и имена их файлов при хранении на диске. Путевые имена всех кустов, за исключением тех, которые используются для профилей пользователей, кодируются в диспетчере конфигурации. По мере того как диспетчер конфигурации загружает кусты, включая профили системы, он записывает путь к каждому кусту в параметрах подраздела *HKLM\SYSTEM\CurrentControlSet \ Control\Hivelist*, удаляя путь при выгрузке куста. Он создает корневые разделы, связывает эти кусты вместе, чтобы построить структуру реестра, с которой вы знакомы и которая показывается редактором реестра.
Вы заметите, что некоторые из этих кустов, перечисленные в табл. 4.5, могут изменяться и не имеют связанных с ними файлов. Система создает эти кусты и управляет ими целиком в памяти, поэтому такие кусты являются ***временными***.

Таблица 4.5. Соответствие файлов на диске путям в реестре

Путь куста реестра	Путь файла куста
HKEY_LOCAL_MACHINE\BCD00000000	\Boot\BCD
HKEY_LOCAL_MACHINE\COMPONENTS	%SystemRoot%\System32\Config\ Components
HKEY_LOCAL_MACHINE\SYSTEM	%SystemRoot%\System32\Config\System
HKEY_LOCAL_MACHINE\SAM	%SystemRoot%\System32\Config\Sam
HKEY_LOCAL_MACHINE\SECURITY	%SystemRoot%\System32\Config\Security
HKEY_LOCAL_MACHINE\SOFTWARE	%SystemRoot%\System32\Config\ Software
HKEY_LOCAL_MACHINE\HARDWARE	Непостоянный куст
HKEY_USERS\<*SID учетной записи локальной службы*>	%SystemRoot%\ServiceProfiles\ LocalService\Ntuser.dat
HKEY_USERS\<*SID учетной записи сетевой службы*>	%SystemRoot%\ServiceProfiles\ NetworkService\NtUser.dat
HKEY_USERS\<*SID имени пользователя*>	\Users\<username>\Ntuser.dat
HKEY_USERS\<*SID имени пользователя*>_Classes	\Users\<username>\AppData\Local\ Microsoft\Windows\Usrclass.dat
HKEY_USERS\.DEFAULT	%SystemRoot%\System32\Config\Default

Система создает непостоянные кусты при каждой своей загрузке. В качестве примера непостоянного куста можно привести HKLM\ HARDWARE, в котором хранится информация о физических устройствах и выделенных этим устройствам ресурсах. Выделение ресурсов и определение установленного оборудования проводятся при каждой загрузке системы, поэтому хранить эти данные на диске было бы нелогично.

Ограничения размера куста
В некоторых случаях размеры куста ограничиваются (напр. для куста HKLM\SYSTEM). Windows поступает таким образом, потому что Winload считывает весь куст HKLM\SYSTEM в физическую память практически сразу же после запуска процесса загрузки, когда разбиение на страницы виртуальной памяти еще не включено. Программа Winload также загружает в физическую память Ntoskrnl и драйверы устройств загрузки, поэтому она должна ограничивать физическую память, выделяемую HKLM\SYSTEM. На 32-разрядных системах Winload позволяет кусту быть размером до 400 Мбайт или размером в половину объема физической памяти системы, в зависимости от того, какой из размеров меньше. На системах x64 нижняя граница составляет 1,5 Гбайт. На системах Itanium она составляет 32 Мбайт.

Символические ссылки реестра
Специальный тип раздела, известный как символические ссылки реестра, позволяет диспетчеру конфигурации связывать разделы с целю организации реестра. Символическая ссылка является разделом, перенаправляющим диспетчер конфигурации на другой раздел. Таким образом, раздел HKLM\SAM является символической ссылкой на раздел в корне куста SAM. Символические ссылки создаются путем указания функции RegCreateKey или функции RegCreateKeyEx параметра REG_CREATE_LINK. Внутри диспетчер конфигурации создаст параметр REG_LINK, называемый SymbolicLinkValue, который будет содержать путь к целевому разделу. Поскольку этот параметр относится к типу REG_LINK, а не к типу REG_SZ, он не будет виден в Regedit, но тем не менее он будет частью куста реестра, хранящегося на диске.

Структура куста
Диспетчер конфигурации логически делит куст на распределяемые единицы, называемые блоками, способом, похожим на то, как файловая система делит диск на кластеры. По определению, размер блока реестра составляет 4096 байт (4 Кб). Когда куст расширяется за счет новых данных, он всегда расширяется за счет увеличения количества блоков. Первый блок куста называется базовым блоком. Базовый блок включает в себя глобальную информацию о кусте, в которую входят:
сигнатура regf, которая идентифицирует файл как куст;
- обновляемые последовательные номера;
- отметка времени, показывающая, когда в последний раз в отношении куста
- применялась операция записи1;
- информация о ремонте или восстановлении реестра, производимом Winload;
- номер версии формата куста;
- контрольная сумма;

- внутр. файловое имя файла куста (\Device\HarddiskVolume1\WINDOWS\SYSTEM32\CONFIG\SAM).

Windows упорядочивает данные реестра, которые куст хранит в контейнерах, называемых **ячейками**. В ячейке может храниться *раздел, параметр, дескриптор безопасности, список подразделов или список параметров раздела*. Четырехбайтовый символьный тег в начале данных ячейки описывает тип данных в виде сигнатуры.

Тип данных	Тип структуры	Описание
Ячейка раздела	CM_KEY_NODE	Ячейка, содержащая раздел реестра, которая также называется узлом раздела (key node). В ячейке раздела содержатся: • сигнатура (kn для раздела, kl для узла ссылки); • отметка времени самого последнего обновления раздела; • индекс ячейки, в которой содержится родительский раздел данного раздела; • индекс ячейки списка подразделов, идентифицирующего подразделы данного раздела; • индекс ячейки дескриптора безопасности данного раздела; • индекс ячейки строки, определяющей имя класса данного раздела; • имя раздела (например, CurrentControlSet). В ней также хранится кэшированная информация, например количество подразделов данного раздела.
Ячейка параметра	CM_KEY_VALUE	Ячейка, содержащая информацию о параметре раздела. Эта ячейка включает сигнатуру (kv), тип параметра (например, REG_ DWORD или REG_BINARY) и имя параметра (например, Boot-Execute). Ячейка параметра содержит также индекс той ячейки, в которой содержатся данные параметра
Ячейка списка подразделов	CM_KEY_INDEX	Ячейка, состоящая из списка индексов ячеек разделов, представляющих собой подразделы общего родительского раздела
Ячейка списка параметров	CM_KEY_INDEX	Ячейка, состоящая из списка индексов ячеек параметров, представляющих собой параметры общего родительского раздел
Ячейка дескриптора безопасности	CM_KEY_ SECURITY	Ячейка, содержащая дескриптор безопасности. Ячейки дескрипторов безопасности включают сигнатуру (ks), помещаемую в заголовок ячейки, и счетчик ссылок, в который записывается количество узлов раздела, совместно использующих дескриптор безопасности. Ячейки дескриптора безопасности могут совместно использоваться несколькими ячейками раздела

Приемник — это размер новой ячейки, округленный вверх до следующей границы блока или страницы, в зависимости от того, что выше. Система рассматривает любое пространство между окончанием ячейки и окончанием приемника в качестве свободного пространства, которое она может выделить другим ячейкам. У приемников также есть заголовки, в которых содержится сигнатура, hbin и поле, в которое записывается смещение приемника в файле куста и размер приемника.

Когда система добавляет ячейки в куст и удаляет их оттуда, куст может содержать пустые приемники, перемежающиеся активными приемниками (схоже с фрагментацией диска). Когда приемник становится пустым, диспетчер конфигурации присоединяет к пустому приемнику любые примыкающие пустые приемники *для формирования как можно более крупного и непрерывного пустого приемника*. Диспетчер конфигурации также *присоединяет друг к другу примыкающие удаленные ячейки для формирования более крупных свободных ячеек*.

Ссылки, образующие структуру куста, называются **индексами ячеек.** Индекс ячейки является смещением ячейки в файле куста за вычетом размера базового блока. Таким образом, индекс ячейки похож на указатель из одной ячейки на другую ячейку, который диспетчер конфигурации интерпретирует относительно начала куста.

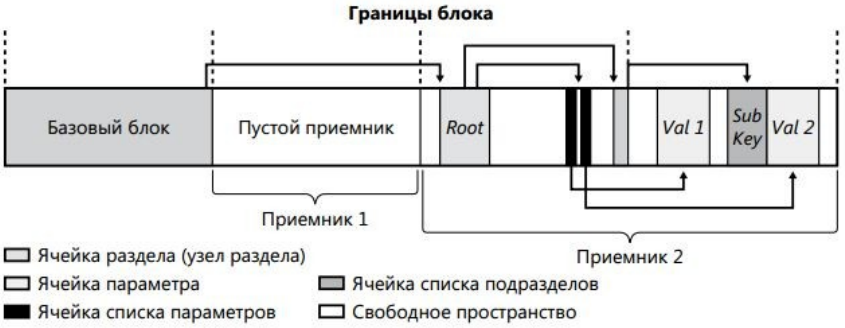


Рис. 4.3. Внутренняя структура куста реестра

В том, чем отличаются друг от друга ячейки и приемники, нетрудно запутаться. Поэтому, чтобы разобраться в различиях, давайте посмотрим на пример плана простого куста реестра. Приводимый в качестве примера на рис. 4.3 файл куста реестра содержит базовый блок и два приемника. Первый приемник пустой, а второй приемник содержит несколько ячеек. Логически, у куста имеются только два раздела: Root и его подраздел Sub Key. У раздела Root имеются два параметра, Val 1 и Val 2. Ячейка списка подразделов определяет местонахождение подраздела, подчиненного разделу Root, а ячейка списка параметров определяет местонахождение параметров раздела Root. Пустые места во втором приемнике являются пустыми ячейками. На рис. 4.3 не показаны ячейки безопасности для двух разделов, которые присутствовали бы в кусте. Для оптимизации поисков параметров и подразделов *диспетчер конфигурации сортирует ячейки списка* подразделов в алфавитном порядке.

Отображение ячеек

Если кусты не растут, диспетчер конфигурации может выполнить всю свою работу по управлению реестром в той версии куста, которая содержится в памяти, как будто куст является файлом. Благодаря индексу ячейки диспетчер конфигурации может *вычислить местонахождение ячейки в памяти путем простого прибавления индекса ячейки, который является смещением в файле куста, к базе образа куста в памяти*. В самом начале загрузки системы именно этим и занимается Winload с кустом SYSTEM: Winload считывает весь куст SYSTEM в память в качестве куста, доступного только для чтения, и прибавляет индексы ячеек к базе образа куста в памяти для определения местонахождения ячеек. Кусты разрастаются по мере добавления новых разделов и параметров, стало быть, система должна выделять пулы выгружаемой памяти (не обязательно непрерывной) для хранения новых приемников, содержащих добавленные разделы и параметры.

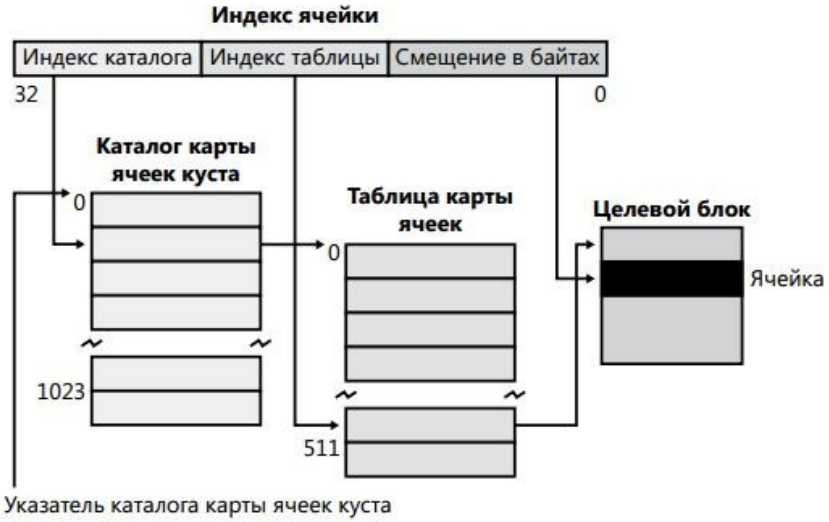


Рис. 4.4. Структура индекса ячейки

Диспетчер конфигурации использует двухуровневую схему (рис. 4.4), в которой в качестве ввода берется индекс ячейки (то есть смещение в файле куста), а возвращается в качестве вывода как адрес в памяти того блока, который занят индексом ячейки, так и адрес в памяти того блока, который занят ячейкой. Следует помнить, что приемник может содержать один или несколько блоков и что кусты прирастают количеством приемников, поэтому Windows всегда представляет приемник в виде непрерывной области памяти. Поэтому все блоки внутри приемника оказываются внутри одного и того же представления диспетчера кэша.

Для реализации отображения диспетчер конфигурации логически делит индекс ячейки на два поля, точно так же, как диспетчер памяти делит на поля виртуальный адрес. Windows интерпретирует первое поле индекса ячейки, как индекс в каталоге отображения ячеек, принадлежащем кусту. Каталог отображения ячеек содержит 1024 записи, каждая из которых ссылается на таблицу отображения ячеек, состоящую из 512 записей отображений. Запись в этой таблице отображения ячеек определяется вторым полем в индексе ячейки. Эта запись определяет адреса приемника и блока памяти ячейки. Не все приемники обязательно проецируются на память, и если при поиске ячейки выдается адрес 0, диспетчер конфигурации отображает приемник в памяти,

убирая, если нужно, отображение другого приемника в обслуживаемом этим диспетчером LRU-списке отображения.

Средства редактирования: regedit.exe, reg.exe.
regedit.exe – редактор реестра (в виде проводника).
reg.exe - редактирования системного реестра из командной строки.

Одна из многих возможностей реестра – это возможность задать **автозапуск программ** при старте ОС. Можно сделать автозапуск как для одного пользователя, так и для всех.
В реестре Windows 7 автозагрузка представлена в нескольких ветвях:
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run] - программы, запускаемые при входе в систему.
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce] - программы, запускаемые только один раз при входе пользователя в систему. После этого ключи программ автоматически удаляются из данного раздела реестра. Программы, которые запускаются в этом разделе, запускаются для всех пользователей в системе.
[HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run] - программы, которые запускаются при входе текущего пользователя в систему
[HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\RunOnce] - программы, которые запускаются только один раз при входе текущего пользователя в систему. После этого ключи программ автоматически удаляются из данного раздела реестра.

Например, чтобы автоматически запускать Блокнот при входе текущего пользователя, открываем Редактор реестра (regedit.exe), переходим в раздел ***[HKEY_CURRENT_USER\Software\Microsoft\Windows \CurrentVersion\Run]*** и добавляем ключ: ***"NOTEPAD.EXE"="C:\WINDOWS\System32\notepad.exe"***

Использование групповой политики для автозапуска
Откройте оснастку "Групповая политика" (gpedit.msc), перейдите на вкладку "Конфигурация компьютера - Административные шаблоны - Система". В правой части оснастки перейдите на пункт «Вход в систему».
По умолчанию эта политика не задана, но вы можете добавить туда программу: включаем политику, нажимаем кнопку "Показать - Добавить", указываем путь к программе, при этом если запускаемая программа находится в папке ..WINDOWS\System32\ то можно указать только название программы, иначе придется указать полный путь к программе.

Функции для работы с реестром

RegCloseKey Закрывает описатель ключа реестра.
RegCreateKeyEx Создает заданный ключ реестра.
RegDeleteKey/RegDeleteKeyEx Удаляет подключ и его значения
RegEnumKeyEx Перечисляет ключи заданного открытого ключа реестра.
RegEnumValue Перечисляет значения ключей заданного открытого ключа реестра.
RegGetValue Получает тип данных и сами данные значение ключа реестра.
RegLoadKey Создает подключ в HKEY_USERS или HKEY_LOCAL_MACHINE и сохраняет заданную информацию из файла в этот подключ.
RegOpenKeyEx Открывает заданный ключ реестра
RegSaveKey/RegSaveKeyEx Сохраняет заданный ключ и его подключи в файл.

4. Понятие окна в ОС Windows. Основные элементы окна. Понятие родительского и дочернего окна. Структура программы с событийным управлением. Минимальная программа для ОС Windows с окном на экране. Создание и отображение окна.

Понятие окна в ОС Windows.

Окно — графически выделенная часть экрана, принадлежащая какому-либо объекту, с которым работает пользователь. Окна могут иметь как произвольные, так и фиксированные (это характерно для диалоговых окон) размеры. Окно может занимать весь экран или только его часть. При этом на экране может быть одновременно выведено несколько (любое количество) окон.

Основные элементы окна. Понятие родительского и дочернего окна.

Каждое окно, создаваемое приложением, имеет **родительское окно**. При этом само оно по отношению к родительскому является дочерним. Какое окно является "основателем рода", т. е. родительским для всех остальных окон? Окна всех приложений располагаются в окне, представляющем собой поверхность рабочего стола Workplace Shell . Это окно, которое называется Desktop Window , создается автоматически при запуске операционной системы. Однако окно Desktop Window само по себе является дочерним по отношению к другому окну - окну Object Window . Это окно не отображается и используется системой Presentation Manager для собственных нужд.

Родительское окно может иметь несколько **дочерних окон**, которые при этом называются окнами-братьями (или окнами-сестрами). Обратное неверно, т. е. у каждого дочернего окна может быть только одно родительское окно. Каждое дочернее окно, в свою очередь, может выступать в роли родительского окна, создавая свои дочерние окна.

Важной особенностью дочерних окон является то, что они всегда располагаются внутри своего родительского окна. Если пользователь попытается переместить дочернее окно за пределы родительского (например, при помощи мыши), будет нарисована только часть дочернего окна.

В том случае, когда в одном родительском окне создано несколько дочерних окон, они могут перекрывать друг друга.

Если пользователь перемещает родительское окно, то дочернее окно будет перемещаться вместе с ним.

Когда пользователь изменяет размеры родительского окна, дочернее окно может отображаться не полностью. Если же пользователь минимизирует родительское окно, дочернее окно исчезает с поверхности экрана. При минимизации дочернего окна оно отображается в родительском окне в виде пиктограммы.

При уничтожении родительского окна все его дочерние окна уничтожаются автоматически.

Структура программы с событийным управлением.

Минимальная программа для ОС Windows с окном на экране.

```
#include <windows.h>
int APIENTRY WinMain(HINSTANCE hInstance,
             HINSTANCE hPrevInstance, LPTSTR lpCmdLine, int nCmdShow)
{
             WNDCLASSEX wcex; HWND hWnd; MSG msg;
             wcex.cbSize = sizeof(WNDCLASSEX);
             wcex.style = CS_DBLCLKS;
             wcex.lpfnWndProc = WndProc;
             wcex.cbClsExtra = 0;
             wcex.cbWndExtra = 0;
             wcex.hInstance = hInstance;
             wcex.hIcon = LoadIcon(NULL, IDI_APPLICATION);
             wcex.hCursor = LoadCursor(NULL, IDC_ARROW);
             wcex.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
             wcex.lpszMenuName = NULL;
             wcex.lpszClassName = "HelloWorldClass";
             wcex.hIconSm = wcex.hIcon;
...
             RegisterClassEx(&wcex);
             hWnd = CreateWindow("HelloWorldClass", "Hello, World!",
```

```
            WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, 0,
            CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
            ShowWindow(hWnd, nCmdShow);
            UpdateWindow(hWnd);
            while (GetMessage(&msg, NULL, 0, 0)) {
                  TranslateMessage(&msg);
                  DispatchMessage(&msg);
            }
            return (int)msg.wParam;
}
...
LRESULT CALLBACK WndProc(HWND hWnd, UINT message,
            WPARAM wParam, LPARAM lParam)
{
            switch (message) {
            case WM_LBUTTONDBLCLK:
            MessageBox(hWnd, "Hello, World!", "Message", MB_OK);
            break;
            case WM_DESTROY:
            PostQuitMessage(0);
            break;
            default:
            return DefWindowProc(hWnd, message, wParam, lParam);
            }
   return 0;
}
```

Создание и отображение окна.

Создание окна:

Функция CreateWindow:

```
HWND CreateWindow(
            LPCTSTR lpClassName,
            LPCTSTR lpWindowName,
            DWORD dwStyle,
            int x,
            int y,
            int nWidth,
            int nHeight,
            HWND hWndParent,
            HMENU hMenu,
            HANDLE hInstance,
            LPVOID lpParam
);
```

1. lpClassName — Имя класса
2. lpWindowName — Имя окна
3. dwStyle — Описывает стиль создаваемого окна
4. x — Позиция по горизонтали верхнего левого угла окна в экранной системе координат.
5. y — Позиция по вертикали верхнего левого угла окна в экранной системе координат.
6. nWidth — Ширина окна в пикселях.
7. nHeight — Высота окна в пикселях.
8. hWndParent — Дескриптор окна, которое является родителем данного.
9. hMenu — Дескриптор меню.
10. hInstance — Дескриптор экземпляра приложения с которым связано данное окно.
11. lpParam — указатель на определяемые пользователем данные.

Функция CreateWindow возвращает дескриптор созданного ею окна (значение типа HWND). Если создать окно не удалось, значение дескриптора равно нулю.

Отображение окна:

Сперва мы вызываем функцию ShowWindow и передаем ей дескриптор только что созданного окна, чтобы Windows знала, какое окно должно быть показано. Мы также передаем число, определяющее в каком виде будет показано окно (обычным, свернутым, развернутым на весь экран и т.д.). После отображения окна мы должны обновить его. Это делает функция UpdateWindow; она получает один аргумент, являющийся дескриптором обновляемого окна.

```
ShowWindow(MainWindowHandle, show);
UpdateWindow(MainWindowHandle)
```

5. Структура программы с событийным управлением. Структура события – оконного сообщения Windows. Очередь сообщений. Цикл приема и обработки сообщений. Процедура обработки сообщений. Процедуры посылки сообщений. Синхронные и асинхронные сообщения

Оконные сообщения

В этой главе я расскажу, как работает подсистема передачи сообщений в Windows применительно к приложениям с графическим пользовательским интерфейсом. Разрабатывая подсистему управления окнами в Windows 2000 и Windows 98, Microsoft преследовала две основные цели:
1) обратная совместимость с 16-разрядной Windows, облегчающая перенос существующих 16-разрядных приложений;
2) отказоустойчивость подсистемы управления окнами, чтобы ни один поток не мог нарушить работу других потоков в системе.
К сожалению, эти цели прямо противоречат друг другу. В 16-разрядной Windows передача сообщения в окно всегда осуществляется синхронно: отправитель не может продолжить работу, пока окно не обработает полученное сообщение. Обычно так и нужно. Но, если на обработку сообщения потребуется длительное время или если окно «зависнет», выполнение отправителя просто прекратится. А значит, такая операционная система не вправе претендовать на устойчивость к сбоям. Это противоречие было серьезным вызовом для команды разработчиков из Microsoft. В итоге было выбрано компромиссное решение, отвечающее двум вышеупомянутым целям.
Для начала рассмотрим некоторые базовые принципы. Один процесс в Windows может создать до 10 000 User-объектов различных типов — значков, курсоров, оконных классов, меню, таблиц клавиш-акселераторов и т. д. Когда поток из какого-либо процесса вызывает функцию, создающую один из этих объектов, последний переходит во владение процесса. Поэтому, если процесс завершается, не уничтожив данный объект явным образом, операционная система делает это за него. Однако два User-объекта *(окна и ловушки)* принадлежат только создавшему их потоку. И вновь, если поток создает окно или устанавливает ловушку, а потом завершается, операционная система автоматически уничтожает окно или удаляет ловушку. *Этот принцип принадлежности окон и ловушек создавшему их потоку оказывает существенное влияние на механизм функционирования окон: поток, создавший окно, должен обрабатывать все его сообщения*. Поясню данный принцип на примере. Допустим, поток создал окно, а затем прекратил работу. Тогда его окно уже не получит сообщение WM_DESTROY или WM_NCDESTROY, потому что поток уже завершился и обрабатывать сообщения, посылаемые этому окну, больше некому.
Это также означает, что каждому потоку, создавшему хотя бы одно окно, система выделяет очередь сообщений, используемую для их диспетчеризации. Чтобы окно в конечном счете получило эти сообщения, поток должен иметь собственный цикл выборки сообщений.

Очередь сообщений потока
У каждого потока должны быть очереди сообщений, полностью независимые от других потоков. Кроме того, для каждого потока нужно смоделировать среду, позволяющую ему самостоятельно управлять фокусом ввода с клавиатуры, активизировать окна, захватывать мышь и т. д.
Создавая какой-либо поток, система предполагает, что он не будет иметь отношения к поддержке пользовательского интерфейса. Это позволяет уменьшить объем выделяемых ему системных ресурсов. Но, как только поток обратится к той или иной GUI-функции (например, для проверки очереди сообщений или создания окна), система автоматически выделит ему дополнительные ресурсы, необходимые для выполнения задач, связанных с пользовательским интерфейсом. А если конкретнее, то система создает структуру **THREADINFO** и сопоставляет ее с этим потоком. Элементы этой структуры используются, чтобы обмануть поток — заставить его считать, будто он выполняется в среде, принадлежащей только ему. THREADINFO — это внутренняя (недокументированная) структура, идентифицирующая *очередь асинхронных сообщений потока (posted-message queue), очередь синхронных сообщений потока (sent-message queue), очередь ответных сообщений (reply-message queue), очередь виртуального ввода (virtualized input queue) и флаги пробуждения (wake flags);* она также включает ряд других переменных-членов, характеризующих локальное состояние ввода для данного потока. На рис. 26-1 показаны структуры THREADINFO, сопоставленные с тремя потоками.
Структура THREADINFO — фундамент всей подсистемы передачи сообщений; читая следующие разделы, время от времени посматривайте на эту иллюстрацию.

Посылка асинхронных сообщений в очередь потока

Когда с потоком связывается структура THREADINFO, он получает свой набор очередей сообщений. Если процесс создает три потока и все они вызывают функцию CreateWindow, то и наборов очередей сообщений будет тоже три. Сообщения ставятся в очередь асинхронных сообщений вызовом функции PostMessage:

*BOOL **PostMessage**(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam);*

При вызове этой функции система определяет, каким потоком создано окно, идентифицируемое параметром hwnd. Далее система выделяет блок памяти, сохраняет в нем параметры сообщения и записывает этот блок в очередь асинхронных сообщений данного потока. Кроме того, функция устанавливает флаг пробуждения QS_POSTMESSAGE (о нем — чуть позже). Возврат из PostMessage происходит сразу после того, как сообщение поставлено в очередь, поэтому вызывающий поток остается в неведении, обработано ли оно процедурой соответствующего окна. На самом деле вполне вероятно, что окно даже не получит это сообщение. Такое возможно, если поток, создавший это окно, завершится до того, как обработает все сообщения из своей очереди.

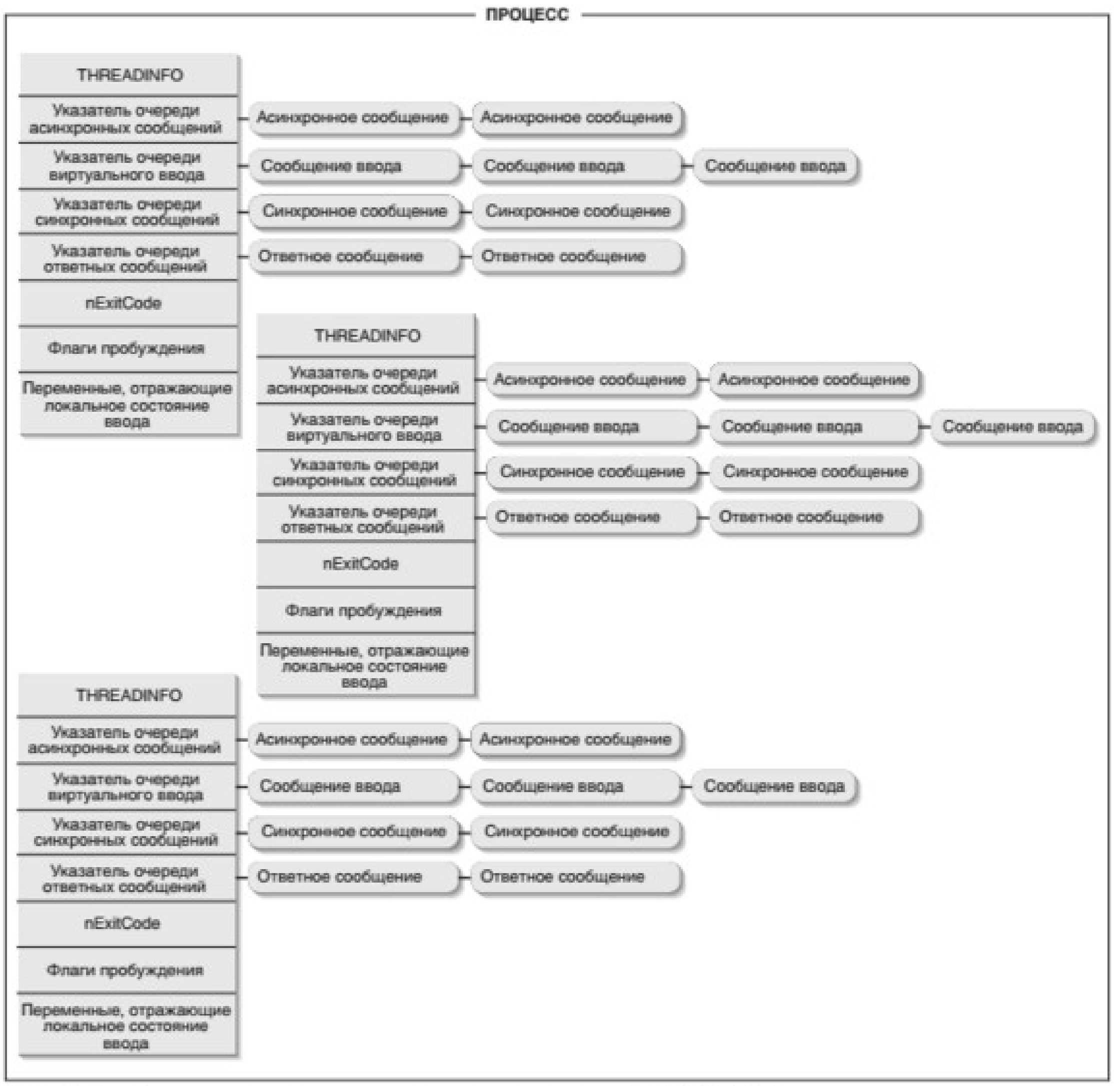


Рис. 26-1. *Три потока и соответствующие им структуры THREADINFO*

Сообщение можно поставить в очередь асинхронных сообщений потока и вызовом PostThreadMessage:

*BOOL **PostThreadMessage**(DWORD dwThreadId, UINT uMsg, WPARAM wParam, LPARAM lParam);*

Какой поток создал окно, можно определить с помощью GetWindowThreadProcessId:

*DWORD **GetWindowThreadProcessId**(HWND hwnd,PDWORD pdwProcessId);*

Она возвращает уникальный общесистемный идентификатор потока, который создал окно, определяемое параметром hwnd. Передав адрес переменной в параметре *pdwProcessId*, можно получить и уникальный общесистемный идентификатор процесса, которому принадлежит этот поток. Но обычно такой идентификатор

не нужен, и мы просто передаем NULL. Нужный поток идентифицируется первым параметром, *dwThreadId*. Когда сообщение помещено в очередь, элемент hwnd структуры MSG устанавливается как NULL. Применяется эта функция, когда приложение выполняет какую-то особую обработку в основном цикле выборки сообщений потока, — в этом случае он пишется так, чтобы после выборки сообщения функцией *GetMessage* (или PeekMessage) код в цикле сравнивал hwnd с NULL и, выполняя эту самую особую обработку, мог проверить значение элемента msg структуры MSG. Если поток определил, что сообщение не адресовано какому-либо окну, DispatchMessage не вызывается, и цикл переходит к выборке следующего сообщения.
Как и PostMessage, функция PostThreadMessage возвращает управление сразу после того, как сообщение поставлено в очередь потока. И вновь вызывающий поток остается в неведении о дальнейшей судьбе сообщения. И, наконец, еще одна функция, позволяющая поместить сообщение в очередь асинхронных сообщений потока:

*VOID **PostQuitMessage**(int nExitCode);*

Она вызывается для того, чтобы завершить цикл выборки сообщений потока. Ее вызов аналогичен вызову:

PostThreadMessage(GetCurrentThreadId(), WM_QUIT, nExitCode, 0);

Но в действительности PostQuitMessage не помещает сообщение ни в одну из очередей структуры THREADINFO. Эта функция просто устанавливает флаг пробуждения QS_QUIT (о нем я тоже расскажу чуть позже) и элемент nExitCode структуры THREADINFO. Так как эти операции не могут вызвать ошибку, функция PostQuitMessage не возвращает никаких значений (VOID).

Посылка синхронных сообщений окнуамс

Оконное сообщение можно отправить непосредственно оконной процедуре вызовом SendMessage:

*LRESULT **SendMessage**(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam);*

Оконная процедура обработает сообщение, ***и только по окончании обработки функция SendMessage вернет управление.***

Вот как работает SendMessage. Если поток вызывает SendMessage для посылки сообщения окну, созданному им же, то функция просто обращается к оконной процедуре соответствующего окна как к подпрограмме. Закончив обработку, оконная процедура передает функции SendMessage некое значение, а та возвращает его вызвавшему потоку. Однако, если поток посылает сообщение окну, созданному другим потоком, операции, выполняемые функцией SendMessage, значительно усложняются. Windows требует, чтобы оконное сообщение обрабатывалось потоком, создавшим окно. Поэтому, если вызвать SendMessage для отправки сообщения окну, созданному в другом процессе и, естественно, другим потоком, Ваш поток не сможет обработать это сообщение — ведь он не работает в адресном пространстве чужого процесса, а потому не имеет доступа к коду и данным соответствующей оконной процедуры. И действительно, Ваш поток приостанавливается, пока другой поток обрабатывает сообщение. Поэтому, чтобы один поток мог отправить сообщение окну, созданному другим потоком, система должна выполнить следующие действия:

Во-первых, переданное сообщение присоединяется к очереди сообщений потока-приемника, в результате чего для этого потока устанавливается флаг QS_SENDMESSAGE.

Во-вторых, если поток-приемник в данный момент выполняет какой-то код и не ожидает сообщений (через вызов GetMessage, PeekMessage или WaitMessage), переданное сообщение обработать не удастся — система не прервет работу потока для немедленной обработки сообщения. Но когда поток-приемник ждет сообщений, система сначала проверяет, установлен ли флаг пробуждения QS_SENDMESSAGE, и, если да, просматривает очередь синхронных сообщений, отыскивая первое из них. В очереди может находиться более одного сообщения. Скажем, несколько потоков одновременно послали сообщение одному и тому же окну. Тогда система просто ставит эти сообщения в очередь синхронных сообщений потока.

Итак, когда поток ждет сообщений, система извлекает из очереди синхронных сообщений первое и вызывает для его обработки нужную оконную процедуру. Если таких сообщений больше нет, флаг QS_SENDMESSAGE сбрасывается. Пока поток-приемник обрабатывает сообщение, поток, отправивший сообщение через SendMessage, простаивает, ожидая появления сообщения в очереди ответных сообщений. По окончании обработки значение, возвращенное оконной процедурой, передается асинхронно в очередь ответных сообщений потока-отправителя. Теперь он пробудится и извлечет упомянутое значение из ответного сообщения. Именно это значение и будет результатом вызова SendMessage. С этого момента поток-отправитель возобновляет работу в обычном режиме.

Ожидая возврата управления функцией SendMessage, поток в основном простаивает. Но кое-чем он может заняться: если другой поток посылает сообщение окну,созданному первым (ожидающим) потоком, система тут же обрабатывает это сообщение, не дожидаясь, когда поток вызовет GetMessage, PeekMessage или WaitMessage. Поскольку Windows обрабатывает межпоточные сообщения описанным выше образом, Ваш поток может зависнуть. Допустим, в потоке, обрабатывающем синхронное сообщение, имеется «жучок», из-за которого поток входит в бесконечный цикл. Что же произойдет с потоком, вызвавшим SendMessage?

Возобновится ли когда-нибудь его выполнение? Значит ли это, что ошибка в одном приложении «подвесит» другое? Ответ — да! Это верно даже в том случае, если оба потока принадлежат одному процессу.

Оконные сообщения

Избегать подобных ситуаций позволяют четыре функции, и первая из них —SendMessageTimeout:

*LRESULT **SendMessageTimeout**(HWND hwnd, UINT uMsg,WPARAM wParam, LPARAM lParam, UINT fuFlags, UINT uTimeout, PDWORD_PTR pdwResult);*

Она позволяет задавать отрезок времени, в течение которого Вы готовы ждать ответа от другого потока на Ваше сообщение. Ее первые четыре параметра идентичны параметрам функции SendMessage. В параметре *fuFlags* можно передавать флаги *SMTO_NORMAL (0), SMTO_ABORTIFHUNG, SMTO_BLOCK, SMTO_NOTIMEOUTIFNOTHUNG* или комбинацию этих флагов.

Флаг *SMTO_ABORTIFHUNG* заставляет SendMessageTimeout проверить, не завис ли поток-приемник2 , и, если да, немедленно вернуть управление. Флаг *SMTO_NOTIMEOUTIFNOTHUNG* сообщает функции, что она должна игнорировать ограничение по времени, если поток-приемник не завис. Флаг *SMTO_BLOCK* предотвращает обработку вызывающим потоком любых других синхронных сообщений до возврата из SendMessageTimeout. Флаг *SMTO_NORMAL* определен в файле WinUser.h как 0; он используется в том случае, если Вы не указали другие флаги.

Я уже говорил, что ожидание потоком окончания обработки синхронного сообщения может быть прервано для обработки другого синхронного сообщения. Флаг SMTO_BLOCK предотвращает такое прерывание. Он применяется, только если поток, ожидая окончания обработки своего сообщения, не в состоянии обрабатывать прочие синхронные сообщения. Этот флаг иногда приводит к взаимной блокировке потоков до конца таймаута. Так, если Ваш поток отправит сообщение другому, а тому нужно послать сообщение Вашему, ни один из них не сможет продолжить обработку, и оба зависнут.

Параметр *uTimeout* определяет таймаут — время (в миллисекундах), в течение которого Вы готовы ждать ответного сообщения. При успешном выполнении функция возвращает TRUE, а результат обработки сообщения копируется по адресу, указанному в параметре pdwResult. Кстати, прототип этой функции в заголовочном файле WinUser.h неверен. Функцию следовало бы определить как возвращающую значение типа BOOL, поскольку значение типа LRESULT на самом деле возвращается через ее параметр. Это создает определенные проблемы, так как SendMessageTimeout вернет FALSE, если Вы передадите неверный описатель окна или если закончится заданный период ожидания. Единственный способ узнать причину неудачного завершения функции — вызвать GetLastError. Последняя вернет 0 (ERROR_SUCCESS), если ошибка связана с окончанием периода ожидания. А если причина в неверном описателе, GetLastError даст код 1400 (ERROR_INVALID_WINDOW_HANDLE). Если Вы обращаетесь к SendMessageTimeout для посылки сообщения окну, созданному вызывающим потоком, система просто вызывает оконную процедуру, помещая возвращаемое значение в pdwResult. Операционная система считает поток зависшим, если он прекращает обработку сообщений более чем на 5 секунд.

SendMessageTimeout, не выполняется до тех пор, пока не заканчивается обработка сообщения, — ведь все эти операции осуществляются одним потоком. Теперь рассмотрим вторую функцию, предназначенную для отправки межпоточных сообщений:

*BOOL **SendMessageCallback**(HWND hwnd, UINT uMsg, WPARAM wParam,LPARAM lParam, SENDASYNCPROC pfnResultCallBack, ULONG_PTR dwData);*

И вновь первые четыре параметра идентичны параметрам функции SendMessage. При вызове Вашим потоком SendMessageCallback отправляет сообщение в очередь синхронных сообщений потока-приемника и тут же возвращает управление вызывающему (т. е. Вашему) потоку. *Закончив обработку сообщения, поток-приемник асинхронно отправляет свое сообщение в очередь ответных сообщений Вашего потока.* Позже система уведомит Ваш поток об этом, вызвав написанную Вами функцию; у нее должен быть следующий прототип:

*VOID CALLBACK **ResultCallBack**(HWND hwnd, UINT uMsg, ULONG_PTR dwData, LRESULT lResult);*

Адрес этой функции обратного вызова передается SendMessageCallback в параметре pfnResultCallBack. А при вызове ResultCallBack в первых двух параметрах передаются описатель окна, закончившего обработку сообщения, и код (значение) самого сообщения. Параметр *dwData* функции ResultCallBack всегда получает значение, переданное SendMessageCallback в одноименном параметре. (Система просто берет то, что указано там, и передает Вашей функции ResultCallBack.) Последний параметр функции ResultCallBack сообщает результат обработки сообщения, полученный от оконной процедуры. Поскольку SendMessageCallback, передавая сообщение другому потоку, немедленно возвращает управление, ResultCallBack вызывается после обработки сообщения потоком-приемником не сразу, а с задержкой. Сначала поток-приемник асинхронно ставит сообщение в очередь ответных сообщений потока-отправителя. Затем при первом же вызове потоком-отправителем любой из функций GetMessage, PeekMessage, WaitMessage или одной из Send-функций сообщение извлекается из очереди ответных сообщений, и лишь потом вызывается Ваша функция ResultCallback.

Существует и другое применение функции SendMessageCallback. В Windows предусмотрен метод, позволяющий разослать сообщение всем перекрывающимся окнам (overlapped windows) в системе; он состоит в том, что Вы вызываете SendMessage и в параметре hwnd передаете ей HWND_BROADCAST (определенный как –1). Этот метод годится только для широковещательной рассылки сообщений, возвращаемые значения которых Вас не интересуют, поскольку функция способна вернуть лишь одно значение, LRESULT. Но, используя SendMessageCallback, можно получить результаты обработки «широковещательного» сообщения от каждого перекрытого окна. Ваша функция SendMessageCallback будет вызываться с результатом обработки сообщения от каждого из таких окон. Если SendMessageCallback вызывается для отправки сообщения окну, созданному вызывающим потоком, система немедленно вызывает оконную процедуру, а после обработки сообщения — функцию ResultCallBack. После возврата из ResultCallback выполнение начинается со строки, следующей за вызовом SendMessageCallback.

Третья функция, предназначенная для передачи межпоточных сообщений:

*BOOL **SendNotifyMessage**(HWND hwnd, UINT uMsg, WPARAM wParam,LPARAM lParam);*

Поместив сообщение в очередь синхронных сообщений потока-приемника, она немедленно возвращает управление вызывающему потоку. Так ведет себя и PostMessage, помните? Но два отличия SendNotifyMessage от PostMessage все же есть.

Во-первых, если SendNotifyMessage посылает сообщение окну, созданному другим потоком, приоритет данного синхронного сообщения выше приоритета асинхронных сообщений, находящихся в очереди потока-приемника. Иными словами, сообщения, помещаемые в очередь с помощью SendNotifyMessage, всегда извлекаются до выборки сообщений, отправленных через PostMessage.

Во-вторых, если сообщение посылается окну, созданному вызывающим потоком, SendNotifyMessage работает точно так же, как и SendMessage, т. е. не возвращает управление до окончания обработки сообщения. Большинство синхронных сообщений посылается окну для уведомления — чтобы сообщить ему об изменении состояния и чтобы оно как-то отреагировало на это, прежде чем Вы продолжите свою работу. Например, *WM_ACTIVATE, WM_DESTROY, WM_ENABLE, WM_SIZE, WM_SETFOCUS, WM_MOVE* и многие другие сообщения — это просто уведомления, посылаемые системой окну в синхронном, а не асинхронном режиме. Поэтому система не прерывает свою работу только ради того, чтобы оконная процедура могла их обработать. Прямо противоположный эффект дает отправка сообщения WM_CREATE — тогда система ждет, когда окно закончит его обработку. Если возвращено значение –1, значит, окно не создано.

И, наконец, четвертая функция, связанная с обработкой межпоточных сообщений:

*BOOL **ReplyMessage**(LRESULT lResult);*

Она отличается от трех описанных выше. В то время как Send-функции используются посылающим сообщения потоком для защиты себя от зависания, *ReplyMessage вызывается потоком, принимающим оконное сообщение*. Вызвав ее, поток как бы говорит системе, что он уже получил результат обработки сообщения и что этот результат нужно упаковать и асинхронно отправить в очередь ответных сообщений потока-отправителя. Последний сможет пробудиться, получить результат и возобновить работу.

Поток, вызывающий ReplyMessage, передает результат обработки сообщения через параметр lResult. После вызова ReplyMessage выполнение потока-отправителя возобновляется, а поток, занятый обработкой сообщения, продолжает эту обработку. Ни один из потоков не приостанавливается — оба работают, как обычно. Когда поток, обрабатывающий сообщение, выйдет из своей оконной процедуры, любое возвращаемое значение просто игнорируется.

Заметьте: ReplyMessage надо вызывать из оконной процедуры, получившей сообщение, но не из потока, вызвавшего одну из Send-функций. Поэтому, чтобы написать «защищенный от зависаний» код, следует заменить все вызовы SendMessage вызовами одной из трех Send-функций и не полагаться на то, что оконная процедура будет вызывать именно ReplyMessage.

Учтите также, что вызов ReplyMessage при обработке сообщения, посланного этим же потоком, не влечет никаких действий. На это и указывает значение, возвращаемое ReplyMessage: TRUE — при обработке межпоточного сообщения и FALSE — при попытке вызова функции для обработки внутрипоточного сообщения. Если Вас интересует, является обрабатываемое сообщение внутрипоточным или межпоточным, вызовите функцию InSendMessage:

*BOOL **InSendMessage**();*

Она возвращает TRUE, если поток обрабатывает межпоточное синхронное сообщение, и FALSE — при обработке им внутрипоточного сообщения (синхронного или асинхронного). Возвращаемые значения функций InSendMessage и ReplyMessage идентичны. Есть еще одна функция, позволяющая определить тип сообщения, которое обрабатывается Вашей оконной процедурой:

*DWORD **InSendMessageEx**(PVOID pvReserved);*

Вызывая ее, Вы должны передать NULL в параметре pvReserved. Возвращаемое значение указывает на тип обрабатываемого сообщения. Значение ISMEX_NOSEND (0) говорит о том, что поток обрабатывает

внутрипоточное синхронное или асинхронное сообщение. Остальные возвращаемые значения представляют собой комбинацию битовых флагов:

ISMEX_SEND Поток обрабатывает межпоточное синхронное сообщение, посланное

ISMEX_NOTIFY Поток обрабатывает межпоточное синхронное сообщение, посланное

ISMEX_CALLBACK Поток обрабатывает межпоточное синхронное сообщение, посланное

ISMEX_REPLIED Поток обрабатывает межпоточное синхронное сообщение и уже вызвал ReplyMessage; поток-отправитель не блокируется

через SendMessage или SendMessageTimeout; если флаг ISMEX_REPLIED не установлен, поток-отправитель блокируется в ожидании ответа через SendNotifyMessage; поток-отправитель не ждет ответа и не блокируется через SendMessageCallback; поток-отправитель не ждет ответа и не блокируется

Передача данных через сообщения

В некоторых оконных сообщениях параметр lParam задает адрес блока памяти. Например, сообщение WM_SETTEXT использует lParam как указатель на строку (с нулевым символом в конце), содержащую новый текст для окна. Рассмотрим такой вызов:

SendMessage(FindWindow(NULL, "Calculator"), WM_SETTEXT, 0, (LPARAM) "A Test Caption");

Вроде бы все достаточно безобидно: определяется описатель окна Calculator и делается попытка изменить его заголовок на «A Test Caption». Но приглядимся к тому, что тут происходит.

В lParam передается адрес строки (с новым заголовком), расположенной в адресном пространстве Вашего процесса. Получив это сообщение, оконная процедура программы Calculator берет lParam и пытается манипулировать чем-то, что, «по ее мнению», является указателем на строку с новым заголовком. Но адрес в lParam указывает на строку в адресном пространстве Вашего процесса, а не программы Calculator. Вот Вам и долгожданная неприятность — нарушение доступа к памяти. Но если Вы все же выполните показанную ранее строку, все будет работать нормально. Что за наваждение? А дело в том, что система отслеживает сообщения WM_SETTEXT и обрабатывает их не так, как большинство других сообщений. При вызове SendMessage внутренний код функции проверяет, не пытаетесь ли Вы послать сообщение WM_SETTEXT. Если это так, функция копирует строку из Вашего адресного пространства в проекцию файла и делает его доступным другому процессу. Затем сообщение посылается потоку другого процесса. Когда поток-приемник готов к обработке WM_SETTEXT, он определяет адрес общей проекции файла (содержащей копию строки) в адресном пространстве своего процесса. Параметру lParam присваивается значение именно этого адреса, и WM_SETTEXT направляется нужной оконной процедуре. После обработки этого сообщения, проекция файла уничтожается. Не слишком ли тут накручено, а? К счастью, большинство сообщений не требует такой обработки — она осуществляется, только если сообщение посылается другому процессу. (Заметьте: описанная обработка выполняется и для любого сообщения, параметры wParam или lParam которого содержат указатель на какую-либо структуру данных.)

А вот другой случай, когда от системы требуется особая обработка, — сообщение WM_GETTEXT. Допустим, Ваша программа содержит код:

char szBuf[200];

SendMessage(FindWindow(NULL, "Calculator"), WM_GETTEXT,

Sizeof(szBuf), (LPARAM) szBuf);

WM_GETTEXT требует, чтобы оконная процедура программы Calculator поместила в буфер, на который указывает szBuf, заголовок своего окна. Когда Вы посылаете это сообщение окну другого процесса, система должна на самом деле послать два сообщения. Сначала — WM_GETTEXTLENGTH. Оконная процедура возвращает число символов в строке заголовка окна. Это значение система использует при создании проекции файла, разделяемой двумя процессами. Создав проекцию файла, система посылает для его заполнения сообщение WM_GET- TEXT. Затем переключается обратно на процесс, первым вызвавший функцию SendMessage, копирует данные из общей проекции файла в буфер, на который указывает szBuf, и заставляет SendMessage вернуть управление. Что ж, все хорошо, пока Вы посылаете сообщения, известные системе. А если мы определим собственное сообщение (WM_USER + x), собираясь отправить его окну другого процесса? Система не «поймет», что нам нужна общая проекция файла для корректировки указателей при их пересылке. Но выход есть — это сообщение WM_COPYDATA:

COPYDATASTRUCT cds;

SendMessage(hwndReceiver, WM_COPYDATA, (WPARAM) hwndSender, (LPARAM) &cds);

COPYDATASTRUCT — структура, определенная в WinUser.h:

typedef struct tagCOPYDATASTRUCT {

ULONG_PTR dwData;

DWORD cbData;

PVOID lpData;

} COPYDATASTRUCT;

Чтобы переслать данные окну другого процесса, нужно сначала инициализировать эту структуру. Элемент dwData резервируется для использования в Вашей программе. В него разрешается записывать любое значение. Например, передавая в другой процесс данные, в этом элементе можно указывать тип данных. Элемент cbData задает число байтов, пересылаемых в другой процесс, а lpData указывает на первый байт данных. Адрес, идентифицируемый элементом lpData, находится, конечно же, в адресном пространстве отправителя. Увидев, что Вы посылаете сообщение WM_COPYDATA, SendMessage создает проекцию файла размером cbData байтов и копирует данные из адресного пространства Вашей программы в эту проекцию. Затем отправляет сообщение окну-приемнику. При обработке этого сообщения принимающей оконной процедурой параметр lParam указывает на структуру COPYDATASTRUCT, которая находится в адресном пространстве процесса-приемника. Элемент lpData этой структуры указывает на проекцию файла в адресном пространстве процесса-приемника.

Вам следует помнить о трех важных вещах, связанных с сообщением WM_COPYDATA.

1) Отправляйте его всегда синхронно; никогда не пытайтесь делать этого асинхронно. Последнее просто невозможно: как только принимающая оконная процедура обработает сообщение, система должна освободить проекцию файла. При передаче WM_COPYDATA как асинхронного сообщения появится неопределенность в том, когда оно будет обработано, и система не сможет освободить память, занятую проекцией файла.

2) На создание копии данных в адресном пространстве другого процесса неизбежно уходит какое-то время. Значит, пока SendMessage не вернет управление, нельзя допускать изменения содержимого общей проекции файла каким-либо другим потоком.

3) Сообщение WM_COPYDATA позволяет 16-разрядным приложениям взаимодействовать с 32-разрядными (и наоборот), как впрочем и 32-разрядным — с 64-разрядными (и наоборот). Это удивительно простой способ общения между новыми и старыми приложениями. К тому же, WM_COPYDATA полностью поддерживается как в Windows 2000, так и в Windows 98. Но, если Вы все еще пишете 16-разрядные Windows-приложения, учтите, что сообщение WM_COPYDATA и структура COPYDATASTRUCT в Microsoft Visual C++ версии 1.52 не определены. Вам придется добавить их определения самостоятельно:

```
// включите этот код в свою 16-разрядную Windows-программу
#define WM_COPYDATA 0x004A
typedef VOID FAR* PVOID;
typedef struct tagCOPYDATASTRUCT {
DWORD dwData;
DWORD cbData;
PVOID lpData;
} COPYDATASTRUCT, FAR* PCOPYDATASTRUCT;
```

Сообщение WM_COPYDATA — мощный инструмент, позволяющий разработчикам экономить массу времени при решении проблем связи между процессами.

6. Ввод данных с манипулятора «мышь». Обработка сообщений мыши. Ввод данных с клавиатуры. Понятие фокуса ввода. Обработка сообщений от клавиатуры.

Ввод с клавиатуры и фокус

Ввод с клавиатуры направляется потоком необработанного ввода (RIT) в очередь виртуального ввода какого-либо потока, но только не в окно. RIT помещает события от клавиатуры в очередь потока безотносительно конкретному окну. Когда поток вызывает GetMessage, событие от клавиатуры извлекается из очереди и перенаправляется окну (созданному потоком), на котором в данный момент сосредоточен фокус ввода (рис. 27-2). Чтобы направить клавиатурный ввод в другое окно, нужно указать, в очередь какого потока RIT должен помещать события от клавиатуры, а также «сообщить» переменным состояния ввода потока, какое окно будет находиться в фокусе. Одним вызовом SetFocus эти задачи не решить. Если в данный момент ввод от RIT получает поток 1, то вызов SetFocus с передачей описателей окон A, B или C приведет к смене фокуса.

ОПЕРАЦИИ С ОКНАМИ

Окно, теряющее фокус, убирает используемый для обозначения фокуса прямоугольник или гасит курсор ввода, а окно, получающее фокус, рисует такой прямоугольник или показывает курсор ввода.

Рис. 27-2. *RIT направляет пользовательский ввод с клавиатуры в очередь виртуального ввода только одного из потоков единовременно*

Предположим, однако, что поток 1 по-прежнему получает ввод от RIT и вызывает SetFocus, передавая ей описатель окна E. В этом случае система не дает функции что-либо сделать, так как окно, на которое Вы хотите перевести фокус, не использует очередь виртуального ввода, подключенную в данный момент к RIT. Когда поток 1 выполнит этот вызов, на экране не произойдет ни смены фокуса, ни каких-либо изменений.

Возьмем другую ситуацию: поток 1 подключен к RIT, а поток 2 вызывает SetFocus, передавая ей описатель окна E. На этот раз значения переменных локального состояния ввода потока 2 изменяются так, что — когда RIT в следующий раз направит события от клавиатуры этому потоку — ввод с клавиатуры получит окно E. Этот вызов не заставит RIT направить клавиатурный ввод в очередь виртуального ввода потока 2. Так как фокус теперь сосредоточен на окне E потока 2, оно получает сообщение WM_SETFOCUS. Если окно E — кнопка, на нем появляется прямоугольник, обозначающий фокус, и в результате на экране могут появиться два окна с такими прямоугольниками (окна A и E). Сами понимаете, это вряд ли кому понравится. Поэтому вызывать SetFocus следует с большой осторожностью — чтобы не создавать подобных ситуаций. Вызов SetFocus безопасен, только если Ваш поток подключен к RIT. Кстати, если Вы переведете фокус на окно, которое, получив сообщение WM_SETFOCUS, показывает курсор ввода, не исключено одновременное появление на экране нескольких окон с таким курсором. Это тоже вряд ли кого обрадует. Когда фокус переводится с одного окна на другое обычным способом (например, щелчком окна), теряющее фокус окно получает сообщение WM_KILLFOCUS. Если окно, получающее фокус, принадлежит другому потоку, переменные локального состояния ввода потока, который владеет окном, теряющим фокус, обновляются так, чтобы показать: окон в фокусе нет. И вызов GetFocus возвращает при этом NULL, заставляя поток считать, что окон в фокусе нет.

*HWND **SetActiveWindow**(HWND hwnd);* активизирует в системе окно верхнего уровня и переводит на него фокус.Как и SetFocus, эта функция ничего не делает, если поток вызывает ее с описателем окна, созданного другим потоком.

Модель аппаратного ввода и локальное состояние ввода

Функцию SetActiveWindow дополняет GetActiveWindow:

*HWND **GetActiveWindow**();*

Она работает так же, как и GetFocus, но возвращает описатель активного окна, указанного в переменных локального состояния ввода вызывающего потока. Так что, если активное окно принадлежит другому потоку, функция возвращает NULL.

Есть и другие функции, влияющие на порядок размещения окон, их статус (активно или неактивно) и фокус:

*BOOL **BringWindowToTop**(HWND hwnd);*

*BOOL **SetWindowPos**(HWND hwnd,HWND hwndInsertAfter,int x,int y,int cx,int cy,UINT fuFlags);*

Обе эти функции работают одинаково (фактически BringWindowToTop вызывает SetWindowPos, передавая ей HWND_TOP во втором параметре). Когда поток, вызывающий любую из этих функций, не связан с RIT, они ничего не делают. В ином случае(когда поток связан с RIT) система активизирует указанное окно. Обратите внимание, что здесь не имеет значения, принадлежит ли это окно вызвавшему потоку. Окно становится активным, а к RIT подключается тот поток, который создал данное окно. Кроме того, значения переменных локального состояния ввода обоих потоков обновляются так, чтобы отразить эти изменения. Иногда потоку нужно вывести свое окно на передний план. Например, Вы запланировали какую-то встречу, используя Microsoft Outlook. Где-то за полчаса до назначенного времени Outlook выводит на экран диалоговое окно с напоминанием о встрече. Если поток Outlook не связан с RIT, это диалоговое окно появится под другими окнами, и Вы его не увидите. Поэтому нужен какой-то способ, который позволил бы привлекать внимание к определенному окну, даже если в данный момент пользователь работает с окном другого приложения.

Вот функция, которая выводит окно на передний план и подключает его поток к RIT:

*BOOL **SetForegroundWindow**(HWND hwnd);*

Одновременно система активизирует окно и переводит на него фокус. Функция, парная SetForegroundWindow:

*HWND **GetForegroundWindow**();*

Она возвращает описатель окна, находящегося сейчас на переднем плане.

В более ранних версиях Windows функция SetForegroundWindow срабатывала всегда. То есть поток, вызвавший ее, всегда мог перевести указанное окно на передний план (даже если оно было создано другим потоком). Однако разработчики стали злоупотреблять этой функцией и нагромождать окна друг на друга. Представьте, я пишу журнальную статью, и вдруг выскакивает окно с сообщением о завершении печати. Если бы я не смотрел на экран, то начал бы вводить текст не в свой документ, а в это окно. Еще больше раздражает, когда пытаешься выбрать команду в меню, а на экране появляется какое-то окно и закрывает меню.

ОПЕРАЦИИ С ОКНАМИ

Чтобы прекратить всю эту неразбериху, Microsoft сделала SetForegroundWindow чуть поумнее. В частности, эта функция срабатывает, только если вызывающий поток уже подключен к RIT или если поток, связанный с RIT в данный момент, не получал ввода на протяжении определенного периода (который задается функцией SystemParametersInfo и значением SPI_SETFOREGROUNDLOCKTIMEOUT). Кроме того, SetForegroundWindow терпит неудачу, когда активно какое-нибудь меню. Если SetForegroundWindow не удается переместить окно на передний план, то его кнопка на панели задач начинает мигать. Заметив это, пользователь будет в курсе, что окно требует его внимания. Чтобы выяснить, в чем дело, пользователю придется активизировать это окно вручную. Управлять режимом мигания окна позволяет функция SystemParametersInfo со значением SPI_SETFOREGROUNDFLASHCOUNT. Из-за такого поведения SetForegroundWindow в систему встроено несколько новых функций. Первая из них, AllowSetForegroundWindow, разрешает потоку указанного процесса успешно вызвать SetForegroundWindow, но только если и вызывающий ее поток может успешно вызвать SetForegroundWindow. Чтобы любой процесс мог выводить окно «поверх» остальных окон, открытых Вашим потоком, передайте в параметре dwProcessId значение ASFW_ANY (определенное как –1):

*BOOL **AllowSetForegroundWindow**(DWORD dwProcessId);*

Кроме того, можно полностью заблокировать работу SetForegroundWindow, вызвав LockSetForegroundWindow:

*BOOL **LockSetForegroundWindow**(UINT uLockCode);*

В параметре uLockCode она принимает либо LSFW_LOCK, либо LSFW_UNLOCK.

Данная функция вызывается системой, когда на экране активно какое-нибудь системное меню, — чтобы никакое окно не могло его закрыть. (Поскольку меню Start не является встроенным, то при его открытии Windows Explorer сам вызывает эти функции.)

Система автоматически снимает блокировку с функции SetForegroundWindow, когда пользователь нажимает клавишу Alt или активизирует какое-либо окно. Так что приложение не может навечно заблокировать SetForegroundWindow.

Другой аспект управления клавиатурой и локальным состоянием ввода связан с массивом синхронного состояния клавиш (synchronous key state array). Этот массив включается в переменные локального состояния ввода каждого потока. В то же время массив асинхронного состояния клавиш (asynchronous key state array) — только один, и он разделяется всеми потоками. Эти массивы отражают состояние всех клавиш на данный момент, и функция GetAsyncKeyState позволяет определить, нажата ли сейчас заданная клавиша:

SHORT GetAsyncKeyState(int nVirtKey);

Параметр nVirtKey задает код виртуальной клавиши, состояние которой нужно проверить. Старший бит результата определяет, нажата в данный момент клавиша (1) или нет (0). Я часто пользовался этой функцией,

определяя при обработке сообщения, отпустил ли пользователь основную (обычно левую) кнопку мыши. Передав значение VK_LBUTTON, я ждал, когда обнулится старший бит. Заметьте, что GetAsyncKeyState всегда возвращает 0 (не нажата), если ее вызывает другой поток, а не тот, который создал окно, находящееся сейчас в фокусе ввода. Функция GetKeyState отличается от GetAsyncKeyState тем, что возвращает состояние клавиатуры на момент, когда из очереди потока извлечено последнее сообщение от клавиатуры: SHORT GetKeyState(int nVirtKey);

Управление курсором мыши

В концепцию локального состояния ввода входит и управление состоянием курсора мыши. Поскольку мышь, как и клавиатура, должна быть доступна всем потокам, Windows не позволяет какому-то одному потоку монопольно распоряжаться курсором мыши, изменяя его форму или ограничивая область его перемещения. Посмотрим, как система управляет этим курсором.

Один из аспектов управления курсором мыши заключается в его отображении или гашении. Если поток вызывает ShowCursor(FALSE), то система скрывает курсор, когда он оказывается на любом окне, созданном этим потоком, и показывает курсор всякий раз, когда он попадает в окно, созданное другим потоком. Другой аспект управления курсором мыши — возможность ограничить его перемещение каким-либо прямоугольным участком. Для этого надо вызвать функцию: *BOOL **ClipCursor**(CONST RECT *prc);*

Она ограничивает перемещение курсора мыши прямоугольником, на который указывает параметр prc. И опять система разрешает потоку ограничить перемещение курсора заданным прямоугольником. Но, когда возникает событие асинхронной активизации, т. е. когда пользователь переключается в окно другого приложения, нажимает клавиши Ctrl+Esc или же поток вызывает SetForegroundWindow, система снимает ограничения на передвижение курсора, позволяя свободно перемещать его по экрану.

И здесь мы подошли к концепции захвата мыши (mouse capture). «Захватывая» мышь (вызовом SetCapture), окно требует, чтобы все связанные с мышью сообщения RIT отправлял в очередь виртуального ввода вызывающего потока, а из нее — установившему захват окну до тех пор, пока программа не вызовет ReleaseCapture. Как и в предыдущих случаях, это тоже снижает отказоустойчивость системы, но без компромиссов, увы, не обойтись. Вызывая SetCapture, поток заставляет RIT помещать все сообщения от мыши в свою очередь виртуального ввода. При этом SetCapture соответственно настраивает переменные локального состояния ввода данного потока. Обычно приложение вызывает SetCapture, когда пользователь нажимает кнопку мыши. Но поток может вызвать эту функцию, даже если нажатия кнопки мыши не было. Если SetCapture вызывается при нажатой кнопке, захват действует для всей системы. Как только система определяет, что все кнопки мыши отпущены, RIT перестает направлять сообщения от мыши исключительно в очередь виртуального ввода данного потока. Вместо этого он передает сообщения в очередь ввода, связанную с окном, «поверх» которого курсор находится в данный момент. И это нормальное поведение системы, когда захват мыши не установлен. Однако для вызвавшего SetCapture потока ничего не меняется. Всякий раз, когда курсор оказывается на любом из окон, созданных установившим захват потоком, сообщения от мыши направляются в окно, применительно к которому этот захват и установлен. Иначе говоря, когда пользователь отпускает все кнопки мыши, захват осуществляется на уровне лишь данного потока, а не всей системы. Если пользователь попытается активизировать окно, созданное другим потоком, система автоматически отправит установившему захват потоку сообщения о нажатии и отжатии кнопок мыши. Затем она изменит переменные локального состояния ввода потока, чтобы отразить тот факт, что поток более не работает в режиме захвата. Словом, Microsoft считает, что захват мыши чаще всего применяется для выполнения таких операций, как щелчок и перетаскивание экранного объекта.

ОПЕРАЦИИ С ОКНАМИ

Последняя переменная локального состояния ввода, связанная с мышью, относится к форме курсора. Всякий раз, когда поток вызывает SetCursor для изменения формы курсора, переменные локального состояния ввода соответствующим образом обновляются. То есть переменные локального состояния ввода всегда запоминают последнюю форму курсора, установленную потоком.

Допустим, пользователь перемещает курсор мыши на окно Вашей программы, окно получает сообщение WM_SETCURSOR, и Вы вызываете SetCursor, чтобы преобразовать курсор в «песочные часы». Вызвав SetCursor, программа начинает выполнять какую-то длительную операцию. (Бесконечный цикл — лучший пример длительной операции. Шутка.) Далее пользователь перемещает курсор из окна Вашей программы в окно другого приложения, и это окно может изменить форму курсора. Для такого изменения переменные локального состояния ввода не нужны. Но переведем курсор обратно в то окно, поток которого по-прежнему занят обработкой. Системе «хочется» послать окну сообщения WM_SETCURSOR, но процедура этого окна не может выбрать их из очереди, так как его поток продолжает свою операцию. Тогда система определяет, какая форма была у курсора в прошлый раз (информация об этом содержится в переменных локального состояния

ввода данного потока), и автоматически восстанавливает ее (в нашем примере — «песочные часы»). Теперь пользователю четко видно, что в этом окне работа еще не закончена и придется подождать.

7. **Вывод информации в окно. Механизм перерисовки окна. Понятие области обновления окна. Операции с областью обновления окна.**

Вывод информации в окно.

Разделение дисплея между прикладными программами осуществляется с помощью окон. Видимая площадь окна может изменяться, что требует постоянного контроля за отображаемой в окне информацией и своевременного восстановления утраченных частей изображения.

ОС не хранит графическую копию рабочей (пользовательской) части каждого окна. Она возлагает ответственность за правильное отображения окна на прикладную программу, посылая ей *WM_PAINT* каждый раз, когда все окно или его часть требует перерисовки.

При операциях с окнами система помечает разрушенные части окна, как подлежащие обновлению и помещает информацию о них, в область обновления - ***UPDATEREGION***

На основании содержимого этой области и происходит восстановление. ОС посылает окну сообщение WM_PAINT всякий раз, когда область обновления окна оказывается не пустой и при условии, что в очереди сообщений приложения нет ни одного сообщения. При получении сообщения WM_PAINT, окно должно перерисовать лишь свою внутреннюю часть, называемую *рабочей областью (ClientArea)*. Все остальные области окна перерисовывает ОС поWM_NCPAINT.

Для ускорения графического вывода Windows осуществляет отсечение. На экране перерисовываются лишь те области окна, которые действительно требуют обновления. Вывод за границами области отсечения игнорируется. Это дает право прикладной программе перерисовывать всю рабочую область в ответ на сообщение WM_PAINT. Лишний вывод ОС отсекает.

Инициатором сообщения WM_PAINT может выступать не только ОС, но и прикладная программа. Чтобы спровоцировать перерисовку окна необходимо вызвать функцию:

*void **InvalidateRect**(HWND, //handleокна*
RECT //эта область требует перерисовки,*
BOOL//нужно ли перед перерисовкой очищать область обновления
);

Очистка производится сообщением *WM_ERASE_BACKGROUND*.

После вызова функции InvalidateRect, окно не перерисовывается сразу (до WM_PAINT). Перерисовка произойдет только при опросе программой очереди сообщений. Когда перерисовка требуется немедленно, то вслед за InvalidateRect вызывается функция: *void **UpdateWindow**(HWND);*

Механизм перерисовки окна.

Перерисовка содержимого окна основана на получении *контекста устройства*, связанного с окном, рисование (вывод графических примитивов) в этом контексте устройства, и освобождение контекста устройства.

В разных случаях получение контекста устройства осуществляется разными функциями ОС. В ответ на сообщение WM_PAINT контекст устройства получается с помощью функции:

*HDC **BeginPaint**(HWND,PAINTSTRUCT*);*
//рисование
*void **EndPaint**(HWND,PAINTSTRUCT*);*

Между вызовами этих 2х функций заключаются вызовы графических примитивов (*Rectangle(),Line()).* Функции BeginPain и EndPaint можно вызывать только на сообщение WM_PAINT.

Иногда бывает необходимо выполнить *перерисовку* окна в какой-то другой момент времени (по сообщению от таймера). В этом случае контекст дисплея получается с помощью функции:

*HDC **GetDC**(HWND);*
А освобождается функцией:
*int **ReleaseDC**(HWND, HDC);*

Вызовы этих функций обязательно должны быть сбалансированы, иначе возникнут сбои в работе ОС. В оконном классе существует стиль, который назначает окну собственный контекст устройства (*CS_OWNDC*). По умолчанию этот флаг сброшен.

В результате, при получении контекста дисплея для окна, возвращается контекст рабочей области Desktop, на которой расположено окно, но с настроенными параметрами для окна.

Если флаг установлен – для окна создается свой собственный контекст дисплея. Функции получения контекста всего лишь возвращают его.
С точки зрения экономии памяти лучше отказаться от использования собственного контекста. Но сегодня память компьютеров велика, и поэтому рекомендуется использовать собственный контекст.

Отправка сообщения в окно.

В Windows осуществляется синхронно: отправитель не может продолжить работу, пока окно не обработает полученное сообщение. Такая операционная система не вправе претендовать на устойчивость к сбоям. Это противоречие было серьезным вызовом для команды разработчиков из Microsoft. В итоге было выбрано компромиссное решение, отвечающее двум вышеупомянутым целям.

Для начала рассмотрим некоторые базовые принципы. Один процесс в Windows может создать до 10 000 User-объектов различных типов — значков, курсоров, оконных классов, меню, таблиц клавиш-акселераторов и т. д. Когда поток из какого-либо процесса вызывает функцию, создающую один из этих объектов, последний переходит во владение процесса. *Поэтому, если процесс завершается, не уничтожив данный объект явным образом, операционная система делает это за него.* Однако два User-объекта (*окна и ловушки*) принадлежат только создавшему их потоку. И вновь, если поток создает окно или устанавливает ловушку, а потом завершается, операционная система автоматически уничтожает окно или удаляет ловушку. Этот принцип принадлежности окон и ловушек создавшему их потоку оказывает существенное влияние на механизм функционирования окон: поток, создавший окно, должен обрабатывать все его сообщения.Допустим, поток создал окно, а затем прекратил работу. Тогда его окно уже не получит сообщение WM_DESTROY или WM_NCDESTROY, потому что поток уже завершился и обрабатывать сообщения, посылаемые этому окну, больше некому.

Это также означает, что каждому потоку, создавшему хотя бы одно окно, система выделяет *очередь сообщений*, используемую для их диспетчеризации. Чтобы окно в конечном счете получило эти сообщения, поток должен иметь собственный цикл выборки сообщений.

Очередь сообщений потока Как я уже говорил, одна из главных целей Windows — предоставить всем приложениям отказоустойчивую среду. Для этого любой поток должен выполняться в такой среде, где он может считать себя единственным. Точнее, у каждого потока должны быть очереди сообщений, полностью независимые от других потоков. Кроме того, для каждого потока нужно смоделировать среду, позволяющую ему самостоятельно управлять фокусом ввода с клавиатуры, активизировать окна, захватывать мышь и т. д. Создавая какой-либо поток, система предполагает, что он не будет иметь отношения к поддержке пользовательского интерфейса. Это позволяет уменьшить объем выделяемых ему системных ресурсов. Но, как только поток обратится к той или иной GUI-функции (например, для проверки очереди сообщений или создания окна), система автоматически выделит ему дополнительные ресурсы, необходимые для выполнения задач, связанных с пользовательским интерфейсом. А если конкретнее, то система создает структуру THREADINFO и сопоставляет ее с этим потоком. Элементы этой структуры используются, чтобы обмануть поток — заставить его считать, будто он выполняется в среде, принадлежащей только ему. THREADINFO — это внутренняя (недокументированная) структура, идентифицирующая очередь асинхронных сообщений потока (posted-message queue), очередь синхронных сообщений потока (sent-message queue), очередь ответных сообщений (reply-message queue), очередь виртуального ввода (virtualized input queue) и флаги пробуждения (wake flags); она также включает ряд других переменных-членов, характеризующих локальное состояние ввода для данного потока. Структура THREADINFO — фундамент всей подсистемы передачи сообщений.

8. Принципы построения графической подсистемы ОС Windows. Понятие контекста устройства. Вывод графической информации на физическое устройство. Управление цветом. Палитры цветов. Графические инструменты. Рисование геометрических фигур

Взаимодействие приложения с GDI осуществляется при непременном участии еще одного посредника — так называемого контекста устройства.

Контекст устройства (device context)— это внутренняя структура данных, которая определяет набор графических объектов и ассоциированных с ними атрибутов, а также графических режимов, влияющих на вывод.

В следующем списке приведены *основные графические объекты*: Перо(pen) для рисования линий. Кисть(brush) для заполнения фона или заливки фигур. Растровое изображение(bitmap) для отображения в указанной области окна. Палитра (palette) для определения набора доступных цветов.Шрифт (font) для вывода текста. Регион(region) для отсечения области вывода.

Если необходимо рисовать на устройстве графического вывода (экране дисплея или принтере), то сначала нужно получить дескриптор контекста устройства. Возвращая этот дескриптор после вызова соответствующих функций, Windows тем самым предоставляет разработчику право на использование данного устройства. После этого дескриптор контекста устройства передается как параметр в функции GDI, чтобы идентифицировать устройство, на котором должно выполняться рисование.

Контекст устройства содержит много атрибутов, определяющих поведение функций GDI. Благодаря этому списки параметров функций GDI содержат только самую необходимую информацию, например начальные координаты или размеры графического объекта. Все остальное система извлекает из контекста устройства. Только если значения по умолчанию не устраивают разработчика, необходимо вызывать функции GDI, изменяющие значения соответствующих атрибутов.

Win32 API поддерживает следующие типы контекстов устройства: контекст дисплея; контекст принтера; контекст в памяти (совместимый контекст); метафайловый контекст; информационный контекст.

Управление цветом:

- typedef DWORD COLORREF;
- COLORREF color = RGB(255, 0, 0); // 0x000000FF
- BYTE GetRValue(DWORD rgb), GetGValue, GetBValue.
- COLOREF GetNearestColor(HDC, COLORREF);
- COLORREF SetBkColor(HDC, COLORREF), GetBkColor.
- LOGPALETTE, CreatePalette, SetPaletteEntries, GetPaletteEntries, SelectPalette, RealizePalette, DeleteObject.

Инструменты для рисования:

- DC – 1 Bitmap , 1 Region, 1 Pen, 1 Brush, 1 Palette, 1 Font.
- HPEN – LOGPEN, CreatePenIndirect, CreatePen.
- HBRUSH – LOGBRUSH, CreateBrushIndirect, CreateBrush.
- HFONT – LOGFONT, CreateFontIndirect, CreateFont.
- HANDLE GetStockObject(int);
- HANDLE SelectObject(HDC, HANDLE);
- bool DeleteObject(HANDLE);

Вывод на физическое устройство:

Вывод изображений на такое устройство, как принтер, может выполняться с использованием тех же приемов, что и вывод в окно приложения. Прежде всего необходимо получить контекст устройства. Затем можно вызывать функции GDI, выполняющие рисование, передавая им идентификатор полученного контекста в качестве параметра.

В отличие от контекста отображения, контекст физического устройства не получается, а создается, для чего используется функция CreateDC :

```
HDC WINAPI CreateDC(
LPCSTR lpszDriver, // имя драйвера
LPCSTR lpszDevice, // имя устройства
LPCSTR lpszOutput, // имя файла или порта вывода
const void FAR* lpvInitData); // данные для инициализации
```

Рисование геометрических фигур

```
BOOL LineTo( HDC hdc, int nXEnd, int nYEnd );
BOOL MoveToEx(HDC hdc, int X, int Y, LPPOINT lpPoint/*old current position*/);
BOOL Rectangle(HDC hdc, int nLeftRect, int nTopRect, int nRightRect, int nBottomRect );
```

BOOL Ellipse(HDC hdc, int nLeftRect, int nTopRect, int nRightRect, int nBottomRect);
BOOL Polygon(
HDC hdc, // handle to DC
CONST POINT *lpPoints, // polygon vertices
int nCount // count of polygon vertices
);
BOOL PolyBezier(
HDC hdc, // handle to device context
CONST POINT* lppt, // endpoints and control points
DWORD cPoints // count of endpoints and control points
);

Графические инструменты
Pen – CreatePen,
Brush – CreateSolidBrush,
Font – CreateFont,
Холст – CreateCompatibleDC (?)

Управление цветом.
RGB –формат. Для кодирования цвета используются переменные с типом данных COLORREF, который определен через тип данных UINT.
COLORREF col;
Col = RGB(255,0,0); // в памяти по байтам: 0,B,G,R.
BYTE RedValue;
RedValue=GetRValue(color) //значения составляющих GetGValue, GetBValue
Позволяет иметь более 16 миллионов оттенков.
Далеко не все графические устройства поддерживают такое количество. Если программа устанавливает цвет, который данное устройство воспроизвести не может ОС заменяет этот цвет на ближайший из числа доступных.
COLOREF GetNearestColor(HDC, COLORREF).
HDC поддерживает понятие фона и фонового цвета.
Некоторые функции могут осуществлять предварительную заливку области фоновым цветом (функции вывода текста).
Установка фонового цвета SetBkColor(…), получение GetBkColor().

Палитры цветов.
Для того чтобы создать палитру, приложение должно заполнить структуру LOGPALETTE , описывающую палитру, и массив структур PALETTEENTRY , определяющий содержимое палитры.
Структура LOGPALETTE и указатели на нее определены в файле windows.h:
typedef struct tagLOGPALETTE
{
WORD palVersion;
WORD palNumEntries;
PALETTEENTRY palPalEntry[1];
} LOGPALETTE;
В поле palNumEntries нужно записать размер палитры (количество элементов в массиве структур PALETTEENTRY).
Сразу после структуры LOGPALETTE в памяти должен следовать массив структур PALETTEENTRY, описывающих содержимое палитры:
typedef struct tagPALETTEENTRY
{
BYTE peRed;
BYTE peGreen;

BYTE peBlue;
BYTE peFlags;
} PALETTEENTRY;
typedef PALETTEENTRY FAR* LPPALETTEENTRY;
Поле peFlags определяет тип элемента палитры и может иметь значения NULL, PC_EXPLICIT , PC_NOCOLLAPSE и PC_RESERVED .

Если поле peFlags содержит значение NULL, в полях peRed, peGreen и peBlue находятся RGB-компоненты цвета. В процессе реализации логической палитры для этого элемента используется описанный нами ранее алгоритм.

Если поле peFlags содержит значение PC_EXPLICIT, младшее слово элемента палитры содержит индекс цвета в системной палитре.

Если поле peFlags содержит значение PC_NOCOLLAPSE, в процессе реализации логической палитры данный элемент будет отображаться только на свободную ячейку системной палитры. Если же свободных ячеек нет, используется обычный алгоритм реализации.

Последнее возможное значение для поля peFlags (PC_RESERVED) используется для анимации палитры с помощью функции AnimatePalette . Анимация палитры позволяет динамически вносить изменения в палитру. Такой элемент палитры после реализации не подвергается изменениям при реализации других палитр, он становится зарезервированным.

После подготовки структуры LOGPALETTE и массива структур PALETTEENTRY приложение может создать логическую палитру, вызвав функцию CreatePalette :
HPALETTE WINAPI CreatePalette(const LOGPALETTE FAR* lplgpl);

В качестве параметра следует передать функции указатель на заполненную структуру LOGPALETTE.

Функция возвращает идентификатор созданной палитры или NULL при ошибке.
Выбор палитры в контекст отображения

Созданная палитра перед использованием должна быть выбрана в контекст отображения. Выбор палитры выполняется функцией SelectPalette :
HPALETTE WINAPI SelectPalette(
HDC hdc, HPALETTE hpal, BOOL fPalBack);

Функция выбирает палитру hpal в контекст отображения hdc, возвращая в случае успеха идентификатор палитры, которая была выбрана в контекст отображения раньше. При ошибке возвращается значение NULL.

Указав для параметра fPalBack значение TRUE, вы можете заставить GDI в процессе реализации палитры использовать алгоритм, соответствующий фоновому окну. Если же этот параметр равен FALSE, используется алгоритм для активного окна (т. е. все ячейки системной палитры, кроме зарезервированных для статических цветов, отмечаются как свободные и используются для реализации палитры).
Реализация палитры

Процедура реализации палитры заключается в вызове функции RealizePalette :
UINT WINAPI RealizePalette(HDC hdc);

Возвращаемое значение равно количеству цветов логической палитры, которое удалось отобразить в системную палитру.
Рисование с использованием палитры

Итак, вы создали палитру, выбрали ее в контекст отображения и реализовали. Теперь приложение может пользоваться цветами из созданной палитры. Но как?

Если приложению нужно создать перо или кисть, определить цвет текста функцией SetTextColor или закрасить область функцией FloofFill (т. е. вызвать одну из функций, которой в качестве параметра передается переменная типа COLORREF, содержащая цвет), вы можете вместо макрокоманды RGB воспользоваться одной из следующих макрокоманд:

```
#define PALETTEINDEX (i) \
((COLORREF)(0x01000000L | (DWORD)(WORD)(i)))
#define PALETTERGB (r,g,b) (0x02000000L | RGB(r,g,b))
```

Макрокоманда PALETTEINDEX позволяет указать вместо отдельных компонент цвета индекс в логической палитре, соответствующий нужному цвету.

Макрокоманда PALETTERGB имеет параметры, аналогичные знакомой вам макрокоманде RGB, однако работает по-другому.

Если цвет определен с помощью макрокоманды RGB, в режиме низкого и среднего цветового разрешения для рисования будет использован ближайший к указанному статический цвет. В режиме высокого цветового разрешения полученный цвет будет полностью соответствовать запрошенному.

Если же для определения цвета использована макрокоманда PALETTERGB, GDI просмотрит системную палитру и подберет из нее цвет, наилучшим образом соответствующий указанному в параметрах макрокоманды.

В любом случае при работе с палитрой GDI не использует для удовлетворения запроса из логической палитры смешанные цвета.

Какой из двух макрокоманд лучше пользоваться?

На этот вопрос нет однозначного ответа.

Макрокоманда PALETTEINDEX работает быстрее, однако с ее помощью можно использовать только те цвета, которые есть в системной палитре. Если ваше приложение будет работать в режиме True Color, лучшего эффекта можно добиться при использовании макрокоманды PALETTERGB, так как для режимов высокого цветового разрешения эта макрокоманда обеспечит более точное цветовое соответствие.

Удаление палитры

Так как палитра является объектом, принадлежащим GDI, а не создавшему ее приложению, после использования палитры приложение должно обязательно ее удалить. Для удаления логической палитры лучше всего воспользоваться макрокомандой DeletePalette , определенной в файле windowsx.h:

```
#define DeletePalette(hpal) \
DeleteObject((HGDIOBJ)(HPALETTE)(hpal))
```

В качестве параметра этой макрокоманде следует передать идентификатор удаляемой палитры.

Учтите, что как и любой другой объект GDI, нельзя удалять палитру, выбранную в контекст отображения. Перед удалением следует выбрать старую палитру, вызвав функцию SelectPalette.

9. **Растровые изображения. Виды растровых изображений. Значки и курсоры. Способ вывода растровых изображений с эффектом прозрачного фона. Аппаратно-зависимые и аппаратно-независимые растровые изображения. Операции с растровыми изображениями. Вывод растровых изображений.**

Типы растровых изображений:

- Bitmap – базовый формат растрового изображения.
- Icon – значок: AND-маска и XOR-маска.
- Cursor – курсор: две маски и точка касания – Hot Spot.

Значки – это небольшая картинка, ассоциируемая с некоторой программой, файлом на экране. Значок является частным случаем растровых изображений.

На экране значки могут иметь не прямоугольную форму, что достигается за счет описания значка двумя точечными рисунками:

- AND-mask. Монохромная.
- XOR-mask. Цветная.

При выводе значков, ОС комбинирует маски по следующему правилу:

Экран = (Экран ANDМонохромная маска)XORЦветная маска.

Накладывая AND-mask, ОС вырезает на экране пустую область с заданным контуром. BAND-mask фигура кодируется с помощью 0, а прозрачный фон с помощью 1. После вывода AND-mask ОС накладывает XOR-mask, содержащую изображения фигур. Изображение фигуры является цветным. На диске значки сохраняются в *.ico формате. В ОС существует несколько форматов значков, которые отличаются по размеру и цвету (16x16, 32x32, 16x32, 64x64).

Курсоры. Указатели мыши. Небольшой образ. По своему представлению в файле и памяти курсор напоминает значки, но существуют некоторые значки. Курсоры могут быть размером 16x16 и 32x32. Важным существенным отличием является наличие в нем горячей точки (hotspot), которая ассоциируется с позицией указателя мыши на экране. *.CUR.

Точечные рисунки – это изображение, представление графической информации, ориентированное на матричное устройство вывода. Точечный рисунок состоит из пикселей, организованных в матрицу. ОС позволяет использовать точечные рисунки двух видов:

1)**Аппаратно-зависимые.** Device Dependent Bitmpap. Рассчитаны только на определенный тип графического адаптера или принтера. Их точки находятся в прямом соответствии с пикселями экрана или другой поверхности отображения.

Если это экран – то информация о пикселях представляется битовыми планами в соответствии с особенностями устройства. Он наименее удобен при операциях с точечным рисунком, но обеспечивает наибольшую скорость графического вывода. Он хорошо подходит для работы с точечными рисунками в оперативной памяти.

При хранении на диске используется аппаратно-независимый формат - BMP,DIB.

2)**Аппаратно-независимые**. Device Independent Bitmpap. Формат хранения аппаратно-независимых точечных рисунков не зависит от используемой аппаратуры. Здесь информация о цвете и самом изображении хранится раздельно.

Цвета собраны в таблицу, а точки изображения кодируют номера цветов таблицы. Под каждую точку изображения может отводиться 1,4,8,16,24 битов изображения. Они могут храниться на диске в сжатом виде.

Для сжатия применяется алгоритм *RunLengthEncoding(RLE)*. Разжатие производится автоматически. Недостаток: обеспечивается более низкая скорость работы.

Вывод растрового изображения с эффектом прозрачного фона: AND-маска – монохромная. Фигура кодируется нулем, прозрачный фон – единицей. Вырезает на экране «черную дыру» там, где должна быть фигура.

Растровая операция – способ комбинирования пикселей исходного изображения с пикселями поверхности отображения целевого контекста устройства. При масштабировании в сторону сжатия некоторые цвета могут пропадать. При растяжении, таких проблем не существует. При сжатии возможно 3 способа сжатия.

РИСОВАНИЕ ТОЧЕЧНЫХ РИСУНКОВ ПРОГРАММНЫМ СПОСОБОМ

После создания контекста виртуального устройства *CreateCompatibleDC()* поверхность отображения в этом контексте устройства имеет нулевые размеры. Если в полученном контексте устройства необходимо нарисовать изображение, то сначала необходимо создать в контексте устройства поверхность отображения (точечный рисунок). И установить его в контексте устройства. Для создания рисунка:

*HBITMAP **CreateCompatibleBitmap**(HDC hdc, //handletoDC*
int nWidth, // width of bitmap, in pixels
int nHeight// height of bitmap, in pixels
);

После вызова этой функции нужно не забыть вызвать *SelectObject*. Созданный точечный рисунок –DDB. Часто при работе с изображениями (растровыми) применяется:

*BOOL **PatBlt**(*
HDC hdc, // handle to DC
int nXLeft, // x-coord of upper-left rectangle corner
int nYLeft, // y-coord of upper-left rectangle corner
int nWidth, // width of rectangle
int nHeight, // height of rectangle
DWORD dwRop// raster operation code
);

Она служит для заливки битового образа некоторым цветом. Цвет определяется цветом кисти.

XOR-маска – цветная. Фигура кодируется цветом, прозрачный фон – нулем. Врезает на экране фигуру на месте «черной дыры».

LoadIcon(),LoadCursor(),LoadBitmap() – не загружают изображение из файла. Изображения из файла загружает функция *LoadImage()*.В результате загрузки изображения, ОС возвращает дескриптор созданного в памяти объекта. После использования объект должен быть удален функцией:

Функция *HBITMAP **LoadBitmap**(HINSTANCE hInstance, LPCTSTR lpBitmapName)* загружает поименованный ресурс карты бит.

Функция *HANDLE **LoadImage**(HINSTANCE hinst, LPCTSTR lpszName, UINT uType, int cxDesired, int cyDesired, UINT fuLoad)* загружает значок, курсор, "живой " курсор или точечный рисунок.

Функция *HANDLE **CopyImage**(HANDLE hImage, UINT uType, int cxDesired, int cyDesired, UINT fuFlags)* создает новое изображение (значок, курсор, или точечный рисунок) и копирует атрибуты указанного изображения в новое изображение. Если необходимо, функция растягивает биты, чтобы размер нового изображения подогнать под требуемый.

Функция *BOOL **DrawIcon**(HDC hDC, int X, int Y, HICON hIcon)* делает иконой в области клиента окна определенного контекста устройства.

Вывести Bitmap вызовом одной функции нельзя. Необходимо создать дополнительное устройство в памяти – memory DC, выбрать в нем Bitmap в качестве поверхности рисования, выполнить перенос изображения из memory DC в window DC.

Вывод растрового изображения:

```
void ShowBitmap(HWND hWnd, HBITMAP hBmp)
{
	HDC winDC = GetDC(hWnd);
	HDC memDC = CreateCompatibleDC(winDC);
	HBITMAP oldBmp = SelectObject(memDC, hBmp);
	BitBlt(winDC, 10, 10, 64, 64, memDC, 0, 0, SRCCOPY);
	SelectObject(memDC, oldBmp);
	DeleteDC(memDC);
	ReleaseDC(hWnd, winDC);
}
```

10. Библиотека работы с двумерной графикой Direct2D. Инициализация библиотеки. Фабрика графических объектов библиотеки Direct2D. Вывод графики средствами библиотеки Direct2D.

Для удовлетворения новых потребностей рынка IT-техноолгий корпорация Microsoft летом 2009 года разработала на базе технологии DirectX 10 набор библиотек для работы и вывода двумерной графики — Direct2D.
Direct2D включает в себя 4 заголовочных файла и одну библиотеку:

. d2d1.h — содержит объявления основных функций Direct2D API на языке C и C++;
. d2d1helper.h — содержит вспомогательный структуры, классы, функции;
. d2dbasetypes.h — определяет основные примитивы Direct2D, включен в d2d1.h;
. d2derr.h — определяет коды ошибок Direct2D. Включен в d2d1.h;
. d2d1.lib — двоичная библиотека, содержащая все объявленные в заголовочных файлах функции.

Как и DirectX, Direct2D построен на модели COM. Основным объектом, который предоставляет интерфейсы для создания других объектов, является Фабрика Direct2D, или объект класса ID2D1Factory. Он имеет в своем составе методы типа CreateResourse, которые позволяют создавать объекты более специфических типов.
Все объекты (ресурсы) в Direct2D делятся на два больших типа — устройство-зависимые (device-dependent) и устройство-независимые (device-independent). Устройство-зависимые объекты ассоциируются с конкретным устройством вывода и должны быть реинициализированы каждый раз, когда устройство, с которым они ассоциируются, требует реинициализации. Устройство-независимые ресурсы существуют без привязки к какому-либо устройству и уничтожаются в конце жизненного цикла программы или по желанию программиста. Классификация и основные примеры ресурсов приведены на рисунке 1.

Рисунок 1 — Классификация ресурсов Direct2D.

Объекты класса ID2DRenderTarget — это устройство-зависимые объекты, которые ассоциируются с конкретным устройством вывода. Это может быть конкретное окно приложения, битовый образ (изображение) или другое устройство. ID2DRenderTarget имеет в своем составе методы BeginDraw() и EndDraw(), между которыми выполняются все операции вывода графической информации на устройство вывода.

В качестве инструмента вывода используются объекты класса ID2DBrush, которые задают цвет и другие парамеиры выводимых объектов (в т.ч. градиентные заливки). И ID2DBrush, и ID2DRenderTarget — устройство-зависимые ресурсы и будучи созданными однажды для конкретного устройства, могут применяться лишь к нему, и должны быть уничтожены всякий раз, когда уничтожается их устройство.
Объект класса ID2DGeometry — устройство-независимый ресурс. Будучи созданным однажды, он может быть использован любым объектов ID2DRenderTarget. ID2DGeometry задает двумерную форму, интерполированную треугольниками.

После того, как работа с ресурсами и объектами завершена, они должны быть уничтожены функцией Release(), которая унаследована ими от базового COM-объекта. При этом по возможности стоит избегать частого создания и освобождения ресурсов, так как этот процесс требует достаточно много ресурсов процессора. Общая схема работы с Direct2D представлена на рисунке 2.

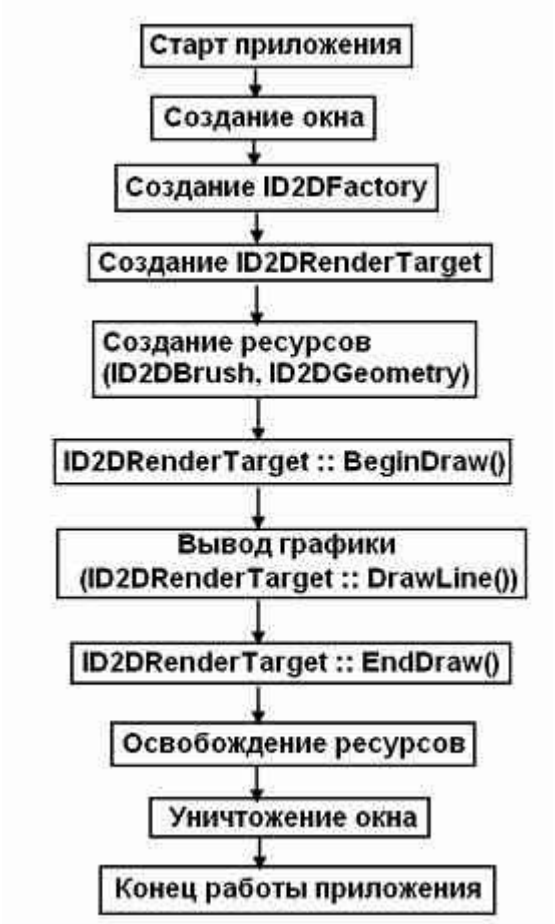


Рисунок 2 — Общая схема использования компонент Direct2D

11. **Вывод текста в ОС Windows. Понятие шрифта. Характеристики шрифта. Понятия физического и логического шрифта. Операции с физическими шрифтами. Операции с логическими шрифтами. Параметры ширины и высоты логического шрифта.**

1) BOOL TextOut
Функция TextOut записывает строку символов в заданном месте, используя текущий выбранный шрифт, цвет фона и цвет текста.
Если функция завершается с ошибкой, величина возвращаемого значения – ноль, иначе – не ноль.
SetTextAlign - устанавливает флажки выравнивания текста для заданного контекста устройства.
GetTextAlign - извлекает настройки выравнивания текста для заданного контекста устройства.
SetTextColor - функция устанавливает цвет текста для заданного контекста устройства.
GetTextColor - функция возвращает цвет текста для заданного контекста устройства.
2) BOOL ExtTextOut
Выводит текст используя текущий выбранный шрифт, цвета фона и текста.
Если строка рисуется, возвращаемое значение является отличным от нуля.
3) BOOL PolyTextOut
рисует несколько строк, используя шрифт и цвета текста, в настоящее время выбранные в заданном контексте устройства. При ошибке – 0, иначе – не ноль.
4)LONG TabbedTextOut

Пишет строку символов в заданном месте, разворачивая позиции табуляции в значения, указанные в массиве позиций табуляции. Текст пишется в текущем выбранном шрифте, цвете фона и цвете текста.
Если функция завершается ошибкой, возвращаемое значение – нуль.
5) int DrawText
Рисует отформатированный текст в заданном прямоугольнике. Если функция завершается успешно, возвращаемое значение - высота текста в логических единицах измерения.
6) int DrawTextEx
Рисует форматированный текст в заданном прямоугольнике.
Если функция завершается с ошибкой, величина возвращаемого значения - ноль.
Шрифт – множество символов со сходными размерами и начертанием контуров. Семейство шрифта – набор шрифтов со сходной шириной символов и гарнитурой:
Шрифты в программе:
Физические – устанавливаемые в операционную систему, файлы.
Логические – запрашиваемые программой у операционной системы, LOGFONT.
Физический шрифт
Установка шрифта:
Скопировать файл шрифта в C:\Windows\Fonts.
Вызвать int AddFontResource(LPCTSTR lpszFilename).
Вызвать SendMessage с кодом WM_FONTCHANGE.
Удаление шрифта:
Вызвать bool RemoveFontResource(LPCTSTR lpszFilename).
Удалить файл шрифта из C:\Windows\Fonts.
Вызвать SendMessage с кодом WM_FONTCHANGE.
Логический шрифт
Создание логического шрифта (LOGFONT):
CreateFontIndirect / CreateFont,
SelectObject,
DeleteObject.
В ширине шрифта различают 3 вида размеров:

- Размер A – отступ слева перед написанием символа
- Размер B – ширина символа
- Размер C – отступ справа от символа

Отступы A и C могут быть отрицательными (когда символ пишется курсивом). Получить значения A,B,C можно с помощью функций:
GetCharABCWidth– только дляTrueTypeFont
GetCharABCWidthFloat
GetCharWidth32
GetCharWidthFloat
Функции для высоты и ширины:
GetTextExtentPoint32
TabbedTextExtent– если есть табуляция. Функция для расчета переноса слов.
GetTextExtentPoint
Ascent – параметр шрифта, называющийся подъемом.
Descent – спуск.
LineGap – пропуск между строками.
Для того, что бы получить все параметры шрифта, необходимо обратиться к одной из двух функций, которые возвращают параметры физического шрифта. Существует 2 уровня представления шрифта:
Высокоуровневый логический – соответствует структура LOGFONT
Низкоуровневый физический – структуры TEXTMETRICS, OUTLINETEXTMETRICS.
Префиксы:
Tm – TEXTMETRICS
Otm – OUTLINETEXTMETRICS
Физические параметры шрифта можно получить с помощью:

GetTextMetrics
GetOutLineTextMetrics
Параметры подъем и спуск шрифта имеют различный смысл:
Существуют пропорциональные и непропорциональные шрифты. Шрифт, который мы создаем – логический.
Физический шрифт – это шрифт, расположенный в каталоге Fonts.
ОС на базе физического создает необходимый нам логический.
Существуют 2 типа шрифтов:

- Растровые (масштабируемые шрифты TrueType)
- Векторные (в чистом виде в Windows таких нет)

Масштабируемые шрифты TrueType описываются сплайнами 3 порядка.PostScript– описываются сплайнами 2ого порядка.
Сплайны третьего порядка позволяют более тонко управлять шрифтом.

12. Системы координат. Трансформации. Матрица трансформаций. Виды трансформаций и их представление в матрице трансформаций. Преобразования в страничной системе координат. Режимы масштабирования.

Системы координат

Существует несколько видов систем координат:

Мировая – world coordinate space (2^32). Обеспечивает параллельный перенос, масштабирование, отражение, поворот, наклон.

Устройства – device coordinate space (2^27). Обеспечивает параллельный перенос (к началу координат на устройстве).

Физическая – physical device coordinate space. Например, клиентская область окна на экране. Для дисплея физическая система координат характеризуется двумя осями Х и У. Х – горизонтально направлена вправо. У – вертикально вниз. Координаты – целые числа.

Логическая (страничная) – page coordinate space (2^32). Устаревшая система координат, основанная на режимах масштабирования (mapping modes). Обеспечивает параллельный перенос, масштабирование, отражение.

Под типом логической системы координат понимается то, как направлены координатные оси и каковы единицы измерения по каждой из координатных осей. Вывод графических примитивов всегда осуществляется в некоторой логической системе координат, которая может не соответствовать физической.

При выводе, Windows осуществляет перерасчет. В логической системе координат направления осей Х и У можно задать, и единицами измерения могут быть не только пиксели устройства, но и десятые, сотые доли миллиметра и дюйма.

При пересчете Windows осуществляет пересчет логической точки (LP) из логического пространства координат, в физическую точку из физической системы координат (DP). Это делается за 3 шага:

1. Параллельный перенос изображения на логической плоскости путем вычитания из координат каждой точки изображения заданных константных значений.

2. Масштабирование полученного изображения путем масштабирования заданной точки (умножением на заданный коэффициент). Изображение переносится на физическую плоскость.

3. Параллельный перенос изображения на физической плоскости за счет добавления заданных константных значений.

$D_X=(L_X-XWO)*XVE/XWE+XVO$

$D_Y=(L_Y-YWO)*YVE/YWE+YVO$

Где:

L_X – координата Х в логической системе

XWO – смещение по оси Х в логической системе

XVO – смещение по оси X в физической системе координат

XVE/XWE – масштабный интерфейс по оси X

В ОС существуют функции, которые выполняют заданные преобразования для массива точек: **LPtoDP()** и **DPtoLP()**.

Матрицы трансформаций

Сама матрица трансформации имеет размер 3x3 и в общем виде записывается так:

$$\begin{bmatrix} a & b & 0 \\ c & d & 0 \\ t_x & t_y & 1 \end{bmatrix}$$

Иногда для простоты третью колонку опускают, поскольку она не оказывает влияния на конечный результат. Новые координаты каждой точки элемента после преобразования с помощью матрицы вычисляются по следующей формуле:

$$x' = a\,x + c\,y + t_x$$
$$y' = b\,x + d\,y + t_y$$

Коэффициент	**Описание**
a	Изменение масштаба по горизонтали. Значение больше 1 расширяет элемент, меньше 1, наоборот, сжимает.
b	Наклон по горизонтали. Положительное значение наклоняет влево, отрицательное вправо.
c	Наклон по вертикали. Положительное значение наклоняет вверх, отрицательное вниз.
d	Изменение масштаба по вертикали. Значение больше 1 расширяет элемент, меньше 1 — сжимает.
tx	Смещение по горизонтали в пикселах. Положительное значение сдвигает элемент вправо на заданное число пикселов, отрицательное значение сдвигает влево.

ty	Смещение по вертикали в пикселах. При положительном значении элемент опускается на заданное число пикселов вниз или вверх при отрицательном значении.

Для работы с трансформацией необходимо включить расширенный графический режим:

int SetGraphicsMode(HDC hdc, int iMode); GM_ADVANCED

Матрица трансформаций представлена структурой:

```
struct XFORM {
    FLOAT eM11;
    FLOAT eM12;
    FLOAT eM21;
    FLOAT eM22;
    FLOAT eDx;
    FLOAT eDy;
};
```

Переменные соответствуют следующим элементам матрицы:

$$\begin{vmatrix} eM11 & eM12 & 0 \\ eM21 & eM22 & 0 \\ eDx & eDy & 1 \end{vmatrix}$$

Виды трансформаций

Параллельный перенос:

$$|x'\ y'\ 1| = |x\ \ y\ 1| * \begin{vmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ eDx & eDy & 1 \end{vmatrix}$$

Масштабирование:

$$|x'\ y'\ 1| = |x\ \ y\ 1| * \begin{vmatrix} eM11 & 0 & 0 \\ 0 & eM22 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Отражение:

$$|x'\ y'\ 1| = |x\ \ y\ 1| * \begin{vmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Поворот:

$$|x'\ y'\ 1| = |\ x\ \ y\ 1| * \begin{vmatrix} cos & sin & 0 \\ -sin & cos & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Наклон:

$$|x'\ y'\ 1| = |\ x\ \ y\ 1| * \begin{vmatrix} 1 & eM12 & 0 \\ eM21 & 1 & 0 \\ 0 & 0 & 1 \end{vmatrix}$$

Режимы масштабирования

В процессе вывода изображения функции графического интерфейса GDI преобразуют логические координаты в физические. Для определения способа такого преобразования используется атрибут с названием режим масштабирования (mapping mode), который хранится в контексте устройства вывода.

Для установки типа масштабирования используется метод контекста устройства **int SetMapMode(HDC hdc, int fnMapMode)**, а для получения типа масштабирования - метод **GetMapMode()**.

Для указания режима масштабирования в файле windows.h определены символьные константы с префиксом MM_ (от Mapping Mode - режим масштабирования).

Восемь существующих режимов масштабирования координат задаются с помощью символьных констант, определенных в файле Wingdi.h:

Режим масштабирования	Логических единиц	Физических единиц	Направление осей	
			X	У
MM_TEXT	1	1 pixel	→	↓
MM_LOMETRIC	10	1 mm	→	↑
MM_HIMETRIC	100	1 mm	→	↑
MM_LOENGLISH	100	1 inch	→	↑
MM_HIENGLISH	1000	1 inch	→	↑
MM_TWIPS	1440	1 inch	→	↑
MM_ISOTROPIC	Задается	Задается	→	↑
MM_ANISOTROPIC	Задается	Задается	→	↑

По умолчанию действует режим MM_TEXT, в котором ось Y имеет направление сверху вниз.

Отличие MM_ISOTROPIC от MM_ANISOTROPIC заключается в том, что, при масштабировании в первом из этих двух режимов, оси координат будут масштабироваться равномерно, то есть изображение будет масштабироваться без искажений. Для определения ориентации осей координат и единиц измерения необходимо использовать функции **SetWindowExtEx** и **SetViewportExtEx**.

13. Понятие ресурсов программ Windows. Виды ресурсов. Операции с ресурсами.

Ресурсы – двоичные данные, записываемые в исполняемый модуль приложения.

Стандартные виды ресурсов:

- Курсор – Cursor
- Картинка – Bitmap
- Значок – Icon
- Меню – Menu
- Окно диалога – Dialog Box
- Таблица строк – String Table
- Таблица сообщений (об ошибках) – Message Table
- Шрифт – Font
- Таблица горячих клавиш – Accelerator Table
- Информация о версии – Version Information
- Ресурс Plug and Play
- Ресурс VXD
- Ресурс HTML
- Манифест приложения – Side-by-Side Assembly Manifest
- Двоичные данные – RCData

Окна выполняемые в монопольном режиме – окна диалога. В ОС окна диалога загружаются из ресурса и выполняются с помощью одной функции.

Программист может создавать свои собственные типы ресурсов. Ресурсы – отдельны от кода и данных. Ресурсы обладают свойством разделяемости. Несколько программ могут использовать копию одного ресурса. В .exe файле ресурс может идентифицироваться числом (0..65536) или строкой. Ресурсы создаются на языке описания ресурсов и располагается в фалах *.rc. При сборке exe файла rc файлы дописываются в exe файл.

Добавление и удаление ресурсов исполняемого модуля:

- HANDLE BeginUpdateResource(LPCTSTR pFileName, bool bDeleteExistingResources);
- bool UpdateResource(HANDLE hUpdate, LPCTSTR lpType, LPCTSTR lpName, WORD wLanguage, void* lpData, DWORD cbData);
- bool EndUpdateResource(HANDLE hUpdate, bool fDiscard);

Функция BeginUpdateResource возвращает дескриптор, который может быть использован функцией UpdateResource для добавления, удаления или замены ресурсов в исполняемом файле.Процесс записи ресурсов в файл начинается с вызова BeginUpdateResource. При этом флаг bDeleteExistingResources задает режим записи: с удалением существующих ресурсов или без.Заканчивается процесс записи вызовом EndUpdateResource. Если флаг bDiscard установлен в TRUE, то запись ресурсов отменяется, в противном случае ресурсы записываются в файл.

Между вызовами этих двух функций можно обновлять ресурсы с помощью функции UpdateResource, причем вызывать ее можно неоднократно.Функция UpdateResource добавляет, удаляет или заменяет ресурс в исполняемом файле.

Загрузка ресурсов из исполняемого модуля:

- HRSRC FindResourceEx(HMODULE hModule, LPCTSTR lpType, LPCTSTR lpName, WORD wLanguage); FindResource, EnumResourceXxx.

· HGLOBAL LoadResource(HMODULE hModule, HRSRC hResInfo); LoadImage, LoadMenu, LoadXxx.

· DWORD SizeofResource(HMODULE hModule, HRSRC hResInfo);

Функция FindResource выясняет место ресурса с заданным типом и именем в указанном модуле.

Функция LoadResource загружает указанный ресурс в глобальную память.

14. Понятие динамически-загружаемой библиотеки. Создание DLL-библиотеки. Использование DLL-библиотеки в программе методом статического импорта процедур. Соглашения о вызовах процедур DLL-библиотеки. Точка входа-выхода DLL-библиотеки.

- Динамически-загружаемая библиотека (DLL) – двоичный модуль операционной системы. Это программа с множеством точек входа. Включает код, данные и ресурсы.
- Подключение DLL называется импортом. Существуют статический импорт и динамический импорт. При статическом импорте динамическая библиотека подключается как статическая, но находится в отдельном исполняемом файле и поэтому может быть заменена перед стартом. При динамическом импорте загрузка и получение адресов функций динамической библиотеки происходит вручную во время работы программы.

· Создавая DLL, Вы создаете набор функций, которые могут быть вызваны из EXE-модуля (или другой DLL). DLL может экспортировать переменные, функции или C++ классы в другие модули. На самом деле я бы не советовал экспортировать переменные, потому что это снижает уровень абстрагирования Вашего кода и усложняет его поддержку. Кроме того, C++ классы можно экспортировать, только если импортирующие их модули транслируются тем же компилятором. Так что избегайте экспорта C++ классов, если Вы не уверены, что разработчики EXE модулей будут пользоваться тем же компилятором.

· При разработке DLL Вы сначала создаете заголовочный файл, в котором содержатся экспортируемые из нее переменные (типы и имена) и функции (прототипы и имена). В этом же файле надо определить все идентификаторы и структуры данных, используемые экспортируемыми функциями и переменными. Заголовочный файл включается во все модули исходного кода Вашей DLL. Более того, Вы должны поставлять его вместе со своей DLL, чтобы другие разработчики могли включать его в свои модули исходного кода, которые импортируют Ваши функции или переменные. Единый заголовочный файл, используемый при сборке DLL и любых исполняемых модулей, существенно облегчает поддержку приложения.

Модуль: MyLib.h
**********/
#ifdef MYLIBAPI
// MYLIBAPI должен быть определен во всех модулях исходного кода DLL
// до включения этого файла
// здесь размещаются все экспортируемые функции и переменные
#else
// этот заголовочный файл включается в исходный код EXE-файла; // указываем, что все функции и переменные импортируются #define MYLIBAPI extern "C" __declspec(dllimport)

#endif
//
// здесь определяются все структуры данных и идентификаторы (символы)
//
// Здесь определяются экспортируемые переменные. // Примечание: избегайте экспорта переменных. MYLIBAPI int g_nResult;

//
// здесь определяются прототипы экспортируемых функций MYLIBAPI int Add(int nLeft, int nRight);

Этот заголовочный файл надо включать в самое начало исходных файлов Вашей DLL следующим образом.

/**********

Модуль: MyLibFile1.cpp
**********/
// сюда включаются стандартные заголовочные файлы Windows и библиотеки C #include <windows.h>

// этот файл исходного кода DLL экспортирует функции и переменные
#define MYLIBAPI extern "C" __declspec(dllexport)

// включаем экспортируемые структуры данных, идентификаторы, функции и переменные

#include "MyLib.h" //

// здесь размещается исходный код этой DLL int g_nResult;

int Add(int nLeft, int nRight) { g_nResult = nLeft + nRight; return(g_nResult);

}

При компиляции исходного файла DLL, показанного на предыдущем листинге, MYLIBAPI определяется как *__declspec(dllexport)* до включения заголовочного файла MyLib.h. Такой модификатор означает, что данная переменная, функция или C++ класс экспортируется из DLL. Заметьте, что идентификатор MYLIBAPI помещен в заголовочный файл до определения экспортируемой переменной или функции.

Также обратите внимание, что в файле MyLibFile1.cpp перед экспортируемой переменной или функцией не ставится идентификатор MYLIBAPI. Он здесь не нужен: проанализировав заголовочный файл, компилятор запоминает, какие переменные и функции являются экспортируемыми.

Идентификатор MYLIBAPI включает *extern*. Пользуйтесь этим модификатором только в коде на C++, но ни в коем случае не в коде на стандартном C. Обычно компиляторы C++ искажают (mangle) имена функций и переменных, что может приводить к серьезным ошибкам при компоновке. Представьте, что DLL написана на C++, а исполняемый код — на стандартном C. При сборке DLL имя функции будет искажено, но при сборке исполняемого модуля — нет. Пытаясь скомпоновать исполняемый модуль, компоновщик сообщит об ошибке: исполняемый модуль обращается к несуществующему идентификатору. Модификатор *extern* не дает компилятору искажать имена переменных или функций, и они становятся доступными исполняемым модулям, написанным на C, C++ или любом другом языке программирования.

Теперь Вы знаете, как используется заголовочный файл в исходных файлах DLL. А как насчет исходных файлов EXE-модуля? В них MYLIBAPI определять не надо: включая заголовочный файл, Вы определяете этот идентификатор как *__declspec(dllimport)*, и при компиляции исходного кода EXE-модуля компилятор поймет, что переменные и функции импортируются из DLL.

Просмотрев стандартные заголовочные файлы Windows (например, WinBase.h), Вы обнаружите, что практически тот же подход исповедует и Microsoft.

Статический импорт DLL-библиотеки

- Экспорт функции при создании DLL:

n __declspec(dllexport) int Min(int X, int Y);

- Импорт функции из DLL:

- Добавить библиотеку DLL в проект Visual Studio.

- __declspec(dllimport) int Min(int X, int Y);

- При сборке проекта будет создана статическая библиотека импорта с расширением LIB. Эта статическая библиотека включается в EXE-файл и содержит код вызова функции Min из DLL-библиотеки.

- Соглашения о вызовах подпрограмм:

- __declspec(dllimport) int __stdcall Min(int X, int Y);

- __cdecl – Параметры передаются на стек в обратном порядке. За освобождение стека после вызова подпрограммы отвечает вызывающая программа.

- __pascal – Передача параметров на стек в прямом порядке. Освобождение стека осуществляет сама вызванная подпрограмма.

- __stdcall – Соглашение для стандартных DLL ОС Windows. Передача параметров на стек происходит в обратном порядке. Освобождение стека выполняет вызванная подпрограмма.

__register – Передача параметров преимущественно через регистры процессора. Не используется при создании DLL, поскольку не стандартизировано.

- Соглашения о вызовах подпрограмм:

n- Разные способы передачи параметров создают трудности. Главная трудность связана с применением соглашения __stdcall. В VisualStudio использование соглашение о вызовах __stdcall вводит определенные правила именования функций в DLL. Функция получает имя: _Имя@КоличествоБайтПараметров.

_Min@8

Библиотека импорта может создаваться вручную на основе существующей DLL библиотеки. Для этого создается текстовый DEF-файл описания DLL библиотеки и включается в проект.

EXPORTS

Min,Max

При наличии DEF-файла компилятор выбирает из него имена для функций.

Функция входа/выхода

В DLL может быть лишь одна функция входа/выхода. Система вызывает ее в некоторых ситуациях (о чем речь еще впереди) сугубо в информационных целях, и обычно она используется DLL для инициализации и очистки ресурсов в конкретных процессах или потоках. Если Вашей DLL подобные уведомления не нужны, Вы не обязаны реализовывать эту функцию. Пример — DLL, содержащая только ресурсы. Но если же уведомления необходимы, функция должна выглядеть так:

```
BOOL WINAPI DllMain(HINSTANCE hinstDll, DWORD fdwReason, PVOID fImpLoad) {
        switch (fdwReason) {
        case DLL_PROCESS_ATTACH:
// DLL проецируется на адресное пространство процесса
break;
case DLL_THREAD_ATTACH: // создается поток break;
        case DLL_THREAD_DETACH:
                // поток корректно завершается
        break;
```

```
case DLL_PROCESS_DETACH:// DLL отключается от адресного пространства процесса break;
}
return(TRUE); // используется только для DLL_PROCESS_ATTACH }
```

Не забывайте, что DLL инициализируют себя, используя функции *DllMain*. К моменту выполнения Вашей *DllMain* другие DLL в том же адресном пространстве могут не успеть выполнить свои функции *DllMain*, т. е. они окажутся неинициализированными. Поэтому Вы должны избегать обращений из *DllMain* к функциям, импортируемым из других DLL. Кроме того, не вызывайте из *DllMain* функции *LoadLibrary(Ex)* и *FreeLibrary*, так как это может привести к взаимной блокировке.

В документации Platform SDK утверждается, что *DllMain* должна выполнять лишь простые виды инициализации — настройку локальной памяти потока, создание объектов ядра, открытие файлов и т. д. Избегайте обращений к функциям, связанным с User, Shell, ODBC, COM, RPC и сокетами (а также к функциям, которые их вызывают), потому что соответствующие DLL могут быть еще не инициализированы. Кроме того, подобные функции могут вызывать *LoadLibrary(Ex)* и тем самым приводить к взаимной блокировке.

Аналогичные проблемы возможны и при создании глобальных или статических C++ объектов, поскольку их конструктор или деструктор вызывается в то же время, что и Ваша *DllMain*.

Уведомление DLL_PROCESS_ATTACH

Система вызывает *DllMain* с этим значением параметра *fdwReason* сразу после того, как DLL спроецирована на адресное пространство процесса. А это происходит, только когда образ DLL-файла проецируется в первый раз. Если затем поток вызовет *LoadLibrary(Ex)* для уже спроецированной DLL, система просто увеличит счетчик числа пользователей этой DLL; так что *DllMain* вызывается со значением DLL_PROCESS_ATTACH лишь раз.

Обрабатывая DLL_PROCESS_ATTACH, библиотека должна выполнить в процессе инициализацию, необходимую ее функциям. Например, в DLL могут быть функции, которым нужна своя куча (создаваемая в адресном пространстве процесса). В этом случае *DllMain* могла бы создать такую кучу, вызвав *HeapCreate* при обработке уведомления DLL_PROCESS_ATTACH, а описатель созданной кучи сохранить в глобальной переменной, доступной функциям DLL.

При обработке уведомления DLL_PROCESS_ATTACH значение, возвращаемое функцией *DllMain*, указывает, корректно ли прошла инициализация DLL. Например, если вызов *HeapCreate* закончился благополучно, следует вернуть TRUE. А если кучу создать не удалось — FALSE. Для любых других значений *fdwReason* — DLL_PROCESS_DETACH, DLL_THREAD_ATTACH или DLL_THREAD_DETACH — значение, возвращаемое *DllMain*, системой игнорируется.

Уведомление DLL_PROCESS_DETACH

При отключении DLL от адресного пространства процесса вызывается ее функция *DllMain* со значением DLL_PROCESS_DETACH в параметре *fdwReason*. Обрабатывая это значение, DLL должна провести очистку в данном процессе. Например, вызвать *HeapDestroy*, чтобы разрушить кучу, созданную ею при обработке уведомления DLL_PROCESS_ATTACH. Обратите внимание: если функция *DllMain* вернула FALSE, получив уведомление DLL_PROCESS_ATTACH, то ее нельзя вызывать с уведомлением DLL_PROCESS_DETACH. Если DLL отключается из-за завершения процесса, то за выполнение кода *DllMain* отвечает поток, вызвавший *ExitProcess* (обычно это первичный поток приложения). Когда Ваша входная функция возвращает управление стартовому коду из библиотеки C/C++, тот явно вызывает *ExitProcess* и завершает процесс.

Если DLL отключается в результате вызова *FreeLibrary* или *FreeLibraryAndExitThread*, код *DllMain* выполняется потоком, вызвавшим одну из этих функций. В случае обращения к *FreeLibrary* управление не возвращается, пока *DllMain* не закончит обработку уведомления

DLL_PROCESS_DETACH. Учтите также, что DLL может помешать завершению процесса, если, например, ее *DllMain* входит в бесконечный цикл, получив уведомление DLL_PROCESS_DETACH. Операционная система уничтожает процесс только после того, как все DLL-модули обработают уведомление DLL_PROCESS_DETACH.

Если процесс завершается в результате вызова *TerminateProcess*, система *не* вызывает *DllMain* со значением DLL_PROCESS_DETACH. А значит, ни одна DLL, спроецированная на адресное пространство процесса, не получит шанса на очистку до завершения процесса. Последствия могут быть плачевны — вплоть до потери данных. Вызывайте *TerminateProcess* только в самом крайнем случае!

Уведомление DLL_THREAD_ATTACH

Когда в процессе создается новый поток, система просматривает все DLL, спроецированные в данный момент на адресное пространство этого процесса, и в каждой из таких DLL вызывает *DllMain* со значением DLL_THREAD_ATTACH. Тем самым она уведомляет DLL модули о необходимости инициализации, связанной с данным потоком. Только что созданный поток отвечает за выполнение кода в функциях *DllMain* всех DLL. Работа его собственной (стартовой) функции начинается лишь после того, как все DLL модули обработают уведомление DLL_THREAD_ATTACH.

Если в момент проецирования DLL на адресное пространство процесса в нем выполняется несколько потоков, система *не* вызывает *DllMain* со значением DLL_ THREAD_ATTACH ни для одного из существующих потоков. Вызов *DllMain* с этим значением осуществляется, только если DLL проецируется на адресное пространство процесса в момент создания потока.

Обратите также внимание, что система не вызывает функции *DllMain* со значением DLL_THREAD_ATTACH и для первичного потока процесса. Любая DLL, проецируемая на адресное пространство процесса в момент его создания, получает уведомление DLL_PROCESS_ATTACH, а не DLL_THREAD_ATTACH.

Уведомление DLL_THREAD_DETACH

Лучший способ завершить поток — дождаться возврата из его стартовой функции, после чего система вызовет *ExitThread* и закроет поток. Эта функция лишь сообщает системе о том, что поток хочет завершиться, но система не уничтожает его немедленно. Сначала она просматривает все проекции DLL, находящиеся в данный момент в адресном пространстве процесса, и заставляет завершаемый поток вызвать *DllMain* в каждой из этих DLL со значением DLL_THREAD_DETACH. Тем самым она уведомляет DLL модули о необходимости очистки, связанной с данным потоком. Например, DLL версия библиотеки C/C++ освобождает блок данных, используемый для управления многопоточными приложениями.

Заметьте, что DLL может не дать потоку завершиться. Например, такое возможно, когда функция *DllMain*, получив уведомление DLL_THREAD_DETACH, входит в бесконечный цикл. А операционная система закрывает поток только после того, как все DLL заканчивают обработку этого уведомления.

Если поток завершается из-за того, что другой поток вызвал для него *TerminateThread*, система *не* вызывает *DllMain* со значением DLL_THREAD_DETACH. Следовательно, ни одна DLL, спроецированная на адресное пространство процесса, не получит шанса на выполнение очистки до завершения потока, что может привести к потере данных. Поэтому *TerminateThread*, как и *TerminateProcess*, можно использовать лишь в самом крайнем случае!

Если при отключении DLL еще выполняются какие-то потоки, то для них *DllMain* не вызывается со значением DLL_THREAD_DETACH. Вы можете проверить это при обработке DLL_PROCESS_DETACH и провести необходимую очистку.

Ввиду упомянутых выше правил не исключена такая ситуация: поток вызывает *LoadLibrary* для загрузки DLL, в результате чего система вызывает из этой библиотеки *DllMain* со значением DLL_PROCESS_ATTACH. (В этом случае уведомление DLL_ THREAD_ATTACH не посылается.) Затем поток, загрузивший DLL, завершается, что приводит к новому вызову *DllMain* — на этот раз со значением DLL_THREAD_DETACH. Библиотека уведомляется о завершении потока, хотя она не получала DLL_ THREAD_ATTACH, уведомляющего о его подключении. Поэтому будьте крайне осторожны при выполнении любой очистки, связанной с конкретным потоком. К счастью, большинство программ пишется так, что *LoadLibrary* и *FreeLibrary* вызываются одним потоком.

15. Понятие динамически-загружаемой библиотеки. Создание DLL-библиотеки. Использование DLL-библиотеки в программе методом динамический импорта процедур.

1 и 2 часть вопроса смотри в вопросе номер 14

Загрузка DLL-библиотеки в память:

HMODULE LoadLibrary(LPCTSTR lpFileName);

HMODULE LoadLibraryEx(LPCTSTR lpFileName, _Reserved_ HANDLE hFile, DWORD dwFlags);

HMODULE GetModuleHandle(LPCTSTR lpModuleName); GetModuleHandleEx.

DWORD GetModuleFileName(HMODULE hModule, LPTSTR lpFilename, DWORD nSize);

Освобождение DLL-библиотеки:

bool FreeLibrary(HMODULE hModule); FreeLibraryAndExitThread.

Получение адреса функции в DLL-библиотеке:

void* GetProcAddress(HMODULE hModule, LPCSTR lpProcName);

Применение:

typedef int TMin(int x, int y); // добавить __stdcall

TMin* pMin;

pMin = (TMin*)GetProcAddress(hModule, "_Min@8");

int a = pMin(10, 20);

16. Понятие динамически-загружаемой библиотеки. Создание в DLL-библиотеке разделяемых между приложениями глобальных данных. Разделы импорта и экспорта DLL-библиотеки. Переадресация вызовов процедур DLL-библиотек к другим DLL-библиотекам. Исключение конфликта версий DLL.

1 часть в вопросе номер 14
Разделяемые данные – shared data:

```
#pragma section("mysection", read, write, shared)

__declspec(allocate("mysection")) int Number = 0;
```

Экспорт

Если модификатор *__declspec(dllexport)* указан перед переменной, прототипом функции или C++классом, компилятор Microsoft C/C++ встраивает в конечный OBJфайл дополнительную информацию. Она понадобится компоновщику при сборке DLL из OBJфайлов.

Обнаружив такую информацию, компоновщик создает LIB-файл со списком идентификаторов, экспортируемых из DLL. Этот LIB-файл нужен при сборке любого EXE-модуля, ссылающегося на такие идентификаторы. Компоновщик также вставляет в конечный DLL-файл таблицу экспортируемых идентификаторов — *раздел экспорта*, в котором содержится список (в алфавитном порядке) идентификаторов экспортируемых функций, переменных и классов. Туда же помещается *относительный виртуальный адрес* (relative virtual address, RVA) каждого идентификатора внутри DLL-модуля.

Воспользовавшись утилитой DumpBin.exe (с ключом *-exports*) из состава Microsoft Visual Studio, мы можем увидеть содержимое раздела экспорта в DLL-модуле.

Импорт

Разрешая ссылки на импортируемые идентификаторы, компоновщик создает в конечном EXE-модуле *раздел импорта* (imports section). В нем перечисляются DLL, необходимые этому модулю, и идентификаторы, на которые есть ссылки из всех используемых DLL.

Воспользовавшись утилитой DumpBin.exe (с ключом *-imports*), мы можем увидеть содержимое раздела импорта. Ниже показан фрагмент полученной с ее помощью таблицы импорта Calc.exe.

```
C:\WINNT\SYSTEM32>DUMPBIN >imports Calc.EXE

        Microsoft (R) COFF Binary File Dumper Version 6.00.8168
        Copyright (C) Microsoft Corp 1992-1998. All rights reserved.
        Dump of file calc.exe
        File Type: EXECUTABLE IMAGE
        Section contains the following imports:
        SHELL32.dll
                10010F4 Import Address Table
                1012820 Import Name Table
```

```
            FFFFFFFF time date stamp
            FFFFFFFF Index of first forwarder reference
         77C42983           7A  ShellAboutW
      MSVCRT.dll
            1001094 Import Address Table
            10127C0 Import Name Table
            FFFFFFFF time date stamp
            FFFFFFFF Index of first forwarder reference
      78010040   295  memmove
      78018124       42  _EH_prolog
      78014C34   2D1  toupper
      78010F6E   2DD  wcschr
      78010668   2E3  wcslen

ADVAPI32.dll
      1001000 Import Address Table
         101272C Import Name Table
      FFFFFFFF time date stamp
      FFFFFFFF Index of first forwarder reference
 779858F4   19A  RegQueryValueExA
 77985196   190  RegOpenKeyExA
 77984BA1   178  RegCloseKey
KERNEL32.dll
      100101C Import Address Table
      1012748 Import Name Table
      FFFFFFFF time date stamp
      FFFFFFFF Index of first forwarder reference
 77ED4134   336  lstrcpyW
 77ED33E8   1E5  LocalAlloc
 77EDEF36    DB  GetCommandLineW
 77ED1610   15E  GetProfileIntW
 77ED4BA4   1EC  LocalReAlloc
Header contains the following bound import information:
 Bound to SHELL32.dll [36E449E0] Mon Mar 08 14:06:24 1999
 Bound to MSVCRT.dll [36BB8379] Fri Feb 05 15:49:13 1999
 Bound to ADVAPI32.dll [36E449E1] Mon Mar 08 14:06:25 1999
 Bound to KERNEL32.dll [36DDAD55] Wed Mar 03 13:44:53 1999
 Bound to GDI32.dll [36E449E0] Mon Mar 08 14:06:24 1999
 Bound to USER32.dll [36E449E0] Mon Mar 08 14:06:24 1999
Summary

2000 .data

      3000 .rsrc
      13000 .text
```

Как видите, в разделе есть записи по каждой DLL, необходимой Calc.exe: Shell32.dll, MSVCRt.dll, AdvAPI32.dll, Kernel32.dll, GDI32.dll и User32.dll. Под именем DLL-модуля выводится список

идентификаторов, импортируемых программой Calc.exe. Например, Calc.exe обращается к следующим функциям из Kernel32.dll: *lstrcpyW, LocalAlloc, GetCommandLineW, GetProfileIntW* и др.

Число слева от импортируемого идентификатора называется «подсказкой» (hint) и для нас несущественно. Крайнее левое число в строке для идентификатора сообщает адрес, по которому он размещен в адресном пространстве процесса. Такой адрес показывается, только если было проведено связывание (binding) исполняемого модуля.

Переадресация вызовов функций

Запись о переадресации вызова функции (function forwarder) — это строка в разделе экспорта DLL, которая перенаправляет вызов к другой функции, находящейся в другой DLL. Например, запустив утилиту DumpBin из Visual C++ для Kernel32.dll в Windows 2000, Вы среди прочей информации увидите и следующее.

```
C:\winnt\system32>DumpBin    Exports Kernel32.dll
        (часть вывода опущена)
.               360  167 HeapAlloc (forwarded to NTDLL.RtlAllocateHeap)
.               361  168 HeapCompact (000128D9)
.               362  169 HeapCreate (000126EF)
.               363  16A HeapCreateTagsW (0001279E)
.               364  16B HeapDestroy (00012750)
.               365  16C HeapExtend (00012773)
.               366  16D HeapFree (forwarded to NTDLL.RtlFreeHeap)
.               367  16E HeapLock (000128ED)
.               368  16F HeapQueryTagW (000127B8)
.               369  170 HeapReAlloc (forwarded to NTDLL.RtlReAllocateHeap)
.               370  171 HeapSize (forwarded to NTDLL.RtlSizeHeap)        (остальное тоже опущено)
```

Здесь есть четыре переадресованные функции. Всякий раз, когда Ваше приложение вызывает *HeapAlloc*, *HeapFree*, *HeapReAlloc* или *HeapSize*, его EXE модуль динамически связывается с Kernel32.dll. При запуске EXE модуля загрузчик загружает Kernel32.dll и, обнаружив, что переадресуемые функции на самом деле находятся в NTDLL.dll, загружает и эту DLL. Обращаясь к *HeapAlloc*, программа фактически вызывает функцию *RtlAllocateHeap* из NTDLL.dll. А функции *HeapAlloc* вообще нет!

При вызове *HeapAlloc* (см. ниже) функция *GetProcAddress* просмотрит раздел эк порта Kernel32.dll и, выяснив, что *HeapAlloc* — переадресуемая функция, рекурсивно вызовет сама себя для поиска *RtlAllocateHeap* в разделе экспорта NTDLL.dll. GetProcAddress(GetModuleHandle("Kernel32"), "HeapAlloc");

Вы тоже можете применять переадресацию вызовов функций в своих DLL. Самый

простой способ — воспользоваться директивой *pragma*: // переадресация к функции из DllWork

```
#pragma comment(linker, "/export:SomeFunc=DllWork.SomeOtherFunc")
```

Эта директива сообщает компоновщику, что DLL должна экспортировать функцию *SomeFunc*, которая на самом деле реализована как функция *SomeOtherFunc* в модуле DllWork.dll. Такая запись нужна для каждой переадресуемой функции.

1. Исключение конфликта версий DLL:
a. **c:\myapp\myapp.exe** загружает старую версию
b. **c:\program files\common files\system\mydll.dll**,
c. а должен загружать **mydll.dll** из своего же каталога.
d. n Создать пустой файл **c:\myapp\myapp.exe.local**.
e. Будет грузиться библиотека **c:\myapp\mydll.dll**.
f. n Создать каталог **c:\myapp\myapp.exe.local**.
g. Будет грузиться **c:\myapp\myapp.exe.local\mydll.dll**.
h. n Создать файл манифеста для приложения. В этом случае .local файлы будут игнорироваться.

Когда разрабатывались первые версии Windows, оперативная память и дисковое пространство были крайне дефицитным ресурсом, так что Windows была рассчитана на предельно экономное их использование — с максимальным разделением между потребителями. В связи с этим Microsoft рекомендовала размещать все модули, используемые многими приложениями (например, библиотеку C/C++ и DLL, относящиеся к MFC) в системном каталоге Windows, где их можно было легко найти.

Однако со временем это вылилось в серьезную проблему: программы установки приложений то и дело перезаписывали новые системные файлы старыми или не полностью совместимыми. Из-за этого уже установленные приложения переставали работать. Но сегодня жесткие диски стали очень емкими и недорогими, оперативная память тоже значительно подешевела. Поэтому Microsoft сменила свою позицию на прямо противоположную: теперь она настоятельно рекомендует размещать все файлы приложения в своем каталоге и ничего не трогать в системном каталоге Windows. Тогда Ваше приложение не нарушит работу других программ, и наоборот.

С той же целью Microsoft ввела в Windows 2000 поддержку перенаправления DLL (DLL redirection). Она заставляет загрузчик операционной системы загружать модули сначала из каталога Вашего приложения и, только если их там нет, искать в других каталогах.

Чтобы загрузчик всегда проверял сначала каталог приложения, нужно всего лишь поместить туда специальный файл. Его содержимое не имеет значения и игнорируется — важно только его имя: оно должно быть в виде AppName.local. Так, если исполняемый файл Вашего приложения — SuperApp.exe, присвойте перенаправляющему файлу имя SuperApp.exe.local.

Функция *LoadLibrary(Ex)* проверяет наличие этого файла и, если он есть, загружает модуль из каталога приложения; в ином случае *LoadLibrary(Ex)* работает так же, как и раньше.

Перенаправление DLL исключительно полезно для работы с зарегистрированными COM объектами. Оно позволяет приложению размещать DLL с COM объектами в своем каталоге, и другие программы, регистрирующие те же объекты, не будут мешать его нормальной работе.

17. Понятие объекта ядра ОС Windows. Виды объектов ядра. Атрибуты защиты объекта ядра. Дескриптор защиты объекта ядра. Создание и удаление объектов ядра.

Что такое объект ядра

Создание, открытие и прочие операции с объектами ядра станут для Вас, как разработчика Windows-приложений, повседневной рутиной. Система позволяет создавать и оперировать с несколькими

типами таких объектов, в том числе: маркерами доступа (access token objects), файлами (file objects), проекциями файлов (file-mapping objects), портами завершения ввода-вывода (I/O completion port objects), заданиями (job objects), почтовыми ящиками (mailslot objects), мьютексами (mutex objects), каналами (pipe objects), процессами (process objects), семафорами (semaphore objects), потоками (thread objects) и ожидаемыми таймерами (waitable timer objects). Эти объекты создаются Windows-функциями. Каждый объект ядра — на самом деле просто блок памяти, выделенный ядром и доступный только ему. Этот блок представляет собой структуру данных, в элементах которой содержится информация об объекте. Некоторые элементы (дескриптор защиты, счетчик числа пользователей и др.) присутствуют во всех объектах, но бо́льшая их часть специфична для объектов конкретного типа. Например, у объекта «процесс» есть идентификатор, базовый приоритет и код завершения, а у объекта «файл» — смещение в байтах, режим разделения и режим открытия. Поскольку структуры объектов ядра доступны только ядру, приложение не может самостоятельно найти эти структуры в памяти и напрямую модифицировать их содержимое. Такое ограничение Microsoft ввела намеренно, чтобы ни одна программа не нарушила целостность структур объектов ядра. Это же ограничение позволяет Microsoft вводить, убирать или изменять элементы структур, не нарушая работы каких-либо приложений. Но вот вопрос: если мы не можем напрямую модифицировать эти структуры, то как же наши приложения оперируют с объектами ядра? Ответ в том, что в Windows предусмотрен набор функций, обрабатывающих структуры объектов ядра по строго определенным правилам. Мы получаем доступ к объектам ядра только через эти функции. Когда Вы вызываете функцию, создающую объект ядра, она возвращает описатель, идентифицирующий созданный объект. Описатель следует рассматривать как «непрозрачное» значение, которое может быть использовано любым потоком Вашего процесса. Этот описатель Вы передаете Windows-функциям, сообщая системе, какой объект ядра Вас интересует. Но об описателях мы поговорим позже (в этой главе). Для большей надежности операционной системы Microsoft сделала так, чтобы значения описателей зависели от конкретного процесса. Поэтому, если Вы передадите такое значение (с помощью какого-либо механизма межпроцессной связи) потоку другого процесса, любой вызов из того процесса со значением описателя, полученного в Вашем процессе, даст ошибку.

Учет пользователей объектов ядра

Объекты ядра принадлежат ядру, а не процессу. Иначе говоря, если Ваш процесс вызывает функцию, создающую объект ядра, а затем завершается, объект ядра может быть не разрушен. В большинстве случаев такой объект все же разрушается; но если созданный Вами объект ядра используется другим процессом, ядро запретит разрушение объекта до тех пор, пока от него не откажется и тот процесс. Ядру известно, сколько процессов использует конкретный объект ядра, поскольку в каждом объекте есть счетчик числа его пользователей. Этот счетчик — один из элементов данных, общих для всех типов объектов ядра. В момент создания объекта счетчику присваивается 1. Когда к существующему объекту ядра обращается другой процесс, счетчик увеличивается на 1. А когда какой-то процесс завершается, счетчики всех используемых им объектов ядра автоматически уменьшаются на 1. Как только счетчик какого-либо объекта обнуляется, ядро уничтожает этот объект.

Защита

Объекты ядра можно защитить дескриптором защиты (security descriptor), который описывает, кто создал объект и кто имеет права на доступ к нему. Дескрипторы защиты обычно используют при написании серверных приложений; создавая клиентское приложение, Вы можете игнорировать это свойство объектов ядра.

Почти все функции, создающие объекты ядра, принимают указатель на структуру SECURITY_ATTRIBUTES как аргумент, например:

```
HANDLE CreateFileMapping( HANDLE hFile, PSECURITY_ATTRIBUTES psa, DWORD flProtect, DWORD dwMaximumSizeHigh, DWORD dwMaximumSizeLow, PCTSTR pszName);
```

Большинство приложений вместо этого аргумента передает NULL и создает объект с защитой по умолчанию. Такая защита подразумевает, что создатель объекта и любой член группы администраторов получают к нему полный доступ, а все прочие к объекту не допускаются. Однако Вы можете создать и инициализировать структуру SECURITY_ATTRIBUTES, а затем передать ее адрес. Она выглядит так:

```
typedef struct _SECURITY_ATTRIBUTES {
DWORD nLength;
LPVOID lpSecurityDescriptor;
BOOL bInheritHandle;
} SECURITY_ATTRIBUTES;
```

Хотя структура называется SECURITY_ATTRIBUTES, лишь один ее элемент имеет отношение к защите — lpSecurityDescriptor. Если надо ограничить доступ к созданному Вами объекту ядра, создайте дескриптор защиты и инициализируйте структуру SECURITY_ATTRIBUTES следующим образом:

```
SECURITY_ATTRIBUTES sa;
sa.nLength = sizeof(sa); // используется для выяснения версий
sa.lpSecurityDescriptor = pSD; // адрес инициализированной SD
sa.bInheritHandle = FALSE; // об этом позже
HANDLE hFileMapping = CreateFileMapping(INVALID_HANDLE_VALUE, &sa, PAGE_READWRITE, 0, 1024, "MyFileMapping");
```

Желая получить доступ к существующему объекту ядра (вместо того чтобы создавать новый), необходимо указать, какие операции Вы намерены проводить над объектом.

Кроме объектов ядра Ваша программа может использовать объекты других типов — меню, окна, курсоры мыши, кисти и шрифты. Они относятся к объектам User или GDI. Новичок в программировании для Windows может запутаться, пытаясь отличить объекты User или GDI от объектов ядра. Как узнать, например, чьим объектом — User или ядра — является данный значок? Выяснить, не принадлежит ли объект ядру, проще всего так: проанализировать функцию, создающую объект. Практически у всех функций, создающих объекты ядра, есть параметр, позволяющий указать атрибуты защиты, — как у CreateFileMapping. В то же время у функций, создающих объекты User или GDI, нет параметра типа PSECURITY_ATTRIBUTES.

Создание объекта ядра

Когда процесс инициализируется в первый раз, таблица описателей еще пуста. Но стоит одному из его потоков вызвать функцию, создающую объект ядра (например, CreateFileMapping), как ядро выделяет для этого объекта блок памяти и инициализирует его; далее ядро просматривает таблицу описателей, принадлежащую данному процессу, и отыскивает свободную запись. Поскольку таблица еще пуста, ядро обнаруживает структуру с индексом 1 и инициализирует ее. Указатель устанавливается на внутренний адрес структуры данных объекта, маска доступа — на доступ без ограничений и, наконец, определяется последний компонент — флаги. Функции, создающие объекты ядра обычно имеют вид HANDLE CreateXxx(…).

Все функции, создающие объекты ядра, возвращают описатели, которые привязаны к конкретному процессу и могут быть использованы в любом потоке данного процесса. Значение описателя представляет собой индекс в таблице описателей, принадлежащей процессу, и таким образом идентифицирует место, где хранится информация, связанная с объектом ядра. Вот поэтому при отладке своего приложения и просмотре фактического значения описателя объекта ядра Вы и видите такие малые величины: 1, 2 и т. д. Но помните, что физическое содержимое описателей не задокументировано и может быть изменено. Всякий раз, когда Вы вызываете функцию, принимающую описатель объекта ядра как аргумент, Вы передаете ей значение, возвращенное одной из Create-функций. При этом функция смотрит в таблицу описателей, принадлежащую Вашему процессу, и считывает адрес нужного объекта ядра. Если Вы передаете неверный индекс (описатель), функция завершается с ошибкой и GetLastError возвращает 6 (ERROR_INVALID_HANDLE). Это связано с тем, что на самом деле описатели представляют собой индексы в таблице, их значения привязаны к конкретному процессу и недействительны в других процессах. Если вызов функции, создающей объект ядра, оказывается неудачен, то обычно возвращается 0 (NULL). Такая ситуация возможна только при острой нехватке памяти или при наличии проблем с защитой. К сожалению, отдельные функции возвращают в таких случаях не 0, а –1 (INVALID_HANDLE_VALUE). Например, если CreateFile не сможет открыть указанный файл, она вернет именно INVALID_HANDLE_VALUE. Будьте очень осторожны при проверке значения, возвращаемого функцией, которая создает объект ядра. Так, для CreateMutex проверка на INVALID_HANDLE_VALUE бессмысленна:

```
HANDLE hMutex = CreateMutex(...);
if (hMutex == INVALID_HANDLE_VALUE) {
// этот код никогда не будет выполнен, так как при ошибке CreateMutex возвращает NULL
}
```

Точно так же бессмыслен и следующий код:

```
HANDLE hFile = CreateFile(...);
if (hFile == NULL) {
// и этот код никогда не будет выполнен, так как при ошибке CreateFile возвращает
INVALID_HANDLE_VALUE (-1)
 }
```

Закрытие объекта ядра

Независимо от того, как именно Вы создали объект ядра, по окончании работы с ним его нужно закрыть вызовом CloseHandle:

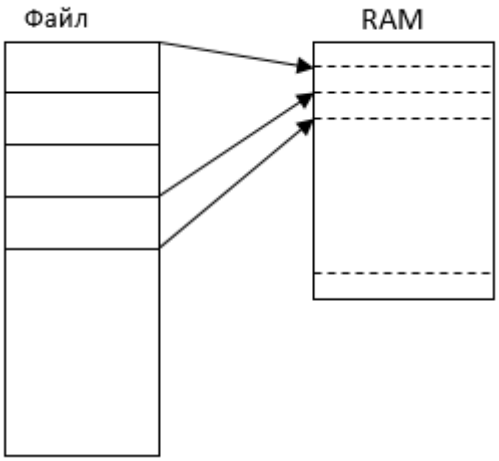
BOOL CloseHandle(HANDLE hobj);

Эта функция сначала проверяет таблицу описателей, принадлежащую вызывающему процессу, чтобы убедиться, идентифицирует ли переданный ей индекс (описатель) объект, к которому этот процесс действительно имеет доступ. Если переданный индекс правилен, система получает адрес структуры данных объекта и уменьшает в этой структуре счетчик числа пользователей; как только счетчик обнулится, ядро удалит объект из памяти. Если же описатель неверен, происходит одно из двух. В нормальном режиме выполнения процесса CloseHandle возвращает FALSE, а GetLastError — код ERROR_INVALID_HANDLE. Но при выполнении процесса в режиме отладки система просто уведомляет отладчик об ошибке. Перед самым возвратом управления CloseHandle удаляет соответствующую запись из таблицы описателей: данный описатель теперь недействителен в Вашем процессе и использовать его нельзя. При этом запись удаляется независимо от того, разрушен объект ядра или нет! После вызова CloseHandle Вы больше не получите доступ к этому объекту ядра; но, если его счетчик не обнулен, объект остается в памяти. Тут все нормально, это означает лишь то, что

объект используется другим процессом (или процессами). Когда и остальные процессы завершат свою работу с этим объектом (тоже вызвав CloseHandle), он будет разрушен. А вдруг Вы забыли вызвать CloseHandle — будет ли утечка памяти? И да, и нет. Утечка ресурсов (тех же объектов ядра) вполне вероятна, пока процесс еще исполняется. Однако по завершении процесса операционная система гарантированно освобождает все ресурсы, принадлежавшие этому процессу, и в случае объектов ядра действует так: в момент завершения процесса просматривает его таблицу описателей и закрывает любые открытые описатели.

18. Проецирование файлов в память. Отличие в механизме проецирования файлов в память в ОС Windows и UNIX/Linux. Действия по проецированию файла в память.

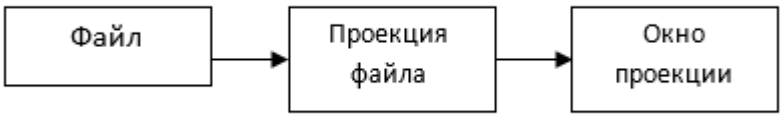
На платформах Win/Unix существуют средства для работы с файлами как с оперативной памятью.



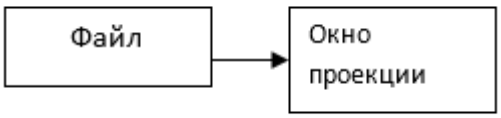
Идея в том, чтобы закрепить за началом файла какой-либо адрес памяти, а дальше выполнять чтение и запись файла методом чтения/записи байтов оперативной памяти. Т.к. файл не может поместится в оперативной памяти целиком, он делится на страницы и в оперативную память подгружаются лишь те страницы, с которыми происходит работа. Адресное пространство файла является виртуальным, оно может значительно превосходить по размерам оперативную память. Для прозрачной поддержки проецирования файлов в память необходимо иметь поддержку виртуальной памяти на уровне процессора и архитектуры компьютера.

В ОС Win процессы работают в виртуальном адресном пространстве, для которого создается на диске файл подкачки (swap). При проецировании файлов в память, файл подкачки не затрагивается, хотя проецирование происходит в виртуальное адресное пространство процесса. Такое возможно за счет аппаратной поддержки сложных таблиц страниц. В WIN – File Mapping; Unix – Memory Mapping.

В ОС Windows отображение файлов в память является двухуровневым:



В Unix схема отображения файлов одноуровневая:



Открытие или создание файла, объекта ядра файл, происходит с помощью функции:

HANDLE CreateFile(

LPCTSTR *lpFileName***,** // путь к файлу

DWORD *dwDesiredAccess***,** // указывается, будет файл читаться, записываться или и то и другое

DWORD *dwShareMode***,** // будет ли файл доступным для совместного использования со стороны других процессов. 0 – запретить сторонним процессам открывать этот файл

LPSECURITY_ATTRIBUTES *lpSecurityAttributes***,** // атрибуты защиты

DWORD *dwCreationDisposition***,** // режим открытия/создания файла

DWORD *dwFlagsAndAttributes***,** // флаги и атрибуты файла

HANDLE *hTemplateFile* // дескриптор файла с дополнительными атрибутами

);

Хотя в Win существует OpenFile, но использовать ее не рекомендуется. Открытие/создание рекомендуется производить CreateFile.

Шаги:

1. Открытие файла CreateFile.

2. Создание объекта ядра под названием «проекция файла».

```
HANDLE CreateFileMapping(
  HANDLE hFile,                  // Дескриптор файла полученный из CreateFile
  LPSECURITY_ATTRIBUTES lpAttributes,       // атрибуты защиты
  DWORD flProtect,               // Флаги.
```

Существуют флаги для отображения в память DLL и выполняемых файлов в формате PE (Portable Executable). Эти флаги обеспечивают автоматическое назначение областям кода и данных соответствующих атрибутов защиты. Коду устанавливается атрибут защиты READONLY, данным – WRITECOPY. Существуют дополнительные флаги, обеспечивающие разделяемость проецируемой памяти между процессами.

```
  DWORD dwMaximumSizeHigh,                  // high-order DWORD of size
```

Максимальный размер файла для режимов, в которых возможна запись в файла. Он может быть больше физического файла на диске. В этом случае размер дискового файла корректируется.

```
  DWORD dwMaximumSizeLow,                   // low-order DWORD of size
  LPCTSTR lpName                                    // Имя объекта ядра.
);
```

Проецирование файла на физическую память:

```
LPVOID MapViewOfFile(
  HANDLE hFileMappingObject,          // Дескриптор на проекцию файла.
  DWORD dwDesiredAccess,                    //  Набор  флагов,  определяющих,  какой все же будет доступ к памяти.
  DWORD dwFileOffsetHigh,             // high-order DWORD of offset
  DWORD dwFileOffsetLow,                    // low-order DWORD of offset
  SIZE_T dwNumberOfBytesToMap         // Размер окна проекции.
);
```

Функция создает окно проекции в проекции физической памяти и возвращает его адрес.

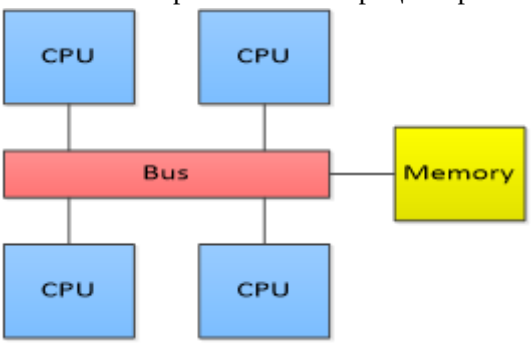
```
BOOL UnmapViewOfFile(
  LPCVOID lpBaseAddress                     // starting address
); //Закрывает окно проекции
```

```
BOOL FlushViewOfFile(
  LPCVOID lpBaseAddress,                    // starting address
  SIZE_T dwNumberOfBytesToFlush        // number of bytes in range
); //Записывает Все изменения в файл
```

Основное назначение файлов, проецируемых файлов – работа со сложными структурами данных, загрузка которых в память иным способом требует большого количества времени и сил.

19. Современные многопроцессорные архитектуры SMP и NUMA. Многоуровневое кэширование памяти в современных процессорах. Проблема перестановки операций чтения и записи в архитектурах с ослабленной моделью памяти. Способы решения проблемы перестановки операций чтения и записи.

Симметричная многопроцессорная архитектура – **SMP** (Symmetric Multi-Processor Architecture):



Главной особенностью систем с архитектурой SMP является наличие общей физической памяти, разделяемой всеми процессорами.

Память служит, в частности, для передачи сообщений между процессорами, при этом все вычислительные устройства при обращении к ней имеют равные права и одну и ту же адресацию для всех ячеек памяти. Поэтому SMP-архитектура называется симметричной. Последнее обстоятельство позволяет очень эффективно обмениваться данными с другими вычислительными устройствами. *SMP-система строится на основе высокоскоростной системной шины, к слотам которой подключаются функциональные блоки типов: процессоры, подсистема ввода/вывода и т.п.* Наиболее известными SMP-системами являются SMP-серверы и рабочие станции на базе процессоров Intel (IBM, HP, Compaq, Dell, ALR, Unisys, DG, Fujitsu и др.). Вся система работает под управлением единой ОС (обычно UNIX-подобной, но для Intel-платформ поддерживается Windows NT). ОС автоматически (в процессе работы) распределяет процессы по процессорам, но иногда возможна и явная привязка.

Основные **преимущества** SMP-систем:

· простота и универсальность для программирования. Архитектура SMP не накладывает ограничений на модель программирования, используемую при создании приложения: обычно используется модель параллельных ветвей, когда все процессоры работают независимо друг от друга. Однако можно реализовать и модели, использующие межпроцессорный обмен. Использование общей памяти увеличивает скорость такого обмена, пользователь также имеет доступ сразу ко всему объему памяти. Для SMP-систем существуют довольно эффективные средства автоматического распараллеливания;

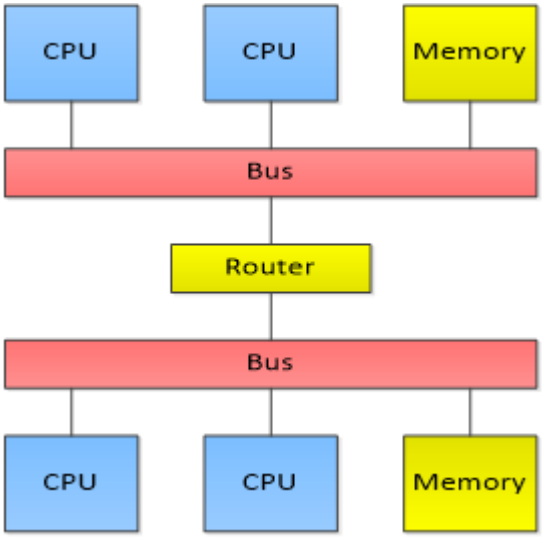
· простота эксплуатации. Как правило, SMP-системы используют систему кондиционирования, основанную на воздушном охлаждении, что облегчает их техническое обслуживание;

· относительно невысокая цена.

Основной **недостаток**: системы с общей памятью плохо масштабируются.

Этот существенный недостаток SMP-систем не позволяет считать их по-настоящему перспективными. Причиной плохой масштабируемости является то, что *в данный момент шина способна обрабатывать только одну транзакцию*, вследствие чего возникают проблемы разрешения конфликтов при одновременном обращении нескольких процессоров к одним и тем же областям общей физической памяти. Вычислительные элементы начинают друг другу мешать. Когда произойдет такой конфликт, зависит от скорости связи и от количества вычислительных элементов. В настоящее время конфликты могут происходить при наличии 8-24 процессоров. Кроме того, системная шина имеет ограниченную (хоть и высокую) пропускную способность (ПС) и ограниченное число слотов. Все это очевидно препятствует увеличению производительности при увеличении числа процессоров и числа подключаемых пользователей. В реальных системах можно задействовать не более 32 процессоров. Для построения масштабируемых систем на базе SMP используются кластерные или NUMA-архитектуры. При работе с SMP-системами используют так называемую парадигму программирования с разделяемой памятью (shared memory paradigm).

Гибридная архитектура с неоднородным доступом к памяти – **NUMA** (Non-Uniform Memory Access Architecture):



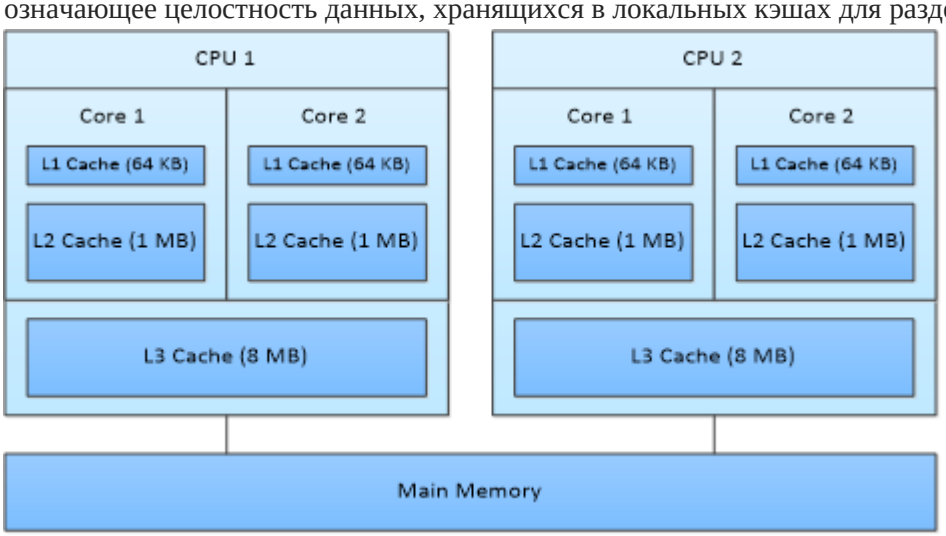
Главная особенность гибридной архитектуры NUMA – неоднородный доступ к памяти.

Гибридная архитектура совмещает достоинства систем с общей памятью и относительную дешевизну систем с раздельной памятью. **Суть** этой архитектуры – в особой организации памяти, а именно: *память физически распределена по различным частям системы, но логически она является общей*, так что пользователь видит единое адресное пространство. Система построена из однородных базовых модулей (плат), состоящих из небольшого числа процессоров и блока памяти. Модули объединены с помощью высокоскоростного коммутатора. Поддерживается единое адресное пространство, аппаратно поддерживается доступ к удаленной памяти, т.е. к памяти других модулей. При этом доступ к локальной памяти осуществляется в несколько раз быстрее, чем к удаленной. По существу, архитектура NUMA является MPP (массивно-параллельной) архитектурой, где в качестве отдельных вычислительных элементов берутся SMP узлы. Доступ к памяти и обмен данными внутри одного SMP-узла осуществляется через локальную память узла и происходит очень быстро, а к процессорам другого SMP-узла тоже есть доступ, но более медленный и через более сложную систему адресации.

Многоуровневое кэширование памяти в современных процессорах.

Специфика конструирования современных ядер процессоров привела к тому, что систему кэширования в подавляющем большинстве процессоров приходится делать многоуровневой.

- Уровни кэш-памяти – L1, L2, L3
- Кэш-память разных процессоров может быть когерентной и некогерентной;(свойство кэшей, означающее целостность данных, хранящихся в локальных кэшах для разделяемого ресурса)



Кэш первого уровня в каждом ядре (самый «близкий» к ядру) традиционно разделяется на две (как правило, равные) части: *кэш команд (L1K) и кэш данных (L1D)*. Это разделение предусматривается так

называемой «гарвардской структурой» ядер процессора, которая по состоянию на сегодня является самой популярной для построения ядер современных процессоров различного назначения. В кэш-памяти команд первого уровня L1K, соответственно, располагаются только команды (с ним работает подсистема подготовки команд для выполнения), а в кэш-памяти данных первого уровня L1D — только данные (они впоследствии, как правило, попадают во внутренние регистры процессора).

Иерархически «над» кэш-памятью первого уровня L1 стоит **кэш-память второго уровня** L2. Он, как правило, в 8 раз больше по объёму, примерно втрое медленнее, и является уже «смешанной» — там располагаются и команды, и данные. В первых многоядерных процессорах у каждого ядра были свои кэш-памяти L1K и L1D, но общая кэш-память L2. В современных ядрах процессоров у каждого ядра есть своя кэш-память второго уровня L2.

Общей для всех ядер процессора является **кэш-память третьего уровня** L3, которая в 4-8 раз больше, чем кэш-память L2 (в расчете на одно ядро), и ещё примерно втрое медленнее (но всё ещё быстрее оперативной памяти).

Алгоритм работы с многоуровневой кэш-памятью в общих чертах не отличается от алгоритма работы с одноуровневой кэш-памятью, но добавляются дополнительные итерации. Сначала информация ищется в кэш-памяти L1, если там промах — ищется в кэш-памяти L2, если снова промах — ищется в кэш-памяти L3, и уже потом, если ни на одном уровне кэш-памяти она не найдена — идёт обращение в оперативную память.

В большинстве ядер современных процессоров кэш-памяти L1 и L2 работают на той же частоте, что и процессорное ядро.

Ширина связей между ядром и кэш-памятью первого уровня, а также самими кэш-памятями может быть до 256 разрядов.

Как правило, кэш-памяти имеют наборно ассоциативную структуру, со степенью ассоциативности от 4 до 16.

Если кэш-память не является когерентной, в многопоточных приложениях может происходить **перестановка операций чтения и записи**. Также перестановки могут возникать благодаря оптимизациям компилятора.

Рассмотрим пример – переменная модифицируется внутри блокировки (критической секции), затем блокировка снимается:

```
LOAD    [&value], %o0  // Загрузить значение переменной в регистр
ADD     %o0, 1, %o0    // Увеличить на единицу
STORE   %o0, [&value]  // Записать в переменную
STORE   0, [&lock]     // Отпустить блокировку
```

Важно, чтобы запись в переменную выполнилась до того, как выполнится запись, отпускающая блокировку. На архитектурах с ослабленной моделью памяти (weak memory ordering) другой процессор может увидеть отпущенную блокировку до того, как увидит новое значение переменной. Возможна ситуация, когда другой процессор захватит блокировку и увидит старое значение переменной.

Существуют следующие **решения** данной проблемы:

- Синхронизация кэшей – cache coherence;
- Барьеры памяти - memory barrier (memory fence);

Для вышеприведенного примера решение с помощью барьера памяти выглядит так:

```
LOAD    [&value], %o0  // Загрузить значение переменной в регистр
ADD     %o0, 1, %o0    // Увеличить на единицу
STORE   %o0, [&value]  // Записать в переменную
MEMORYBARRIER          // Разделить операции барьером памяти
STORE   0, [&lock]     // Отпустить блокировку
```

В процессорах x86 и SPARC применяется **строгая модель памяти** (strong memory ordering), а именно, модель со строгим порядком записи – total store ordering (TSO):

- Чтения упорядочиваются в соответствии с предыдущими чтениями;
- Записи упорядочиваются в соответствии с предыдущими чтениями и записями;
- Это означает, что чтения могут «проглядеть» предыдущие записи, но не могут проглядеть предыдущие чтения, а записи не могут «проглядеть» предыдущие чтения и записи.

20. Средства распараллеливания вычислений в ОС Windows. Понятия процесса и потока. Достоинства и недостатки процессов и потоков. Создание и завершение процесса. Запуск процессов по цепочке

Многозада́чность — свойство операционной системы или среды программирования обеспечивать возможность параллельной (или псевдопараллельной) обработки нескольких поцессов.

Существует 2 типа многозадачности:

1. Процессная многозадачность
2. Поточная многозадачность

Процесс (process)

1. Процесс – выполняемая программа, которая имеет свое виртуальное адресное пространство, выполняемый код, указатели на объекты ядра, данные, уникальный идентификатор, как минимум один выполняющий поток.
2. Надежное средство. Аварийное завершение процесса не приводит к утечке ресурсов или нарушению целостности данных в других процессах.
3. Менее эффективное средство. Поскольку процессы работают в разных адресных пространствах, необходимо использовать средства Inter-Process Communication (IPC) для доступа к общим данным.

Поток/Нить (thread)

1. Поток – выполняемая подпрограмма процесса, разделяющая с другими потоками общие ресурсы процесса.
2. Эффективное средство. Расход памяти при распараллеливании минимален. Основные расходы памяти связаны с организацией стека на каждый параллельный поток. Производительность при работе с общими данными максимальна, поскольку потоки работают в общем адресном пространстве.
3. Менее надежное средство. Аварийное завершение потока часто приводит к утечке памяти процесса или даже к аварийному завершению процесса. Из-за общей памяти, целостность общих данных может быть нарушена.

Процесс

Описатель экземпляра процесса Любому EXE- или DLL-модулю, загружаемому в адресное пространство процесса, присваивается уникальный описатель экземпляра. Описатель экземпляра Вашего EXEфайла передается как первый параметр функции (w)WinMain — hinstExe (HINSTANCE). Базовый адрес, по которому загружается приложение, определяется компоновщиком. Компоновщик Visual C++ использует по умолчанию базовый адрес 0x00400000 — самый нижний в Windows 98, начиная с которого в ней допускается загрузка образа исполняемого файла. Указав параметр /BASE: адрес (в случае компоновщика от Microsoft), можно изменить базовый адрес, по которому будет загружаться приложение.

Функция GetModuleHandle:

```
HMODULE GetModuleHandle(
PCTSTR pszModule);
```

возвращает описатель/базовый адрес, указывающий, куда именно (в адресном пространстве процесса) загружается EXE- или DLL-файл. Если адрес не найден, возвращает NULL.

При попытке загрузить исполняемый файл в Windows 98 по базовому адресу ниже 0x00400000 загрузчик переместит его на другой адрес.

Есть еще две важные вещи, касающиеся GetModuleHandle. Во-первых, она проверяет адресное пространство только того процесса, который ее вызвал. Если этот процесс не использует никаких функций, связанных со стандартными диалоговыми окнами, то, вызвав GetModuleHandle и передав ей аргумент «ComDlg32», Вы получите NULL — пусть даже модуль ComDlg32.dll и загружен в адресное пространство какогонибудь другого процесса. Во-вторых, вызов этой функции и передача ей NULL дает в результате базовый адрес EXE-файла в адресном пространстве процесса. Так что, вызывая функцию в виде GetModuleHandle(NULL) — даже из кода в DLL, — Вы получаете базовый адрес EXE-, а не DLL-файла.

Командная строка При создании новому процессу передается командная строка, которая почти никогда не бывает пустой — как минимум, она содержит имя исполняемого файла, использованного при создании этого процесса. Однако, как Вы увидите ниже (при обсуждении функции CreateProcess), возможны случаи, когда процесс получает командную строку, состоящую из единственного символа — нуля, завершающего строку. В момент запуска приложения стартовый код из библиотеки C/C++ считывает командную строку процесса, пропускает имя исполняемого файла и заносит в параметр pszCmdLine функции (w)WinMain указатель на оставшуюся часть командной строки. Параметр pszCmdLine всегда указывает на ANSI-строку. Но, заменив WinMain на wWinMain, Вы получите доступ к Unicode-версии командной строки для своего процесса.

Указатель на полную командную строку процесса можно получить и вызовом функции GetCommandLine:

PTSTR GetCommandLine();

Она возвращает указатель на буфер, содержащий полную командную строку, включая полное имя (вместе с путем) исполняемого файла.

Во многих приложениях безусловно удобнее использовать командную строку, предварительно разбитую на отдельные компоненты, доступ к которым приложение может получить через глобальные переменные __argc и __argv (или __wargv). Функция CommandLineToArgvW расщепляет Unicode-строку на отдельные компоненты:

PWSTR CommandLineToArgvW(
PWSTR pszCmdLine,
int pNumArgs);

Переменные окружения

С любым процессом связан блок переменных окружения — область памяти, выделенная в адресном пространстве процесса. Каждый блок содержит группу строк такого вида:

VarName1=VarValue1\0
VarName2=VarValue2\0
VarName3=VarValue3\0
M
VarNameX=VarValueX\0
\0

Первая часть каждой строки — имя переменной окружения. За ним следует знак равенства и значение, присваиваемое переменной. Строки в блоке переменных окружения должны быть отсортированы в алфавитном порядке по именам переменных. Знак равенства разделяет имя переменной и ее значение, так что его нельзя использовать как символ в имени переменной. Важную роль играют и пробелы. Например, объявив две переменные:

XYZ= Windows (обратите внимание на пробел за знаком равенства)
ABC=Windows

и сравнив значения переменных XYZ и ABC, Вы увидите, что система их различает, — она учитывает любой пробел, поставленный перед знаком равенства или после него.

Вот что будет, если записать, скажем, так:

XYZ =Home (обратите внимание на пробел перед знаком равенства)
XYZ=Work

Вы получите первую переменную с именем «XYZ», содержащую строку «Home», и вторую переменную «XYZ», содержащую строку «Work».

Конец блока переменных окружения помечается дополнительным нулевым символом.

Обычно дочерний процесс наследует набор переменных окружения от родительского. Однако последний способен управлять тем, какие переменные окружения наследуются дочерним процессом, а какие — нет.

Под наследованием я имею в виду, что дочерний процесс получает свою копию блока переменных окружения от родительского, а не то, что дочерний и родительский процессы совместно используют один и тот же блок. Так что дочерний процесс может добавлять, удалять или модифицировать переменные в своем блоке, и эти изменения не затронут блок, принадлежащий родительскому процессу.

Переменные окружения обычно применяются для тонкой настройки приложения. Пользователь создает и инициализирует переменную окружения, затем запускает приложение, и оно, обнаружив эту переменную, проверяет ее значение и соответствующим образом настраивается.

GetEnvironmentVariable позволяет выявлять присутствие той или иной переменной окружения и определять ее значение:
DWORD GetEnvironmentVariable(
PCTSTR pszName,
PTSTR pszValue,
DWORD cchValue);
При вызове GetEnvironmentVariable параметр pszName должен указывать на имя интересующей Вас переменной, pszValue — на буфер, в который будет помещено значение переменной, а в cchValue следует сообщить размер буфера в символах. Функция возвращает либо количество символов, скопированных в буфер, либо 0, если ей не удалось обнаружить переменную окружения с таким именем.
Кстати, в реестре многие строки содержат подставляемые части, например:
%USERPROFILE%\My Documents - Часть, заключенная в знаки процента, является подставляемой. В данном случае в строку должно быть подставлено значение переменной окружения USERPROFILE.
Поскольку такие подстановки делаются очень часто, в Windows есть функция
ExpandEnvironmentStrings:
DWORD ExpandEnvironmentStrings(
PCTSTR pszSrc,
PTSTR pszDst,
DWORD nSize);
Параметр pszSrc принимает адрес строки, содержащей подставляемые части, а параметр pszDst — адрес буфера, в который записывается развернутая строка. Параметр nSize определяет максимальный размер буфера в символах.
Наконец, функция SetEnvironmentVariable позволяет добавлять, удалять и модифицировать значение переменной:
DWORD SetEnvironmentVariable(
PCTSTR pszName,
PCTSTR pszValue);
Она устанавливает ту переменную, на чье имя указывает параметр pszName, и присваивает ей значение, заданное параметром pszValue. Если такая переменная ужесуществует, функция модифицирует ее значение. Если же в pszValue содержится NULL, переменная удаляется из блока.
Для манипуляций с блоком переменных окружения всегда используйте именно эти функции. Строки в блоке переменных нужно отсортировать в алфавитном порядке по именам переменных (тогда GetEnvironmentVariable быстрее находит нужные переменные), а SetEnvironmentVariable как раз и следит за порядком расположения переменных.

Привязка к процессорам

Обычно потоки внутри процесса могут выполняться на любом процессоре компьютера. Однако их можно закрепить за определенным подмножеством процессоров из числа имеющихся на компьютере. Это свойство называется привязкой к процессорам (processor affinity). По умолчанию Windows 2000 использует нежесткую привязку (soft affinity) потоков к процессорам. Это означает, что при прочих равных условиях, система пытается выполнять поток на том же процессоре, на котором он работал в последний раз. При таком подходе можно повторно использовать данные, все еще хранящиеся в кэше процессора. Дочерние процессы наследуют привязку к процессорам от родительских.

Режим обработки ошибок

С каждым процессом связан набор флагов, сообщающих системе, каким образом процесс должен реагировать на серьезные ошибки: повреждения дисковых носителей, необрабатываемые исключения, ошибки операций поиска файлов и неверное выравнивание данных. Процесс может указать системе, как обрабатывать каждую из этих ошибок, через функцию SetErrorMode:
UINT SetErrorMode(UINT fuErrorMode);
Параметр fuErrorMode — это набор флагов, комбинируемых побитовой операцией OR:
Флаги:

· *SEM_FAILCRITICALERRORS* Система не выводит окно с сообщением от обработчика критических ошибок и возвращает ошибку в вызывающий процесс

· *SEM_NOGPFAULTERRORBOX* Система не выводит окно с сообщением о нарушении общей защиты; этим флагом манипулируют только отладчики, самостоятельно обрабатывающие нарушения общей защиты с помощью обработчика исключений

· *SEM_NOOPENFILEERRORBOX* Система не выводит окно с сообщением об отсутствии искомого файла

· *SEM_NOALIGNMENTFAULTEXCEPT* Система автоматически исправляет нарушения в выравнивании данных, и они становятся невидимы приложению; этот флаг не действует на процессорах x86

По умолчанию дочерний процесс наследует от родительского флаги, указывающие на режим обработки ошибок. Иначе говоря, если у процесса в данный момент установлен флаг SEM_NOGPFAULTERRORBOX и он порождает другой процесс, этот флаг будет установлен и у дочернего процесса. Однако «наследник» об этом не уведомляется, и он вообще может быть не рассчитан на обработку ошибок такого типа (в данном случае — нарушений общей защиты). В результате, если в одном из потоков дочернего процесса все-таки произойдет подобная ошибка, этот процесс может завершиться, ничего не сообщив пользователю. Но родительский процесс способен предотвратить наследование дочерним процессом своего режима обработки ошибок,

указав при вызове функции CreateProcess флаг CREATE_DEFAULT_ERROR_MODE.

Текущие диск и каталог для процесса

Текущий каталог текущего диска — то место, где Windows-функции ищут файлы и подкаталоги, если полные пути в соответствующих параметрах не указаны. Например, если поток в процессе вызывает функцию CreateFile, чтобы открыть какой-нибудь файл, а полный путь не задан, система просматривает список файлов в текущем каталоге текущего диска. Этот каталог отслеживается самой системой, и, поскольку такая информация относится ко всему процессу, смена текущего диска или каталога одним из потоков распространяется и на остальные потоки в данном процессе.

Поток может получать и устанавливать текущие каталог и диск для процесса с помощью двух функций:

DWORD GetCurrentDirectory(
DWORD cchCurDir,
PTSTR pszCurDir);
BOOL SetCurrentDirectory(PCTSTR pszCurDir);

Текущие каталоги для процесса

Система отслеживает текущие диск и каталог для процесса, но не текущие каталоги на каждом диске. Однако в операционной системе предусмотрен кое-какой сервис для манипуляций с текущими каталогами на разных дисках. Он реализуется через переменные окружения конкретного процесса. Например:

=C:=C:\Utility\Bin
=D:=D:\Program Files

Эти переменные указывают, что текущим каталогом на диске C является \Utility\-Bin, а на диске D — Program Files.

Если Вы вызываете функцию, передавая ей путь с именем диска, отличного от текущего, система сначала просматривает блок переменных окружения и пытается найти переменную, связанную с именем указанного диска. Если таковая есть, система выбирает текущий каталог на заданном диске в соответствии с ее значением, нет —текущим каталогом считается корневой.

Структуры данных для процессов и потоков

В WRK за *управление процессами* отвечает *диспетчер* процессов (base\ntos\ps), а многие важные структуры данных описаны в заголовочных файлах base\ntos\inc\ps.h и base\ntos\inc\ke.h.

Процесс в *Windows* описывается структурой данных EPROCESS [5]. Эта структура в WRK содержится в файле base\ntos\inc\ps.h (строка 238). Рассмотрим некоторые её поля.

· Pcb (Process Control Block – блок управления процессом) – представляет собой структуру KPROCESS, хранящую данные, необходимые для планирования потоков, в том числе указатель на список потоков процесса (файл base\ntos\inc\ke.h, строка 944).

· CreateTime и ExitTime – время создания и завершения процесса.

· UniqueProcessId – уникальный идентификатор процесса.

· ActiveProcessLinks – элемент двунаправленного списка (тип LIST_ENTRY), содержащего активные процессы.

· QuotaUsage, QuotaPeak, CommitCharge – квоты (ограничения) на используемую память.

· ObjectTable – таблица дескрипторов процесса.

· Token – маркер доступа.

· ImageFileName – имя исполняемого файла.

· ThreadListHead – двунаправленный список потоков процесса.

· Peb (Process Environment Block – блок переменных окружения процесса) – информация об образе исполняемого файла (файл public\sdk\inc\pebteb.h, строка 75).

· PriorityClass – класс приоритета процесса (см. лекцию 9 "Планирование потоков").

Структура для потока в *Windows* называется ETHREAD и описывается в файле base\ntos\inc\ps.h (строка 545). Её основные поля следующие:

· Tcb (Thread Control Block – блок управления потоком) – поле, которое является структурой типа KTHREAD (файл base\ntos\inc\ke.h, строка 1069) и необходимо для планирования потоков.

· CreateTime и ExitTime – время создания и завершения потока.

· Cid – структура типа CLIENT_ID, включающая два поля – идентификатор процесса-владельца данного потока и идентификатор самого потока.

· ThreadsProcess – указатель на структуру EPROCESS процесса-владельца.

· StartAddress – адрес системной стартовой функции потока. При создании потока сначала вызывается системная стартовая функция, которая запускает пользовательскую стартовую функцию.

· Win32StartAddress – адрес пользовательской стартовой функции.

Создание процессов

Процесс подсистемы Windows создается, когда приложение вызывает одну из функций создания процесса или каким-то образом попадает в нее. К таким функциям относятся CreateProcess, CreateProcessAsUser, CreateProcessWithTokenW и CreateProcessWithLogonW. Создание процесса Windows состоит из нескольких этапов, выполняемых тремя частями операционной системы: Windows библиотекой Kernel32.dll, работающей на стороне клиента (при выполнении процедур CreateProcessAsUser, CreateProcessWithTokenW и CreateProcessWithLogonW
часть этой работы сначала делается библиотекой Advapi32.dll), исполняющей системой Windows и процессом подсистемы Windows (Csrss).
Основные этапы создания процесса с помощью Windows-функции CreateProcess:
Создание процессов в *Windows* включает 7 этапов:

1. Проверка и преобразование параметров.

Параметры функции CreateProcess проверяются на *корректность* и преобразуются к внутреннему формату системы. Проверка приемлемости параметров; преобразование флагов и настроек подсистемы Windows в их более подходящие аналоги; анализ, проверка приемлемости и преобразование списка атрибутов в его более подходящий аналог.

2. Открытие исполняемого файла.

Параметр pszCommandLine позволяет указать полную командную строку, используемую функцией CreateProcess при создании нового процесса. Разбирая эту строку,функция полагает, что первый компонент в ней представляет собой имя исполняемого файла, который Вы хотите запустить. Если в имени этого файла не указано расширение, она считает его EXE. Далее функция приступает к поиску заданного файла и делает это в следующем порядке:
1. Каталог, содержащий EXE-файл вызывающего процесса.
2. Текущий каталог вызывающего процесса.
3. Системный каталог Windows.

4. Основной каталог Windows.
5. Каталоги, перечисленные в переменной окружения PATH.
Конечно, если в имени файла указан полный путь доступа, система сразу обращается туда и не просматривает эти каталоги. Найдя нужный исполняемый файл, она создает новый процесс и проецирует код и данные исполняемого файла на адресное пространство этого процесса.
Происходит *поиск* файла, который содержит запускаемую программу. Обычно это *файл* с расширением .EXE, но могут быть также расширения *.COM*, *.PIF*, .BAT, *.CMD*. Определяется тип исполняемого файла:

· Windows приложение (.EXE) – продолжается нормальное создание процесса;

· приложение MS-DOS или Win16 (.EXE, .COM, .PIF) – запускается образ поддержки Ntvdm.exe;
· командный файл (.BAT, .CMD) – запускается образ поддержки Cmd.exe;
· приложение POSIX – запускается образ поддержки Posix.exe.

3. Создание объекта "Процесс".

Формируются структуры данных EPROCESS, KPROCESS, PEB, инициализируется *адресное пространство* процесса. Для этого вызывается системная *функция* NtCreateProcess (*файл* base\ntos\ps\create.c, строка 826), которая затем вызывает функцию NtCreateProcessEx (тот же *файл*, строка 884), а та, в свою *очередь*, функцию PspCreateProcess (тот же *файл*, строка 1016).

Замечание. Начиная с *Windows Vista* при создании процесса вызов нескольких функций (NtCreateProcess, NtWriteVirtualMemory, NtCreateThread) заменен вызовом одной функции NtCreateUserProcess.

Рассмотрим некоторые важные действия, выполняемые функцией PspCreateProcess.

· Если в параметрах функции PspCreateProcess указан процесс-родитель:
o по его дескриптору определяется указатель на объект EPROCESS (функция ObReferenceObjectByHandle, строка 1076);
o наследуется от процесса родителя маска привязки к процессорам (Affinity, строка 1092).
· Устанавливается минимальный и максимальный размеры рабочего набора (WorkingSetMinimum = 20 МБ и WorkingSetMaximum = 45 МБ, строки 1093 и 1094, см. лекцию 11 "Управление памятью").
· Создается объект "Процесс" (структура EPROCESS) при помощи функции ObCreateObject (строка 1108).
· Инициализируется двунаправленный список потоков при помощи функции InitializeListHead (строка 1134).
· Копируется таблица дескрипторов родительского процесса (строка 1270).
· Создается структура KPROCESS при помощи функции KeInitializeProcess (строка 1289).
· Маркер доступа и другие данные, связанные с безопасностью (см. лекцию 13 "Безопасность", копируются из родительского процесса (функция PspInitializeProcessSecurity, строка 1302).
· Устанавливается приоритет процесса, равный Normal; однако, если приоритет родительского процесса был Idle или Below Normal, то данный приоритет наследуется (строки 1307–1312, см. лекцию 9 "Планирование потоков").
· Инициализируется адресное пространство процесса (строки 1337–1476).
· Генерируется уникальный идентификатор процесса (функция ExCreateHandle) и сохраняется в поле UniqueProcessId структурыEPROCESS (строки 1482–1488).
· Создается блок PEB и записывается в соответствующее поле структуры EPROCESS (строки 1550–1607).
· Созданный объект вставляется в хвост двунаправленного списка всех процессов (строки 1613–1615) и в таблицу дескрипторов (строки 1639–1644). Первая вставка обеспечивает доступ к процессу по имени, вторая – по ID.

· Определяется время создания процесса (функция KeQuerySystemTime) и записывается в поле CreateTime структуры EPROCESS(строка 1733).

4. Создание основного потока.

Формируется *структура данных* ETHREAD, *стек* и *контекст* потока, генерируется *идентификатор* потока. *Поток* создается при помощи функции NtCreateThread, определенной в файле base\ntos\ps\create.c, (строка 117), которая вызывает функцию PspCreateThread (тот же*файл*, строка 295). При этом выполняются следующие действия:

· создается объект ETHREAD (строка 370).

· Заполняются поля структуры ETHREAD, связанные с процессом-владельцем, – указатель на структуру EPROCESS (ThreadsProcess) и идентификатор процесса (Cid.UniqueProcess) (строки 396 и 398).

· Генерируется уникальный идентификатор потока (функция ExCreateHandle) и сохраняется в поле Cid.UniqueThread структурыEPROCESS (строки 400–402).

· Заполняются стартовые адреса потока, системный (StartAddress) и пользовательский (Win32StartAddress) (строки 468-476).

· Инициализируются поля структуры KTHREAD при помощи вызова функции KeInitThread (строки 490–498 для потока пользовательского режима и 514–522 для потока режима ядра).

· Функция KeStartThread заполняет остальные поля структуры ETHREAD и вставляет поток в список потоков процесса (строка 564).

· Если при вызове функции PspCreateThread установлен флаг CreateSuspended ("Приостановлен") поток переводится в состояние ожидания (функция KeSuspendThread, строка 660); иначе вызывается функция KeReadyThread (строка 809), которая ставит поток в очередь готовых к выполнению потоков (см. лекцию 9 "Планирование потоков").

5. Уведомление подсистемы Windows.

Подсистеме *Windows* отправляется сообщение о вновь созданных процессе и его основном потоке, в которое входят их дескрипторы, идентификаторы и другая *информация*. Подсистема *Windows* добавляет новый процесс в общий *список* всех процессов и готовится к запуску основного потока.

6. Запуск основного потока.

Основной *поток* стартует, но начинают выполняться системные функции, завершающие *создание процесса* – осуществляется его*инициализация потока (если только не был установлен флаг создания приостановленного потока — CREATE_SUSPENDED)*.

7. Инициализация процесса.

Завершение в контексте нового процесса и потока инициализации адресного пространства (например, загрузка требуемых DLL-библиотек) и начало выполнения программы

· Проверяется, не запущен ли процесс в отладочном режиме;

· проверяется, следует ли производить предвыборку блоков памяти (тех участков памяти, которые при прошлом запуске использовались в течение первых 10 секунд работы процесса);

· инициализируются необходимые компоненты и структуры данных процесса, например, диспетчер кучи;

· загружаются динамически подключаемые библиотеки (DLL – Dynamic Link Library);

· начинается выполнение стартовой функции потока.

Если класс приоритета для нового процесса не указан, по умолчанию для него устанавливается класс приоритета обычный — Normal, если только класс приоритета процесса, который его создал, не был Idle (Простой) или Below Normal (ниже обычного), в таком случае класс приоритета нового процесса будет таким же, как и у его создателя. Если для нового процесса указан класс приоритета Real-time (выполнения в режиме

реального времени) и код, вызвавший процесс, не имеет привилегии на повышение приоритета при планировании — Increase Scheduling Priority, — вместо него используется класс приоритета High (высокий).
Когда поток в приложении вызывает CreateProcess, система создает объект ядра «процесс» с начальным значением счетчика числа его пользователей, равным 1. Это объект — не сам процесс, а компактная структура данных, через которую операционная система управляет процессом. (Объект ядра «процесс» следует рассматривать как структуру данных со статистической информацией о процессе.) Затем система создает для нового процесса виртуальное адресное пространство и загружает в него код и данные как для исполняемого файла, так и для любых DLL (если таковые требуются). Далее система формирует объект ядра «поток» (со счетчиком, равным 1) для первичного потока нового процесса. Как и в первом случае, объект ядра «поток» — это
компактная структура данных, через которую система управляет потоком. Первичный поток начинает с исполнения стартового кода из библиотеки C/C++, который в конечном счете вызывает функцию WinMain, wWinMain, main или wmain в Вашей программе. Если системе удастся создать новый процесс и его первичный поток, CreateProcess вернет TRUE.
BOOL CreateProcess(
PCTSTR pszApplicationName,
PTSTR pszCommandLine,
PSECURITY_ATTRIBUTES psaProcess,
PSECURITY_ATTRIBUTES psaThread,
BOOL bInheritHandles,
DWORD fdwCreate,
PVOID pvEnvironment,
PCTSTR pszCurDir,
PSTARTUPINFO psiStartInfo,
PPROCESS_INFORMATION ppiProcInfo);
pszApplicationName и pszCommandLine - имя исполняемого файла, которым будет пользоваться новый процесс, и командную строку, передаваемую этому процессу.
Обратите внимание на тип параметра pszCommandLine: PTSTR. Он означает, что CreateProcess ожидает передачи адреса строки, которая не является константой. Дело в том, что CreateProcess в процессе своего выполнения модифицирует переданную командную строку, но перед возвратом управления восстанавливает ее.
psaProcess, psaThread и bInheritHandles. Чтобы создать новый процесс, система должна сначала создать объекты ядра «процесс» и «поток» (для первичного потока процесса). Поскольку это объекты ядра, родительский процесс получает возможность связать с ними атрибуты защиты. Параметры psaProcess и psaThread позволяют определить нужные атрибуты защиты для объектов «процесс» и «поток» соответственно. В эти параметры можно занести NULL, и система закрепит за данными объектами дескрипторы защиты по умолчанию. В качестве альтернативы можно объявить и инициализировать две структуры SECURITY_ATTRIBUTES; тем самым Вы создадите и присвоите объектам «процесс» и «поток» свои атрибуты защиты.
Структуры SECURITY_ATTRIBUTES для параметров psaProcess и psaThread используются и для того, чтобы какой-либо из этих двух объектов получил статус наследуемого любым дочерним процессом.
Параметр *fdwCreate* определяет флаги, влияющие на то, как именно создается новый процесс. Параметр *fdwCreate* разрешает задать и класс приоритета процесса. Однако это необязательно и даже, как правило, не рекомендуется; система присваивает новому процессу класс приоритета по умолчанию.
Параметр *pvEnvironment* указывает на блок памяти, хранящий строки переменных окружения, которыми будет пользоваться новый процесс. Обычно вместо этого параметра передается NULL, в результате чего дочерний процесс наследует строки переменных окружения от родительского процесса.
pszCurDir позволяет родительскому процессу установить текущие диск и каталог для дочернего процесса. Если его значение — NULL, рабочий каталог нового процесса будет тем
же, что и у приложения, его породившего.
Параметр psiStartInfo указывает на структуру STARTUPINFO:
typedef struct _STARTUPINFO {

DWORD cb;
PSTR lpReserved;
PSTR lpDesktop;
PSTR lpTitle;
DWORD dwX;
DWORD dwY;
DWORD dwXSize;
DWORD dwYSize;
DWORD dwXCountChars;
DWORD dwYCountChars;
DWORD dwFillAttribute;
DWORD dwFlags;
WORD wShowWindow;
WORD cbReserved2;
PBYTE lpReserved2;
HANDLE hStdInput;
HANDLE hStdOutput;
HANDLE hStdError;
*} STARTUPINFO, *LPSTARTUPINFO;*

Разработчики приложений часто забывают о необходимости инициализации этой структуры. Если Вы не обнулите ее элементы, в них будет содержаться мусор, оставшийся в стеке вызывающего потока. Функция CreateProcess, получив такую структуру данных, либо создаст новый процесс, либо нет — все зависит от того, что именно окажется в этом мусоре.

Параметр *ppiProcInfo* указывает на структуру PROCESS_INFORMATION, которую Вы должны предварительно создать; ее элементы инициализируются самой функцией CreateProcess. Структура представляет собой следующее:

```
struct PROCESS_INFORMATION
{
 HANDLE hProcess; //описатель процесса
 HANDLE hThread; //описатель потока в пределах текущего процесса
 DWORD dwProcessId; //уникальный id процесса в пределах системы
 DWORD dwThreadId; //уникальный id потока в пределах системы
};
```

Как я уже говорил, создание нового процесса влечет за собой создание объектов ядра «процесс» и «поток». В момент с оздания система присваивает счетчику каждого объекта начальное значение — единицу. Далее функция CreateProcess (перед самым возвратом управления) открывает объекты «процесс» и «поток» и заносит их описатели, специфичные для данного процесса, в элементы hProcess и hThread структуры PROCESS_INFORMATION. Когда CreateProcess открывает эти объекты, счетчики каждого из них увеличиваются до 2.

Завершение процесса

Процесс можно завершить четырьмя способами:

· входная функция первичного потока возвращает управление (рекомендуемый способ); единственный способ, гарантирующий корректную очистку всех ресурсов, принадлежавших первичному потоку. При этом:

1. любые C++-объекты, созданные данным потоком, уничтожаются соответствующими деструкторами;
2. система освобождает память, которую занимал стек потока;
3. система устанавливает код завершения процесса (поддерживаемый объектом ядра «процесс») — его и возвращает Ваша входная функция;
4. счетчик пользователей данного объекта ядра «процесс» уменьшается на 1.

· один из потоков процесса вызывает функцию VOID ExitProcess(UINT fuExitCode); (нежелательный способ); Эта функция завершает процесс (после завершения всех его потомков) и заносит в параметр fuExitCode код завершения процесса.

· поток другого процесса вызывает функцию TerminateProcess (тоже нежелательно);

BOOL TerminateProcess(
HANDLE hProcess,
UINT fuExitCode);

Главное отличие этой функции от ExitProcess в том, что ее может вызвать любой поток и завершить любой процесс. Параметр hProcess идентифицирует описатель завершаемого процесса, а в параметре fuExitCode возвращается код завершения процесса. Процесс не получает абсолютно никаких уведомлений о том, что он завершается, и приложение не может ни выполнить очистку, ни предотвратить свое неожиданное завершение (если оно, конечно, не использует механизмы защиты). При этом теряются все данные, которые процесс не успел переписать из памяти на диск. TerminateProcess — функция асинхронная, т. е. она сообщает системе, что Вы хотите завершить процесс, но к тому времени, когда она вернет управление, процесс может быть еще не уничтожен. Так что, если Вам нужно точно знать момент завершения процесса, используйте WaitForSingleObject или аналогичную функцию, передав ей описатель этого процесса.

· все потоки процесса умирают по своей воле (большая редкость). В такой ситуации (а она может возникнуть, если все потоки вызвали ExitThread или их закрыли вызовом TerminateThread) операционная система больше не считает нужным «содержать» адресное пространство данного процесса. Обнаружив, что в процессе не исполняется ни один поток, она немедленно завершает его. При этом код завершения процесса приравнивается коду завершения последнего потока.

Что происходит при завершении процесса:

1. Выполнение всех потоков в процессе прекращается.
2. Все User- и GDI-объекты, созданные процессом, уничтожаются, а объекты ядра закрываются (если их не использует другой процесс).
3. Код завершения процесса меняется со значения STILL_ACTIVE на код, переданный в ExitProcess или TerminateProcess.
4. Объект ядра «процесс» переходит в свободное, или незанятое (signaled), состояние. Прочие потоки в системе могут приостановить свое выполнение вплоть до завершения данного процесса.
5. Счетчик объекта ядра «процесс» уменьшается на 1.

Запуск процесса по цепочке:

```
CreateProcess(…, &pi); // PROCESS_INFORMATION pi;
CloseHandle(pi.hThread);
WaitForSingleObject(pi.hProcess);
GetExitCodeProcess(pi.hProcess, &exitCode); // DWORD exitCode;
CloseHandle(pi.hProcess);
```

21. Средства распараллеливания вычислений в ОС Windows. Понятия процесса и потока. Создание и завершение потока. Приостановка и возобновление потока. Контекст потока.

Многозада́чность — свойство операционной системы или среды программирования обеспечивать возможность параллельной (или псевдопараллельной) обработки нескольких процессов.

Существует 2 типа многозадачности:

· Процессная многозадачность
· Поточная многозадачность

Процесс (process)

· Процесс – выполняемая программа, которая имеет свое виртуальное адресное пространство, выполняемый код, указатели на объекты ядра, данные, уникальный идентификатор, как минимум один выполняющий поток.

· Надежное средство. Аварийное завершение процесса не приводит к утечке ресурсов или нарушению целостности данных в других процессах.

· Менее эффективное средство. Поскольку процессы работают в разных адресных пространствах, необходимо использовать средства Inter-Process Communication (IPC) для доступа к общим данным.

Поток/Нить (thread)

· Поток – выполняемая подпрограмма процесса, разделяющая с другими потоками общие ресурсы процесса.

· Эффективное средство. Расход памяти при распараллеливании минимален. Основные расходы памяти связаны с организацией стека на каждый параллельный поток. Производительность при работе с общими данными максимальна, поскольку потоки работают в общем адресном пространстве.

· Менее надежное средство. Аварийное завершение потока часто приводит к утечке памяти процесса или даже к аварийному завершению процесса. Из-за общей памяти, целостность общих данных может быть нарушена.

Процесс фактически состоит из двух компонентов: объекта ядра «процесс» и адресного пространства. Так вот, любой поток тоже состоит из двух компонентов:

o объекта ядра, через который операционная система управляет потоком. Там же

o хранится статистическая информация о потоке;

стека потока, который содержит параметры всех функций и локальные переменные, необходимые потоку для выполнения кода.

Процесс ничего не исполняет, он просто служит контейнером потоков. Потоки всегда создаются в контексте какого-либо процесса, и вся их жизнь проходит только в его границах. На практике это означает, что потоки исполняют код и манипулируют данными в адресном пространстве процесса.

Как видите, процессы используют куда больше системных ресурсов, чем потоки. Причина кроется в адресном пространстве. Создание виртуального адресного пространства для процесса требует значительных системных ресурсов. При этом ведется масса всяческой статистики, на что уходит немало памяти. В адресное пространство загружаются EXE- и DLL-файлы, а значит, нужны файловые ресурсы. С другой стороны, потоку требуются лишь соответствующий объект ядра и стек; объем статистических сведений о потоке невелик и много памяти не занимает.

Поток (thread) определяет последовательность исполнения кода в процессе. При инициализации процесса система всегда создает первичный поток. Начинаясь со стартового кода из библиотеки C/C++, который в свою очередь вызывает входную функцию (WinMain, wWinMain, main или wmain) из Вашей программы, он живет до того момента, когда входная функция возвращает управление стартовому коду и тот вызывает функцию ExitProcess.

Многопоточное приложение легче масштабируется. Каждый поток можно закрепить за определенным процессором.

Обычно в приложении существует один поток, отвечающий за поддержку пользовательского интерфейса, — он создает все окна и содержит цикл GetMessage. Любые другие потоки в процессе являются рабочими (т. е. отвечают за вычисления, вводвывод и другие операции) и не создают никаких окон. Поток пользовательского интерфейса, как правило, имеет более высокий приоритет, чем рабочие потоки, — это нужно для того, чтобы он всегда быстро реагировал на действия пользователя.

Функции потоков:

· В отличие от входной функции первичного потока, у которой должно быть одно из четырех имен: main, wmain, WinMain или wWinMain, — функцию потока можно назвать как угодно. Однако, если в программе несколько функций потоков, Вы должны присвоить им разные имена, иначе компилятор или компоновщик решит, что Вы создаете несколько реализаций единственной функции.

· Поскольку входным функциям первичного потока передаются строковые параметры, они существуют в ANSI- и Unicode-версиях: main — wmain и WinMain — wWinMain. Но функциям потоков передается единственный параметр, смысл которого определяется Вами, а не операционной системой. Поэтому здесь нет проблем с ANSI/Unicode.

· Функция потока должна возвращать значение, которое будет использоваться как код завершения потока. Здесь полная аналогия с библиотекой C/C++: код завершения первичного потока становится кодом завершения процесса.

· Функции потоков (да и все Ваши функции) должны по мере возможности обходиться своими параметрами и локальными переменными. Так как к статической или глобальной переменной могут одновременно обратиться несколько потоков, есть риск повредить ее содержимое. Однако параметры и локальные переменные создаются в стеке потока, поэтому они в гораздо меньшей степени подвержены влиянию другого потока.

Создание потока. Функция **CreateThread**

Если Вы хотите создать дополнительные потоки, нужно вызвать из первичного потока функцию CreateThread:

HANDLE CreateThread(
PSECURITY_ATTRIBUTES psa;
DWORD cbStack;
PTHREAD_START_ROUTINE pfnStartAddr;
PVOID pvParam;
DWORD fdwCreate;
PDWORD pdwThreadID);

При каждом вызове этой функции система создает объект ядра «поток». Это не сам поток, а компактная структура данных, которая используется операционной системой для управления потоком и хранит статистическую информацию о потоке. Так что объект ядра «поток» — полный аналог объекта ядра «процесс».

Параметр ***psa*** является указателем на структуру SECURITY_ATTRIBUTES. Если Вы хотите, чтобы объекту ядра «поток» были присвоены атрибуты защиты по умолчанию (что чаще всего и бывает), передайте в этом параметре NULL. А чтобы дочерние процессы смогли наследовать описатель этого объекта, определите структуру SECURITY_ATTRIBUTES и инициализируйте ее элемент bInheritHandle значением TRUE.

Параметр ***cbStack*** определяет, какую часть адресного пространства поток сможет использовать под свой стек. Каждому потоку выделяется отдельный стек. Функция CreateProcess, запуская приложение, вызывает CreateThread, и та инициализирует первичный поток процесса. При этом CreateProcess заносит в параметр cbStack значение, хранящееся в самом исполняемом файле. Управлять этим значением позволяет ключ /STACK компоновщика:

/STACK:[reserve] [,commit]

Аргумент *reserve* определяет объем адресного пространства, который система должна зарезервировать под стек потока (по умолчанию — 1 Мб). Значение аргумента reserve устанавливает верхний предел для стека, и это ограничение позволяет прекращать деятельность функций с бесконечной рекурсией. Аргумент *commit* задает объем физической памяти, который изначально передается области, зарезервированной под стек (по умолчанию — 1 страница). По мере исполнения кода в потоке Вам, весьма вероятно, понадобится отвести под стек больше одной страницы памяти. При переполнении стека возникнет исключение. Перехватив это исключение, система передаст зарезервированному пространству еще одну страницу (или столько, сколько указано в аргументе commit). Такой механизм позволяет динамически увеличивать размер стека лишь по необходимости.

Параметр ***pfnStartAddr*** определяет адрес функции потока, с которой должен будет начать работу создаваемый поток, а параметр ***pvParam*** идентичен параметру pvParam функции потока. CreateThread лишь передает этот параметр по эстафете той функции, с которой начинается выполнение создаваемого потока. Таким образом, данный параметр позволяет передавать функции потока какое-либо инициализирующее значение. Оно может быть или просто числовым значением, или указателем на структуру данных с дополнительной информацией. Учтите, что Windows — операционная система с вытесняющей многозадачностью, а следовательно, новый поток и поток, вызвавший CreateThread, могут выполняться одновременно.

Параметр ***fdwCreate*** определяет дополнительные флаги, управляющие созданием потока.

Он принимает одно из двух значений: 0 (исполнение потока начинается немедленно) или CREATE_SUSPENDED. В последнем случае система создает поток, инициализирует его и приостанавливает до последующих указаний. Флаг CREATE_SUSPENDED позволяет программе изменить какие-либо свойства потока перед тем, как он начнет выполнять код. Правда, необходимость в этом возникает довольно редко.

Последний параметр функции CreateThread — это адрес переменной типа DWORD, в которой функция возвращает идентификатор, приписанный системой новому потоку.

Рождение потока

Жизненный цикл потока начинается тогда, когда программа создает новый поток.

Запрос перекочевывает исполняющей системе Windows, где диспетчер процесса выделяет пространство под объект потока и вызывает ядро для инициализации блока управления потока (KTHREAD). Действия из следующего списка предпринимаются внутри Windows-функции CreateThread в библиотеке Kernel32.dll для создания потока Windows:

1. Функция CreateThread преобразует параметры Windows API к более подходящим флагам и создает соответствующую структуру, описывающую параметры
объекта (OBJECT_ATTRIBUTES).
2. Функция CreateThread создает список атрибутов с двумя записями: идентификатором клиента — client ID и адресом TEB. Это позволяет функции CreateThread получить эти значения, как только поток будет создан.
3. Вызывается функция NtCreateThreadEx, предназначенная для создания контекста пользовательского режима, а также для исследования и сбора списка атрибутов. Затем она вызывает функцию PspCreateThread для создания приостановленного объекта потока исполняющей системы (см. раздел «Порядок работы функции CreateProcess», описания этапов 3 и 5, предыдущий вопрос).
4. Функция CreateThread выделяет контекст активации для потока, используемый поддержкой смежной сборки (side-by-side assembly). Затем она запрашивает стек активации, чтобы посмотреть, не требует ли он активации, и проводит активацию в случае необходимости. Указатель на стек активации сохраняется в TEB нового потока.
5. Функция CreateThread уведомляет подсистему Windows о новом потоке,
и подсистема выполняет ряд настроечной работы для нового потока.
6. Вызвавшему коду возвращается дескриптор и идентификатор потока, сгенерированные на этапе 3.
7. За исключением того случая, когда вызывающий код создал поток с установленным флагом создания приостановленного потока —CREATE_SUSPENDED, теперь
поток возобновляется и может быть запланирован для выполнения. Когда поток
начинает работать, он перед вызовом кода по фактическому стартовому адресу,
указанному пользователем, выполняет действия, описанные ранее в разделе
«Этап 7. Выполнение инициализации процесса в контексте нового процесса».

Завершение потока

Поток можно завершить четырьмя способами:

· функция потока возвращает управление (рекомендуемый способ); Функцию потока следует проектировать так, чтобы поток завершался только после того, как она возвращает управление. Это единственный способ, гарантирующий корректную очистку всех ресурсов, принадлежавших Вашему потоку. При этом:

1. любые C++-объекты, созданные данным потоком, уничтожаются соответствующими деструкторами;
2. система корректно освобождает память, которую занимал стек потока;
3. система устанавливает код завершения данного потока (поддерживаемый объектом ядра «поток») — его и возвращает Ваша функция потока;
4. счетчик пользователей данного объекта ядра «поток» уменьшается на 1.

· поток самоуничтожается вызовом функции VOID *ExitThread(DWORD dwExitCode);* (нежелательный способ); При этом освобождаются все ресурсы операционной системы, выделенные данному потоку, но C/C++-ресурсы (например, объекты, созданные из C++-классов) не очищаются. В параметр dwExitCode Вы помещаете значение, которое система рассматривает как код завершения потока.

· один из потоков данного или стороннего процесса вызывает функцию BOOL *TerminateThread(HANDLE hThread, DWORD dwExitCode);* (нежелательный способ); В отличие от ExitThread, которая уничтожает только вызывающий поток, эта функция завершает поток, указанный в параметре hThread. В параметр dwExitCode Вы помещаете значение, которое система рассматривает как код завершения потока. помещаете значение, которое система рассматривает как код завершения потока. После того как поток будет уничтожен, счетчик пользователей его объекта ядра «поток» уменьшится на 1. TerminateThread — функция асинхронная, т. е. она сообщает системе, что Вы хотите завершить поток, но к тому времени, когда она вернет управление, поток может быть еще не уничтожен. Так что, если Вам нужно точно знать момент завершения потока, используйте или аналогичную функцию, передав ей описатель этого потока. Корректно написанное приложение не должно вызывать эту функцию, поскольку поток не получает никакого уведомления о завершении; из-за этого он не может выполнить должную очистку ресурсов. Уничтожение потока при вызове ExitThread или возврате управления из функции потока приводит к разрушению его стека. Но если он завершен функцией TerminateThread, система не уничтожает стек, пока не завершится и процесс, которому принадлежал этот поток. Так сделано потому, что другие потоки
могут использовать указатели, ссылающиеся на данные в стеке завершенного

потока. Если бы они обратились к несуществующему стеку, произошло бы нарушение доступа. Кроме того, при завершении потока система уведомляет об этом все DLL, подключенные к процессу — владельцу завершенного потока. Но при вызове TerminateThread такого не происходит, и процесс может быть завершен некорректно.

· завершается процесс, содержащий данный поток (тоже нежелательно). Функции ExitProcess и TerminateProcess, тоже завершают потоки. Единственное отличие в том, что они прекращают выполнение всех потоков, принадлежавших завершенному процессу. При этом гарантируется высвобождение любых выделенных процессу ресурсов, в том числе стеков потоков. Однако эти две функции уничтожают потоки принудительно — так, будто для каждого из них вызывается функция TerminateThread.

Что происходит при завершении потока:

1. Освобождаются все описатели User-объектов, принадлежавших потоку. В Windows большинство объектов принадлежит процессу, содержащему поток, из которого они были созданы. Сам поток владеет только двумя User-объектами: окнами и ловушками (hooks). Когда поток, создавший такие объекты, завершается, система уничтожает их автоматически. Прочие объекты разрушаются, только когда завершается владевший ими процесс.

2. Код завершения потока меняется со STILL_ACTIVE на код, переданный в функцию ExitThread или TerminateThread.

3. Объект ядра «поток» переводится в свободное состояние.

4. Если данный поток является последним активным потоком в процессе, завершается и сам процесс.

5. Счетчик пользователей объекта ядра «поток» уменьшается на 1.

При завершении потока сопоставленный с ним объект ядра «поток» не освобождается до тех пор, пока не будут закрыты все внешние ссылки на этот объект. Когда поток завершился, толку от его описателя другим потокам в системе в общем немного. Единственное, что они могут сделать, — вызвать функцию GetExitCodeThread, проверить, завершен ли поток, идентифицируемый описателем hThread, и, если да, определить его код завершения

BOOL GetExitCodeThread(
HANDLE hThread,
PDWORD pdwExitCode);

Код завершения возвращается в переменной типа DWORD, на которую указывает pdwExitCode. Если поток не завершен на момент вызова GetExitCodeThread, функция записывает в эту переменную идентификатор STILL_ACTIVE (0x103). При успешном вызове функция возвращает TRUE.

Внутреннее устройство потока

Создав объект ядра «поток», система выделяет стеку потока память из адресного пространства процесса и записывает в его самую верхнюю часть два значения. (Стеки потоков всегда строятся от старших адресов памяти к младшим.) Первое из них является значением параметра pvParam, переданного Вами функции CreateThread, а второе — это содержимое параметра pfnStartAddr, который Вы тоже передаете в CreateThread. У каждого потока собственный набор регистров процессора, называемый контекстом потока. Контекст отражает состояние регистров процессора на момент последнего исполнения потока и записывается в структуру CONTEXT (она определена в заголовочном файле WinNT.h). Эта структура содержится в объекте ядра «поток».

```
struct _CONTEXT // специфична для процессора x86
{
        DWORD ContextFlags;
        DWORD Dr0; DWORD Dr1; DWORD Dr2;
        DWORD Dr3; DWORD Dr6; DWORD Dr7;
        FLOATING_SAVE_AREA FloatSave;
        DWORD SegGs; DWORD SegFs; DWORD SegEs; DWORD SegDs;
        DWORD Edi; DWORD Esi; DWORD Ebx; DWORD Edx;
  DWORD Ecx; DWORD Eax; DWORD Ebp;
  DWORD Eip; DWORD SegCs; DWORD EFlags;
  DWORD Esp; DWORD SegSs;
        BYTE ExtendedRegisters[MAXIMUM_SUPPORTED_EXTENSION];
```

};
Указатель команд (IP) и указатель стека (SP) — два самых важных регистра в контексте потока. Вспомните: потоки выполняются в контексте процесса. Соответственно эти регистры всегда указывают на адреса памяти в адресном пространстве процесса. Когда система инициализирует объект ядра «поток», указателю стека в структуре CONTEXT присваивается тот адрес, по которому в стек потока было записано значение pfnStartAddr, а указателю команд — адрес недокументированной (и неэкспортируемой) функции VOID BaseThreadStart (PTHREAD_START_ROUTINE pfnStartAddr, PVOID pvParam). Эта функция содержится в модуле Kernel32.dll, где, кстати, реализована и функция CreateThread.
Когда новый поток выполняет BaseThreadStart, происходит следующее.
· Ваша функция потока включается во фрейм структурной обработки исключений (далее для краткости — SEH-фрейм), благодаря чему любое исключение, если оно происходит в момент выполнения Вашего потока, получает хоть какую-то обработку, предлагаемую системой по умолчанию.
· Система обращается к Вашей функции потока, передавая ей параметр pvParam, который Вы ранее передали функции CreateThread.
· Когда Ваша функция потока возвращает управление, BaseThreadStart вызывает ExitThread, передавая ей значение, возвращенное Вашей функцией. Счетчик числа пользователей объекта ядра «поток» уменьшается на 1, и выполнение потока прекращается.
· Если Ваш поток вызывает необрабатываемое им исключение, его обрабатывает SEH-фрейм, построенный функцией BaseThreadStart. Обычно в результате этого появляется окно с каким-нибудь сообщением, и, когда пользователь закрывает его, BaseThreadStart вызывает ExitProcess и завершает весь процесс, а не только тот поток, в котором произошло исключение.
Именно благодаря BaseThreadStart Ваша функция потока получает возможность вернуть управление по окончании своей работы. BaseThreadStart, вызывая функцию потока, заталкивает в стек свой адрес возврата и тем самым сообщает ей, куда надо вернуться. Но сама BaseThreadStart не возвращает управление. Иначе возникло бы нарушение доступа, так как в стеке потока нет ее адреса возврата.
При инициализации первичного потока его указатель команд устанавливается на другую недокументированную функцию — BaseProcessStart. VOID BaseProcessStart(PPROCESS_START_ROUTINE pfnStartAddr). Единственное различие между этими функциями в отсутствии ссылки на параметр pvParam. Функция BaseProcessStart обращается к стартовому коду библиотеки C/C++, который выполняет необходимую инициализацию, а затем вызывает Вашу входную функцию main, wmain, WinMain или wWinMain. Когда входная функция возвращает управление, стартовый код библиотеки C/C++ вызывает ExitProcess. Поэтому первичный поток приложения, написанного на C/C++, никогда не возвращается в BaseProcessStart.
Если Вы пользуетесь небезопасными в многопоточной среде функциями, то должны создавать потоки библиотечной функцией _beginthreadex, а не Windows-функцией CreateThread:
unsigned long _beginthreadex(
*void *security,*
unsigned stack_size,
*unsigned (*start_address)(void *),*
*void *arglist,*
unsigned initflag,
*unsigned *thrdaddr);*
У функции _beginthreadex тот же список параметров, что и у CreateThread, но их имена и типы несколько отличаются. (Группа, которая отвечает в Microsoft за разработку и поддержку библиотеки C/C++, считает, что библиотечные функции не должны зависеть от типов данных Windows.)
Несколько важных моментов, связанных с _beginthreadex.
· Каждый поток получает свой блок памяти tiddata, выделяемый из кучи, которая принадлежит библиотеке C/C++. (Структура tiddata определена в файле Mtdll.h.)
· Адрес функции потока, переданный _beginthreadex, запоминается в блоке памяти tiddata. Там же сохраняется и параметр, который должен быть передан этой функции.
· Функция _beginthreadex вызывает CreateThread, так как лишь с ее помощью операционная система может создать новый поток.

· При вызове CreateThread сообщается, что она должна начать выполнение нового потока с функции _threadstartex, а не с того адреса, на который указывает pfnStartAddr. Кроме того, функции потока передается не параметр pvParam, а адрес структуры tiddata.

· Если все проходит успешно, _beginthreadex, как и CreateThread, возвращаетописатель потока. В ином случае возвращается NULL.

Приостановка и возобновление потока

В объекте ядра «поток» имеется переменная — счетчик числа простоев данного потока. При вызове CreateProcess или CreateThread он инициализируется значением, равным 1, которое запрещает системе выделять новому потоку процессорное время. Такая схема весьма разумна: сразу после создания поток не готов к выполнению, ему нужно время для инициализации.

После того как поток полностью инициализирован, CreateProcess или CreateThread проверяет, не передан ли ей флаг CREATE_SUSPENDED, и, если да, возвращает управление, оставив поток в приостановленном состоянии. В ином случае счетчик простоев обнуляется, и поток включается в число планируемых — если только он не ждет какого-то события (например, ввода с клавиатуры).

Создав поток в приостановленном состоянии, Вы можете настроить некоторые его свойства (например, приоритет). Закончив настройку, Вы должны разрешить выполнение потока. Для этого вызовите ***DWORD ResumeThread(HANDLE hThread);*** и передайте описатель потока, возвращенный функцией CreateThread (описатель можно взять и из структуры, на которую указывает параметр ppiProcInfo, передаваемый в CreateProcess).

Если вызов ResumeThread прошел успешно, она возвращает предыдущее значение счетчика простоев данного потока; в ином случае — 0xFFFFFFFF.

Выполнение отдельного потока можно приостанавливать несколько раз. Если поток приостановлен 3 раза, то и возобновлен он должен быть тоже 3 раза — лишь тогда система выделит ему процессорное время. Выполнение потока можно приостановить не только при его создании с флагом CREATE_SUSPENDED, но и вызовом ***DWORD SuspendThread(HANDLE hThread);***

Любой поток может вызвать эту функцию и приостановить выполнение другого потока (конечно, если его описатель известен). Хоть об этом нигде и не говорится (но я все равно скажу!), приостановить свое выполнение поток способен сам, а возобновить себя без посторонней помощи — нет. Как и ResumeThread, функция SuspendThread возвращает предыдущее значение счетчика простоев данного потока.

Поток можно приостанавливать не более чем MAXIMUM_SUSPEND_COUNT раз (в файле WinNT.h это значение определено как 127). Обратите внимание, что SuspendThread в режиме ядра работает асинхронно, но в пользовательском режиме не выполняется, пока поток остается в приостановленном состоянии.

Создавая реальное приложение, будьте осторожны с вызовами SuspendThread, так как нельзя заранее сказать, чем будет заниматься его поток в момент приостановки. Например, он пытается выделить память из кучи и поэтому заблокировал к ней доступ. Тогда другим потокам, которым тоже нужна динамическая память, придется ждать его возобновления. SuspendThread безопасна только в том случае, когда Вы точно знаете, что делает (или может делать) поток, и предусматриваете все меры для исключения вероятных проблем и взаимной блокировки потоков.

22. Понятие пула потоков. Архитектура пула потоков. Операции с потоками при работе с пулом потоков.

Понятие

При написании программ приходится сталкиваться с ситуациями, когда хотелось бы сделать какую-либо небольшую задачу асинхронной, но создание потока оказывается слишком накладным. Для этого существует концепция пула потоков (thread pool). Вкратце, суть идеи состоит в том, что существует некоторый набор потоков (пул), который может расширятся при необходимости, либо уменьшаться. Разработчику необходимо указывать функции, которые нужно выполнить асинхронно, а реализация пула потоков сама берется выполнить задачу наиболее эффективно используя возможности многоядерных процессоров. Посмотрим на то, что предлагает Windows API.

Операции

Простейший сценарий предлагается функцией QueueUserWorkItem. Она принимает указатель на функцию с одним параметром. Указанная задача передается в пул потоков и будет выполнена в соответствии с указанными флагами (флаги помогают пулу потоков определить как лучше выполнить задачу).

Функция RegisterWaitForSingleObject позволяет указать задачу, которая будет выполняться по событию (Event, Mutex, Semaphore, Console input и прочее). Если событие не возникает, то задача выполняется по истечении указанного периода времени. Это, например, удобно использовать для асинхронного отображения видео кадров приходящих по сети. При получении кадра он выводится, а если кадров долго нет, то показывается специальный обновляемый кадр с сообщением о проблеме.

Ещё одна функция — CreateTimerQueueTimer — позволяет создать асинхронный таймер. В этом случае задача ставится в очередь на выполнение регулярно(если другое не задано) через указанный период времени. Уже ясно, что задача выполняется в отдельном потоке, в отличии от обычного таймера Windows.
Интересно отметить, что реализация пула потоков отличается в разных версиях Windows. В Windows XP создается всего 2 потока на 2-х ядерном процессоре, что может оказаться недостаточно для эффективного использования ресурсов. При этом Windows 7, видимо, учитывает не только количество ядер процессора, но и его загрузку в целом. И для той же программы может быть создано более 10 потоков.

Компоненты поддержки пула потоков

		Компонент поддержки		
		ожидания	*ввода-вывода*	*Других операцийтаймера*
Начальное число потоков	Всегда 1	1	0	0
Когда поток создается	При вызове первой функции таймера пула потоков	Один поток для каждых 63зарегистрированныхо бъектов	В системе применяются эвристические методы, но на создание потока влияют следующие факторы • после добавления	В системе применяются эвристические методы, но на создание потока влияют следующие факторы • после добавления

			потока прошло определенное время • установлен флаг WT_EXECUTELONGFUNCTION • число элементов в очереди превышает пороговое значение	потока прошло определенное время • установлен флаг WT_EXECUTELONGFUNCTION • число элементов в очереди превышает пороговое значение
Когда поток разрушается	При завершении процесса	Приотсутствииизарегистрированныхобъектов ожидания	При отсутствии у потока текущих запросов на ввод-вывод и простое в течение определенного порогового времени (около минуты)	При простое потока в течение определенногопорогового времени (около минусы)
Как поток ждет	В "тревожном"состоянии	*WaitForMultipleObjectsEx*	В "тревожном" состоянии	*GetQueuedCompletionStatus*
Когда поток пробуждается	Приосвобождении«ожидаемого таймера", которыйпосылает в очередь APC-вызов	При освобождении объекта ядра	При посылке в очередь APC-вызова или завершении запроса на ввод- вывод	При поступлении запроса о статусе завершения или о завершении ввода вывода (портзавершениятребует, чтобы число потоков не превышало число процессоров более чем в 2 раза)

Термин «рабочие фабрики» относится к внутреннему механизму, используемому для реализации пулов потоков пользовательского режима. Устаревшие процедуры пула потоков были полностью реализованы в пользовательском режиме внутри библиотеки Ntdll.dll , и Windows API предоставляет различные процедуры для вызова соответствующих подпрограмм, обеспечивающих таймеры ожидания, ожидающие функции обратного вызова и автоматическое создание и удаление потоков в зависимости от объема проделываемой работы.

Поскольку ядро может иметь непосредственный контроль над планированием, созданием и завершением потоков без обычных издержек, связанных с выпол- нением этих операций из пользовательского режима, большинство функциональных механизмов, требующихся для поддержки реализации в Windows пула потоков пользовательского режима, теперь располагаются в ядре, что также упрощает код, который нужно создавать разработчикам. Например, создание ра- бочего пула в удаленном процессе может быть осуществлено с помощью одного- единственного API-вызова вместо сложной череды вызовов в виртуальной памяти, которые обычно для этого требовались. Согласно этой модели, библиотека Ntdll.dll просто предоставляет интерфейсы и высокоуровневые API-функции, требуемые для создания интерфейса с кодом рабочей фабрики.

Этот механизм пула потоков, управляемого ядром, управляется с помощью такого типа диспетчера объектов, который называется TpWorkerFactory , а так- же с помощью четырех исходных системных вызовов для управления фабри- кой и ее работниками (NtCreateWorkerFactory , NtWorkerFactory Worker Ready , NtRelease WorkerFactoryWorker , NtShutdownWorkerFactory), двух исходных

вызовов запросов-установок (NtQueryInformationWorkerFactory и NtSetInformation- WorkerFactory) и ждущего вызова (NtWaitForWorkViaWorkerFactory).

Как и все другие исходные системные вызовы, эти вызовы предоставляют пользовательскому режиму дескриптор объекта TpWorkerFactory , в котором содержится такая информация, как имя и атрибуты объекта, заданная маска доступа и дескриптор безопасности. Но в отличие от других системных вызовов, упакованных Windows API, управление пулом потоков обрабатывается исходным кодом библиотеки Ntdll.dll , это означает, что разработчики работают с непроницаемым дескриптором (указателем TP_WORK), принадлежащем Ntdll. dll , в котором хранится настоящий дескриптор.

Как следует из названия «рабочая фабрика», реализация этой фабрики отвечает за размещение рабочих потоков (и за вызов кода в заданной точке входа рабочего потока пользовательского режима), поддержку минимального и максимального числа потоков (для постоянных рабочих пулов, либо для полностью динамических пулов), а также за другую учетную информацию. Это позволяет выполнять такие операции, как завершение работы пула потоков с помощью единственного обращения к ядру, поскольку ядро является единственным компонентом, отвечающим за создание и завершение потоков.

Поскольку ядро создает новые потоки по мере надобности в динамическом режиме, основываясь на предоставленных минимальных и максимальных количествах, это также повышает масштабируемость приложений, использующих новую реализацию пула потоков. Рабочая фабрика создаст новый поток при выполнении всех следующих условий:

- Количество доступных работником ниже максимального количества работ- ников, указанного для фабрики (по умолчанию это количество составляет 500 работников).

- Рабочая фабрика имеет привязанные к ней объекты (привязанным объектом может быть, к примеру, ALPC-порт, на котором строит свое ожидание рабочий поток) или поток, активированный в пуле.

- Существуют отложенные пакеты запросов ввода-вывода (IRP-пакеты), свя- занные с рабочим потоком.

- Разрешено создание динамических потоков.

И она завершит работу потоков при простое свыше 10 секунд (по умолчанию).

Кроме того, при старой реализации разработчики всегда могли воспользо- ваться максимально возможным количеством потоков, основанном на количе- стве процессоров в системе. Благодаря поддержке динамических процессоров в Windows Server теперь есть возможность использовать для приложений пулы потоков для автоматического использования новых процессоров, добавленных в ходе выполнения кода.

Следует заметить, что поддержка рабочей фабрики является всего лишь обо- лочкой для управления обычными задачами, которая в противном случае должна была бы выполняться в пользовательском режиме (с потерей производитель- ности). Основная часть логики нового кода пула потоков остается в той стороне этой архитектуры, которая относится к библиотеке Ntdll.dll . (Теоретически, используя недокументированные функции для рабочих фабрик, можно создать другую реализацию пула потоков.) К тому же это не код рабочих фабрик предоставляет масштабируемость, внутренние

механизмы ожидания и эффективность обработки рабочих заданий. Этим занимается куда более старый компонентWindows, который уже рассматривался — порт завершения ввода-вывода, или,если быть точнее, очередей ядра — kernel queues (KQUEUE).

Фактически, при создании рабочей фабрики должен быть уже создан код пользовательского режима и должен быть передан его дескриптор. Именно через этот порт завершения ввода-вывода реализация пользовательского режима будет выстраивать в очередь на работу, а также находиться в ожидании работы путем использования системных вызовов рабочей фабрики вместо API-функций порта завершения ввода-вывода. Но, согласно внутреннему устройству, «освобождающий» вызов рабочей фабрики (который ставит работу в очередь) является оболочкой вокруг процедуры IoSetIoCompletionEx , которая увеличивает количество отложенной работы, а «ожидающий» вызов является оболочкой вокруг процедуры IoRemoveIoCompletion . Обе эти процедуры совершают вызов в код реализации очереди ядра. Задача кода рабочей фабрики заключается в управлении либо постоянными,статическими, либо динамическими пулами потоков, в создании оболочки модели порта завершения ввода-вывода в интерфейсах, пытающихся предотвратить остановку продвижения рабочих очередей путем автоматического создания динамических потоков, и в упрощении глобальной очистки и завершении операций в ходе запроса на завершение работы фабрики, а также в беспрепятственной блокировке новых запросов к фабрике при таком развитии событий.

23. Распределение процессорного времени между потоками ОС Windows. Механизм приоритетов. Класс приоритета процесса. Относительный уровень приоритета потока. Базовый и динамический приоритеты потока. Операции с приоритетами.

Внутренне Windows использует 32 уровня приоритета, от 0 до 31. Эти значения разбиваются на части следующим образом:

- шестнадцать уровней реального времени (от 16 до 31);
- шестнадцать изменяющихся уровней (от 0 до 15), из которых уровень 0 зарезервирован для потока обнуления страниц.

Рис. 5.14. Уровни приоритета потоков

Уровни приоритета потоков назначаются исходя из двух разных позиций: одной от Windows API и другой от ядра Windows. Сначала Windows API систематизирует процессы по классу приоритета, который им присваивается при создании (номера представляют внутренний индекс PROCESS_PRIORITY_CLASS, распознаваемый ядром): Реального времени — Real-time (4), Высокий — High (3), Выше обычного — Above Normal (7), Обычный — Normal (2), Ниже обычного — Below Normal (5) и Простоя — Idle (1). Затем назначается относительный приоритет отдельных потоков внутри этих процессов. Здесь номера представляют изменение приоритета, применяющееся к базовому приоритету процесса: Критичный по времени — Time-critical (15), Наивысший — Highest (2), Выше обычного — Above-normal (1), Обычный —Normal (0), Ниже обычного — Below-normal (–1), Самый низший — Lowest (–2)и Простоя — Idle (–15).

Таблица 5.3. Отображение приоритетов ядра Windows на Windows API

Класс приоритета/ Относительный приоритет	Realtime	High	Above	Normal	Below Normal	Idle
Time Critical (+ насыщение)	31	15	15	15	15	15
Highest (+2)	26	15	12	10	8	6
Above Normal (+1)	25	14	11	9	7	5
Normal (0)	24	13	10	8	6	4
Below Normal (–1)	23	12	9	7	5	3
Lowest (–2)	22	11	8	6	4	2
Idle (– насыщение)	16	1	1	1	1	1

Поэтому в Windows API каждый поток имеет базовый приоритет, являющийся функцией класса приоритета процесса и его относительного приоритета процесса.

В ядре класс приоритета процесса преобразуется в базовый приоритет путем использования процедуры PspPriorityTable и показанных ранее индексов PROCESS_PRIORITY_CLASS, устанавливающих приоритеты 4, 8, 13, 14, 6 и 10 соответственно. (Это фиксированное отображение, которое не может быть изменено.) Затем применяется относительный приоритет потока в качестве разницы для этого базового приоритета.

Например, наивысший «Highest»-поток получит базовый приоритет потока на два уровня выше, чем базовый приоритет его процесса.

Следует заметить, что относительные приоритеты потоков Time-Critical и Idle сохраняют свои соответствующие значения независимо от класса приоритета процесса (если только это не Realtime). Причина в том, что Windows API требует насыщения (saturation) приоритета от ядра путем передачи 16 или –16 в качестве запрошенного относительного приоритета (вместо 15 или –15). Тогда это будет распознано ядром как запрос на насыщение, и в структуре KTHREAD будет установлено поле Saturation. Это приведет к тому, что при положительном насыщении поток получит наивысший возможный приоритет внутри его класса приоритета (динамического или реального времени) или при отрицательном насыщении он получит самый низкий из возможных приоритетов. Кроме того, последующие запросы на изменение базового приоритета процесса уже не будут влиять на базовый приоритет этих потоков, потому что насыщенные потоки пропускаются в коде обработки. В то время как у процесса имеется только одно базовое значение приоритета, у каждого потока имеется два значения приоритета: текущее и базовое. Решения по планированию принимаются исходя из текущего приоритета. система при определенных обстоятельствах на короткие периоды времени повышает приоритет потоков в динамическом диапазоне (от 0 до 15). Windows никогда не регулирует приоритет потоков в диапазоне реального времени (от 16 до 31), поэтому они всегда имеют один и тот же базовый и текущий приоритет.

Исходный базовый приоритет потока наследуется из базового приоритета процесса. Процесс по умолчанию наследует свой базовый приоритет у того процесса, который его создал. Приоритет процесса может быть также изменен после создания процесса путем использования функции SetPriorityClass или различных инструментальных средств, предлагающих такую функцию, например Диспетчера задач и Process Explorer.

Как правило, пользовательские приложения и службы запускаются с обычным базовым приоритетом (normal), поэтому их исходный поток чаще всего выполняется с уровнем приоритета 8. Но некоторые системные процессы Windows (например, диспетчер сеансов, диспетчер управления службами и процесс идентификации локальной безопасности) имеют немного более высокий базовый приоритет, чем тот, который используется по умолчанию для класса Normal (8). Это более высокое значение по умолчанию гарантирует, что все потоки в этих процессах будут запускаться с более высоким приоритетом, превышающим значение по умолчанию равное 8.

Функции:

bool **SetPriorityClass**(HANDLE hProcess, DWORD dwPriorityClass);

Устанавливает класс приоритета для указанного процесса . Это значение вместе с приоритетным значением каждого потока процесса , определяет базовый уровень приоритета каждого потока.

hProcess - Дискриптор процесса

DWORD - Класс приоритета для процесса

DWORD **GetPriorityClass**(HANDLE hProcess);

Получает класс приоритета для указанного процесса . Это значение , вместе с значением приоритета каждого потока процесса , определяет базовый уровень приоритета каждого потока.

Возвращаемое значение
Если функция завершается успешно , возвращаемое значение является приоритетным класс указанного процесса .

Если функция завершается ошибкой, возвращаемое значение равно нулю . Чтобы получить дополнительную информацию об ошибке, вызовите GetLastError

```
bool SetThreadPriority(HANDLE hThread, int nPriority);
```

Устанавливает значение приоритета для указанного потока . Это значение , вместе с классом приоритета процесса потока, после определяет базовый уровень приоритета потока.

```
int GetThreadPriority(HANDLE hThread);
```

Получает значение приоритета для указанного потока . Это значение , вместе с классом приоритета процесса потока, после определяет базовый уровень приоритета потока.

```
bool SetProcessPriorityBoost(HANDLE hProcess,bool disablePriorityBoost);
```

Включает или отключает способность системы временно повысить приоритет указанного процесса .

SetThreadPriorityBoost.

Включает или отключает способность системы временно повысить приоритет

потока.

```
bool GetProcessPriorityBoost(HANDLE hProcess,
bool* pDisablePriorityBoost); GetThreadPriorityBoost.
bool SwitchToThread(); // yield execution to another thread
void Sleep(DWORD dwMilliseconds);
DWORD SleepEx(DWORD dwMilliseconds, bool bAlertable);
```

Динамический приоритет:

Когда окно потока активизируется или поток находится в состоянии ожидания сообщений и получает сообщение или поток заблокирован на объекте ожидания и объект освобождается, ОС увеличивает его приоритет на 2, спустя квант времени ОС понижает приоритет на 1, спустя еще квант времени понижает еще на 1.

Динамический приоритет потока не может быть меньше базового приоритета и не может быть больше приоритета с номером 15. ОС не выполняет корректировку приоритета для потоков с приоритетом от 16 до 31. Приоритеты с 16 по 31 – приоритеты реального времени, их использовать не рекомендуется, причем даже в тех случаях, когда программа выполняет критические по времени операции. Поток, выполняющийся с приоритетом реального времени будет иметь даже больший приоритет, чем драйвер мыши или клавиатуры и чем другие драйверы ОС.

24. Механизмы синхронизации потоков одного и разных процессов в ОС Windows. Обзор и сравнительная характеристика механизмов синхронизации.

Между потоками одного процесса:

- Критическая секция – Critical Section
- Ожидаемое условие – Condition Variable
- Атомарная операция – Interlocked (Atomic) Function
- Барьер синхронизации – Synchronization Barrier

Между потоками любых локальных процессов:

- Блокировка – Mutex
- Семафор – Semaphore
- Событие – Event
- Ожидаемый таймер – Waitable Timer

Между потоками удаленных процессов:

- Почтовый ящик – Mailslot
- Труба – Named/Unnamed Pipe
- Windows Socket

Критическая секция (critical section) — это небольшой участок кода, требующий монопольного доступа к каким-то общим данным. Она позволяет сделать так, чтобы единовременно только один поток получал доступ к определенному ресурсу. Естественно, система может в любой момент вытеснить Ваш поток и подключить к процессору другой, но ни один из потоков, которым нужен занятый Вами ресурс, не получит процессорное время до тех пор, пока Ваш поток не выйдет за границы критической секции. Запомните несколько важных вещей. Если у Вас есть ресурс, разделяемый несколькими потоками, Вы должны создать экземпляр структуры CRITICAL_SECTION.

Структура CRITICAL_SECTION похожа на туалетную кабинку в самолете, а данные, которые нужно защитить, — на унитаз. Туалетная кабинка (критическая секция) в самолете очень маленькая, и единовременно в ней может находиться только один человек (поток), пользующийся унитазом (защищенным ресурсом).

Если у Вас есть ресурсы, всегда используемые вместе, Вы можете поместить их в одну кабинку — единственная структура CRITICAL_SECTION будет охранять их всех. Но если ресурсы не всегда используются вместе (например, потоки 1 и 2 работают с одним ресурсом, а потоки 1 и 3 — с другим), Вам придется создать им по отдельной кабинке, или структуре CRITICAL_SECTION.

Теперь в каждом участке кода, где Вы обращаетесь к разделяемому ресурсу, вызывайте EnterCriticalSection, передавая ей адрес структуры CRITICAL_SECTION, которая выделена для этого ресурса. Иными словами, поток, желая обратиться к ресурсу, должен сначала убедиться, нет ли на двери кабинки знака «занято». Структура CRITICAL_SECTION идентифицирует кабинку, в которую хочет войти поток, а функция EnterCriticalSection — тот инструмент, с помощью которого он узнает, свободна или занята кабинка. EnterCriticalSection допустит вызвавший ее поток в кабинку, если определит, что та свободна. В ином случае (кабинка занята) EnterCriticalSection заставит его ждать, пока она не освободится.

Поток, покидая участок кода, где он работал с защищенным ресурсом, должен вызвать функцию LeaveCriticalSection. Тем самым он уведомляет систему о том, что

кабинка с данным ресурсом освободилась. Если Вы забудете это сделать, система будет считать, что ресурс все еще занят, и не позволит обратиться к нему другим ждущим потокам. То есть Вы вышли из кабинки и оставили на двери знак «занято».

Самое сложное — запомнить, что любой участок кода, работающего с разделяемым ресурсом, нужно заключить в вызовы функций EnterCriticalSection и LeaveCriticalSection. Если Вы забудете сделать это хотя бы в одном месте, ресурс может быть поврежден.

Преимущество критических секций в том, что они просты в использовании и выполняются очень быстро, так как реализованы на основе Interlocked-функций. А главный недостаток — нельзя синхронизировать потоки в разных процессах.

```
struct CRITICAL_SECTION {
      LONG LockCount;
      LONG RecursionCount;
      HANDLE OwningThread;
      HANDLE LockSemaphore;
      ULONG_PTR SpinCount;
};
void InitializeCriticalSection(CRITICAL_SECTION* lpCriticalSection);
void EnterCriticalSection(CRITICAL_SECTION* lpCriticalSection);
void LeaveCriticalSection(CRITICAL_SECTION* lpCriticalSection);
bool TryEnterCriticalSection(CRITICAL_SECTION* lpCriticalSection);
bool InitializeCriticalSectionAndSpinCount(CRITICAL_SECTION* lpCriticalSection,
DWORD dwSpinCount); SetCriticalSectionSpinCount.
```

Ожидаемое условие – механизм синхронизации, позволяющий потокам дождаться выполнения некоторого (сложного) условия. Состоит из критической секции и переменной условия.

*void **InitializeConditionVariable**(CONDITION_VARIABLE* CondVariable);* - возвращает указатель на условную переменную

Замечания:

Потоки могут атомарно снять блокировку и войти в состояние сна используя **SleepConditionVariableCS и SleepConditionVariableSRW**. Потоки просыпается с помощью **WakeConditionVariable и WakeAllConditionVariable.**

Переменные условия являются объектнами пользовательского режима, которые не могут быть разделены между процессами. Переменную условия нельзя скопирвоать или переместить. Процесс не может модифицировать объект, и вместо этого должен рассматривать его как логически непрозначный. Только используя функции условий переменной можно управлять их состоянием.

bool **SleepConditionVariableCS**(CONDITION_VARIABLE* CondVariable, CRITICAL_SECTION* CriticalSection, DWORD dwMilliseconds);

Параметры:

ConditionVariable [in, out] - указатель на условную переменную, которая обязательно должно быть проинициализирована с помощью *InitializeConditionVariable*.

CriticalSection [in, out] – указатель на объект критической секции. Эта критическая секция должна быть введена только при ее вызове. Входить в критическую секцию необходимо только один раз во время вызова SleepConditionVariableCS .

dwMilliseconds [in] - Тайм-аут интервал, в миллисекундах. Если интервал ожидания истекает, функции повторного требует критическую секцию и возвращает ноль. Если dwMilliseconds равна

нулю, то функция проверяет состояния указанных объектов и возвращает немедленно. Если dwMilliseconds бесконечна, интервал ожидания работы функции никогда не истекает.

Возвращаемое значение:

Если функция завершается успешно, возвращаемое значение отлично от нуля.

Если функция падает или интервал ожидания истекает, то возвращаемое значение равно нулю. Чтобы получить дополнительную информацию об ошибке, вызовите GetLastError. Возможные коды ошибок включают ERROR_TIMEOUT, который указывает, что тайм-аут интервал прошел, прежде чем другой поток пытается разбудить спящий поток.

Замечания

Поток, который спит на условной переменной можно активировать, прежде чем указанный интервал ожидания истекло, используя функцию WakeConditionVariable или WakeAllConditionVariable. В этом случае, поток просыпается, когда процесс просыпания завершен, а не когда его интервал ожидания истекает. После того как поток проснулся, он вновь приобретает критическую секцию и входит в нее, когда поток вошел в состояние сна.

Переменные условия подлежат паразитных пробуждениями (те, которые не связаны с явным пробуждением) и украденных пробуждениями (другой поток управляет запуском перед просыпающимся потоком). Таким образом, вы должны перепроверить предикат (обычно в цикле while) после возврата из режима сна.

*bool **SleepConditionVariableSRW**(CONDITION_VARIABLE* CondVariable, SRWLOCK* SRWLock, DWORD dwMilliseconds, ULONG Flags);* - спит на указанной переменной условия и реализует специальную блокировку как атомарную операцию

Параметры

ConditionVariable [in, out] -||-

SRWLock [in, out] –указатель блокировки, используется вместе с флагами.

dwMilliseconds [in] -||-

Flags [in] Если этот параметр CONDITION_VARIABLE_LOCKMODE_SHARED, SRW блокировка в разделенном режиме (shared mode). Иначе – монопольна.

Возвращаемое значение:

Если функция завершается успешно, возвращаемое значение отлично от нуля, ошибка - 0. Чтобы получить дополнительную информацию об ошибке, вызовите GetLastError. Если тайм-аут истекает функция возвращает FALSE, и GetLastError возвращает ERROR_TIMEOUT.

Замечания:

Если блокировка разблокирована, когда эта функция вызывается, поведение функции не определено. + что из предыдущей функции

*void **WakeConditionVariable**(CONDITION_VARIABLE* CondVariable);*

Замечания:

WakeAllConditionVariable - просыпаются все ожидающие потоки в то время как WakeConditionVariable - просыпается только один поток. Проснувшись один поток аналогично настройке авто-сброс событие, а все бодрствующие потоки похожи на пульсирующий ручной событие сброса (Waking one thread is similar to setting an auto-reset event, while waking all threads is similar to pulsing a manual reset event but more reliable).

*void **WakeAllConditionVariable**(CONDITION_VARIABLE* CondVariable);*

Пример использования ожидаемого условия:

```
// CRITICAL_SECTION criticalSection;
// CONDITION_VARIABLE conditionVariable;
EnterCriticalSection(&criticalSection);
try {
      while (DataDoesntSatisfyCondition()) // функция программиста
            SleepConditionVariableCS(&conditionVariable, &criticalSection,
 INFINITE);
      }
catch (...){
      LeaveCriticalSection(&criticalSection);
      throw;
}
LeaveCriticalSection(&criticalSection);
```

Пример использования ожидаемого условия:

```
// CRITICAL_SECTION criticalSection;
// CONDITION_VARIABLE conditionVariable;
EnterCriticalSection(&criticalSection);
try {
      ChangeData(); // процедура программиста
      WakeAllConditionVariableCS(&conditionVariable);
}
catch (...) {
      LeaveCriticalSection(&criticalSection);
      throw;
}
LeaveCriticalSection(&criticalSection);
```

Атомарная операция – простая операция над машинным словом, которая или выполняется целиком, или не выполняется вообще.

На компьютерах с процессорами семейства x86 эти функции выдают по шине аппаратный сигнал, не давая другому процессору обратиться по тому же адресу памяти. На платформе Alpha Interlocked-функции действуют примерно так:

1. Устанавливают специальный битовый флаг процессора, указывающий, что данный адрес памяти сейчас занят.

2. Считывают значение из памяти в регистр.

3. Изменяют значение в регистре.

4. Если битовый флаг сброшен, повторяют операции, начиная с п. 2. В ином случае значение из регистра помещается обратно в память.

Вас, наверное, удивило, с какой это стати битовый флаг может оказаться сброшенным? Все очень просто. Его может сбросить другой процессор в системе, пытаясь модифицировать тот же адрес памяти, а это заставляет Interlocked-функции вернуться в п. 2.

Другой важный аспект, связанный с Interlocked-функциями, состоит в том, что они выполняются чрезвычайно быстро. Вызов такой функции обычно требует не более 50 тактов процессора, и при этом не происходит перехода из пользовательского режима в режим ядра (а он отнимает не менее 1000 тактов).

LONG ***InterlockedIncrement****(LONG*Addend);*

InterlockedDecrement, ***InterlockedAnd***, ***InterlockedOr***, ***InterlockedXor***.

InterlockedExchangeAdd полностью заменяет обе эти устаревшие функции. Новая функция умеет добавлять и вычитать произвольные значения, а функции InterlockedIncrement и InterlockedDecrement увеличивают и уменьшают значения только на 1.

LONG ***InterlockedExchange****(LONG* Target, LONG Value);* ***InterlockedExchangePointer****.* InterlockedExchange и InterlockedExchangePointer монопольно заменяют текущее значение переменной типа LONG, адрес которой передается в первом параметре, на значение, передаваемое во втором параметре. В 32-разрядном приложении обе функции работают с 32-разрядными значениями, но в 64-разрядной программе первая
оперирует с 32-разрядными значениями, а вторая — с 64-разрядными.

LONG ***InterlockedCompareExchange****(LONG* Destination,*
LONG Exchange, LONG Comparand);
InterlockedCompareExchangePointer. Они выполняют операцию сравнения и присвоения на уровне атомарного доступа. В 32-разрядном приложении обе функции работают с 32-разрядными значениями, но в 64-разрядном приложении InterlockedCompareExchange используется для 32-разрядных значений, а InterlockedCompareExchangePointer — для 64-разрядных.

Функция сравнивает текущее значение переменной типа LONG (на которую указывает параметр plDestination) со значением, передаваемым в параметре lComparand.
Если значения совпадают, *plDestination получает значение параметра lExchange; в ином случае *plDestination остается без изменений. Функция возвращает исходное значение *plDestination. И не забывайте, что все эти действия выполняются как единая атомарная операция.

InterlockedBitTestAnd(**Set**/**Reset**/**Complement**).
InterlockedXxx64, **InterlockedXxxNoFence**,
InterlockedXxxAcquire, **InterlockedXxxRelease**.

Обратите внимание на отсутствие Interlocked-функции, позволяющей просто считывать значение какой-то переменной, не меняя его. Она и не нужна. Если один поток модифицирует переменную с помощью какой-либо Interlocked-функции в тот момент, когда другой читает содержимое той же переменной, ее значение, прочитанное вторым потоком, всегда будет достоверным. Он получит либо исходное, либо измененное значение переменной. Поток, конечно, не знает, какое именно значение он считал, но главное, что оно корректно и не является некоей произвольной величиной. В большинстве приложений этого вполне достаточно.

Ожидание

Объекты ядра Windows могут находится в одном из двух состояний:

- Свободном состоянии (signaled)
- Занятом (not signaled)

Синхронизация – ожидание освобождения объекта ядра:

DWORD ***WaitForSingleObject****(HANDLE hHandle, DWORD dwMilliseconds); -*

Когда поток вызывает эту функцию, первый параметр, hHandle, идентифицирует объект ядра, поддерживающий состояния «свободен-занят». Второй параметр, dwMilliseconds, указывает, сколько времени (в миллисекундах) поток готов ждать освобождения объекта. Следующий вызов сообщает системе, что поток будет ждать до тех пор, пока не завершится процесс, идентифицируемый описателем hProcess:

WaitForSingleObject(hProcess, INFINITE);

Возвращаемое значение функции WaitForSingleObject указывает, почему вызывающий поток снова стал планируемым. Если функция возвращает WAIT_OBJECT_0, объект свободен, а если WAIT_TIMEOUT — заданное время ожидания (таймаут) истекло.

При передаче неверного параметра (например, недопустимого описателя) WaitForSingleObject возвращает WAIT_FAILED. Чтобы выяснить конкретную причину ошибки, вызовите функцию GetLastError.

В параметре dwMilliseconds можно передать 0, и тогда WaitForSingleObject немедленно вернет управление.

DWORD ***WaitForMultipleObjects****(DWORD nCount,*

const HANDLE lpHandles, bool bWaitAll, DWORD dwMilliseconds);* - аналогична WaitForSingleObject с тем исключением, что позволяет ждать освобождения сразу нескольких объектов или какого-то одного из списка объектов.

Параметр *dwCount* определяет количество интересующих Вас объектов ядра. Его значение должно быть в пределах от 1 до MAXIMUM_WAIT_OBJECTS (в заголовочных файлах Windows оно определено как 64). Параметр *lpHandles* — это указатель на массив описателей объектов ядра.

Если параметр *fWaitAll* = TRUE, функция не даст потоку возобновить свою работу, пока не освободятся все объекты, иначе – функция ждет освобождения только одного из объектов.

Параметр *dwMilliseconds* идентичен одноименному параметру функции WaitForSingleObject. Если Вы указываете конкретное время ожидания, то по его истечении функция в любом случае возвращает управление. И опять же, в этом параметре обычно передают INFINITE (будьте внимательны при написании кода, чтобы не создать ситуацию взаимной блокировки).

Возвращаемое значение функции WaitForMultipleObjects сообщает, почему возобновилось выполнение вызвавшего ее потока. Значения WAIT_FAILED(неверный параметр) и WAIT_TIMEOUT (заданное время ожидания истекло) никаких пояснений не требуют.

Если Вы передали TRUE в параметре fWaitAll и все объекты перешли в свободное состояние, функция возвращает значение *WAIT_OBJECT_0*.

Если же fWaitAll приравнен FALSE, она возвращает управление, как только освобождается любой из объектов. Вы, по-видимому, захотите выяснить, какой именно объект освободился. В этом случае возвращается значение от WAIT_OBJECT_0 до WAIT_OBJECT_0 + dwCount – 1. Иначе говоря, если возвращаемое значение не равно WAIT_TIMEOUT или WAIT_FAILED, вычтите из него значение WAIT_OBJECT_0, и Вы получите индекс в массиве описателей, на который указывает второй параметр функции WaitForMultipleObjects. Индекс подскажет Вам, какой объект перешел в незанятое состояние.

Значение WAIT_ABANDONED

Отсюда возникает вопрос: а что будет, если поток, которому принадлежит мьютекс, завершится, не успев его освободить? В таком случае система считает, что произошел отказ от мьютекса, и автоматически переводит его в свободное состояние (сбрасывая при этом все его счетчики в исходное состояние). Если этот мьютекс ждут другие потоки, система, как обычно, «по-честному» выбирает один из потоков и позволяет ему захватить мьютекс. Тогда Wait-функция возвращает потоку WAIT_ABANDONED вместо WAIT_OBJECT_0, и тот узнает, что мьютекс освобожден некорректно. Данная ситуация, конечно, не самая лучшая. Выяснить, что сделал с защищенными данными завершенный поток — бывший владелец объекта-мьютекса, увы, невозможно.

В реальности программы никогда специально не проверяют возвращаемое значение на WAIT_ABANDONED, потому что такое завершение потоков происходит очень редко.

DWORD **WaitForSingleObjectEx**(HANDLE hHandle, DWORD dwMillisec, bool bAlertable);
WaitForMultipleObjectsEx. Ждет, пока указанный объект не находится в сигнальном состоянии,

завершение подпрограммы ввода / вывода или асинхронный вызов процедуры (APC) ставится в очередь в потоке, или тайм-аут интервал истекает.

bAlertable [in] если этот параметр = TRUE и поток в состоянии ожидания, функция возвращает, когда система стоит в очереди , завершение подпрограммы ввода / вывода или APC, и поток запускает подпрограмму или функцию. В противном случае, функция не возвращает ничего, и процедура завершение или функция APC не выполняется. (If this parameter is TRUE and the thread is in the waiting state, the function returns when the system queues an I/O completion routine or APC, and the thread runs the routine or function. Otherwise, the function does not return, and the completion routine or APC function is not executed.)

Барьеры – весьма своеобразное средство синхронизации. Идея его в том, чтобы в определенной точке ожидания собралось заданное число потоков управления. Только после этого они смогут продолжить выполнение. (Поговорка "семеро одного не ждут" к барьерам не применима.) Барьеры полезны для организации коллективных распределенных вычислений в многопроцессорной конфигурации, когда каждый участник (поток управления) выполняет часть работы, а в точке сбора частичные результаты объединяются в общий итог.
Функции, ассоциированные с барьерами, подразделяются на следующие группы (#include <pthread.h>).

инициализация и разрушение барьеров:

```
int pthread_barrier_init (pthread_barrier_t *restrict barrier, const pthread_barrierattr_t  *restrict attr,
     unsigned count);

int pthread_barrier_destroy (
  pthread_barrier_t *barrier);
```

синхронизация на барьере:

```
int pthread_barrier_wait (
  pthread_barrier_t *barrier);
```

инициализация и разрушение атрибутных объектов барьеров:

```
int pthread_barrierattr_init (
  pthread_barrierattr_t *attr);

int pthread_barrierattr_destroy (
  pthread_barrierattr_t *attr);
```

опрос и установка атрибутов барьеров в атрибутных объектах:

```
int pthread_barrierattr_getpshared
     (const pthread_barrierattr_t
       *restrict attr,
     int *restrict pshared);

int pthread_barrierattr_setpshared
     (pthread_barrierattr_t *attr,
       int pshared);
```

Аргумент count в функции инициализации барьера pthread_barrier_init(). Он задает количество синхронизируемых потоков управления. Столько потоков должны вызвать функцию pthread_barrier_wait(), прежде чем каждый из них сможет успешно завершить вызов и продолжить выполнение. (Разумеется, значение count должно быть положительным.)

Когда к функции pthread_barrier_wait() обратилось требуемое число потоков управления, одному из них (стандарт POSIX-2001 не специфицирует, какому именно) в качестве результата возвращается именованная константа PTHREAD_BARRIER_SERIAL_THREAD, а всем другим выдаются нули. После этого барьер возвращается в начальное (инициализированное) состояние, а выделенный поток может выполнить соответствующие объединительные действия.

Описанная схема работы проиллюстрирована:

```
if ((status = pthread_barrier_wait(
  &barrier)) ==
      PTHREAD_BARRIER_SERIAL_THREAD) {
 /* Выделенные (обычно – объединительные) */
 /* действия.     */
 /* Выполняются каким-то одним потоком */
 /* управления */

} else {
   /* Эта часть выполняется всеми */
   /* прочими потоками */
   /* управления */
   if (status != 0) {
   /* Обработка ошибочной ситуации */
   } else {
      /* Нормальное "невыделенное" */
      /* завершение ожидания */
      /* на барьере */
   }
}

/* Повторная синхронизация – */
/* ожидание завершения выделенных действий */
status = pthread_barrier_wait (&barrier);
/* Продолжение параллельной работы */
```

Отметим, что для барьеров отсутствует вариант синхронизации с контролем времени ожидания.

Блокировка – mutex (**mut**ually **ex**clusive), бинарный семафор. Используется для обеспечения монопольного доступа к некоторому ресурсу со стороны нескольких потоков (различных процессов).

Для мьютексов определены следующие правила:

- если его идентификатор потока равен 0 (у самого потока не может быть такой идентификатор), мьютекс не захвачен ни одним из потоков и находится в свободном состоянии;
- если его идентификатор потока не равен 0, мьютекс захвачен одним из потоков и находится в занятом состоянии;
- в отличие от других объектов ядра мьютексы могут нарушать обычные правила, действующие в операционной системе
-

HANDLE ***CreateMutex****(SECURITY_ATTRIBUTES* lpMutexAttributes,*
bool bInitialOwner, LPCTSTR lpName); Создание объекта-мьютекса

lpName - Передавая в нем NULL, Вы создаете безымянный (анонимный) объект ядра. В этом случае Вы можете разделять объект между процессами либо через, либо с помощью DuplicateHandle. А чтобы разделять объект по имени, Вы должны присвоить ему какое-нибудь имя. Тогда вместо

NULL в параметре *lpName* нужно передать адрес строки с именем, завершаемой нулевым символом. Имя может быть длиной до MAX_PATH знаков.

Чтобы создать наследуемый описатель, родительский процесс выделяет и инициализирует структуру SECURITY_ATTRIBUTES, а затем передает ее адрес требуемой Create-функции.

Большинство приложений вместо этого аргумента передает NULL и создает объект с защитой по умолчанию. Такая защита подразумевает, что создатель объекта и любой член группы администраторов получают к нему полный доступ, а все прочие к объекту не допускаются. Однако Вы можете создать и инициализировать структуру

SECURITY_ATTRIBUTES, а затем передать ее адрес. Она выглядит так:

```
typedef struct _SECURITY_ATTRIBUTES {
DWORD nLength;
LPVOID lpSecurityDescriptor;
BOOL bInheritHandle;
} SECURITY_ATTRIBUTES;
```

Хотя структура называется SECURITY_ATTRIBUTES, лишь один ее элемент имеет отношение к защите — lpSecurityDescriptor. Если надо ограничить доступ к созданному Вами объекту ядра, создайте дескриптор защиты и инициализируйте структуру SECURITY_ATTRIBUTES следующим образом:

```
SECURITY_ATTRIBUTES sa;
sa.nLength = sizeof(sa); // используется для выяснения версий
sa.lpSecurityDescriptor = pSD; // адрес инициализированной SD
sa.bInheritHandle = FALSE; // об этом позже
HANDLE hFileMapping = CreateFileMapping(INVALID_HANDLE_VALUE, &sa,
PAGE_READWRITE, 0, 1024, "MyFileMapping");
```

Параметр *bInitialOwner* определяет начальное состояние мьютекса. Если в нем передается FALSE (что обычно и бывает), объект-мьютекс не принадлежит ни одному из потоков и поэтому находится в свободном состоянии. При этом его идентификатор потока и счетчик рекурсии равны 0. Если же в нем передается TRUE, идентификатор потока, принадлежащий мьютексу, приравнивается идентификатору вызывающего потока, а счетчик рекурсии получает значение 1. Поскольку теперь идентификатор потока отличен от 0, мьютекс изначально находится в занятом состоянии.

Поток получает доступ к разделяемому ресурсу, вызывая одну из Wait-функций и передавая ей описатель мьютекса, который охраняет этот ресурс. Wait-функция проверяет у мьютекса идентификатор потока: если его значение не равно 0, мьютекс свободен; в ином случае оно принимает значение идентификатора вызывающего потока, и этот поток остается планируемым.

Если Wait-функция определяет, что у мьютекса идентификатор потока не равен 0 (мьютекс занят), вызывающий поток переходит в состояние ожидания. Система запоминает это и, когда идентификатор обнуляется, записывает в него идентификатор ждущего потока, а счетчику рекурсии присваивает значение 1, после чего ждущий поток вновь становится планируемым. Все проверки и изменения состояния объекта-мьютекса выполняются на уровне атомарного доступа.

Для мьютексов сделано одно исключение в правилах перехода объектов ядра из одного состояния в другое. Допустим, поток ждет освобождения занятого объекта-мьютекса. В этом случае поток обычно засыпает (переходит в состояние ожидания).

Однако система проверяет, не совпадает ли идентификатор потока, пытающегося захватить мьютекс, с аналогичным идентификатором у мьютекса. Если они совпадают, система по-прежнему выделяет потоку процессорное время, хотя мьютекс все еще

занят. Подобных особенностей в поведении нет ни у каких других объектов ядра в системе. Всякий раз, когда поток захватывает объект-мьютекс, счетчик рекурсии в этом объекте увеличивается на 1. Единственная ситуация, в которой значение счетчика рекурсии может быть больше 1, — поток захватывает один и тот же мьютекс несколько раз, пользуясь упомянутым исключением из общих правил.

HANDLE ***OpenMutex****(DWORD dwDesiredAccess, bool bInheritHandle, LPCTSTR lpName);* - любой процесс может получить свой («процессо-зависимый») описатель существующего объекта «мьютекс», вызвав OpenMutex.

DWORD ***WaitForSingleObject****(HANDLE hHandle, DWORD dwMilliseconds);* -выше описывалось

bool ***ReleaseMutex****(HANDLE hMutex);* - освободить мьютекс для других потоков

bool **CloseHandle**(HANDLE hObject); - зыкрывает открытый объект ядра, не ошибка – вернет значение, отоичное от 0.

Мьютексы и критические секции

Мьютексы и критические секции одинаковы в том, как они влияют на планирование ждущих потоков, но различны по некоторым другим характеристикам. Эти объекты сравниваются в следующей таблице.

Характеристики	Объект-мьютекс	Объект — критическая секция
Быстродействие	Малое	Высокое
Возможность использования за границами процесса	Да	Нет
Объявление	*HANDLE hmtx;*	*CRITICAL_SECTION cs;*
Инициализация	*hmtx = CreateMutex (NULL, FALSE, NULL);*	*InitializeCriticalSection(&cs);*
Очистка	*CloseHandle(hmtx);*	*DeleteCriticalSection(&cs);*
Бесконечное ожидание	*WaitForSingleObject (hmtx, INFINITE);*	*EnterCriticalSection(&cs);*
Ожидание в течение 0 мс	*WaitForSingleObject (hmtx, 0);*	*TryEnterCriticalSection(&cs);*
Ожидание в течение произвольного периода времени	*WaitForSingleObject (hmtx, dwMilliseconds);*	Невозможно
Освобождение	*ReleaseMutex(hmtx);*	*LeaveCriticalSection(&cs);*
Возможность параллельного ожидания других объектов ядра	Да (с помощью *WaitForMultipleObjects* или аналогичной функции)	Нет

– **Семафор** объект ядра, использующийся для учета ресурсов. Семафор имеет внутри счетчик. Этот счетчик снизу ограничен значением 0 (семафор занят) и некоторым верхним значением N. В диапазоне 1..N семафор является свободным. Семафоры можно считать обобщением блокировки на несколько ресурсов.

Объекты ядра «семафор» используются для учета ресурсов. Как и все объекты ядра, они содержат *счетчик числа пользователей*, но, кроме того, поддерживают *два 32-битных значения со знаком: одно определяет максимальное число ресурсов (контролируемое семафором), другое используется как счетчик текущего числа ресурсов.*

Допустим, я разрабатываю серверный процесс, в адресном пространстве которого выделяется буфер для хранения клиентских запросов. Размер этого буфера «зашит» в код программы и рассчитан на хранение максимум пяти клиентских запросов. Если новый клиент пытается связаться с сервером, когда эти пять запросов еще не обработаны, генерируется ошибка, которая сообщает клиенту, что сервер занят и нужно повторить попытку позже.

При инициализации мой серверный процесс создает пул из пяти потоков, каждый из которых готов обрабатывать клиентские запросы по мере их поступления.

Изначально, когда запросов от клиентов еще нет, сервер не разрешает выделять процессорное время каким-либо потокам в пуле. Но как только серверу поступает,
скажем, три клиентских запроса одновременно, три потока в пуле становятся планируемыми, и система начинает выделять им процессорное время. Для слежения за ресурсами и планированием потоков семафор очень удобен. Максимальное число ресурсов задается равным 5, что соответствует размеру буфера. Счетчик текущего числа ресурсов первоначально получает нулевое значение, так как клиенты еще не выдали ни одного запроса. Этот счетчик увеличивается на 1 в момент приема очередного клиентского запроса и на столько же уменьшается, когда запрос передается на обработку одному из серверных потоков в пуле.

Для семафоров определены следующие правила:

- когда счетчик текущего числа ресурсов становится больше 0, семафор переходит в свободное состояние;
- если этот счетчик равен 0, семафор занят;
- система не допускает присвоения отрицательных значений счетчику текущего числа ресурсов;
- счетчик текущего числа ресурсов не может быть больше максимального числа ресурсов.

HANDLE ***CreateSemaphore****(SECURITY_ATTRIBUTES* lpSecurityAttributes, LONG lInitialCount, LONG lMaximumCount, LPCTSTR lpName);* - создание объекта-семафора. Параметр *lMaximumCount* сообщает системе максимальное число ресурсов, обрабатываемое Вашим приложением. Поскольку это 32-битное значение со знаком, предельное число ресурсов может достигать 2 147 483 647. Параметр *lInitialCount* указывает, сколько из этих ресурсов доступно изначально (на данный момент). При инициализации моего серверного процесса клиентских запросов нет, поэтому я вызываю CreateSemaphore так: *HANDLE hSem = CreateSemaphore(NULL, 0, 5, NULL);* счетчик числа пользователей данного объекта ядра равен 1, так как я только что создал этот объект;
Поскольку счетчику текущего числа ресурсов присвоен 0, семафор находится в занятом состоянии. А это значит, что любой поток, ждущий семафор, просто засыпает.

HANDLE ***OpenSemaphore****(DWORD dwDesiredAccess, bool bInheritHandle, LPCTSTR lpName);* - любой процесс может получить свой («процессо-зависимый») описатель существующего объекта «семафор».

DWORD ***WaitForSingleObject****(HANDLE hHandle, DWORD dwMilliseconds);*

Поток получает доступ к ресурсу, вызывая одну из Wait-функций и передавая ей описатель семафора, который охраняет этот ресурс. Wait-функция проверяет у семафора счетчик текущего числа ресурсов: если его значение больше 0 (семафор свободен), уменьшает значение этого счетчика на 1, и вызывающий поток остается планируемым. Очень важно, что семафоры выполняют эту операцию проверки и присвоения на уровне атомарного доступа; иначе говоря, когда Вы запрашиваете у семафора какой-либо ресурс, операционная система проверяет, доступен ли этот ресурс, и, если да, уменьшает счетчик текущего числа ресурсов, не позволяя вмешиваться в эту операцию другому потоку. Только после того как счетчик ресурсов будет уменьшен на 1, доступ к ресурсу сможет запросить другой поток.

Если Wait-функция определяет, что счетчик текущего числа ресурсов равен 0 (семафор занят), система переводит вызывающий поток в состояние ожидания. Когда
другой поток увеличит значение этого счетчика, система вспомнит о ждущем потоке
и снова начнет выделять ему процессорное время (а он, захватив ресурс, уменьшит
значение счетчика на 1).

bool ***ReleaseSemaphore****(HANDLE hSemaphore, LONG lReleaseCount, LONG* lpPreviousCount);* - Поток увеличивает значение счетчика текущего числа ресурсов, вызвав эту функцию.

Она просто складывает величину *lReleaseCount* со значением счетчика текущего числа ресурсов. Обычно в параметре *lReleaseCount* передают 1, но это вовсе не обязательно: я часто передаю в нем значения, равные или большие 2. Функция возвращает исходное значение счетчика ресурсов в **lpPreviousCount*. Если Вас не интересует это значение (а в большинстве программ так оно и есть), передайте в параметре *lpPreviousCount* значение NULL.

Было бы удобнее определять состояние счетчика текущего числа ресурсов, не меняя его значение, но такой функции в Windows нет. Поначалу я думал, что вызовом ReleaseSemaphore с передачей ей во втором параметре нуля можно узнать истинное значение счетчика в переменной типа LONG, на которую указывает параметр plPreviousCount. Но не вышло: функция занесла туда нуль. Я передал во втором параметре заведомо большее число, и — тот же результат. Тогда мне стало ясно: получить значение этого счетчика, не изменив его, невозможно.

bool ***CloseHandle****(HANDLE hObject);*

Ожидаемый таймер – объект ядра, самостоятельно переходящий в свободное состояние в определенное время и/или через определенные промежутки времени.

HANDLE ***CreateWaitableTimer****(SECURITY_ATTRIBUTES* lpSecurityAttributes, BOOL bManualReset, LPCTSTR lpTimerName);* - создание объекта ожидаемый таймер

По аналогии с событиями параметр *fManualReset* определяет тип ожидаемого таймера: со сбросом вручную или с автосбросом. Когда освобождается таймер со сбросом вручную, возобновляется выполнение всех потоков, ожидавших этот объект, а когда в свободное состояние переходит таймер с автосбросом — лишь одного из потоков.

HANDLE ***OpenWaitableTimer****(DWORD dwDesiredAccess,*
bool bInheritHandle, LPCTSTR lpTimerName); - любой процесс может получить свой («процессо-зависимый») описатель существующего объекта «ожидаемый таймер»

Объекты «ожидаемый таймер» всегда создаются в занятом состоянии. Чтобы сообщить таймеру, в какой момент он должен перейти в свободное состояние, вызовите функцию *bool* ***SetWaitableTimer****(HANDLE hTimer, const LARGE_INTEGER* pDueTime, LONG lPeriod, TIMERAPCROUTINE* pfnCompletionRoutine,*
void lpArgToCompletionRoutine, bool fResume);*

hTimer определяет нужный таймер. Следующие два параметра (*pDueTime* и *lPeriod*) используются совместно: первый из них задает, когда таймер должен сработать в первый раз, второй определяет, насколько часто это должно происходить в дальнейшем.

fResume - полезен на компьютерах с поддержкой режима сна. Обычно в нем передают FALSE, и в приведенных ранее фрагментах кода я тоже делал так. Но если Вы, скажем, пишете программу-планировщик, которая позволяет устанавливать таймеры для напоминания о запланированных встречах, то должны передавать в этом параметре TRUE. Когда таймер сработает, машина выйдет из режима сна (если она находилась в нем), и пробудятся потоки, ожидавшие этот таймер. Далее программа сможет проиграть какой-нибудьWAV-файл и вывести окно с напоминанием о предстоящей встрече. Если же Вы передадите FALSE в параметре fResume, объект-таймер перейдет в

свободное состояние, но ожидавшие его потоки не получат процессорное время, пока компьютер не выйдет из режима сна.

У Вас есть возможность создать очередь асинхронных вызовов процедур (asynchronous procedure call, APC) для потока, вызывающего SetWaitableTimer в момент, когда таймер свободен.Обычно при обращении к функции SetWaitableTimer Вы передаете NULL в параметрах *pfnCompletionRoutine* и *pvArgToCompletionRoutine*. В этом случае объект-таймер переходит в свободное состояние в заданное время. Чтобы таймер в этот момент поместил в очередь вызов APC-функции, нужно реализовать данную функцию и передать ее адрес в SetWaitableTimer. APC-функция должна выглядеть примерно так:

VOID APIENTRY TimerAPCRoutine(VOID pvArgToCompletionRoutine,*
DWORD dwTimerLowValue, DWORD dwTimerHighValue) {
// здесь делаем то, что нужно
*}*создается программистом; вызывается системой с помощью **QueueUserAPC** -добавляет (asynchronous procedure call) APC объекта пользовательского режима в APC очередь уазанного потока.

*bool **CancelWaitableTimer**(HANDLE hTimer);* Эта очень простая функция принимает описатель таймера и отменяет его (таймер), после чего тот уже никогда не сработает, — если только Вы не переустановите его повторным вызовом SetWaitableTimer. Кстати, если Вам понадобится перенастроить таймер, то вызывать CancelWaitableTimer перед повторным обращением к SetWaitableTimer не требуется; каждый вызов SetWaitableTimer автоматически отменяет предыдущие настройки перед установкой новых.

*bool **CloseHandle**(HANDLE hObject);*

Оконный таймер – механизм посылки таймерных сообщений через определенные промежутки времени.

*UINT_PTR **SetTimer**(HWND hWnd, UINT_PTR nIDEvent, UINT uElapse, TIMERPROC lpTimerFunc);*

*void CALLBACK **TimerProc**(HWND hwnd, UINT uMsg, UINT_PTR idEvent, DWORD dwTime);*

*bool **KillTimer**(HWND hWnd, UINT_PTR uIDEvent);*

Спин-блокировки представляют собой чрезвычайно низкоуровневое средство синхронизации, предназначенное в первую очередь для применения в многопроцессорной конфигурации с разделяемой памятью. Они обычно реализуются как атомарно устанавливаемое булево значение (истина – блокировка установлена). Аппаратура поддерживает подобные блокировки командами вида "проверить и установить".

При попытке установить спин-блокировку, если она захвачена кем-то другим, как правило, применяется активное ожидание освобождения, с постоянным опросом в цикле состояния блокировки. Естественно, при этом занимается процессор, так что спин-блокировки следует устанавливать только на очень короткое время и их владелец не должен приостанавливать свое выполнение.

Для описываемых блокировок стандарт POSIX-2001 не предусматривает установки с ограниченным ожиданием (накладные расходы по времени на ограничение обычно превысят само время ожидания).

По сравнению с мьютексами спин-блокировки могут иметь то преимущество, что (активное) ожидание и установка не связаны спереключением контекстов, активизацией планировщика и т.п.

Если ожидание оказывается кратким, минимальными оказываются и накладные расходы. Приложение, чувствительное к подобным тонкостям, в каждой конкретной ситуации может выбрать наиболее эффективное средство синхронизации.

Спин-блокировки обслуживаются следующими группами функций (#include <pthread.h>):

инициализация и разрушение спин-блокировок:

```
int pthread_spin_init (pthread_spinlock_t *lock, int pshared);

    int pthread_spin_destroy (pthread_spinlock_t *lock);
```

установка спин-блокировки:

```
    int pthread_spin_lock (pthread_spinlock_t *lock);

    int pthread_spin_trylock (pthread_spinlock_t *lock);
```

снятие спин-блокировки:

```
    int pthread_spin_unlock (pthread_spinlock_t *lock);
```

Применительно к спин-блокировкам было решено не возиться с атрибутными объектами, а единственный поддерживаемый атрибут – признак использования несколькими процессами – задавать при вызове функции pthread_spin_init().

Поскольку между применением мьютексов и спин-блокировок много общего, мы не будем приводить примеры программ, использующихспин-блокировки. Ограничимся маленьким, слегка модифицированным характерным фрагментом реализации функции sigsuspend() в библиотеке glibc .

```
pthread_spin_lock (&ss->lock);
/* Восстановим старую маску */
ss->blocked = oldmask;
/* Проверим ждущие сигналы */
pending = ss->pending & ~ss->blocked;
pthread_spin_unlock (&ss->lock);
```

Спин-блокировка устанавливается на очень короткий участок кода; естественно, она должна быть реализована весьма эффективно, чтобы накладные расходы не оказались чрезмерными.

Критические секции и спин-блокировка

Когда поток пытается войти в критическую секцию, занятую другим потоком, он немедленно приостанавливается. А это значит, что поток переходит из пользовательского режима в режим ядра (на что затрачивается около 1000 тактов процессора). Цена такого перехода чрезвычайно высока. На многопроцессорной машине поток, владеющий ресурсом, может выполняться на другом процессоре и очень быстро освободить ресурс. Тогда появляется вероятность, что ресурс будет освобожден еще до того, как вызывающий поток завершит переход в режим ядра.

Microsoft повысила быстродействие критических секций, включив в них спинблокировку. Теперь, когда Вы вызываете EnterCriticalSection, она выполняет заданное число циклов спин-блокировки, пытаясь получить доступ к ресурсу. И лишь в том случае, когда все попытки заканчиваются неудачно, функция переводит поток в режим ядра, где он будет находиться в состоянии ожидания.

Для использования спин-блокировки в критической секции нужно инициализировать счетчик циклов, вызвав:

BOOL ***InitializeCriticalSectionAndSpinCount****(PCRITICAL_SECTION pcs,DWORD dwSpinCount);*

Как и в InitializeCriticalSection, первый параметр— адрес структуры критической секции. Но во втором параметре, *dwSpinCount*, передается число циклов спин-блокировки при попытках получить доступ к ресурсу до перевода потока в со стояние ожидания. Этот параметр может принимать значения от 0 до 0x00FFFFFF.

Учтите, что на однопроцессорной машине значение параметра *dwSpinCount* игнорируется и считается равным 0. Дело в том, что применение спин-блокировки в такой системе бессмысленно: поток, владеющий ресурсом, не сможет освободить его, пока другой поток «крутится» в циклах спин-блокировки.

Вы можете изменить счетчик циклов спин-блокировки вызовом:

DWORD ***SetCriticalSectionSpinCount****(PCRITICAL_SECTION pcs, DWORD dwSpinCount);*

Могут возникнуть трудности в подборе значения dwSpinCount, но здесь нужно просто поэкспериментировать. Имейте в виду, что для критической секции, стоящей на страже динамической кучи Вашего процесса, этот счетчик равен 4000.

25. Синхронизация потоков в пределах одного процесса ОС Windows. Критическая секция. Операции с критической секцией. Атомарные операции.

Атомарный доступ: семейство Interlocked-функций

Большая часть синхронизации потоков связана с атомарным доступом (atomic access) — монопольным захватом ресурса обращающимся к нему потоком. Возьмем простой пример.

```
// определяем глобальную переменную
long g_x = 0;
DWORD WINAPI ThreadFunc1(PVOID pvParam) {
      g_x++;
      return(0);
}
DWORD WINAPI ThreadFunc2(PVOID pvParam) {
      g_x++;
      return(0);
}
```

Я объявил глобальную переменную g_x и инициализировал ее нулевым значением. Теперь представьте, что я создал два потока: один выполняет ThreadFunc1, другой — ThreadFunc2. Код этих функций идентичен: обе увеличивают значение глобальной переменной g_x на 1. Поэтому Вы, наверное, подумали: когда оба потока завершат свою работу, значение g_x будет равно 2. Так ли это? Может быть. При таком коде заранее сказать, каким будет конечное значение g_x, нельзя. И вот почему. Допустим, компилятор сгенерировал для строки, увеличивающей g_x на 1, следующий код:

```
MOV EAX, [g_x]  ; значение из g_x помещается в регистр
INC EAX  ; значение регистра увеличивается на 1
MOV [g_x], EAX  ; значение из регистра помещается обратно в g_x
```

Вряд ли оба потока будут выполнять этот код в одно и то же время. Если они бу

дут делать это по очереди — сначала один, потом другой, тогда мы получим такую

картину:

```
MOV EAX, [g_x] ; поток 1: в регистр помещается 0
INC EAX ; поток 1: значение регистра увеличивается на 1
MOV [g_x], EAX ; поток 1: значение 1 помещается в g_x
MOV EAX, [g_x] ; поток 2: в регистр помещается 1
INC EAX ; поток 2: значение регистра увеличивается до 2
MOV [g_x], EAX ; поток 2: значение 2 помещается в g_x
```

После выполнения обоих потоков значение g_x будет равно 2. Это то, что мы ожидали: взяв переменную с нулевым значением, дважды увеличили ее на 1 и получили в результате 2. Прекрасно. Но Windows — это среда, которая поддерживает многопоточность и вытесняющую многозадачность. Значит, процессорное время в любой момент может быть отнято у одного потока и передано другому. Тогда код, приведенный мной выше, может выполняться и таким образом:

```
MOV EAX, [g_x]  ; поток 1: в регистр помещается 0
INC EAX  ; поток 1: значение регистра увеличивается на 1
MOV EAX, [g_x] ; поток 2: в регистр помещается 0
INC EAX ; поток 2: значение регистра увеличивается на 1
MOV [g_x], EAX ; поток 2: значение 1 помещается в g_x
MOV [g_x], EAX ; поток 1: значение 1 помещается в g_x
```

А если код будет выполняться именно так, конечное значение g_x окажется равным 1, а не 2.

Решение этой проблемы должно быть простым. Все, что нам нужно, — это способ, гарантирующий приращение значения переменной на уровне атомарного доступа, т. е. без прерывания другими потоками. Семейство Interlocked функций как раз

и дает нам ключ к решению подобных проблем. Большинство разработчиков программного обеспечения недооценивает эти функции, а ведь они невероятно полезны и очень просты для понимания. Все функции из этого семейства манипулируют переменными на уровне атомарного доступа. Взгляните на InterlockedExchangeAdd:

LONG InterlockedExchangeAdd(PLONG plAddend, LONG lIncrement);

Вы вызываете эту функцию, передавая адрес переменной типа LONG и указываете добавляемое значение. InterlockedExchangeAdd гарантирует, что операция будет выполнена атомарно.

Как же работают Interlocked функции? Ответ зависит от того, какую процессорную платформу Вы используете. На компьютерах с процессорами семейства x86 эти функции выдают по шине аппаратный сигнал, не давая другому процессору обратиться по тому же адресу памяти. На платформе Alpha Interlocked функции действуют примерно так:

1. Устанавливают специальный битовый флаг процессора, указывающий, что данный адрес памяти сейчас занят.

2. Считывают значение из памяти в регистр.

3. Изменяют значение в регистре.

4. Если битовый флаг сброшен, повторяют операции, начиная с п. 2. В ином случае значение из регистра помещается обратно в память.(Его может сбросить другой процессор в истеме, пытаясь модифицировать тот же адрес памяти, а это заставляет Interlocked функции вернуться в п. 2.)

Вовсе не обязательно вникать в детали работы этих функций. Вам нужно знать лишь одно: они гарантируют монопольное изменение значений переменных независимо от того, как именно компилятор генерирует код и сколько процессоров установлено в компьютере. Однако Вы должны позаботиться о выравнивании адресов переменных, передаваемых этим функциям, иначе они могут потерпеть неудачу.

Другой важный аспект, связанный с Interlocked функциями, состоит в том, что они выполняются чрезвычайно быстро. Вызов такой функции обычно требует не более 50 тактов процессора, и при этом не происходит перехода из пользовательского режима в режим ядра (а он отнимает не менее 1000 тактов).

Вот еще две функции из этого семейства:

```
LONG InterlockedExchange(PLONG plTarget, LONG lValue);
PVOID InterlockedExchangePointer(PVOID* ppvTarget, PVOID pvValue);
```

InterlockedExchange и InterlockedExchangePointer монопольно заменяют текущее значение переменной типа LONG, адрес которой передается в первом параметре, назначение, передаваемое во втором параметре. В 32 разрядном приложении обе функции работают с 32 разрядными значениями, но в 64 разрядной программе первая оперирует с 32 разрядными значениями, а вторая — с 64 разрядными. Все функции возвращают исходное значение переменной. InterlockedExchange чрезвычайно полезна при реализации спин блокировки (spinlock):

```
// глобальная переменная, используемая как индикатор того, занят ли разделяемый ресурс
BOOL g_fResourceInUse = FALSE;
    . . .
      void Func1() {
// ожидаем доступа к ресурсу
while (InterlockedExchange(&g_fResourceInUse, TRUE) == TRUE)
Sleep(0);
// получаем ресурс в свое распоряжение
. . .
// доступ к ресурсу больше не нужен
InterlockedExchange(&g_fResourceInUse, FALSE);
      }
```

Применяйте эту методику с крайней осторожностью, потому что процессорное время при спин блокировке тратится впустую. Процессору приходится постоянно сравнивать два значения, пока одно из них не будет «волшебным образом» изменено другим потоком. Учтите: этот код подразумевает, что все потоки, использующие спин-блокировку, имеют одинаковый уровень приоритета. К тому же, Вам, наверное, придется отключить динамическое повышение приоритета этих потоков (вызовом SetProcessPriorityBoost или SetThreadPriorityBoost).

Спин блокировка предполагает, что защищенный ресурс не бывает занят надолго. И тогда эффективнее делать так: выполнять цикл, переходить в режим ядра и ждать. Многие разработчики повторяют цикл некоторое число раз (скажем, 4000) и, если ресурс к тому времени не освободился, переводят поток в режим ядра, где он спит, ожидая освобождения ресурса (и не расходуя процессорное время). По такой схеме реализуются критические секции (critical sections).

Последняя пара Interlocked функций выглядит так:

```
PVOID InterlockedCompareExchange(PLONG plDestination, LONG lExchange, LONG lComparand);
PVOID InterlockedCompareExchangePointer(PVOID* ppvDestination, PVOID pvExchange, PVOID
pvComparand);
```

Они выполняют операцию сравнения и присвоения на уровне атомарного доступа. В 32 разрядном приложении обе функции работают с 32 разрядными значения ми, но в 64 разрядном приложении InterlockedCompareExchange используется для 32 разрядных значений, а InterlockedCompareExchangePointer — для 64 разрядных.

Функция сравнивает текущее значение переменной типа LONG (на которую указывает параметр plDestination) со значением, передаваемым в параметре lComparand. Если значения совпадают, *plDestination получает значение параметра lExchange; в ином случае *plDestination остается без изменений. Функция возвращает исходное значение *plDestination.

В Windows есть и другие функции из этого семейства, но ничего нового по сравнению с тем, что мы уже рассмотрели, они не делают. Вот еще две из них:

```
LONG InterlockedIncrement(PLONG plAddend);
LONG InterlockedDecrement(PLONG plAddend);
```

InterlockedExchangeAdd полностью заменяет обе эти устаревшие функции. Новая функция умеет добавлять и вычитать произвольные значения, а функции Interlocked Increment и InterlockedDecrement увеличивают и уменьшают значения только на 1.

Критические секции

Критическая секция (critical section) — это небольшой участок кода, требующий монопольного доступа к каким то общим данным. Она позволяет сделать так, чтобы единовременно только один поток получал доступ к определенному ресурсу. Естественно, система может в любой момент вытеснить Ваш поток и подключить к процессору другой, но ни один из потоков, которым нужен занятый Вами ресурс, не получит процессорное время до тех пор, пока Ваш поток не выйдет за границы критической секции. Пример использования критической секции:

```
const int MAX_TIMES = 1000;
int g_nIndex = 0;
DWORD g_dwTimes[MAX_TIMES];
CRITICAL_SECTION g_cs;
DWORD WINAPI FirstThread(PVOID pvParam) {
   for (BOOL fContinue = TRUE; fContinue; ) {
      EnterCriticalSection(&g_cs);
      if (g_nIndex < MAX_TIMES) {
      g_dwTimes[g_nIndex] = GetTickCount();
       g_nIndex++;
      } else fContinue = FALSE;
      LeaveCriticalSection(&g_cs);
   }
   return(0);
}
DWORD WINAPI SecondThread(PVOID pvParam) {
   for (BOOL fContinue = TRUE; fContinue; ) {
      EnterCriticalSection(&g_cs);
      if (g_nIndex < MAX_TIMES) {
      g_nIndex++;
      g_dwTimes[g_nIndex 1] = GetTickCount();
      } else fContinue = FALSE;
      LeaveCriticalSection(&g_cs);
   }
   return(0);
}
```

Я создал экземпляр структуры данных CRITICAL_SECTION — g_cs, а потом «обернул» весь код, работающий с разделяемым ресурсом (в нашем примере это строки с g_nIndex и g_dwTimes), вызовами EnterCriticalSection и LeaveCriticalSection. Заметьте, что при вызовах этих функций я передаю адрес g_cs.

Запомните несколько важных вещей. Если у Вас есть ресурс, разделяемый несколькими потоками, Вы должны создать экземпляр структуры CRITICAL_SECTION. Структура CRITICAL_SECTION похожа на туалетную кабинку в самолете, а данные, которые нужно защитить, — на унитаз. Туалетная кабинка (критическая секция) в самолете очень маленькая, и единовременно в ней может находиться только один человек (поток), пользующийся унитазом (защищенным ресурсом).

Если у Вас есть ресурсы, всегда используемые вместе, Вы можете поместить их в одну кабинку — единственная структура CRITICAL_SECTION будет охранять их всех. Но если ресурсы не всегда используются вместе (например, потоки 1 и 2 работают с одним ресурсом, а потоки 1 и 3 — с другим), Вам придется создать им по отдельной кабинке, или структуре CRITICAL_SECTION.

Самое сложное — запомнить, что любой участок кода, работающего с разделяемым ресурсом, нужно заключить в вызовы функций EnterCriticalSection и LeaveCriticalSection. Если Вы забудете сделать это хотя бы в одном месте, ресурс может быть поврежден. Так, если в FirstThread убрать вызовы EnterCriticalSection и LeaveCriticalSection, содержимое переменных g_nIndex и g_dwTimes станет

некорректным — даже несмотря на то что в SecondThread функции EnterCriticalSection и LeaveCriticalSection вызываются правильно.

Обычно структуры CRITICAL_SECTION создаются как глобальные переменные, доступные всем потокам процесса. Но ничто не мешает нам создавать их как локальные переменные или переменные, динамически размещаемые в куче. Есть только два условия, которые надо соблюдать. Во первых, все потоки, которым может понадобиться ресурс, должны знать адрес структуры CRITICAL_SECTION, которая защищает этот ресурс. Вы можете получить ее адрес, используя любой из существующих механизмов. Во вторых, элементы структуры CRITICAL_SECTION следует инициализировать до обращения какого либо потока к защищенному ресурсу. Структура инициализируется вызовом:

```
VOID InitializeCriticalSection(PCRITICAL_SECTION pcs);
```

Эта функция инициализирует элементы структуры CRITICAL_SECTION, на которую указывает параметр pcs. Поскольку вся работа данной функции заключается в инициализации нескольких переменных членов, она не дает сбоев и поэтому ничего не возвращает (void). InitializeCriticalSection должна быть вызвана до того, как один из потоков обратится к EnterCriticalSection. В документации Platform SDK недвусмысленно сказано, что попытка воспользоваться неинициализированной критической секцией даст непредсказуемые результаты.

Если Вы знаете, что структура CRITICAL_SECTION больше не понадобится ни одному потоку, удалите ее, вызвав DeleteCriticalSection:

```
VOID DeleteCriticalSection(PCRITICAL_SECTION pcs);
```

Она сбрасывает все переменные члены внутри этой структуры. Естественно, нельзя удалять критическую секцию в тот момент, когда ею все еще пользуется какой-либо поток.

Участок кода, работающий с разделяемым ресурсом, предваряется вызовом:

```
VOID EnterCriticalSection(PCRITICAL_SECTION pcs);
```

Первое, что делает EnterCriticalSection, — исследует значения элементов структуры CRITICAL_SECTION. Если ресурс занят, в них содержатся сведения о том, какой поток пользуется ресурсом. EnterCriticalSection выполняет следующие действия.

· Если ресурс свободен, EnterCriticalSection модифицирует элементы структуры, указывая, что вызывающий поток занимает ресурс, после чего немедленно возвращает управление, и поток продолжает свою работу (получив доступ к ресурсу).

· Если значения элементов структуры свидетельствуют, что ресурс уже захвачен вызывающим потоком, EnterCriticalSection обновляет их, отмечая тем самым, сколько раз подряд этот поток захватил ресурс, и немедленно возвращает управление. Такая ситуация бывает нечасто — лишь тогда, когда поток два раза подряд вызывает EnterCriticalSection без промежуточного вызова LeaveCriticalSection.

· Если значения элементов структуры указывают на то, что ресурс занят другим потоком, EnterCriticalSection переводит вызывающий поток в режим ожидания. Это потрясающее свойство критических секций: поток, пребывая в ожидании, не тратит ни кванта процессорного времени! Система запоминает, что данный поток хочет получить доступ к ресурсу, и — как только поток, занимавший этот ресурс, вызывает LeaveCriticalSection — вновь начинает выделять нашему потоку

процессорное время. При этом она передает ему ресурс, автоматически обновляя элементы структуры CRITICAL_SECTION.

Внутреннее устройство EnterCriticalSection не слишком сложно; она выполняет лишь несколько простых операций. Чем она действительно ценна, так это способностью выполнять их на уровне атомарного доступа. Даже если два потока на многопроцессорной машине одновременно вызовут EnterCriticalSection, функция все равно корректно справится со своей задачей: один поток получит ресурс, другой — перейдет в ожидание.

Поток, переведенный EnterCriticalSection в ожидание, может надолго лишиться доступа к процессору, а в плохо написанной программе — даже вообще не получить его. Когда именно так и происходит, говорят, что поток «голодает».

В действительности потоки, ожидающие освобождения критической секции, никогда не блокируются «навечно». EnterCriticalSection устроена так, что по истечении определенного времени, генерирует исключение.

Вместо EnterCriticalSection Вы можете воспользоваться:
```
BOOL TryEnterCriticalSection(PCRITICAL_SECTION pcs);
```

Эта функция никогда не приостанавливает выполнение вызывающего потока. Но возвращаемое ею значение сообщает, получил ли этот поток доступ к ресурсу. Если при ее вызове указанный ресурс занят другим потоком, она возвращает FALSE.

TryEnterCriticalSection позволяет потоку быстро проверить, доступен ли ресурс, и, если нет, заняться чем нибудь другим. Если функция возвращает TRUE, значит, она обновила элементы структуры CRITICAL_SECTION так, чтобы они сообщали о захвате ресурса вызывающим потоком. Отсюда следует, что для каждого вызова функции TryEnterCriticalSection, где она возвращает TRUE, надо предусмотреть парный вызов LeaveCriticalSection.

В конце участка кода, использующего разделяемый ресурс, должен присутствовать вызов:
```
VOID LeaveCriticalSection(PCRITICAL_SECTION pcs);
```

Эта функция просматривает элементы структуры CRITICAL_SECTION и уменьшает счетчик числа захватов ресурса вызывающим потоком на 1. Если его значение больше 0, LeaveCriticalSection ничего не делает и просто возвращает управление. Если значение счетчика достигло 0, LeaveCriticalSection сначала выясняет, есть ли в системе другие потоки, ждущие данный ресурс в вызове EnterCriticalSection. Если есть хотя бы один такой поток, функция настраивает значения элементов структуры, чтобы они сигнализировали о занятости ресурса, и отдает его одному из ждущих потоков (поток выбирается «по справедливости»). Если же ресурс никому не нужен, LeaveCriticalSection соответственно сбрасывает элементы структуры. Как и EnterCriticalSection, функция LeaveCriticalSection выполняет все действия на уровне атомарного доступа. Однако LeaveCriticalSection никогда не приостанавливает поток, а управление возвращает немедленно.

26. Синхронизация потоков в пределах одного процесса ОС Windows. Ожидаемое условие (монитор Хора). Операции с ожидаемым условием. Пример использования ожидаемого условия для синхронизации потоков.

Ожидаемое условие – механизм синхронизации, позволяющий потокам дождаться выполнения некоторого (сложного) условия. Состоит из критической секции и переменной условия. Условные переменные обеспечивают присущую Windows реализацию синхронизации набора потоков, ожидающих конкретного результата проверки условия. Хотя эта операция была возможна и с применением других методов синхронизации в пользовательском режиме, атомарного механизма для проверки результата проверки условия и для начала ожидания изменения этого результата не существовало. Это потребовало использования в отношении таких фрагментов кода этой дополнительной синхронизации. Поток пользовательского режима инициализирует условную переменную путем вызова функции InitializeConditionVariable для установки ее исходного состояния. Когда ему нужно инициировать ожидание, связанное с этой переменной, он вызывает функцию SleepConditionVariableCS, которая использует критический раздел (который поток должен инициализировать) для ожидания изменений переменной. После изменения переменной устанавливающий поток должен использовать функцию WakeConditionVariable (или функцию WakeAllConditionVariable). В результате этого вызова освобождается критический раздел либо одного, либо всех ожидающих потоков, в зависимости от того, которая из этих функция была использована. До условных переменных для отправки сигнала об изменении переменной, например состояния рабочей очереди, часто использовалось либо уведомительное событие, либо синхронизирующее событие (в Windows API они называются авто матическим перезапуском — auto-reset, или ручным перезапуском — manual-reset). Ожидание изменения требует получения, а затем освобождения критического раздела, сопровождаемого ожиданием события. После ожидания критический раздел должен быть получен повторно. Во время этих серий получений и освобождений у потока может быть переключен контекст, вызывая проблемы, если один из потоков называется PulseEvent. При использовании условных переменных получение критического раздела может быть поддержано приложением во время вызова функции SleepConditionVariableCS, и он может быть освобожден только после выполнения работы. Это делает код записи рабочей очереди (и подобных ей реализаций) более простым и предсказуемым. Внутри системы условные переменные могут рассматриваться как порт существующих алгоритмов пуш-блокировок, имеющихся в режиме ядра, с дополнительным усложнением в виде получения и освобождения критических разделов в API-функции SleepConditionVariableCS. Условные переменные по размеру равны указателям (точно так же, как и пуш-блокировки), избегают использования диспетчера, автоматически оптимизируют во время операций ожидания список ожиданий и защищают от сопровождений блокировки. Кроме того, условные переменные полностью используют события с ключом, а не обычный объект события, который бы использовался разработчиками по своему усмотрению, что еще больше оптимизирует код даже в случаях возникновения конкуренции.

- `void InitializeConditionVariable(CONDITION_VARIABLE* CondVariable);`
- `bool SleepConditionVariableCS(CONDITION_VARIABLE* CondVariable, CRITICAL_SECTION* CriticalSection, DWORD dwMilliseconds);`
- `bool SleepConditionVariableSRW(CONDITION_VARIABLE* CondVariable, SRWLOCK* SRWLock, DWORD dwMilliseconds, ULONG Flags);`
- `void WakeConditionVariable(CONDITION_VARIABLE* CondVariable);`
- `void WakeAllConditionVariable(CONDITION_VARIABLE* CondVariable);`

Пример использования ожидаемого условия:

```
// CRITICAL_SECTION criticalSection;
```

```
// CONDITION_VARIABLE conditionVariable;
EnterCriticalSection(&criticalSection);
try{
            while (DataDoesntSatisfyCondition()) // функция программиста
               SleepConditionVariableCS(&conditionVariable, &criticalSection,
INFINITE);
}
catch (...){
            LeaveCriticalSection(&criticalSection);
            throw;
}
LeaveCriticalSection(&criticalSection);
       Пример использования ожидаемого условия:
// CRITICAL_SECTION criticalSection;
// CONDITION_VARIABLE conditionVariable;
EnterCriticalSection(&criticalSection);
try {
            ChangeData(); // процедура программиста
            WakeAllConditionVariableCS(&conditionVariable);
}
catch (...){
            LeaveCriticalSection(&criticalSection);
            throw;
}
LeaveCriticalSection(&criticalSection);
```

События — самая примитивная разновидность объектов ядра. Они содержат счетчик числа пользователей (как и все объекты ядра) и две булевы переменные: одна сообщает тип данного объекта-события, другая — его состояние (свободен или занят).

События просто уведомляют об окончании какой-либо операции. Объекты-события бывают двух типов: со сбросом вручную (manual-reset events) и с автосбросом (auto-reset events). Первые позволяют возобновлять выполнение сразу нескольких ждущих потоков, вторые — только одного.

Объекты-события обычно используют в том случае, когда какой-то поток выполняет инициализацию, а затем сигнализирует другому потоку, что тот может продолжить работу. Инициализирующий поток переводит объект «событие» в занятое состояние и приступает к своим операциям. Закончив, он сбрасывает событие в свободное
состояние. Тогда другой поток, который ждал перехода события в свободное состояние, пробуждается и вновь становится планируемым.

Пример:

Выполнение некоторым поток действий в контексте другого потока.

```
HANDLE CreateEvent(SECURITY_ATTRIBUTES* lpSecurityAttributes,
bool bManualReset, bool bInitialState, LPCTSTR lpName);
```

Параметр *fManualReset* (булева переменная) сообщает системе, хотите Вы создать событие со сбросом вручную (TRUE) или с автосбросом (FALSE). Параметр *fInitialState* определяет начальное состояние события — свободное (TRUE) или занятое (FALSE).

После того как система создает объект-событие, CreateEvent возвращает описатель события, специфичный для конкретного процесса. Потоки из других процессов могут получить доступ к этому объекту:

1) вызовом CreateEvent с тем же параметром lpName;

2) наследованием описателя;

3) применением функции DuplicateHandle;

4) вызовом . *HANDLE **OpenEvent**(DWORD fdwAccess, BOOL fInherit, PCTSTR pszName);* с передачей в параметре pszName имени, совпадающего с указанным в аналогичном параметре функции CreateEvent.

Ненужный объект ядра «событие» следует, как всегда, закрыть вызовом *bool **CloseHandle**(HANDLE hObject);*.

Создав событие, Вы можете напрямую управлять его состоянием. Чтобы перевести его в свободное состояние, Вы вызываете:

*BOOL **SetEvent**(HANDLE hEvent);*

А чтобы поменять его на занятое:

*BOOL **ResetEvent**(HANDLE hEvent);*

Для событий с автосбросом действует следующее правило. Когда его ожидание потоком успешно завершается, этот объект автоматически сбрасывается в занятое состояние. Отсюда и произошло название таких объектов-событий. Для этого объекта обычно не требуется вызывать ResetEvent, поскольку система сама восстанавливает его состояние. А для событий со сбросом вручную никаких побочных эффектов успешного ожидания не предусмотрено.

Закончив свою работу с данными, поток вызывает SetEvent, которая разрешает системе возобновить выполнение следующего из двух ждущих потоков. И опять мы не знаем, какой поток выберет система, но так или иначе кто-то из них получит монопольный доступ к тому же блоку памяти. Когда и этот поток закончит свою работу, он тоже вызовет SetEvent, после чего с блоком памяти сможет монопольно оперировать третий, последний поток. Обратите внимание, что использование события с автосбросом снимает проблему с доступом вторичных потоков к памяти как для чтения, так и для записи; Вам больше не нужно ограничивать их доступ только чтением.

bool **PulseEvent**(HANDLE hEvent); –освобождает событие и тут же переводит его обратно в занятое состояние; ее вызов равнозначен последовательному вызову SetEvent и ResetEvent. Если Вы вызываете PulseEvent для события со сбросом вручную, любые потоки, ждущие этот объект, становятся планируемыми. При вызове этой функции применительно к событию с автосбросом пробуждается только один из ждущих потоков. А если ни один из потоков не ждет объект-событие, вызов функции не дает никакого эффекта. Абсолютно неясно, какой из потоков заметит этот импульс и станет планируемым.

В следующей таблице суммируются сведения о различных объектах ядра применительно к синхронизации потоков.

Объект	Находится в занятом состоянии, когда:	Переходит в свободное состояние, когда:	Побочный эффект успешного ожидания
Процесс	процесс еще активен	процесс завершается (*ExitProcess*, *TerminateProcess*)	Нет
Поток	поток еще активен	поток завершается (*ExitThread*, *TerminateThread*)	Нет
Задание	время, выделенное заданию, еще не истекло	время, выделенное заданию, истекло	Нет
Файл	выдан запрос на ввод-вывод	завершено выполнение запроса на ввод-вывод	Нет
Консольный ввод	ввода нет	ввод есть	Нет
Уведомление об изменении файла	в файловой системе нет изменений	файловая система обнаруживает изменения	Сбрасывается в исходное состояние
Событие с автосбросом	вызывается *ResetEvent*, *PulseEvent* или ожидание успешно завершилось	вызывается *SetEvent* или *PulseEvent*	Сбрасывается в исходное состояние
Событие со сбросом вручную	вызывается *ResetEvent* или *PulseEvent*	вызывается *SetEvent* или *PulseEvent*	Нет
Ожидаемый таймер с автосбросом	вызывается *CancelWaitableTimer* или ожидание успешно завершилось	наступает время срабатывания (*SetWaitableTimer*)	Сбрасывается в исходное состояние
Ожидаемый таймер со сбросом вручную	вызывается *CancelWaitableTimer*	наступает время срабатывания (*SetWaitableTimer*)	Нет
Семафор	ожидание успешно завершилось	счетчик > 0 (*ReleaseSemaphore*)	Счетчик уменьшается на 1
Мьютекс	ожидание успешно завершилось	поток освобождает мьютекс (*ReleaseMutex*)	Передается потоку во владение
Критическая секция (пользовательского режима)	ожидание успешно завершилось (*(Try)EnterCriticalSection*)	поток освобождает критическую секцию (*LeaveCriticalSection*)	Передается потоку во владение

Interlocked-функции (пользовательского режима) никогда не приводят к исключению потока из числа планируемых; они лишь изменяют какое-то значение и тут же возвращают управление.

27. Синхронизация потоков разных процессов с помощью объектов ядра. Понятие свободного и занятого состояния объекта ядра. Процедуры ожидания освобождения объекта ядра. Перевод объекта ядра в свободное состояние. Объекты синхронизации: блокировки, семафоры, события.

Почти все объекты ядра годятся и для решения задач синхронизации. В случае синхронизации потоков о каждом из этих объектов говорят, что он находится либо в свободном (signaled state), либо в занятом состоянии (nonsignaled state). Переход из одного состояния в другое осуществляется по правилам, определенным Microsoft для каждого из объектов ядра. Так, объекты ядра «процесс» сразу после создания всегда находятся в занятом состоянии. В момент завершения процесса операционная система автоматически освобождает его объект ядра «процесс», и он навсегда остается в этом состоянии.

Объект ядра «процесс» пребывает в занятом состоянии, пока выполняется сопоставленный с ним процесс, и переходит в свободное состояние, когда процесс завершается. Внутри этого объекта поддерживается булева переменная, которая при создании объекта инициализируется как FALSE («занято»). По окончании работы процесса операционная система меняет значение этой переменной на TRUE, сообщая тем самым, что объект свободен.

Если Вы пишете код, проверяющий, выполняется ли процесс в данный момент, Вам нужно лишь вызвать функцию, которая просит операционную систему проверить значение булевой переменной, принадлежащей объекту ядра «процесс». Тут нет ничего сложного. Вы можете также сообщить системе, чтобы та перевела Ваш поток в состояние ожидания и автоматически пробудила его при изменении значения булевой переменной с FALSE на TRUE. Тогда появляется возможность заставить поток в родительском процессе, ожидающий завершения дочернего процесса, просто заснуть до освобождения объекта ядра, идентифицирующего дочерний процесс. В дальнейшем Вы увидите, что в Windows есть ряд функций, позволяющих легко решать эту задачу. Я только что описал правила, определенные Microsoft для объекта ядра «процесс». Точно такие же правила распространяются и на объекты ядра «поток». Они тоже сразу после создания находятся в занятом состоянии. Когда поток завершается, операционная система автоматически переводит объект ядра «поток» в свободное состояние. Таким образом, используя те же приемы, Вы можете определить, выполняется ли в данный момент тот или иной поток. Как и объект ядра «процесс», объект ядра «поток» никогда не возвращается в занятое состояние.

Следующие объекты ядра бывают в свободном или занятом состоянии:

- процессы
- уведомления об изменении файлов
- потоки
- события
- задания
- ожидаемые таймеры
- файлы
- семафоры
- консольный ввод
- мьютексы

Потоки могут засыпать и в таком состоянии ждать освобождения какого-либо объекта. Правила, по которым объект переходит в свободное или занятое состояние, зависят от типа этого объекта.

Wait функции позволяют потоку в любой момент приостановиться и ждать освобождения какого-либо объекта ядра. Из всего семейства этих функций чаще всего используется WaitForSingleObject:

```
DWORD WaitForSingleObject(HANDLE hObject, DWORD dwMilliseconds);
```

Когда поток вызывает эту функцию, первый параметр, hObject, идентифицирует объект ядра, поддерживающий состояния «свободен занят». (То есть любой объект, упомянутый в списке из предыдущего раздела.) Второй параметр, dwMilliseconds, указывает, сколько времени (в миллисекундах) поток готов ждать освобождения объекта. Следующий вызов сообщает системе, что поток будет ждать до тех пор, пока не завершится процесс, идентифицируемый описателем hProcess:
```
WaitForSingleObject(hProcess, INFINITE);
```
В данном случае константа INFINITE, передаваемая во втором параметре, подсказывает системе, что вызывающий поток готов ждать этого события хоть целую вечность. Именно эта константа обычно и передается функции WaitForSingleObject, но Вы можете указать любое значение в миллисекундах. Кстати, константа INFINITE определена как 0xFFFFFFFF (или –1). Разумеется, передача INFINITE не всегда безопасна. Если объект так и не перейдет в свободное состояние, вызывающий поток никогда не проснется; одно утешение: тратить драгоценное процессорное время он при этом не будет.
Вот пример, иллюстрирующий, как вызывать WaitForSingleObject со значением таймаута, отличным от INFINITE:
```
DWORD dw = WaitForSingleObject(hProcess, 5000);
switch (dw) {
      case WAIT_OBJECT_0:
      // процесс завершается
      break;
      case WAIT_TIMEOUT:
      // процесс не завершился в течение 5000 мс
      break;
      case WAIT_FAILED:
      // неправильный вызов функции (неверный описатель?)
      break;
}
```
Данный код сообщает системе, что вызывающий поток не должен получать процессорное время, пока не завершится указанный процесс или не пройдет 5000 мс (в зависимости от того, что случится раньше). Поэтому функция вернет управление либо до истечения 5000 мс, если процесс завершится, либо примерно через 5000 мс, если процесс к тому времени не закончит свою работу. Заметьте, что в параметре dwMilliseconds можно передать 0, и тогда WaitForSingleObject немедленно вернет управление. Возвращаемое значение функции WaitForSingleObject указывает, почему вызывающий поток снова стал планируемым. Если функция возвращает WAIT_OBJECT_0, объект свободен, а если WAIT_TIMEOUT — заданное время ожидания (таймаут) истекло. При передаче неверного параметра (например, недопустимого описателя) WaitForSingleObject возвращает WAIT_FAILED. Чтобы выяснить конкретную причину ошибки, вызовите функцию GetLastError. Функция WaitForMultipleObjects аналогична WaitForSingleObject с тем исключением, что позволяет ждать освобождения сразу нескольких объектов или какого то одного из списка объектов:
```
DWORD WaitForMultipleObjects(
DWORD dwCount,
CONST HANDLE* phObjects,
BOOL fWaitAll,
DWORD dwMilliseconds);
```
Параметр dwCount определяет количество интересующих Вас объектов ядра. Его значение должно быть в пределах от 1 до MAXIMUM_WAIT_OBJECTS (в заголовочных файлах Windows оно определено как 64). Параметр phObjects — это указатель на массив описателей объектов ядра. WaitForMultipleObjects приостанавливает поток и заставляет его ждать освобождения либо всех заданных объектов ядра, либо одного из них. Параметр fWaitAll как раз и определяет, чего именно Вы хотите от функции. Если он равен TRUE, функция не даст потоку возобновить свою работу, пока не освободятся все объекты. Параметр dwMilliseconds идентичен одноименному параметру функции

WaitForSingleObject. Если Вы указываете конкретное время ожидания, то по его истечении функция в любом случае возвращает управление. И опять же, в этом параметре обычно передают INFINITE (будьте внимательны при написании кода, чтобы не создать ситуацию взаимной блокировки). Возвращаемое значение функции WaitForMultipleObjects сообщает, почему возобновилось выполнение вызвавшего ее потока. Значения WAIT_FAILED и WAIT_TIMEOUT никаких пояснений не требуют. Если Вы передали TRUE в параметре fWaitAll и все объекты перешли в свободное состояние, функция возвращает значение WAIT_OBJECT_0. Если же fWaitAll приравнен FALSE, она возвращает управление, как только освобождается любой из объектов. Вы, по видимому, захотите выяснить, какой именно объект освободился. В этом случае возвращается значение от WAIT_OBJECT_0 до WAIT_OBJECT_0 + dwCount – 1. Иначе говоря, если возвращаемое значение не равно WAIT_TIMEOUT или WAIT_FAILED, вычтите из него значение WAIT_OBJECT_0, и Вы получите индекс в массиве описателей, на который указывает второй параметр функции WaitForMultipleObjects. Индекс подскажет Вам, какой объект перешел в незанятое состояние. Поясню сказанное на примере.

```
HANDLE h[3];
h[0] = hProcess1;
h[1] = hProcess2;
h[2] = hProcess3;
DWORD dw = WaitForMultipleObjects(3, h, FALSE, 5000);
switch (dw) {
      case WAIT_FAILED:
      // неправильный вызов функции (неверный описатель?)
      break;
      case WAIT_TIMEOUT:
      // ни один из объектов не освободился в течение 5000 мс
      break;
      case WAIT_OBJECT_0 + 0:
      // завершился процесс, идентифицируемый h[0], т. е. описателем (hProcess1)
      break;
      case WAIT_OBJECT_0 + 1:
      // завершился процесс, идентифицируемый h[1], т. е. описателем (hProcess2)
      break;
      case WAIT_OBJECT_0 + 2:
      // завершился процесс, идентифицируемый h[2], т. е. описателем (hProcess3)
      break;
}
```

Если Вы передаете FALSE в параметре fWaitAll, функция WaitForMultipleObjects сканирует массив описателей (начиная с нулевого элемента), и первый же освободившийся объект прерывает ожидание. Это может привести к нежелательным последствиям. Например, Ваш поток ждет завершения трех дочерних процессов; при этом Вы передали функции массив с их описателями. Если завершается процесс, описатель которого находится в нулевом элементе массива, WaitForMultipleObjects возвращает управление. Теперь поток может сделать то, что ему нужно, и вновь вызвать эту функцию, ожидая завершения другого процесса. Если поток передаст те же три описателя, функция немедленно вернет управление, и Вы снова получите значение WAIT_OBJECT_0. Таким образом, пока Вы не удалите описатели тех объектов, об освобождении которых функция уже сообщила Вам, код будет работать некорректно.

Перевод в свободное(сигнальное) состояние.

Тип объекта	Устанавливается в сигнальное состояние, когда	Влияние на ожидающие потоки
Процесс	Завершается	Освобождаются все потоки

	последний поток	
Поток	Завершается поток	Освобождаются все потоки
Событие(уведомительного типа)	Поток устанавливает событие	Освобождаются все потоки
Событие(синхронизирующего типа)	Поток устанавливает событие	Один поток освобождается и может получить повышение приоритета; объект события сбрасывается
Шлюз(блокирующего типа)	Поток сообщает о шлюзе	Освобождается и получает более высокий приоритет первый ожидающий поток
Шлюз (сигнализирующего типа)	Поток сообщает о типе	Освобождается первый ожидающий поток
Событие, снабженное ключом	Поток устанавливает событие с помощью ключа	Освобождается поток, ожидающий ключа и относящийся к тому же процессу, что и сигнализирующий поток
Семафор	Счетчик семафора уменьшается на 1	Освобождается один поток
Таймер (уведомительного типа)	Настало установленное время, или истек интервал времени	Освобождаются все потоки
Таймер (синхронизирующего типа)	Настало установленное время, или истек интервал времени	Освобождается один поток
Мьютекс	Поток освобождает мьютекс	Освобождается один поток, который становится владельцем мьютекса
Очередь	Элемент помещается в очередь	Освобождается один поток

Блокировка – mutex (**mut**ually **ex**clusive), бинарный семафор. Используется для обеспечения монопольного доступа к некоторому ресурсу со стороны нескольких потоков (различных процессов).

- `HANDLE` **`CreateMutex`**`(SECURITY_ATTRIBUTES* lpMutexAttributes, bool bInitialOwner, LPCTSTR lpName);`
- `HANDLE` **`OpenMutex`**`(DWORD dwDesiredAccess, bool bInheritHandle, LPCTSTR lpName);`
- `DWORD` **`WaitForSingleObject`**`(HANDLE hHandle, DWORD dwMilliseconds);`
- `bool` **`ReleaseMutex`**`(HANDLE hMutex);`
- `bool` **`CloseHandle`**`(HANDLE hObject);`

Семафор – объект ядра, использующийся для учета ресурсов. Семафор имеет внутри счетчик. Этот счетчик снизу ограничен значением 0 (семафор занят) и некоторым верхним значением N. В диапазоне 1..N семафор является свободным. Семафоры можно считать обобщением блокировки на несколько ресурсов.

- HANDLE **CreateSemaphore**(SECURITY_ATTRIBUTES* lpSecurityAttributes, LONG lInitialCount, LONG lMaximumCount, LPCTSTR lpName);
- HANDLE **OpenSemaphore**(DWORD dwDesiredAccess, bool bInheritHandle, LPCTSTR lpName);
- DWORD **WaitForSingleObject**(HANDLE hHandle, DWORD dwMilliseconds);
- bool **ReleaseSemaphore**(HANDLE hSemaphore, LONG lReleaseCount, LONG* lpPreviousCount);
- bool **CloseHandle**(HANDLE hObject);

Событие – примитивный объект синхронизации, применяемый для уведомления одного или нескольких потоков об окончании какой-либо операции. Событие бывает двух типов:

Событие со сбросом вручную – manual-reset event;

Событие с автосбросом – auto-reset event.

Пример:

Выполнение некоторым поток действий в контексте другого потока.

Функции:

- HANDLE **CreateEvent**(SECURITY_ATTRIBUTES* lpSecurityAttributes,
- bool bManualReset, bool bInitialState, LPCTSTR lpName); **OpenEvent**.
- bool **SetEvent**(HANDLE hEvent); bool **ResetEvent**(HANDLE hEvent);
- bool **PulseEvent**(HANDLE hEvent); – если это событие со сбросом вручную, то запускаются все ожидающие потоки; если это событие с автосбросом, то запускается лишь один из ожидающих потоков.
- bool **CloseHandle**(HANDLE hObject);

28. Синхронизация потоков разных процессов с помощью объектов ядра. Понятие свободного и занятого состояния объекта ядра. Процедуры ожидания освобождения объекта ядра. Ожидаемые таймеры. Оконные таймеры.

Первую часть вопроса читаем в вопросе 27.

Ожидаемые таймеры (waitable timers) — это объекты ядра, которые самостоятельно переходят в свободное состояние в определенное время или через регулярные промежутки времени. Чтобы создать ожидаемый таймер, достаточно вызвать функцию CreateWaitableTimer:

HANDLE CreateWaitableTimer(PSECURITY_ATTRIBUTES psa, BOOL fManualReset, PCTSTR pszName);

О параметрах psa и pszName я уже рассказывал в главе 3. Разумеется, любой процесс может получить свой («процессо зависимый») описатель существующего объекта «ожидаемый таймер», вызвав OpenWaitableTimer:

HANDLE OpenWaitableTimer(DWORD dwDesiredAccess, BOOL bInheritHandle, PCTSTR pszName);

По аналогии с событиями параметр fManualReset определяет тип ожидаемого таймера: со сбросом вручную или с автосбросом. Когда освобождается таймер со сбросом вручную, возобновляется выполнение всех потоков, ожидавших этот объект, а когда в свободное состояние переходит таймер с автосбросом — лишь одного из потоков.

Объекты «ожидаемый таймер» всегда создаются в занятом состоянии. Чтобы сообщить таймеру, в какой момент он должен перейти в свободное состояние, вызовите функцию SetWaitableTimer:

BOOL SetWaitableTimer(HANDLE hTimer, const LARGE_INTEGER *pDueTime, LONG lPeriod, PTIMERAPCROUTINE pfnCompletionRoutine, PVOID pvArgToCompletionRoutine, BOOL fResume);

Эта функция принимает несколько параметров, в которых легко запутаться. Очевидно, что hTimer определяет нужный таймер. Следующие два параметра (pDueTime и lPeriod) используются совместно: первый из них задает, когда таймер должен сработать в первый раз, второй определяет, насколько часто это должно происходить в дальнейшем.

Следующий код демонстрирует, как установить таймер на первое срабатывание через 5 секунд после вызова SetWaitableTimer:

```
// объявляем свои локальные переменные
HANDLE hTimer;
LARGE_INTEGER li;
// создаем таймер с автосбросом
hTimer = CreateWaitableTimer(NULL, FALSE, NULL);
// таймер должен сработать через 5 секунд после вызова SetWaitableTimer;
// задаем время в интервалах по 100 нс
const int nTimerUnitsPerSecond = 10000000;
// делаем полученное значение отрицательным, чтобы SetWaitableTimer
// знала: нам нужно относительное, а не абсолютное время
li.QuadPart = (5 * nTimerUnitsPerSecond);
// устанавливаем таймер (он срабатывает сначала через 5 секунд,
// а потом через каждые 6 часов)
SetWaitableTimer(hTimer, &li, 6 * 60 * 60 * 1000, NULL, NULL, FALSE);
```

Обычно нужно, чтобы таймер сработал только раз — через определенное (абсолютное или относительное) время перешел в свободное состояние и уже больше никогда не срабатывал. Для этого достаточно передать 0 в параметре lPeriod. Затем можно либо вызвать CloseHandle, чтобы закрыть таймер, либо перенастроить таймер повторным вызовом SetWaitableTimer с другими параметрами. И о последнем параметре функции SetWaitableTimer — fResume. Он полезен на компьютерах с поддержкой режима сна. Обычно в нем передают FALSE, и в приведенных ранее фрагментах кода я тоже делал так. Но если Вы, скажем, пишете программу планировщик, которая

позволяет устанавливать таймеры для напоминания о запланированных встречах, то должны передавать в этом параметре TRUE. Когда таймер сработает, машина выйдет из режима сна (если она находилась в нем), и пробудятся потоки, ожидавшие этот таймер. Далее программа сможет проиграть какой-нибудь WAV файл и вывести окно с напоминанием о предстоящей встрече. Если же Вы передадите FALSE в параметре fResume, объект таймер перейдет в свободное состояние, но ожидавшие его потоки не получат процессорное время, пока компьютер не выйдет из режима сна.
Рассмотрение ожидаемых таймеров было бы неполным, пропусти мы функцию CancelWaitableTimer:
BOOL CancelWaitableTimer(HANDLE hTimer);
Эта очень простая функция принимает описатель таймера и отменяет его (таймер), после чего тот уже никогда не сработает, — если только Вы не переустановите его повторным вызовом SetWaitableTimer. Кстати, если Вам понадобится перенастроить таймер, то вызывать CancelWaitableTimer перед повторным обращением к SetWaitableTimer не требуется; каждый вызов SetWaitableTimer автоматически отменяет предыдущие настройки перед установкой новых.

29. Структура системного программного интерфейса ОС Windows (Native API). Nt-функции и Zw-функции в пользовательском режиме и режиме ядра ОС Windows.

Все функции ядра Windows, доступные пользовательским приложениям, экспортируются библиотекой NtDll.dll. Системные функции называются **Native API**.
Native API - в основном недокументированный интерфейс программирования приложений (API), предназначенный для внутреннего использования в операционных системах семейства Windows NT, выпущенных Microsoft. В основном он используется во время загрузки системы, когда другие компоненты Windows недоступны, а также функциями системных библиотек (например, kernel32.dll), которые реализуют функциональность Windows API. Точкой входа программ, использующих Native API является функция DriverEntry(), так же как и в драйверах устройств Windows. В то же время, в отличие от драйверов, программы, использующие Native API, выполняются в третьем кольце защиты, так же как и обычные приложения Windows. Большая часть вызовов Native API **реализована в ntoskrnl.exe**, а доступ к ним предоставляется к программам режима пользователя ntdll.dll. Некоторые вызовы Native API реализованы напрямую **в режиме пользователя внутри ntdll.dll**.
Как уже ранее упоминалось, пользовательские приложения не вызывают напрямую системные службы Windows. Вместо этого ими используется одна или несколько DLL-библиотек подсистемы. Эти библиотеки экспортируют документированный интерфейс, который может быть использован программами, связанными с данной подсистемой. Например, API-функции Windows реализованы в DLL-библиотеках подсистемы Windows, таких, как **Kernel32.dll, Advapi32.dll, User32.dll и Gdi32.dll**.
Когда приложение вызывает функцию, находящуюся в DLL-библиотеке подсистемы, может реализоваться одно из трех обстоятельств:
· Функция полностью реализована в режиме пользователя в DLL-библиотеке подсистемы. Иными словами, процессу подсистемы среды никакое сообщение не отправляется и никакие службы исполняющей системы Windows не вызываются. Функция выполняется в пользовательском режиме, а результаты возвращаются вызвавшему ее коду.
· Функция требует один или несколько вызовов исполняющей системы Windows. Например, Windows-функции ReadFile и WriteFile включают соответственно вызовы базовых внутренних (и недокументированных) системных служб ввода-вывода Windows NtReadFile и NtWriteFile.
· Функция требует проведения в процессе подсистемы среды окружения некоторой работы. (Процессы подсистем среды окружения, запущенные в пользовательском режиме, отвечают за обслуживание состояния клиентских приложений, находящихся под их управлением.) В этом случае к подсистеме среды окружения делается клиент-серверный запрос посредством отправки подсистеме сообщения для выполнения той или иной операции. Затем DLL-библиотека подсистемы, прежде чем вернуть управление вызывающему коду, ждет ответа.

Ntdll.dll является специальной библиотекой системной поддержки, предназначенной, главным образом, для использования DLL-библиотек подсистем. В ней содержатся функции двух типов:
- функции-заглушки, обеспечивающие переходы от диспетчера системных служб к системным службам исполняющей системы Windows;
- вспомогательные внутренние функции, используемые подсистемами, DLL-библиотеками подсистем и другими исходными образами.

Первая группа функций предоставляет интерфейс к службам исполняющей системы Windows, которые могут быть вызваны из пользовательского режима. К этой группе относятся более чем 400 функций, среди которых NtCreateFile, NtSetEvent и т. д. Как уже отмечалось, основная часть возможностей, присущих данным функциям, доступна через Windows API. Но некоторые возможности недоступны и предназначены для использования только внутри операционной системы.

Для каждой из этих функций в Ntdll содержится точка входа с именем, совпадающим с именем функции. Код внутри функции содержит зависящую от конкретной архитектуры инструкцию, осуществляющую переход в режим ядра для вызова диспетчера системных служб, который после проверки ряда параметров вызывает настоящую системную службу режима ядра, реальный код которой содержится в файле Ntoskrnl.exe.

Исполняющая система Windows находится на верхнем уровне файла **Ntoskrnl.exe**. (Ядро составляет его нижний уровень.)

Системные функции имеют префикс Nt, например NtReadFile. В режиме ядра дополнительно существуют парные функции с префиксом Zw, например ZwReadFile. Они вызываются драйверами вместо Nt-функций. В пользовательском режиме Zw-имена тоже объявлены, но эквивалентны Nt-именам.
Если Nt-функция вызывается внутри ядра:
· Проверка параметров не выполняется.
· Такая функция может быть недоступна в ядре, т.е. может не экспортироваться модулем Ntoskrnl.exe.
· Вызов Nt-функции с передачей ей указателей на память ядра закончится ошибкой.
Вместо Nt-функций модули ядра вызывают Zw-функции.
· Zw-функция делает следующее:
· Загружает в регистр EAX номер функции.
· Загружает в регистр EDX указатель на вершину параметров в стеке ядра.
· Вызывает соответствующую ей Nt-функцию. При этом проверка параметров не выполняется.

Больше о Zw и режиме ядра в следующем вопросе.

30. Системный вызов ОС Windows. Алгоритм системного вызова. Особенность системного вызова из режима ядра.

Собственные **системные службы** (или **системные вызовы**). Недокументированные базовые службы в операционной системе, вызываемые при работе в пользовательском режиме. Например, NtCreateUserProcess является внутренней службой, которую функция Windows CreateProcess вызывает для создания нового процесса.
Пользовательские приложения осуществляют переключение из пользовательского режима в режим ядра при осуществлении вызова системной службы. Например, Windows-функции ReadFile, в конечном счете, необходим вызов внутренней стандартной программы Windows, управляющей чтением данных из файла. Поскольку эта стандартная программа обращается к структурам внутренних системных данных, она должна работать в режиме ядра. Переход из режима пользователя в режим ядра осуществляется за счет использования специальной инструкции процессора, которая заставляет процессор переключиться в режим ядра и войти в код диспетчеризации системных служб, вызывающий соответствующую внутреннюю функцию в Ntoskrnl.exe или в Win32k.sys. Перед тем как вернуть управление пользовательскому потоку, процессор переключается в прежний, пользовательский режим работы. Таким образом, операционная система защищает саму себя и свои данные от прочтения и модификации со стороны пользовательских процессов.
Каждой **Nt-функции сопоставлен номер**. Номера зависят от версии Windows, полагаться на них не следует. Номер функции – это индекс в двух системных таблицах: nt!KiServiceTable и nt!KiArgumentTable. В первой таблице (System Service Descriptor Table – SSDT) хранятся адреса Nt-функций, во второй таблице – объемы параметров в байтах. Глобальная переменная nt!KeServiceDescriptorTable хранит три значения: указатель на таблицу nt!KiServiceTable, указатель на таблицу nt!KiArgumentTable и количество элементов в этих таблицах.

Алгоритм системного вызова (действия, выполняемые Nt-функцией библиотеки NtDll.dll):
- Загрузить в регистр EAX номер Nt-функции.
- Загрузить в регистр EDX указатель на вершину параметров в стеке (ESP).
- Вызвать прерывание для перехода процессора в режим ядра:
  - int 0x2E – на старых процессорах,
  - sysenter – на современных процессорах Intel,
  - syscall – на современных процессорах AMD.
- Если используется прерывание, то вызывается обработчик прерывания (Interrupt Service Routine – ISR), зарегистрированный в таблице обработчиков прерываний (Interrupt Descriptor Table – IDT) под номером 0x2E. Этот обработчик вызывает функцию ядра KiSystemService().
- Если используется специальная инструкция (sysenter или syscall), то происходит вызов функции, адрес которой хранится в специальном внутреннем регистре процессора (Model Specific Register – MSR). Этот регистр хранит адрес функции ядра KiFastCallEntry().
- После перехода в режим ядра все параметры, передаваемые в Nt-функцию, находятся на стеке пользовательского режима.

Алгоритм системного вызова (продолжение в режиме ядра):
- По номеру функции в регистре EAX отыскать в nt!KiArgumentTable количество байтов, занимаемое параметрами на стеке.
- Скопировать параметры со стека пользовательского режима на стек ядра. После переключение процессора в режим ядра стек тоже переключен.
- По номеру функции в регистре EAX отыскать в nt!KiServiceTable адрес функции для вызова.
- Выполнить вызов функции. Функция выполняется в контексте вызывающего процесса и потока и поэтому обращается к указателям пользовательского режима напрямую.

- Если функция вызвана из пользовательского режима, выполняется проверка параметров. Скалярные значения проверяются на допустимые диапазоны. Указатели проверяются с помощью функций ProbeForRead() и ProbeForWrite() в блоке __try { } __except { }.
- Вернуться из режима ядра в пользовательский режим с помощью:
  - iret – на старых процессорах,
  - sysexit – на современных процессорах Intel,
  - sysret – на современных процессорах AMD.

Вызов **в режиме ядра**:
Если аргументы, передаваемые системной службе, указывают на буферы в пользовательском пространстве, эти буферы должны быть проверены на доступность, прежде чем код режима ядра сможет копировать данные в эти буферы или из них. Эта проверка осуществляется только тогда, когда предыдущий режим (previous mode) потока установлен на пользовательский режим. Предыдущий режим является значением (режим ядра или пользовательский режим), которое ядро сохраняет в потоке, когда в нем выполняется обработчик системного прерывания и идентифицируется уровень привилегий входящего исключения, системного прерывания или системного вызова. В качестве оптимизации, если системный вызов поступает от драйвера или от самого ядра, проверка и захват параметров пропускаются, и все параметры считаются указывающими на допустимые буферы режима ядра (также разрешается доступ к данным в режиме ядра).
Поскольку системные вызовы могут также осуществляться кодом, выполняемым в режиме ядра, давайте рассмотрим способ их реализации. Так как код для каждого системного вызова выполняется в режиме ядра, а вызывающая программа уже выполняется в режиме ядра, можно прийти к выводу, что какоголибо прерывания или операции sysenter не требуется: центральный процессор уже находится на нужном уровне привилегий, и драйверы, как и ядро, должны только лишь иметь возможность непосредственного вызова требуемой функции. В случае исполняющей системы именно это и происходит: ядро имеет доступ ко всем своим собственным процедурам и может просто вызвать их точно так же, как вызывает стандартные процедуры. Но внешне драйверы могут получить доступ к этим системным вызовам только тогда, когда эти вызовы экспортированы подобно другим стандартным API-функциям режима ядра. Фактически, экспортировано довольно много системных вызовов. Но такой способ доступа для драйверов не предусматривается. Вместо этого драйверы должны использовать Zw-версии этих вызовов, то есть вместо NtCreateFile они должны использовать ZwCreateFile. Эти Zw-версии должны быть также вручную экспортированы ядром, их немного, но на них имеется полная документация и поддержка.
Из-за рассмотренного ранее понятия предыдущего режима Zw-версии официально доступны только для драйверов. Поскольку значение предыдущего режима обновляется только при каждом создании ядром фрейма системного прерывания, в связи с простым API-вызовом оно изменяться не будет, поскольку никакого фрейма системного вызова сгенерировано не будет. При непосредственном вызове таких функций, как NtCreateFile, ядро сохраняет значение предыдущего режима, которое показывает, что оно относится к пользовательскому режиму, обнаруживает, что переданный адрес относится к адресу режима ядра, и не выполняет вызов, правильно полагая, что приложения пользовательского режима не должны передавать указатели режима ядра. Но на самом деле ведь это не так, тогда как же ядро должно разобраться в правильном предыдущем режиме? Ответ заключается в Zw-вызовах.
Эти экспортированные API-функции на самом деле не являются простыми псевдонимами или оболочками Nt-версий. Вместо этого они являются своеобразными «батутами» для прыжка к соответствующим системным Nt-вызовам, использующим тот же самый механизм диспетчеризации системных вызовов. Вместо генерирования прерывания или использования инструкции sysenter, которые не отличались бы скоростью работы и (или) не поддерживались бы, они создают искусственный стек прерывания (стек, который центральный процессор сгенерировал бы после прерывания) и непосредственно вызывают процедуру KiSystemService, фактически имитируя прерывание центрального процессора. Обработчик выполняет те же операции, что и при поступлении этого вызова из пользовательского режима, за исключением того, что он обнаруживает фактический уровень привилегий, с которым поступил вызов, и устанавливает для предыдущего режима значение режима ядра (kernel). Теперь функция NtCreateFile видит, что вызов поступил из ядра, и больше не отвечает отказом.

Tl;dr Zw-функции предназначены для вызова из режима ядра. Они вызывают Nt-версию, имитируя прерывание CPU, тем самым обеспечивают правильную работу из режима ядра (выставление предыдущего режима в режим ядра).

31.Отладка драйверов ОС Windows. Средства отладки драйверов. Посмертный анализ. Живая отладка.

Отладка драйверов:
-Поддержка отладки ядра обеспечивается самим ядром Windows. Включается через командную строку: bcdedit /debug on
-В процессорах архитектуры x86 имеются специальные отладочные регистры DR0-DR7. Они позволяют отладчику ставить контрольные точки на чтение и запись памяти, а также на порты ввода-вывода.
-Современное средство отладки: Visual Studio 2013 Professional + WDK 8.1.
-Начиная с Windows Vista, обеспечивается создание ряда драйверов, работающих в пользовательском режиме.
-Для отладки применяется Visual Studio.
-Отладчик с графическим пользовательским интерфейсом – WinDBG
-Отладчик командной строки – KD
-KMDManager – для регистрации и запуска драйвера в системе(если не используем Visual Studio)
-DebugView – для получения отладочных сообщений, переданных такими командами, как DbgPrint(“message”), от драйвера (если не используем Visual Studio).

Отладчики позволяют выполнять три типа отладки ядра: »
- Открыть аварийный дамп-файл, созданный в результате синего экрана.
-Подключиться к удаленной системе и исследовать ее состояние (установить контрольные точки, если ведется отладка кода драйвера устройства).
-Подключиться к локальной системе и исследовать ее состояние. Это называется «локальной отладкой ядра».

Режимы отладчика :
-Live Debugging (прямая отладка)
-Отладчик во время работы ОС соединяется с ней через порты via Serial/1394/USB2.0
-Отладчик соединяется с ОС, что позволяет исследовать системную память
-Отладка Post Mortem
-Состояние ОС описано в crash dump файле
-Отладчик загружает crash dump файл и начинает анализ причин сбоя

Для посмертного анализа (postmortem analysis) нужно:
-Включить в операционной системе создание дампов памяти (Small memory dump (256 KB) или Kernel memory dump);
-Включить отладку в ядре: bcdedit /debug on
-Настроить канал связи отладчика с ядром: bcdedit /dbgsettings
-В отладчике включить загрузку символьной информации о ядре Windows;
-Открыть файл chash dump файл в отладчике.
Для живой отладки (live debugging) нужно:
-Соединить два компьютера через один из следующих интерфейсов:
-Serial, IEEE 1394, USB 2.0. или именованный канал (при использовании виртуальной машины);
-Включить отладку в ядре: bcdedit /debug on;
-Настроить канал связи отладчика с ядром: bcdedit /dbgsettings;
-В отладчике включить загрузку символьной информации о ядре Windows;
-В Visual Studio собрать драйвер;
-Поставить контрольную точку в исходном коде и установить драйвер

32.Структуры данных общего назначения в режиме ядра ОС Windows. Представление строк стандарта Unicode. Представление двусвязных списков.

1.СД общего назначения

Ядру известно, сколько процессов использует конкретный объект ядра, поскольку в каждом объекте есть счетчик числа его пользователей. Этот счетчик — один из элементов

данных, общих для всех типов объектов ядра. В момент создания объекта счетчику присваивается 1. Когда к существующему объекту ядра обращается другой процесс, счетчик увеличивается на 1. А когда какой-то процесс завершается, счетчики всех используемых им объектов ядра автоматически уменьшаются на единицу.

Диспетчер куч (heap manager) управляет кучами, как системными, так и пользовательскими, разбивая пространство кучи на блоки и организуя списки блоков одинакового размера. Если приходит запрос на выделение блока памяти из кучи, то диспетчер куч пытается подобрать свободный блок подходящего размера. Если же заранее известно, что потребуются блоки памяти фиксированного размера, но количество этих блоков и частота их использования не известны, то следует использовать, по соображениям лучшей производительности, так называемые ассоциативные списки (look-aside lists), которые существуют только в ядре. Главное отличие ассоциативных списков от пулов в том, что из ассоциативных списков можно выделять блоки памяти только фиксированного и заранее определенного размера, а из пулов любого, причем память из ассоциативных списков выделяется быстрее, так как нет необходимости подбирать подходящую область свободной памяти.

При работе с ассоциативным списком, то первой же проблемой, которую придется решить, кроме, конечно, создания самого ассоциативного списка, будет проблема управления блоками памяти, которые необходимо из этого ассоциативного списка будет выделять. В частности, где и как хранить адреса блоков - ведь к ним надо обращаться, а потом еще и освобождать. Поскольку количество этих блоков заранее не известно, то проблема достаточно серьезная. Для решения подобных задач существуют особые конструкции. Всего таких конструкций три:

* Односвязный список (singly linked list);

* S-список (S-list, sequenced singly-linked list.), являющийся развитием односвязного списка;

* Двусвязный список (doubly linked list).

Рассмотрим немного поподробнее такие понятия как ассоциативный и двусвязный списки. Хотя обе эти конструкции называются списками, по сути, это совершенно разные вещи. Перевод английского термина look-aside list на русский язык крайне абстрактен и плохо отражает суть. Look-aside дословно можно перевести как: "смотреть по сторонам". Смысл с том, что look-aside list представляет собой набор или список заранее выделенных системой блоков памяти. В каждый момент времени какие-то блоки могут быть свободны, а какие-то заняты. При запросе на выделение очередного блока задача системы пройти по списку ("посмотреть по сторонам ") и найти близлежащий свободный блок. В русском же переводе, look-aside превращается в "ассоциативный", и становится совершенно непонятно, что с чем здесь ассоциируется. Тем не менее, перевод этот устоявшийся - хочешь, не хочешь, придется применять. Т.о. ассоциативный список - это фактически особая системная куча, работающая по определенным правилам.

Двусвязный же список - это просто форма организации данных. С помощью двусвязных списков удобно связывать друг с другом однородные структуры данных и иметь возможность перебирать их в любом направлении. Конечно можно избежать использования ассоциативных списков, но пройти мимо использования двусвязного списка трудно. Двусвязные списки очень интенсивно используются самой системой для организации внутренних структур.

2.Unicode Unicode — стандарт кодирования символов, позволяющий представить знаки почти всех письменных языков.

Unicode — стандарт, первоначально разработанный Apple и Xerox в 1988 г. В 1991 г. был создан консорциум для совершенствования и внедрения Unicode. В него вошли компании Apple, Compaq, Hewlett-Packard, IBM, Microsoft, Oracle, Silicon Graphics, Sybase, Unisys и Xerox. (Полный список компаний — членов консорциума см. на www.Unicode.org.) Эта группа компаний наблюдает за соблюдением стандарта Uni- code, описание которого Вы найдете в книге The Unicode Standard издательства Addison-Wesley (ее электронный вариант можно получить на том же www.Uni- code.org).

Строки в Unicode просты и логичны. Все символы в них представлены 16-битны- ми значениями (по 2 байта на каждый). В них нет особых байтов, указывающих, чем является следующий байт — частью того же символа или новым символом. Это зна- чит, что прохождение по строке реализуется простым увеличением или уменьшением значения указателя. Функции CharNext, CharPrev и IsDBCSLeadByte больше не нужны. Так как каждый символ — 16-битное число, Unicode позволяет кодировать 65 536 символов, что более чем достаточно для работы с любым языком. Разительное отли- чие от 256 знаков, доступных в однобайтовом наборе! В настоящее время кодовые позиции 1 определены для арабского, китайского, гре- ческого, еврейского, латинского (английского) алфавитов, а также для кириллицы (русского), японской каны, корейского хангыль и некоторых других алфавитов. Кро- ме того, в набор символов включено большое количество знаков препинания, мате- матических и технических символов, стрелок, диакритических и других знаков. Все вместе они занимают около 35 000 кодовых позиций, оставляя простор для будущих расширений. Эти 65 536 символов разбиты на отдельные группы.

Unicode Windows отличается от других операционных систем тем, что большинство внутренних текстовых строк в ней хранится и обрабатывается в виде расширенных 16-разрядных символьных кодов Unicode. По своей сути Unicode является стандартом международных наборов символов, определяющим 16-разрядные значения для наиболее известных во всем мире наборов символов. Поскольку многие приложения работают со строками, состоящими из 8-разрядных (однобайтовых) ANSI-символов, многие Windows-функции, которым передаются строковые параметры, имеют две точки входа: в версии Unicode(расширенной, 16-разрядной) и в версии ANSI (узкой, 8-разрядной).

При вызове узкой версии Windows-функции происходит небольшое снижение произ- водительности, поскольку входящие строковые аргументы перед обработкой преобразуются в Unicode, а после обработки, при возвращении приложению, проходят обратное преобразование из Unicode в ANSI. Поэтому если у вас есть старая служба или старый фрагмент кода, который нужно запустить под управлением Windows, но этот фрагмент создан и с использованием текстовых строк, состоящих из ANSI-символов, Windows для своего собственного использования преобразует ANSI-символы в Unicode. Но Windows никогда не станет конвертировать данные внутри файлов: решение о том, в какой кодировки хранить данные: в Unicode или в ANSI, принимается самим приложением. Независимо от языка, во всех версиях Windows содержатся одни и те же функции. Вместо использования отдельных версий для каждого языка, в Windows используется единый универсальный двоичный код, поэтому отдельно взятая установка может поддерживать несколько языков (путем добавления различных язы- ковых пакетов).

Windows 2000 — операционная система, целиком и полностью построенная на Unicode. Все базовые функции для создания окон, вывода текста, операций со строками и т. д. ожидают передачи Unicode-строк. Если какой-то функции Windows передается ANSI-строка, она сначала преобразуется в Unicode и лишь потом передается операционной системе. Если Вы ждете результата функции в виде ANSI-строки, операционная система преобразует строку — перед возвратом в приложение — из Unicode в ANSI. Все эти преобразования протекают скрытно от Вас, но, конечно, на них тратятся и лишнее время, и лишняя память.

3.Двусвязные списки

Двусвязный список представляет собой список, связующая часть которого состоит из двух полей. Одно поле указывает на левого соседа данного элемента списка, другое — на правого. Кроме этого, со списком связаны два указателя — на голову и хвост списка. Более удобным может быть представление списка, когда функции этих указателей выполняет один из элементов списка, одно из полей связи которого указывает на последний элемент списка.

Преимущество использования двусвязных списков — в свободе передвижения по списку в обоих направлениях, в удобстве исключения элементов. Возникает вопрос о том, как определить начало списка. Для

этого существуют по крайней мере две возможности: определить два указателя на оба конца списка или определить голову списка, указатели связей которой адресуют первый и последний элемент списка.
В общем случае одно- и двусвязные списки представляют собой линейные связанные структуры. Но в определенных случаях второй указатель двусвязного списка может задавать порядок следования элементов, не являющийся обратным по отношению к первому указателю.

33. Механизм прерываний ОС Windows. Аппаратные и программные прерывания. Понятие прерывания, исключения и системного вызова. Таблица векторов прерываний (IDT).

Прерывания и исключения являются условиями операционной системы, отвлекающими процессор на выполнение кода, находящегося за пределами нормального потока управления. Они могут быть обнаружены либо аппаратными, либо программными средствами. Термин *системное прерывание* относится к механизму процессора, предназначенному для захвата выполняемого потока при возникновении исключения или прерывания и для передачи управления в определенное место в операционной системе. В Windows процессор передает управление *обработчику системного прерывания*, который является функцией, характерной для того или иного прерывания или исключения.

Ядро различает прерывания и исключения следующим образом. Прерывание является асинхронным событием (которое может произойти в любое время), не связанным с текущей задачей процессора. Прерывания генерируются, главным образом, устройствами ввода-вывода, процессорными часами или таймерами, и они могут быть разрешены (включены) или запрещены (выключены). Исключение, напротив, является синхронным условием, которое обычно возникает в результате выполнения конкретной инструкции. (Аварийные завершения работы, например, из-за машинного сбоя, обычно не связаны с выполнением инструкции.) Повторение исключений может быть вызвано повторным запуском программы с теми же данными и при тех же условиях. В качестве примеров исключений можно привести нарушения доступа к памяти, определенные инструкции отладки и ошибки деления на нуль. Ядро также считает исключениями вызовы системных служб (хотя технически они являются системными прерываниями).

Аппаратно генерируемые прерывания обычно исходят от устройств ввода-вывода, которые должны уведомить процессор о том, что они нуждаются в обслуживании. Устройства, управляемые прерываниями, позволяют операционной системе использовать процессор максимально эффективно, совмещая основную обработку данных с операциями ввода-вывода. Поток запускает передачу ввода-вывода в адрес устройства или от него, а затем может выполнять другую полезную работу, пока устройство не завершит передачу. Когда устройство завершит работу, оно прерывает процессор на свое обслуживание. Как правило, прерываниями управляются устройства указания координат, принтеры, клавиатуры, дисковые приводы и сетевые карты.

Прерывания могут также генерироваться системными программами. Например, ядро может выдать программное прерывание для инициирования диспетчера потока и для асинхронного проникновения в выполнение потока.

Как только работа процессора прерывается, этот процессор требует от контроллера запрос прерывания (Interrupt request, IRQ). Контроллер прерываний превращает IRQ в номер прерывания, использует этот номер в качестве индекса в структуре под названием *таблица диспетчеризации прерываний* (Interrupt dispatch table, IDT) и передает управление соответствующей процедуре обработки прерывания. Во время загрузки системы Windows заполняет IDT указателями на процедуры ядра, обрабатывающими каждое прерывание и исключение.

34. Аппаратные прерывания. Программируемый контроллер прерываний. Механизм вызова прерываний. Обработка аппаратных прерываний. Понятие приоритета прерываний (IRQL). Понятие процедуры обработки прерываний (ISR).

На аппаратных платформах, поддерживаемых Windows, внешние прерывания ввода-вывода поступают на одну из линий контроллера прерываний. В свою очередь, контроллер прерывает работу процессора по отдельной линии. Как только работа процессора будет прервана, этот процессор требует от контроллера запрос прерывания (Interrupt request, IRQ). Контроллер прерываний превращает IRQ в номер прерывания, использует этот номер в качестве индекса в структуре под названием *таблица диспетчеризации прерываний* (Interrupt dispatch table, IDT) и передает управление соответствующей процедуре обработки прерывания. Во время загрузки системы Windows заполняет IDT указателями на процедуры ядра, обрабатывающими каждое прерывание и исключение.

Windows отображает аппаратные IRQ-запросы на номера прерываний в IDT и также использует IDT для настройки обработчиков системных прерываний для исключений.

Контроллер прерываний x86

В большинстве систем x86 используется либо программируемый контроллер прерываний (Programmable Interrupt Controller, PIC) i8259A, либо один из вариантов усовершенствованного программируемого контроллера прерываний (Advanced Programmable Interrupt Controller, APIC) i82489. Современные контроллеры комплектуются контроллером APIC

Фактически APIC-контроллер состоит из нескольких компонентов: APIC ввода-вывода, получающего прерывания от устройств, локальных APIC-контроллеров, получающих прерывания от APIC ввода-вывода на шине и прерывающих работу того центрального процессора, с которым они связаны, и i8259A-совместимого контроллера прерываний, который переводит APIC-ввод в сигналы, эквивалентные PIC-контроллеру. Поскольку в системе может быть несколько APIC-контроллеров ввода-вывода, у материнских плат обычно есть часть основных логических устройств, расположенных между ними и процессорами. Эти логические устройства отвечают за реализацию алгоритмов процедур прерываний, которые наряду с балансировкой нагрузки аппаратных прерываний между процессорами и попыткой получить преимущества от локализации, доставляют прерывания от устройств тому же самому процессору, которому только что было направлено предыдущее прерывание того же типа. Обычные программы могут перепрограммировать APIC-контроллеры ввода-вывода, применив фиксированный алгоритм маршрутизации, обходящий логику набора микросхем. Windows делает это путем программирования APIC-контроллеров в режиме маршрутизации «прерывания одного процессора в следующем наборе».

Контроллеры прерываний x64

Поскольку архитектура x64 совместима с операционными системами x86, системы x64 должны предоставлять такие же контроллеры прерываний, что и системы x86. Но существенная разница состоит в том, что x64-версии не будут запускаться на системах, которые не имеют APIC, поскольку для управления прерываниями они используют APIC.

Контроллеры прерываний IA64

Архитектура IA64 зависит от модернизированного усовершенствованного программируемого контроллера Streamlined Advanced Programmable Interrupt Controller (SAPIC), который является результатом развития контроллера APIC. Даже если во встроенной программе присутствует балансировка и маршрутизация, Windows ими не пользуется, вместо этого она назначает прерывания процессорам статически, по кругу.

Уровни запросов программных прерываний (IRQL)

Хотя контроллеры прерываний устанавливают приоритетность прерываний, Windows устанавливает свою собственную схему приоритетности прерываний, известную как *уровни запросов прерываний* (IRQL). В ядре IRQL-уровни представлены в виде номеров от 0 до 31 на системах x86 и в виде номеров от 0 до 15 на системах x64 и IA64, где более высоким номерам соответствуют прерывания с более высоким приоритетом. Хотя ядро определяет для программных прерываний стандартный набор IRQL-уровней, HAL отображает номера аппаратных прерываний на IRQL-уровни.

Прерывания обслуживаются в порядке их приоритета, и прерывания с более высоким уровнем приоритета получают преимущество в обслуживании. При возникновении прерывания с высоким уровнем процессор сохраняет состояние прерванного потока и запускает связанный с прерыванием диспетчер системных прерываний. Тот, в свою очередь, поднимает IRQL и вызывает процедуру обработки прерывания. После выполнения этой процедуры диспетчер прерываний понижает IRQL-уровень процессора до значения, на котором он был до возникновения прерывания, а затем загружает сохраненное состояние машины. Прерванный поток продолжает выполнение с того места, в котором оно было прервано. Когда ядро понижает IRQL, могут реализоваться те прерывания с более низким уровнем приоритета, которые были замаскированы. Если так и происходит, ядро повторяет процесс для обработки новых прерываний.

ISR (Interrupt Service Routine) — специальная процедура, вызываемая по прерыванию для выполнения его обработки. Обработчики прерываний могут выполнять множество функций, которые зависят от причины, которая вызвала прерывание.

Обработчик прерываний — это низкоуровневый эквивалент обработчика событий. Эти обработчики вызываются либо по аппаратному прерыванию, либо соответствующей инструкцией в программе, и соответственно обычно предназначены для взаимодействия с устройствами или для осуществления вызова функций операционной системы.

35. Понятие приоритета прерываний (IRQL). Приоритеты прерываний для процессора x86 или x64. Процедура обработки прерываний (ISR). Схема обработки аппаратных прерываний.

IRQL (англ. Interrupt ReQuest Level) — уровень запроса прерывания. Механизм программно-аппаратной приоритезации, применяемый для синхронизации в операционных системах семейства Windows NT.

IRQL является программным атрибутом (из-за того, что не поддерживается аппаратно) процессора и указывает приоритет кода, исполняющегося на этом процессоре по отношению к прерываниям и другим асинхронным событиям. Для аппаратных прерываний, в большинстве случаев, IRQL реализуется аппаратно (пример: понятие приоритета прерывания в контроллере i8259A или приоритет задачи, указываемый в регистре TPR в APIC), однако код операционной системы сам может логически находиться на разных приоритетах, в таком случае дополнительные уровни IRQL реализуются программно. Например, приоритет планировщика потоков или DPC выше, чем приоритет пользовательских потоков. Если бы это было не так, тогда потоки могли бы вытеснить планировщик и тем самым «отключить» вытесняющую многозадачность, в свою очередь планировщик может быть сделан прерываемым аппаратными прерываниями. В Windows NT применяется 32 уровня IRQL (в скобках указано числовое значение):

- `High (31)`
- `Power fail (30)`
- `IPI (29)`
- `Clock (28)`
- `Profile (27)`

Диапазон аппаратных прерываний, называемых Devices IRQL/DIRQL (от 26 до 3)

- `DPC/DISPATCH (2)`
- `APC (1)`
- `PASSIVE (0)`

1) IRQL с уровнем HIGH (запрещаются выполняться все прерывания), работает в режиме ядра, при этом прерывания не теряются, а поступают в очередь на обработку.

2) уровень Power Fail – отказ электропитания – блокирует все процессы в случае отказа электропитания.

3) Interprocessor Interrupt – межпроцессорное прерывание – используется при взаимодействии процессоров и запроса на выполнение какой-то операции. Тоже достаточно приоритетное прерывание, которое блокирует все остальные.

4) уровень Clock – используется для управления системными часами, распределением и измерением времени, выделяемого исполняемым потокам.

5) уровень Profile – прерывание, которое блокирует все остальные ниже него, когда включен режим измерения производительности.

6) уровень устройства Devive (1…n) – задает приоритеты прерываний от тех или иных устройств.

7) прерывания обработки отложенных процедур DPC. Этот IRQL предоставляется программным прерыванием. которое генерируется ядром или драйверами устройств.

8) уровень Passive – ничего не блокирует, позволяет все прерывать и выполнять соответствующие обработчики.)

Это означает, например, что планировщик (работающий на уровне DPC/DISPATCH) может быть прерван аппаратными прерываниями, межпроцессорными прерываниями (IPI) и т. д., но не может быть прерван

асинхронными процедурами (APC) и обычными потоками, работающими на уровне PASSIVE. Межпроцессорные прерывания IPI могут быть прерваны сбоем электропитания (прерывание на уровне Power fail), но не могут быть прерваны обычными аппаратными прерываниями от устройств и т. д.

Также IRQL помогает отслеживать и выявлять логические ошибки при проектировании ОС. Легендарная ошибка с сообщением IRQL_NOT_LESS_OR_EQUAL означает следующую ситуацию: драйвер или другой привилегированный код с IRQL >= DPC/DISPATCH обратился к отсутствующей в памяти странице, требуется вызов подсистемы, подгружающей страницы с диска, однако эта подсистема в соответствии с архитектурой Windows NT имеет IRQL меньше, чем DPC/DISPATCH. Следовательно, она не имеет права прерывать тот код, который вызвал ошибку страницы. В то же время привилегированный код не может продолжить выполнение, пока страница не будет загружена. Возникает логический тупик, который, собственно, и приводит к краху ОС.

Прерывание (англ. interrupt) — сигнал, сообщающий процессору о наступлении какого-либо события. При этом выполнение текущей последовательности команд приостанавливается и управление передаётся обработчику прерывания, который реагирует на событие и обслуживает его, после чего возвращает управление в прерванный код.

В зависимости от источника возникновения сигнала прерывания делятся на:

* асинхронные или внешние (аппаратные) — события, которые исходят от внешних источников (например, периферийных устройств) и могут произойти в любой произвольный момент: сигнал от таймера, сетевой карты или дискового накопителя, нажатие клавиш клавиатуры, движение мыши. Факт возникновения в системе такого прерывания трактуется как запрос на прерывание (англ. Interrupt request, IRQ);

* синхронные или внутренние — события в самом процессоре как результат нарушения каких-то условий при исполнении машинного кода: деление на ноль или переполнение, обращение к недопустимым адресам или недопустимый код операции;

* программные (частный случай внутреннего прерывания) — инициируются исполнением специальной инструкции в коде программы. Программные прерывания как правило используются для обращения к функциям встроенного программного обеспечения (firmware), драйверов и операционной системы.

Маскирование. В зависимости от возможности запрета внешние прерывания делятся на:

* маскируемые — прерывания, которые можно запрещать установкой соответствующих битов в регистре маскирования прерываний (в x86-процессорах — сбросом флага IF в регистре флагов);

* немаскируемые (англ. Non maskable interrupt, NMI) — обрабатываются всегда, независимо от запретов на другие прерывания. К примеру, такое прерывание может быть вызвано сбоем в микросхеме памяти.

Обработчики прерываний обычно пишутся таким образом, чтобы время их обработки было как можно меньшим, поскольку во время их работы могут не обрабатываться другие прерывания, а если их будет много (особенно от одного источника), то они могут теряться.

Группа команд, выполняемых в ответ на запрос прерывания, называется подпрограммой обработки прерывания. Подпрограммы обработки прерываний во многом сходны с другими подпрограммами за исключением того, что они автоматически вызываются аппаратным механизмом вызова, а не командами вызова подпрограмм, и все регистры CPU08, за исключением регистра H, сохраняются в стеке.

Механизм прерываний чаще всего поддерживает приоритезацию и маскирование прерываний. Приоритезация означает, что все источники прерываний делятся на классы и каждому классу назначается свой уровень приоритета запроса на прерывание. Приоритеты могут обслуживаться как относительные и абсолютные. Обслуживание запросов прерываний по схеме с относительными приоритетами заключается в том, что при одновременном поступлении запросов прерываний из разных классов выбирается запрос, имеющий высший

приоритет. Однако в дальнейшем при обслуживании этого запроса процедура обработки прерывания уже не откладывается даже в том случае, когда появляются более приоритетные запросы — решение о выборе нового запроса принимается только в момент завершения обслуживания очередного прерывания. Если же более приоритетным прерываниям разрешается приостанавливать работу процедур обслуживания менее приоритетных прерываний, то это означает, что работает схема приоритезации с абсолютными приоритетами.

Если процессор (или компьютер, когда поддержка приоритезации прерываний вынесена во внешний по отношению к процессору блок) работает по схеме с абсолютными приоритетами, то он поддерживает в одном из своих внутренних регистров переменную, фиксирующую уровень приоритета обслуживаемого в данный момент прерывания. При поступлении запроса из определенного класса его приоритет сравнивается с текущим приоритетом процессора, и если приоритет запроса выше, то текущая процедура обработки прерываний вытесняется, а по завершении обслуживания нового прерывания происходит возврат к прерванной процедуре.

Упорядоченное обслуживание запросов прерываний наряду со схемами приоритетной обработки запросов может выполняться механизмом маскирования запросов. Собственно говоря, в описанной схеме абсолютных приоритетов выполняется маскирование — при обслуживании некоторого запроса все запросы с равным или более низким приоритетом маскируются, то есть не обслуживаются. Схема маскирования предполагает возможность временного маскирования прерываний любого класса независимо от уровня приоритета.

Обобщенно последовательность действий аппаратных и программных средств по обработке прерывания можно описать следующим образом:

1. При возникновении сигнала (для аппаратных прерываний) или условия (для внутренних прерываний) прерывания происходит первичное аппаратное распознавание типа прерывания. Если прерывания данного типа в настоящий момент запрещены (приоритетной схемой или механизмом маскирования), то процессор продолжает поддерживать естественный ход выполнения команд. В противном случае в зависимости от поступившей в процессор информации (уровень прерывания, вектор прерывания или тип условия внутреннего прерывания) происходит автоматический вызов процедуры обработки прерывания, адрес которой находится в специальной таблице операционной системы, размещаемой либо в регистрах процессора, либо в определенном месте оперативной памяти.

2. Автоматически сохраняется некоторая часть контекста прерванного потока, которая позволит ядру возобновить исполнение потока процесса после обработки прерывания. В это подмножество обычно включаются значения счетчика команд, слова состояния машины, хранящего признаки основных режимов работы процессора (пример такого слова — регистр EFLA6S в Intel Pentium), а также нескольких регистров общего назначения, которые требуются программе обработки прерывания. Может быть сохранен и полный контекст процесса, если ОС обслуживает данное прерывание со сменой процесса. Однако в общем случае это не обязательно, часто обработка прерываний выполняется без вытеснения текущего процесса.

3. Решение о перепланировании процессов может быть принято в ходе обработки прерывания, например, если это прерывание от таймера и после наращивания значения системных часов выясняется, что процесс исчерпал выделенный ему квант времени. Однако это совсем не обязательно — прерывание может выполняться и без смены процесса, например прием очередной порции данных от контроллера внешнего устройства чаще всего происходит в рамках текущего процесса, хотя данные, скорее всего, предназначены другому процессу

4. Одновременно с загрузкой адреса процедуры обработки прерываний в счетчик команд может автоматически выполняться загрузка нового значения слова состояния машины (или другой системной структуры, например селектора кодового сегмента в процессоре Pentium), которое определяет режимы работы процессора при обработке прерывания, в том числе работу в привилегированном режиме. В некоторых моделях процессоров переход в привилегированный режим за счет смены состояния машины при обработке прерывания является единственным способом смены режима. Прерывания практически во всех мультипрограммных ОС обрабатываются в привилегированном режиме модулями ядра, так как при этом обычно нужно выполнить ряд

критических операций, от которых зависит жизнеспособность системы, — управлять внешними устройствами, перепланировать потоки и т. п.

5. Временно запрещаются прерывания данного типа, чтобы не образовалась очередь вложенных друг в друга потоков одной и той же процедуры. Детали выполнения этой операции зависят от особенностей аппаратной платформы, например может использоваться механизм маскирования прерываний. Многие процессоры автоматически устанавливают признак запрета прерываний в начале цикла обработки прерывания, в противном случае это делает программа обработки прерываний

6. После того как прерывание обработано ядром операционной системы, прерванный контекст восстанавливается и работа потока возобновляется с прерванного места. Часть контекста восстанавливается аппаратно по команде возврата из прерываний (например, адрес следующей команды и слово состояния машины), а часть — программным способом, с помощью явных команд извлечения данных из стека. При возврате из прерывания блокировка повторных прерываний данного типа снимается

36. **Программные прерывания. Понятие отложенной процедуры (DPC). Назначение отложенных процедур. Механизм обслуживания отложенных процедур. Операции с отложенными процедурами.**

Назначение программных прерываний:

- запуск диспетчеризации потоков;
- обработка не критичных по времени прерываний;
- обработка истечения времени таймера;
- асинхронное выполнение процедуры в контексте конкретного потока;
- поддержка асинхронных операций ввода-вывода.

DPC (Deferred Procedure Call).

Ядро всегда поднимает IRQL процессора до уровня DPC/dispatch или выше, когда ему нужно синхронизировать доступ к общим структурам ядра. Тем самым блокируются дополнительные программные прерывания и диспетчеризация потоков. Когда ядро обнаруживает необходимость диспетчеризации, оно запрашивает прерывание уровня DPC/dispatch, но процессор задерживает прерывание, поскольку IRQL находится на этом или на более высоком уровне. Когда ядро завершает свою текущую работу, процессор видит, что оно собирается поставить IRQL ниже уровня DPC/dispatch, и проверяет наличие каких-либо отложенных прерываний диспетчеризации. Если такие прерывания имеются, IRQL понижается до уровня DPC/dispatch, и происходит обработка прерываний диспетчеризации. Активизация диспетчера потоков с помощью программного прерывания является способом, позволяющим отложить диспетчеризацию, пока для нее не сложатся нужные обстоятельства. Но Windows использует программные прерывания для того, чтобы отложить и другие типы обработки.

Кроме диспетчеризации потоков ядро обрабатывает на этом уровне IRQL [DPC/dispatch] и отложенные вызовы процедур (DPC). DPC является функцией, выполняющей ту системную задачу, которая менее критична по времени, чем текущая задача. Функции называются отложенными (deferred), потому что они не требуют немедленного выполнения.

Отложенные вызовы процедур дают операционной системе возможность генерировать прерывание и выполнять системную функцию в режиме ядра.

Ядро использует DPC-вызовы для обслуживания истечений времени таймера (и освобождения потоков, которые ожидают истечения времени таймеров) и для перепланирования времени использования процессора после истечения времени, выделенного потоку (кванта потока). Драйверы устройств используют DPC-вызовы для обработки прерываний. Для обеспечения своевременного обслуживания программных прерываний Windows совместно с драйверами устройств старается сохранять IRQL ниже IRQL-уровней устройств. Одним из способов достижения этой цели является выполнение ISR-процедурами драйверов устройств минимально необходимой работы для оповещения своих устройств, сохранения временного состояния прерывания и задержки передачи данных или обработки других, менее критичных по времени прерываний для выполнения в DPC-вызовах на IRQL-уровне DPC/dispatch.

DPC-вызов представлен DPC-объектом, который является объектом управления ядра, невидимым для программ пользовательского режима, но видимым для драйверов устройств и другого системного кода. Наиболее важной информацией, содержащейся в DPC-объекте, является адрес системной функции, которую ядро вызовет при обработке DPC-прерывания. DPC-процедуры, ожидающие выполнения, хранятся в очередях, управляемых ядром, — по одной очереди на каждый процессор. Они называются DPC-очередями. Для запроса DPC системный код вызывает ядро для инициализации DPC-объекта, а затем помещает этот объект в DPC-очередь.

По умолчанию ядро помещает DPC-объекты в конец DPC-очереди того процессора, при работе которого была запрошена DPC-процедура (обычно того процессора, на котором выполняется ISR-процедура). Но драйвер устройства может отменить такое поведение, указав DPC-приоритет (низкий, средний, выше среднего или высокий, где по умолчанию используется средний приоритет) и нацелив DPC на конкретный процессор. DPC-вызов, нацеленный на конкретный центральный процессор, известен как целевой DPC. Если DPC имеет высокий приоритет, ядро ставит DPC-объект в начало очереди, в противном случае для всех остальных приоритетов оно ставит объект в конец очереди.

Ядро обрабатывает DPC-вызовы, когда IRQL-уровень процессора готов понизиться с IRQL-уровня DPC/dispatch или более высокого уровня до более низкого IRQL-уровня (APC или passive). Windows обеспечивает пребывание IRQL на уровне DPC/dispatch и извлекает DPC-объекты из очереди текущего процессора до тех пор, пока она не будет исчерпана (то есть ядро «расходует» очередь), вызывая по очереди каждую DPC-функцию. Ядро даст возможность IRQL-уровню упасть ниже уровня DPC/dispatch и позволить продолжить обычное выполнение потока только когда очередь истощится.

Поскольку потоки пользовательского режима выполняются при низком IRQL, высоки шансы на то, что DPC-вызов прервет выполнение обычного пользовательского потока. DPC-процедуры выполняются независимо от того, какой поток запущен, стало быть, когда запускается DPC-процедура, она не может выстроить предположение насчет того, чье адресное пространство, какого именно процесса в данный момент отображается. DPC-процедуры могут вызывать функции ядра, но они не могут вызывать системные службы, генерировать ошибки отсутствия страницы или же создавать или ожидать объекты диспетчеризации. Тем не менее они могут обращаться к невыгружаемым адресам системной памяти, поскольку системное адресное пространство отображено всегда, независимо от того, что из себя представляет текущий процесс.

[Из презентации Суркова К.А.]

DPC-процедура представляется структурой ядра KDPC, в которой хранится адрес callback-процедуры, составляющей подпрограмму DPC. Структура KDPC создается и ставится в очередь в специальный список отложенных процедур процессора. Он находится здесь: KPCR.PrcbData.DpcListHead. Стек для выполнения DPC-процедур находится здесь: KPCR.PrcbData.DpcStack.

Функции работы с DPC:
- KeInitializeDpc()
- KeInsertQueueDpc()
- KeRemoveQueueDpc()
- KeSetTargetProcessorDpc()
- KeSetImportanceDpc()
- KeFlushQueuedDpcs()

37. **Понятие асинхронной процедуры (APC). Назначение асинхронных процедур. Типы асинхронных процедур. Операции с асинхронными процедурами.**

Асинхронные вызовы процедур дают пользовательским программам и системному коду способ выполнения в контексте конкретного пользовательского потока (а следовательно, в адресном пространстве конкретного процесса). Поскольку APC-вызовы выстраиваются в очередь на выполнение в контексте конкретного потока и запускаются на IRQL-уровне, который ниже уровня DPC/dispatch, они не подпадают под такие же ограничения, которые накладываются на DPC. APC-процедура может получать ресурсы (объекты), ждать дескрипторов объектов, справляться с ошибками отсутствия страницы и вызывать системные службы.

APC-вызовы описываются объектом управления ядра, называемым APC-объектом. APC-вызовы, ожидающие выполнения, помещаются в управляемую ядром APC-очередь. В отличие от очереди DPC, которая видна всей системе, APC-очередь относится к конкретному потоку — у каждого потока есть своя собственная APC-очередь. При запросе на помещение в очередь APC-вызова ядро вставляет ее в очередь, принадлежащую тому потоку, который будет выполнять APC-процедуру. Ядро, в свою очередь, запрашивает программное прерывание на APC-уровне а затем, когда поток через некоторое время начнет работать, в нем выполняется APC-вызов.

Существует два вида APC-вызовов: режима ядра и пользовательского режима. APC-вызовы режима ядра не требуют разрешения от целевого потока на запуск в его контексте, а APC-вызовы пользовательского потока требуют такого разрешения. APC-вызовы режима ядра прерывают поток и выполняются без его вмешательства или разрешения. Есть также два вида APC-вызовов режима ядра: обычные и специальные. Специальные APC-вызовы выполняются на уровне APC и позволяют APC-процедуре изменять некоторые APC-параметры. Обычные APC-вызовы выполняются на уровне passive и получают измененные параметры от специальной APC-процедуры (или исходные параметры, если они не были изменены).

APC-вызовы пользовательского режима используются несколькими Windows API-функциями, такими как ReadFileEx, WriteFileEx и QueueUserAPC... Например, функции ReadFileEx и WriteFileEx позволяют вызывающему коду указать подпрограмму завершения, вызываемую при окончании операции ввода-вывода. Завершение ввода-вывода реализуется постановкой APC-вызова в очередь того потока, который выдал запрос на ввод-вывод. Но обратный вызов процедуры завершения не обязательно происходит при постановке APC-вызова в очередь, поскольку APC-вызовы пользовательского режима доставляются потоку только в том случае, когда он находится в готовности к работе в режиме ожидания. [Пояснение из презентации Суркова К.А.] Если в очередь потоку ставится APC, и поток находится в состоянии ожидания объекта ядра, APC-процедура все-таки не заставит поток проснуться для ее выполнения. Чтобы поток просыпался для выполнения APC-процедур, он должен ожидать объекты с помощью alertable-функций, например, WaitForSingleObjectEx(…, true), SleepEx(…, true).

[Из презентации Суркова К.А.]

APC-процедура представляется структурой ядра KAPC. Структура KAPC содержит указатели на три подпрограммы:
- RundownRoutine – выполняется, если из-за удаления потока удаляется структура KAPC.
- KernelRoutine – выполняется на уровне приоритета APC_LEVEL.
- NormalRoutine – выполняется на уровне приоритета PASSIVE_LEVEL.

Структура KAPC создается и ставится в одну из двух очередей потока: одна очередь предназначена для APC режима ядра, вторая – для APC пользовательского режима. Начала очередей находятся в массиве из двух элементов: KTHREAD.ApcState.ApcListHead[].

В пользовательском режиме: - QueueUserApc()

38. **Понятие асинхронной процедуры (APC). Асинхронные процедуры режима ядра: специальная и нормальная APC-процедуры. Асинхронные процедуры пользовательского режима.**

[Почитай еще раз предыдущий вопрос]

Существует два вида APC-вызовов: режима ядра и пользовательского режима. APC-вызовы режима ядра не требуют разрешения от целевого потока на запуск в его контексте, а APC-вызовы пользовательского потока требуют такого разрешения. APC-вызовы режима ядра прерывают поток и выполняются без его вмешательства или разрешения. Есть также два вида APC-вызовов режима ядра: обычные и специальные. Специальные APC-вызовы выполняются на уровне APC и позволяют APC-процедуре изменять некоторые APC-параметры. Обычные APC-вызовы выполняются на уровне passive и получают измененные параметры от специальной APC-процедуры (или исходные параметры, если они не были изменены).

Исполняющая система использует APC-вызовы режима ядра для выполнения работы операционной системы, которая должна быть завершена в адресном пространстве (в контексте) конкретного потока. Она может использовать специальные APC-вызовы, чтобы направить поток, к примеру, на остановку выполнения прерываемой системной службы или для записи результатов асинхронной операции ввода-вывода в адресном пространстве потока. Подсистемы среды окружения используют специальные APC-вызовы режима ядра, чтобы заставить поток приостановить или завершить свою работу, или же получить или установить контекст его выполнения в пользовательском режиме. Подсистема для UNIX-приложений использует APC-вызовы режима ядра для имитации доставки UNIX-сигналов процессам подсистемы для UNIX-приложений.

Другое важное применение APC-вызовов режима ядра относится к приостановке или завершению потока. Поскольку эти операции могут инициироваться произвольными потоками и направлены на другие произвольные потоки, ядро использует APC для запроса контекста потока, а также для завершения потока. Драйверы устройств часто блокируют APC-вызовы или входят в критическую или охраняемую область, чтобы воспрепятствовать выполнению этих операций в тот момент, когда они удерживают блокировку, в противном случае блокировка может быть никогда не снята, и система зависнет.

APC-вызовы режима ядра используются также драйверами устройств. Например, если инициирована операция ввода-вывода и поток перешел в режим ожидания, может быть спланирован запуск другого потока в другом процессе. Когда устройство завершит передачу данных, система ввода-вывода должна каким-то образом вернуться в контекст потока, инициировавшего ввод-вывод, чтобы он мог скопировать результаты операции ввода-вывода в буфер в адресном пространстве процесса, содержащего этот поток. Для выполнения этого действия система ввода-вывода использует специальный APC-вызов режима ядра, если только приложение не использовало API-функция SetFileIoOverlappedRange или порты завершения ввода-вывода, — в таком случае либо буфер в памяти будет глобальным, либо копирование его произойдет только после того, как поток извлечет из порта признак завершения.

[Из презентации Суркова К.А.]

Специальная APC-процедура режима ядра:
- KAPC.KernelRoutine выполняется на уровне приоритета APC_LEVEL.
- Помещается в очередь APC режима ядра после других специальных APC.
- Вызывается перед нормальными APC режима ядра.
- Вызывается, если IRQL == PASSIVE_LEVEL и поток не находится в защищенной секции (guarded region).
- Специальная APC используется для завершения процесса, для передачи результатов ввода-вывода в адресное пространство потока.

Нормальная APC-процедура режима ядра:
- KAPC.NormalRoutine выполняется на уровне приоритета PASSIVE_LEVEL.
- Вызывается, если IRQL == PASSIVE_LEVEL, поток не находится в защищенной или критической секции (critical region) и не выполняет специальную APC ядра.

- Нормальной APC разрешено делать все системные вызовы.
- Нормальная APC используется ОС для завершения обработки запроса от драйвера – Interrupt Request Packet (IRP).

Функции управления APC-режимами ядра:
- KeInitializeApc()
- KeInsertQueueApc()
- KeRemoveQueueApc()
- KeFlushQueueApc()
- KeAreApcsDisabled()
- KeEnterGuardedRegion()
- KeLeaveGuardedRegion()
- KeEnterCriticalRegion()
- KeLeaveCriticalRegion()

39. Понятие элемента работы (Work Item). Назначение элементов работы. Операции с элементами работы. Очереди элементов работы. Обслуживание элементов работы.

WorkItem выполняет обработку для элемента работы, который был в очереди IoQueueWorkItem.

```
IO_WORKITEM_ROUTINE WorkItem;
VOID WorkItem(
  _In_     PDEVICE_OBJECT DeviceObject,
  _In_opt_ PVOID          Context
)
{ ... }
```

Параметры

DeviceObject - Указатель на один из объектов устройств вызывающего абонента. Это указатель, который был принят в качестве DeviceObject параметра IoAllocateWorkItem, когда рабочий элемент был выделен, или как IoObject параметра IoInitializeWorkItem, когда рабочий элемент инициализирован.

Context - Определяет контекстную информацию конкретного драйвера. Это значение, которое было принято в качестве контекстного параметра IoQueueWorkItem, когда рабочий элемент был помещен в очередь.

Возвращаемое значение - None

Замечания

Драйвер ставит в очередь рабочий элемент, вызывая IoQueueWorkItem, и текущий поток выполняет рабочий элемент. Рабочий элемент должен иметь определенный лимит времени для выполнения. В противном случае, у системы будет deadlock.

Пример

Чтобы определить возвращаемое значение рабочего элемента, сначало нам надо определить декларацию функции(которая и определяет тип возвращаемого значения). Windows предоставляет набор функциональных типов обратного вызова для драйверов. Указание типа помогает различным верификаторам распознать ошибки.

Например, ниже определение своего рабочего элемента MyWorkItem:

```
IO_WORKITEM_ROUTINE MyWorkItem;
```

И реализация:

```
_Use_decl_annotations_
ПУСТОТА
  MyWorkItem (
    PDEVICE_OBJECT DeviceObject,
    PVOID Контекст
    )
  {
      // Функция тела
  }
```

(Инфа из книги Рихтера)

Допустим, у Вас есть серверный процесс с основным потоком, который ждет клиентский запрос. Получив его, он порождает отдельный поток для обработки этого запроса. Тем самым основной поток освобождается для приема следующего клиентского запроса. Такой сценарий типичен в клиент-серверных приложениях. Хотя он и так-то незатейлив, при желании его можно реализовать с использованием новых функций пула потоков. Получая клиентский запрос, основной поток вызывает:

```
BOOL QueueUserWorkItem(
      PTHREAD_START_ROUTINE pfnCallback,
      PVOID pvContext,
      ULONG dwFlags);
```

Эта функция помещает «рабочий элемент» (work item) в очередь потока в пуле и тут же возвращает управление. Рабочий элемент — это просто вызов функции (на которую ссылается параметр pfnCallback), принимающей единственный параметр, pvContext. В конечном счете какой-то поток из пула займется обработкой этого элемента, в результате чего будет вызвана Ваша функция. У этой функции обратного вызова, за реализацию которой отвечаете Вы, должен быть следующий прототип:

```
DWORD WINAPI WorkItemFunc(PVOID pvContext);
```

Несмотря на то что тип возвращаемого значения определен как DWORD, на самом деле оно игнорируется. Обратите внимание, что Вы сами никогда не вызываете CreateThread. Она вызывается из пула потоков, автоматически создаваемого для Вашего процесса, а к функции WorkItemFunc обращается один из потоков этого пула.

Кроме того, данный поток не уничтожается сразу после обработки клиентского запроса, а возвращается в пул, оставаясь готовым к обработке любых других элементов, помещаемых в очередь. Ваше приложение может стать гораздо эффективнее, так как Вам больше не придется создавать и уничтожать потоки для каждого клиентского запроса. А поскольку потоки связаны с определенным портом завершения, количество одновременно работающих потоков не может превышать число процессоров более чем в 2 раза. За счет этого переключения контекста происходят реже.

Многое в пуле потоков происходит скрытно от разработчика: QueueUserWorkItem проверяет число потоков, включенных в сферу ответственности компонента поддержки других операций (не относящихся к вводу-выводу), и в зависимости от текущей нагрузки (количества рабочих элементов в очереди) может передать ему другие потоки. После этого QueueUserWorkItem выполняет операции, эквивалентные вызову PostQueuedCompletionStatus, пересылая информацию о рабочем элементе в порт завершения ввода-вывода. В конечном счете поток, ждущий на этом объекте, извлекает Ваше сообщение (вызовом GetQueuedCompletionStatus) и обращается к Вашей функции. После того как она возвращает управление, поток вновь вызывает GetQueuedCompletionStatus, ожидая появления следующего рабочего элемента.

Очереди элементов работы

IoQueueWorkItem

IoQueueWorkItem ассоциирует WorkItem с рабочим элементом и вставляет его в очередь для дальнейшей обрабоки системным процессом.

```
VOID IoQueueWorkItem(
  _In_     PIO_WORKITEM         IoWorkItem,
  _In_     PIO_WORKITEM_ROUTINE WorkerRoutine,
  _In_     WORK_QUEUE_TYPE      QueueType,
  _In_opt_ PVOID                Context
```

```
);
```

Параметры

IoWorkItem [in] - Указатель на структуру IO_WORKITEM, которая резервируется с помощью IoAllocateWorkItem или инициализируется с помощью IoInitializeWorkItem.

WorkerRoutine [in] - указатель на WorkItem.

QueueType [in] - определяет значение WORK_QUEUE_TYPE, которое обусловливает тип системного потока для управления рабочим элементом. Драйвера должны определять DelayedWorkQueue.

Context [in, optional] - определяет драйверную информацию для рабочего элемента.

40. Управление памятью в ОС Windows. Менеджер памяти. Виртуальная память процесса. Управление памятью в пользовательском режиме. Страничная виртуальная память. Куча (свалка, heap). Проецирование файлов в память.

Менеджер памяти.
Основные функции менеджера памяти: отображение адресного пространства процесса на конкретные области физической памяти (размещение); распределение памяти между конкурирующими процессами (выборка); контроль доступа к адресным пространствам процессов; выгрузка процессов (целиком или частично) во внешнюю память, когда в оперативной памяти недостаточно места (замещение); учет свободной и занятой памяти.

Аппаратно Memory Manager реализован на Memory Management Unit (MMU), являющимся частью процессора. Принцип работы современных MMU основан на разделении виртуального адресного пространства (одномерного массива адресов, используемых центральным процессором) на участки одинакового, как правило несколько килобайт. Младшие n бит адреса (смещение внутри страницы) остаются неизменными. Старшие биты адреса представляют собой номер (виртуальной) страницы. MMU обычно преобразует номера виртуальных страниц в номера физических страниц используя буфер ассоциативной трансляции.

Виртуальная память процесса.
У каждого процесса есть своя собственная виртуальная память, именуемая адресным пространством, в которой исполняется код этот процесса и его данные, на которые этот код ссылается и которыми управляет. 32-битные процессы используют 32-битные указатели на адреса в виртуальной памяти, которые создают абсолютный верхний предел в 4 ГБ на объем виртуальной памяти, которую 32-битный процесс может адресовать. Однако, чтобы операционная система могла обратиться к своему собственному коду и данным и к коду и данным, выполняющегося в настоящее время процесса, без необходимости изменять адресное пространство, она делает свою виртуальную память видимой из адресных пространств всех процессов. По умолчанию 32-битная версия Windows разделяет адресное пространство процесса поровну между системой и активным процессом, создавая границу в 2 Гб для каждого.

Некоторые приложения управляют большими структурами данных, объем которых намного превышает доступное для них адресное пространство. Windows поддерживает параметры загрузки (спецификатор increaseuserva в базе данных конфигурации загрузки — Boot Configuration Database.), которые дают процессам, выполняющим специально помеченные программы (в заголовке исполняемого образа должен быть установлен флаг признака большого адресного пространства), возможность использования до 3 Гбайт закрытого адресного пространства (оставляя 1 Гбайт для операционной системы).

Часть виртуальной памяти процесса, которая находится резидентно в физической памяти, называется рабочим набором – Working Set. Диапазон рабочего набора устанавливается функцией SetProcessWorkingSetSize(). Стандартный минимальный рабочий набор – 50 страниц по 4 КБ (200 КБ), стандартный максимальный рабочий набор – 345 страниц по 4 КБ (1380 КБ).

Управление памятью в пользовательском режиме.
В пользовательском режиме (user mode) доступ к регистрам и памяти ограничен. Приложению не будет позволено работать с памятью за пределами набора адресов, установленного ОС, или обращаться напрямую к регистрам устройств.

<u>Страничная виртуальная память:</u>
Выделение: VirtualAlloc(), VirtualAllocEx(), VirtualAllocExNuma(), VirtualFree(), VirtualFreeEx(). Гранулярность в user mode – 64 КБ.
Защита страниц: VirtualProtect(), VirtualProtectEx().

Фиксация страниц в физической памяти: VirtualLock(), VirtualUnlock().
Информация: VirtualQuery(), VirtualQueryEx().

Куча (свалка) – Heap:
Создание: HeapCreate(), HeapDestroy().
Выделение: HeapAlloc(), HeapReAlloc(), HeapSize(), HeapFree(). Гранулярность – 8 байтов на x86, 16 байтов на x64.
Информация: HeapValidate(), HeapWalk(), HeapQueryInformation(), HeapSetInformation().
Кучи процесса: GetProcessHeap() – стандартная куча равная 1 MB, GetProcessHeaps() – все кучи процесса.

Проецирование файлов в память – File Mapping:
Объект ядра, описывающий отображение фрагмента файла в диапазон виртуальных адресов, называется разделом (Section Object).

Страничная виртуальная память.
В случае страничной организации памяти и виртуальная и физическая память представляются в виде набора неперекрывающихся блоков одинакового размера, называемых страницами. Передача информации (считывание, запись) всегда осуществляется целыми страницами.

При страничной организации виртуальный адрес памяти требуемого элемента задается в виде номера страницы и смещения относительно начала страницы. Любой выполняемый процесс (программа) имеет дело только с виртуальными адресами и не знает физических адресов данных, с которыми работает. Для преобразования виртуальных адресов в физические используются таблицы страниц (Page Walk), размещаемые в оперативной памяти. Важно, что каждому процессу соответствует собственная таблица страниц.

Преобразование логических (виртуальных) адресов в физические происходит следующим образом. Когда какой-либо выполняемый процесс обращается по виртуальному адресу, в котором содержится информация о номере требуемой страницы и смещении в пределах страницы, происходит обращение к таблице страниц этого процесса, в которой каждому номеру страницы поставлен в соответствие физический адрес страницы в памяти.

Таким образом, по номеру страницы определяется физический адрес этой страницы в памяти. Далее с учетом известного смещения в пределах требуемой страницы определяется физический адрес искомого элемента памяти

Любой процесс может выполняться только в том случае, если используемые им страницы памяти размещаются в оперативной памяти. При отсутствии запрашиваемой страницы в оперативной памяти возникает исключительная ситуация — страничное нарушение (page fault). Тогда затребованная страница подкачивается из внешней памяти (swap-файла) в свободный страничный кадр физической памяти, а при отсутствии свободных страничных кадров в оперативной памяти первоначально в swap-файл выгружается мало используемая страница памяти.

При страничной организации памяти может возникать проблема ее внутренней фрагментации (внешней фрагментации в данном случае принципиально не существует). Внутренняя фрагментация памяти происходит по причине того, что адресное пространство процесса может занимать только целое число страниц, при этом некоторые страницы заняты не полностью.

Важно отметить, что при страничной организации памяти любому процессу доступна лишь та физическая память, которая ему соответствует, и не доступна память другого процесса, поскольку процесс не имеет возможности адресовать память за пределами своей таблицы страниц, включающей только его собственные страницы.

Проецирование файлов в память.
Отображение файла в память — это такой способ работы с файлами, при котором всему файлу или некоторой непрерывной части этого файла ставится в соответствие определённый участок памяти (диапазон адресов оперативной памяти). При этом чтение данных из этих адресов фактически приводит к чтению данных из отображенного файла, а запись данных по этим адресам приводит к записи этих данных в файл. Примечательно то, что отображать на память часто можно не только обычные файлы, но и файлы устройств.

После запуска процесса операционная система отображает его файл на память, для которой разрешено выполнение (атрибут executable). Большинство систем, использующих отображение файлов используют методику загрузка страницы по первому требованию, при которой файл загружается в память не целиком, а небольшими частями, размером со страницу памяти, при этом страница загружается только тогда, когда она действительно нужна.

Другой общеупотребимый случай использования отображений — создание разделяемых несколькими процессами фрагментов памяти. Процесс в защищенном режиме, вообще говоря, не позволяет другим процессам обращаться к «своей» памяти. Программы, которые пытаются обратиться не к своей памяти генерируют исключительные ситуации invalid page faults или segmentation violation. Есть несколько способов безопасно (без возникновения исключительных ситуаций) сделать память доступной нескольким процессам, и использование файлов, отображенных на память — один из наиболее популярных способов сделать это. Два или более приложений могут одновременно отобразить один и тот же физический файл на свою память и обратиться к этой памяти.

41. Управление памятью в пользовательском режиме ОС Windows. Оптимизация работы кучи с помощью списков предыстории (Look-aside Lists) и низко-фрагментированной кучи (Low Fragmentation Heap).

У каждого процесса имеется как минимум одна куча — куча процесса, предлагаемая по умолчанию. Эта куча создается при запуске процесса и не удаляется, пока существует процесс. Ее размер по умолчанию составляет 1 Мбайт, но ее можно сделать больше, указав начальный размер в файле образа с помощью флага /HEAP компоновщика. Однако этот размер является всего лишь начальным, и по мере необходимости он автоматически увеличивается. (В файле образа можно также указать изначально подтвержденный размер.)

Куча, предлагаемая по умолчанию, может быть явно использована программой или неявно какой-нибудь из внутренних Windows-функций. Приложение может послать запрос к куче процесса, предлагаемой по умолчанию, вызвав Windows-функцию GetProcessHeap. Процессы могут также создавать дополнительные закрытые кучи, используя для этого функцию HeapCreate. Когда закрытая куча становится процессу ненужной, он может вернуть виртуальное адресное пространство, вызвав функцию HeapDestroy. Каждым процессом обслуживается массив со сведениями обо всех кучах, и программный поток может запросить их с помощью Windows-функции GetProcessHeaps.

Управление выделениями памяти для кучи может осуществляться либо в больших областях памяти, зарезервированных из диспетчера памяти с помощью функции VirtualAlloc, либо из объектов отображаемых на память файлов в адресном пространстве процесса. Последний подход используется на практике довольно редко, но хорошо подходит для сценариев, согласно которым содержимое блоков должно совместно использоваться двумя процессами или компонентами, работающими в режиме ядра и в пользовательском режиме.

Списки предысторий (look-aside lists)

Применяются менеджером кучи для выделения-освобождения элементов фиксированного размера. В ядре могут явно применяться драйверами для проведения операций ввода-вывода над фиксированными участками памяти (используя ExXxxLookasideListEx или ExXxxLookasideList). После того, как драйвер инициализирует список предыстории, ОС хранит несколько динамически распределенных буферов заданного размера (доступных для повторного использования). Формат и содержание данных буферов (также известных как записи) находится в списке предыстории определенного драйвера.

ОС управляет состоянием всех страничных и нестраничных списков предысторий, динамически отслеживая потребность в расположении и перерасположении записей в списках. Когда потребность в расположении записи высокая, ОС увеличивает число записей, которые она хранит в каждом из списков. Когда потребность исчезает, она снова высвобождает запись и возвращает ее в системный пул.

Списки предысторий потокобезопасны. Список содержит встроенную синхронизацию для разрешения нескольким параллельно работающим потокам получать доступ к списку. Тем не менее, для предотвращения возможной утечки и нарушения данных, потоки которые разделяют между собой список должны синхронизировать инициализацию и удаление списка.

Структура *PAGED_LOOKASIDE_LIST* описывает списки предысторий, содержащие страничные буферы. Следующие системные подпрограммы поддерживают работу со списками предысторий, которые описываются с помощью структуры *PAGED_LOOKASIDE_LIST*:
ExAllocateFromPagedLookasideList
ExDeletePagedLookasideList
ExFreeToPagedLookasideList
ExInitializePagedLookasideList

Для работы со списками, содержащими нестраничные буферы, используется структура *NPAGED_LOOKASIDE_LIST*. Следующие системные подпрограммы поддерживают работу со списками предысторий, которые описываются с помощью структуры *NPAGED_LOOKASIDE_LIST:*
ExAllocateFromNPagedLookasideList
ExDeleteNPagedLookasideList
ExFreeToNPagedLookasideList
ExInitializeNPagedLookasideList

Списки представлены в виде 128 связных списков свободных блоков. Каждый список содержит элементы строго определенного размера – от 8 байтов до 1 КБ на x86 и от 16 байтов до 2 КБ на x64.

Когда блок памяти освобождается, он помещается в список предыстории, соответствующий его размеру. Затем, если запрашивается блок памяти такого же размера, он берется из списка предыстории методом LIFO. Для организации списка предыстории используются функции InterlockedPushEntrySList() и InterlockedPopEntrySList().

Раз в секунду ОС уменьшает глубину списков предыстории с помощью функции ядра KiAdjustLookasideDepth().

Слабо фрагментированная куча
Многие выполняемые в Windows приложения используют относительно небольшой объем памяти кучи (обычно менее 1 Мбайт). Для этого класса приложений сохранять малый объем памяти для каждого процесса помогает оптимальная политика диспетчера кучи. Но эта стратегия для больших процессов и мультипроцессорных машин не масштабируется. В таких случаях память, доступная для использования в куче, может быть уменьшена в результате фрагментации кучи. В сценариях, где параллельно выполняются разные потоки на разных процессорах, производительность может снижаться.

Причина в том, что нескольким процессорам одновременно нужно модифицировать один и тот же участок памяти (например, начало ассоциативного списка для блока конкретного размера), создавая тем самым большую конкуренцию при доступе к соответствующей строке кэша.

Технология слабо фрагментированной кучи (LFH) позволяет избегать фрагментации путем управления выделенными блоками в заранее заданных диапазонах размеров блоков, которые называются корзинами (buckets). Когда процесс выделяет память из кучи, LFH предлагает корзину, отображаемую на наименьший блок, подходящий под требуемый размер. (Наименьший блок имеет размер 8 байт.) Первая корзина служит для выделения в диапазоне от 1 до 8 байт, вторая — в диапазоне от 9 до 16 байт и т. д., вплоть до тридцать второй корзины, предназначенной для выделения в диапазоне от 249 до 256 байт, за которой следует тридцать третья корзина, служащая для выделения в диапазоне от 257 до 272 байт и т. д. И наконец, очередь доходит до сто двадцать восьмой корзины, которая в итоге используется для выделения в диапазоне от 15 873 до 16 384 байт. Все это известно как система двоичного дружелюбия (binary buddy).

В LFH эти проблемы решаются путем использования диспетчера основной кучи и ассоциативных списков. Имеющийся в Windows диспетчер кучи реализует алгоритм автоматической настройки, который может по умолчанию подключить LFH при определенных условиях, таких как конфликты блокировки или наличие широко распространенных размеров выделения памяти, при работе с которыми подключение LFH способствует более высокой производительности системы. Для больших куч существенный процент операций выделения памяти часто касается относительно небольшого количества корзин определенных размеров. Стратегия выделения памяти, принятая в LFH, заключается в оптимизации использования таких моделей путем эффективной работы с блоками одинакового размера.

В целях обеспечения масштабируемости LFH расширяет часто используемые внутренние структуры на ряд слотов, количество которых вдвое превышает текущее количество процессоров, имеющихся на машине. Назначение потоков этим слотам осуществляется LFH-компонентом, который называется диспетчером

родственности (affinity manager). Изначально LFH задействует для выделения кучи первый слот, но если при обращении к каким-либо внутренним данным обнаруживается конфликтная ситуация, LFH переключает текущий программный поток на другой слот. Последующие конфликтные ситуации ведут к «расползанию» потоков на дополнительные слоты. Для лучшей локальности и минимизации общего потребления памяти управление этими слотами осуществляется для корзин каждого размера.

Даже если слабо фрагментированная куча подключена к куче в качестве внешнего уровня, в случаях наименее востребованных размеров для выделения памяти могут по-прежнему использоваться соответствующие функции основной кучи, а для наиболее востребованных размеров выделение памяти будет осуществляться из LFH. Отключить LFH можно с помощью API-функции HeapSetInformation с классом HeapCompatibilityInformation.

42. Структура виртуальной памяти в ОС Windows. Виды страниц. Состояния страниц. Структура виртуального адреса. Трансляция виртуального адреса в физический. Кэширование виртуальных адресов.

Виды и состояния страниц

Виртуальная память состоит из двух типов страниц:
Малые страницы – 4 КБ. Большие страницы – 4 МБ на x86 или 2 МБ на x64. Большие страницы используются для Ntoskrnl.exe, Hal.dll, данных ядра, описывающих резидентную память ядра и состояние физических страниц памяти.
При вызове VirtualAlloc() можно указать флаг MEM_LARGE_PAGE. Для всей страницы применяется единый режим защиты и блокировки памяти.

Страницы в виртуальном адресном пространстве процесса могут быть свободными (free), зарезервированными (reserved), подтвержденными (committed) или общими (shareable).

Подтвержденными и общими являются страницы, при обращении к которым в итоге происходит преобразование с указанием настоящих страниц в физической памяти. Подтвержденные страницы называются также закрытыми (private) — это название отражает тот факт, что они, в отличие от общих страниц, не могут использоваться совместно с другими процессами (а могут, разумеется, использоваться только одним процессом).

Закрытые страницы выделяются с помощью Windows-функций VirtualAlloc, VirtualAllocEx и VirtualAllocExNuma. Эти функции позволяют программному потоку резервировать адресное пространство, а затем подтверждать части зарезервированного пространства. Промежуточное «зарезервированное» состояние позволяет потоку отложить непрерывный диапазон виртуальных адресов для возможного использования в будущем (например, в качестве массива), потребляя при этом незначительные системные ресурсы, а затем выделить подтвержденные части зарезервированного пространства, как только это потребуется для выполнения приложения. Или же, если требуемые объемы известны заранее, поток может выполнить резервирование и подтверждение в одном вызове функции. В любом случае, подтвержденные в итоге страницы затем становятся доступны потоку. Попытки обращения к свободной или зарезервированной памяти приводят к исключению, поскольку страница не отображена ни на одно из хранилищ, способных разрешить ссылку.

Если к подтвержденным (закрытым) страницам до этого еще не было обращений, они создаются во время первого обращения в качестве страниц, подлежащих заполнению нулевыми байтами. Закрытые (подтвержденные) страницы могут быть позже автоматически записаны операционной системой в страничный файл (файл подкачки), если это потребуется для удовлетворения спроса.

При первом обращении к общей странице со стороны какого-нибудь процесса она считывается из связанного отображаемого файла (если только раздел не связан с файлом подкачки, тогда страница создается заполненной нулевыми байтами). Позже, если она все еще находится в физической памяти, то есть является резидентной (resident), при повторном и последующих обращениях процессов можно просто использовать то же самое содержимое страницы, которое уже находится в памяти.

Общие страницы могут также заранее извлекаться из памяти самой системой.

Структура виртуального адреса

Номер таблицы страниц в каталоге таблиц (Page directory index - Page Directory Entry).
Номер страницы в таблице страниц (Page table index - Page Table Entry).
Смещение в странице – Byte index.

Трансляция виртуального адреса в физический

Трансляция виртуального адреса – это определение реального (физического) расположение ячейки памяти с данным виртуальным адресом, т. е. преобразование виртуального адреса в физический.

Информация о соответствии виртуальных адресов физическим хранится в таблицах страниц. В системе для каждого процесса поддерживается множество записей о страницах: если размер страницы 4 КБ, то чтобы хранить информацию обо всех виртуальных страницах в 32 разрядной системе требуется более миллиона записей (4 ГБ / 4 КБ = 1 048 576). Эти записи о страницах сгруппированы в таблицы страниц (Page Table), запись называется PTE (Page Table Entry). В каждой таблице содержится 1024 записи, таким образом, максимальное количество таблиц страниц для процесса – 1024 (1 048 576 / 1024 = 1024). Половина от общего количества – 512 таблиц – отвечают за пользовательское виртуальное адресное пространство, другая половина – за системное виртуальное адресное пространство.

Таблицы страниц хранятся в виртуальной памяти. Информация о расположении каждой из таблиц страниц находится в каталоге страниц (Page Directory), единственном для процесса. Записи этого каталога называются PDE (Page Directory Entry). Таким образом, процесс трансляции является двухступенчатым: сначала по виртуальному адресу определяется запись PDE в каталоге страниц, затем по этой записи находится соответствующая таблица страниц, запись PTE которой указывает на требуемую страницу в физической памяти.

Кэширование виртуальных адресов

Буфер ассоциативной трансляции (TLB) — это специализированный кэш центрального процессора, используемый для ускорения трансляции адреса виртуальной памяти в адрес физической памяти. TLB используется всеми современными процессорами с поддержкой страничной организации памяти. TLB содержит фиксированный набор записей (от 8 до 4096) и является ассоциативной памятью. Каждая запись содержит соответствие адреса страницы виртуальной памяти адресу физической памяти.

В современных процессорах может быть реализовано несколько уровней TLB с разной скоростью работы и размером. Самый верхний уровень TLB будет содержать небольшое количество записей, но будет работать с очень высокой скоростью, вплоть до нескольких тактов. Последующие уровни становятся медленнее, но, вместе с тем и больше.

Иногда верхний уровень TLB разделяется на 2 буфера, один для страниц, содержащих исполняемый код, и другой — для обрабатываемых данных.

Следует упомянуть, что хотя большинство кэшей физически тэгированы, некоторые могут быть виртуально тэгированы. Это решение увеличивает производительность, так как отсутствует необходимость в просмотре TLB, но привносит серьёзную проблему, связанную с дублированием строк. Если две задачи используют одну и ту же физическую строку, находящуюся под разными адресами в их виртуальных пространствах, и одна из задач изменяет свою копию, то вторая задача может оставить этот факт без должного внимания и продолжать использовать свою устаревшую копию. Самым простым способом избежания этой проблемы является сброс кэш-памяти (cache flush) при каждом переключении между задачами, но этот вариант наименее приемлем с точки зрения быстродействия. Иначе можно снабдить каждую строку дополнительным полем, известным как ASID (Address Space IDentifier) или RID (Region IDentifier).

43. Управление памятью в режиме ядра ОС Windows. Пулы памяти. Выделение и освобождение памяти в пулах памяти. Структура описателя пула памяти. Доступ к описателям пулов памяти на однопроцессорной и многопроцессорной системах.

Пул памяти (Memory Pool) – динамически расширяемая область виртуальной памяти в режиме ядра, в которой драйверы и ядро выделяют для себя память.

Существуют два типа пулов памяти:

1. Пул резидентной памяти (Non-Paged Pool), в ядре один.
2. Пул страничной памяти (Paged Pool). На многопроцессорных системах в ядре 5 страничных пулов, на однопроцессорных системах – 3 пула.

Дополнительно операционная система создает и поддерживает:

1. Страничный пул сеансовой памяти (Session Pool).
2. Специальный отладочный пул (Special Pool), состоящий из резидентной и страничной памяти.
3. Пул резидентной памяти, защищенный от исполнения кода (No-Execute Non-Paged Pool – NX Pool). Начиная с Windows 8, все драйверы должны держать резидентные данные именно в этом пуле.

Резидентный и страничные пулы растут до установленного максимума:

1. Non-Paged Pool – 256 МБ на x86, 128 ГБ на x64.
2. Paged Pool – 2 ГБ на x86, 128 ГБ на x64.

Выделение памяти в пуле:

1. ExAllocatePoolWithTag(POOL_TYPE PoolType, SIZE_T NumberOfBytes, ULONG Tag);

PoolType – тип выделяемого пула памяти. Определяется с помощью структуры POOL_TYPE.

NumberOfBytes – число выделяемых байтов.

Tag – тэг пула, использующийся для выделенной памяти. Представляет из себя строковый литерал не более 4 символов, заключённых в одиночные кавычки. Например ‘Tag1’, хранится как строка в обратном порядке “1gaT”. Каждый ASCII символ, входящий в тэг должен иметь значение от 0x20 (пробел) до 0x126 (тильда).

ExAllocatePoolWithTag возвращает NULL если в пуле недостаточно свободной памяти для удовлетворения запроса. Иначе возвращает указатель на выделенную память.

Если NumberOfBytes имеет значение PAGE_SIZE или большее, то выделяется выровненный по границе страницы буфер. Если меньше, то выделяется буфер без выравнивания по границе страницы и без пересечения границ страниц. Выделение памяти меньше чем PAGE_SIZE не требует выравнивание по границе страницы, но выравнивается по 8-байтовой границе для x32 систем и по 16-байтовой границе для x64 систем.

2. ExAllocatePoolWithQuota(POOL_TYPE PoolType, SIZE_T NumberOfBytes);

ExAllocatePoolWithQuota возвращает указатель на выделенный участок памяти. Если запрос удовлетворить не удаётся выбрасывается исключение.

При NumberOfBytes больше, меньше или равном PAGE_SIZE работает аналогично с ExAllocatePoolWithTag.

Эта функция вызывается высокоуровневыми драйверами, которые выделяют память для удовлетворения запросов в контексте процесса, выполняющего I/O запрос. Низкоуровневые драйвера используют ExAllocatePoolWithTag.

3. ExAllocatePoolWithQuotaTag(POOL_TYPE PoolType, SIZE_T NumberOfBytes, ULONG Tag);

Возвращает указатель на выделенную память и выбрасывает исключение в случае неудачи, если не определена POOL_QUOTA_FAIL_INSTEAD_OF_RAISE.

Используется высокоуровневыми драйверами, которые выделяют память для удовлетворения запросов в контексте процесса, выполняющего I/O запрос. Низкоуровневые драйвера используют ExAllocatePoolWithTag.

4. ExAllocatePool(POOL_TYPE PoolType, SIZE_T NumberOfBytes);

Возвращает NULL, если невозможно выделить память. В случае успеха возвращает указатель на выделенную память.

5. ExAllocatePoolWithTagPriority(POOL_TYPE PoolType, SIZE_T NumberOfBytes, ULONG Tag, EX_POOL_PRIORITY Priority);

Priority – приоритет запроса. Может принимать одно из следующих значений:

a. LowPoolPriority – система может отменить запрос, если он выполняется медленно или использует много ресурсов

b. NormalPoolPriority – система может отменить запрос, если он выполняется очень медленно или используетс очень много ресурсов. Большинство драйверов используют этот приоритет.

c. HighPoolPriority – система не может прервать этот запрос, пока есть ресурсы.

Возвращает NULL если не удалось выделить память и не определен POOL_RAISE_IF_ALLOCATION_FAILURE. В случае успеха возвращает указатель на выделенную память.

Освобождение памяти в пуле:

1. ExFreePoolWithTag(void* P, ULONG Tag);
2. ExFreePool(void* P).

Структура описателя пула памяти (POOL_DESCRIPTOR):

```
struct POOL_DESCRIPTOR {
      POOL_TYPE PoolType;
      KGUARDED_MUTEX PagedLock;
      ULONG NonPagedLock;
      LONG RunningAllocs;
      LONG RunningDeAllocs;
      LONG TotalBigPages;
      LONG ThreadsProcessingDereferrals;
      ULONG TotalBytes;
      ULONG PoolIndex;
      LONG TotalPages;
      VOID** PendingFrees;
      LONG PendingFreeDepth;
      LIST_ENTRY ListHeads[512];
};
```

Доступ к описателям пулов на однопроцессорной системе:

Переменная nt!PoolVector хранит массив указателей на описатели пулов. Первый указывает на описатель резидентного пула. Остальные указывают на пять описателей страничных пулов. Доступ к ним можно получить через переменную nt!ExpPagedPoolDescriptor. Это массив из 5 указателей на описатели страничных пулов. Количество страничных пулов хранится в переменной nt!ExpNumberOfPagedPools.

Доступ к описателям пулов на многопроцессорной системе:

Каждый NUMA-узел описывается структурой KNODE, в которой хранятся указатели на свой резидентный пул и свой страничный пул NUMA-узла. Указатель на структуру KNODE можно получить из массива nt!KeNodeBlock, в котором хранятся указатели на все KNODE-структуры NUMA-узлов. Указатели на описатели резидентных пулов всех NUMA-узлов хранятся в массиве nt!ExpNonPagedPoolDescriptor. Количество всех резидентных пулов в системе определяется переменной nt!ExpNumberOfNonPagedPools. Указатели на описатели страничных пулов всех NUMA-узлов хранятся в массиве nt!ExpPagedPoolDescriptor (по одному на NUMA-узел плюс один). Количество всех страничных пулов в системе определяется переменной nt!ExpNumberOfPagedPools.

44. Пулы памяти ОС Windows. Пул подкачиваемой памяти, пул неподкачиваемой памяти, пул сессии, особый пул. Тегирование пулов. Структура данных пула. Выделение и освобождение памяти в пулах памяти. Организация списков свободных блоков в пуле памяти. Заголовок блока пула памяти.

Пул памяти (Memory Pool) – динамически расширяемая область виртуальной памяти в режиме ядра, в которой драйверы и ядро выделяют для себя память.

Существуют два типа пулов памяти:

1. Пул резидентной памяти (пул не подкачиваемой памяти, Non-Paged Pool), в ядре один. Ядро и драйвера устройств используют пул не подкачиваемой памяти для хранения информации, к которой можно получить доступ, если система не может обработать paged fault.
2. Пул страничной памяти (пул подкачиваемой памяти, Paged Pool). На многопроцессорных системах в ядре 5 страничных пулов, на однопроцессорных системах – 3 пула. Пул подкачиваемой памяти получил своё название из-за того, что Windows может записывать хранимую информацию в файл подкачки, тем самым позволяя повторно использовать занимаемую физическую память. Для пользовательского режима виртуальной памяти, когда драйвер или система ссылается на пул подкачиваемой памяти, который находится в файле подкачки, операция вызывает прерывание и менеджер памяти читает информацию назад в физическую память.

Дополнительно операционная система создает и поддерживает:

1. Страничный пул сеансовой памяти (Session Pool). Особый страничный пул, в котором располагаются данные сессии.
2. Специальный отладочный пул (Special Pool), состоящий из резидентной и страничной памяти. Страничный/не страничный с возможностью обнаружения нарушения целостности данных (используется для отладки).
3. Пул резидентной памяти, защищенный от исполнения кода (No-Execute Non-Paged Pool – NX Pool). Начиная с Windows 8, все драйверы должны держать резидентные данные именно в этом пуле.

Резидентный и страничные пулы растут до установленного максимума:

1. Non-Paged Pool – 256 МБ на x86, 128 ГБ на x64.
2. Paged Pool – 2 ГБ на x86, 128 ГБ на x64.

Выделение памяти в пуле:

1. ExAllocatePoolWithTag(POOL_TYPE PoolType, SIZE_T NumberOfBytes, ULONG Tag);

PoolType – тип выделяемого пула памяти. Определяется с помощью структуры POOL_TYPE.

NumberOfBytes – число выделяемы

Tag – тэг пула, использующийся для выделенной памяти. Представляет из себя строковый литерал не более 4 символов, заключённых в одиночные кавычки. Например ‘Tag1’, но определяется она обычно в обратном порядке, т. е. ‘1gaT’. Каждый ASCII символ, входящий в тэг должен иметь значение от 0x20 (пробел) до 0x126 (тильда).

ExAllocatePoolWithTag возвращает NULL если в полу недостаточно свободной памяти для удовлетворения запроса. Иначе возвращает указатель на выделенную память.

Если NumberOfBytes имеет значение PAGE_SIZE или большее, то выделяется выровненный по границе страницы буфер. Если меньше, то выделяется буфер без выравнивания по границе страницы и без пересечения границ страниц. Выделение памяти меньше чем PAGE_SIZE не требует выравнивание по границе страницы, но выравнивается по 8-байтовой границе для x32 систем и по 16-байтовой границе для x64 систем.

2. ExAllocatePoolWithQuota(POOL_TYPE PoolType, SIZE_T NumberOfBytes);

ExAllocatePoolWithQuota возвращает указатель на выделенный участок памяти. Если запрос удовлетворить не удаётся выбрасывается исключение.

При NumberOfBytes больше, меньше или равном PAGE_SIZE работает аналогично с ExAllocatePoolWithTag.

Эта функция вызывается высокоуровневыми драйверами, которые выделяют память для удовлетворения запросов в контексте процесса, выполняющего I/O запрос. Низкоуровневые драйвера используют ExAllocatePoolWithTag.

3. ExAllocatePoolWithQuotaTag(POOL_TYPE PoolType, SIZE_T NumberOfBytes, ULONG Tag);

Возвращает указатель на выделенную память и выбрасывает исключение в случае неудачи, если не определена POOL_QUOTA_FAIL_INSTEAD_OF_RAISE.

Используется высокоуровневыми драйверами, которые выделяют память для удовлетворения запросов в контексте процесса, выполняющего I/O запрос. Низкоуровневые драйвера используют ExAllocatePoolWithTag.

4. ExAllocatePool(POOL_TYPE PoolType, SIZE_T NumberOfBytes);

Возвращает NULL, если невозможно выделить память. В случае успеха возвращает указатель на выделенную память.

5. ExAllocatePoolWithTagPriority(POOL_TYPE PoolType, SIZE_T NumberOfBytes, ULONG Tag, EX_POOL_PRIORITY Priority);

Priority – приоритет запроса. Может принимать одно из следующих значений:

a. LowPoolPriority – система может отменить запрос, если он выполняется медленно или использует много ресурсов

b. NormalPoolPriority – система может отменить запрос, если он выполняется очень медленно или используетс очень много ресурсов. Большинство драйверов используют этот приоритет.

c. HighPoolPriority – система не может прервать этот запрос, пока есть ресурсы.

Возвращает NULL если не удалось выделить память и не определен POOL_RAISE_IF_ALLOCATION_FAILURE. В случае успеха возвращает указатель на выделенную память.

Освобождение памяти в пуле:

1. ExFreePoolWithTag(void* P, ULONG Tag);
2. ExFreePool(void* P).

Структура данных пула:

1. Система содержит структуру POOL_DESCRIPTOR для каждого пула
2. Переменная PoolVector содержит массив указателей, которые указывают на дескрипторы страничных/нестраничных пулов

a. Точка входа страничного пула указывает на массив дескрипторов пула

b. Точка входа нестраничного пула указывает на единственный дескриптор пула

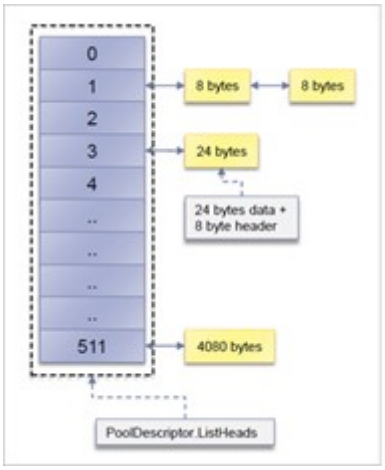
3. Блокам пула предшествует структура POOL_HEADER которая содержит отслеживаемую информацию о выделенных и свободных блоках
4. POOL_HEADER содержит:

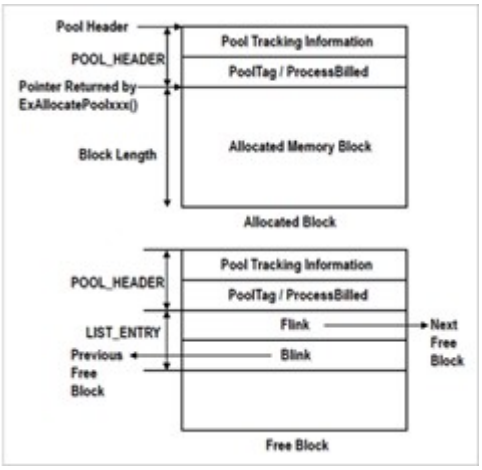
a. Тег пула для блока пула

b. Указатель EPROCESS, который является квотой, взимаемой за блок пула

c. Индекс пула, которому блок принадлежит (только для страничных пулов)

d. Размер блока пула и размер предыдущего блока





В описателе пула содержится массив ListHeads: это 512 двусвязных списков свободных блоков. В каждом списке находятся блоки строго определенного размера от 8 до 4088 байтов с шагом 8 байтов (полезный размер от 0 до 4080). Гранулярность определяется наличием заголовка POOL_HEADER.

Каждый блок памяти имеет заголовок POOL_HEADER, в котором содержится следующая информация:

1. PreviousSize – размер предыдущего блока.
2. PoolIndex – индекс пула в массиве описателей, которому блок принадлежит.
3. BlockSize – размер блока: (NumberOfBytes + 0xF) >> 3 на x86 или (NumberOfBytes + 0x1F) >> 4 на x64.
4. PoolType – тип пула (резидентный, страничный, сеансовый, т.д.).
5. PoolTag – тег драйвера, выделившего блок.

6. ProcessBilled – указатель на процесс (структуру EPROCESS), из квоты которого выделен блок (только на x64).

45. Управление памятью в режиме ядра ОС Windows. Оптимизация использования оперативной памяти с помощью списков предыстории – Look-aside Lists.

Пул памяти (Memory Pool) – динамически расширяемая область виртуальной памяти в режиме ядра, в которой драйверы и ядро выделяют для себя память.

Существуют два типа пулов памяти:

1. Пул резидентной памяти (Non-Paged Pool), в ядре один.
2. Пул страничной памяти (Paged Pool). На многопроцессорных системах в ядре 5 страничных пулов, на однопроцессорных системах – 3 пула.

Дополнительно операционная система создает и поддерживает:

1. Страничный пул сеансовой памяти (Session Pool).
2. Специальный отладочный пул (Special Pool), состоящий из резидентной и страничной памяти.
3. Пул резидентной памяти, защищенный от исполнения кода (No-Execute Non-Paged Pool – NX Pool). Начиная с Windows 8, все драйверы должны держать резидентные данные именно в этом пуле.

Резидентный и страничные пулы растут до установленного максимума:

1. Non-Paged Pool – 256 МБ на x86, 128 ГБ на x64.
2. Paged Pool – 2 ГБ на x86, 128 ГБ на x64.

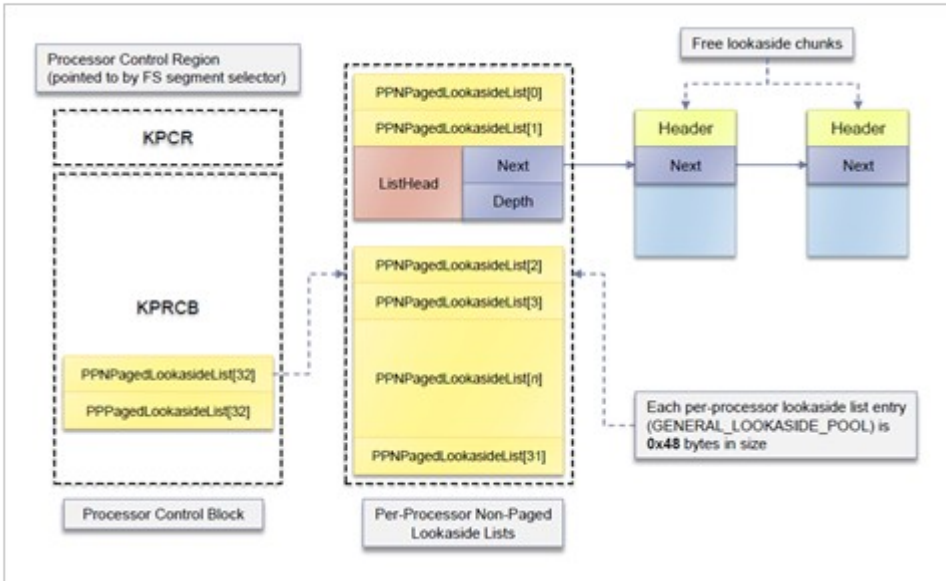
Выделение памяти в пуле:

1. ExAllocatePoolWithTag(POOL_TYPE PoolType, SIZE_T NumberOfBytes, ULONG Tag);
2. ExAllocatePoolWithQuota(POOL_TYPE PoolType, SIZE_T NumberOfBytes);
3. ExAllocatePoolWithQuotaTag(POOL_TYPE PoolType, SIZE_T NumberOfBytes, ULONG Tag);
4. ExAllocatePool(POOL_TYPE PoolType, SIZE_T NumberOfBytes);
5. ExAllocatePoolWithTagPriority(POOL_TYPE PoolType, SIZE_T NumberOfBytes, ULONG Tag, EX_POOL_PRIORITY Priority);

Освобождение памяти в пуле:

1. ExFreePoolWithTag(void* P, ULONG Tag);
2. ExFreePool(void* P).

Списки предыстории:



На каждый процессор создается набор списков предыстории:

1. 32 списка предыстории на процессор.
2. В каждом списке – свободные блоки строго определенного размера

от 8 до 256 байтов с гранулярностью 8 (x86).

Функции работы со списками предыстории:

1. ExAllocateFromPagedLookasideList(PPAGED_LOOKASIDE_LIST Lookaside);

Lookaside – указатель на структуру PAGED_LOOKASIDE_LIST, которую вызывающий инициализирует с помощью ExInitializePagedLookasideList.
В случае неудачи возвращает NULL. В противном случае возвращает указатель на первый элемент. Если список не пустой функция удалит первую запись и вернёт указатель на неё.
2. ExInitializePagedLookasideList(PPAGED_LOOKASIDE_LIST Lookaside, PALLOCATE_FUNCTION Allocate, PFREE_FUNCTION Free, ULONG Flags, SIZE_T Size, ULONG Tag, USHORT Depth);
Lookaside – указатель на структуру PAGED_LOOKASIDE_LIST
Allocate – указатель на функцию, которая вызывается, если список предыстории пуст. Может быть NULL. Если NULL будет вызвана функция ExAllocateFromPagedLookasideList. Пользовательская функция должна соответствовать прототипу: void* XxxAllocate (POOL_TYPE PoolType, SIZE_T NumberOfBytes, ULONG Tag);
Free – указатель на функцию, которая вызывается для освобождения элемента списка, если список полный. Если NULL, то вызывается ExFreeToPagedLookasideList. Пользовательская функция должна соответствовать прототипу: void XxxFree(PVOID Buffer);
Flag – начиная с Windows 8, этот параметр определяет значение для модификации стандартного поведения ExInitializePagedLookasideList. До Windows 8 не используется и должно быть равно 0. Может принимать значения:
a. POOL_NX_ALLOCATION – выделяет неисполняемую память.
b. POOL_RAISE_IF_ALLOCATION_FAILURE – если при выделении произойдёт ошибка, выбросить исключение.
Size – определяет размер в байтах каждой записи в списке
Tag – определяет тэг пула, используемого при выделении памяти для элементов списка.
Depth – зарезервировано. Должно равняться 0.
3. ExFreeToPagedLookasideList(PPAGED_LOOKASIDE_LIST Lookaside, void* Entry);
Entry – указатель на запись, которая будет освобождена.
4. ExDeletePagedLookasideList(PPAGED_LOOKASIDE_LIST Lookaside);
5. ExAllocateFromNPagedLookasideList(PNPAGED_LOOKASIDE_LIST Lookaside);
6. ExInitializeNPagedLookasideList(PNPAGED_LOOKASIDE_LIST Lookaside, PALLOCATE_FUNCTION Allocate, PFREE_FUNCTION Free, ULONG Flags, SIZE_T Size, ULONG Tag, USHORT Depth);
Параметры аналогичные с ExInitializePagedLookasideList, кроме первого. Здесь Lookaside – указатель на структуру NPAGED_LOOKASIDE_LIST.
7. ExFreeToNPagedLookasideList(PNPAGED_LOOKASIDE_LIST Lookaside, void* Entry);
8. ExDeleteNPagedLookasideList(PNPAGED_LOOKASIDE_LIST Lookaside).
В блоке состояния процессора KPRCB хранятся 16 специальных списков предыстории. **Специальные списки предыстории применяются для:**
1. Информации о создании объектов.
2. Пакетов ввода-вывода (I/O Request Packet – IRP), применяемых для управления драйверами.
3. Таблиц описания памяти (Memory Descriptor List – MDL), применяемых для обмена данными на аппаратном уровне. Для каждого сеанса в MM_SESSION_SPACE хранятся 25 сеансовых списков предыстории.

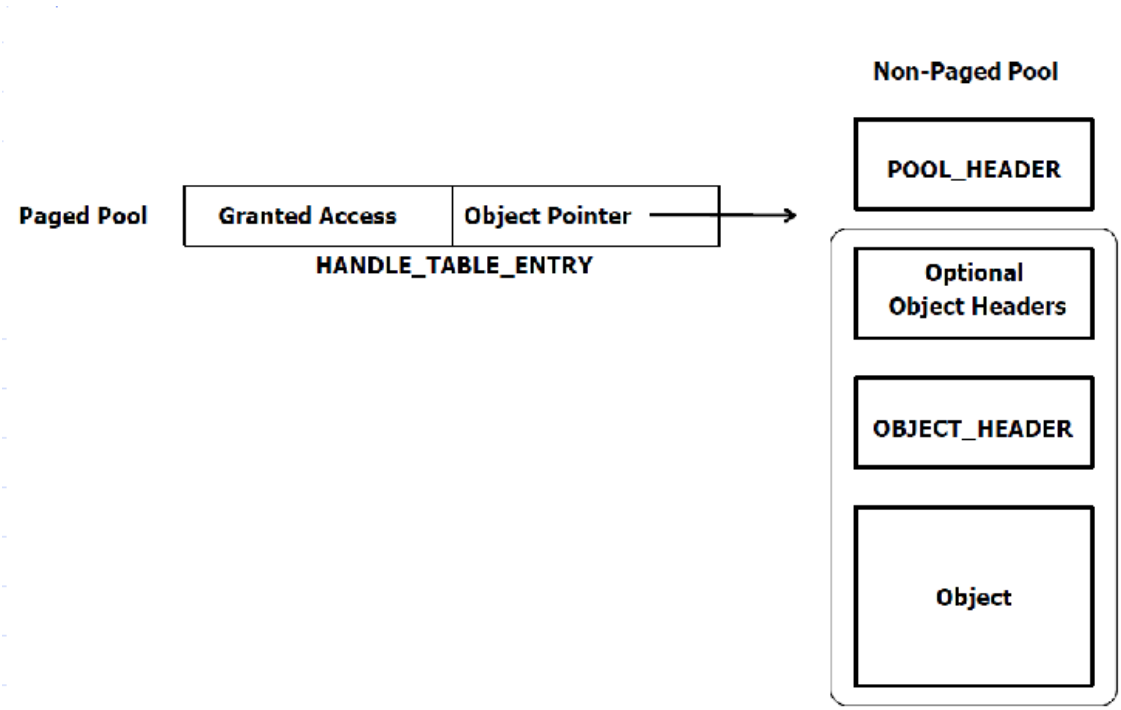
46. Представление объекта ядра в памяти. Менеджер объектов.

Каждый объект ядра — представляет собой блок памяти, выделенный ядром и доступный только ему. Этот блок представляет собой структуру данных, в элементах которой содержится

информация об объекте. Некоторые элементы (дескриптор защиты, счетчик числа пользователей и др.) присутствуют во всех объектах, но большая их часть специфична для объектов конкретного типа. Например, у объекта "процесс" есть идентификатор, базовый приоритет и код завершения, а у объекта "файл" — смещение в байтах, режим разделения и режим открытия. Объекты ядра принадлежат ядру, а не процессу. Иначе говоря, если Ваш процесс вызывает функцию, создающую объект ядра, а затем завершается, объект ядра может быть не разрушен. В большинстве случаев такой объект все же разрушается; но если созданный Вами объект ядра используется другим процессом, ядро запретит разрушение объекта до тех пор, пока от него не откажется и тот процесс.

Представление объектов ядра в памяти:

- Таблица указателей на объекты ядра размещается в страничном пуле. Каждый элемент таблицы описывается структурой HANDLE_TABLE_ENTRY, в которой содержится режим доступа к объекту и указатель на объект.
- Объект хранится в резидентном пуле.
- В заголовке объекта хранится указатель на дескриптор защиты объекта, размещающийся в страничном пуле.
- Заголовок блока резидентного пула содержит тег, в котором закодирован тип объекта. Например, у файловых объектов тег называется "File". Это удобно при отладке.



Диспетчер объектов (object manager) управляет большинством интересных объектов режима ядра, используемых на исполнительном уровне. Это процессы, потоки, файлы, семафоры, устройства и драйверы ввода-вывода, таймеры и многое другое. В Windows структуры данных ядра имеют много общего, так что очень полезно управлять ими неким унифицированным способом. Является частью ядра (ntoskrnl.exe).

Диспетчер объектов предоставляет следующие средства: управление размещением и освобождением памяти для объектов, учет квот, поддержка доступа к объектам при помощи описателей, подсчет ссылок на указатели в режиме ядра и ссылок на описатели, именование объектов в пространстве имен NT, предоставление расширяемого механизма для управления жизненным циклом любого объекта. Те структуры данных ядра, которым нужны эти средства, управляются диспетчером объектов.

Диспетчер предоставляет **общий, унифицированный механизм использования** системных ресурсов.

Обеспечиваемое диспетчером объектов единообразие имеет различные аспекты. Все эти объекты используют один и тот же механизм для создания, уничтожения и учета в системе квот. Ко всем этим объектам можно обращаться из процессов пользовательского режима при помощи описателей. Существует унифицированное соглашение для управления указателями, ссылающимися на объекты из ядра. Объектам можно давать имена в пространстве имен NT (которое управляется диспетчером объектов). Объекты диспетчеризации (которые начинаются с обычной структуры данных для сигнализации событий) могут использовать обычные интерфейсы синхронизации и уведомления (вроде WaitForMultipleObjects). Существует обычная система безопасности с использованием списков управления доступом (ACL), обязательная для открываемых по имени объектов, а также проверки доступа при каждом использовании описателя. Есть даже средства (для трассировки использования объектов) для помощи разработчикам режима ядра при отладке программ.

Диспетчер объектов представлен множеством функции ObXXX в ядре NT. Некоторые функций не экспортируются и используются другими подсистемами только внутри ядра. Доступны для использования вне ядра следующие, в основном недокументированные, функции.

ObAssignSecurity, ObCheckCreateObjectAccess, ObCheckObjectAccess, ObCreateObject, ObDereferenceObject, ObFindHandleForObject, ObGetObjectPointerCount, ObGetObjectSecurity, ObInsertObject, ObMakeTemporaryObject, ObOpenObjectByName, ObOpenObjectByPointer, ObQueryNameString, ObQueryObjectAuditingByHandle, ObReferenceObjectByHandle, ObReferenceObjectByName, ObReferenceObjectByPointer, ObReleaseObjectSecurity, ObSetSecurityDescriptorInfo, ObfDereferenceObject, ObfReferenceObject

Диспетчер объектов сам опирается в своей работе на некоторые объекты. Один из них это объект-тип. Объект-тип нужен для того, что бы вынести часть общих свойств объектов. Объекты-типы создаются с помощью функций ObCreateObjectType(...) во время инициализации подсистем ядра.

Грубо говоря, любой объект можно разделить на две части: одна изкоторых содержит информацию необходимую самому диспетчеру (назовем эту часть заголовком). Вторая заполняется и определяется нуждами подсистемы, создавшей объект. (Это - тело объекта). Не смотря на то, что диспетчер объектов оперирует заголовками и не интересуется, собственно, содержимым объекта, имеется несколько объектов, которые используются самим диспетчером объектов. Объект-тип, как и следует из названия, определяет тип объекта. У каждого объекта есть ссылка на его объект-тип, к телу которого менеджер объектов обращается очень часто. Одно из полей объекта-типа - это имя типа. В NT существуют следующие типы

Type, Directory, SymbolicLink, Event, EventPair, Mutant, Semaphore, Windows, Desktop, Timer, File, IoCompletion, Adapter, Controller, Device, Driver, Key, Port, Section, Process, Thread, Token, Profile

47. Фиксация данных в физической памяти ОС Windows. Таблица описания памяти (MDL) и ее использование.

Драйверам необходимо фиксировать данные в физической памяти, чтобы работать с ними на высоких уровнях прерываний. Способы получения физической памяти:

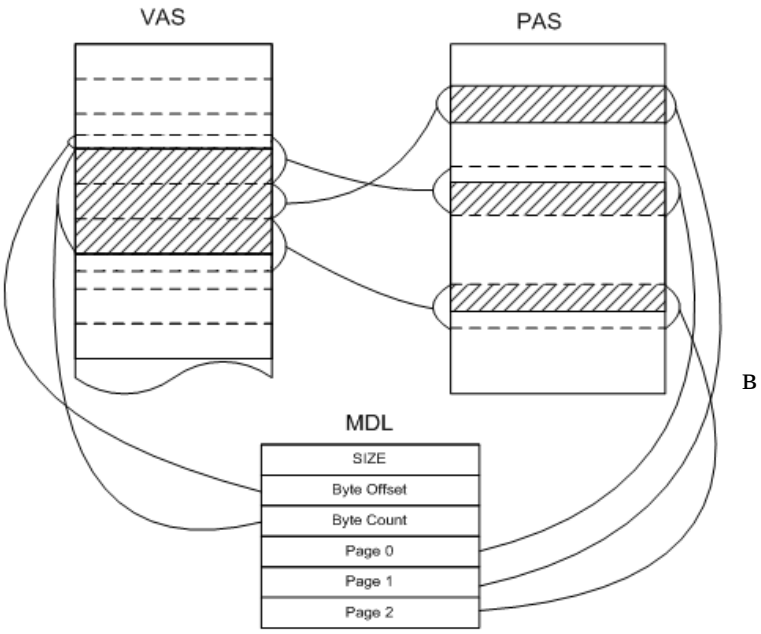
1. Выделить физически непрерывный блок в резидентном пуле. Дорого!
2.
3. Выделить физически прерывающийся блок в резидентном пуле.
4.
5. Выделить блок в страничном пуле и зафиксировать его страницы в физической памяти.

6.
7. Создать виртуальной памяти таблицу описания памяти, которая отображает непрерывный блок памяти в прерывающийся блок физической памяти. Таблица описания памяти называется Memory Descriptor List (MDL).

Функции для работы с физической памятью:

1. MmAllocateContiguousMemory() – выделяет непрерывный невыгружаемый (non-paged) участок памяти и отображает его в системном адресном пространстве
2. MmGetPhysicalAddress() – возвращает физический адрес, соответствующий виртуальному адресу;
3. MmLockPageableCodeSection() – блокирует часть кода драйвера, содержащий набор подпрограмм драйвера, отмеченных специальной директивой компилятора, в системном пространстве;
4. MmLockPageableDataSection() – блокирует весь раздел данных драйверов в системном пространстве;
5. MmLockPageableSectionByHandle() – блокирует выгружаемый код или раздел данных в системной памяти, увеличивая счетчик ссылок на дескрипторе секции;
6. MmUnlockPageableImageSection() – разблокирует часть кода драйвера или данных драйвера, которые ранее были заблокированы функциями выше;
7. MmPageEntireDriver() – делает весь код и данные драйвера выгружаемыми;
8. MmResetDriverPaging() – восстанавливает статус всех секций кода драйвера, данный им при компиляции, и все секции, которые были созданы как нестраничные, фиксируются в оперативной памяти;
9. MmMapIoSpace() – отображает физический диапазон адресов в невыгружаемую память.

Таблица описания памяти (Memory Descriptor List – MDL):



MDL - структура данных, описывающая отображение буфера виртуальной памяти (Virtual Address Space) в физическую память (Physical Address Space).

Диспетчер памяти использует структуру MDL для описания набора страниц физической памяти, составляющих буфер виртуальной памяти в контексте памяти некоторого процесса. Интерпретация MDL не зависит от контекста памяти, поскольку MDL оперирует со страницами физической памяти. Получив для данного буфера описание в виде MDL, драйвер в дальнейшем может использовать буфер в контексте памяти любого процесса. Для того, чтобы обращаться к такой памяти, необходимо получить для MDL адрес памяти в системном адресном пространстве. Сделать это можно с помощью функции MmGetSystemAddressForMdl().

Кроме того, буфер, описанный с помощью MDL, может быть использован для операций DMA. Для этого физический адрес внутри MDL должен быть транслирован в логический адрес (имеющий смысл только для данного устройства DMA) с помощью функции loMapTransfer(). MDL

предназначен для описания буфера данных, непрерывного в виртуальной памяти. Однако страницы физической памяти, список которых собственно и содержит MDL, могут располагаться в памяти произвольным образом. Это дает возможность «собирать» непрерывный в виртуальной памяти буфер из различных фрагментов физической памяти без копирования памяти.

MDLs используются:

- чтобы отобразить нестраничную память
- драйверами, для прямого доступ к памяти DMA (Direct Memory Access)

Заголовок MDL, за которым следует массив номеров страниц:

```
struct MDL {
      PMDL Next; // Цепочка буферов. Можно обратиться напрямую.
      SHORT Size; // Размер структуры, включает массив номеров страниц.
      SHORT MdlFlags; // Флаги. Можно обратиться напрямую.
      PEPROCESS Process; // Процесс, из квоты которого выделили память.
      PVOID MappedSystemVa; // Виртуальный адрес системной памяти.
      PVOID StartVa; // Виртуальный адрес буфера (page aligned).
      ULONG ByteCount; // Размер буфера.
      ULONG ByteOffset; // Смещение до буфера относительно StartVa.
};
```

Функции для работы со структурой MDL:

1. MDL* IoAllocateMdl(PVOID VirtualAddress, ULONG Length,
2. bool SecondaryBuffer, bool ChargeQuota, PIRP Irp) – выделяет память под список дескрипторов памяти;
3. IoFreeMdl() – освобождает память
4. IoBuildPartialMdl() – строит новый список дескрипторов памяти (MDL), представляющий часть буфера, которая описана с помощью существующей MDL;
5. MmInitializeMdl() – макрос, инициализирующий заголовок MDL;
6. MmSizeOfMdl() – размер структуры;
7. MmBuildMdlForNonPagedPool();
8. MmGetMdlVirtualAddress() – возвращает базовый адрес буфера, описываемого MDL структурой;
9. MmGetMdlByteCount();
10. MmGetMdlByteOffset();
11. MmGetSystemAddressForMdlSafe();
12. MmProbeAndLockPages();
13. MmUnlockPages();
14. MmMapLockedPages();
15. MmMapLockedPagesSpecifyCache();
16. MmUnmapLockedPages().

48. Понятие драйвера ОС Windows. Виды драйверов. Типы драйверов в режиме ядра. Точки входа в драйвер.

Драйвер – программа, обеспечивающая физическое взаимодействие ОС с физическим устройством. Драйвер обрабатывает прерывания устройства, поддерживает очередь запросов и преобразует запросы в команды управления устройством.
В отличие от прикладной программы, драйвер не является процессом и не имеет потока исполнения. Вместо этого любая функция драйвера выполняется в контексте того потока и процесса, в котором она была вызвана. При этом вызов может происходить от прикладной программы или драйвера, либо возникать в результате прерывания. В первом случае контекст исполнения драйвера точно известен - это прикладная программа. В третьем случае контекст исполнения случайный, поскольку прерывание (и, соответственно, исполнение кода драйвера) может произойти при выполнении любой прикладной программы. Во втором случае контекст исполнения может быть как известным, так и случайным - это зависит от контекста исполнения функции вызывающего драйвера.
Под вызовом драйвера здесь подразумевается не обычный вызов функции, а передача так называемого запроса ввода/вывода.
Различают несколько видов драйверов:

- Драйвер, получающий запросы ввода/вывода из прикладной программы, называют драйвером высшего уровня. Если такой драйвер не пользуется услугами других драйверов, он называется монолитным.
- Драйвер, получающий запросы ввода/вывода от другого драйвера, называют промежуточным, если он пользуется услугами других драйверов, или драйвером низшего уровня, если он не пользуется услугами других драйверов.

В NT существует ***два типа драйверов***: драйверы пользовательского режима и драйверы режима ядра. В дальнейшем, говоря «драйвер», мы будем подразумевать драйвер режима ядра. Такие драйверы являются частью исполнительной системы, а более точно - элементами диспетчера ввода/вывода (архитектура NT и ее компоненты будут обсуждаться ниже). Как следует из названия, при работе драйвера режима ядра процессор находится в режиме ядра.
Драйвер NT располагается в файле с расширением .sys и имеет стандартный PE-формат (PE - Portable Executable).
Драйверы реализованы как самостоятельные модули с четко определенным интерфейсом взаимодействия с ОС. Все драйверы имеют определенный системой набор стандартных функций драйвера (standard driver routines) и некоторое число внутренних функций, определенных разработчиком.
*Драйверы режима ядра можно разбить на **три** типа:*

- драйверы высшего уровня (highest level drivers);
- драйверы промежуточного уровня (intermediate drivers);
- драйверы низшего уровня (lowest level drivers).

Как будет показано ниже, такое разбиение обусловлено многоуровневой моделью драйверов (layered driver model). Для сохранения общности изложения, монолитный драйвер можно включить в эту схему, хотя он не использует многоуровневую архитектуру. В этом случае он будет «гибридом» - драйвером, принадлежащим одновременно к нескольким типам. Например, монолитный драйвер, имеющий интерфейс с приложением и осуществляющий доступ к оборудованию, будет одновременно драйвером высшего и низшего уровня.
Кроме того, в зависимости от назначения драйвера, он может являться каким-либо специализированным драйвером, то есть удовлетворять дополнительному набору требований к своей структуре. Можно привести следующие типы специализированных драйверов:

- драйверы файловой системы;
- сетевые драйверы.

Отдельно необходимо упомянуть архитектуру WDM - Windows Driver Model. Эта архитектура позволяет создавать драйверы для Windows 98 и Windows 2000, совместимые на уровне двоичного кода.
Характеристики драйверов - это совокупность следующих вопросов:
1. Поддержка динамической загрузки и выгрузки (однако могут быть исключения).
2. Необходимость следовать определенным протоколам взаимодействия с системой, нарушение которых чаще всего ведет к «синему экрану» (Blue Screen Of Death, BSOD).
3. Возможность «наслоения» драйверов поверх друг друга. В Win2000 эта возможность возведена в абсолют, хотя монолитные драйвера все еще поддерживаются.
4. Поскольку драйвера являются частью ядра ОС, они могут сделать с системой все, что угодно, поэтому основная проблема — это закрытость архитектуры ОС.

Со временем появились 2 типа драйверов:
Драйверы устройств (**низкоуровневые или аппаратные драйверы**) – традиционные драйверы и **высокоуровневые** драйверы, которые располагаются в общей модели подсистемы ввода-вывода над традиционными драйверами. При таком подходе повышается гибкость и расширяемость функций по управлению устройством — вместо жесткого набора функций, сосредоточенных в единственном драйвере, администратор ОС может выбрать требуемый набор функций, установив нужный высокоуровневый драйвер.

На практике чаще всего используют от двух до пяти уровней драйверов — слишком большое количество уровней может снизить скорость операций ввода-вывода. Несколько драйверов, управляющих одним устройством, но на разных уровнях, можно рассматривать как набор отдельных драйверов или как один многоуровневый драйвер.

В подсистеме управления графическими устройствами, такими как графические мониторы и принтеры на нижнем уровне работают аппаратные драйверы, которые позволяют управлять конкретным графическим адаптером или принтером определенного типа, Самый верхний уровень подсистемы составляет менеджер окон, который создает для каждого приложения виртуальный образ экрана в виде набора окон, в которые приложение может выводить свои графические данные. Менеджер управляет окнами, отображая их в определенную область физического экрана или делая их невидимыми, а также предоставляет к ним совместный доступ с контролем прав доступа

Аппаратные драйверы взаимодействуют с системой прерываний. Драйверы более высоких уровней вызываются уже не по прерываниям, а по инициативе аппаратных драйверов или драйверов вышележащего уровня.

Архитектура драйвера Windows NT использует ***модель точек входа***, в которой Диспетчер Ввода/вывода вызывает специфическую подпрограмму в драйвере, когда требуется, чтобы драйвер выполнил специфическое действие. В каждую точку входа передается определенный набор параметров для драйвера, чтобы дать возможность ему выполнить требуемую функцию.
Базовая структура драйвера состоит из набора точек входа, наличие которых обязательно, плюс некоторое количество точек входа, наличие которых зависит от назначения драйвера.

Далее перечисляются точки входа либо классы точек входа драйвера:
1. DriverEntry. Диспетчер Ввода/вывода вызывает эту функцию драйвера при первоначальной загрузке драйвера. Внутри этой функции драйверы выполняют инициализацию как для себя, так и для любых устройств, которыми они управляют. Эта точка входа требуется для всех NT драйверов.
2. Диспетчерские (Dispatch) точки входа. Точки входа Dispatch драйвера вызываются Диспетчером Ввода/вывода, чтобы запросить драйвер инициировать некоторую операцию ввода/вывода.
3. Unload. Диспетчер Ввода/вывода вызывает эту точку входа, чтобы запросить драйвер подготовиться к немедленному удалению из системы. Только драйверы, которые поддерживают выгрузку, должны реализовывать эту точку входа. В случае вызова этой функции, драйвер будет выгружен из системы при выходе из этой функции вне зависимости от результата ее работы.
4. Fast I/O. Вместо одной точки входа, на самом деле это набор точек входа. Диспетчер Ввода/вывода или Диспетчер Кэша вызывают некоторую функцию быстрого ввода/вывода (Fast I/O), для инициирования некоторого действия "Fast I/O". Эти подпрограммы поддерживаются почти исключительно драйверами файловой системы.
5. Управление очередями запросов IRP (сериализация - процесс выполнения различных транзакций в нужной последовательности). Два типа очередей: Системная очередь (Startlo) и очереди, управляемые драйвером. Диспетчер Ввода/вывода использует точку входа Startlo только в драйверах, которые используют механизм Системной Очереди (System Queuing). Для таких драйверов, Диспетчер Ввода/вывода вызывает эту точку входа, чтобы начать новый запрос ввода/вывода.
6. Reinitialize. Диспетчер Ввода/вывода вызывает эту точку входа, если она была зарегистрирована, чтобы позволить драйверу выполнить вторичную инициализацию.
7. Точка входа процедуры обработки прерывания (ISR- Interrupt Service Routine).
Эта точка входа присутствует, только если драйвер поддерживает обработку прерывания. Как только драйвер подключен к прерываниям от своего устройства, его ISR будет вызываться всякий раз, когда одно из его устройств запрашивает аппаратное прерывание.
8. Точки входа вызовов отложенных процедур (DPC - Deferred Procedure Call). Два типа DPC: DpcForlsr и CustomDpc. Драйвер использует эти точки входа, чтобы завершить работу, которая должна быть сделана в результате появления прерывания или другого специального условия. Процедура отложенного вызова

выполняет большую часть работы по обслуживанию прерывания от устройства, которое не требует высокого уровня прерывания IRQL, ассоциированного с процессором.
9. SynchCritSection. Диспетчер Ввода/вывода вызывает эту точку входа в ответ на запрос драйвера на захват одной из спин-блокировок его ISR.
10. AdapterControl. Диспетчер Ввода/вывода вызывает эту точку входа, чтобы указать, что общедоступные DMA-ресурсы драйвера доступны для использования в передаче данных. Только некоторые драйверы устройств DMA реализуют эту точку входа.
11. Cancel. Драйвер может определять точку входа Cancel для каждого IRP, который он содержит во внутренней очереди. Если Диспетчер Ввода/вывода хочет отменить конкретный IRP, он вызывает подпрограмму Cancel, связанную с этим IRP.
12. loCompletion. Эту точку входа для каждого IRP может устанавливать драйвер верхнего уровня при многоуровневой организации. Диспетчер Ввода/вывода вызывает эту подпрограмму после того, как все драйверы нижнего уровня завершили IRP.
13. loTimer. Для драйверов, которые инициализировали и запустили поддержку loTimer, Диспетчер Ввода/вывода вызывает эту точку входа приблизительно каждую секунду.
14. CustomTimerDpc. Эта точка входа вызывается, когда истекает время для запрошенного драйвером таймера.
Говоря о точках входа в драйвер, необходимо отметить контекст, при котором эти точки входа могут быть вызваны. Контекст исполнения определяется двумя составляющими:

- исполняемый в настоящее время поток (контекст планирования потока - thread scheduling context);
- контекст памяти процесса, которому принадлежит поток.

Текущий контекст исполнения может принадлежать одному из трех классов:

- контекст процесса «System» (далее - системный контекст);
- контекст конкретного потока и процесса;
- контекст случайного потока и процесса (далее - случайный контекст).

Различные точки входа в драйвер могут вызываться в контексте, принадлежащем одному из этих классов.

DriverEntry всегда вызывается в системном контексте.

Диспетчерские функции для драйверов верхнего уровня (то есть получающих запрос от прикладных программ) вызываются в контексте инициирующего запрос потока.

Диспетчерские функции драйверов остальных уровней, получающие запрос от драйвера верхнего уровня, вызываются в случайном контексте.

Все точки входа, связанные с сериализацией запросов ввода/вывода или с обработкой прерываний и DPC, вызываются в случайном контексте..

49. Объект, описывающий драйвер. Объект, описывающий устройство. Объект, описывающий файл. Структура и взаимосвязь объектов.

Файловый объект - это объект, видимый из режима пользователя, который представляет всевозможные открытые источники или приемники ввода/вывода: файл на диске или устройство (физическое, логическое, виртуальное). Физическим устройством может быть, например, последовательный порт, физический диск; логическим -логический диск; виртуальным - виртуальный сетевой адаптер, именованный канал, почтовый ящик. Всякий раз, когда некоторый поток открывает файл, создается новый файловый объект с новым набором атрибутов. В любой момент времени сразу несколько файловых объектов могут быть ассоциированы с одним разделяемым виртуальным файлом, но каждый такой файловый объект имеет уникальный описатель, корректный только в контексте процесса, поток которого инициировал открытие файла. Файловые объекты, как и другие объекты, имеют иерархические имена, охраняются объектной защитой, поддерживают синхронизацию и обрабатываются системными сервисами.

Объект драйвера в системе соответствует отдельному драйверу. Именно он дает диспетчеру ввода-вывода адрес процедур диспетчеризации (точек входа) всех
драйверов. Объект-драйвер описывает также, где драйвер загружен в физическую память и размер драйвера. Объект-драйвер описывается частично документированной структурой данных DRIVER_OBJECT. Этот объект создается менеджером ввода/вывода при загрузке драйвера в систему, после чего диспетчер вызывает процедуру инициализации драйвера DriverEntry и передает ей указатель на объект-драйвер. Структура DriverObject содержит множество полей, которые определяют поведение будущего драйвера. Наиболее ключевые из них — это указатели на так называемые вызываемые (или callback) функции, то есть функции, которые будут вызываться при наступлении определенного события.Объект-драйвер является скрытым для кода пользовательского уровня, то есть только определенные компоненты уровня ядра (в том числе и диспетчер ввода/вывода) знают внутреннюю структуру этого типа объекта и могут получать доступ ко всем данным, содержащимся в объекте, напрямую.

Диспетчер ввода/вывода определяет тип объекта - **объект-устройство**, используемый для представления физического, логического или виртуального устройства, чей драйвер был загружен в систему. Формат объекта-устройство определяется частично документированной структурой данных DEVICE_OBJECT. Хотя объект-устройство может быть создан в любое время посредством вызова функции loCreateDeviceQ, обычно он создается внутри DriverEntry. Объект-устройство описывает характеристики устройства, такие как требование по выравниванию буферов и местоположение очереди устройства, для хранения поступающих пакетов запросов ввода/вывода.

Как показано на рис. 8.6, объект устройства ссылается на свой объект драйвера, благодаря чему диспетчер ввода-вывода узнает, процедуру какого драйвера следует вызвать при получении очередного запроса на ввод или вывод. Объект устройства позволяет найти объект драйвера, соответствующий обслуживающему устройство драйверу. После этого происходит обращение к объекту драйвера через указанный в исходном запросе номер функции. Каждый номер функции соответствует точке входа драйвера.

Рис. 8.6. Объект драйвера

С объектом драйвера часто связывается несколько объектов устройств. Список последних представляет собой перечень физических и логических устройств, управляемых данным драйвером. Например, для каждого раздела жесткого диска существует отдельный объект устройства с информацией, касающейся данного раздела. Но при этом для доступа ко всем разделам используется один драйвер жесткого диска. При выгрузке драйвера из системы диспетчер ввода-вывода прибегает к очереди объектов устройств, чтобы определить, на какие устройства повлияет удаление данного драйвера.

50. Понятие пакета ввода-вывода (IRP). Структура пакета ввода-вывода. Схема обработки пакета ввода-вывода при открытии файла.

Пакет ввода-вывода – Input-Output Request Packet (IRP)

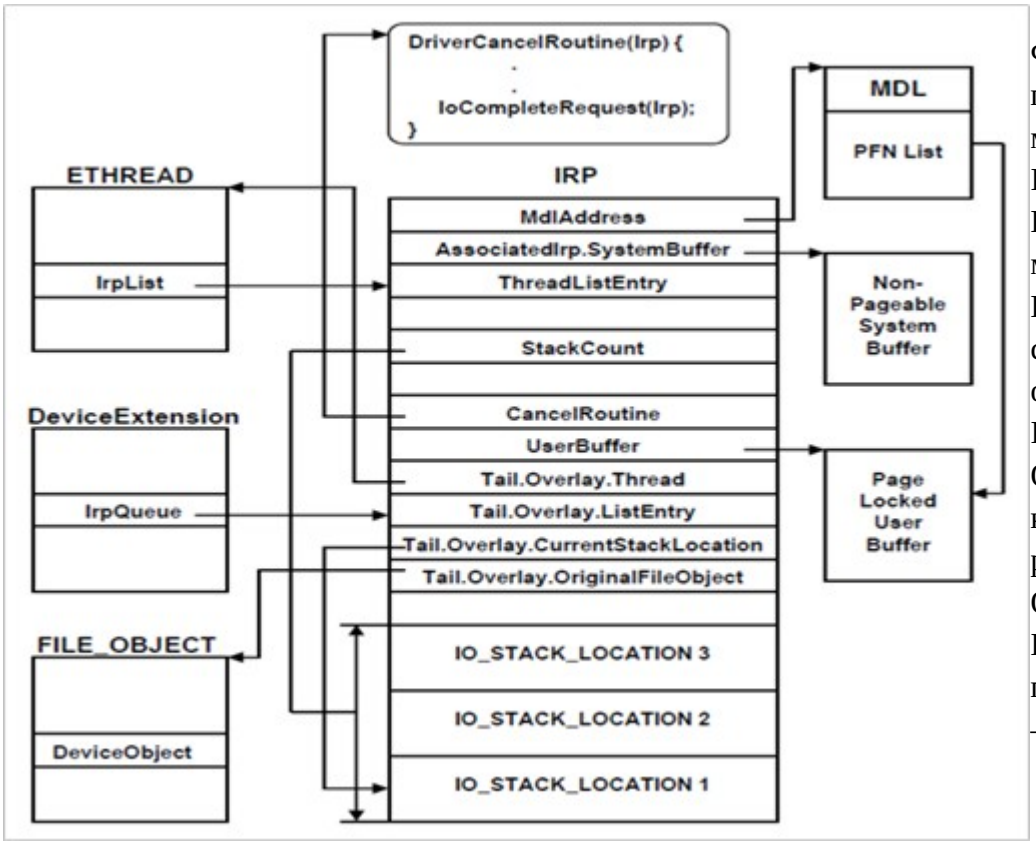
IRP пакет (I/O request packet) — структура данных ядра Windows, обеспечивающая обмен данными между приложениями и драйвером, а также между драйвером и драйвером. Представляет запрос ввода-вывода.

Создается с помощью **IoAllocateIrp**() или **IoMakeAssociatedIrp**() или **IoBuildXxxRequest**(). Память выделяется из списка предыстории в резидентном пуле.

Удаляется вызовом **IoCompleteRequest**().

Диспетчируется драйверу на обработку с помощью **IoCallDriver**().

Структура IRP:



Фиксированную (IRP) и переменную часть в виде массива записей IO_STACK_LOCATION.

Количество элементов массива – поле StackCount.

На каждый драйвер в стеке драйверов создается отдельная запись IO_STACK_LOCATION.

Объект FILE_OBJECT, с которым осуществляется работа, – Tail.Overlay.OriginalFileObject.

Буфер данных в пользовательской памяти – UserBuffer.

Буфер данных в системной памяти – AssociatedIrp.SystemBuffer. Соответствующая буферу таблица описания памяти – MdlAddress. Указатель на поток (ETHREAD), в очереди которого находится IRP, – Tail.Overlay.Thread. Список IRP потока хранится в ETHREAD.IrpList.

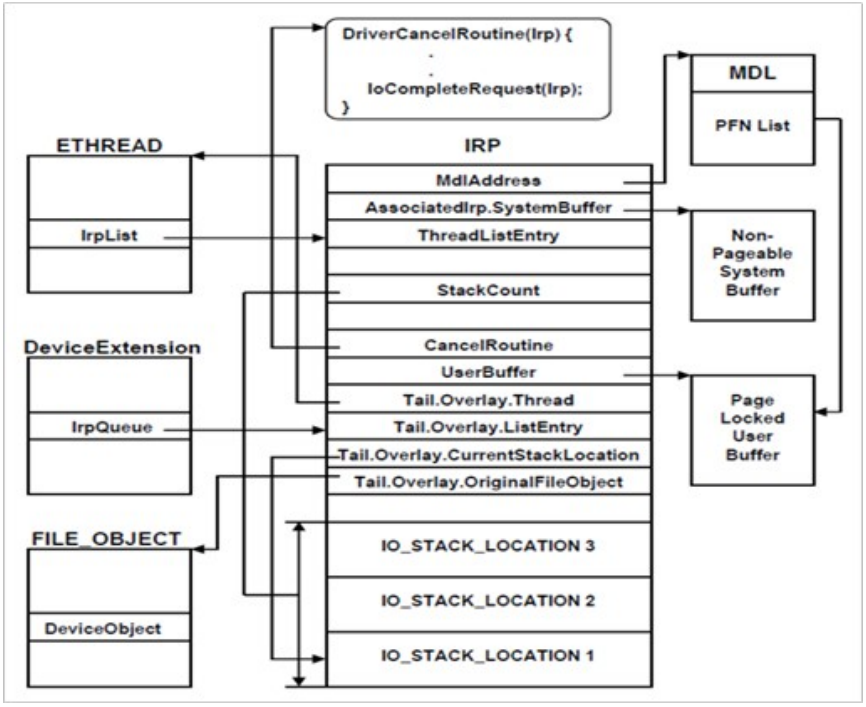


Схема обработки пакета ввода-вывода при открытии файла

1.Подсистема ОС вызывает функцию открытия файла в ядре. Эта функция реализована в менеджере ввода-вывода.

2.Менеджер ввода-вывода обращается к менеджеру объектов, чтобы по имени файла создать FILE_OBJECT. При этом осуществляется проверка прав пользователя на обращение к файлу.

3.При открытии файл может находиться на еще не смонтированном томе. В таком случае открытие файла приостанавливается, выполняется монтирование тома на внешнем устройстве и обработка продолжается.

4.Менеджер ввода-вывода создает и инициализирует IRP-пакет с помощью IoAllocateIrp(). В IRP-пакете инициализируется IO_STACK_LOCATION верхнего драйвера в стеке драйверов.

5.Менеджер ввода-вывода вызывает процедуру XxxDispatchCreate() верхнего драйвера. Процедура драйвера вызывает IoGetCurrentIrpStackLocation(), чтобы получить доступ к параметрам запроса. Она проверяет, не кэширован ли файл. Если нет, то вызывает IoCopyCurrentIrpStackLocationToNext() для создания IO_STACK_LOCATION следующего драйвера в стеке, затем IoSetCompletionRoutine() для получения уведомления о завершении обработки IRP-пакета и вызывает IoCallDriver(), делегируя обработку процедуре YyyDispatchCreate() следующего драйвера в стеке.

6.Каждый драйвер в стеке выполняет свою часть обработки IRP-пакета.

7.Последний драйвер в стеке в своей процедуре YyyDispatchCreate() устанавливает в IRP поле IoStatus и вызывает у менеджера ввода-вывода процедуру IoCompleteRequest(), чтобы завершить обработку IRP-пакета. Она проходит в IRP по массиву записей IO_STACK_LOCATION и в каждой вызывает процедуру CompletionRoutine (указывает на XxxIoCompletion() драйвера).

8.Менеджер ввода-вывода проверяет в IRP.IoStatus и копирует соответствующий код возврата в адресное пространство подсистемы ОС (пользовательского процесса).

9.Менеджер ввода-вывода удаляет IRP-пакет с помощью IoFreeIrp().

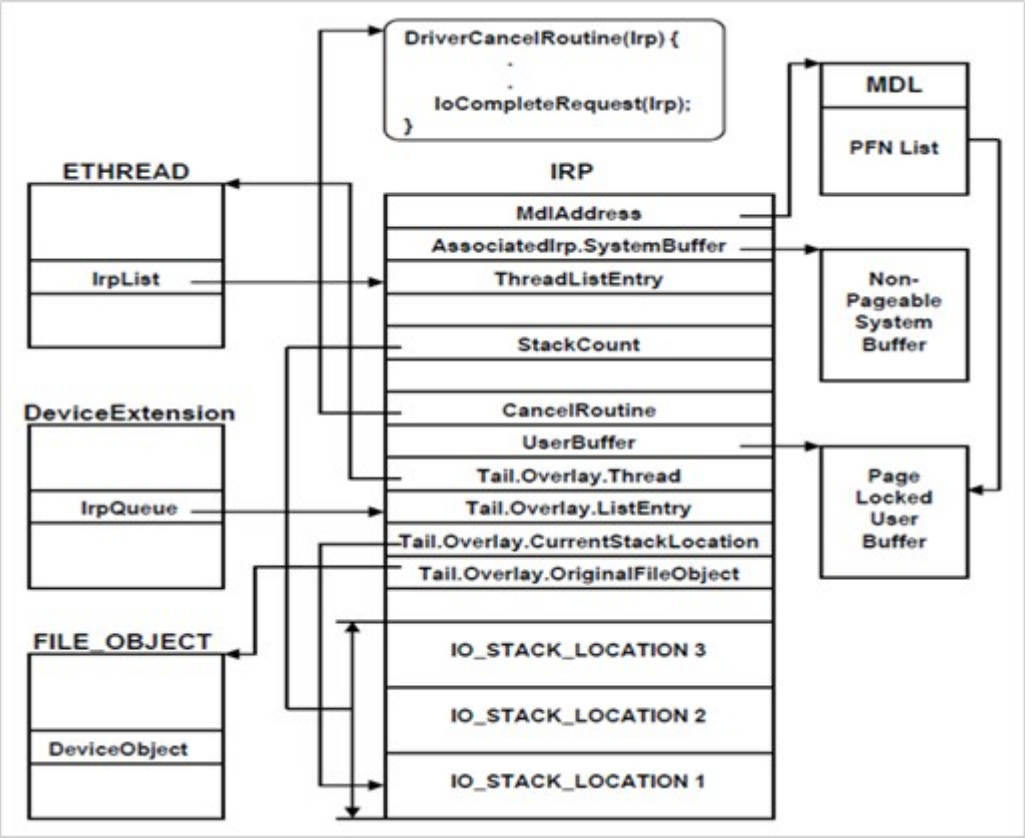
10.В адресном пространстве пользователя создается описатель для FILE_OBJECT и возвращается подсистеме ОС как результат открытия файла. В случае ошибки возвращается ее код.

51. Понятие пакета ввода-вывода (IRP). Структура пакета ввода-вывода. Схема обработки пакета ввода-вывода при выполнении чтения-записи файла.

Понятие пакета IRP и его структура

Берём из предыдущего вопроса…

Схема обработки IRP при выполнении чтения-записи файла



1.Менеджер ввода-вывода обращается к драйверу файловой системы с IRP-пакетом, созданным для выполнения чтения-записи файла. Драйвер обращается к своей записи IO_STACK_LOCATION и определяет, какую именно операцию он должен выполнить.

2.Драйвер файловой системы для выполнения операции с файлом может создавать свои IRP с помощью IoAllocateIrp(). Или же он может в уже имеющемся IRP сформировать IO_STACK_LOCATION для драйвера более низкого уровня с помощью IoGetNextIrpStackLocation().

3.Если драйвер создает собственные IRP-пакеты, он должен зарегистрировать в них свою процедуру ZzzIoCompletion(), которая выполнит удаление IRP-пакетов после обработки драйверами нижнего уровня. За удаление своих IRP каждый драйвер отвечает сам. Менеджер ввода-вывода отвечает за удаление своего IRP, созданного для выполнения ввода-вывода.

Драйвер файловой системы устанавливает в IO_STACK_LOCATION указатель CompletionRoutine на свою процедуру XxxIoCompletion(), формирует IO_STACK_LOCATION для драйвера более низкого уровня с помощью IoGetNextIrpStackLocation(), вписывая нужные значения параметров, и обращается к драйверу более низкого уровня с помощью IoCallDriver().

4.Управление передается драйверу устройства процедуре YyyDispatchRead/Write(), зарегистрированной в объекте DRIVER_OBJECT под номером IRP_MJ_XXX. Драйвер устройства не может выполнить операцию ввода-вывода в синхронном режиме. Он помечает IRP-пакет с помощью IoMarkIrpPending() как требующий ожидания обработки или передачи другой процедуре YyyDispatch().

5.Менеджер ввода-вывода получает информацию, что драйвер устройства занят, и ставит IRP в очередь к объекту DEVICE_OBJECT драйвера.

6.Когда устройство освобождается, в драйвере устройства вызывается процедура обработки прерываний (ISR). Она обнаруживает IRP в очереди к устройству и с помощью IoRequestDpc() создает DPC-процедуру для обработки IRP на более низком уровне приоритета прерываний.

7.DPC-процедура с помощью IoStartNextPacket() извлекает из очереди IRP и выполняет ввод-вывод. Наконец, она устанавливает статус-код в IRP и вызывает IoCompleteRequest().

8.В каждом драйвере в стеке вызывается процедура завершения XxxIoCompletion(). В драйвере файловой системы она проверяет статус-код и либо повторяет запрос (в случае сбоя), либо завершает его удалением всех собственных IRP (если они были). В конце, IRP-пакет оказывается в распоряжении менеджера ввода-вывода, который возвращает вызывающему потоку результат в виде NTSTATUS.

52. Перехват API-вызовов ОС Windows в пользовательском режиме. Внедрение DLL с помощью реестра. Внедрение DLL с помощью ловушек. Внедрение DLL с помощью дистанционного потока.

Внедрение DLL с использованием реестра

Зарегистрировать DLL в реестре (имя не должно содержать пробелы):

```
HKLM\Software\Microsoft\Windows_NT\CurrentVersion\Windows\AppInit_DLLs
```

Значением параметра AppInit_DLLs может быть как имя одной DLL (с указанием пути доступа), так и имена нескольких DLL, разделенных пробелами или запятыми. Поскольку пробел используется здесь в качестве разделителя, в именах файлов не должно быть пробелов. Система считывает путь только первой DLL в списке — пути остальных DLL игнорируются, поэтому лучше размещать свои DLL в системном каталоге Windows, чтобы не указывать пути.

При следующей перезагрузке компьютера Windows сохранит значение этого параметра. Далее, когда User32.dll будет спроецирован на адресное пространство процесса, этот модуль получит уведомление DLL_PROCESS_ATTACH и после его обработки вызовет LoadLibrary для всех DLL, указанных в параметре AppInit_DLLs. В момент загрузки каждая DLL инициализируется вызовом ее функции DllMain с параметром fwdReason, равным DLL_PROCESS_ATTACH. Поскольку внедряемая DLL загружается на такой ранней стадии создания процесса, будьте особенно осторожны при вызове функций. Проблем с вызовом функций Kernel32.dll не должно быть, но в случае других DLL они вполне вероятны — User32.dll не проверяет, успешно ли загружены и инициализированы эти DLL. Правда, в Windows 2000 модуль User32.dll ведет себя несколько иначе, но об этом — чуть позже. Это простейший способ внедрения DLL. Все, что от Вас требуется, — добавить значение в уже существующий параметр реестра. Однако он не лишен недостатков.

1. Ваша DLL проецируется на адресные пространства только тех процессов, на которые спроецирован и модуль User32.dll. Его используют все GUI-приложения, но большинство программ консольного типа — нет. Поэтому такой метод не годится для внедрения DLL, например, в компилятор или компоновщик.
2. Ваша DLL проецируется на адресные пространства всех GUI-процессов. Но Вам-то почти наверняка надо внедрить DLL только в один или несколько определенных процессов. Чем больше процессов попадет «под тень» такой DLL, тем выше вероятность аварийной ситуации. Ведь теперь Ваш код выполняется потоками этих процессов, и, если он зациклится или некорректно обратится к памяти, Вы повлияете на поведение и устойчивость соответствующих процессов. Поэтому лучше внедрять свою DLL в как можно меньшее число процессов.
3. Ваша DLL проецируется на адресное пространство каждого GUI-процесса в течение всей его жизни. Тут есть некоторое сходство с предыдущей проблемой. Желательно не только внедрять DLL в минимальное число процессов, но и проецировать ее на эти процессы как можно меньшее время. Так что лучшее решение — внедрять DLL только на то время, в течение которого она действительно нужна конкретной программе.

Внедрение DLL с помощью ловушек

Внедрение DLL в адресное пространство процесса возможно и с применением ловушек. Чтобы они работали так же, как и в 16-разрядной Windows, Microsoft пришлось создать механизм, позволяющий внедрять DLL в адресное пространство другого процесса. Рассмотрим его на примере. Процесс A устанавливает ловушку WH_GETMESSAGE и наблюдает за сообщениями, которые обрабатываются окнами в системе. Ловушка устанавливается вызовом SetWindowsHookEx:

```
HHOOK hHook = SetWindowsHookEx(WH_GETMESSAGE, GetMsgProc, hinstDll, 0);
```

Аргумент WH_GETMESSAGE определяет тип ловушки, а параметр GetMsgProc — адрес функции (в адресном пространстве Вашего процесса), которую система должна вызывать всякий раз, когда окно собирается обработать сообщение. Параметр hinstDll идентифицирует DLL, содержащую функцию GetMsgProc. В Windows значение hinstDll для DLL фактически задает адрес в виртуальной памяти, по которому DLL спроецирована на адресное пространство процесса. И, наконец, последний аргумент, 0, указывает поток, для которого предназначена ловушка. Поток может вызвать SetWindowsHookEx и передать ей идентификатор другого потока в системе. Передавая 0, мы сообщаем системе, что ставим ловушку для всех существующих в ней GUI-потоков. Теперь посмотрим, как все это действует:

1. Поток процесса B собирается направить сообщение какому-либо окну.
2. Система проверяет, не установлена ли для данного потока ловушка WH_GETMESSAGE.
3. Затем выясняет, спроецирована ли DLL, содержащая функцию GetMsgProc, на адресное пространство процесса B.
4. Если указанная DLL еще не спроецирована, система отображает ее на адресное пространство процесса B и увеличивает счетчик блокировок (lock count) проекции DLL в процессе B на 1.
5. Система проверяет, не совпадают ли значения hinstDll этой DLL, относящиеся к процессам A и B. Если hinstDll в обоих процессах одинаковы, то и адрес GetMsgProc в этих процессах тоже одинаков. Тогда система может просто вызвать GetMsgProc в адресном пространстве процесса A. Если же hinstDll различны, система определяет адрес функции GetMsgProc в адресном пространстве процесса B по формуле: GetMsgProc B = histDll B + (GetMsgProc A – hinstDll A) Вычитая hinstDll A из GetMsgProc A, Вы получаете смещение (в байтах) адреса функции GetMsgProc. Добавляя это смещение к hinstDll B, Вы получаете адрес GetMsgProc, соответствующий проекции DLL в адресном пространстве процесса B.
6. Счетчик блокировок проекции DLL в процессе B увеличивается на 1.
7. Вызывается GetMsgProc в адресном пространстве процесса B.
8. После возврата из GetMsgProc счетчик блокировок проекции DLL в адресном пространстве процесса B уменьшается на 1.

Кстати, когда система внедряет или проецирует DLL, содержащую функцию фильтра ловушки, проецируется вся DLL, а не только эта функция. А значит, потокам, выполняемым в контексте процесса B, теперь доступны все функции такой DLL. Итак, чтобы создать подкласс окна, сформированного потоком другого процесса, можно сначала установить ловушку WH_GETMESSAGE для этого потока, а затем — когда будет вызвана функция GetMsgProc — обратиться к SetWindowLongPtr и создать подкласс. Разумеется, процедура подкласса должна быть в той же DLL, что и GetMsgProc. В отличие от внедрения DLL с помощью реестра этот способ позволяет в любой момент отключить DLL от адресного пространства процесса, для чего достаточно вызвать:

```
BOOL UnhookWindowsHookEx(HHOOK hHook);
```

Когда поток обращается к этой функции, система просматривает внутренний список процессов, в которые ей пришлось внедрить данную DLL, и уменьшает счетчик ее блокировок на 1. Как только этот счетчик обнуляется, DLL автоматически выгружается. Вспомните: система увеличивает его непосредственно перед вызовом GetMsgProc. Это позволяет избежать нарушения доступа к памяти. Если бы счетчик не увеличивался, то другой поток мог бы вызвать UnhookWindowsHookEx в тот момент, когда поток процесса B пытается выполнить код GetMsgProc. Все это означает, что нельзя создать подкласс окна и тут же убрать ловушку — она должна действовать в течение всей жизни подкласса.

Внедрение DLL с помощью удаленных потоков

Третий способ внедрения DLL — самый гибкий. В нем используются многие особенности Windows: процессы, потоки, синхронизация потоков, управление виртуальной памятью, поддержка DLL и

Unicode. Большинство Windows-функций позволяет процессу управлять лишь самим собой, исключая тем самым риск повреждения одного процесса другим. Однако есть и такие функции, которые дают возможность управлять чужим процессом. Изначально многие из них были рассчитаны на применение в отладчиках и других инструментальных средствах. Но ничто не мешает использовать их и в обычном приложении. Внедрение DLL этим способом предполагает вызов функции LoadLibrary потоком целевого процесса для загрузки нужной DLL. Так как управление потоками чужого процесса сильно затруднено, Вы должны создать в нем свой поток. К счастью, Windows-функция CreateRemoteThread делает эту задачу несложной:

```
HANDLE CreateRemoteThread( HANDLE hProcess, PSECURITY_ATTRIBUTES psa, DWORD
dwStackSize, PTHREAD_START_ROUTINE pfnStartAddr, PVOID pvParam, DWORD fdwCreate, PDWORD
pdwThreadId);
```

Она идентична CreateThread, но имеет дополнительный параметр hProcess, идентифицирующий процесс, которому будет принадлежать новый поток. Параметр pfnStartAddr определяет адрес функции потока. Этот адрес, разумеется, относится к удаленному процессу — функция потока не может находиться в адресном пространстве Вашего процесса.

Но как заставить этот поток загрузить нашу DLL? Ответ прост: нужно, чтобы он вызвал функцию LoadLibrary:

```
HINSTANCE LoadLibrary(PCTSTR pszLibFile);
```

К счастью, прототипы LoadLibrary и функции потока очень похожи друг на друга. Вот как выглядит прототип функции потока:

```
DWORD WINAPI ThreadFunc(PVOID pvParam);
```

Обе функции принимают единственный параметр и возвращают некое значение. Кроме того, обе используют одни и те же правила вызова — WINAPI. Это крайне удачное стечение обстоятельств, потому что нам как раз и нужно создать новый поток, адрес функции которого является адресом LoadLibrary. По сути, требуется выполнить примерно такую строку кода:

```
HANDLE hThread = CreateRemoteThread(hProcessRemote, NULL, 0, LoadLibrary, "C:\\
MyLib.dll", 0, NULL);
```

Новый поток в удаленном процессе немедленно вызывает LoadLibrary, передавая ей адрес полного имени DLL. Все просто. Однако Вас ждут две проблемы. Первая в том, что нельзя вот так запросто, как я показал выше, передать LoadLibrary в четвертом параметре функции CreateRemoteThread. Причина этого весьма неочевидна. При сборке программы в конечный двоичный файл помещается раздел импорта. Этот раздел состоит из серии шлюзов к импортируемым функциям. Так что, когда Ваш код вызывает функцию вроде LoadLibrary, в разделе импорта модуля генерируется вызов соответствующего шлюза. А уже от шлюза происходит переход к реальной функции. Следовательно, прямая ссылка на LoadLibrary в вызове CreateRemoteThread преобразуется в обращение к шлюзу LoadLibrary в разделе импорта Вашего модуля. Передача адреса шлюза в качестве стартового адреса удаленного потока заставит этот поток выполнить неизвестно что. И скорее всего это закончится нарушением доступа. Чтобы напрямую вызывать LoadLibrary, минуя шлюз, Вы должны выяснить ее точный адрес в памяти с помощью GetProcAddress. Вызов CreateRemoteThread предполагает, что Kernel32.dll спроецирована в локальном процессе на ту же область памяти, что и в удаленном. Kernel32.dll используется всеми приложениями, и, как показывает опыт, система проецирует эту DLL в каждом процессе по одному и тому же адресу. Так что CreateRemoteThread надо вызвать так:

```
// получаем истинный адрес LoadLibrary в Kernel32.dll
PTHREAD_START_ROUTINE pfnThreadRtn = (PTHREAD_START_ROUTINE)
GetProcAddress(GetModuleHandle(TEXT("Kernel32")), "LoadLibrary");
HANDLE hThread = CreateRemoteThread(hProcessRemote, NULL, 0, pfnThreadRtn, "C:\\
MyLib.dll", 0, NULL);
```

Отлично, одну проблему мы решили. Но я говорил, что их две. Вторая связана со строкой, в которой содержится полное имя файла DLL. Строка «C:\\MyLib.dll» находится в адресном пространстве

вызывающего процесса. Ее адрес передается только что созданному потоку, который в свою очередь передает его в LoadLibrary. Но, когда LoadLibrary будет проводить разыменование (dereferencing) этого адреса, она не найдет по нему строку с полным именем файла DLL и скорее всего вызовет нарушение доступа в потоке удаленного процесса; пользователь увидит сообщение о необрабатываемом исключении, и удаленный процесс будет закрыт. Все верно: Вы благополучно угробили чужой процесс, сохранив свой в целости и сохранности! Эта проблема решается размещением строки с полным именем файла DLL в адресном пространстве удаленного процесса. Впоследствии, вызывая CreateRemoteThread, мы передадим ее адрес (в удаленном процессе). На этот случай в Windows предусмотрена функция VirtualAllocEx, которая позволяет процессу выделять память в чужом адресном пространстве:

```
PVOID VirtualAllocEx( HANDLE hProcess, PVOID pvAddress, SIZE_T dwSize, DWORD
flAllocationType, DWORD flProtect);
```

А освободить эту память можно с помощью функции VirtualFreeEx.

```
BOOL VirtualFreeEx( HANDLE hProcess, PVOID pvAddress, SIZE_T dwSize, DWORD
dwFreeType);
```

Обе функции аналогичны своим версиям без суффикса Ex в конце. Единственная разница между ними в том, что эти две функции требуют передачи в первом параметре описателя удаленного процесса. Выделив память, мы должны каким-то образом скопировать строку из локального адресного пространства в удаленное. Для этого в Windows есть две функции:

```
BOOL ReadProcessMemory( HANDLE hProcess, PVOID pvAddressRemote, PVOID pvBufferLocal,
DWORD dwSize, PDWORD pdwNumBytesRead);
BOOL WriteProcessMemory( HANDLE hProcess, PVOID pvAddressRemote, PVOID pvBufferLocal,
DWORD dwSize, PDWORD pdwNumBytesWritten);
```

Параметр hProcess идентифицирует удаленный процесс, pvAddressRemote и pvBufferLocal определяют адреса в адресных пространствах удаленного и локального процесса, а dwSize — число передаваемых байтов. По адресу, на который указывает параметр pdwNumBytesRead или pdwNumBytesWritten, возвращается число фактически считанных или записанных байтов. Теперь, когда Вы понимаете, что я пытаюсь сделать, давайте суммируем все сказанное и запишем это в виде последовательности операций, которые Вам надо будет выполнить.

1.Выделите блок памяти в адресном пространстве удаленного процесса через VirtualAllocEx.

2.Вызвав WriteProcessMemory, скопируйте строку с полным именем файла DLL в блок памяти, выделенный в п. 1.

3. Используя GetProcAddress, получите истинный адрес функции LoadLibrary внутри Kernel32.dll.

4.Вызвав CreateRemoteThread, создайте поток в удаленном процессе, который вызовет соответствующую функцию LoadLibrary, передав ей адрес блока памяти, выделенного в п. 1. На этом этапе DLL внедрена в удаленный процесс, а ее функция DllMain получила уведомление DLL_PROCESS_ATTACH и может приступить к выполнению нужного кода. Когда DllMain вернет управление, удаленный поток выйдет из LoadLibrary и вернется в функцию BaseThreadStart, которая в свою очередь вызовет ExitThread и завершит этот поток. Теперь в удаленном процессе имеется блок памяти, выделенный в п. 1, и DLL, все еще «сидящая» в его адресном пространстве. Для очистки после завершения удаленного потока потребуется несколько дополнительных операций.

5. Вызовом VirtualFreeEx освободите блок памяти, выделенный в п. 1.

6. С помощью GetProcAddress определите истинный адрес функции FreeLibrary внутри Kernel32.dll.

7. Используя CreateRemoteThread, создайте в удаленном процессе поток, который вызовет FreeLibrary с передачей HINSTANCE внедренной DLL.

53. Перехват API-вызовов ОС Windows в пользовательском режиме. Замена адреса в таблице импорта. Перехват в точке входа в процедуру с помощью подмены начальных инструкций (Microsoft Detours).

Замена адреса в таблице импорта.

Как Вам уже известно, в разделе импорта содержится список DLL, необходимых модулю для нормальной работы. Кроме того, в нем перечислены все идентификаторы, которые модуль импортирует из каждой DLL. Вызывая импортируемую функцию, поток получает ее адрес фактически из раздела импорта. Поэтому, чтобы перехватить определенную функцию, надо лишь изменить ее адрес в разделе импорта. Все! И никакой зависимости от процессорной платформы. А поскольку Вы ничего не меняете в коде функции, то и о синхронизации потоков можно не беспокоиться. Вот функция, которая делает эту сказку былью. Она ищет в разделе импорта модуля ссылку на идентификатор по определенному адресу и, найдя ее, подменяет адрес соответствующего идентификатора.

```
void ReplaceIATEntryInOneMod(PCSTR pszCalleeModName, PROC pfnCurrent, PROC pfnNew,
HMODULE hmodCaller) {
	ULONG ulSize;
	PIMAGE_IMPORT_DESCRIPTOR pImportDesc = (PIMAGE_IMPORT_DESCRIPTOR)
	ImageDirectoryEntryToData(hmodCaller, TRUE, IMAGE_DIRECTORY_ENTRY_IMPORT,
	&ulSize);
	if (pImportDesc == NULL) return;  // в этом модуле нет раздела импорта
	// находим дескриптор раздела импорта со ссылками на функции DLL (вызываемого
	модуля)
	for (; pImportDesc->Name; pImportDesc++) {
		PSTR pszModName = (PSTR) ((PBYTE) hmodCaller + pImportDesc->Name);
		if (lstrcmpiA(pszModName, pszCalleeModName) == 0) break;
	}
	if (pImportDesc->Name == 0)
	// этот модуль не импортирует никаких функций из данной DLL
	return;
	// получаем таблицу адресов импорта (IAT) для функций DLL
	PIMAGE_THUNK_DATA pThunk = (PIMAGE_THUNK_DATA) ((PBYTE) hmodCaller + pImportDesc-
	>FirstThunk);
	// заменяем адреса исходных функций адресами своих функций
	for (; pThunk->u1.Function; pThunk++) {
		// получаем адрес адреса функции
		PROC* ppfn = (PROC*) &pThunk->u1.Function;
		// та ли это функция, которая нас интересует?
		BOOL fFound = (*ppfn == pfnCurrent);
		if (fFound) { // адреса сходятся; изменяем адрес в разделе импорта
			WriteProcessMemory(GetCurrentProcess(), ppfn, &pfnNew,sizeof(pfnNew),
			NULL);
			return; // получилось; выходим
		}
	}
	// если мы попали сюда, значит, в разделе импорта нет ссылки на нужную функцию
}
```

Чтобы понять, как вызывать эту функцию, представьте, что у нас есть модуль с именем DataBase.exe. Он вызывает ExitProcess из Kernel32.dll, но мы хотим, чтобы он обращался к MyExitProcess в нашем модуле DBExtend.dll. Для этого надо вызвать ReplaceIATEntryInOneMod следующим образом.

```
PROC pfnOrig = GetProcAddress(GetModuleHandle("Kernel32"), "ExitProcess");
HMODULE hmodCaller = GetModuleHandle("DataBase.exe");
```

```
void ReplaceIATEntryInOneMod(
"Kernel32.dll", // модуль, содержащий ANSI-функцию
pfnOrig, // адрес исходной функции в вызываемой DLL
MyExitProcess, // адрес заменяющей функции
hmodCaller); // описатель модуля, из которого надо вызывать новую функцию
```

Первое, что делает ReplaceIATEntryInOneMod, — находит в модуле hmodCaller раздел импорта. Для этого она вызывает ImageDirectoryEntryToData и передает ей IMAGE_DIRECTORY_ENTRY_IMPORT. Если последняя функция возвращает NULL, значит, в модуле DataBase.exe такого раздела нет, и на этом все заканчивается. Если же в DataBase.exe раздел импорта присутствует, то ImageDirectoryEntryToData возвращает его адрес как указатель типа PIMAGE_IMPORT_DESCRIPTOR. Тогда мы должны искать в разделе импорта DLL, содержащую требуемую импортируемую функцию. В данном примере мы ищем идентификаторы, импортируемые из Kernel32.dll (имя которой указывается в первом параметре ReplaceIATEntryInOneMod). В цикле for сканируются имена DLL. Заметьте, что в разделах импорта все строки имеют формат ANSI (Unicode не применяется). Вот почему я вызываю функцию lstrcmpiA, а не макрос lstrcmpi. Если программа не найдет никаких ссылок на идентификаторы в Kernel32.dll, то и в этом случае функция просто вернет управление и ничего делать не станет. А если такие ссылки есть, мы получим адрес массива структур IMAGE_THUNK_DATA, в котором содержится информация об импортируемых идентификаторах. Далее в списке из Kernel32.dll ведется поиск идентификатора с адресом, совпадающим с искомым. В данном случае мы ищем адрес, соответствующий адресу функции ExitProcess. Если такого адреса нет, значит, данный модуль не импортирует нужный идентификатор, и ReplaceIATEntryInOneMod просто возвращает управление. Но если адрес обнаруживается, мы вызываем WriteProcessMemory, чтобы заменить его на адрес подставной функции. Я применяю WriteProcessMemory, а не InterlockedExchangePointer, потому что она изменяет байты, не обращая внимания на тип защиты страницы памяти, в которой эти байты находятся. Так, если страница имеет атрибут защиты PAGE_READONLY, вызов InterlockedExchangePointer приведет к нарушению доступа, а WriteProcessMemory сама модифицирует атрибуты защиты и без проблем выполнит свою задачу. С этого момента любой поток, выполняющий код в модуле DataBase.exe, при обращении к ExitProcess будет вызывать нашу функцию. А из нее мы сможем легко получить адрес исходной функции ExitProcess в Kernel32.dll и при необходимости вызвать ее. Обратите внимание, что ReplaceIATEntryInOneMod подменяет вызовы функций только в одном модуле. Если в его адресном пространстве присутствует другая DLL, использующая ExitProcess, она будет вызывать именно ExitProcess из Kernel32.dll. Если Вы хотите перехватывать обращения к ExitProcess из всех модулей, Вам придется вызывать ReplaceIATEntryInOneMod для каждого модуля в адресном пространстве процесса. Но и в этом случае могут быть проблемы. Например, что получится, если после вызова ReplaceIATEntryInAllMods какой-нибудь поток вызовет LoadLibrary для загрузки новой DLL? Если в только что загруженной DLL имеются вызовы ExitProcess, она будет обращаться не к Вашей функции, а к исходной. Для решения этой проблемы Вы должны перехватывать функции LoadLibraryA, LoadLibraryW, LoadLibraryExA и LoadLibraryExW и вызывать ReplaceIATEntryInOneMod для каждого загружаемого модуля. И, наконец, есть еще одна проблема, связанная с GetProcAddress. Допустим, поток выполняет такой код:

```
typedef int (WINAPI *PFNEXITPROCESS)(UINT uExitCode);
PFNEXITPROCESS pfnExitProcess = (PFNEXITPROCESS)
GetProcAddress( GetModuleHandle("Kernel32"), "ExitProcess");
pfnExitProcess(0);
```

Этот код сообщает системе, что надо получить истинный адрес ExitProcess в Kernel32.dll, а затем сделать вызов по этому адресу. Данный код будет выполнен в обход Вашей подставной функции. Проблема решается перехватом обращений к GetProcAddress. При ее вызове Вы должны возвращать адрес своей функции.

Перехват в точке входа в процедуру с помощью подмены начальных инструкций

Библиотека Detours облегчает перехват вызовов функций. Перехватывающий код применяется динамически во время выполнения. Detours заменяет первые несколько инструкций целевой функции на безусловный переход к предоставленной пользователем функции. Инструкции целевой функции сохраняются в функции-трамплине. Трамплин состоит из инструкций, удаленных из целевой функции и безусловного перехода к остальной части целевой функции. Когда выполнение достигает целевую функцию, управление передается непосредственно к пользовательской функции перехвата. Функция перехвата выполняет всю необходимую предварительную обработку. Она может вернуть управление исходной функции или может вызвать функцию-трамплин, которая вызывает целевую функцию без перехвата. Когда целевая функция завершается, она возвращает управление функции перехвата. Функция перехвата выполняет соответствующие постобработки и возвращает управление исходной функции. На рисунке 1 показан логический поток управления для вызова функции с и без перехвата.

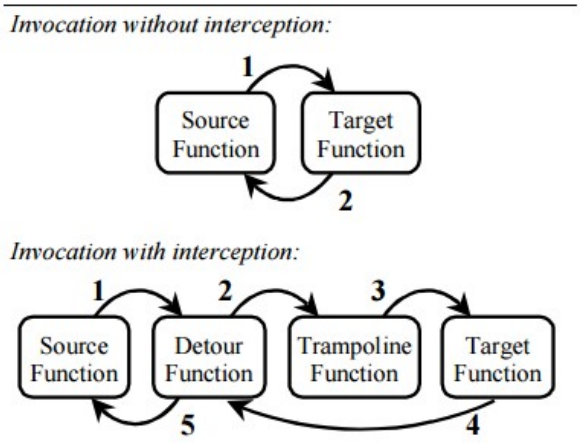


Figure 1. Invocation with and without

Библиотека Detours перехватывает целевые функции, переписав их двоичный образ в памяти. Для каждой целевой функции, Detours фактически переписывает две функции: целевую функцию и соответствующую функцию-трамплин. Функция-трамплин может быть выделена либо динамически, либо статически. Статически выделенный трамплин всегда вызывает целевую функцию без перехвата. До вставки перехвата, статический трамплин содержит один прыжок к целевой функции. После вставки, трамплин содержит начальные инструкции из целевой функции и прыжка к остальной части целевой функции.

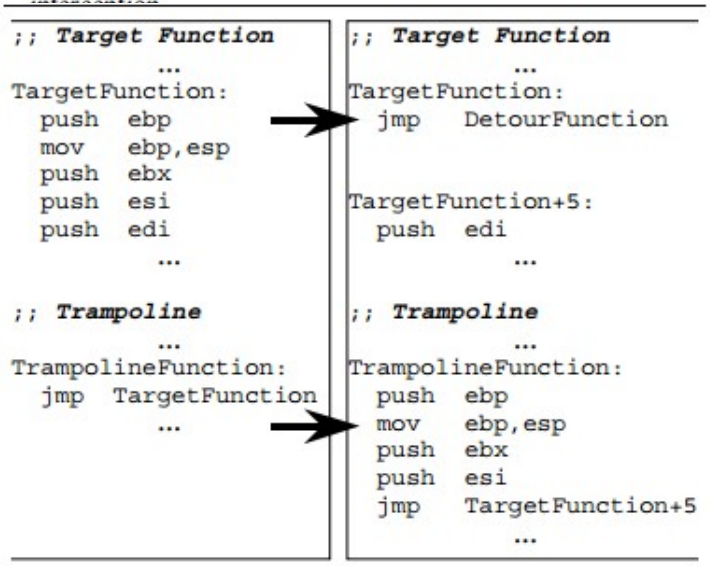


Figure 2. Trampoline and target functions, before and after insertion of the detour (left and right).

Рисунок 2 показывает вставку перехвата. Для перехвата целевой функции, Detours сперва выделяет память для динамической функции-трамплина (если статического трамплина не предусмотрено), а затем открывает доступ на запись в целевой функции и в трамплине. Начиная с первой инструкции, Detours копирует инструкции из целевой функции в трамплин как минимум 5 байт (достаточно для безусловного оператора перехода). Если целевая функция менее 5 байт, Detours прерывается и возвращает код ошибки. Detours добавляет инструкцию перехода в конец трамплина к первой не скопированной инструкции целевой функции. Detours записывает инструкцию безусловного перехода в функцию перехвата первой инструкцией целевой функции. Наконец, Detours восстанавливает исходные разрешения доступа для страниц целевой функции и трамплина и очищает кэш команд CPU с помощью вызова FlushInstructionCache.

54. Перехват API-вызовов ОС Windows в режиме ядра. Таблица системных функций KeServiceDescriptorTable. Таблица системных функций KeServiceDescriptorTableShadow. Понятие UI-потока. Защита от перехвата (Kernel Patch Protection) в 64-разрядной ОС Windows.

KeServiceDescriptorTable
Переменная, указывающая на таблицу API-функций ядра. Экспортируется ядром и видна драйверам. Номер функции может зависеть от версии ОС. По номеру функции можно заменить адрес функции в этой таблице. Таблица защищена от модификации, поэтому перед заменой нужно или отключить бит защиты страницы, или создать доступное для записи отображение таблицы (writable Kernel Virtual Address (KVA) mapping).

KeServiceDescriptorTableShadow
Переменная, указывающая на таблицы API-функций ядра и Win32k.sys. Не экспортируется ядром и не видна драйверам. Это осложняет перехват функций работы с окнами и графикой. UI-потоки содержат указатель на эту таблицу в ETHREAD.Tcb.ServiceTable, но смещение до этого поля отличается в каждой ОС. UI-поток должен обращаться к драйверу за перехватом UI-функций. Перехват UI-функций во время загрузки ОС становится проблематичен.

Защита от перехвата – Kernel Patch Protection
Реализована на 64-разрядной платформе.
Самым популярным местом модифицирования ядра была таблица системных вызовов. При помощи модификации указателей на функции системных вызовов можно было легко их перехватывать, фильтровать, логировать и т. п. Причем этот патч был популярен как для руткитов, так и для антивирусного ПО. Другие интересные для патча объекты — таблицы дескрипторов (GDT, IDT). Через модифицирование глобальной таблицы дескрипторов можно было изменять атрибуты сегментов, создавая бекдоры для кода, а через таблицу дескрипторов прерываний можно было перехватывать… прерывания! Продвинутые же парни сплайсили непосредственно функции ядра.

Соответственно, первая версия PatchGuard защищала:
- таблицы системных вызовов (SST),
- глобальную таблицу дескрипторов (GDT),
- таблицу дескрипторов прерываний (IDT),
- образ ядра,
- ядерные стеки.

С развитием NT перерабатывалось множество компонентов ядра, в том числе и PatchGuard. На текущий момент уже сложно перечислить все, что защищается с его помощью:
- множество системных образов, не только образ ядра (nt, hal, WerLiveKernelApi, tm, clfs, pshed, kdcom, bootvid, ci, msrpc, ndis, ntfs, tcpip, fltmgr),
- критически важные структуры данных ядра (например, список процессов),
- набор MSR (например, model specific регистр IA32_LSTAR),
- KdpStub — процедура отладчика, получающая управление после исключений.

В случае модификации вызывается KeBugCheckEx() с кодом CRITICAL_STRUCTURE_CORRUPTION. При этом стек зачищается, чтобы осложнить реверс-инжиниринг.
Инвестиции в обход механизма Kernel Patch Protection себя не окупают. Microsoft изменяет работу этого механизма в новых обновлениях.
Подробнее про Kernel Patch Protection (PatchGuard) на хабре:

55 Перехват API-вызовов менеджера объектов ОС Windows в режиме ядра.

Callbacks - фу́нкция обра́тного вы́зова — передача исполняемого кода в качестве одного из параметров другого кода. Обратный вызов позволяет в функции исполнять код, который задаётся в аргументах при её вызове. Этот код может быть определён в других контекстах программного кода и быть недоступным для прямого вызова из этой функции. Некоторые алгоритмические задачи в качестве своих входных данных имеют не только числа или объекты, но и действия (алгоритмы), которые естественным образом задаются как обратные вызовы.
Kernel Object Manager (OB)
Управляет пространства имен NT
Общие схемы для управления ресурсами
Расширяемый метод на основе модели для постройки системных объектов
Управление памятью на основе подсчёта ссылок.
Единая модель обеспечения безопасности
Поддерживает хендл-ориентированный -доступ к объектам системы.
Общие механизмы для использования системных ресурсов.
Установка разрешенного набора Callback-процедур для некоторых подсистем ядра:
Object Manager Callbacks
Process Collbacks
Thread Collbacks
Module Load Callbacks
Regusrty Callbacks
File System Mini-Filters

Требования к драйверам
Драйвер должен быть скомпанован с ключом.
Драйвер должен быть подписан сертификатом производителя ПО.
Должен быть включен режим действия тестовых сертификатов
Могут устанавливать коллбэк процедуры на конкурентной основе. Для них применяются уровни перехвата.

Win32k.sys

Callback-CALLBACK_OBJECT

DeleteProcedure=ExpDeleteCallback

WindowsStation, Desktop

CloseProcedure=ExpWin32CloseProcedure
DeleteProcedure=ExpWin32DeleteProcedure
OkayToCloseProcedure=ExpWin32OkayToCloseProcedure
ParseProcedure=ExpWin32ParseProcedure
OpenProcedure=ExpWin32OpenProcedure

56: Перехват API-вызовов создания и уничтожения процессов и потоков ОС Windows в режиме ядра.

Внедрение DLL в адресное пространство процесса — замечательный способ узнать, что происходит в этом процессе. Однако простое внедрение DLL не дает достаточной информации. Зачастую надо точно знать, как потоки определенного процесса вызывают различные функции, а иногда и изменять поведение той или иной Windows функции.

Мне известна одна компания, которая выпустила DLL для своего приложения, работающего с базой данных. Эта DLL должна была расширить возможности основного продукта. При закрытии приложения DLL получала уведомление DLL_PROCESS_DETACH и только после этого проводила очистку ресурсов. При этом DLL должна была вызывать функции из других DLL для закрытия сокетов, файлов и других ресурсов, но к тому моменту другие DLL тоже получали уведомление DLL_PROCESS_DETACH, так что корректно завершить работу никак не удавалось.

Для решения этой проблемы компания наняла меня, и я предложил поставить ловушку на функцию ExitProcess. Как Вам известно, вызов ExitProcess заставляет систему посылать библиотекам уведомление DLL_PROCESS_DETACH. Перехватывая вызов ExitProcess, мы гарантируем своевременное уведомление внедренной DLL о вызове этой функции. Причем уведомление приходит до того, как аналогичные уведомления посылаются другим DLL. В этот момент внедренная DLL узнает о завершении процесса и успевает провести корректную очистку. Далее вызывается функция ExitProcess, что приводит к рассылке уведомлений DLL_PROCESS_DETACH остальным DLL, и они корректно завершаются. Это же уведомление получает и внедренная DLL, но ничего особенного она не делает, так как уже выполнила свою задачу.

В этом примере внедрение DLL происходило как бы само по себе: приложение было рассчитано на загрузку именно этой DLL. Оказываясь в адресном пространстве процесса, DLL должна была просканировать EXE модуль и все загружаемые DLL модули, найти все обращения к ExitProcess и заменить их вызовами функции, находящейся во внедренной DLL. (Эта задача не так сложна, как кажется.) Подставная функция (функция ловушки), закончив свою работу, вызывала настоящую функцию ExitProcess из Kernel32.dll.

Данный пример иллюстрирует типичное применение перехвата API вызовов, который позволил решить насущную проблему при минимуме дополнительного кода.

Перехват API-вызовов подменой кода

Перехват API вызовов далеко не новый метод — разработчики пользуются им уже многие годы. Когда сталкиваешься с проблемой, аналогичной той, о которой я толь ко что рассказал, то первое, что приходит в голову, — установить ловушку, подменив часть исходного кода. Вот как это делается:

1. Найдите адрес функции, вызов которой Вы хотите перехватывать (например, ExitProcess в Kernel32.dll).
2. Сохраните несколько первых байтов этой функции в другом участке памяти.
3. На их место вставьте машинную команду JUMP для перехода по адресу подставной функции. Естественно, сигнатура Вашей функции должна быть такой же, как и исходной, т. е. все параметры, возвращаемое значение и правила вызова должны совпадать.
4. Теперь, когда поток вызовет перехватываемую функцию, команда JUMP перенаправит его к Вашей функции. На этом этапе Вы можете выполнить любой нужный код.
5. Снимите ловушку, восстановив ранее сохраненные (в п. 2) байты.
6. Если теперь вызвать перехватываемую функцию (таковой больше не являющуюся), она будет работать так, как работала до установки ловушки.
7. После того как она вернет управление, Вы можете выполнить операции 2 и 3 и тем самым вновь поставить ловушку на эту функцию.

Этот метод был очень популярен среди программистов, создававших приложения для 16 разрядной Windows, и отлично работал в этой системе. В современных системах у этого метода возникло несколько серьезных недостатков, и я настоятельно не рекомендую его применять. Во первых, он создает зависимость от конкретного процессора из за команды JUMP, и, кроме того, приходится вручную писать машинные коды. Во вторых, в системе с вытесняющей многозадачностью данный метод вообще не годится. На замену кода в начале функции уходит какое то время, а в этот момент перехватываемая функция может понадобиться другому потоку. Результаты могут быть просто катастрофическими!

Перехват API-вызовов с использованием раздела импорта:

Данный способ API перехвата решает обе упомянутые мной проблемы. Он прост и довольно надежен. Но для его понимания нужно иметь представление о том, как осуществляется динамическое связывание. В частности, Вы должны разбираться в структуре раздела импорта модуля. В главе 19 я достаточно подробно объяснил, как создается этот раздел и что в нем находится. Читая последующий материал, Вы всегда можете вернуться к этой главе.
Как Вам уже известно, в разделе импорта содержится список DLL, необходимых модулю для нормальной работы. Кроме того, в нем перечислены все идентификаторы, которые модуль импортирует из каждой DLL. Вызывая импортируемую функцию, поток получает ее адрес фактически из раздела импорта.
Поэтому, чтобы перехватить определенную функцию, надо лишь изменить ее адрес в разделе импорта. Все! И никакой зависимости от процессорной платформы. А поскольку Вы ничего не меняете в коде функции, то и о синхронизации потоков можно не беспокоиться.
Вот функция, которая делает эту сказку былью. Она ищет в разделе импорта модуля ссылку на идентификатор по определенному адресу и, найдя ее, подменяет адрес соответствующего идентификатора.

```
void ReplaceIATEntryInOneMod(PCSTR pszCalleeModName,
PROC pfnCurrent, PROC pfnNew, HMODULE hmodCaller)
{
    ULONG ulSize;
    PIMAGE_IMPORT_DESCRIPTOR pImportDesc = (PIMAGE_IMPORT_DESCRIPTOR)
    ImageDirectoryEntryToData(hmodCaller, TRUE,
    IMAGE_DIRECTORY_ENTRY_IMPORT, &ulSize);
    if (pImportDesc == NULL)
    return; // в этом модуле нет раздела импорта
    // находим дескриптор раздела импорта со ссылками
    // на функции DLL (вызываемого модуля)
    for (; pImportDesc >Name; pImportDesc++)
    {
        PSTR pszModName = (PSTR)
        ((PBYTE) hmodCaller + pImportDesc >Name);
        if (lstrcmpiA(pszModName, pszCalleeModName) == 0)
        break;
    }
    if (pImportDesc >Name == 0)
    // этот модуль не импортирует никаких функций из данной DLL
    return;
    // получаем таблицу адресов импорта (IAT) для функций DLL
    PIMAGE_THUNK_DATA pThunk = (PIMAGE_THUNK_DATA)
    ((PBYTE) hmodCaller + pImportDesc >FirstThunk);
    // заменяем адреса исходных функций адресами своих функций
    for (; pThunk >u1.Function; pThunk++)
    {
        // получаем адрес адреса функции
        PROC* ppfn = (PROC*) &pThunk >u1.Function;
        // та ли это функция, которая нас интересует?
        BOOL fFound = (*ppfn == pfnCurrent);
        // см. текст программы примера, в котором
        // содержится трюковый код для Windows 98
        if (fFound)
        {
            // адреса сходятся; изменяем адрес в разделе импорта
```

```
            WriteProcessMemory(GetCurrentProcess(), ppfn, &pfnNew,
            sizeof(pfnNew), NULL);
            return; // получилось; выходим
        }
    }
    // если мы попали сюда, значит, в разделе импорта
    // нет ссылки на нужную функцию
}
```

Чтобы понять, как вызывать эту функцию, представьте, что у нас есть модуль с именем ataBase.exe. Он вызывает ExitProcess из Kernel32.dll, но мы хотим, чтобы он обращался к MyExitProcess в нашем модуле DBExtend.dll. Для этого надо вызвать ReplaceIATEntryInOneMod следующим образом:

```
PROC pfnOrig = GetProcAddress(GetModuleHandle("Kernel32"), "ExitProcess");
HMODULE hmodCaller = GetModuleHandle("DataBase.exe");

void ReplaceIATEntryInOneMod(
"Kernel32.dll", // модуль, содержащий ANSI-функцию
pfnOrig, // адрес исходной функции в вызываемой DLL
MyExitProcess, // адрес заменяющей функции
hmodCaller); // описатель модуля, из которого надо вызывать новую функцию
```

Первое, что делает ReplaceIATEntryInOneMod, — находит в модуле hmodCaller раз- дел импорта. Для этого она вызывает ImageDirectoryEntryToData и передает ей IMAGE_- DIRECTORY_ENTRY_IMPORT. Если последняя функция возвращает NULL, значит, в модуле DataBase.exe такого раздела нет, и на этом все заканчивается.

Если же в DataBase.exe раздел импорта присутствует, то ImageDirectoryEntryToData возвращает его адрес как указатель типа PIMAGE_IMPORT_DESCRIPTOR. Тогда мы должны искать в разделе импорта DLL, содержащую требуемую импортируемую функцию. В данном примере мы ищем идентификаторы, импортируемые из Kernel32.dll (имя которой указывается в первом параметре ReplaceIATEntryInOneMod). В цикле for ска- нируются имена DLL. Заметьте, что в разделах импорта все строки имеют формат ANSI (Unicode не применяется). Вот почему я вызываю функцию lstrcmpiA, а не макрос lstrcmpi.

Если программа не найдет никаких ссылок на идентификаторы в Kernel32.dll, то и в этом случае функция просто вернет управление и ничего делать не станет. А если такие ссылки есть, мы получим адрес массива структур IMAGE_THUNK_DATA, в кото- ром содержится информация об импортируемых идентификаторах. Далее в списке из Kernel32.dll ведется поиск идентификатора с адресом, совпадающим с искомым. В данном случае мы ищем адрес, соответствующий адресу функции ExitProcess.

Если такого адреса нет, значит, данный модуль не импортирует нужный иденти- фикатор, и ReplaceIATEntryInOneMod просто возвращает управление. Но если адрес обнаруживается, мы вызываем WriteProcessMemory, чтобы заменить его на адрес под- ставной функции. Я применяю WriteProcessMemory, а не InterlockedExchangePointer, потому что она изменяет байты, не обращая внимания на тип защиты страницы па- мяти, в которой эти байты находятся. Так, если страница имеет атрибут защиты PAGE_READONLY, вызов InterlockedExchangePointer приведет к нарушению доступа, а WriteProcessMemory сама модифицирует атрибуты защиты и без проблем выполнит свою задачу.

С этого момента любой поток, выполняющий код в модуле DataBase.exe, при об- ращении к ExitProcess будет вызывать нашу функцию. А из нее мы сможем легко полу- чить адрес исходной функции ExitProcess в Kernel32.dll и при необходимости вызвать ее.

Обратите внимание, что ReplaceIATEntryInOneMod подменяет вызовы функций только в одном модуле. Если в его адресном пространстве присутствует другая DLL, использующая ExitProcess, она будет вызывать именно ExitProcess из Kernel32.dll.

Если Вы хотите перехватывать обращения к ExitProcess из всех модулей, Вам при- дется вызывать ReplaceIATEntryInOneMod для каждого модуля в адресном пространстве процесса. Я, кстати, написал еще одну функцию, ReplaceIATEntryInAllMods. С помощью Toolhelp-функций она перечисляет все модули, загруженные в адресное пространство процесса, и для каждого из них вызывает ReplaceIATEntryInOneMod, передавая в качестве последнего параметра описатель соответствующего модуля.

Но и в этом случае могут быть проблемы. Например, что получится, если после вызова ReplaceIATEntryInAllMods какой-нибудь поток вызовет LoadLibrary для загрузки новой DLL? Если в только что загруженной DLL имеются вызовы ExitProcess, она будет обращаться не к Вашей функции, а к исходной. Для решения этой проблемы Вы должны перехватывать функции LoadLibraryA, LoadLibraryW, LoadLibraryExA и LoadLibraryExW и вызывать ReplaceIATEntryInOneMod для каждого загружаемого модуля.
И, наконец, есть еще одна проблема, связанная с GetProcAddress. Допустим, поток выполняет такой код:

```
typedef int (WINAPI *PFNEXITPROCESS)(UINT uExitCode);
PFNEXITPROCESS pfnExitProcess = (PFNEXITPROCESS) GetProcAddress( GetModuleHandle("Kernel32"),
"ExitProcess");
pfnExitProcess(0);
```

Этот код сообщает системе, что надо получить истинный адрес ExitProcess в Kernel32.dll, а затем сделать вызов по этому адресу. Данный код будет выполнен в обход Вашей подставной функции. Проблема решается перехватом обращений к GetProcAddress. При ее вызове Вы должны возвращать адрес своей функции.

57. Перехват операций с реестром в ОС Windows в режиме ядра.

TODO: Вечером доделаем
Стр 535 // ПОХОДУ нет, но перепроверь

http://www.nobunkum.ru/ru/rootkits-windbg // вроде ссылка на реальный взлом ядра

http://www.e-reading.club/chapter.php/89563/158/Russinovich,_Solomon_-_1.Vnutrennee_ustroiistvo_Windows_(gl._1-4).html /// ещё ссыль, но не в курсе, что там, у меня не загрузилось

Перехват в режиме ядра. Позволяет перехватывать любые функции, в том числе и экспортируемые ядром. Наиболее труден для обнаружения в случае успеха, так как позволяет фальсифицировать любые данные, предоставляемые операционной системой. Требует написания специального компонента для взаимодействия с ядром драйвера. Может привести к BSOD при неправильном программировании в режиме ядра. Может быть обнаружен на фазе загрузки драйвера в ядро или при проверке активных драйверов, а также при проверке ядра на изменения. Более трудный в программировании метод, чем сплайсинг, но более гибкий, так как позволяет перехватить функции самого ядра, а не только WinAPI функции, которые служат лишь посредником между ядром и программой, которая что-либо запрашивает у операционной системы.

Перехват в режиме ядра

Он основан на модификации структур данных ядра и функций. Главными мишенями воздействия являются таблицы

- **IDT** Таблица диспетчеризации прерываний. Довольно важным для перехвата является прерывание, обрабатывающее обращение к таблице служб SSDT (0x2E).
- **SSDT (System Service Dispatch Table)** Таблица диспетчеризации системных сервисов. Обращаясь к ней, система по номеру запрещенного сервиса может получить адрес соответствующего сервиса ядра и вызвать его. А таблица SSPT содержит общий размер параметров, передаваемых системному сервису.
- **psActiveprocess** Структура ядра, хранящая список процессов в системе.
- **IRP** Таблица драйвера, которая хранит указатели на функции обработки IRP-пакетов.
- **EPROCESS** Структура ядра, хранящая большое количество информации о процессе, включая, например, PID (идентификатор процесса).

Такого рода руткиты называются DKOM-руткитами, то есть руткитами, основанными на непосредственной модификации объектов ядра. В руткитах для систем Windows Server 2003 и XP эта технология была модернизирована, так как в этих ОС появилась защита от записи некоторых областей памяти ядра. Windows Vista и 7 получили дополнительную защиту ядра PatchGuard, однако все эти технологии были преодолены руткитописателями. В то же время перехват системных функций в режиме ядра — основа проактивных систем защиты и гипервизоров.

58. Перехват операций с файлами в ОС Windows в режиме ядра. Мини-фильтры файловой системы.

TODO: Вечером доделаем

Стр 562 // возможно, в этих страницах будет что-нибудь хорошее Стр 600+

Внедрение кода в среде Windows 98 через проецируемый в память файл

Эта задача в Windows 98, по сути, тривиальна. В ней все 32-разрядные приложения делят верхние два гигабайта своих адресных пространств. Выделенный там блок памяти доступен любому приложению. С этой целью Вы должны использовать проецируемые в память файлы. Сначала Вы создаете проекцию файла, а потом вызываете MapViewOfFile и делаете ее видимой. Далее Вы записываете нужную информацию в эту область своего адресного пространства (она одинакова во всех адресных пространствах). Чтобы все это работало, Вам, вероятно, придется вручную писать машинные коды, а это затруднит перенос программы на другую процессорную платформу. Но вряд ли это должно Вас волновать — все равно Windows 98 работает только на процессорах типа x86.

Данный метод тоже довольно труден, потому что Вам нужно будет заставить поток другого процесса выполнять код в проекции файла. Для этого понадобятся какие-то средства управления удаленным потоком. Здесь пригодилась бы функция CreateRemoteThread, но Windows 98 ее не поддерживает. Увы, готового решения этой проблемы у меня нет.

Рассмотрим, что из себя в общем виде представляет MiniFilter:

Фильтрация осуществляется через так называемый Filter Manager, который поставляется с операционной системой Windows, активируется только при загрузке мини фильтров. Filter Manager подключается напрямую к стеку файловой системы. Мини фильтры регистрируются на обработку данных по операциям ввода/вывода при помощи функционала Filter Manager, получая, таким образом, косвенный доступ к файловой системе. После регистрации и запуска мини фильтр получает набор данных по операциям ввода/вывода, которые были указаны при конфигурировании, при необходимости может вносить изменения в эти данные, таким образом влияя на работу файловой системы.

На следующей схеме в упрощенном виде показано как функционирует Filter Manager.

Пользовательские запросы ввода/вывода

User Mode

Kernel Mode