Уровни операционной системы Windows представлены на рис. 2.1. Интерфейсы программирования, которые используются для построения приложений, располагаются под уровнями апплетов и графического интерфейса пользователя. Основой данных интерфейсов программирования являются библиотеки кода (DLL), динамически используемые программами для получения доступа к функциональным возможностям операционной системы. В Windows есть также и несколько реализованных в виде служб интерфейсов программирования. Приложения используют удаленные вызовы процедур (RPC) для обмена со службами пользовательского режима.

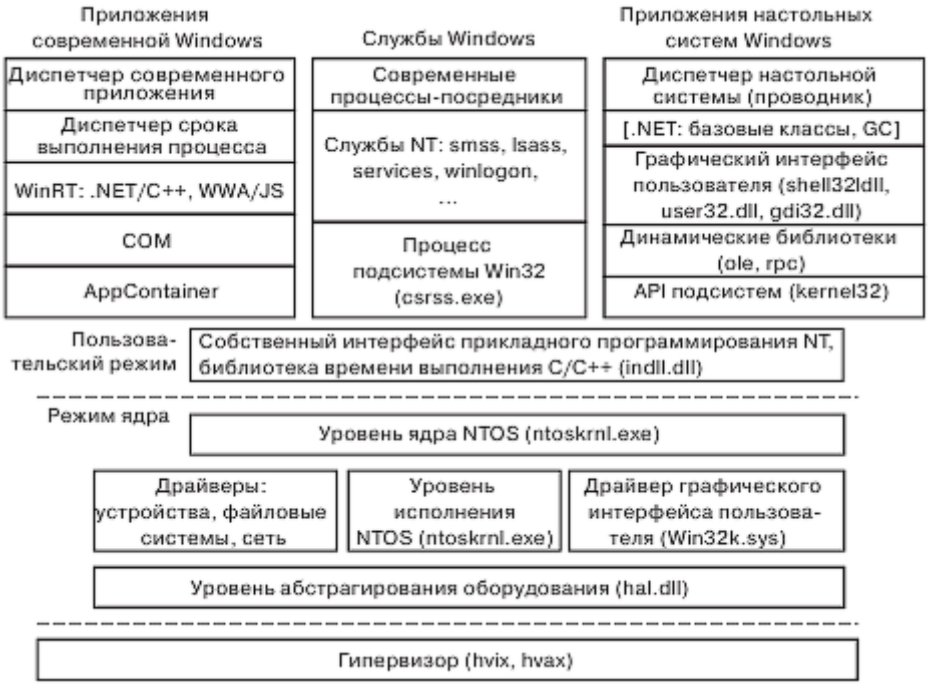

Рис. 2.1. Уровни программирования в Windows

Программа режима ядра NTOS (ntoskrnl.exe) является ядром операционной системы NT и обеспечивает традиционные интерфейсы системных вызовов. Доступные всем разработчикам интерфейсы пользовательского режима принадлежат реализованным при помощи подсистем, которые функционируют поверх уровней NTOS.

Изначально разработчикам были доступны три интерфейса: OS/2, POSIX и Win32. Поддержка OS/2 закончилась с появлением Windows XP. Поддержка POSIX прекратилась с выходом операционной системы Windows 8.1. В настоящий момент большинство Windows-приложений создаются с помощью API-интерфейсов, основанных на подсистеме Win32, причем компания Microsoft осуществляет поддержку и других интерфейсов прикладного программирования.

Начиная с Windows 8, разработчикам предоставляется набор API-интерфейсов WinRT, а также, чтобы обеспечить переход к модели единой операционной системы, одинаково работающей на персональных компьютерах, серверах, смартфонах, планшетных компьютерах и игровых приставках, был создан новый набор современных средств разработки Modern Software Development Kit. API-интерфейсы данного комплекта средств разработки доступны для программ на C++ и C#.

Кроме новых интерфейсов WinRT API в MSDK также включены и многие интерфейсы Win32 API. MSDK продолжает поддерживать модель потоков Win32 для организации параллельных действий внутри процессора. Также сохраняется поддержка большинства API-функций Win32 для

управления виртуальной памятью. Однако из MSDK были убраны не рекомендованные к применению в Win32 API-интерфейсы, а также все API-функции ANSI. Для API-интерфейсов MSDK используется исключительно Unicode.

Подсистемы NT (рис. 2.2) состоят из следующих компонентов: процесса подсистемы, набора библиотек, в том числе библиотеки времени выполнения подсистемы, и поддержки в режиме ядра системы. Процесс подсистемы запускается диспетчером сеансов smss.exe и является службой. Диспетчер сеансов является первой запускаемой программой NT в пользовательском режиме. Запускается она по запросу CreateProcess из Win32 API. Windows поддерживает модель подсистем, основанную на Win32.

Функции операционной системы высокого уровня реализуются набором библиотек. Для обмена между процессами, которые используют подсистему, и самим процессом подсистемы набор библиотек использует заглушки процедур. Средства LPC (Local Procedure Call), работающие в режиме ядра, используются для вызовов процесса подсистемы и процедурных вызовов между процессами.

Интерфейсом прикладного программирования Win32 или, по-другому, Win32 API, называют вызовы функций Win32. Полная документация по данным интерфейса доступна для разработчиков на официальных информационных ресурсах компании Microsoft. Win32 API получил свою реализацию в виде библиотек функций. Функции Win32 API представляют собой системные вызовы, которые либо обертывают системные вызовы NT, либо выполняют собственные задачи в пользовательском режиме. Большая часть функциональности недоступных для сторонних разработчиков собственных интерфейсов прикладного программирования NT все же доступна посредством вызовов соответствующих функций Win32 API. Причем существующие функции Win32 API редко меняются при выходе новых версий Windows, но библиотеки пополняются множеством новых функций.

2.1. Примеры вызовов Win32 API и тех вызовов NT API, для которых они являются оболочками

Вызов Win32	Собственный вызов NT API
CreateProcess	NtCreateProcess
CreateThread	NtCreateThread
SuspendThread	NtSuspendThread
CreateSemaphore	NtCreateSemaphore
ReadFile	NtReadFile
DeleteFile	NtSetInformationFile
CreateFileMapping	NtCreateSection
VirtualAlloc	NtAllocateVirtualMemory
MapViewOfFile	NtMapViewOfSection
DuplicateHandle	NtDuplicateObject
CloseHandle	NtClose

Win32 отличается от Unix-подобных систем подробными интерфейсами, которые принимают множество параметров. Кроме того, для решения конкретной задачи существуют несколько способов ее исполнения, а также может использоваться гибрид функций низкого и высокого уровней. Но главное достоинство Win32 в виде богатого набора интерфейсов приводит, однако, к усложнению как раз из-за дублирования функций, плохого разбиения на уровни, а также смешивания функций высокого и низкого уровней.

Для разработчиков, использующих Win32 API, доступны вызовы для работы с системным реестром, например, вызовы для создания и удаления ключей, поиска значений в ключах и т. д. Основные вызовы представлены в табл. 2.3.

2.3. Основные вызовы Win32 API для работы с реестром

Функция Win32 API	Описание
RegCreateKeyEx	Создать новый ключ реестра
RegDeleteKey	Удалить ключ реестра
RegOpenKeyEx	Открыть ключ, чтобы получить его описатель
RegEnumKeyEx	Перечислить подключи того ключа, описатель которого задан
RegQueryValueEx	Поиск значения в ключе

При завершении работы системы данные реестра сохраняются на диске. Так как целостность этих данных критична для работоспособности системы, система выполняет резервное копирование всех данных на диск (чтобы избежать повреждения при отказе системы). Потеря или повреждение системного реестра может привести к необходимости повторной установки как прикладного, так и системного программного обеспечения.

Системные вызовы (system calls) — это интерфейс между операционной системой и прикладными программами. Они позволяют программам запрашивать услуги у ядра ОС, такие как доступ к файловой системе, управление процессами, сетевое взаимодействие и управление памятью.

Примеры системных вызовов:

- Открытие файла
- Создание процесса
- Управление сигналами
- Чтение или запись в устройства

Каждый системный вызов связан с операцией на уровне ядра, которая обычно требует определенных прав доступа. Программы могут вызывать системные команды через библиотеки стандартного ввода/вывода, либо через вызовы API, специфичные для операционной системы.

Запуск внешних приложений или команд с использованием программного кода позволяет управлять другими программами из одной среды. В Java это можно делать через:

Runtime.getRuntime().exec(): обеспечивает простой доступ к системным командам и запуск процессов.

ProcessBuilder: предоставляет более гибкий и детализированный способ управления процессами, позволяет настраивать параметры, перенаправлять ввод/вывод и управлять потоками.

Перезагрузка и выключение системы — это операции, которые требуют прав администратора, так как они изменяют состояние системы. Эти команды могут быть вызваны через системные команды:

В Windows это команды типа shutdown и restart, которые можно запустить через терминал или программно.

В Linux используются команды reboot и shutdown.

В программировании, вызов перезагрузки или выключения системы часто требует выполнения этих команд с правами суперпользователя.

Любая программа, которая должна запустить внешнее приложение, перезагрузить систему или изменить конфигурацию автозапуска, использует средства для выполнения системных вызовов и запуска процессов. В Java это Runtime.exec() или ProcessBuilder.

```
// Пример запуска внешней команды
String command = "some-command";
Process process = Runtime.getRuntime().exec(command);
```

После запуска команды или процесса, программа может читать вывод или статус работы этого процесса. В Java это делается с помощью потоков ввода-вывода (InputStream и OutputStream).

```
Process process = Runtime.getRuntime().exec(command);
BufferedReader reader = new BufferedReader(new InputStreamReader(process.getInputStream()));
String line;
while ((line = reader.readLine()) != null) {
   System.out.println(line);
}
```

Запуск процессов из программы позволяет выполнять внешние приложения или скрипты напрямую из кода. Методы для этого варьируются в зависимости от используемой платформы, но общие принципы одинаковы.

Пример запуска сторонего приложения из кода:

```
import java.io.IOException;
import java.util.Arrays;

public class ProcessManagement {
   public static void main(String[] args) {
      try {
         // Создаем процесс для запуска внешней программы
         ProcessBuilder pb = new ProcessBuilder("notepad.exe");
         Process process = pb.start();
         // Ожидаем завершения процесса
         int exitCode = process.waitFor();
         System.out.println("Процесс завершён с кодом: " + exitCode);
      } catch (IOException | InterruptedException e) {
         e.printStackTrace();
```

```
    }
  }
}
```

Одной из основных задач программного управления системой является работа с файловой системой, которая включает создание, копирование, перемещение и удаление файлов и каталогов.

Пример работы с файлами:

```
import java.io.File;
import java.io.IOException;

public class FileManagement {
  public static void main(String[] args) {
    File file = new File("example.txt");
    // Создаем новый файл
    try {
      if (file.createNewFile()) {
        System.out.println("Файл создан: " + file.getName());
      } else {
        System.out.println("Файл уже существует.");
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
    // Проверяем существование файла
    if (file.exists()) {
      System.out.println("Файл существует.");
    }
    // Удаляем файл
    if (file.delete()) {
      System.out.println("Файл удалён.");
    } else {
      System.out.println("Не удалось удалить файл.");
    }
```

```
    }
}
```

Чтение системных ресурсов позволяет программе мониторить использование ЦП, памяти и других компонентов, что особенно важно в системах мониторинга и управления ресурсами.

Пример считывания системных ресурсов:

```
import java.lang.management.ManagementFactory;
import com.sun.management.OperatingSystemMXBean;


public class SystemMonitoring {
    public static void main(String[] args) {
        OperatingSystemMXBean osBean =
ManagementFactory.getPlatformMXBean(OperatingSystemMXBean.class);
        // Получение загрузки процессора
        double cpuLoad = osBean.getSystemCpuLoad();
        System.out.println("Загрузка процессора: " + (cpuLoad * 100) + "%");
        // Получение свободной памяти
        long freeMemory = osBean.getFreePhysicalMemorySize();
        System.out.println("Свободная память: " + (freeMemory / 1024 / 1024) + " MB");
    }
}
```

Пример полностью функционального приложения с объединением 2-3 задач:

```
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.lang.management.ManagementFactory;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.util.concurrent.TimeUnit;
import com.sun.management.OperatingSystemMXBean;

public class SystemMonitor {
    private static final String LOG_FILE = "system_monitor_log.txt";
    private static final File FIRST_RUN_FILE = new File("first_run.txt");
    public static void main(String[] args) {
        if (!FIRST_RUN_FILE.exists()) {
            addToAutostart();
            try {
                FIRST_RUN_FILE.createNewFile();
            } catch (IOException e) {
                System.out.println("Не удалось создать файл первого запуска.");
            }
        }
        while (true) {
            try {
                monitorSystem();
                TimeUnit.MINUTES.sleep(5);  // Ожидание 5 минут
            } catch (InterruptedException | IOException e) {
                e.printStackTrace();
            }
        }
    }
```

```
    private static void monitorSystem() throws IOException {
        OperatingSystemMXBean osBean =
ManagementFactory.getPlatformMXBean(OperatingSystemMXBean.class);
        double cpuLoad = osBean.getSystemCpuLoad() * 100;
        long freeMemory = osBean.getFreePhysicalMemorySize() / (1024 * 1024);
        long totalMemory = osBean.getTotalPhysicalMemorySize() / (1024 * 1024);
        long usedMemory = totalMemory - freeMemory;

        String logEntry = String.format("%s - CPU: %.2f%% | Memory: %d MB used / %d MB total",
            LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss")),
            cpuLoad, usedMemory, totalMemory);

        try (FileWriter writer = new FileWriter(LOG_FILE, true)) {
            writer.write(logEntry + "\n");
            System.out.println("Данные сохранены: " + logEntry);
        }
    }

    private static void addToAutostart() {
        String osName = System.getProperty("os.name").toLowerCase();
        String currentPath = new
File(SystemMonitor.class.getProtectionDomain().getCodeSource().getLocation().getPath()).getAbsolut
ePath();

        try {
            if (osName.contains("win")) {
                // Добавление в автозапуск в реестре Windows
                String command = "reg add HKCU\\Software\\Microsoft\\Windows\\CurrentVersion\\Run /v
SystemMonitor /t REG_SZ /d \"" + currentPath + "\" /f";
                Runtime.getRuntime().exec(command);
                System.out.println("Программа добавлена в автозагрузку (Windows)");
            } else if (osName.contains("linux")) {
                // Создание .desktop файла для автозагрузки на Linux
```

```
            String userHome = System.getProperty("user.home");
            String desktopEntry = String.format(
                "[Desktop Entry]\n" +
                "Type=Application\n" +
                "Exec=java -jar %s\n" +
                "Hidden=false\n" +
                "NoDisplay=false\n" +
                "X-GNOME-Autostart-enabled=true\n" +
                "Name[en_US]=System Monitor\n" +
                "Name=System Monitor\n" +
                "Comment=Monitoring system resources",
                currentPath
            );

            File autostartFile = new File(userHome + "/.config/autostart/SystemMonitor.desktop");
            try (FileWriter writer = new FileWriter(autostartFile)) {
                writer.write(desktopEntry);
            }
            System.out.println("Программа добавлена в автозагрузку (Linux)");
        }
    } catch (IOException e) {
        e.printStackTrace();
        System.out.println("Не удалось добавить программу в автозагрузку.");
    }
  }
}
```

Задачи на исполнение:

1) Напишите программу на Java, которая запускает текстовый редактор (например, Notepad на Windows или Gedit на Linux). После запуска программа должна дождаться его завершения и вывести сообщение с кодом завершения процесса. Перезагружает систему и открывает созданный текстовый файл

2) Создайте программу, которая добавляет ваше приложение в автозапуск операционной системы после чего перезагружает компьютер:
    - Для Windows: добавьте запись в реестр для автозапуска.
    - Для Linux: создайте файл .service в /etc/systemd/system/ для автозагрузки программы при старте системы.

3) Напишите программу, которая вызывает перезагрузку и выключение операционной системы предварительно записав сообщение в текстовом файле на рабочем столе. Предусмотрите разные реализации для Windows и Linux.

4) Создайте программу, которая будет выводить текущую загрузку процессора и свободное пространство оперативной памяти каждые 5 секунд так же что-бы она записывалась в автозапуск. Используйте `OperatingSystemMXBean` для Java

5) Создайте программу, которая будет выводить текущую загрузку процессора и свободное пространство оперативной памяти каждые 5 секунд. Используйте `OperatingSystemMXBean` для Java

6) Реализуйте программу, которая запускает несколько системных команд (например, `ping` и `tracert`) одновременно и выводит результат их выполнения а так же записывает результат выполнения в файлы. Используйте `ProcessBuilder` в Java

7) Создайте программу, которая считывает и выводит системные логи (например, логи запуска системы или ошибки) так же добавьте выполнение данных функций через равные промежутки времени (используйте планировщик заданий Виндовс):
    - Для Linux: прочитайте логи из /var/log/syslog или /var/log/messages.
    - Для Windows: используйте команду wevtutil для чтения логов.

8) Напишите программу, которая добавляет правило в брандмауэр для разрешения входящих соединений на порт 8080. Реализуйте разные подходы для Linux (используя `iptables`) и для Windows (используя `netsh`) , а так же реализуйте проверку работы портов и ip адресов командами ping и tracert + ipconfig.