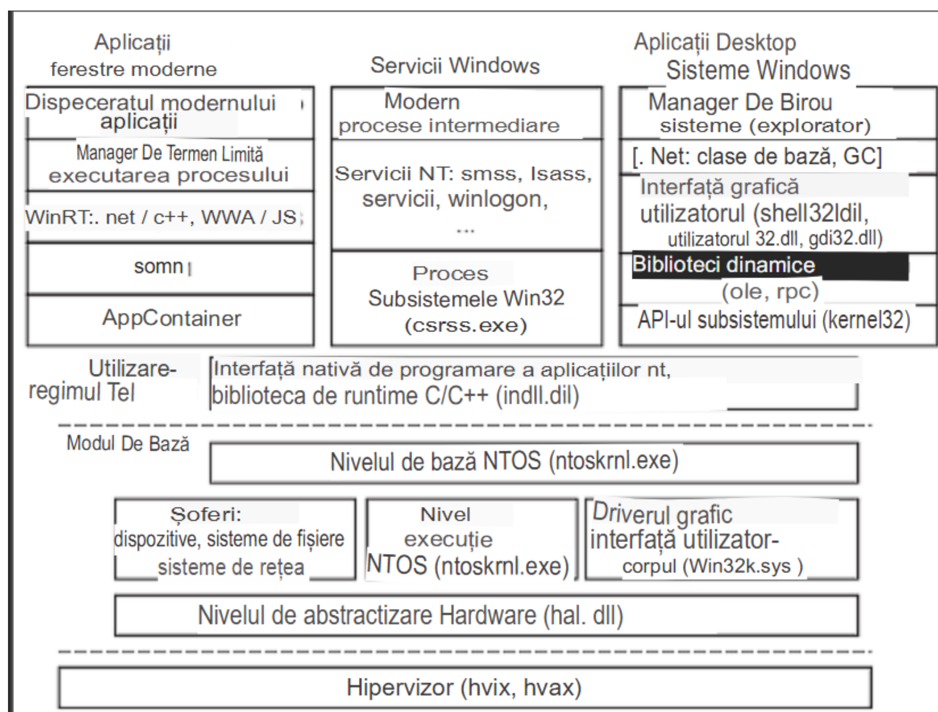
Nivelurile sistemului de operare Windows sunt prezentate în figura 2.1. Interfețele de programare utilizate pentru dezvoltarea aplicațiilor sunt plasate sub nivelurile de applet și interfață grafică pentru utilizator. Baza acestor interfețe de programare o reprezintă bibliotecile de cod (DLL), utilizate dinamic de programe pentru a accesa funcționalitățile sistemului de operare. În Windows există și câteva interfețe de programare implementate sub formă de servicii. Aplicațiile folosesc apeluri de proceduri la distanță (RPC) pentru a comunica cu serviciile din modul utilizator.



Img. 2.1. Nivele de programare in Windows

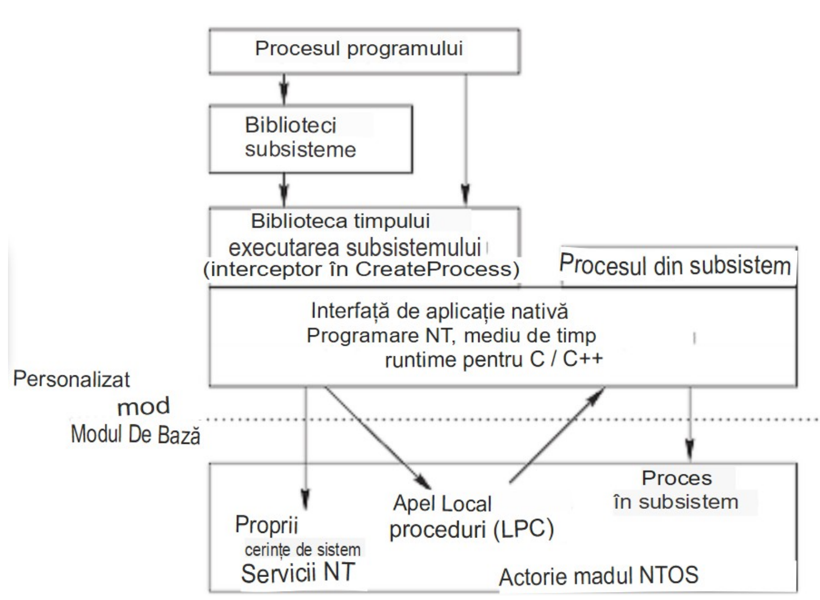
Programul de mod kernel NTOS (ntoskrnl.exe) reprezintă nucleul sistemului de operare NT și asigură interfețele tradiționale pentru apelurile de sistem. Interfețele de mod utilizator, disponibile tuturor dezvoltatorilor, aparțin subsistemelor implementate care funcționează pe baza nivelurilor NTOS.
Inițial, dezvoltatorii aveau acces la trei interfețe: OS/2, POSIX și Win32. Suportul pentru OS/2 s-a încheiat odată cu lansarea Windows XP, iar suportul pentru POSIX a fost întrerupt odată cu apariția sistemului de operare Windows 8.1. În prezent, majoritatea aplicațiilor pentru Windows sunt dezvoltate folosind interfețele API bazate pe subsistemul Win32, iar Microsoft continuă să ofere suport și pentru alte interfețe de programare aplicative.

Începând cu Windows 8, dezvoltatorii au acces la un set de interfețe API WinRT, iar pentru a facilita tranziția către modelul unui sistem de operare unificat, funcționând la fel pe computere personale, servere, telefoane inteligente, tablete și console de jocuri, a fost creat un nou set de instrumente moderne de dezvoltare, Modern Software Development Kit. Interfețele API din acest set de instrumente sunt disponibile pentru programele scrise în C++ și C#.

Pe lângă noile interfețe API WinRT, MSDK include și multe dintre interfețele API Win32. MSDK continuă să susțină modelul de fire Win32 pentru organizarea acțiunilor paralele în cadrul procesorului. Suportul pentru majoritatea funcțiilor API Win32 pentru gestionarea memoriei virtuale este de asemenea păstrat. Totuși, din MSDK au fost eliminate interfețele API

Win32 nerecomandate pentru utilizare, precum și toate funcțiile API ANSI. Pentru interfețele API din MSDK se folosește exclusiv Unicode.

Subsistemele NT (figura 2.2) sunt formate din următoarele componente: procesul subsistemului, un set de biblioteci, inclusiv biblioteca de runtime a subsistemului, și suport în mod kernel pentru sistem. Procesul subsistemului este lansat de managerul de sesiuni smss.exe și funcționează ca un serviciu. Managerul de sesiuni este primul program NT lansat în mod utilizator și este lansat la cererea CreateProcess din API-ul Win32. Windows susține un model de subsisteme bazat pe Win32.



Funcțiile de nivel înalt ale sistemului de operare sunt implementate printr-un set de biblioteci. Pentru schimbul de date între procesele care utilizează subsistemul și procesul subsistemului în sine, setul de biblioteci folosește stub-uri de proceduri. Mijloacele LPC (Local Procedure Call), care funcționează în mod kernel, sunt utilizate pentru apelurile procesului subsistemului și apelurile procedurale între procese.

Interfața de programare aplicativă Win32 sau, altfel spus, Win32 API, se referă la apelurile funcțiilor Win32. Documentația completă a acestor interfețe este disponibilă pentru dezvoltatori pe resursele oficiale de informare ale companiei Microsoft. Win32 API este implementat sub formă de biblioteci de funcții. Funcțiile Win32 API reprezintă apeluri de sistem care fie înfășoară apeluri de sistem NT, fie îndeplinesc sarcini proprii în mod utilizator. O mare parte din funcționalitatea interfețelor de programare aplicative NT, inaccesibile dezvoltatorilor terți, este totuși disponibilă prin apelurile corespunzătoare ale funcțiilor Win32 API. De asemenea, funcțiile existente ale Win32 API se schimbă rar odată cu lansarea unor noi versiuni de Windows, dar bibliotecile sunt îmbogățite cu multe funcții noi.

2.1. Exemple de apeluri API Win32 și acele apeluri API NT pentru care acestea sunt shell-uri

Apel Win32	Apel propriu 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 se diferențiază de sistemele de tip Unix prin interfețele detaliate care acceptă numeroși parametri. În plus, pentru rezolvarea unei sarcini specifice există mai multe metode de implementare, iar uneori poate fi utilizat un hibrid de funcții de nivel scăzut și înalt. Principalul avantaj al Win32, reprezentat de un set bogat de interfețe, duce însă la o complexitate mai mare din cauza duplicării funcțiilor, a diviziunii slabe pe niveluri și a amestecului de funcții de nivel înalt și scăzut.

Pentru dezvoltatorii care utilizează Win32 API, sunt disponibile apeluri pentru lucrul cu registrul de sistem, cum ar fi apeluri pentru crearea și ștergerea de chei, căutarea de valori în chei etc. Principalele apeluri sunt prezentate în tabelul 2.3.

2.3. Apeluri de bază ale API Win32 pentru lucrul cu registrul

Функция Win32 API	Descrierea
RegCreateKeyEx	Creați o nouă cheie de registru
RegDeleteKey	Ștergerea unei chei de registru
RegOpenKeyEx	Deschiderea unei chei pentru a obține descriptorul acesteia
RegEnumKeyEx	Listarea subcheilor cheii al cărei descriptor este dat
RegQueryValueEx	Căutarea unei valori într-o cheie

La închiderea sistemului, datele din registru sunt salvate pe disc. Deoarece integritatea acestor date este esențială pentru funcționarea corectă a sistemului, sistemul realizează o copie de rezervă a tuturor datelor pe disc (pentru a evita deteriorarea în caz de defectare a sistemului). Pierderea sau deteriorarea registrului de sistem poate duce la necesitatea reinstalării atât a software-ului aplicativ, cât și a software-ului de sistem.

Apelurile de sistem (system calls) reprezintă interfața dintre sistemul de operare și programele aplicative. Ele permit programelor să solicite servicii de la nucleul OS, precum accesul la sistemul de fișiere, gestionarea proceselor, interacțiunea în rețea și gestionarea memoriei.

Exemple de apeluri de sistem:

• Deschiderea unui fișier

• Crearea unui proces

• Gestionarea semnalelor

• Citirea sau scrierea pe dispozitive

Fiecare apel de sistem este asociat cu o operațiune la nivel de kernel, care, în mod obișnuit, necesită anumite drepturi de acces. Programele pot invoca comenzi de sistem prin biblioteci de intrare/ieșire standard sau prin apeluri API specifice sistemului de operare.

Lansarea aplicațiilor sau comenzilor externe prin cod permite gestionarea altor programe dintr-un singur mediu. În Java, acest lucru se poate face prin:

- `Runtime.getRuntime().exec()`: oferă acces simplu la comenzi de sistem și la lansarea de procese.
- `ProcessBuilder`: oferă o metodă mai flexibilă și detaliată de gestionare a proceselor, permițând configurarea parametrilor, redirecționarea intrării/ieșirii și gestionarea fluxurilor.

Repornirea și oprirea sistemului sunt operațiuni care necesită drepturi de administrator, deoarece modifică starea sistemului. Aceste comenzi pot fi invocate prin comenzi de sistem: În Windows, acestea sunt comenzi de tip shutdown și restart, care pot fi lansate prin terminal sau programatic.

În Linux sunt utilizate comenzile reboot și shutdown.
În programare, apelarea comenzii de repornire sau de închidere a sistemului necesită, de obicei, execuția acestor comenzi cu drepturi de superutilizator.

Orice program care trebuie să lanseze o aplicație externă, să repornească sistemul sau să modifice configurația de autolansare folosește mijloace pentru efectuarea apelurilor de sistem și pentru lansarea de procese. În Java, acest lucru se realizează prin Runtime.exec() sau ProcessBuilder.

```
// Exemplu de lansare a unei comenzi externe
String command = "some-command";
Process process = Runtime.getRuntime().exec(command);
```

După rularea unei comenzi sau a unui proces, un program poate citi rezultatul sau starea de funcționare a procesului respectiv. În Java, acest lucru se realizează utilizând fluxuri de intrare/ieșire (InputStream și 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);
}
```

Rularea proceselor din cadrul unui program vă permite să executați aplicații sau scripturi externe direct din cod. Metodele de realizare variază în funcție de platforma utilizată, dar principiile generale sunt aceleași.

Un exemplu de rulare a unei aplicații terțe din cod:

```
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();
    }
  }
}
```

Una dintre sarcinile principale ale gestionării sistemului software este lucrul cu sistemul de fișiere, care include crearea, copierea, mutarea și ștergerea fișierelor și directoarelor.

Exemplu de lucru cu fișiere:

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


public class FileManagement {
  public static void main(String[] args) {
    File file = new File("example.txt");
    // Creem fisier nou
    try {
      if (file.createNewFile()) {
        System.out.println("File created: " + file.getName());
      } else {
        System.out.println("File already exists.");
      }
    } catch (IOException e) {
      e.printStackTrace();
    }
    // Verificați dacă fișierul există
    if (file.exists()) {
      System.out.println("File already exists .");
    }
    // Удаляем файл
    if (file.delete()) {
      System.out.println("Fișier șters.");
    } else {
```

```
            System.out.println("Eșec la ștergerea fișierului.");
        }
    }
}
```

Citirea resurselor sistemului permite unui program să monitorizeze utilizarea CPU, a memoriei și a altor componente, ceea ce este deosebit de important în sistemele de monitorizare și gestionare a resurselor.

Exemplu de citire a resurselor sistemului:

```
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);
        // Obținerea sarcinii CPU
        double cpuLoad = osBean.getSystemCpuLoad();
        System.out.println("Cpu load: " + (cpuLoad * 100) + "%");
        // Obținerea de memorie liberă
        long freeMemory = osBean.getFreePhysicalMemorySize();
        System.out.println("Free Memory " + (freeMemory / 1024 / 1024) + " MB");
    }
}
```

Un exemplu de aplicație complet funcțională cu 2-3 sarcini combinate:

```
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("A eșuat crearea primului fișier de pornire.");
      }
    }
    while (true) {
      try {
        monitorSystem();
        TimeUnit.MINUTES.sleep(5);  // Așteptare 5 minute
      } 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("Dta saved: " + 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")) {
                // Adăugarea la autorun în Registrul 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("Programul a fost adăugat la autoloader (Windows)");
            } else if (osName.contains("linux")) {
                // Crearea unui fișier .desktop pentru autoloading pe 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("Program adăugat la autoloader (Linux)");
        }
    } catch (IOException e) {
        e.printStackTrace();
        System.out.println("A eșuat adăugarea programului la autoloader.");
    }
  }
}
```

Sarcini de execuție :

1) Scrieți un program în Java care lansează un editor de text (de exemplu, Notepad pe Windows sau Gedit pe Linux). După lansare, programul trebuie să aștepte finalizarea editorului și să afișeze un mesaj cu codul de terminare al procesului. Apoi, repornește sistemul și deschide fișierul de text creat.

2) Creați un program care adaugă aplicația dvs. în pornirea automată a sistemului de operare și apoi repornește calculatorul:

- Pentru Windows: adăugați o înregistrare în registru pentru pornirea automată.
- Pentru Linux: creați un fișier `.service` în `/etc/systemd/system/` pentru a porni programul automat la startul sistemului.

3) Scrieți un program care repornește și închide sistemul de operare, înregistrând în prealabil un mesaj într-un fișier text pe desktop. Asigurați implementări diferite pentru Windows și Linux.

4) Creați un program care afișează încărcarea curentă a procesorului și spațiul liber din memoria RAM la fiecare 5 secunde, asigurând și adăugarea acestuia în pornirea automată. Folosiți `OperatingSystemMXBean` pentru Java.

5) Creați un program care afișează încărcarea curentă a procesorului și spațiul liber din memoria RAM la fiecare 5 secunde. Folosiți `OperatingSystemMXBean` pentru Java.

6) Realizați un program care lansează mai multe comenzi de sistem (de exemplu, `ping` și `tracert`) simultan și afișează rezultatul execuției acestora, precum și salvează rezultatul în fișiere. Folosiți `ProcessBuilder` în Java.

7) Creați un program care citește și afișează jurnalele de sistem (de exemplu, jurnalele de pornire ale sistemului sau erorile) și adăugați execuția acestor funcții la intervale regulate (folosiți Planificatorul de sarcini în Windows):

- Pentru Linux: citiți jurnalele din `/var/log/syslog` sau `/var/log/messages`.
- Pentru Windows: folosiți comanda `wevtutil` pentru a citi jurnalele.

8) Scrieți un program care adaugă o regulă în firewall pentru a permite conexiuni de intrare pe portul 8080. Realizați abordări diferite pentru Linux (folosind `iptables`) și pentru Windows (folosind `netsh`), și implementați verificarea funcționării porturilor și adreselor IP folosind comenzile `ping`, `tracert` și `ipconfig`.