NAME:KEVIN NYAKWARA KENGERE

REG NO:P101/4394G/24

COURSE TITLE:COM 124

COURSE NAME:OPERATING SYSTEMS

LECTURER NAME:ZABLON OKARI

SIGNATURE _____

System Call Implementation: How It Works and How It's Implemented

What Is a System Call?

A **system call** (syscall) is a mechanism used by user-space programs to request services from the kernel, such as accessing hardware (e.g., disk, network) or performing privileged operations (e.g., process creation, memory management). Since direct hardware access is restricted for security and stability reasons, system calls act as an interface between applications and the operating system.

How System Calls Work

1. **User Program Makes a Request**
   - The program invokes a library function (e.g., `open()`, `read()`, `write()`, etc.).
   - This function translates the request into a system call number.
2. **System Call Invocation**
   - The system call number is placed in a specific CPU register.
   - Additional arguments are passed via registers or the stack.
   - A special instruction (e.g., `syscall` on x86_64, `int 0x80` on older x86) triggers a transition to kernel mode.
3. **Kernel Execution**
   - The kernel looks up the system call number in a **system call table**.
   - It invokes the corresponding kernel function.
   - The function performs the requested operation (e.g., reading from a file, allocating memory).
4. **Returning the Result**
   - The kernel places the return value (or an error code) in a register.
   - It executes a **return-from-trap** instruction to switch back to user mode.
   - The user-space program continues execution with the system call's return value.

How System Calls Are Implemented

1. System Call Table

The kernel maintains a **system call table**, which maps system call numbers to kernel functions. Each system call is assigned a unique number.

Example (Linux `syscall_64.tbl` on x86_64):

```
c
CopyEdit
0   common  read      sys_read
1   common  write     sys_write
2   common  open      sys_open
3   common  close     sys_close
```

---

2. System Call Handler

When a system call is invoked, a **trap handler** (or interrupt handler) executes. It performs:

- Mode switch (User → Kernel)
- System call lookup (via the system call table)
- Kernel function execution
- Returning to user mode

Example (Linux x86_64 entry point in assembly):

```
assembly
CopyEdit
SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t, count)
{
    return vfs_read(fd, buf, count);
}
```

Here, `SYSCALL_DEFINE3` is a macro defining a system call named `read` with three parameters.

---

3. Library Wrapper (C Standard Library - glibc)

User-space programs typically use **glibc** wrappers for system calls.

Example:

```
c
CopyEdit
#include <unistd.h>
```

```
ssize_t bytes_read = read(fd, buffer, size);
```

Here, `read()` in glibc calls the `syscall` instruction internally.

System Call Invocation Across Architectures

Different architectures use different instructions for system calls:

Architecture	Instruction
x86 (32-bit)	`int 0x80`
x86 (64-bit)	`syscall`
ARM	`svc`
RISC-V	`ecall`

Security and Performance Considerations

- **Security**: The kernel verifies system call arguments to prevent exploits (e.g., buffer overflows).
- **Performance**: Switching between user and kernel modes has overhead. Optimizations include:
  - **Fast system calls** (`syscall` vs. `int 0x80`)
  - **VDSo (Virtual Dynamic Shared Object)** for avoiding unnecessary kernel transitions.

Virtual Machines (VMs): How They Work and How They Are Implemented

What Is a Virtual Machine?

A **Virtual Machine (VM)** is a software-based emulation of a physical computer that runs an operating system (OS) and applications just like a physical machine. VMs enable multiple OS instances to run on a single physical hardware system, improving resource utilization and flexibility.

How Virtual Machines Work

VMs function through **virtualization**, a process where software creates an abstraction layer between the physical hardware and the guest operating system. This allows multiple VMs to share the same physical resources while remaining isolated from each other.

Key Components of Virtual Machines

1. **Host Machine**
   - The physical hardware running the virtualization software.
2. **Hypervisor (Virtual Machine Monitor - VMM)**
   - A software layer that manages VMs, allocating CPU, memory, and other resources.
3. **Guest OS**
   - The operating system installed inside a virtual machine.
4. **Virtual Hardware**
   - Emulated CPU, memory, disk, and network interfaces provided by the hypervisor.

How Virtualization Works

There are two primary types of virtualization:

1. Full Virtualization

- The guest OS runs unmodified, as if it were running on real hardware.
- The hypervisor intercepts and translates privileged instructions (e.g., accessing hardware).
- Example: **VMware, VirtualBox, Microsoft Hyper-V**.

Steps in Full Virtualization

1. The guest OS requests access to hardware resources.
2. The hypervisor traps privileged instructions.
3. The hypervisor translates and executes instructions on behalf of the VM.
4. The VM receives results as if it had executed them directly.

2. Paravirtualization

- The guest OS is **modified** to be "aware" of virtualization.
- Instead of trapping privileged instructions, the OS makes **hypercalls** to the hypervisor.
- Example: **Xen, KVM (Kernel-based Virtual Machine)**.

Steps in Paravirtualization

1. The guest OS makes hypercalls instead of direct hardware access.
2. The hypervisor processes these hypercalls.

3. The hypervisor executes requests efficiently.

Types of Hypervisors

Hypervisors manage VMs and can be classified into two types:

Type-1 Hypervisor (Bare Metal)

- Runs directly on hardware without a host OS.
- Provides better performance and efficiency.
- Example: **VMware ESXi, Microsoft Hyper-V, Xen**.

Type-2 Hypervisor (Hosted)

- Runs on top of a host OS.
- Easier to set up but has performance overhead.
- Example: **VirtualBox, VMware Workstation**.

How Virtual Machines Are Implemented

1. CPU Virtualization

- The hypervisor uses **binary translation** or **hardware-assisted virtualization** (Intel VT-x, AMD-V) to execute privileged instructions safely.
- In hardware-assisted virtualization, the CPU provides special instructions to speed up execution.

2. Memory Virtualization

- Each VM has its own **virtual memory**, mapped to physical memory by the hypervisor.
- Techniques like **shadow page tables** and **Extended Page Tables (EPT)** improve performance.

3. I/O Virtualization

- Devices like disks, networks, and GPUs are **emulated** or **passed through** to VMs.
- **Device passthrough** allows VMs to directly access physical hardware.

4. Storage Virtualization

- VMs use **virtual disk images** (e.g., VMDK, VDI) that act as hard drives.
- Some setups use **direct-attached storage (DAS)** or **network-attached storage (NAS)** for better scalability.

5. Network Virtualization

- VMs connect to virtual switches, bridges, or software-defined networks (SDN).
- **NAT, Bridged, and Host-Only** networking allow different network configurations.

Performance Optimization Techniques

1. **Hardware-Assisted Virtualization** (Intel VT-x, AMD-V) for reducing CPU overhead.
2. **Memory Ballooning** dynamically allocates RAM to VMs based on demand.
3. **Storage Deduplication** reduces redundant data storage across VMs.
4. **Paravirtualized Drivers** improve I/O and network performance.
5. **GPU Passthrough** enables high-performance graphics in VMs.

Use Cases of Virtual Machines

1. **Server Consolidation**: Running multiple servers on one physical machine.
2. **Cloud Computing**: AWS, Azure, and Google Cloud use VMs for scalable infrastructure.
3. **Software Testing**: Running different OS versions for development.
4. **Security and Isolation**: Sandboxing applications and malware analysis.
5. **Legacy System Support**: Running old OS and software on modern hardware.

Java Virtual Machine (JVM): How It Works and How It Is Implemented

What Is the Java Virtual Machine (JVM)?

The **Java Virtual Machine (JVM)** is an engine that provides a runtime environment to execute Java applications. It converts Java **bytecode** into machine code that the host system can execute, making Java platform-independent through its "Write Once, Run Anywhere" (WORA) principle.

How the JVM Works

Java programs do not run directly on the operating system. Instead, they execute within the JVM, which abstracts the underlying hardware and OS.

1. Compilation Process

The execution of Java programs involves multiple steps:

1. **Java Source Code (.java file)**
   - A programmer writes Java code in `.java` files.
2. **Compilation to Bytecode (.class file)**
   - The **Java Compiler (javac)** compiles the source code into an intermediate form called **bytecode**.
   - Bytecode is platform-independent and stored in `.class` files.
3. **Execution by JVM**
   - The JVM loads `.class` files, interprets or compiles the bytecode into machine code, and executes it.

---

2. Key Components of the JVM

The JVM consists of several subsystems that manage execution:

1. Class Loader Subsystem

- Responsible for **loading Java classes** into memory.
- Loads classes dynamically when needed.
- **Three class loaders**:
  - **Bootstrap ClassLoader** (loads core Java libraries, e.g., `java.lang.Object`).
  - **Extension ClassLoader** (loads Java extensions, e.g., `javax` packages).
  - **Application ClassLoader** (loads user-defined classes).

2. Runtime Memory Areas

The JVM organizes memory into different regions:

Memory Area	Description
Method Area	Stores class structures, method data, and runtime constant pool.
Heap	Stores Java objects and class instances.
Stack	Stores method call frames and local variables.
PC Register	Stores the address of the current instruction.
Native Method Stack	Stores native (non-Java) method information.

3. Execution Engine

- The **Execution Engine** translates bytecode into machine code.
- It consists of:
    - **Interpreter**: Executes bytecode line by line (slow but simple).
    - **JIT Compiler (Just-In-Time Compiler)**: Converts bytecode into native machine code at runtime for faster execution.
    - **Garbage Collector (GC)**: Manages automatic memory allocation and deallocation.

4. Native Interface (JNI)

- **Java Native Interface (JNI)** allows Java to interact with native (C/C++) libraries.

How the JVM Is Implemented

The JVM is implemented as a software program that runs on various platforms. Different implementations exist, but they all adhere to the **Java Virtual Machine Specification**.

1. Popular JVM Implementations

- **Oracle HotSpot JVM** (default in JDK)
- **OpenJDK JVM** (open-source reference implementation)
- **Eclipse OpenJ9** (optimized for cloud and small devices)
- **GraalVM** (high-performance JVM with polyglot capabilities)

2. Just-In-Time (JIT) Compilation

- The JVM initially **interprets** bytecode.
- Frequently executed methods are **compiled** to native machine code for better performance.
- JIT compilers include:
    - **C1 (Client Compiler)** – Optimized for quick startup.
    - **C2 (Server Compiler)** – Optimized for long-running applications.

3. Garbage Collection (GC)

The JVM automatically manages memory using **garbage collection** to reclaim unused objects.

Types of Garbage Collectors:

- **Serial GC** (for small applications)
- **Parallel GC** (for multi-core processors)
- **G1 (Garbage First) GC** (for large applications)
- **ZGC & Shenandoah GC** (low-latency for real-time applications)

JVM vs. Physical/Virtual Machines

Feature	Java Virtual Machine (JVM)	Virtual Machine (VM, e.g., VMware)
Purpose	Runs Java applications	Emulates entire OS/hardware
Abstraction Level	High-level (Java bytecode)	Low-level (hardware, OS)
Isolation	Limited to Java processes	Full OS isolation
Portability	Platform-independent	Tied to specific OS/architecture