Introduction to System Call:

A system call is a procedure or a programmatic way from which a computer application or a program request a particular service from the kernel of the operating system in which it is executed on. It is a way for program to interact with the operating system. Without system call, a program cannot make contact with the operating system and thus hardware resources.

System call provides the services of the operating system to the user programs via the Application Program Interface (API). It provide an interface between a process and an operating system to allow user-level processes to request services from the operating system.

System calls are the only way or the only entry point for an application into the kernel system. All programs that needs certain resources must use system calls.

A system call can be written in high level languages like C or Pascal in assembly language. If a high level language is used, then OS may directly invoke the system calls, which are predefined functions.

A system call is initiated by the application or program that needs to access the certain resources in the operating system. The kernel then handles the request, perform necessary operations, and send the result back to the process that needs it.

Without system call, each program would need to implement its own methods for accessing hardware system services, leading to inconsistent and error-prone behavior.

2. Why Do We Need System Calls?

1. **Security and Protection**
    - Prevents user programs from doing things that could harm the system or affect other programs.
    - The OS can check whether a request is valid and safe before allowing it.
2. **Abstraction**
    - Programs don't need to know the intricate details of hardware operations (like how to read a disk sector).
    - The OS abstracts away hardware complexity, offering simpler functions (like read(file, buffer, length)).
3. **Resource Management**
    - The kernel manages CPU time, memory allocation, and device access.
    - System calls allow user programs to request these resources in an orderly way.

---

3. General Flow of a System Call

1. **Program Makes a Request**:
    - The program calls a function in a system library (e.g., read(), write(), fork(), etc. in Unix-like systems).
2. **Trap/Interrupt into Kernel**:
    - The CPU switches from user mode (non-privileged) to kernel mode (privileged) through a special instruction or interrupt.
3. **Kernel Processes the Request**:
    - The kernel checks permissions, finds data in memory or on disk, updates system structures, etc.
4. **Return to User Mode**:
    - The kernel returns control back to the program along with any requested data (e.g., bytes read from a file).

4. Common Types of System Calls

Different textbooks and operating systems categorize system calls slightly differently. A common classification is:

1. **Process Control**
    - Creating, managing, and terminating processes.
    - Examples: fork(), exec(), exit(), wait(), etc.
2. **File Management**
    - Creating, opening, reading, writing, and closing files.
    - Examples: open(), read(), write(), close(), unlink().
3. **Device Management**
    - Requesting or releasing devices, reading or writing to devices, handling device properties.
    - Examples: ioctl() (for controlling devices), read()/write() on special device files.
4. **Information Maintenance**
    - Getting or setting system data, time, or process attributes.
    - Examples: getpid(), gettimeofday(), settimeofday(), uname().
5. **Communications**
    - Creating and managing communications channels (pipes, sockets, shared memory).
    - Examples: pipe(), socket(), connect(), bind(), listen(), accept().

Examples in Unix-Like Systems

- **Process Control**: fork(), exec(), wait(), exit().
- **File Management**: open(), close(), read(), write(), lseek().
- **Device Management**: Often just using file operations on device files, plus specialized calls like ioctl().
- **Information Maintenance**: getpid(), getppid(), uname().
- **Communications**: socket(), connect(), bind(), listen(), accept(), send(), recv().

---

5. System Calls in Various Operating Systems

5.1 Linux & Other Unix-Like Systems

- Linux provides a **POSIX**-compliant set of system calls (POSIX is a set of standards).
- Typical calls: fork(), exec(), open(), read(), write(), etc.
- In actual code, you often use C library functions (part of glibc) that wrap these calls. For instance, calling printf() eventually uses write() system call to the file descriptor stdout.

5.2 Windows

- Windows has a set of **Win32 API** functions. Some are higher-level functions, while others directly wrap system calls inside the Windows kernel (ntdll.dll).
- Examples: CreateFile(), ReadFile(), WriteFile(), CreateProcess(), ExitProcess(), etc.
- Windows also uses system calls like NtCreateFile(), NtReadFile(), but these are usually not called directly by most developers (they use the higher-level Win32 APIs).

5.3 macOS

- macOS (Darwin) is based on a Unix-like kernel (BSD + Mach).
- It supports similar POSIX system calls (like fork(), open(), etc.) plus Mach-specific calls (like mach_msg() for Mach ports).

---

6. Examples of System Calls in Action

1. **Creating a New Process (Linux/Unix)**
    - fork() splits the current process into two: parent and child.
    - The child can then call exec() to load a new program into its memory and run it.
    - The parent can use wait() to wait until the child finishes.
2. **Reading a File (Linux/Unix)**
    - open("file.txt", O_RDONLY) → returns a file descriptor if the file exists.
    - read(fd, buffer, size) → reads size bytes from the file into buffer.

    - close(fd) → closes the file descriptor.
3. **Creating a Socket (Networking)**
    - socket(AF_INET, SOCK_STREAM, 0) → creates a TCP socket.
    - connect(sockfd, &address, addrlen) → connect to a remote server.
    - send(sockfd, data, length, 0) → sends data over the socket.
    - recv(sockfd, buffer, length, 0) → receives data from the socket.

---

7. How Are System Calls Invoked at a Low Level?

- On **x86 architecture** (common on PCs), system calls can be triggered with instructions like int 0x80 (older method) or syscall (newer, more efficient method) on Linux.
- On **ARM architecture** (common on smartphones), there is a similar mechanism with svc (Supervisor Call) instructions.
- In **Windows**, system calls are invoked via software interrupts as well, but typically hidden behind ntdll.dll. You rarely see them directly in user code.

The important point: **user mode** → (trigger special instruction) → **kernel mode** → (perform OS action) → (return to user mode).

---

8. Performance & Overheads

- **Context Switch Overhead**:
    - Whenever you jump from user mode to kernel mode and back, there's some overhead.
- **Minimizing System Calls**:
    - For performance-sensitive applications, developers often try to reduce the number of system calls. For example, instead of calling write() for every byte of data, they might buffer data and write once per chunk.

---

9. Security Implications

- **Privilege Levels**:
    - The kernel runs in a high-privilege mode. If user code could run in that mode, any crash or malicious action would take down the system.
- **Validation**:
    - The OS checks parameters (e.g., file descriptors, pointers) to ensure the request is valid. This helps prevent buffer overflows or illegal memory access.

---

10. Summary

- **System calls** are the critical interface between user programs and the OS kernel.
- They handle **process control**, **file and device management**, **memory allocation**, **inter-process communication**, and more.
- Different operating systems provide different sets of calls or APIs, but the concept is the same: user programs must go through the kernel for privileged operations.
- Minimizing and securing system calls is key to both **performance** and **security**.

Key Takeaways

1. **Definition**: System calls are functions that let user-level software request kernel-level services.
2. **Purpose**: They ensure security, resource management, and hardware abstraction.
3. **Types**: Cover process creation, file I/O, device control, IPC, and more.
4. **Mechanics**: Typically involve switching the CPU from *user mode* to *kernel mode*, doing work, and switching back.
5. **Examples**: fork(), read(), write(), open(), socket(), exec(), etc. (Unix-like), CreateFile(), ReadFile(), CreateProcess(), etc. (Windows).

That's the gist of **system calls**—the glue between your programs and the core of the operating system!

Frequently Asked Questions on System Call in Operating Systems – FAQs

How does a system call work?
When a program executes a system call, it transitions from user mode to kernel mode, which is a higher privileged mode. The transition is typically initiated by invoking a specific function or interrupting instruction provided by the programming language or the operating system.

Once in kernel mode, the system call is handled by the operating system. The kernel performs the requested operation on behalf of the program and returns the result. Afterward, control is returned to the user-level program, which continues its execution.

Why do programs need to use system calls?
Programs use system calls to interact with the underlying operating system and perform essential tasks that require privileged access or system-level resources.

What are some examples of system calls?
Examples include opening and closing files, reading and writing data, creating processes, allocating memory, and performing network operations like sending and receiving data.

Why are system calls necessary?
System calls are necessary as they enable applications to request essential services from the operating system, such as file access, memory management, and hardware control, ensuring secure and efficient interaction with computer resources.